Chapter 2

Equations

Chapter 2

2.1 Norms

Length

We have been talking about errors in the calculation of real numbers. On the real line we use the absolute value to define a concept of length and consequently the concepts of neighbourhood, open sets, convergence and continuity. From this we are led to the idea of error measurement, both the absolute error and the relative error.

Now in linear algebra we deal with vectors and matrices. It is convenient to introduce a measure of length for these objects as well. That is, we want to generalise the concept of absolute value to the case of vectors and matrices. The generalised concept is that of a norm.

Vector Length

It is possible to use these properties to characterise the concept of distance from the origin in any vector space V.

Definition 1. *We say that a function $\|\cdot\|$ on a vector space V is a "norm" (our name for distance from the origin) if the following properties hold:*

(N1) $\|\boldsymbol{x}\| \geq 0$ *for all* $\boldsymbol{x} \in V$.

(N2) $\|\boldsymbol{x}\| = 0$ *if and only if* $\boldsymbol{x} = \mathbf{0}$.

(N3) $\|\boldsymbol{x}+\boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ *for all* $\boldsymbol{x}, \boldsymbol{y} \in V$.

(N4) $\|c \cdot \boldsymbol{x}\| = |c| \; \|\boldsymbol{x}\|$ *for all* $c \in \mathbb{R}$ *and* $\boldsymbol{x} \in V$.

Note that we have used bold-face lower case to denote a vector in V.

Norms

Norms are used on vector spaces as a means of measuring the distance between vectors in the space; that is, they define a concept of distance.

This is analogous to the absolute value on $\mathbb{R}$. Suppose that $\boldsymbol{y} \in \mathbb{R}^n$ approximates $\boldsymbol{x} \in \mathbb{R}^n$. For a given vector norm $\|\cdot\|$ we define the absolute error to be

$$\text{error}_a = \|\boldsymbol{x} - \boldsymbol{y}\|$$

and the relative error to be

$$\text{error}_r = \frac{\|\boldsymbol{x} - \boldsymbol{y}\|}{\|\boldsymbol{x}\|}.$$

Note that the value of the error will depend on the norm that is chosen.

2.1.1 Examples of Common Norms

L_1 **Norm**

Definition 2 (L_1 Norm)**.** *The L_1 norm on $\mathbb{R}^n$ is given by*

$$\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|$$

where $\boldsymbol{x} = (x_1, x_2, ..., x_n)^T$. This function is a norm on $\mathbb{R}^n$.

Example 1 (L_1 Norm)**.**

$$\left\|[-1, 2-4]^T\right\|_1 = 1+2+4 = 7.$$

L_∞ Norm

Another important norm is the maximum or infinity norm,

Definition 3 (L_∞ Norm)**.** *The L_∞ norm on $I\!R^n$ is given by*

$$\|\boldsymbol{x}\|_\infty = \max\{|x_i| : i = 1, .., n\}.$$

Example 2 (L_∞ Norm)**.**

$$\left\|[-1, 2-4]^T\right\|_\infty = \max\{1, 2, 4\} = 4.$$

L_2 Norm

Perhaps the norm that you are most familiar with is the usual Euclidean norm.

Definition 4 (Euclidean Norm)**.** *The Euclidean norm on $I\!R^n$ is given by*

$$\|\boldsymbol{x}\|_2 = \left(\sum_{i=1}^{n} x_i^2\right)^{1/2}$$

where $\boldsymbol{x} = (x_1, ..., x_n)^T$.

Note that the Euclidean norm can be defined in terms of the scalar product. Namely,

$$\|\boldsymbol{x}\|_2^2 = \boldsymbol{x}^T\boldsymbol{x} = \boldsymbol{x} \cdot \boldsymbol{x}$$

where $\cdot$ denote scalar product.

Example 3 (Euclidean Norm)**.**

$$\left\|[-1, 2-4]^T\right\|_2 = \sqrt{1^2 + 2^2 + 4^2} \approx 4.5826.$$

L_p Norms

We can define a whole family of norms depending on a parameter $p \geq 1$. These norms are called the Hölder L_p norms or just the p-norms.

Definition 5 (L_p norm)**.** *The L_p norm on $I\!R^n$ is*

$$\|\boldsymbol{x}\|_p = [|x_1|^p + |x_2|^p + ... + |x_n|^p]^{1/p}$$

where $\boldsymbol{x} = (x_1, x_2, ..., x_n)^T \in I\!R^n$, $n \geq 1$.

The notation for the ∞-norm follows from the result that

$$\|\boldsymbol{x}\|_p \to \|\boldsymbol{x}\|_\infty \text{ as } p \to \infty.$$

Hölder's inequality

A classic result concerning p-norms is Hölder's inequality.

Theorem 1 (Hölder's Inequality)**.** *If $\boldsymbol{x}, \boldsymbol{y} \in I\!R^n$ and $p > 1$ then*

$$|\boldsymbol{x}^T\boldsymbol{y}| \leq \|\boldsymbol{x}\|_p\|\boldsymbol{y}\|_q; \qquad \frac{1}{p} + \frac{1}{q} = 1.$$

Example 4 (Hölder's Inequality)**.**

$$[-1, 2-4] \times [2, 0, -1]^T = 2.$$

$$\left\|[-1, 2-4]^T\right\|_2 \approx 4.5826.$$

$$\left\|[2, 0, -1]^T\right\|_2 \approx 2.2361.$$

2.1.2 Matrix Norms

General Definition of a Matrix Norm

We will denote the set of $n \times m$ (n rows, m columns) matrices by $\mathbb{R}^{n\times m}$.

Definition 6 (Mutually Consistent)**.** *Suppose that we have matrix norms* $\|\cdot\|_{qm}$, $\|\cdot\|_{nq}$ *and* $\|\cdot\|_{nm}$ *on* $\mathbb{R}^{q\times m}$, $\mathbb{R}^{n\times q}$, *and* $\mathbb{R}^{n\times m}$ *respectively. These norms are said to be mutually consistent if*

(N5) $\|AB\|_{nm} \leq \|A\|_{nq}\|B\|_{qm}$ *for all* $A \in \mathbb{R}^{n\times q}$ *and* $B \in \mathbb{R}^{q\times m}$

Unless otherwise stated, we will always work with matrix norms which are mutually consistent.

Operator Matrix Norms

The most convenient matrix norms to use are the norms derived from vector norms on the domain and range spaces of the matrix operators. Let us suppose that we have a vector norm $\|\cdot\|_n$ on $\mathbb{R}^n$ and a vector norm $\|\cdot\|_m$ on $\mathbb{R}^m$.

The operator matrix norm associated with the two given vector norms, $\|\cdot\|_{nm}$ on the space $\mathbb{R}^{n\times m}$ is

Definition 7 (Operator Matrix Norms)**.** *For all* $A \in \mathbb{R}^{n\times m}$

$$\|A\|_{nm} = \max\left\{\frac{\|A\boldsymbol{x}\|_n}{\|\boldsymbol{x}\|_m} : x \in \mathbb{R}^m, x \neq 0\right\}.$$

p–norms

Of the matrix operator norms, the p-norms are the most common. The matrix p-norms ($\|\cdot\|_p$) are associated with the vector p-norms via the definition;

Definition 8 (Operator Matrix Norms (P-norms))**.** *For all* $A \in \mathbb{R}^{n\times m}$

$$\|A\|_p = \max\left\{\frac{\|A\boldsymbol{x}\|_p}{\|\boldsymbol{x}\|_p} : x \in \mathbb{R}^m, x \neq 0\right\}.$$

Note that we have an inconsistency in our notation. $\|\cdot\|_p$ denotes either a vector p-norm or a matrix p-norm depending on its argument.

Additional Properties of Operator Matrix Norms

As well as the usual properties of a norm, operator norms have the following additional properties:

Proposition 1. *Suppose* $A \in \mathbb{R}^{n\times m}$ *and* $p \geq 1$.

1. *There exists a vector* $\boldsymbol{x}^* \in \mathbb{R}^m$ *such that* $\|\boldsymbol{x}^*\|_p = 1$ *and* $\|A\|_p = \|A\boldsymbol{x}^*\|_p$.
2. *For all* $\boldsymbol{x} \in \mathbb{R}^m$, $\|A\boldsymbol{x}\|_p \leq \|A\|_p\|\boldsymbol{x}\|_p$.

Calculating 1-norm and ∞-norm

The calculation of the matrix 1-norm and ∞-norm is straight forward.

Proposition 2. *Let the* i, j *entry of a matrix* $A \in \mathbb{R}^{n\times m}$ *be denoted* $a_{i,j}$. *Then*

(1) $\|A\|_\infty = \max_i \sum_{j=1}^{m} |a_{i,j}|$.

(2) $\|A\|_1 = \max_j \sum_{i=1}^{n} |a_{i,j}|$.

The 1-norm of a matrix is the maximum of the column sums of the absolute values of the entries of the matrix. The ∞-norm is the maximum of the row sums.

Calculating 1-norm and ∞-norm

Example 5 (Matrix Norms)**.** *Suppose*

$$A = \begin{bmatrix} 1 & 3 & -2 \\ 1 & 3 & 5 \\ -4 & 6 & 6 \end{bmatrix}.$$

The absolute column sums are 6, 12 and 13. The absolute row sums are 6, 9 and 16. Hence $\|A\|_1 = 13$ *and* $\|A\|_\infty = 16$.

Frobenius Norm

Another useful matrix norm is the Frobenius norm (F-norm). It is defined

$$\|A\|_F = \left[\sum_{i=1}^{n}\sum_{j=1}^{m} |a_{i,j}|^2\right]^{1/2} = \left[\operatorname{trace}(AA^T)\right]^{1/2}.$$

This norm cannot be defined as an operator norm. It is commonly used as an easily computed estimate of $\|A\|_2$. This follows since

$$\frac{\|A\|_F}{\sqrt{n}} \le \|A\|_2 \le \|A\|_F.$$

Example 6 (Frobenius Norm)**.**

$$\left\| \begin{bmatrix} 1 & 3 & -2 \\ 1 & 3 & 5 \\ -4 & 6 & 6 \end{bmatrix} \right\|_F = \left(\text{trace}\left(\begin{bmatrix} 14 & 0 & 2 \\ 0 & 35 & 44 \\ 2 & 44 & 88 \end{bmatrix} \right) \right)^{1/2} \approx 11.7047.$$

Note that the L_2 norm of the above matrix is 10.6252 to 4 decimal places.

Calculating the 2-norm

The matrix 2-norm in general is difficult to calculate. The calculation of the norm is equivalent to the calculation of the maximum singular value of the matrix which is equivalent to an eigenvalue calculation for the matrix A^TA.

Example 7 (2-norm of a Matrix)**.** *Let us calculate the* 2*-norm of the matrix*

$$A = \begin{bmatrix} 0 & 0 \\ 2 & 0 \end{bmatrix}.$$

Now

$$A^TA = \begin{bmatrix} 4 & 0 \\ 0 & 0 \end{bmatrix}$$

which obviously has eigenvalues 4 *and* 0*. Hence the singular values of* A *are* 2 *and* 0 *and so* $\|A\|_2 = 2$.

2.2 Gaussian Elimination

- Consider the linear system

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ &\vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

- It can be written as the form $A\boldsymbol{x} = \boldsymbol{b}$ by introducing

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- We will assume that A is non-singular or invertible, i.e. there is another $n \times n$ matrix B such that $BA = I$, where I is the $n \times n$ identity matrix.

- Linear systems appear in many applications:
  - Regression by least squares method in statistics
  - Linear programming in optimization
  - Numerical solutions of ordinary differential equations
  - Numerical solutions of partial differential equations
  - Solving nonlinear equations by linearization
  - $\cdots$

- The size of $A\boldsymbol{x} = \boldsymbol{b}$ is usually huge in applications; it is impossible to find its solution by hand.

- Need to develop algorithms to let computer do the job.

- The algorithms for solving linear systems fall into two categories:
  - Direct methods;
  - Iterative methods.

- Direct methods are those methods that solve linear equations exactly in a finite number of operations (provided exact arithmetic is used).

- These methods are used when the matrix is dense that is when most of the entries in the matrix are non-zero.

- The most common direct method is Gaussian elimination.

- The basic idea in Gaussian elimination is to multiply A by finite many simple lower triangular matrices to convert A to an upper triangular matrix, leading to the factorization

$$A = LU$$

  with L a lower triangular matrix and U an upper triangular matrix. Such factorization is called LU factorization.

2.2.1 LU Factorisation

- LU factorization can be realized by the elementary row operations.

- There are three types of elementary row operations:
  - swapping two rows

- multiplying a row by a non-zero number
- adding a multiple of one row to another row

Example 8. *Convert $\boldsymbol{a} = [a_1, a_2, a_3, a_4]^T$ with $a_2 \neq 0$ into the form $[a_1, a_2, 0, 0]^T$. This can be achieved by*

- *multiplying the second row by $-\frac{a_3}{a_2}$ and adding to the third row;*
- *multiplying the second row by $-\frac{a_4}{a_2}$ and adding to the fourth row.*

In terms of matrix language, we have

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -\frac{a_3}{a_2} & 1 & 0 \\ 0 & -\frac{a_4}{a_2} & 0 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ 0 \\ 0 \end{bmatrix}$$

Definition 9 (Elementary Matrices (Multipliers))**.** *Given a column vector $\boldsymbol{a} = [a_1, a_2, \cdots, a_n]^T$ with $a_k \neq 0$. We have*

$$E_k\boldsymbol{a} = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -m_{k+1} & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -m_n & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where $m_j = a_j/a_k$. Thus the matrix E_k is designed to nullify all of the elements below a_k in the vector $\boldsymbol{a}$; E_k is called an elementary matrix or multiplier associated with $\boldsymbol{a}$.

Properties

Let $\boldsymbol{e}_k$ denote the column vector with 1 on spot k and 0 elsewhere.

1. E_k is lower triangular with unit main diagonal.
2. $E_k = I - \boldsymbol{m}_k\boldsymbol{e}_k^T$, where $\boldsymbol{m}_k = [0 \;\cdots\; 0 \; m_{k+1} \;\cdots\; m_n]^T$.
3. $E_k^{-1} = I + \boldsymbol{m}_k\boldsymbol{e}_k^T$ (E_k^{-1} will be denoted by L_k).
4. If $k < j$ then $E_kE_j = I - \boldsymbol{m}_k\boldsymbol{e}_k^T - \boldsymbol{m}_j\boldsymbol{e}_j^T$.
5. $E_1E_2 \cdots E_{n-1} = I - \sum_{k=1}^{n-1} \boldsymbol{m}_k\boldsymbol{e}_k^T$ – lower triangular matrix.

The first two items are obvious. For the third item, use $\boldsymbol{e}_k^T\boldsymbol{m}_k = 0$ and

$$\begin{aligned} \left(I - \boldsymbol{m}_k\boldsymbol{e}_k^T\right)\left(I + \boldsymbol{m}_k\boldsymbol{e}_k^T\right) &= I - \left(\boldsymbol{m}_k\boldsymbol{e}_k^T\right)\left(\boldsymbol{m}_k\boldsymbol{e}_k^T\right) \\ &= I - \boldsymbol{m}_k\left(\boldsymbol{e}_k^T\boldsymbol{m}_k\right)\boldsymbol{e}_k^T = I. \end{aligned}$$

The last two items can be checked by using $\boldsymbol{e}_k^T\boldsymbol{m}_j = 0$ for $k < j$.

Example 9 (Elementary Matrices)**.** *Given $a = [2 \quad 4 \; -2]^T$, we have*

$$\begin{aligned} E_1 a &= \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}, \\ E_2 a &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 0 \end{bmatrix}, \\ L_1 = E_1^{-1} &= \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \\ L_2 = E_2^{-1} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{2} & 1 \end{bmatrix}. \end{aligned}$$

LU Factorisation

Given a square matrix $A = (a_{i,j})$. We first perform the Gaussian elimination on the first column to make the elements below $a_{1,1}$ zero, we then perform the Gaussian elimination on the second column to make the elements below the position $(2, 2)$ zero, we continue this procedure until the last column. This leads to

$$E_{n-1} \cdots E_2 E_1 A = U$$

where U is an upper triangular matrix. Hence

$$\begin{aligned} A &= E_1^{-1} E_2^{-1} \cdots E_{n-1}^{-1} U \\ &= L_1 L_2 \cdots L_{n-1} U \\ &= LU. \end{aligned}$$

where $L = L_1 L_2 \cdots L_{n-1}$ is lower triangular.

L can be obtained easily once $E_1, \cdots, E_{n-1}$ are available. Indeed

$$E_k = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\ell_{k+1,k} & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\ell_{n,k} & 0 & \cdots & 1 \end{bmatrix}$$

$$\Longrightarrow L_k = E_k^{-1} = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & \ell_{k+1,k} & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \ell_{n,k} & 0 & \cdots & 1 \end{bmatrix}$$

$$\Longrightarrow L = L_1 L_2 \cdots L_{n-1} = \begin{bmatrix} 1 & & & & \\ \ell_{2,1} & 1 & & & \\ \vdots & \ell_{3,2} & \ddots & & \\ \vdots & \vdots & \ddots & 1 & \\ \ell_{n,1} & \ell_{n,2} & \cdots & \ell_{n,n-1} & 1 \end{bmatrix}$$

Hence L is a lower triangular matrix with unit main diagonal.

Example 10 (LU factorisation)**.** *Consider the matrix*

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

We have

$$E_1 A = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{bmatrix}.$$

$$E_2 E_1 A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} = U.$$

Therefore, we obtain the LU Factorisation $A = LU$, where

$$L = L_1 L_2 = E_1^{-1} E_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix}.$$

Algorithm (LU factorisation)

```
L = I
for k=1:n-1
  for i=k+1:n
    L(i,k) = A(i,k)/A(k,k)
    A(i,k) = 0.0
    for j=k+1:n
       A(i,j)=A(i,j)-L(i,k)*A(k,j)

U = A
```

Existence and Uniqueness

Example 11. *The matrix* $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ *does not have an LU factorisation.*

Question: Under what conditions does a square matrix have LU factorisation?

To answer this question, we need the following definition.

Definition 10. *Let A^r denote the $r \times r$ submatrix in the first r rows and columns of A. The leading principal $r \times r$ minor of A is the determinant of A^r, written* $\det(A^r)$.

Theorem 2 (*LU* factorisation)**.** *If the first $n-1$ leading principal minors of an $n \times n$ matrix A do not vanish, then there is a lower triangular matrix L with unit main diagonal and an upper triangular matrix U such that $A = LU$. This triangular factorisation is unique.*

- Every symmetric positive definite matrix has LU factorisation.

Proof. By induction or by constructing the elimination process and observing that the process may be continued as long as the pivots are non-zero. The pivots will be non-zero since the product of the first j pivots is equal to $\det(A^j) \neq 0$. □

2.2.2 Solving Linear systems

Consider the linear system

$$A\boldsymbol{x} = \boldsymbol{b}$$

with LU factorisation $A = LU$, where

$$L = \begin{bmatrix} \ell_{1,1} & & & \\ \ell_{2,1} & \ell_{2,2} & & \\ \vdots & \vdots & \ddots & \\ \ell_{n,1} & \ell_{n,2} & \cdots & \ell_{n,n} \end{bmatrix}, \quad U = \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ & u_{2,2} & \cdots & u_{2,n} \\ & & \ddots & \vdots \\ & & & u_{n,n} \end{bmatrix}.$$

We have $LU\boldsymbol{x} = \boldsymbol{b}$. Set $\boldsymbol{y} = U\boldsymbol{x}$. Then the solution of $A\boldsymbol{x} = \boldsymbol{b}$ can be found by the following two steps:

1. Solve the lower triangular system $L\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$ by forward substitution.
2. Solve the upper triangular system $U\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$ by back substitution.

Forward Substitution

The lower triangular system $L\boldsymbol{y} = \boldsymbol{b}$ takes the form

$$\begin{array}{ccccccc} \ell_{1,1}y_1 & & & & & = & b_1 \\ \ell_{2,1}y_1 & + & \ell_{2,2}y_2 & & & = & b_2 \\ \vdots & & \vdots & \ddots & & = & \vdots \\ \ell_{n,1}y_1 & + & \ell_{n,2}y_2 & + \ldots + & \ell_{n,n}y_n & = & b_n \end{array}$$

By the first equation we can obtain

$$y_1 = \frac{b_1}{\ell_{1,1}}.$$

Inductively, the solution is given by forward substitution

$$y_i = \frac{b_i - \sum_{k=1}^{i-1} \ell_{i,k} y_k}{\ell_{i,i}}, \quad i = 1, 2, \ldots, n.$$

Backward Substitution

The upper triangular system $U\boldsymbol{x} = \boldsymbol{y}$ takes the form

$$\begin{array}{ccccccc} u_{1,1}x_1 & + & u_{1,2}x_2 & + \ldots + & u_{1,n}x_n & = & y_1 \\ & & u_{2,2}x_2 & + \ldots + & u_{2,n}x_n & = & y_2 \\ & & & \ddots & \vdots & \vdots & \vdots \\ & & & & u_{n,n}x_n & = & y_n \end{array}$$

By the last equation we obtain

$$x_n = \frac{y_n}{u_{n,n}}.$$

Inductively, the solution $\boldsymbol{x}$ is given by back substitution

$$x_i = \frac{y_i - \sum_{k=i+1}^{n} u_{i,k} x_k}{u_{i,i}}, \quad i = n, n-1, ..., 1.$$

- The forward substitution step can be done at the same time as the LU factorisation by considering the augmented matrix $[A \;\; \boldsymbol{b}]$. Indeed

$$E_{n-1}\dots E_1[A \;\; \boldsymbol{b}] = [E_{n-1}\dots E_1 A \;\; E_{n-1}\dots E_1\boldsymbol{b}] = [U \;\; L^{-1}\boldsymbol{b}] = [U \;\; \boldsymbol{y}].$$

Example 12. *Consider the linear system $A\boldsymbol{x} = \boldsymbol{b}$ where*

$$A = \begin{bmatrix} 1 & 3 & 1 \\ 1 & -2 & -1 \\ 2 & 1 & 2 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 10 \\ -6 \\ 10 \end{bmatrix}.$$

We work on the augmented matrix $[A \;\; \boldsymbol{b}]$. Then

$$E_1[A \;\; \boldsymbol{b}] = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 1 & 10 \\ 1 & -2 & -1 & -6 \\ 2 & 1 & 2 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 1 & 10 \\ 0 & -5 & -2 & -16 \\ 0 & -5 & 0 & -10 \end{bmatrix},$$

$$\begin{aligned} E_2E_1[A \;\; \boldsymbol{b}] &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 1 & 10 \\ 0 & -5 & -2 & -16 \\ 0 & -5 & 0 & -10 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 3 & 1 & 10 \\ 0 & -5 & -2 & -16 \\ 0 & 0 & 2 & 6 \end{bmatrix}. \end{aligned}$$

By back substitution we obtain the solution

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

Algorithm (Solve linear system by LU factorisation)

```
M = [A  b]
for k=1:n−1
  for i=k+1:n
     q = M(i,k)/M(k,k)
     for j = k:n+1
        M(i,j) = M(i, j) − q*M(k,j)
     end
  end
end
x(n) = M(n,n+1)/M(n,n)
for i = n−1:−1:1
   z = 0
   for j = i+1:n
      z = z + M(i,j)*x(j)
   end
   x(i) = (M(i,n+1)−z)/M(i,i)
end
```

Flops - Gaussian Elimination

The most expensive part of the algorithm on LU factorisation involves the row operations which can be written as three nested for loops

```
L = I
for k=1:n−1
  for i=k+1:n
    L(i,k) = A(i,k)/A(k,k)
    A(i,k) = 0.0
    for j=k+1:n
       A(i,j)=A(i,j)-L(i,k)*A(k,j)

U = A
```

The number of floating point operations required to carry out this simple implementation of Gaussian elimination can be estimated as follows. (The order of the loops can be interchanged!)

- The inner loop takes $2(n-k)$ arithmetic operations.
- The next loop does this $n-k$ times (we ignore the line $L(i,k) = A(i,k)/A(k,k)$).
- So the inner two loops take $2(n-k)^2$ operations.
- The total arithmetic operations are

$$\sum_{k=1}^{n-1} 2(n-k)^2 = 2\sum_{l=1}^{n-1} l^2 = \frac{2}{3}n^3 + O(n^2).$$

Here we used the identity

$$\sum_{l=1}^{n-1} l^2 = \frac{1}{3}n^3 + O(n^2).$$

Calculating $\sum_{l=1}^{n-1} l^2$

Note that

$$(l+1)^3 - l^3 = 3l^2 + 3l + 1.$$

Therefore

$$\sum_{l=1}^{n-1}\left((l+1)^3 - l^3\right) = 3\sum_{l=1}^{n-1} l^2 + 3\sum_{l=1}^{n-1} l + n - 1.$$

This implies that

$$n^3 - 1 = 3\sum_{l=1}^{n-1} l^2 + 3\cdot\frac{n(n-1)}{2} + n - 1.$$

Hence

$$\sum_{l=1}^{n-1} l^2 = \frac{1}{3}(n^3-1) - \frac{n(n-1)}{2} - \frac{n-1}{3} = \frac{1}{3}n^3 + O(n^2)$$

As an example, suppose that the time for one operation on a particular computer is 15ms. Then the time for elimination and back substitution is given as a function of n in Table 2.1.

n	elimination	substitution
10	0.0050s	0.0008s
100	5.000s	0.075s
1000	5000.0s	7.5s

Table 2.1: Computation time for elimination and back substitution for matrices of size n by n.

From the above table it is seen that solving a linear system of equations with 100 unknowns is a simple task on a modern computer. However a full 1000×1000 matrix is already near the limit of what can be solved at a reasonable cost on a small computer.

Flops - Back or Forward Substitution

- Let us calculate the number of flops required to solve a triangular system using back or forward substitution.
- There are n divisions and approximately $2\sum_{i=1}^{n-1}(n-i) = n(n-1)$ multiplications and additions.
- Hence back or forward substitution requires $O(n^2)$ flops. This is less than the number of operations needed for LU factorisation.

2.2.3 Breakdown and Pivoting

- We have seen matrices which do not have LU factorisation.
- Even if LU factorisation exists, it may incur backward instability as the following example shows.

Example 13. *Consider the linear system $A\boldsymbol{x} = \boldsymbol{b}$, where*

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2+\varepsilon & 5 \\ 4 & 6 & 8 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

We have

$$E_1A = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2+\varepsilon & 5 \\ 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & \varepsilon & 3 \\ 0 & 2 & 4 \end{bmatrix},$$

$$E_2E_1A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{2}{\varepsilon} & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \\ 0 & \varepsilon & 3 \\ 0 & 2 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & \varepsilon & 3 \\ 0 & 0 & 4-\frac{6}{\epsilon} \end{bmatrix}.$$

Therefore $A = LU$ with

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & \frac{2}{\varepsilon} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 1 & 1 \\ 0 & \varepsilon & 3 \\ 0 & 0 & 4-\frac{6}{\epsilon} \end{bmatrix}.$$

If ε is on the order of machine accuracy, the 4 in $4-\frac{6}{\varepsilon}$ in U is insignificant. Thus, instead of L and U, during computation we actually use

$$\tilde{L} = L, \qquad \tilde{U} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & \varepsilon & 3 \\ 0 & 0 & -\frac{6}{\epsilon} \end{bmatrix}.$$

Direct check shows

$$\tilde{L}\tilde{U} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2+\varepsilon & 5 \\ 4 & 6 & 4 \end{bmatrix} \neq A.$$

Thus $\tilde{L}\tilde{U}$ is significantly different from A. Furthermore, if we use L and U we can get the exact solution

$$\boldsymbol{x} = \begin{bmatrix} \frac{4\varepsilon-7}{2\varepsilon-3} \\ \frac{2}{2\varepsilon-3} \\ -\frac{2(\varepsilon-1)}{2\varepsilon-3} \end{bmatrix} \approx \begin{bmatrix} 7/3 \\ -2/3 \\ -2/3 \end{bmatrix}$$

However, if we use $\tilde{L}$ and $\tilde{U}$, we get

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} \frac{11}{3} - \frac{2\varepsilon}{3} \\ -2 \\ \frac{2\varepsilon}{3} - \frac{2}{3} \end{bmatrix} \approx \begin{bmatrix} 11/3 \\ -2 \\ -2/3 \end{bmatrix}$$

Clearly $\tilde{\boldsymbol{x}}$ is significantly different from $\boldsymbol{x}$ even if $\tilde{L}$ and $\tilde{U}$ are close to L and U. Thus the LU factorisation method is not backward stable. □

The above issues on Gaussian elimination can be prevented by the pivoting strategy which permutes the order of rows or columns.

Example 14. *Consider the matrix*

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 5 \\ 4 & 6 & 8 \end{bmatrix}$$

We have

$$E_1 A = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 5 \\ 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 3 \\ 0 & 2 & 4 \end{bmatrix}$$

The standard procedure can not go further which means A does not have LU factorisation.

We may avoid this problem by swapping rows. Look at column 1 of A, the number 4 is the largest (in magnitude). So we interchange row 1 and row 3:

$$P_1 A = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 5 \\ 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 4 & 6 & 8 \\ 2 & 2 & 5 \\ 1 & 1 & 1 \end{bmatrix}$$

where P_1 is a permutation matrix (A permutation matrix is a matrix obtained from the identity matrix by permuting rows or columns). Now we perform the elimination procedure:

$$E_1 P_1 A = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ -1/4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 6 & 8 \\ 2 & 2 & 5 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 6 & 8 \\ 0 & -1 & 1 \\ 0 & -1/2 & -1 \end{bmatrix}$$

Next look at the last two elements in column 2 of $E_1 P_1 A$, the number -1 is the largest in magnitude. So no need to swap rows which amounts to multiplying it by the identity $P_2 = I$. We then perform the next elimination:

$$E_2 P_2 E_1 P_1 A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1/2 & 1 \end{bmatrix} \begin{bmatrix} 4 & 6 & 8 \\ 0 & -1 & 1 \\ 0 & -1/2 & -1 \end{bmatrix} = \begin{bmatrix} 4 & 6 & 8 \\ 0 & -1 & 1 \\ 0 & 0 & -3/2 \end{bmatrix}$$

Set

$$P = P_2 P_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

and

$$E = E_2 P_2 E_1 P_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1/2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ -1/4 & 0 & 1 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ -1/4 & -1/2 & 1 \end{bmatrix}.$$

we obtain $EPA = U$ and hence $PA = LU$, where

$$L = E^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 1/4 & 1/2 & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 4 & 6 & 8 \\ 0 & -1 & 1 \\ 0 & 0 & -3/2 \end{bmatrix}.$$

This shows that, although A may not have LU factorisation, after multiplying by a suitable permutation matrix, it has LU factorisation. $\square$

$$\underbrace{\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & a_{i,k} & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix}}_{\text{Pivot selection}} \xrightarrow{P} \underbrace{\begin{bmatrix} \times & \times & \times & \times & \times \\ & a_{i,k} & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix}}_{\text{Row interchange}} \xrightarrow{E} \underbrace{\begin{bmatrix} \times & \times & \times & \times & \times \\ & a_{i,k} & \times & \times & \times \\ & 0 & \times & \times & \times \\ & 0 & \times & \times & \times \\ & 0 & \times & \times & \times \end{bmatrix}}_{\text{Elimination}}$$

Partial Pivoting

- In general, back instability can be avoided by swapping rows before applying E_k, i.e., we perform
$$E_{n-1}P_{n-1}\cdots E_2P_2E_1P_1A = U.$$
- The strategy we use for swapping rows in step k is to find the largest element (in magnitude) in column k below (and including) the main diagonal – the so-called pivot element – and swap its row with row k.
- This process is referred to as partial (row) pivoting.
- By introducing
$$\tilde{E}_k = P_{n-1}\cdots P_{k+1}E_kP_{k+1}^{-1}\cdots P_{n-1}^{-1}, \quad k = 1, \cdots, n-1,$$
the Gauss elimination with partial pivoting can be written in the form
$$(\tilde{E}_{n-1}\cdots\tilde{E}_1)(P_{n-1}\cdots P_1)A = U.$$

The purpose of the complete pivot strategy is to ensure that U cannot be drastically larger than A. In 1961 Wilkinson produced a clever proof to show that for this strategy

$$g_n \le \sqrt{n}f(n)$$

where

$$f(n) = \left(2^1 3^{1/2} 4^{1/3} \ldots n^{1/(n-1)}\right)^{1/2} \sim n^{\frac{1}{4}log(n)} \text{as } n \to \infty.$$

This function is much smaller than the corresponding bound of 2^{n-1} for the partial pivoting strategy.

In practice it is observed that partial pivoting usually produces a pivot ratio which is $O(1)$. Hence it is only in rare instances that complete pivoting is actually used.

- Since only permutations P_j with $j > k$ is applied to E_k in the formula of $\tilde{E}_k$, we can verify that $\tilde{E}_k$ has the same structure as E_k.

- Let
$$\begin{aligned} L &= (\tilde{E}_{n-1}\dots\tilde{E}_1)^{-1} = \tilde{E}_1^{-1}\dots\tilde{E}_{n-1}^{-1}, \\ P &= P_{n-1}\cdots P_1. \end{aligned}$$
Then L is a lower triangular matrix with unit main diagonal and P is a permutation matrix. Moreover $PA = LU$.

- Due to the pivoting strategy, each element $\ell_{i,j}$ of L satisfies $|\ell_{i,j}| \leq 1$.

Theorem 3. *For any square matrix A, there exists a permutation matrix P such that $PA = LU$ and each element of L satisfies $|\ell_{i,j}| \leq 1$.*

Algorithm (Gaussian elimination with pivoting)

```
U = A, L = I,  P = I
for k=1:n−1
  select i ≥ k to maximize |U(i,k)|
  U(k,k:n) ⟷ U(i,k:n)       (interchange two rows)
  L(k,1:k−1) ⟷ L(i,1:k−1)
  P(k,:) ⟷ P(i,:)
  for j=k+1:n
     L(j,k) = U(j,k)/U(k,k)
     for m = k:n
        U(j,m) = U(j,m) − L(j,k)*U(k,m)
     end
  end
end
```

In partial pivoting, the selection of pivot at step k only incurs $n-k$ operations and hence only $\sum_{k=1}^{n-1}(n-k) = O(n^2)$ operations overall which is significantly lower than $2n^3/3$ required by Gaussian elimination.

Example 15 (Pivoting Example)**.** *Consider the matrix*
$$A = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix}$$

We first interchange the first and second rows:
$$P_1A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 4 & 9 & -3 \\ 2 & 4 & -2 \\ -2 & -3 & 7 \end{bmatrix}$$

We now perform the first elimination step:
$$E_1P_1A = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix}\begin{bmatrix} 4 & 9 & -3 \\ 2 & 4 & -2 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 4 & 9 & -3 \\ 0 & -1/2 & -1/2 \\ 0 & 3/2 & 11/2 \end{bmatrix}$$

Next we interchange the second and third rows:
$$P_2E_1P_1A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} 4 & 9 & -3 \\ 0 & -1/2 & -1/2 \\ 0 & 3/2 & 11/2 \end{bmatrix} = \begin{bmatrix} 4 & 9 & -3 \\ 0 & 3/2 & 11/2 \\ 0 & -1/2 & -1/2 \end{bmatrix}.$$

We then perform the next elimination:

$$\begin{aligned} E_2P_2E_1P_1A &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/3 & 1 \end{bmatrix} \begin{bmatrix} 4 & 9 & -3 \\ 0 & 3/2 & 11/2 \\ 0 & -1/2 & -1/2 \end{bmatrix} \\ &= \begin{bmatrix} 4 & 9 & -3 \\ 0 & 3/2 & 11/2 \\ 0 & 0 & 4/3 \end{bmatrix} \end{aligned}$$

Therefore we obtain $PA = LU$ with

$$U = \begin{bmatrix} 4 & 9 & -3 \\ 0 & 3/2 & 11/2 \\ 0 & 0 & 4/3 \end{bmatrix}.$$

$$P = P_2P_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

and

$$L = (E_2P_2E_1P_2^{-1})^{-1} = (P_2E_1^{-1}P_2^{-1})E_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & -1/3 & 1 \end{bmatrix}$$

Complete Pivoting

- If, at step k of Gaussian elimination, the $(n-k+1)^2$ entries around the right lower corner are considered as possible pivot, and we pick the largest one (in magnitude) among them, use column interchanges as well as row interchanges to bring it to the pivotal position, the corresponding method is called *complete pivoting.*
- For complete pivoting, at step k there are $(n-k+1)^2$ entries to be examined to determine the largest. Thus, the total cost of selecting pivots requires about

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

  operations.
- This adds significant cost to Gaussian elimination.
- Thus, complete pivoting is an expensive strategy and is rarely used .

Solve linear system by pivoting

Consider the linear equations

$$Ax = b.$$

Assume we have the factorisation $PA = LU$. Then

$$LU\boldsymbol{x} = P\boldsymbol{b}$$

Set $\boldsymbol{y} = U\boldsymbol{x}$. Then the solution $\boldsymbol{x}$ can be found by the following two steps:

1. Solve $L\boldsymbol{y} = P\boldsymbol{b}$ by forward substitution;
2. Solve $U\boldsymbol{x} = \boldsymbol{y}$ by back substitution.

The first step can be done at the same time as the factorisation by considering the augmented matrix $[A \;\; \boldsymbol{b}]$. Indeed

$$\begin{aligned} E_{n-1}P_{n-1}\dots E_1P_1[A \quad \boldsymbol{b}] &= [E_{n-1}P_{n-1}\dots E_1P_1A \quad E_{n-1}P_{n-1}\dots E_1P_1\boldsymbol{b}] \\ &= [U \;\; L^{-1}P\boldsymbol{b}] = [U \quad \boldsymbol{y}]. \end{aligned}$$

Example 16. *Consider the linear system $A\boldsymbol{x} = \boldsymbol{b}$, where*

$$A = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix}, \qquad \boldsymbol{b} = \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix}.$$

We have

$$E_1P_1[A \;\; \boldsymbol{b}] = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 9 & -3 & 8 \\ 2 & 4 & -2 & 2 \\ -2 & -3 & 7 & 10 \end{bmatrix}$$

$$= \begin{bmatrix} 4 & 9 & -3 & 8 \\ 0 & -1/2 & -1/2 & -2 \\ 0 & 3/2 & 11/2 & 14 \end{bmatrix},$$

$$\begin{aligned} E_2P_2E_1P_1[A \;\; \boldsymbol{b}] &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/3 & 1 \end{bmatrix} \begin{bmatrix} 4 & 9 & -3 & 8 \\ 0 & 3/2 & 11/2 & 14 \\ 0 & -1/2 & -1/2 & -2 \end{bmatrix} \\ &= \begin{bmatrix} 4 & 9 & -3 & 8 \\ 0 & 3/2 & 11/2 & 14 \\ 0 & 0 & 4/3 & 8/3 \end{bmatrix} \end{aligned}$$

By back substitution we obtain

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix}$$

Algorithm (Solve linear system by pivoting)

```
M = [A  b]
for k=1:n−1
  select i ≥ k to maximize |M(i,k)|
  M(k,k:n) ⟷ M(i,k:n)      (interchange two rows)
  for j=k+1:n
     q = M(j,k)/M(k,k)
     for m = k:n+1
        M(j,m) = M(j,m) − q*M(k,m)
     end
  end
end
x(n) = M(n,n+1)/M(n,n)
for i = n−1:−1:1
   z = 0
   for j = i+1:n
      z = z + M(i,j)*x(j)
   end
   x(i) = (M(i,n+1)−z)/M(i,i)
end
```

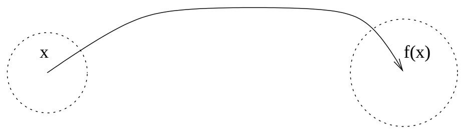


Figure 2.1: Well Conditioned Problem: Small perturbations in the data get mapped to small perturbations in the solution.

2.3 Conditioning and Stability

2.3.1 Abstract Framework

Conditioning and Stability

First some definitions:

Definition 11 (Conditioning)**.** *Conditioning pertains to the perturbation behaviour of mathematical problems.*

Definition 12 (Stability)**.** *Stability (usually) pertains to the perturbation behaviour of algorithms used to solve the problem on a computer.*

In computational science we are always working with an underlying mathematical problem, and trying to develop a reasonable algorithm to approximate the solution of the underlying problem.

As we are always working with approximations, we must have a good idea of the properties of our mathematical problem and our algorithm with regard to perturbations in our input conditions. Of course at the end of the day we want to know how accurate our method is. A very good reference for this section is the book "Numerical Linear Algebra" by Nick Trefethen and David Bau.

In this section we will provide a technique to measure this perturbation behaviour and calculate the behaviour of a few very common problems.

Abstract Framework

Essentially every mathematical problem can be viewed as a process that maps some data to a solution. That is a mathematical problem can be represented as a function from a set of possible input data, to a set of possible solutions.

To be specific we consider the data to be chosen from a vector space X and the solution to exist in a vector space Y.

The abstract problem is represented as a function $f : X \to Y$.

The function f can be very general, but usually we can assume that f is continuous.

We are interested in the behaviour of the problem f at a particular data point x.

Well Conditioned

Definition 13 (Well Conditioned)**.** *A well conditioned problem at x is one with the property: All small perturbations of x lead to only small perturbations of $f(x)$.*

Definition 14 (Ill Conditioned)**.** *An ill conditioned problem at a point x is one with the property: Some small perturbations of x lead to large perturbations of $f(x)$.*

Absolute Condition Number

But what does small and large mean?

Consider a perturbed data point $x + \delta x$. The perturbation is given by δx. The corresponding change in the solution is given by

$$\delta f = f(x + \delta x) - f(x).$$

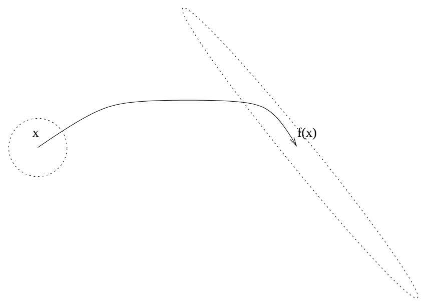


Figure 2.2: Ill Conditioned Problem: Some small perturbations in the data can get mapped to large perturbations in the solution.

Definition 15 (Absolute Condition Number)**.** *The absolute condition number for the problem f at the point x is given by*

$$\hat{\kappa} = \max_{\delta x} \frac{\|\delta f\|}{\|\delta x\|}$$

where the maximum is really taken over infinitesimally small δx.

If f is differentiable then

$$\hat{\kappa} = \|J(x)\|$$

where J is the Jacobian matrix, since $J(x) \approx \|\delta f\|/\|\delta x\|$.

Recall $f(x + \delta x) = f(x) + J(x)\delta x + \mathcal{O}(\delta x^2)$.

Relative Condition Number

But we are always working with relative errors and so we should also work with a relative condition number. The relative condition number is simply the relative change in the solution over the relative change in the data.

Definition 16. *The relative condition number for the problem f at the point x is given by*

$$\kappa = \max_{\delta x} \frac{\|\delta f\|}{\|f(x)\|} \bigg/ \frac{\|\delta x\|}{\|x\|}$$

where the maximum is really taken over infinitesimally small δx.

If f is differentiable then

$$\kappa = \frac{\|J(x)\|}{\|f(x)\|/\|x\|}$$

where J is the Jacobian matrix.

Example of a Well-conditioned Problem

Let's consider the problem of multiplying a number by 3. We hope that this is a well conditioned problem. But lets calculate the associated condition number.

Example 17 ($f(x) = 3x$)**.** *So $f(x) = 3x$. This function is differentiable, so we can calculate $J(x) = f'(x) = 3$. Hence the condition number is given by*

$$\kappa = \frac{\|J(x)\|}{\|f(x)\|/\|x\|} = \frac{3}{3x/x} = 1.$$

This problem is well conditioned.

Consider the problem of calculating the condition number of the square root $\sqrt{x}$.

The derivative of the square root function becomes infinite as x gets close to zero. So the absolute condition number can be arbitrarily large.

But what about the relative condition number?

Example 18 ($f(x) = \sqrt{x}$)**.** *Now $f(x) = \sqrt{x}$ and so $J(x) = f'(x) = 1/(2\sqrt{x})$. Hence the (relative) condition number is given by*

$$\kappa = \frac{\|J(x)\|}{\|f(x)\|/\|x\|} = \frac{1}{2\sqrt{x}}\frac{1}{\sqrt{x}/x} = \frac{1}{2}.$$

So once again this problem is also well conditioned.

Example of an Ill-conditioned Problem

Consider the problem of subtracting two real numbers. So $f(x_1, x_2) = x_1 - x_2$ and $J(\boldsymbol{x}) = \left[\frac{\partial f}{\partial x_1}\ \frac{\partial f}{\partial x_2}\right] = [1\ \ -1]$.

Now the Jacobian is a vector, and we need to decide which norm to use to measure the size of it. We will calculate the condition number using the L_∞ norm (because it is easy to calculate).

Note that the L_∞ matrix norm is the maximum absolute row sum, so $\|J(\boldsymbol{x})\|_\infty = |1| + |-1| = 2$. The L_∞ vector norm is simply the maximum absolute component of a vector. So $\|\boldsymbol{x}\|_\infty = \max\{|x_1|, |x_2|\}$.

Example 19 ($f(x_1, x_2) = x_1 - x_2$)**.** *Hence the condition number is given by*

$$\kappa = \frac{\|J(\boldsymbol{x})\|_\infty}{\|f(\boldsymbol{x})\|_\infty/\|\boldsymbol{x}\|_\infty} = \frac{2}{|x_1 - x_2|/\max\{|x_1|, |x_2|\}} = \frac{2\max\{|x_1|, |x_2|\}}{|x_1 - x_2|}.$$

This problem is ill conditioned if $|x_1 - x_2|$ is small, i.e. when $x_1 \simeq x_2$. This is just the case in which catastrophic cancellation occurs.

2.3.2 Finding the Roots of a Polynomial

Example: Roots of a Polynomial

Consider a polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1}x^{n-1} + x^n$ with n roots $x_1, x_2, ..., x_n$.

The problem is to check out the conditioning of the j-th root x_j to perturbations δa_i of the i-th coefficient a_i.

It is possible to show that perturbations of the j-th root δx_j satisfy

$$\delta x_j = -\frac{\sum_{i=0}^{n-1} \delta a_i\, x_j^i}{p'(x_j)}.$$

Example 20 (Roots of a Polynomial)**.** *From this we conclude that the condition number for this problem is*

$$\kappa = \max_{\delta a_i} \frac{|\delta x_j|}{|x_j|} \Bigg/ \frac{|\delta a_i|}{|a_i|} = \frac{|a_i x_j^{i-1}|}{|p'(x_j)|}.$$

This condition number can often be very large.

How do we obtain the equation for δx_j in terms of δa_i? The polynomial $p(x)$ can be written in two forms

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1}x^{n-1} + x^n \\ &= (x - x_1)(x - x_2)\cdots(x - x_n). \end{aligned}$$

Now let's consider the polynomial $\tilde{p}(x)$ in which the jth root has been perturbed, i.e.

$$\begin{aligned}\tilde{p}(x) &= (x-x_1)(x-x_2)\cdots(x-x_j-\delta x_j)\cdots(x-x_n) \\ &= (x-x_1)(x-x_2)\cdots(x-x_n) \\ &\quad -\delta x_j(x-x_1)\cdots(x-x_{j-1})(x-x_{j+1})\cdots(x-x_n).\end{aligned} \tag{2.1}$$

In terms of perturbation of coefficients

$$\tilde{p}(x) = (a_0+\delta a_0)+(a_1+\delta a_1)x+\cdots+(a_{n-1}+\delta a_{n-1})x^{n-1}+x^n. \tag{2.2}$$

We can simplify the formulas by evaluating the polynomial at one of the roots x_j. Then from Equation (2.1)

$$\tilde{p}(x_j) = -\delta x_j(x_j-x_1)\cdots(x_j-x_{j-1})(x_j-x_{j+1})\cdots(x_j-x_n).$$

Note that the first term drops out as $p(x_j)=0$.

Now the remaining term can be simplified because

$$p'(x_j) = (x_j-x_1)\cdots(x_j-x_{j-1})(x_j-x_{j+1})\cdots(x_j-x_n)$$

(just differentiate $p(x)$ with $p(x)$ in the product of roots form) and so

$$\tilde{p}(x_j) = -\delta x_j p'(x_j).$$

From Equation (2.2)

$$\tilde{p}(x_j) = \delta a_0+\delta a_1 x_j+\cdots+\delta a_{n-1}x_j^{n-1}$$

Together

$$-\delta x_j p'(x_j) = \delta a_0+\delta a_1 x_j+\cdots+\delta a_{n-1}x_j^{n-1}$$

which leads to the required result.

Wilkinson's Example

A classic example was provided by the father of numerical analysis, the Englishman Jim Wilkinson (A colleague of Alan Turing).

As a test of one of his root solving algorithms, Wilkinson considered the problem of finding the roots of the polynomial

$$p(x) = (x-1)(x-2)(x-3)\cdots(x-20).$$

He calculated the coefficients of the polynomial and then tried to recover the known roots 1 to 20. He found that his code would not reproduce the roots to any sensible accuracy.

Consider the condition of the calculation of the 15-th root with perturbations in the 15-th coefficient. Now, $a_{15}\simeq 1.67\times 10^9$ and so

$$\kappa \simeq \frac{1.67\times 10^9\times 15^{14}}{5!14!} \simeq 5.1\times 10^{13}.$$

Working in single precision (8 significant figures), we would not expect to any correct digits in our root calculation.

This caused Wilkinson much worry as he originally thought his program had a bug.

2.3.3 Condition Number of a Matrix

Condition Number of Matrix-Vector Multiplication

The problem we want to analyse is given a matrix A and a vector $\boldsymbol{x}$, what is the value of $\boldsymbol{y} = A\boldsymbol{x}$ and how is it perturbed when $\boldsymbol{x}$ is perturbed.

So the $\boldsymbol{x}$ is the data and $\boldsymbol{y} = A\boldsymbol{x}$ is the solution. (Note that we are thinking of A as being exact and given)

Hence $\boldsymbol{x} \in X$, the data space, and $\boldsymbol{y} = A\boldsymbol{x} \in Y$, the solution space.

Determine the perturbation in the solution in terms of perturbation of the data.

$$\delta\boldsymbol{y} = A(\boldsymbol{x} + \delta\boldsymbol{x}) - A\boldsymbol{x} = A\delta\boldsymbol{x}.$$

Example 21 (Condition Number of Matrix-Vector Multiplication)**.** *Then the condition number for this case is:*

$$\begin{aligned}\kappa &= \max_{\delta\boldsymbol{x}} \frac{\|\delta\boldsymbol{y}\|}{\|\boldsymbol{y}\|} \Big/ \frac{\|\delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \\ &= \max_{\delta\boldsymbol{x}} \frac{\|A\delta\boldsymbol{x}\|}{\|A\boldsymbol{x}\|} \Big/ \frac{\|\delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|}\ .\end{aligned}$$

Condition Number of a Matrix

Looking at the previous example, and noting that $\|A\delta\boldsymbol{x}\| \leq \|A\|\|\delta\boldsymbol{x}\|$ gives

$$\kappa \leq \|A\| \frac{\|\boldsymbol{x}\|}{\|A\boldsymbol{x}\|}.$$

If A is invertible we have $\|\boldsymbol{x}\|/\|A\boldsymbol{x}\| \leq \|A^{-1}\|$ and so

$$\kappa \leq \|A\|\|A^{-1}\|.$$

Definition 17 (Condition Number of a Matrix)**.** *The condition number of a matrix is defined as*

$$\kappa(A) = \|A\|\|A^{-1}\|.$$

This condition number reappears in most conditioning results associated with matrices.

When we obtain a numerical solution $\boldsymbol{y}$ of an equation

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

it comes as no surprise that the solution, $\boldsymbol{y}$ will exhibit a small error or residual

$$\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{y}.$$

Hence $\boldsymbol{r}$ measures the absolute error in the calculated value of $\boldsymbol{b}$. It would be reasonable to assume that if the norm of $\boldsymbol{r}$ was negligible, then the norm of the $\boldsymbol{x} - \boldsymbol{y}$ would be negligible. The following example shows that this assumption is not always correct. Suppose we have the system

$$A = \begin{bmatrix} 0.2161 & 0.1441 \\ 1.2969 & 0.8648 \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} 0.1440 \\ 0.8642 \end{bmatrix},$$

and we obtain an approximate solution

$$\boldsymbol{y} = \begin{bmatrix} 0.9911 \\ -.4870 \end{bmatrix}.$$

Then the residual is

$$\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{y} = \begin{bmatrix} -.00000001 \\ 0.00000001 \end{bmatrix},$$

which appears to be small, so it is tempting to conclude that $\boldsymbol{y}$ must be close to the true solution. But $\boldsymbol{y}$ does not contain a single correct digit! The correct solution is

$$\boldsymbol{x} = \left[\begin{array}{c} 2 \\ -2 \end{array} \right].$$

What is needed is a way to test a problem or matrix to see if this type of behaviour will occur.

So lets see if condition number can give us an idea of what is going on. Consider solving $A\boldsymbol{x} = \boldsymbol{b}$, thinking of A being the data and $\boldsymbol{x}$ as the solution.

The problem is characterised as $f(A) = \boldsymbol{x}$ where $A\boldsymbol{x} = \boldsymbol{b}$. Note that $f(A+\delta A) = \boldsymbol{x} + \delta\boldsymbol{x}$. The perturbation $\delta\boldsymbol{x}$ is related to δA via the equation $(A + \delta A)(\boldsymbol{x} + \delta\boldsymbol{x}) = \boldsymbol{b}$.

Then

$$A\boldsymbol{x} + \delta A\boldsymbol{x} + A\delta\boldsymbol{x} + \delta A\delta\boldsymbol{x} = \boldsymbol{b}$$

and so to first order

$$\delta A\boldsymbol{x} + A\delta\boldsymbol{x} = 0.$$

Hence

$$\delta\boldsymbol{x} = -A^{-1}\delta A\boldsymbol{x}.$$

The condition number of this problem is

$$\begin{aligned} \kappa &= \frac{\|f(A+\delta A) - f(A)\|}{\|f(A)\|} \Big/ \frac{\|\delta A\|}{\|A\|} \\ &= \frac{\|\boldsymbol{x} + \delta\boldsymbol{x} - \boldsymbol{x}\|}{\|\boldsymbol{x}\|} \Big/ \frac{\|\delta A\|}{\|A\|} \\ &= \frac{\|\delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \Big/ \frac{\|\delta A\|}{\|A\|}. \end{aligned}$$

Using $\delta\boldsymbol{x} = -A^{-1}\delta A\boldsymbol{x}$ we have

$$\begin{aligned} \kappa &\leq \frac{\|A^{-1}\|\|\delta A\|\|\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \frac{\|A\|}{\|\delta A\|} \\ &= \|A^{-1}\|\|A\| \\ &= \kappa(A). \end{aligned}$$

The matrix A of the numerical example above is very ill-conditioned.

$$A^{-1} = \left[\begin{array}{cc} -8648000. & 14410000. \\ 129690000. & -21610000. \end{array} \right]$$

and

$$\kappa(A) \approx 2 \times 10^8.$$

Observing that $\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{x}$ we see that

$$\frac{\|\boldsymbol{y} - \boldsymbol{x}\|/\|x\|}{\|\boldsymbol{r}\|/\|\boldsymbol{b}\|} \approx \frac{1/2}{10^{-8}} < \kappa(A).$$

Had $\kappa(A)$ been known in advance, the example would not have come as a surprise.

2.3.4 Stability and Accuracy

Accuracy

Let us try to setup an abstract framework for the algorithm. Now recall that the *Mathematical problem* is characterised by a function

$$f : X \to Y.$$

Similarly the *Algorithm* can be viewed as another map

$$\tilde{f} : X \to Y.$$

Note that $\tilde{f}$ will be affected by lots of subtle effects such as rounding errors, convergence tolerances and maybe even load on computer!

A good algorithm $\tilde{f}$ should approximate the associated problem f. A quantitative measure of this is

- Absolute error $||f(x) - \tilde{f}(x)||$
- Relative error $\dfrac{||f(x) - \tilde{f}(x)||}{||f(x)||}$

If $\tilde{f}$ is a good algorithm we would expect

$$\frac{||f(x) - \tilde{f}(x)||}{||f(x)||} = O(\varepsilon),$$

where ε is the machine epsilon.

It is important to have an idea of the possible conditioning of the underlying problem, but it is also very important to have a similar concept for the associated algorithms.

If f is ill conditioned the goal of accuracy is unreasonably ambitious. For instance, just rounding the data into the floating point system would lead to significant change in the result. So how do we recognise a good algorithm for an ill conditioned problem. You might ask, why even spend time on developing a “good” algorithm for problems for which we cannot expect accurate solutions. Well often we don’t know before hand whether a problem is ill-conditioned, for instance a generic linear system of equations. Sometimes the problem will be well-conditioned, sometimes ill-conditioned. What we want is for our algorithm to somehow mimic the behaviour of the underlying mathematical problem. It can’t mimic it perfectly, but at least we can hope that the algorithm has a similar conditioning behaviour.

An appropriate aim is to develop *stable* algorithms.

Stability

Definition 18 (Stability). *An algorithm $\tilde{f}$ is stable if for each $x \in X$*

$$\frac{||f(\tilde{x}) - \tilde{f}(x)||}{||f(\tilde{x})||} = O(\varepsilon)$$

for some $\tilde{x}$ with

$$\frac{||x - \tilde{x}||}{||x||} = O(\varepsilon).$$

As very nicely stated by Nick Trefethen and David Bau in their book:
A stable algorithm gives nearly the right answer to nearly the right question.

Backward Stability

Often algorithms satisfy a stronger and simpler condition.

Definition 19 (Backward Stability)**.** *An algorithm $\tilde{f}$ for a problem f is backward stable if for each $x \in X$*

$$\tilde{f}(x) = f(\tilde{x})$$

for some $\tilde{x}$ with

$$\frac{||x - \tilde{x}||}{||x||} = O(\varepsilon).$$

A backward stable algorithm gives exactly the right answer to nearly the right question.

Example: Stability of Floating Point Arithmetic

All standard operations $+$, $-$, $\times$ and $\div$ are stable. Let's consider subtraction $\tilde{f}(x_1, x_2) = \mathtt{fl}(\mathtt{fl}(x_1) - \mathtt{fl}(x_2))$.

Example 22 ($\tilde{f}(x_1, x_2) = \mathtt{fl}(\mathtt{fl}(x_1) - \mathtt{fl}(x_2))$)**.** *Analyse*

$$\begin{aligned}
\mathtt{fl}(\mathtt{fl}(x_1) - \mathtt{fl}(x_2)) &= \mathtt{fl}(x_1(1+\delta_1) - x_2(1+\delta_2)) \\
&= (x_1(1+\delta_1) - x_2(1+\delta_2))(1+\delta_3) \\
&= x_1(1+\delta_1)(1+\delta_2) - x_2(1+\delta_2)(1+\delta_3) \\
&= x_1(1+\delta_4) - x_2(1+\delta_5)
\end{aligned}$$

where $|\delta_4|, |\delta_5| \leq 2\varepsilon$.

So $\tilde{f}(x_1, x_2) = \tilde{x}_1 - \tilde{x}_2$ where

$$\frac{||x_1 - \tilde{x}_1||}{||x_1||} = O(\varepsilon) \quad \text{and} \quad \frac{||x_2 - \tilde{x}_2||}{||x_2||} = O(\varepsilon).$$

An Unstable Algorithm

Example 23 (Eigenvalues)**.** *Let us calculate the eigenvalues of a matrix using the algorithm;*

1. *find the coefficients of the characteristic polynomial,*
2. *find the roots of the polynomial.*

We have already seen that finding roots of a polynomial is ill conditioned. So overall algorithm will be unstable.

Try finding the eigenvalues of

$$\begin{bmatrix} 1 + 10^{-14} & 0 \\ 0 & 1 \end{bmatrix}$$

using this algorithm. (I get $x_{\pm} = 1.0 \pm 1.490 \times 10^{-08} i$). We lose 8 significant figures of accuracy.

2.3.5 Accuracy of a Backward Stable Algorithm

Backward Stability and Relative Errors

Suppose a backward stable algorithm $\tilde{f}$ is applied to solve a problem f with condition number κ. Then the relative errors satisfy

$$\frac{||f(x) - \tilde{f}(x)||}{||f(x)||} = O(\kappa(x)\,\varepsilon).$$

Proof. By backwards stability $\tilde{f}(x) = f(\tilde{x})$ so

$$\begin{aligned}\frac{||f(x)-\tilde{f}(x)||}{||f(x)||} &= \frac{||f(x)-f(\tilde{x})||}{||f(x)||} \\ &= \left[\frac{||f(x)-f(\tilde{x})||}{||f(x)||}\,\frac{||x||}{||x-\tilde{x}||}\right]\frac{||x-\tilde{x}||}{||x||} \\ &= O(\kappa(x)\,\varepsilon).\end{aligned}$$

□

This is actually a very important result. Recall that the father of numerical analysis, Jim Wilkinson had so much trouble calculating accurately the roots of polynomials. Well he was also very involved (during and after the second world war) in the solution of linear systems of equations. Great mathematicians, such a John von Neumann had analysed the errors involved in solving linear systems and had concluded that it was virtually impossible to solve linear systems with over 20 equation. At least is was not possible to provide a mathematical foundation to provide assurance that the results obtained for large systems of equations were accurate. This was because von Neumann had provided a *forward error analysis.* It was only later that von Neumann realised that a method employing backwards stability should be used. The method of determining the backwards stability of an algorithm is known as *backward error analysis.*

Gaussian Elimination

It can be shown that Gaussian elimination is “usually” a backward stable algorithm for solving linear systems.

The solution produced by Gaussian elimination with partial pivoting (explained in the matrix module), $\tilde{\boldsymbol{x}}$, satisfies $(A+\delta A)(\tilde{\boldsymbol{x}}) = \boldsymbol{b}$ where

$$\frac{\|\delta A\|}{\|A\|} \le \rho\, n\, \varepsilon.$$

The factor ρ is called the growth factor and is equal to the ratio of largest value of U to largest value of A. So $\tilde{\boldsymbol{x}}$ solves nearly the right question, provided ρ is not too large.

The growth factor ρ can be large. Consider

$$A = \begin{bmatrix} 1 & & 0 & 0 & 1 \\ -1 & \ddots & & 0 & 1 \\ -1 & \ddots & \ddots & & 1 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ -1 & & -1 & -1 & 1 \end{bmatrix}.$$

Then $\rho = 2^{n-1}$. But this is very rare. For most matrices $\rho \approx 2$–10 and indeed seems to satisfy $O(n^{1/2})$ on average.

In partial pivoting we interchange rows, but we can also interchange columns as well as rows. This leads to complete pivoting, where the growth factor has been shown to act like $n^{1/2}$.

So Gaussian Elimination is “backwards stable”. So the computed solution $\tilde{\boldsymbol{x}}$ satisfies the equation $(A+\delta A)\,\tilde{\boldsymbol{x}} = \boldsymbol{b}$.

Now $\delta A\,\tilde{\boldsymbol{x}} = \boldsymbol{b} - A\,\tilde{\boldsymbol{x}} = \boldsymbol{r}$ (residual). The norm of the residual is bounded

$$\|r\| \le \|\delta A\|\,\|\tilde{\boldsymbol{x}}\|$$

and so

$$\frac{\|r\|}{\|A\|\,\|\boldsymbol{x}\|} \le \frac{\|\delta A\|}{\|A\|} \le \rho\, n\, \varepsilon.$$

The error satisfies

$$\frac{\|\boldsymbol{x}-\tilde{\boldsymbol{x}}\|}{\|\boldsymbol{x}\|} \leq \kappa(A)\rho\, n\, \varepsilon.$$

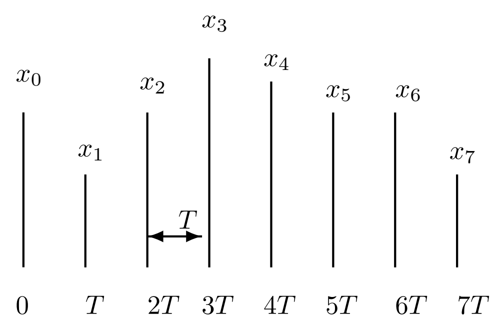
So the Residual is usually small $\approx \varepsilon$ (provided the growth factor is small) where as the the error can be very large if $\kappa(A)$ is large.

2.4 Discrete Fourier Transformation

2.4.1 Frequency Domains

Remotely controlling a processing plant – example of signals

- digital signals are sent between processing plant and controlling computer
- these signals are functions of time, sampled at intervals T
- same situation as in interpolation but here we consider trigonometric polynomials instead of algebraic ones



Time and Frequency

- the data sent between controller and plant are in the *time domain*
- here we will discuss *frequency domain* representations which are the coefficients in sums of sines and cosines
- frequency of component $\sin(2\pi f t)$ is f
- this is used for periodic and non-periodic functions, for sampled signals and even for finite signals
- alternative methods based on wavelets for example or chirplets
- also used in image processing and for the solution of PDEs
- algebra is simpler if one uses complex numbers ...

Background

Theorem 4 (Euler's Identity)**.**

$$e^{i\theta} = \cos\theta + i\sin\theta.$$

$$\begin{aligned} e^{-i\theta} &= \cos(-\theta) + i\sin(-\theta) \\ &= \cos\theta - i\sin\theta. \end{aligned}$$

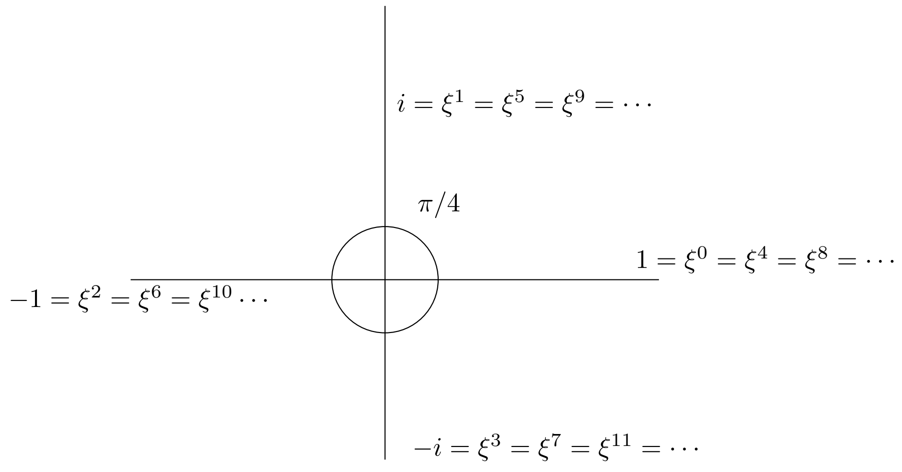

$i = \xi^1 = \xi^5 = \xi^9 = \cdots$
$\pi/4$
$1 = \xi^0 = \xi^4 = \xi^8 = \cdots$
$-1 = \xi^2 = \xi^6 = \xi^{10} \cdots$
$-i = \xi^3 = \xi^7 = \xi^{11} = \cdots$

Figure 2.3: The n roots of unity.

Thus

$$\cos(2\pi kt) = \frac{e^{2\pi ikt} + e^{-2\pi ikt}}{2}.$$
$$\sin(2\pi kt) = \frac{e^{2\pi ikt} - e^{-2\pi ikt}}{2i}.$$

For a given n, let

$$\xi = e^{\frac{2\pi i}{n}},$$

be the primitive nth root of unity. The n roots of unity are given by ξ^k, $k = 0, \cdots, n-1$.

2.4.2 Fourier Transformation

Continuous Fourier Transform

Definition 20 (Continuous Fourier Transformation)**.** *If $h(t)$ is defined for all $t \in (-\infty, \infty)$, then we can write*

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-2\pi ift} df,$$

where $H(f)$ is the continuous Fourier transform (CFT) of h and

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi ift} dt.$$

If t is time, in say units of seconds, then f represents frequency in units of cycles per second or Hertz.

Some Properties of the CFT

It is a linear transform.

if	then
$h(t)$ is real	$H(-f) = \overline{H(f)}$
$h(t)$ is imaginary	$H(-f) = -\overline{H(f)}$
$h(t)$ is even	$H(-f) = H(f)$
$h(t)$ is odd	$H(-f) = -H(f)$
$h(t)$ is real and even	$H(f)$ is real and even
$h(t)$ is real and odd	$H(f)$ is imaginary and odd
$h(t)$ is imaginary and even	$H(f)$ is imaginary and odd
$h(t)$ is imaginary and odd	$H(f)$ is real and even

As an example, lets show that if $h(t)$ is real $H(-f) = \overline{H(f)}$.

Proof. Note $\overline{a + ib} = a - ib$. So if, $h(t)$ is real

$$\begin{aligned} \overline{H(f)} &= \overline{\int_{-\infty}^{\infty} h(t)\exp(2\pi i f t)dt} \\ &= \overline{\int_{-\infty}^{\infty} h(t)[\cos(2\pi f t) + i\sin(2\pi f t)]dt} \\ &= \int_{-\infty}^{\infty} h(t)[\cos(2\pi f t) - i\sin(2\pi f t)]dt \\ &= \int_{-\infty}^{\infty} h(t)e^{-2i\pi f t}dt \\ &= H(-f). \end{aligned}$$

□

Now show if $h(t)$ is even $(h(-t) = h(t))$ $H(-f) = H(f)$.

Proof. Set $t' = -t$. Note $dt = -dt'$.

$$\begin{aligned} H(-f) &= \int_{-\infty}^{\infty} h(t)e^{i2\pi(-f)t}dt \\ &= \int_{\infty}^{-\infty} h(-t')e^{i2\pi(-f)(-t')}(-dt') \\ &= \int_{-\infty}^{\infty} h(-t')e^{i2\pi f t'}(dt') \\ &= \int_{-\infty}^{\infty} h(t')e^{i2\pi f t'}(dt') \\ &= H(f). \end{aligned}$$

□

Finally, using the above results it is straight forward to show that if $h(t)$ is real and even, $H(f)$ is real and even.

Proof. Since $h(t)$ is even, $H(-f) = H(f)$. That is $H(f)$ is even. Now, since $h(t)$ is real $\overline{H(f)} = H(-f) = H(f)$ and thus $H(f)$ is real. □

Some Transformation Rules

Let $h(t) \Longleftrightarrow H(f)$ imply that $h(t)$ and $H(f)$ are transform pairs. Then

$$\begin{aligned} h(at) &\Longleftrightarrow \frac{1}{|a|}H\left(\frac{f}{a}\right) && \text{time scaling} \\ \frac{1}{|b|}h\left(\frac{t}{b}\right) &\Longleftrightarrow H(bf) && \text{frequency scaling} \\ h(t - t_0) &\Longleftrightarrow H(f)e^{2\pi i f t_0} && \text{time shifting} \\ h(t)e^{-2\pi i f_0 t} &\Longleftrightarrow H(f - f_0) && \text{frequency shifting} \end{aligned}$$

Lets show $h(at) \Longleftrightarrow \frac{1}{|a|}H\left(\frac{f}{a}\right)$.

Proof. Let $f' = af$. Then $df = df'/a$.

$$\begin{aligned} h(at) &= \int_{-\infty}^{\infty} H(f)e^{-2\pi i f(at)}df \\ &= \int_{-\infty}^{\infty} H(f'/a)e^{-2\pi i f' t}df'/|a| \\ &= \int_{-\infty}^{\infty} \left(\frac{H(f'/a)}{|a|}\right) e^{-2\pi i f' t}df'. \end{aligned}$$

□

Show $h(t-t_0) \iff H(f)e^{2\pi i f t_0}$.

Proof.

$$\begin{aligned} h(t-t_0) &= \int_{-\infty}^{\infty} H(f)e^{-2\pi i f(t-t_0)}df \\ &= \int_{-\infty}^{\infty} H(f)e^{-2\pi i f t}e^{2\pi i f t_0}df \\ &= \int_{-\infty}^{\infty} \left(H(f)e^{2\pi i f t_0}\right) e^{-2\pi i f t}df. \end{aligned}$$

□

2.4.3 Discrete Fourier Transformation

Discrete Fourier Transform Definition

Definition 21 (Discrete Fourier Transformation)**.** *The discrete Fourier transform (DFT)*

$$\boldsymbol{x} \to \hat{\boldsymbol{x}}$$

of n points x_k, $(0 \le k \le n-1)$ is

$$\hat{x}_k := \sum_{j=0}^{n-1} x_j e^{(2\pi jk/n)i} = \sum_{j=0}^{n-1} x_j \xi^{kj},$$

where $\xi = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$.

That is

$$\hat{\boldsymbol{x}} = F\boldsymbol{x}$$

where $F_{kj} = \xi^{kj}$.

Properties of the Discrete Fourier Transform

Some properties of the discrete Fourier transformation as given below. Note the similarity to the results for the continuous Fourier transformation.

- Linearity

  $$\alpha\boldsymbol{x} + \beta\boldsymbol{y} \to \alpha\hat{\boldsymbol{x}} + \beta\hat{\boldsymbol{y}},$$

  for any scalars α, β.

- Time Shifting

  $$x_{j-j_0} \to e^{(2\pi j_0 k i)/n}\hat{x}_k.$$

  Note that the indexing is mod n.

- Frequency Shifting

$$e^{-(2\pi j k_0 i)/n} x_j \to \hat{x}_{k-k_0}.$$

Lets prove the time shifting property.

Proof.

$$\begin{aligned}
\sum_{j=0}^{n-1} x_{j-j_0}\xi^{kj} &= \sum_{j=0}^{n-1} x_{j-j_0}\xi^{(j-j_0)k}\xi^{j_0 k} \\
&= \xi^{j_0 k}\sum_{j=-j_0}^{n-1-j_0} x_j \xi^{jk} \\
&= \xi^{j_0 k}\left(\sum_{j=0}^{n-1-j_0} x_j\xi^{jk} + \sum_{j=-j_0}^{-1} x_j\xi^{jk}\right) \\
&= \xi^{j_0 k}\left(\sum_{j=0}^{n-1-j_0} x_j\xi^{jk} + \sum_{j=-j_0}^{-1} x_{j+n}\xi^{(j+n)k}\right) \\
&= \xi^{j_0 k}\left(\sum_{j=0}^{n-1-j_0} x_j\xi^{jk} + \sum_{j=n-j_0}^{n-1} x_j\xi^{(j)k}\right) \\
&= \xi^{j_0 k}\hat{x}_k.
\end{aligned}$$

□

Gauss Relationship

Theorem 5 (Gauss Relationship)**.**

$$\sum_{k=0}^{n-1}\xi^{ak} = \begin{cases} n & \textit{if } \xi^a = 1 \\ 0 & \textit{otherwise} \end{cases},$$

where a is an integer.

Proof. Firstly note that $\sum_{k=0}^{n-1} b^k = \frac{1-b^n}{1-b}$.

If $b = \xi^a = e^{\frac{2\pi a}{n}i}$, then

$$\begin{aligned}
\sum_{k=0}^{n-1}\xi^{ak} &= \frac{1-\xi^{an}}{1-\xi^a} \\
&= \frac{1-\exp(\frac{2\pi}{n}i\,an)}{1-\exp(\frac{2\pi}{n}i\,a)} \\
&= \frac{\exp(\frac{\pi}{n}i\,an)\left(\exp(\frac{-\pi}{n}i\,an)-\exp(\frac{\pi}{n}ian)\right)}{\exp(\frac{\pi}{n}i\,a)\left(\exp(\frac{-\pi}{n}i\,a)-\exp(\frac{\pi}{n}ia)\right)} \\
&= \exp(\frac{\pi}{n}i\,a(n-1))\frac{2i\sin\left(\frac{\pi a}{n}n\right)}{2i\sin\left(\frac{\pi a}{n}\right)} \\
&= \exp(\frac{\pi}{n}i\,a(n-1))\frac{\sin(\pi a)}{\sin\left(\frac{\pi a}{n}\right)}.
\end{aligned}$$

The numerator is 0 for values of a (a is an integer). The denominator is 0 if and only if a is a multiple of n.

L'Hopitals rule says that if both the numerator and denominator are 0, the fraction has magnitude n. More precisely. If $f(c) = g(c) = 0$, L'Hopitals rule says

$$\lim_{x\to c}\frac{f(x)}{g(x)} = \lim_{x\to 0}\frac{f'(x)}{g'(x)} = \frac{\pi\cos(\pi a)}{\pi/n\cos(\pi a/n)} = n.$$

Alternatively, $\sum_{k=0}^{n-1} 1^k = n$, since $\xi^a = 1$ if and only if a is a multiple of n. □

Inverse Discrete Fourier Transform

Theorem 6. *From the Gauss Relationship we get*

$$F\bar{F} = nI,$$

($\bar{F}$ is the complex conjugate).

Proof.

$$\begin{aligned}(F\bar{F})_{kj} &= \sum_{l=0}^{n-1} F_{kl}\bar{F}_{lj}\\ &= \sum_{l=0}^{n-1}\xi^{kl}\overline{\xi^{lj}}\\ &= \sum_{l=0}^{n-1}\xi^{kl}\xi^{-lj}\\ &= \sum_{l}\xi^{l(k-j)}\\ &= \left\{\begin{array}{cc} n & \text{if } j=k\\ 0 & \text{otherwise}\end{array}\right. ,\end{aligned}$$

since

$$\sum_{k=0}^{n-1}\xi^{ak} = \left\{\begin{array}{cc} n & \text{if } \xi^a = 1\\ 0 & \text{otherwise}\end{array}\right. .$$

□

The above theorem proves that the inverse of F exists and in-turn defines the inverse discrete Fourier transformation.

Theorem 7 (Inverse Discrete Fourier Transformation)**.** *The inverse discrete Fourier transform (IDFT) is*

$$\boldsymbol{x} = \frac{1}{n}\bar{F}\hat{\boldsymbol{x}}.$$

Or,

$$x_k = \frac{1}{n}\sum_{j=0}^{n-1}\hat{x}_j\xi^{-kj}.$$

2.4.4 Convolutions

Circular Convolution

Definition 22 (Circular Convolution)**.** *The circular convolution of two n-tuples* $\boldsymbol{x}$ *and* $\boldsymbol{y}$

$$\boldsymbol{x} * \boldsymbol{y} = \boldsymbol{z}$$

is an n-tuple $\boldsymbol{z}$ *given by the rule*

$$z_k = \sum_{j=0}^{n-1} x_j y_{k-j}$$

(where subscripts are computed modulo n).

Example 24 (Circular Convolution)**.** *Given* $x = (1, 1, 2)$ *and* $y = (-1, 3, 4)$*, then*

$$\begin{aligned} z_0 &= 1 \times -1 + 1 \times 4 + 2 \times 3 = 9 \\ z_1 &= 1 \times 3 + 1 \times -1 + 2 \times 4 = 10 \\ z_2 &= 1 \times 4 + 1 \times 3 + 2 \times -1 = 5. \end{aligned}$$

Recall $z_0 = \sum_{j=0}^{n-1} x_j y_{-j} = \sum_{j=0}^{n-1} x_j y_{n-j}$.

Transformations of Convolutions

Theorem 8 (Convolution Transformation)**.** *The transformation of a convolution is a coordinate wise product and the transformation of a coordinate wise-product is* $1/n$ *times a convolution. That is*

$$\boldsymbol{x} * \boldsymbol{y} \to \hat{\boldsymbol{x}}.\hat{\boldsymbol{y}},$$

$$\boldsymbol{x}.\boldsymbol{y} \to \frac{1}{n}\hat{\boldsymbol{x}} * \hat{\boldsymbol{y}},$$

Proof. Let $\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{y}$. Then

$$\begin{aligned} \hat{z}_k &= \sum_{j=0}^{n-1} z_j \xi^{kj} \\ &= \sum_{j=0}^{n-1} (\boldsymbol{x} * \boldsymbol{y})_j \xi^{kj} \\ &= \sum_{j=0}^{n-1} \left(\sum_{m=0}^{n-1} x_m y_{j-m} \right) \xi^{kj} \\ &= \sum_{m=0}^{n-1} x_m \sum_{j=0}^{n-1} y_{j-m} \xi^{kj} \\ &= \sum_{m=0}^{n-1} x_m \xi^{km} \sum_{j=0}^{n-1} y_{j-m} \xi^{(j-m)k} \\ &= \left(\sum_{m=0}^{n-1} x_m \xi^{km} \right) \left(\sum_{j=0}^{n-1} y_j \xi^{kj} \right) \\ &= \hat{x}_k.\hat{y}_k. \end{aligned}$$

The last step follows since y_{j-m} and $\xi^{(j-m)k}$ is modulo n.

Now assume $\hat{\boldsymbol{z}} = \hat{\boldsymbol{x}} * \hat{\boldsymbol{y}}$, in which case

$$\begin{aligned}
\hat{z}_k &= \sum_{j=0}^{n-1} \hat{x}_j \hat{y}_{k-j} \\
&= \sum_{j=0}^{n-1} \left(\sum_{p=0}^{n-1} x_p \xi^{pj} \right) \left(\sum_{q=0}^{n-1} y_q \xi^{q(k-j)} \right) \\
&= \left(\sum_{p,q=0}^{n-1} x_p y_q \xi^{qk} \right) \left(\sum_{j=0}^{n-1} \xi^{j(p-q)} \right) \\
&= n \sum_{p=0}^{n-1} x_p y_p \xi^{pk}.
\end{aligned}$$

The last steps follows from the Gauss Identity. □

Theorem 9. *Filtering a periodic input $\boldsymbol{u} = (u_0, u_1, \cdots, u_{n-1})$ by circular convolution by the finite impulse response,*

$$\boldsymbol{h} = (h_0, h_1, \cdots, h_{n-1})$$

($\boldsymbol{y} = \boldsymbol{h} * \boldsymbol{u}$) becomes a point-wise product in the frequency domain. That is

$$\hat{\boldsymbol{y}} = \hat{\boldsymbol{h}}.\hat{\boldsymbol{u}}.$$

2.4.5 Fast Fourier Transformation

Fast Fourier Transform

How should we calculate the DFT? Calculating

$$\xi^{kj} = \cos 2\pi kj/n + i \sin 2\pi kj/n$$

is expensive. It is possible to pre-calculate and store ξ^{kj} but n^2 multiplications remain.

Lets consider the case where $n = 4$. The discrete Fourier transform is

$$\begin{aligned}
\hat{x}_0 &= x_0\xi^0 + x_1\xi^0 + x_2\xi^0 + x_3\xi^0, \\
\hat{x}_1 &= x_0\xi^0 + x_1\xi^1 + x_2\xi^2 + x_3\xi^3, \\
\hat{x}_2 &= x_0\xi^0 + x_1\xi^2 + x_2\xi^4 + x_3\xi^6, \\
\hat{x}_3 &= x_0\xi^0 + x_1\xi^3 + x_2\xi^6 + x_3\xi^9.
\end{aligned}$$

16 multiplications and 12 additions are required.

Lets rewrite the discrete Fourier transformation as

$$\begin{aligned}
\hat{x}_0 &= (x_0 + \xi^0 x_2) + \xi^0(x_1 + \xi^0 x_3), \\
\hat{x}_1 &= (x_0 + \xi^2 x_2) + \xi^1(x_1 + \xi^2 x_3), \\
\hat{x}_2 &= (x_0 + \xi^4 x_2) + \xi^2(x_1 + \xi^4 x_3), \\
\hat{x}_3 &= (x_0 + \xi^6 x_2) + \xi^3(x_1 + \xi^6 x_3).
\end{aligned}$$

This rewrite reduces the number of operations to 12 multiplications and 12 additions/subtractions. The above idea can be extended to larger problems.

If $n = 2^\alpha$, we can use the fast-Fourier transform (FFT). Specifically, suppose $n = 2m$, $k \leq m-1$.

Then

$$\begin{aligned}\hat{x}_k &= x_0 + x_1\xi^k + x_2\xi^{2k} + \cdots + x_{n-1}\xi^{(n-1)k}\\ &= x_0 + x_2(\xi^2)^k + x_4(\xi^2)^{2k} + \cdots + x_{2(m-1)}(\xi^2)^{(m-1)k}\\ &+ \xi^k(x_1 + x_3(\xi^2)^k + x_5(\xi^2)^{2k} + \cdots + x_{2m-1}(\xi^2)^{(m-1)k}).\end{aligned}$$

Hence

$$\hat{x}_k = \hat{y}_k + \xi^k\hat{z}_k$$

where

$$\hat{y}_k = y_0 + y_1\zeta^k + y_2\zeta^{2k} + \cdots + y_{m-1}\zeta^{(m-1)k}$$
$$\hat{z}_k = z_0 + z_1\zeta^k + z_2\zeta^{2k} + \cdots + z_{m-1}\zeta^{(m-1)k}$$

and $\zeta = \xi^2$, $y_i = x_{2i}$ and $z_i = x_{2i+1}$.

That is, the DFT of a frame of length $n = 2m$ is the weighted sum of two DFTs of half the frame length.

An additional simplification can also be used. Since $\xi^{n/2} = -1$, we have

$$\begin{aligned}\hat{x}_k &= \hat{y}_k + \xi^k\hat{z}_k\\ \hat{x}_{k+m} &= \hat{y}_k - \xi^k\hat{z}_k,\end{aligned}$$

for $(0 \le k \le m-1)$. To see why, observe that

$$\begin{aligned}\hat{x}_{k+m} &= x_0 + x_1\xi^{(k+m)} + x_2\xi^{2(k+m)} + \cdots\\ &= x_0 + x_2(\xi^2)^{(k+m)} + \cdots + x_{2(m-1)}(\xi^2)^{(m-1)(k+m)}\\ &+ \xi^{(k+m)}\left(x_1 + x_3(\xi^2)^{(k+m)} + \cdots + x_{2m-1}(\xi^2)^{(m-1)(k+m)}\right).\end{aligned}$$

But,

$$\begin{aligned}\xi^{2(k+m)} &= \xi^{2k}\xi^{2m}\\ &= \xi^{2k}\xi^{n}\\ &= \xi^{2k},\end{aligned}$$

and

$$\begin{aligned}\xi^{(k+m)} &= \xi^{k}\xi^{m}\\ &= \xi^{k} \times \xi^{n/2}\\ &= -\xi^{k}.\end{aligned}$$

So

$$\begin{aligned}\hat{x}_{k+m} &= x_0 + x_2(\xi^2)^k + \cdots + x_{2(m-1)}(\xi^2)^{(m-1)k}\\ &- \xi^k\left(x_1 + x_3(\xi^2)^k + \cdots + x_{2m-1}(\xi^2)^{(m-1)k}\right)\\ &= \hat{y}_k - \xi^k\hat{z}_k.\end{aligned}$$

Applying the above simplification to the 4×4 example gives,

$$\begin{aligned}\hat{x}_0 &= (x_0 + \xi^0x_2) + \xi^0(x_1 + \xi^0x_3) = (x_0 + \xi^0x_2) + \xi^0(x_1 + \xi^0x_3),\\ \hat{x}_1 &= (x_0 + \xi^2x_2) + \xi^1(x_1 + \xi^2x_3) = (x_0 - \xi^0x_2) + \xi^1(x_1 - \xi^0x_3),\\ \hat{x}_2 &= (x_0 + \xi^4x_2) + \xi^2(x_1 + \xi^4x_3) = (x_0 + \xi^0x_2) - \xi^0(x_1 + \xi^0x_3),\\ \hat{x}_3 &= (x_0 + \xi^6x_2) + \xi^3(x_1 + \xi^6x_3) = (x_0 - \xi^0x_2) - \xi^1(x_1 - \xi^0x_3),\end{aligned}$$

which only requires 4 multiplications and 8 add/sub.

Matrix Notation

In general we can factor the nth Fourier matrix

$$F_n = (\xi^{kj})$$

into

$$F_n = B_n \begin{pmatrix} F_m & 0 \\ 0 & F_m \end{pmatrix} P_n$$

where B_n is the *butterfly* matrix

$$B_n = \begin{pmatrix} I_m & \Theta_m \\ I_m & -\Theta_m \end{pmatrix}.$$

I_m is the $m \times m$ identity matrix, and

$$\Theta = \text{diag}(1, \xi_n^1, \xi_n^2, \cdots, \xi_n^{(m-1)}).$$

P_n is the permutation matrix that orders all of the even numbered entries first.

For example, consider

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \xi^1 & \xi^2 & \xi^3 \\ 1 & \xi^2 & \xi^4 & \xi^6 \\ 1 & \xi^3 & \xi^6 & \xi^9 \end{bmatrix},$$

where $\xi = \exp(2\pi i/4) = i$.

Then

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Let P_4 be the 4×4 permutation matrix

$$P_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

P_4 permutes the indices so all of the even ones are grouped first

$$F_4 P_4 = \left[\begin{array}{rr|rr} 1 & 1 & 1 & 1 \\ 1 & -1 & i & -i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & -i & i \end{array}\right]$$

Now define the diagonal matrix

$$\Theta_2 = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \text{diag}(1, \xi^1).$$

Let B_4 be the butterfly matrix

$$B_4 = \begin{bmatrix} I_2 & \Theta_2 \\ I_2 & -\Theta_2 \end{bmatrix}.$$

Recall

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Then

$$F_4 = B_4 \begin{bmatrix} F_2 & 0 \\ 0 & F_2 \end{bmatrix} P_4.$$

Note B_4 finds the coefficients, the middle diagonal matrix does the lower order FFT and P_4 permutes to even followed by odd.

We can continue to factor the matrices. Two successive factorisations would look like

$$\begin{aligned} F_n \quad &= \quad B_n \begin{pmatrix} B_{n/2} & 0 \\ 0 & B_{n/2} \end{pmatrix} \begin{pmatrix} F_{n/4} & 0 & 0 & 0 \\ 0 & F_{n/4} & 0 & 0 \\ 0 & 0 & F_{n/4} & 0 \\ 0 & 0 & 0 & F_{n/4} \end{pmatrix} \\ &\begin{pmatrix} P_{n/2} & 0 \\ 0 & P_{n/2} \end{pmatrix} P_n. \end{aligned}$$

The permutations can be done very quickly using bit ordering techniques. The Θ_k matrices are generally stored off-line. There are very efficient implementations of the algorithm to make good use of the cache and vector processors etc.

Algorithm 1. *FFT(h, H, n, ξ)*

```
1: if $n = 1$ then
2:   $H[0] = h[0]$
3: else
4:   for $k = 0$ to $\frac{n}{2} - 1$ do
5:     $p[k] = h[2k]$
6:     $s[k] = h[2k+1]$
7:   end for
8:   FFT(p, q, $n/2$, $\xi^2$)
9:   FFT(s, t, $n/2$, $\xi^2$)
10:  for $k = 0$ to $n - 1$ do
11:    $H[k] = q[k \ mod(n/2)] + \xi^k t[k \ mod(n/2)]$
12:  end for
13: end if
```

Oboe Problem

Lets use the following code to simulate the frequencies found in the Oboe.

```
fS = 8192.0                 # set sampling rate
t = arange(0,fS-1)          # sampling vector
t = t/fS                    # sampling times
w = 2.0*pi*440.0            # radian A 440 Hz
x = sin(w*t)                # fundamental
x = x+0.2*sin(2*w*t)        # first overtone
x = x+0.1*sin(3.01*w*t)     # second overtone
x = x+0.4*random_sample(len(x))+x
                            # wind noise
subplot(311), plot(t, x), title('Oboe')
                            # plot the input
```

(Note: have not shown import statements)

If we perform the following transformation

```
n = 8192;                 # set FFT frame length
m = 400;                  # set viewing window min herz
M = 1400;                 # set viewing window max herz
xhat = fft(x);            # perform FFT
xhat = xhat/n;            # normalise
```

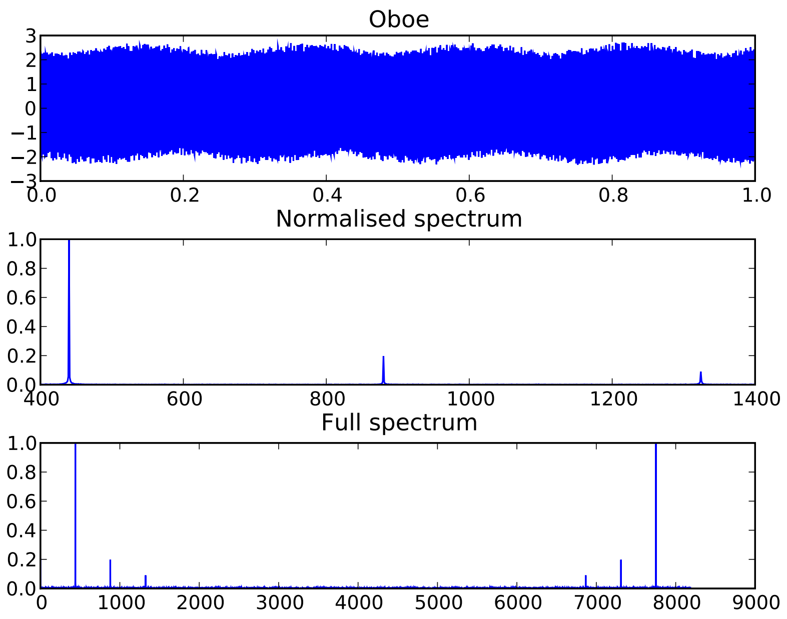


Figure 2.4: The top diagram shows the signal in the time domain. It is a discrete signal of the A-note being played by the oboe. The bottom two diagrams show the signal in the frequency domain. Once transformed into the frequency domain it is clear the A-note is predominantly made up of three different frequencies.

```
f = arange(0,n—1);          # initialise viewing window
f = f[m:M];                 # define viewing window
yhat = xhat[m:M];           # ordinate viewing window
subplot(312), plot(f,abs(yhat)), title('Normalised spectrum');
                            # plot the spectrum
subplot(313), plot(t*fS,abs(xhat)), title('Full spectrum')
                            # plot the full, spectrum
subplots_adjust(hspace=0.4), show()
```

(Note: have not shown import statements)

We get the spectrum diagrams shown in Figure 2.4

2.4.6 Two Dimensional Discrete Fourier Transformations

Definition of the Two Dimensional Transformation

Definition 23 (Two Dimensional Discrete Fourier Transformation)**.** *The discrete Fourier transform of an $m \times n$ matrix $X = (x_{ij})$ is an $m \times n$ matrix $\hat{X} = (\hat{x}_{ij})$.*

$$\begin{aligned} \hat{x}_{ij} &= \sum_{p,q=1}^{m,n} x_{pq}\xi_m^{pi}\xi_n^{qj}, \\ x_{ij} &= (mn)^{-1}\sum_{p,q=1}^{m,n} \hat{x}_{pq}\xi_m^{-pi}\xi_n^{-qj}. \end{aligned}$$

Image Processing

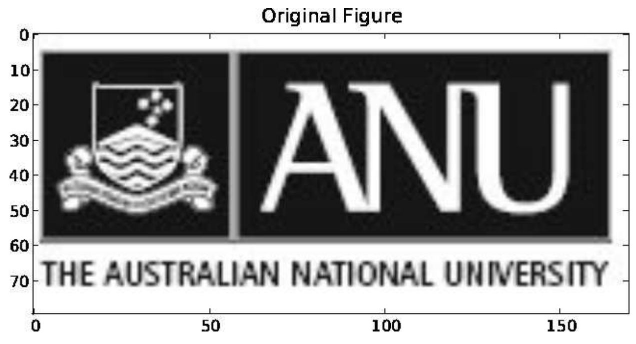


Figure 2.5: Original photo.

Think of an image as composed of $m \times n$ pixels with each pixel assigned some integer intensity $0 \le x_{ij} \le W$. The image may be enhanced by performing some filter H on the image X to create a new image Y.

Many of these routines can be written as two-dimensional circular convolutions

$$Y = H * X$$

$$y_{ij} = \sum_{p,q=}^{m,n} h_{p,q} x_{i-p,j-q}.$$

(subscripts are computed modulo m and n). Note that as with the one dimensional transformations $X * Y \to \hat{X}.\hat{Y}$ and $X.Y \to (mn)^{-1}\hat{X} * \hat{Y}$.

As a convolution in the time domain is equivalent to pointwise product in the frequency domain these transformations are often performed in the frequency domain.

For example, high frequency components of an image are usually associated with noise. An *ideal low pass filter* will remove the high frequency components. In particular the filter is given by

$$\widehat{h}_{ij} = \begin{cases} 1 & \text{if } \sqrt{(i^2 + j^2)} \le k \\ 0 & \text{otherwise} \end{cases},$$

where k is the cut-off frequency.

Example - Image Blurring

Image X transmitted through a channel with known linear blurring B giving the blurred image $Y = B * X$ can be de-blurred by inverting the convolution. This is done with the aid of a DFT.

To de-blur an image, transform the blurred image $Y = B * X$ to the frequency-domain version $\hat{Y}$, divide each entry by the corresponding entry of blurring $\hat{B}$, then transform back;

$$x_{ij} = (mn)^{-1} \sum_{p,q=1}^{m,n} \frac{\hat{y}_{pq}}{\hat{b}_{pq}} \xi_m^{pi} \xi_n^{pj}.$$

(assuming $\hat{b}_{pq} \neq 0$).

Consider the original photo given in Figure 2.5.

The photo can be read into Python using the following code

```
from matplotlib.image import imread
from matplotlib.pyplot import imshow, show, gray, figure, title
from numpy.random import random_sample
from numpy.fft import fft, ifft
from numpy import amax, amin, mean
```

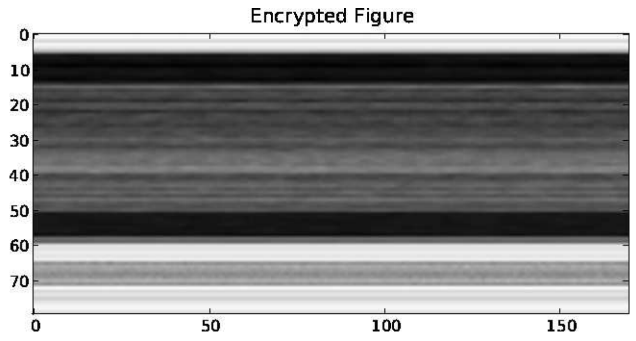


Figure 2.6: The blurred image.

```
# Read in the image file

X=imread('anu.png')          # import as a matrix
X = mean(X, 2)               # read as RGB convert to gray
figure(1),imshow(X), gray()  # plot the image
title('Original Figure')     # give the image a title
```

Use the circular convolution with the blurring matrix B to encrypt the photo.

```
B = random_sample(X.shape) # choose a blurring matrix
Xhat = fft(X)              # transform X;
Bhat = fft(B)              # transform B;
Yhat = Bhat*Xhat           # multiply entrywise
Y = ifft(Yhat)             # transform back
Y = abs(Y)                 # convert to float
```

View scrambled image:

```
Z = Y-amin(Y)
Z = 255.0*Z/amax(Z)          # rescale
figure(2),imshow(Z), gray()  # display the encrypted image
title('Encrypted Figure')    # give the image a title
```

The resulting photo is given in Figure 2.6.
Finally, use the inverse transformation to decode the image.

```
Yhat = fft(Y)          # transform Y;
Bhat = fft(B)          # transform B;
Xhat = Yhat/Bhat       # divide entrywise
X1 = fft(Xhat)         # transform back
X1 = real(X1)          # delete small imag. pargs
X1 = round(X1)         # round to nearest integer
imshow(X1)             # display the image
```

The decoded photo is shown in Figure 2.7

2.5 Iterative Solution of Linear Equations

- Gaussian elimination is a direct method for solving linear system, that gives exact solution (up to the machine accuracy) in a finite number of operations.

- It requires $\frac{1}{3}n^3 + O(n^2)$ operations and n^2 storage locations.

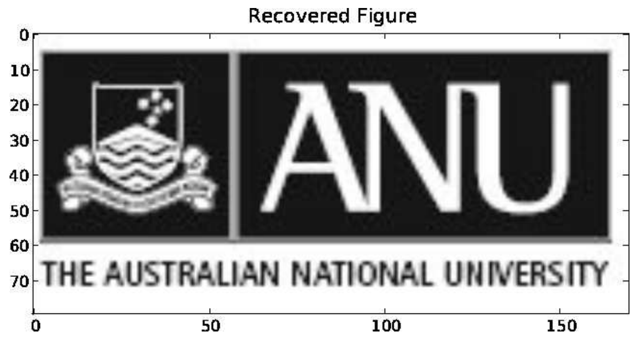


Figure 2.7: decrypted image.

- Thus, Gaussian elimination method is good for dense matrices with not too large size. In case n is huge or the matrices have special structure, it is not competitive.

- In applications, matrices usually have large size and possibly with special structure such as sparsity (i.e. zeros predominate the matrix).

- In this and next sections we will discuss iterative methods which might produce approximate solution within satisfactory accuracy using less number of operations.

Overview

- Consider the linear system
$$A\boldsymbol{x} = \boldsymbol{b}.$$

- Many iterative methods have been developed to solve linear systems.

- One of the basic ideas is to find a matrix M (called the conditioning matrix) such that

    - $A - M$ is small in certain sense;
    - $M\boldsymbol{x} = \boldsymbol{b}$ is easily solved.

- Then the equation $A\boldsymbol{x} = \boldsymbol{b}$ can be written as
$$M\boldsymbol{x} = \boldsymbol{b} - (A - M)\boldsymbol{x}.$$

- In case M is invertible, we have
$$\boldsymbol{x} = M^{-1}\boldsymbol{b} - M^{-1}(A - M)\boldsymbol{x}.$$

- This leads to the fixed point iteration: pick a starting vector $\boldsymbol{x}^{(0)}$ and define
$$\boldsymbol{x}^{(k+1)} = M^{-1}\boldsymbol{b} - M^{-1}(A - M)\boldsymbol{x}^{(k)}.$$
for $k = 0, 1, \cdots$.

- Various iterative methods can be produced by choosing M suitably.

Splitting of Matrices

Let $A = (a_{i,j})$ be an $n \times n$ matrix and write

$$A = L + D + U$$

where

- L is the lower triangle (below the diagonal) of A,
- D is the main diagonal of A,
- U is the upper triangle (above the diagonal) of A;

that is

$$L = \begin{bmatrix} 0 & & & \\ a_{2,1} & 0 & & \\ \vdots & \vdots & \ddots & \\ a_{n,1} & a_{n,2} & \cdots & 0 \end{bmatrix}, \quad D = \begin{bmatrix} a_{1,1} & & & \\ & a_{2,2} & & \\ & & \ddots & \\ & & & a_{n,n} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{1,2} & \cdots & a_{1,n} \\ & 0 & \cdots & a_{2,n} \\ & & \ddots & \vdots \\ & & & 0 \end{bmatrix}.$$

2.5.1 Jacobi Method

Jacobi method

Use the decomposition $A = L + D + U$, we can write $A\boldsymbol{x} = \boldsymbol{b}$ as

$$D\boldsymbol{x} = \boldsymbol{b} - (L + U)\boldsymbol{x}.$$

In case none of the diagonal entries are zero we have

$$\boldsymbol{x} = D^{-1}\boldsymbol{b} - D^{-1}(L + U)\boldsymbol{x}.$$

This motivates the iteration

$$\boldsymbol{x}^{(k+1)} = D^{-1}\boldsymbol{b} - D^{-1}(L + U)\boldsymbol{x}^{(k)}$$

which is the Jacobi method.

- In terms of components, Jacobi method takes the form

$$x_1^{(k+1)} = \Big(b_1 - \sum_{j\neq 1} a_{1,j}x_j^{(k)}\Big)/a_{1,1}$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$
$$x_i^{(k+1)} = \Big(b_i - \sum_{j\neq i} a_{i,j}x_j^{(k)}\Big)/a_{i,i}$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$
$$x_n^{(k+1)} = \Big(b_n - \sum_{j\neq n} a_{n,j}x_j^{(k)}\Big)/a_{n,n}$$

  for $k = 0, 1, \cdots$.

- Note that the computation of every component $x_j^{(k+1)}$ of $\boldsymbol{x}^{(k+1)}$ is independent of other components. Thus Jacobi method can be implemented in a parallel manner.

Example 25 (Jacobi method)**.** *Consider the system $A\boldsymbol{x} = \boldsymbol{b}$ where*

$$A = \begin{bmatrix} 6 & -2 & 2 \\ -2 & 5 & 1 \\ 2 & 1 & 4 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} -1 \\ 8 \\ 8 \end{bmatrix},$$

which has a solution $\boldsymbol{x} = [-0.5, 1, 2]^T$. *Jacobi method for this system is*

$$\begin{aligned}
x_1^{(k+1)} &= \frac{1}{6}\left(-1 + 2x_2^{(k)} - 2x_3^{(k)}\right),\\
x_2^{(k+1)} &= \frac{1}{5}\left(8 + 2x_1^{(k)} - x_3^{(k)}\right),\\
x_3^{(k+1)} &= \frac{1}{4}\left(8 - 2x_1^{(k)} - x_2^{(k)}\right).
\end{aligned}$$

Example 26 (continued)**.** *Starting with* $\boldsymbol{x}^{(0)} = \mathbf{0}$, *we obtain the results tabulated in Table 2.2.*

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	0	0	0
1	-0.166667	1.6	2.0
2	-0.3	1.133333	1.683333
3	-0.35	1.143333	1.866667
4	-0.407778	1.086667	1.889167
5	-0.434167	1.059056	1.932222
10	-0.491339	1.008028	1.990504

Table 2.2: Accuracy of the Jacobi method (5 decimal place accuracy is obtained after 23 iterations)

Algorithm (Jacobi method)

```
input A and b
take an initial guess $x_0$
k = 0   (iteration number)
while a stopping criterion is not satisfied do
   for i = 1:n
      z = 0
      for j = 1:n
         if j $\neq$ i do
            z = z + A(i,j)*$x_0$(j)
         end
      end
      $x$(i) = (b(i)−z)/A(i,i)
   end
   k = k+1
   $x_0$ = $x$
end
output x and k
```

2.5.2 The Gauss-Seidel Method

Gauss-Seidel Method

Use $A = L + D + U$ to write $A\boldsymbol{x} = \boldsymbol{b}$ in the form $(L + D)\boldsymbol{x} = \boldsymbol{b} - U\boldsymbol{x}$, or equivalently

$$\boldsymbol{x} = (L + D)^{-1}(\boldsymbol{b} - U\boldsymbol{x}).$$

This leads to the iterative method

$$\boldsymbol{x}^{(k+1)} = (L + D)^{-1}(\boldsymbol{b} - U\boldsymbol{x}^{(k)})$$

or

$$(L + D)\boldsymbol{x}^{(k+1)} = \boldsymbol{b} - U\boldsymbol{x}^{(k)}, \quad k = 0, 1, ...$$

Consequently

$$\boldsymbol{x}^{(k+1)} = D^{-1}\left(\boldsymbol{b} - L\boldsymbol{x}^{(k+1)} - U\boldsymbol{x}^{(k)}\right)$$

which is the Gauss-Seidel method.

- In terms of components, Gauss-Seidel method takes the form

$$x_i^{(k+1)} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j}x_j^{(k)}\right), \quad i = 1, ..., n$$

  for $k = 0, 1, \cdots$.

- Gauss-Seidel method is closely related to Jacobi method, the only difference being that the improved values $\boldsymbol{x}^{(k+1)}$ are used as soon as they are available.

- Gauss-Seidel method is not a parallel method.

Example 27 (Gauss-Seidel method)**.** *Consider the system in the previous example, Gauss-Seidel method is*

$$\begin{aligned} x_1^{(k+1)} &= (-1 + 2x_2^{(k)} - 2x_3^{(k)})/6, \\ x_2^{(k+1)} &= (8 + 2x_1^{(k+1)} - x_3^{(k)})/5, \\ x_3^{(k+1)} &= (8 - 2x_1^{(k+1)} - x_2^{(k+1)})/4. \end{aligned}$$

Example 28 (continued)**.** *Starting with $\boldsymbol{x}^{(0)} = \boldsymbol{0}$, we obtain the results tabulated in Table 2.3.*

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	0	0	0
1	-0.166667	1.533333	1.7
2	-0.222222	1.171111	1.818333
3	-0.382407	1.083370	1.920361
4	-0.445664	1.037662	1.963416
5	-0.475251	1.017216	1.983322
10	-0.499510	1.000341	1.999670

Table 2.3: Accuracy of the Gauss-Seidel method (5 decimal place accuracy is obtained after 13 iterations)

Algorithm (Gauss-Seidel method)

```
input A and b
take an initial guess x
k = 0   (iteration number)
while a stopping criterion is not satisfied do
   for i = 1:n
      z = 0
      for j = 1:n
         if j ≠ i do
            z = z + A(i,j)*x(j)
         end
      end
      x(i) = (b(i)-z)/A(i,i)
   end
   k = k+1
end
output x and k
```

2.5.3 Norms

- The important question for an iterative method is whether $\boldsymbol{x}^{k+1}$ converges to $\boldsymbol{x}^*$ (the solution of the linear system) and if so how fast.
- We need to introduce a function to measure the closeness of two vectors.
- On real line $\mathbb{R}$, the absolute value function $|x|$ measures the distance of x to the origin.
- On Euclidean spaces $\mathbb{R}^n$, the function
$$\|\boldsymbol{x}\| = \sqrt{x_1^2 + \cdots + x_n^2}$$
measures the distance of the point $\boldsymbol{x}$ to the origin.
- By abstracting the common properties of these two functions, it leads to the concept of norm.

Definition 24 (Vector norm)**.** *A function $\|\cdot\|$ on a vector space V is called a norm if it satisfies the following properties:*

- *Nonnegativity: $\|\boldsymbol{x}\| \geq 0$ for all $\boldsymbol{x} \in V$; $\|\boldsymbol{x}\| = 0$ if and only if $\boldsymbol{x} = \boldsymbol{0}$.*
- *Triangle inequality: $\|\boldsymbol{x}+\boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ for all $\boldsymbol{x}, \boldsymbol{y} \in V$.*
- *Homogeneity: $\|c\boldsymbol{x}\| = |c|\ \|\boldsymbol{x}\|$ for all $c \in \mathbb{R}$ and $\boldsymbol{x} \in V$.*

Given two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, the quantity $\|\boldsymbol{x}-\boldsymbol{y}\|$ is called the distance between $\boldsymbol{x}$ and $\boldsymbol{y}$ measured by the norm $\|\cdot\|$.

Example 29. *For $\boldsymbol{x} = [x_1, x_2, \cdots, x_n]^T$, we define*

$$\begin{aligned} 2-norm: \quad & \|\boldsymbol{x}\|_2 = \left(\sum_{i=1}^n |x_i|^2\right)^{1/2} \\ 1-norm: \quad & \|\boldsymbol{x}\|_1 = \sum_{i=1}^n |x_i| \\ \infty-norm: \quad & \|\boldsymbol{x}\|_\infty = \max\{|x_1|, |x_2|, \cdots, |x_n|\}. \end{aligned}$$

In general, for $1 \leq p \leq \infty$, we define the p-norm by

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

Verification of the p-norms are indeed norms is easy except the triangle inequality which is the famous Minkowski inequality

$$\left(\sum_{i=1}^n |x_i+y_i|^p\right)^{1/p} \leq \left(\sum_{i=1}^n |x_i|^p\right)^{1/p} + \left(\sum_{i=1}^n |y_i|^p\right)^{1/p}$$

Example 30. *Let $\boldsymbol{x} = [1, -1, 0]^T$ and $\boldsymbol{y} = [3, 1, 2]^T$. Then*

$$\begin{array}{lll} \|\boldsymbol{x}\|_1 = 2, & \|\boldsymbol{y}\|_1 = 6, & \|\boldsymbol{x}-\boldsymbol{y}\|_1 = 6, \\ \|\boldsymbol{x}\|_2 = \sqrt{2}, & \|\boldsymbol{y}\|_2 = \sqrt{14}, & \|\boldsymbol{x}-\boldsymbol{y}\|_2 = \sqrt{12}, \\ \|\boldsymbol{x}\|_\infty = 1, & \|\boldsymbol{y}\|_\infty = 3, & \|\boldsymbol{x}-\boldsymbol{y}\|_\infty = 2. \end{array}$$

Matrix norm

Definition 25 (Matrix norm)**.** *Given an $m \times n$ matrix A, we can view it as a mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$:*

$$\boldsymbol{x} \in \mathbb{R}^n \longrightarrow A\boldsymbol{x} \in \mathbb{R}^m.$$

Let $\|\cdot\|$ denote the norms on $\mathbb{R}^m$ and $\mathbb{R}^n$. We define

$$\|A\| = \sup\left\{ \frac{\|A\boldsymbol{x}\|}{\|\boldsymbol{x}\|} : \boldsymbol{x} \in \mathbb{R}^n,\ \boldsymbol{x} \neq \mathbf{0} \right\},$$

that is, $\|A\|$ is the smallest number C such that

$$\|A\boldsymbol{x}\| \leq C\|x\|, \quad \textit{for all } \boldsymbol{x} \in \mathbb{R}^n$$

We call $\|A\|$ the induced norm of A.

Properties of Matrix Norm

- By definition, there holds

$$\|A\boldsymbol{x}\| \leq \|A\|\|\boldsymbol{x}\| \qquad \text{for all } \boldsymbol{x} \in \mathbb{R}^n.$$

- It can be shown that

$$\|A\| = \max_{\|\boldsymbol{x}\|=1} \|A\boldsymbol{x}\|,$$

  i.e. there exists $\boldsymbol{x}^* \in \mathbb{R}^n$ with $\|\boldsymbol{x}^*\| = 1$ such that $\|A\boldsymbol{x}^*\| = \|A\|$.

- For any $m \times n$ matrix A and $n \times p$ matrix B, we have

$$\|AB\boldsymbol{x}\| \leq \|A\|\|B\boldsymbol{x}\| \leq \|A\|\|B\|\|\boldsymbol{x}\| \quad \text{for all } \boldsymbol{x} \in \mathbb{R}^p.$$

  Therefore

$$\|AB\| \leq \|A\|\|B\|.$$

When p-norm of vectors is used, it leads to the p-norm of matrices.

Definition 26. *For $1 \leq p \leq \infty$, we use $\|A\|_p$ to denote the norm of A induced by p-norm of vectors, i.e.*

$$\begin{aligned} \|A\|_p &= \sup\left\{ \frac{\|A\boldsymbol{x}\|_p}{\|\boldsymbol{x}\|_p} : \boldsymbol{x} \in \mathbb{R}^n,\ \boldsymbol{x} \neq 0 \right\} \\ &= \max_{\|\boldsymbol{x}\|_p=1} \|A\boldsymbol{x}\|_p \end{aligned}$$

and call it the p-norm of A.

The following result gives the formula for calculating p-norms of matrix when $p = 1, 2, \infty$.

Theorem 10. *Let $A = (a_{i,j})$ be an $m \times n$ matrix. Then*

$$\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{m} |a_{i,j}|, \quad \|A\|_2 = \sqrt{\lambda_{\max}(A^TA)}, \quad \|A\|_\infty = \max_{1\leq i\leq m} \sum_{j=1}^{n} |a_{i,j}|,$$

where $\lambda_{\max}(A^TA)$ denotes the largest eigenvalue of A^TA.

Therefore

- if $\boldsymbol{c}_1, \cdots, \boldsymbol{c}_n$ denote the columns of A, then

$$\|A\|_1 = \max\{\|\boldsymbol{c}_1\|_1, \cdots, \|\boldsymbol{c}_n\|_1\};$$

- if $\boldsymbol{r}_1, \cdots, \boldsymbol{r}_m$ denote the rows of A, then

$$\|A\|_\infty = \max\{\|\boldsymbol{r}_1\|_1, \cdots, \|\boldsymbol{r}_m\|_1\};$$

Example 31. *Consider the matrix*

$$A = \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}.$$

We have $\|A\|_1 = 3$ *and* $\|A\|_\infty = 2$*. To determine* $\|A\|_2$*, we note that*

$$A^T A = \begin{bmatrix} 1 & 0 \\ -1 & 2 \end{bmatrix}\begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ -1 & 5 \end{bmatrix}$$

Solving the characteristic equation

$$0 = \det(\lambda I - A^T A) = \lambda^2 - 6\lambda + 4$$

We obtain the two eigenvalues of $A^T A$*:*

$$\lambda_1 = 3 + \sqrt{5}, \qquad \lambda_2 = 3 - \sqrt{5}.$$

Therefore $\|A\|_2 = \sqrt{3 + \sqrt{5}} \approx 2.2882$.

Spectral Radius

Definition 27. *The spectral radius* $\rho(A)$ *of a matrix* A *is defined by*

$$\rho(A) = \max_{1 \le j \le n} |\lambda_j|$$

where $\lambda_1, \cdots, \lambda_n$ *denote the eigenvalues of* A*.*

- There exist matrices B with $\rho(B) = 0$ and $\|B\| > 0$ for any norm $\|\cdot\|$. For instance, consider $B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$.
- $\rho(A) \le \|A\|$ for any induced matrix norm. Indeed, let λ_j be the eigenvalue of A with $|\lambda_j| = \rho(A)$ and let $\boldsymbol{x}$ be a corresponding eigenvector, i.e. $A\boldsymbol{x} = \lambda_j \boldsymbol{x}$. Then

$$|\lambda_j|\|\boldsymbol{x}\| = \|A\boldsymbol{x}\| \le \|A\|\|\boldsymbol{x}\| \Longrightarrow \rho(A) = |\lambda_j| \le \|A\|.$$

2.5.4 Convergence Analysis

Error Reduction

We turn to the convergence analysis of the general iterative method

$$\boldsymbol{x}^{(k+1)} = M^{-1}\boldsymbol{b} - M^{-1}(A - M)\boldsymbol{x}^{(k)}. \tag{2.3}$$

Let $\boldsymbol{x}^*$ be the solution, i.e. $A\boldsymbol{x}^* = \boldsymbol{b}$. Then

$$\begin{aligned} \boldsymbol{x}^{(k+1)} - \boldsymbol{x}^* &= M^{-1}A\boldsymbol{x}^* + (I - M^{-1}A)\boldsymbol{x}^{(k)} - \boldsymbol{x}^* \\ &= (I - M^{-1}A)(\boldsymbol{x}^{(k)} - \boldsymbol{x}^*). \end{aligned}$$

Let $\boldsymbol{e}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x}^*$ denote the error vector and let $E = I - M^{-1}A$. Then

$$\boldsymbol{e}^{(k+1)} = E\boldsymbol{e}^{(k)}.$$

We will call E the error matrix associated with the method. By induction we obtain

$$\boldsymbol{e}^{(k)} = E^k\boldsymbol{e}^{(0)}, \qquad k = 0, 1, \cdots .$$

Therefore

$$\|\boldsymbol{e}^{(k)}\| = \|E^k\boldsymbol{e}^{(0)}\| \leq \|E^k\|\|\boldsymbol{e}^{(0)}\| \leq \|E\|^k\|\boldsymbol{e}^{(0)}\|.$$

where $\|E\|$ can be any induced norm on E. This shows the following result.

Theorem 11. *If $\|E\| = \rho < 1$, then for any initial guess $\boldsymbol{x}^{(0)}$ the sequence $\{\boldsymbol{x}^{(k)}\}$ defined by (2.3) converges to the solution $\boldsymbol{x}^*$ of $A\boldsymbol{x} = \boldsymbol{b}$ and*

$$\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^*\| \leq \rho^k\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\|.$$

In terms of spectral radius, the above theorem can be strengthened to the following version.

Theorem 12 (Convergence using Spectral Radius)**.** *$E^k\boldsymbol{e}^{(0)} \to \mathbf{0}$ for every $\boldsymbol{e}^{(0)}$ as $k \to \infty$ if and only if $\rho(E) < 1$. Therefore, the method (2.3) with any initial guess $\boldsymbol{x}^{(0)}$ is convergent if and only if $\rho(E) < 1$.*

Application to Jacobi and Gauss-Seidel Methods

- Jacobi and Gauss-Seidel methods take the form (2.3) with

$$M_J = D \qquad \text{and} \qquad M_{GS} = D + L$$

respectively.

- Thus, the error matrices associated with them are

$$E_J = I - D^{-1}A = -D^{-1}(L+U),$$
$$E_{GS} = I - (D+L)^{-1}A = -(D+L)^{-1}U.$$

- If A is singular, then there is $\boldsymbol{x} \neq 0$ with $A\boldsymbol{x} = 0$. Thus

$$E_J\boldsymbol{x} = \boldsymbol{x} \quad \text{and} \quad E_{GS}\boldsymbol{x} = \boldsymbol{x}.$$

i.e. E_J and E_{GS} have an eigenvalue equal to 1. So $\rho(E_J) \geq 1$ and $\rho(E_{GS}) \geq 1$. Thus, for some initial guess, Jacobi and Gauss-Seidel methods may not converge in case A is singular.

- Conditions on A should be imposed to guarantee that $\rho(E_J) < 1$ and $\rho(E_{GS}) < 1$.

- Consider $\rho(E_J)$ first. Recall that $\rho(E_J) \leq \|E_J\|$ for any induced matrix norm. In particular

$$\rho(E_J) \leq \|E\|_\infty = \|D^{-1}(L+U)\|_\infty = \max_{1\leq i\leq n} \frac{1}{|a_{i,i}|}\sum_{\substack{j=1\\j\neq i}}^{n} |a_{i,j}|.$$

- Recall also that similarity transform does not change the eigenvalues and hence the spectral radius. Thus

$$\rho(E_J) = \rho(DE_JD^{-1}) = \rho((L+U)D^{-1}) \leq \|(L+U)D^{-1}\|_1$$
$$= \max_{1\leq j\leq n} \frac{1}{|a_{j,j}|}\sum_{\substack{i=1\\i\neq j}}^{n} |a_{i,j}|.$$

The above discussion motivates the following definition.

Definition 28 (Diagonal Dominance)**.** *An $n \times n$ matrix A is diagonally dominated if either*

$$\sum_{\substack{j=1\\ j\neq i}}^{n} |a_{i,j}| < |a_{i,i}|, \qquad i = 1, ..., n$$

or

$$\sum_{\substack{i=1\\ i\neq j}}^{n} |a_{i,j}| < |a_{j,j}|, \qquad j = 1, ..., n.$$

The above argument has showed that if A is diagonally dominated then $\rho(E_J) < 1$. Therefore, we have

Theorem 13 (Convergence of Jacobi method)**.** *If A is diagonally dominated, then Jacobi method converges for any initial guess.*

Next we consider $\rho(E_{GS})$ for Gauss-Seidel method. The analysis is harder. However, we can show that if A is diagonally dominated, then

$$\rho(E_{GS}) < 1.$$

Thus we can conclude the following result.

Theorem 14 (Convergence of Gauss-Seidel method)**.** *If A is diagonally dominated, then Gauss-Seidel method converges for any initial guess.*

2.5.5 Relaxation Methods

- Jacobi and Gauss-Seidel methods can converge very slowly.
- It is of practical importance to modify them to improve the speed of convergence.
- We will use the relaxation strategy to achieve the goal.

Jacobi Method with Relaxation

- We first consider Jacobi method

$$\boldsymbol{x}^{(k+1)} = D^{-1}\boldsymbol{b} - D^{-1}(L+U)\boldsymbol{x}^{(k)}.$$

- Using $L + U = A - D$ we can write

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + D^{-1}\left(\boldsymbol{b} - A\boldsymbol{x}^{(k)}\right).$$

  That is, $\boldsymbol{x}^{(k+1)}$ is computed from the previous iterate $\boldsymbol{x}^{(k)}$ by adding the correction term $D^{-1}(\boldsymbol{b} - A\boldsymbol{x}^{(k)})$.

- It is natural to modify it by adding into $\boldsymbol{x}^{(k)}$ a suitable multiple of $D^{-1}(\boldsymbol{b} - A\boldsymbol{x}^{(k)})$. This leads to the method

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \omega D^{-1}(\boldsymbol{b} - A\boldsymbol{x}^{(k)})$$

  which is called the Jacobi method with relaxation, or the simultaneous relaxation method, where $\omega \in \mathbb{R}$ is a relaxation parameter.

- Let $\boldsymbol{x}^*$ be the solution of $A\boldsymbol{x} = \boldsymbol{b}$. Then
$$\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^* = (I - \omega D^{-1}A)(\boldsymbol{x}^{(k)} - \boldsymbol{x}^*).$$

- Thus, the corresponding error matrix is
$$\begin{aligned} E_J(\omega) &= (I - \omega D^{-1}A) = I - \omega D^{-1}(D + L + U) \\ &= (1-\omega)I - \omega D^{-1}(L+U) \\ &= (1-\omega)I + \omega E_J, \end{aligned}$$
where $E_J = -D^{-1}(L+U)$ is the error matrix of Jacobi method.

- Our goal is to determine for which ω the method is convergent and for which ω the spectral radius $\rho(E_J(\omega))$ is minimal.

Theorem 15.
- *If the Jacobi method converges, then so does the Jacobi method with relaxation for all $0 < \omega \leq 1$.*
- *If E_J has only real eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ lying in the interval $(-1, 1)$, then $\rho(E_J(\omega))$ is minimized at*
$$\omega_{\min} = \frac{2}{2 - \lambda_1 - \lambda_n}$$
and
$$\rho(E_J(\omega_{\min})) = \frac{\lambda_n - \lambda_1}{2 - \lambda_n - \lambda_1}.$$
If $\lambda_1 \neq -\lambda_n$, we have
$$\rho(E_J(\omega_{\min})) < \rho(E_J).$$

Successive Over-Relaxation

- We next consider the Gauss-Seidel method
$$\boldsymbol{x}^{(k+1)} = D^{-1}\left(\boldsymbol{b} - L\boldsymbol{x}^{(k+1)} - U\boldsymbol{x}^{(k)}\right).$$

- We can rewrite as
$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + D^{-1}\left(\boldsymbol{b} - L\boldsymbol{x}^{(k+1)} - (U+D)\boldsymbol{x}^{(k)}\right).$$

- By introducing a relaxation parameter $\omega \neq 0$, it motivates to define $\boldsymbol{x}^{(k+1)}$ from $\boldsymbol{x}^{(k)}$ by
$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \omega D^{-1}\left(\boldsymbol{b} - L\boldsymbol{x}^{(k+1)} - (U+D)\boldsymbol{x}^{(k)}\right).$$

- This iterative method is called the successive over-relaxation (SOR) method. For $\omega = 1$ it reduces to the Gauss-Seidel method. The goal is to choose ω so that the rate of convergence is maximised.

- Let
$$\boldsymbol{x}_{GS}^{(k)} = D^{-1}\left(\boldsymbol{b} - U\boldsymbol{x}^{(k)} - L\boldsymbol{x}^{(k+1)}\right)$$
be the Gauss-Seidel step. We can write the SOR method as
$$\boldsymbol{x}^{(k+1)} = (1-\omega)\boldsymbol{x}^{(k)} + \omega \boldsymbol{x}_{GS}^{(k)}.$$

Algorithm (Successive over-relaxation method)

```
input A, b and $\omega$
take an initial guess x
k = 0   (iteration number)
while a stopping criterion is not satisfied do
   for i = 1:n
      z = 0
      for j = 1:n
         if j $\neq$ i do
            z = z + A(i,j)*x(j)
         end
      end
      x(i) = (1-$\omega$)*x(i) + $\omega$*(b(i)-z)/A(i,i)
   end
   k = k+1
end
output x and k
```

Example 32 (SOR)**.** *Consider again the system $A\boldsymbol{x} = \boldsymbol{b}$, where*

$$A = \begin{bmatrix} 6 & -2 & 2 \\ -2 & 5 & 1 \\ 2 & 1 & 4 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} -1 \\ 8 \\ 8 \end{bmatrix},$$

which has the solution $\boldsymbol{x} = [-0.5, 1, 2]^T$. The SOR method is

$$\begin{aligned} x_1^{(k+1)} &= (1-\omega)x_1^{(k)} + \frac{\omega}{6}\left(-1 + 2x_2^{(k)} - 2x_3^{(k)}\right), \\ x_2^{(k+1)} &= (1-\omega)x_2^{(k)} + \frac{\omega}{5}\left(8 + 2x_1^{(k+1)} - x_3^{(k)}\right), \\ x_3^{(k+1)} &= (1-\omega)x_3^{(k)} + \frac{\omega}{4}\left(8 - 2x_1^{(k+1)} - x_2^{(k+1)}\right). \end{aligned}$$

Example 33 (continued)**.** *With $\omega = 1.15$ the results of an SOR calculation are presented in Table 2.4.*

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	0	0	0
1	-0.191667	1.751833	1.906446
2	-0.222227	1.036493	1.843806
3	-0.467803	1.045262	1.991903
4	-0.484375	1.002260	1.915800
5	-0.498250	1.002404	1.999566
10	-0.499998	1.000000	1.999999

Table 2.4: Accuracy of the SOR method (5 decimal place accuracy is obtained after 8 iterations)

2.5.6 Convergence of SOR

- We next derive the error matrix of the SOR method.
- We have

$$(\omega L + D)\boldsymbol{x}^{(k+1)} = \omega\boldsymbol{b} + D\boldsymbol{x}^{(k)} - \omega(D+U)\boldsymbol{x}^{(k)}.$$

Therefore, by using $U + D = A - L$,

$$\begin{aligned} \boldsymbol{x}^{(k+1)} &= (\omega L + D)^{-1}\left(\omega \boldsymbol{b} + D\boldsymbol{x}^{(k)} - \omega(D+U)\boldsymbol{x}^{(k)}\right) \\ &= (\omega L + D)^{-1}\left(\omega \boldsymbol{b} - \omega A\boldsymbol{x}^{(k)} + (\omega L + D)\boldsymbol{x}^{(k)}\right) \\ &= \boldsymbol{x}^{(k)} - \omega(\omega L + D)^{-1}(A\boldsymbol{x}^{(k)} - \boldsymbol{b}) \end{aligned}$$

- Let $\boldsymbol{x}^*$ be the solution of $A\boldsymbol{x} = \boldsymbol{b}$. Then

$$\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^* = \left(I - \omega(\omega L + D)^{-1}A\right)(\boldsymbol{x}^{(k)} - \boldsymbol{x}^*)$$

- Hence the error matrix of the SOR method is

$$\begin{aligned} E_{GS}(\omega) &= I - \omega(\omega L + D)^{-1}A = (\omega L + D)^{-1}(\omega L + D - \omega A) \\ &= (\omega L + D)^{-1}\left((1-\omega)D - \omega U\right). \end{aligned}$$

- Thus, the SOR method is convergent if and only if $\rho(E_{GS}(\omega)) < 1$.
- In the error matrix
$$E_{GS}(\omega) = (\omega L + D)^{-1}((1-\omega)D - \omega U),$$
the first factor is lower triangular with $1/a_{1,1}, \cdots, 1/a_{n,n}$ on the main diagonal, and the second factor is upper triangular with $(1-\omega)a_{1,1}, \cdots, (1-\omega)a_{n,n}$ on the main diagonal. Thus
$$\det(E_{GS}(\omega)) = \left(\prod_{i=1}^{n} \frac{1}{a_{i,i}}\right)\left(\prod_{i=1}^{n}(1-\omega)a_{i,i}\right) = (1-\omega)^n.$$
- Let $\lambda_1, \cdots, \lambda_n$ be eigenvalues of $E_{GS}(\omega)$. Then
$$\lambda_1 \cdots \lambda_n = \det(E_{GS}(\omega)) = (1-\omega)^n.$$
- Thus
$$\rho(E_{GS}(\omega))^n \geq |\lambda_1| \cdots |\lambda_n| = |1-\omega|^n.$$
which implies
$$\rho(E_{GS}(\omega)) \geq |\omega - 1|.$$
- The convergence of SOR method forces $\rho(E_{GS}(\omega)) < 1$ and hence forces $|\omega - 1| < 1$.

Lemma 1. *In order for the SOR method to be convergent, the relaxation parameter ω must satisfy $|\omega - 1| < 1$, i.e. $0 < \omega < 2$.*

When A has special structure, the converse of above result also holds.

Theorem 16. *If A is symmetric and positive definite, then the SOR method converges for any initial guess if and only if $0 < \omega < 2$.*

- A right choice of the relaxation parameter ω can improve the speed of convergence considerably.
- However, the calculation of the optimal relaxation parameter, i.e the parameter minimising the spectral radius, is difficult except in some simple case.
- Usually it is obtained only approximately by trial and error, based on trying several values of ω and observing the effect on the speed of convergence.

We conclude this subsection by a comparison of the convergence speed on the Jacobi method, Gauss-Seidel method and the SOR method tested on the linear system $A\boldsymbol{x} = \boldsymbol{b}$ of size $n = 200$, where

$$A = \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ n \end{bmatrix}.$$

For the SOR method we take the relaxation parameter $\omega = 1.9$. The following figure plots the results on the relative error versus the iteration numbers. It clearly demonstrates the advantage of SOR method over Gauss-Seidel method, and Gauss-Seidel method over Jacobi method.

2.5.7 Iterative Refinement

Iterative Improvement and Gaussian Elimination

- When solving a linear system $A\boldsymbol{x} = \boldsymbol{b}$ by a direct method such as Gaussian elimination, due to the presence of rounding errors, the computed solution may sometimes deviate from the exact solution.
- Iterative refinement is an iterative method to improve the accuracy of numerical solutions to linear systems.
- Assume the basic solution method is Gaussian elimination. It provides a factorisation LU that is the exact factorisation of a matrix close to A.
- Then the iterative refinement method takes the following form:
  1. Take $\boldsymbol{x}^{(0)}$ to be the solution obtained by Gaussian elimination.
  2. Given an approximation $\boldsymbol{x}^{(k)}$, compute $\boldsymbol{b} - A\boldsymbol{x}^{(k)}$ using *double precision* and round to single precision to obtain the residual $\boldsymbol{r}^{(k)}$.
  3. Find $\boldsymbol{e}^{(k)}$ of $LU\boldsymbol{e}^{(k)} = \boldsymbol{r}^{(k)}$ using back and forward substitutions.
  4. Let $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{e}^{(k)}$.
  5. Continue until the difference between $\boldsymbol{x}^{(k+1)}$ and $\boldsymbol{x}^{(k)}$ is within a given tolerance.

Example 34. *Consider the system*

$$\begin{bmatrix} 0.20000 & 0.16667 & 0.14286 \\ 0.16667 & 0.14286 & 0.12500 \\ 0.14286 & 0.12500 & 0.11111 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0.50953 \\ 0.43453 \\ 0.37897 \end{bmatrix}.$$

The exact solution is $\boldsymbol{x} = (1, 1, 1)^T$. *If floating point arithmetic with* 5 *digits is used then Gaussian elimination will give the computed triangular factors*

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.83335 & 1 & 0 \\ 0.71430 & 1.49874 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 0.20000 & 0.16667 & 0.14286 \\ 0 & 0.00397 & 0.00595 \\ 0 & 0 & 0.00015 \end{bmatrix}$$

and computed solution

$$\boldsymbol{x}^{(0)} = (1.0345, 0.89673, 1.06667)^T.$$

The first step of iterative refinement involves calculating the residual $\boldsymbol{r}^{(0)} = \boldsymbol{b} - A\boldsymbol{x}^{(0)}$ in double precision arithmetic (in this case 10 digits).

$$A\boldsymbol{x}^{(0)} = \begin{bmatrix} 0.5095324653 \\ 0.4345190593 \\ 0.3789619207 \end{bmatrix} \quad \text{and so} \quad \boldsymbol{r}^{(0)} = 10^{-5} \begin{bmatrix} -0.24653 \\ 1.09407 \\ 0.80793 \end{bmatrix}.$$

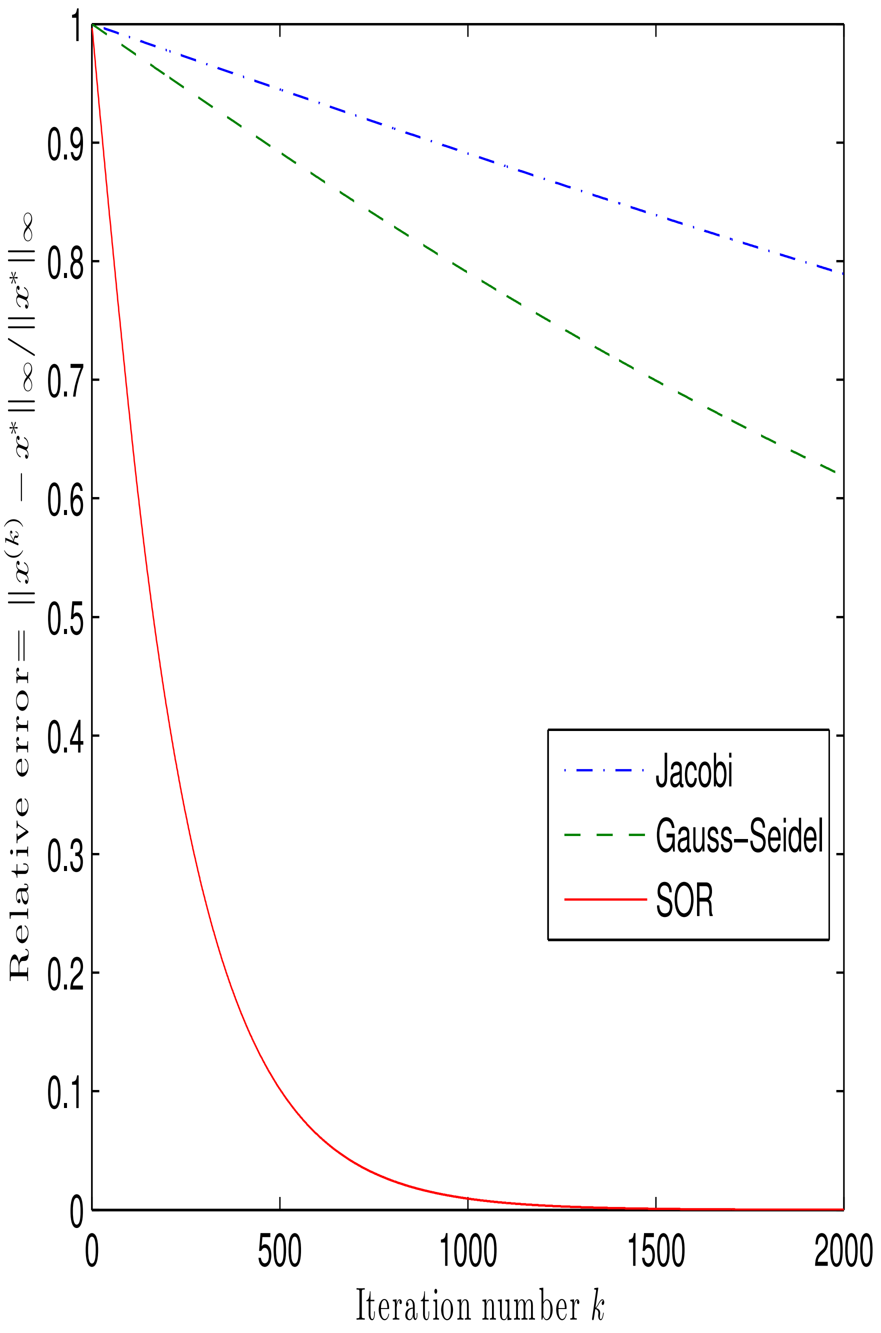


Figure 2.8: Comparison of convergence speed for Jacobi method, Gauss-Seidel method and the SOR method with $\omega = 1.9$

Then we must solve for $\boldsymbol{e}^{(0)}$ using backward and forward substitution. We get

$$\boldsymbol{e}^{(0)} = \begin{bmatrix} -0.03709 \\ 0.09955 \\ -0.06424 \end{bmatrix}, \qquad \boldsymbol{x}^{(1)} = \boldsymbol{x}^{(0)} + \boldsymbol{e}^{(0)} = \begin{bmatrix} 1.00136 \\ 0.99628 \\ 1.00243 \end{bmatrix}.$$

Note that the error in the corrected solution $\boldsymbol{x}^{(1)}$ is approximately 30 times smaller than those in $\boldsymbol{x}^{(0)}$.

If we continue the iterations then the approximate solutions $\boldsymbol{x}^{(k)}$ converge rapidly to the exact solution. This is clearly illustrated in Table 2.5.

k	$\boldsymbol{x}_1^{(k)}$	$\boldsymbol{x}_2^{(k)}$	$\boldsymbol{x}_3^{(k)}$
0	1.03845	0.89673	1.06667
1	1.00136	0.99628	1.00243
2	1.00005	0.99986	1.00009
3	1.00000	1.00000	1.00000

Table 2.5: Improved accuracy using Iterative Improvement.

2.6 The Conjugate Gradient Method

2.6.1 Introduction

- In this section we consider the linear system

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

where A is symmetric and positive definite. We will develop other iterative solvers.

- Our linear system can be rephrased into a minimisation problem. Indeed, consider the quadratic function

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x},$$

then we have the following theorem:

Theorem 17. *Assume that A is symmetric and positive definite. Then $\boldsymbol{x}^*$ is a solution of $A\boldsymbol{x} = \boldsymbol{b}$ if and only if*

$$f(\boldsymbol{x}^*) = \min_{\boldsymbol{x}\in\mathbb{R}^n} f(\boldsymbol{x}).$$

Proof. Let $\boldsymbol{x}$ and $\boldsymbol{x}^*$ be arbitrary vectors. Rewrite $\boldsymbol{x} = \boldsymbol{x}^* + \boldsymbol{z}$. Then

$$\begin{aligned} f(\boldsymbol{x}) &= f(\boldsymbol{x}^* + \boldsymbol{z}) \\ &= f(\boldsymbol{x}^*) + \boldsymbol{z}^T(A\boldsymbol{x}^* - \boldsymbol{b}) + \frac{1}{2}\boldsymbol{z}^T A\boldsymbol{z}. \end{aligned}$$

Here we have used the assumed symmetry of A. If $A\boldsymbol{x}^* = \boldsymbol{b}$, then

$$f(\boldsymbol{x}) = f(\boldsymbol{x}^*) + \boldsymbol{z}^T A\boldsymbol{z}$$

and so, by the positive definiteness of A, we have

$$f(\boldsymbol{x}) \geq f(\boldsymbol{x}^*) \quad \text{for all } \boldsymbol{x}.$$

On the other hand, if $f(\boldsymbol{x}) \geq f(\boldsymbol{x}^*)$ for all $\boldsymbol{x}$, then the term $\boldsymbol{z}^T(A\boldsymbol{x}^* - \boldsymbol{b})$ must be non-negative for all $\boldsymbol{z}$. This is possible only if $A\boldsymbol{x}^* = \boldsymbol{b}$. □

- We will use this equivalence to generate iterative schemes to solve linear equations associated with positive definite matrices.
- The iterative schemes we will consider are of the form
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k \tag{2.4}$$
where $\boldsymbol{d}_k$ is called the search direction, and $\alpha_k > 0$ is the step size.
- Search directions and step sizes are chosen to ensure that the values $f(\boldsymbol{x}^k)$ decrease to the minimum value, and hence $\boldsymbol{x}^k \to \boldsymbol{x}^*$, the solution of $A\boldsymbol{x} = \boldsymbol{b}$.
- We will consider two particular methods, the gradient method and the conjugate gradient method.

Gradient method and steepest descent

- For a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, its *gradient* ∇f is defined by
$$\nabla f(x) = \begin{bmatrix} \partial_1 f \\ \vdots \\ \partial_n f \end{bmatrix}.$$
It is known that, for any vector $\boldsymbol{d}$,
$$\frac{d}{d\alpha} f(\boldsymbol{x} + \alpha \boldsymbol{d}) = \boldsymbol{d}^T \nabla f(\boldsymbol{x} + \alpha \boldsymbol{d}).$$
- Suppose we have an iterative scheme in which $\boldsymbol{x}^{k+1}$ is given by (2.4), i.e. $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$. Then an application of Taylor's formula gives
$$f(\boldsymbol{x}_{k+1}) = f(\boldsymbol{x}_k) + \alpha_k \boldsymbol{d}_k^T \nabla f(\boldsymbol{x}_k) + o(\alpha_k) \text{ as } \alpha_k \to 0.$$

2.6.2 Gradient Method: Steepest Descent

- Suppose that $\nabla f(\boldsymbol{x}_k) \neq \boldsymbol{0}$. We can then ensure $f(\boldsymbol{x}_{k+1}) < f(\boldsymbol{x}_k)$ (at least for small α_k) if
$$\boldsymbol{d}_k^T \nabla f(\boldsymbol{x}_k) < 0. \tag{2.5}$$
- We say that $\boldsymbol{d}_k$ is a *descent direction* if (2.5) holds.
- Various different methods can be developed by the choices of descent directions $\boldsymbol{d}_k$ and step sizes α_k.
- Clearly $\boldsymbol{d}_k := -\nabla f(x_k)$ is a descent direction at $\boldsymbol{x}_k$ because
$$\boldsymbol{d}_k^T \nabla f(\boldsymbol{x}_k) = -\|\nabla f(\boldsymbol{x}_k)\|_2^2 < 0.$$
With such choices of $\boldsymbol{d}_k$, it leads to the gradient method
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k), \quad k = 0, 1, \cdots$$
with suitable choices of α_k.
- If the step size α_k is chosen optimally, in the sense that
$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) = \min_{\alpha > 0} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k), \tag{2.6}$$
the corresponding gradient method is called the method of steepest descent. It is easy to see that α_k must then satisfy
$$\left.\frac{d}{d\alpha} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)\right|_{\alpha = \alpha_k} = 0.$$
This is equivalent to
$$\boldsymbol{d}_k^T \nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) = 0.$$

2.6.3 Example

- We return to the quadratic function
$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}.$$
Its gradient is given by $\nabla f(\boldsymbol{x}) = A\boldsymbol{x} - \boldsymbol{b}$.
- Thus, the corresponding gradient method for solving $A\boldsymbol{x} = \boldsymbol{b}$ becomes
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k\boldsymbol{d}_k, \quad \text{where } \boldsymbol{d}_k = \boldsymbol{b} - A\boldsymbol{x}_k.$$
- If α_k is chosen optimally in the sense of (2.6), we have
$$\begin{aligned} 0 &= \boldsymbol{d}_k^T\nabla f(\boldsymbol{x}_k + \alpha_k\boldsymbol{d}_k) = \boldsymbol{d}_k^T\left(A(\boldsymbol{x}_k + \alpha_k\boldsymbol{d}_k) - \boldsymbol{b}\right) \\ &= \boldsymbol{d}_k^T(A\boldsymbol{x}_k - \boldsymbol{b}) + \alpha_k\boldsymbol{d}_k^T A\boldsymbol{d}_k \\ &= -\boldsymbol{d}_k^T\boldsymbol{d}_k + \alpha_k\boldsymbol{d}_k^T A\boldsymbol{d}_k \end{aligned}$$
which gives
$$\alpha_k = \frac{\boldsymbol{d}_k^T\boldsymbol{d}_k}{\boldsymbol{d}_k A\boldsymbol{d}_k}$$

Summarizing the above, we obtain the following algorithm for solving $A\boldsymbol{x} = \boldsymbol{b}$ with symmetrix positive definite matrix A.

Algorithm: The method of steepest descent[1.0ex]

```
• Pick a starting point $\boldsymbol{x}_0 \in \mathbb{R}^n$ and set $k := 0$;
• For $k = 0, 1, \cdots$ do
      $\boldsymbol{d}_k = \boldsymbol{b} - A\boldsymbol{x}_k$;
      $\alpha_k = \frac{\boldsymbol{d}_k^T\boldsymbol{d}_k}{\boldsymbol{d}_k^T A\boldsymbol{d}_k}$;
      $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k\boldsymbol{d}_k$;
      $k = k + 1$;
• End
```

Example 35 (The method of steepest descent)**.** *Consider the system $A\boldsymbol{x} = \boldsymbol{b}$ where*
$$A = \begin{bmatrix} 6 & -2 & 2 \\ -2 & 5 & 1 \\ 2 & 1 & 4 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} -1 \\ 8 \\ 8 \end{bmatrix},$$
which has a solution $\boldsymbol{x} = [-0.5, 1, 2]^T$. The table gives the computational result by the method of steepest descent with $\boldsymbol{x}_0 = \boldsymbol{0}$:

- The above example shows that the method of steepest descent, like the Jacobi method, is a slow convergent method.
- By further investigation, one can show that
$$(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k)^T(\boldsymbol{x}_k - \boldsymbol{x}_{k-1}) = 0 \quad \text{for } k = 1, 2, \cdots$$
for the sequence $\{\boldsymbol{x}_k\}$ defined by the method of steepest descent. This means that $\{\boldsymbol{x}_k\}$ moves to the exact solution in a zig-zag way which makes the method slow.

k	$\boldsymbol{x}_k$		
0	0	0	0
1	-0.181690	1.453521	1.453521
2	-0.158173	1.170584	1.739398
3	-0.368425	1.186741	1.772684
4	-0.358339	1.071113	1.892521
5	-0.445509	1.077346	1.905873
10	-0.489939	1.005050	1.992366

An important question is how fast does this method converge. For a positive definite matrix A, the condition number (with respect to the L_2 norm) is given by

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

where $\lambda_{max}(A)$ and $\lambda_{min}(A)$ are the maximum and minimum eigenvalues of A. It is worth noting that these eigenvalues satisfy

$$\lambda_{max}(A) = \max_{\boldsymbol{y}\in\mathbb{R}^m} \frac{\boldsymbol{y}\cdot A\boldsymbol{y}}{\|\boldsymbol{y}\|_2} \qquad \lambda_{min}(A) = \min_{\boldsymbol{y}\in\mathbb{R}^m} \frac{\boldsymbol{y}\cdot A\boldsymbol{y}}{\|\boldsymbol{y}\|_2}$$

The second ratio is known as the Rayleigh Quotient. Also recall that if B is a symmetric matrix with eigenvalues $\lambda_1(B), \ldots, \lambda_n(B)$ then the L_2 operator norm of B satisfies

$$\|B\|_2 = \max_j |\lambda_j(B)|.$$

To obtain an idea of how the gradient method converges, let us consider a steepest descent method with a constant step length. That is, let us consider the iterative method where $\alpha_k = \alpha$, a constant, and

$$\boldsymbol{d}^{k+1} = \boldsymbol{r}^k = \boldsymbol{b} - A\boldsymbol{x}^k.$$

Then

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \alpha(\boldsymbol{b} - A\boldsymbol{x}^k). \tag{2.7}$$

Now the exact solution $\boldsymbol{x}$ must also satisfy

$$\boldsymbol{x} = \boldsymbol{x} + \alpha(\boldsymbol{b} - A\boldsymbol{x}). \tag{2.8}$$

Subtracting Equations (2.7) and (2.8) leads to the equation

$$\boldsymbol{x}^{k+1} - \boldsymbol{x} = (I - \alpha A)(\boldsymbol{x}^k - \boldsymbol{x}).$$

If we let $\boldsymbol{e}^k$ denote the error at the kth step, then we have $\boldsymbol{e}^{k+1} = E\boldsymbol{e}^k$ where the error matrix (see the Section 2.5) is given by $E = I - \alpha A$. For convergence we need $\|I - \alpha A\|_2 < 1$. Consequently, we need $\max_j |1 - \alpha\lambda_j(A)| < 1$. Since all the eigenvalues are positive, we conclude that this implies that

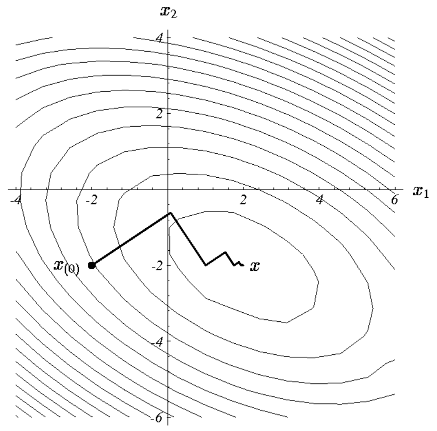
$$\alpha < \frac{2}{\lambda_{\max}(A)}.$$

Let us choose $\alpha = 1/\lambda_{\max}(A)$. It turns out that this choice is close to the best choice. Then we have

$$\|I - \alpha A\|_2 = 1 - \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)} = 1 - \frac{1}{\kappa_2(A)}.$$

This shows that as the condition number increases, the convergence rate decreases. In fact we see that the number of iterations required to obtain a specified accuracy will be proportional to the condition number.

Figure 2.9 shows an example calculation based on the steepest descent method. Observe that all of the steps only go in one of two directions. What if we could combine the steps so we only took one big step in each direction? Then the solution would be found in two steps. The Conjugate gradient method tries to achieve that goal.



The method of Steepest Descent.

Figure 2.9: Possible zig-zag behaviour of steepest descent

2.6.4 Conjugate Gradient Method

Conjugate Gradient Method

- The method of steepest descent has a tendency to zig-zag into the solution. This can make the method convergent slowly.
- We next consider the conjugate gradient method which is a fast algorithm; it was proposed by Hestens and Stiefel in 1950.
- We start with the description of a general class of iterative methods — conjugate direction methods.

Definition 29. *A set of nonzero vectors $\{\boldsymbol{d}_0, \boldsymbol{d}_1, \cdots, \boldsymbol{d}_{n-1}\}$ in $\mathbb{R}^n$ is called conjugate with respect to A (or A-conjugate) if*

$$\boldsymbol{d}_i^T A \boldsymbol{d}_j = 0 \quad \text{for all } i \neq j.$$

One can check that any nonzero A-conjugate vectors $\{\boldsymbol{d}_0, \cdots, \boldsymbol{d}_{n-1}\}$ are linearly independent and hence form a basis of $\mathbb{R}^n$.

- Given a set of nonzero A-conjugate vectors $\{\boldsymbol{d}_0, \cdots, \boldsymbol{d}_{n-1}\}$ in $\mathbb{R}^n$. A conjugate direction method is to find the minimizer of

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A \boldsymbol{x} - \boldsymbol{b}^T \boldsymbol{x}$$

  in n steps by successively minimizing $f(\boldsymbol{x})$ along the individual directions $\boldsymbol{d}_i$.

- To be more precise, let $\boldsymbol{x}_0 \in \mathbb{R}^n$ be a starting point, it generates a sequence $\{\boldsymbol{x}_k\}$ by setting

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k,$$

  where α_k is chosen by

$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) = \min_{\alpha \in \mathbb{R}} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k).$$

- α_k can be determined by solving the equation

$$\frac{d}{d\alpha} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) = 0.$$

- Note that

$$\begin{aligned} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) &= \frac{1}{2}(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)^T A (\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) - \boldsymbol{b}^T(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) \\ &= \frac{1}{2}\alpha^2 \boldsymbol{d}_k^T A \boldsymbol{d}_k + \alpha \boldsymbol{d}_k^T A \boldsymbol{x}_k + \frac{1}{2}\boldsymbol{x}_k^T A \boldsymbol{x}_k - \boldsymbol{b}^T \boldsymbol{x}_k - \alpha \boldsymbol{d}_k^T \boldsymbol{b} \end{aligned}$$

  Therefore

$$\begin{aligned} \frac{d}{d\alpha} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) &= \alpha \boldsymbol{d}_k^T A \boldsymbol{d}_k + \boldsymbol{d}_k^T A \boldsymbol{x}_k - \boldsymbol{d}_k^T \boldsymbol{b} \\ &= \alpha \boldsymbol{d}_k^T A \boldsymbol{d}_k + \boldsymbol{d}_k^T (A\boldsymbol{x}_k - \boldsymbol{b}). \end{aligned}$$

- Thus α_k satisfies the equation

$$\alpha_k \boldsymbol{d}_k^T A \boldsymbol{d}_k + \boldsymbol{d}_k^T (A\boldsymbol{x}_k - \boldsymbol{b}) = 0.$$

  Let $\boldsymbol{r}_k = A\boldsymbol{x}_k - \boldsymbol{b}$. Then

$$\alpha_k = -\frac{\boldsymbol{d}_k^T \boldsymbol{r}_k}{\boldsymbol{d}_k^T A \boldsymbol{d}_k}.$$

- Consequently, a conjugate direction method can be formed by repeating the steps

$$\begin{aligned} \boldsymbol{r}_k &= A\boldsymbol{x}_k - \boldsymbol{b}, \\ \alpha_k &= -\frac{\boldsymbol{d}_k^T \boldsymbol{r}_k}{\boldsymbol{d}_k^T A \boldsymbol{d}_k}, \\ \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k \end{aligned} \tag{2.9}$$

  from an initial guess $\boldsymbol{x}_0$, once a set of nonzero A-conjugate vectors $\{\boldsymbol{d}_0, \cdots, \boldsymbol{d}_{n-1}\}$ are known.

- How to find a set of nonzero A-conjugate vectors $\{\boldsymbol{d}_0, \cdots, \boldsymbol{d}_{n-1}\}$ in a computationally efficient way?

- Conjugate gradient method is a conjugate direction method with the conjugate vectors constructed successively during computation.

- It starts with

$$\boldsymbol{r}_0 = A\boldsymbol{x}_0 - \boldsymbol{b} \quad \text{and} \quad \boldsymbol{d}_0 = -\boldsymbol{r}_0$$

  and construct each new $\boldsymbol{d}_{k+1}$ as the linear combination of $\boldsymbol{r}_{k+1}$ and $\boldsymbol{d}_k$ of the form

$$\boldsymbol{d}_{k+1} = -\boldsymbol{r}_{k+1} + \beta_{k+1}\boldsymbol{d}_k,$$

  where β_{k+1} is chosen so that $\boldsymbol{d}_{k+1}$ and $\boldsymbol{d}_k$ are A-conjugate, i.e.

$$0 = \boldsymbol{d}_{k+1}^T A \boldsymbol{d}_k = -\boldsymbol{r}_{k+1}^T A \boldsymbol{d}_k + \beta_{k+1} \boldsymbol{d}_k^T A \boldsymbol{d}_k$$

  and hence

$$\beta_{k+1} = \frac{\boldsymbol{r}_{k+1}^T A \boldsymbol{d}_k}{\boldsymbol{d}_k^T A \boldsymbol{d}_k}.$$

- Combining this construction of $\boldsymbol{d}_{k+1}$ with (2.9), it leads to the following conjugate gradient method.

Algorithm: Conjugate gradient method[1.0ex]

- Pick a starting point $\boldsymbol{x}_0 \in \mathbb{R}^n$;

- Set $\boldsymbol{r}_0 := A\boldsymbol{x}_0 - \boldsymbol{b}$, $\boldsymbol{d}_0 := -\boldsymbol{r}_0$, and $k := 0$;
- While $\boldsymbol{r}_k \neq 0$ do

$$\alpha_k = -\frac{\boldsymbol{r}_k^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T A \boldsymbol{d}_k};$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k;$$

$$\boldsymbol{r}_{k+1} = A\boldsymbol{x}_{k+1} - \boldsymbol{b};$$

$$\beta_{k+1} := \frac{\boldsymbol{r}_{k+1}^T A \boldsymbol{d}_k}{\boldsymbol{d}_k^T A \boldsymbol{d}_k};$$

$$\boldsymbol{d}_{k+1} := -\boldsymbol{r}_{k+1} + \beta_{k+1} \boldsymbol{d}_k;$$

$$k = k + 1;$$

- End

- One can show that the vectors $\{\boldsymbol{d}_0, \boldsymbol{d}_1, \cdots, \boldsymbol{d}_{n-1}\}$ constructed by the algorithm are A-conjugate.
- The sequence $\{\boldsymbol{x}_k\}$ generated by the conjugate gradient method converges to the solution $\boldsymbol{x}^*$ of $A\boldsymbol{x} = \boldsymbol{b}$ in at most n steps.

Example 36. *Consider the linear system $A\boldsymbol{x} = \boldsymbol{b}$ of size $n = 600$, where*

$$A = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ n \end{bmatrix}.$$

We solve it by the conjugate gradient method, the method of steepest decent, and the SOR method with $\omega = 1.9$. The following figure plots the results on the relative error versus the iteration numbers.

2.7 The solution of nonlinear equations

nonlinear equations

- general form of equation, x real

$$f(x) = 0$$

- exact solution x^*

examples on how to choose $f(x)$:

1. geometric: compute length requires square root: $f(x) = x^2 - d$
2. decision problems: when cost $g(x)$ hits g_0: $f(x) = g(x) - g_0$
3. stationary states of dynamical systems
   - discrete time $x(t+1) = g(x(t))$: $f(x) = x - g(x)$
   - continuous time $x'(t) = g(x(t))$: $f(x) = g(x)$
4. optimisation of $g(x)$: maximum: $f(x) = g'(x)$

What is your favorite nonlinear equation? Can you recall an application?

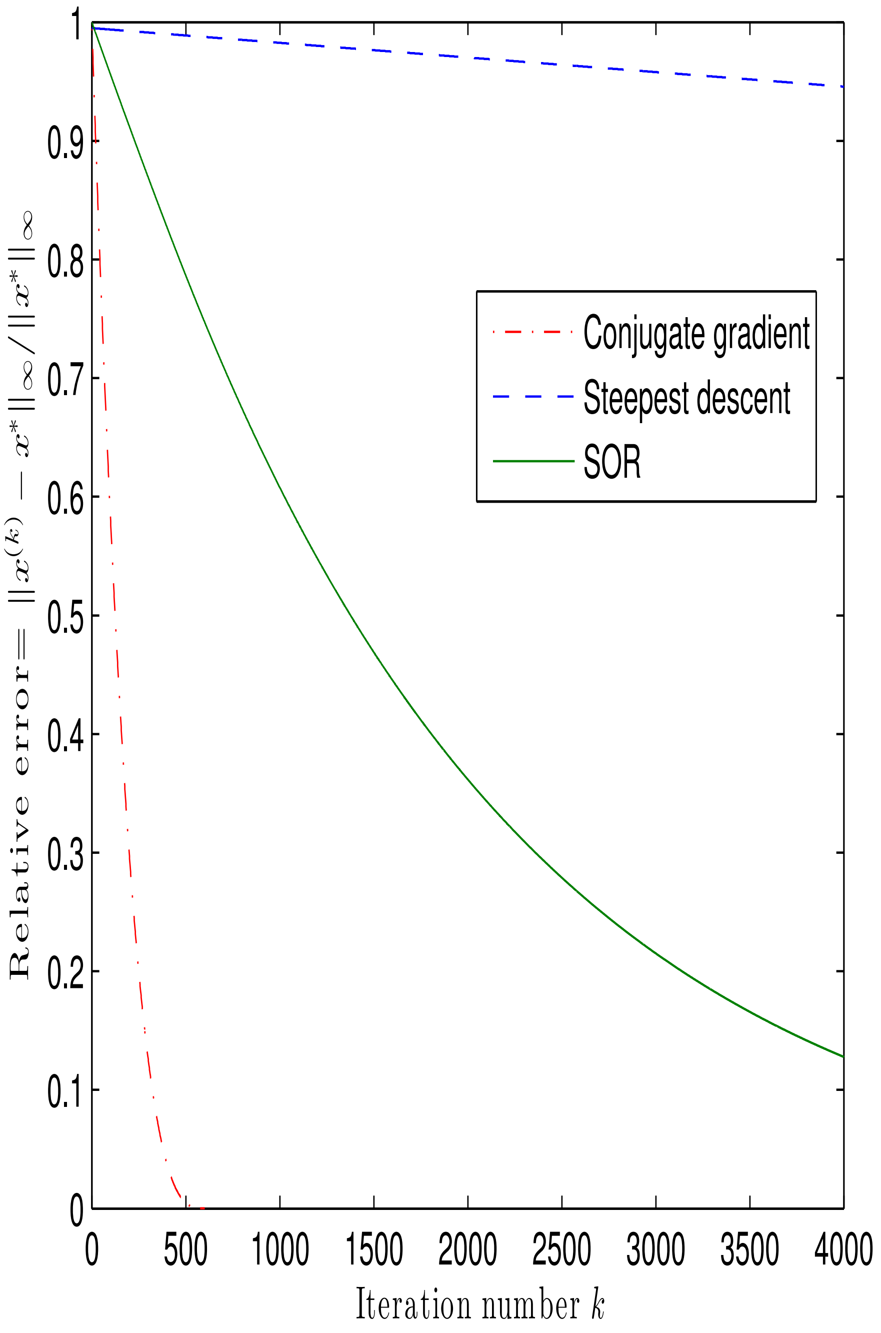


Figure 2.10: Comparison of convergence speed for the conjugate gradient method, the method of teepest descent, and the SOR method with $\omega = 1.9$

simple examples, explicit equations

- solution of linear equation $f(x) = ax + b$, $a \neq 0$:

$$x^* = -b/a$$

- quadratic equation $x^2 - d$ (square root of d)
  - has two solutions $x^*_{1,2} = \pm\sqrt{d}$ if $d > 0$
  - has one solution $x^* = 0$ if $d = 0$
  - has no (real) solutions if $d < 0$

the determination of the square root requires a numerical algorithm which is typically part of the system library (math in Python)

- range reduction: one only needs an algorithm to compute $\sqrt{d}$ for $1 < d < 4$ as

$$\sqrt{4^k d} = 2^k \sqrt{d}$$

  for all integer k

continuous functions f – getting help from calculus course

Theorem 18 (Bolzano's theorem)**.** *If $f(x)$ is a continuous real valued function on the interval $[a, b]$ and $f(a) \cdot f(b) \leq 0$ then there exists a solution $x = x^* \in [a, b]$ of the equation $f(x) = 0$.*

- consequence of intermediate value theorem for continuous functions
- existence of solution: If we know real numbers a and b such that $f(a)$ and $f(b)$ have different signs then we know that there is a solution of $f(x)$ between a and b

2.7.1 Basic iterative methods

iterations

- determine a sequence $x_0, x_1, \dots$ by

$$x_{n+1} = F(x_0, x_1, \dots, x_n)$$

  such that

$$x_n \to x^*, \quad \text{for } n \to \infty$$

2.7.2 The Bisection Method

Bisection Method

Now we can restrict our attention to the case $1 \leq s < 4$ (if $s = 1$ then $\sqrt{s} = 1$). In this case $f(x) = x^2 - s$ satisfies $f(1) < 0$ and $f(2) > 0$. See Figure 2.11.

bisection

This sign change together with the continuity of $f(x)$ guarantees that at least one root exists (by the intermediate value theorem). In fact since the function is non-decreasing, it can only cross the x-axis once: there is exactly one root, r. Further, from the inequalities above, this root must lie in the interval $1 \leq x < 2$, so we have a starting approximation.

The bisection method constructs successively smaller intervals guaranteed to contain a root using the above sign change condition. Essentially, we repeatedly divide the current interval in half and take the half for which the function has the opposite sign at its ends.

Let us be specific and assume $s = 3$ so $f(x) = x^2 - 3$. For the above function we start with the interval $[1, 2]$, with $f(1) = -2 < 0$, $f(2) = 1 > 0$. Subdividing by computing the midpoint 1.5,

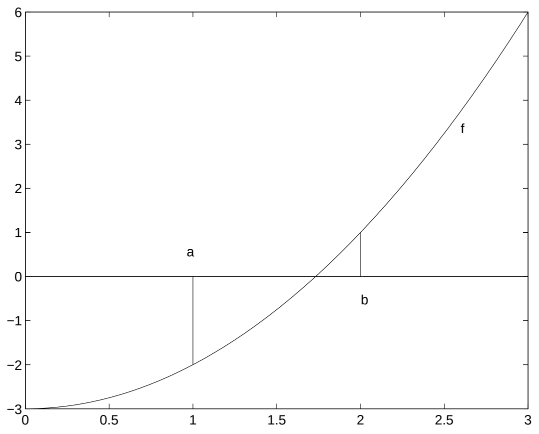


Figure 2.11: Calculation of $\sqrt{3}$ by finding the root of the equation $f(x) = x^2 - 3$.

one finds that $f(1.5) = -0.75 < 0$ so the sign change is in the right-hand half: the new interval containing the root is $[1.5, 2]$.

Some example iterations are given in Figures 2.12, 2.13, 2.14, 2.15, 2.16 and 2.17. A more complete set of the midpoints m, function values $f(m)$ and intervals $[a, b]$ are given in Tables 2.6 and 2.7.

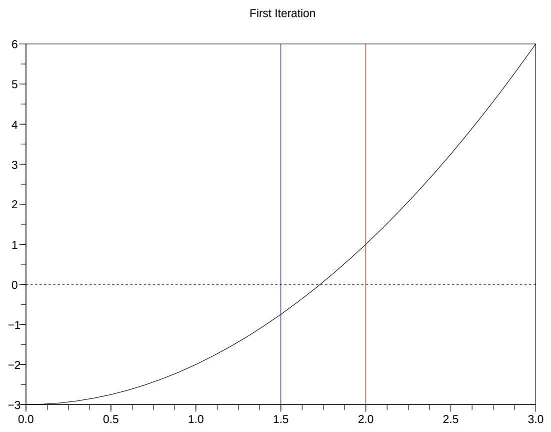


Figure 2.12: Calculation of $\sqrt{3}$ by finding the root of the equation $f(x) = x^2 - 3$. The interval for the first iteration is $[1.5, 2]$.

Each extra decimal place requires an extra three or four steps: this observation will be explained below.

Bisection Algorithm: First Version

Having seen the idea of zooming in on a root by repeated interval halving let us try to construct a precise algorithm, suitable for computer use to apply the bisection method to an arbitrary continuous function f.

%MH20072006 changed $|b - a|$ to $|b - a|/2$ in last step

Algorithm 2. *Simple Case*

```
1: Find $a$, $b$ that surround a root [$f(a)$,$f(b)$ of opposite signs].
2: Evaluate $f(a)$ and $f(b)$.
```

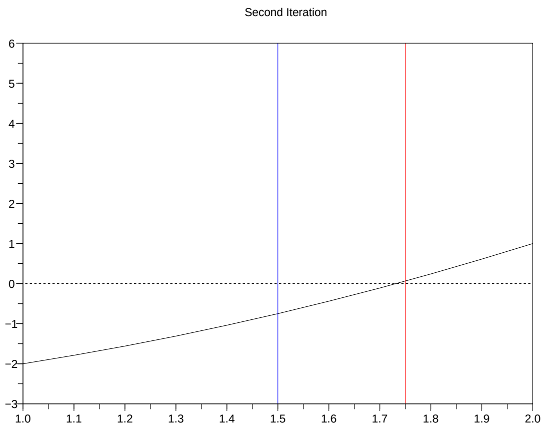


Figure 2.13: Calculation of $\sqrt{3}$ by finding the root of the equation $f(x) = x^2 - 3$. The interval for the second iteration is $[1.5, 1.75]$. Note the change in the range of the x-axis.

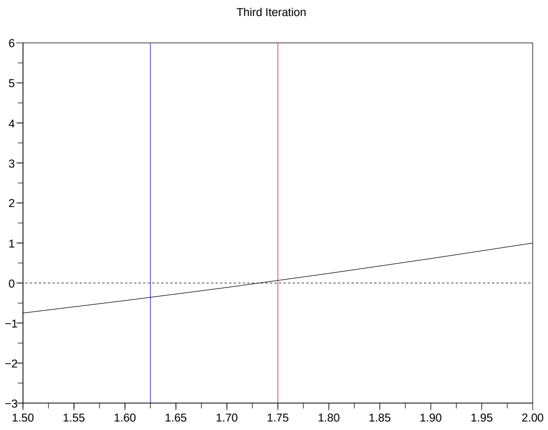


Figure 2.14: Calculation of $\sqrt{3}$ by finding the root of the equation $f(x) = x^2 - 3$. The interval for the third iteration is $[1.625, 1.75]$. Note the change in the range of the x-axis.

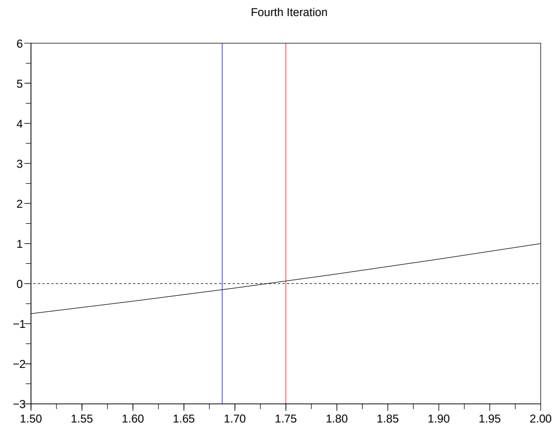


Figure 2.15: Calculation of $\sqrt{3}$ by finding the root of the equation $f(x) = x^2 - 3$. The interval for the fourth iteration is $[1.6875, 1.75]$.

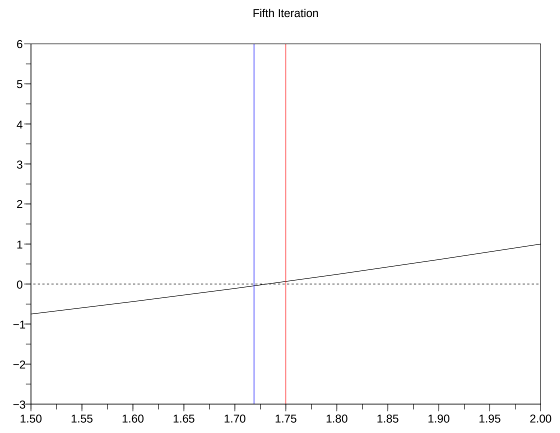


Figure 2.16: Calculation of $\sqrt{3}$ by finding the root of the equation $f(x) = x^2 - 3$. The interval for the fifth iteration is $[1.71875, 1.75]$.

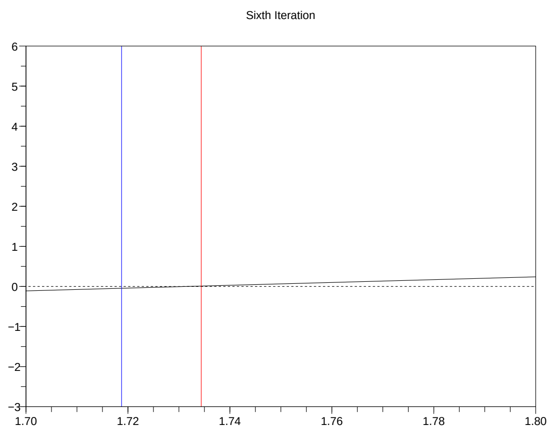


Figure 2.17: Calculation of $\sqrt{3}$ by finding the root of the equation $f(x) = x^2 - 3$. The interval for the sixth iteration is $[1.71875, 1.734375]$. Note the change in the range of the x-axis.

n	m	$f(m)$	a	b
0	1.50000000	-0.75000000	1.00000000	2.00000000
1	1.75000000	0.06250000	1.50000000	2.00000000
2	1.62500000	-0.35937500	1.50000000	1.75000000
3	1.68750000	-0.15234375	1.62500000	1.75000000
4	1.71875000	-0.04589844	1.68750000	1.75000000
5	1.73437500	0.00805664	1.71875000	1.75000000
6	1.72656250	-0.01898193	1.71875000	1.73437500
7	1.73046875	-0.00547791	1.72656250	1.73437500
8	1.73242188	0.00128555	1.73046875	1.73437500
9	1.73144531	-0.00209713	1.73046875	1.73242188
10	1.73193359	-0.00040603	1.73144531	1.73242188
11	1.73217773	0.00043970	1.73193359	1.73242188
12	1.73205566	0.00001682	1.73193359	1.73217773

Table 2.6: Calculation of $\sqrt{3} = 1.73205080756888$ using the bisection method for $0 \leq n \leq 12$

```
3: Compute the midpoint of the interval $m = (a+b)/2$ and evaluate $f(m)$.
4: if the approximation to the root is not yet accurate enough then
5:     replace either $a$ or $b$ with $m$ and go back to 3
6: end if
7: Stop with $m$ as the approximate value of the root and $|b-a|/2$ the maximum size of the error.
```

```
# first implementation of bisection algorithm
def bisect(f, inita, initb, tol):

    fval = tol+1.0    # initialise the function evaluation
    a = inita         # set the starting value of a
    b = initb         # set the starting value of b

    # while |f(m)| > tol continue to iterate
    while (abs(fval)> tol):
        m = (a+b)/2.0      # find the midpoint
        fval = f(m)        # evaluate function at the midpoint
```

n	m	$f(m)$	a	b
13	1.73199463	-0.00019461	1.73193359	1.73205566
14	1.73202515	-0.00008889	1.73199463	1.73205566
15	1.73204041	-0.00003603	1.73202515	1.73205566
16	1.73204803	-0.00000961	1.73204041	1.73205566
17	1.73205185	0.00000361	1.73204803	1.73205566
18	1.73204994	-0.00000300	1.73204803	1.73205185
19	1.73205090	0.00000031	1.73204994	1.73205185
20	1.73205042	-0.00000135	1.73204994	1.73205090
21	1.73205066	-0.00000052	1.73205042	1.73205090
22	1.73205078	-0.00000011	1.73205066	1.73205090
23	1.73205084	0.00000010	1.73205078	1.73205090
24	1.73205081	-0.00000000	1.73205078	1.73205084
25	1.73205082	0.00000005	1.73205081	1.73205084
26	1.73205081	0.00000002	1.73205081	1.73205082
27	1.73205081	0.00000001	1.73205081	1.73205081

Table 2.7: Calculation of $\sqrt{3} = 1.73205080756888$ using the bisection method for $13 \leq n \leq 27$

```
        if (fval < 0.0):  # find the new range [a, b]
            a = m
        else:
            b = m
        print([a, b, m, fval])  # display intermediate result

    return a, b
```

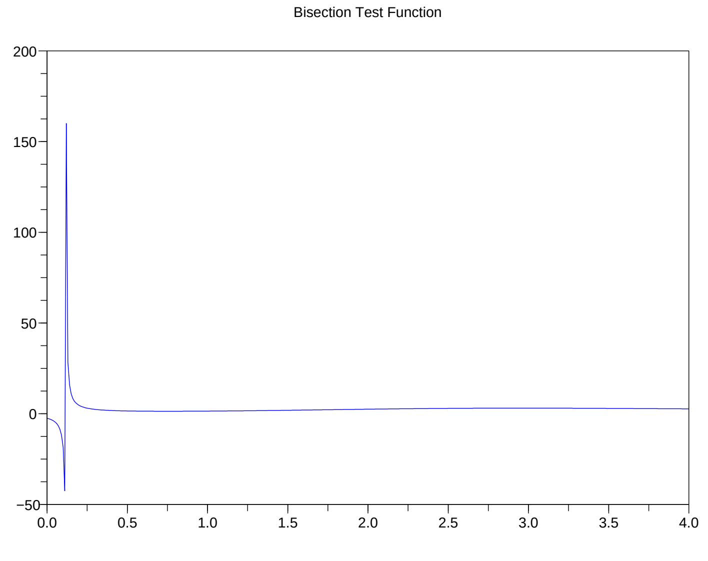


Figure 2.18: Plot of the function $f(x) = \frac{x^3+4x^2+3x+5}{2x^3-9x^2+18x-2}$ over the interval $[0, 4]$.

as can be seen in the following section of the output from Python, the algorithm failed to find the root. note that the last column lists $f(m)$.

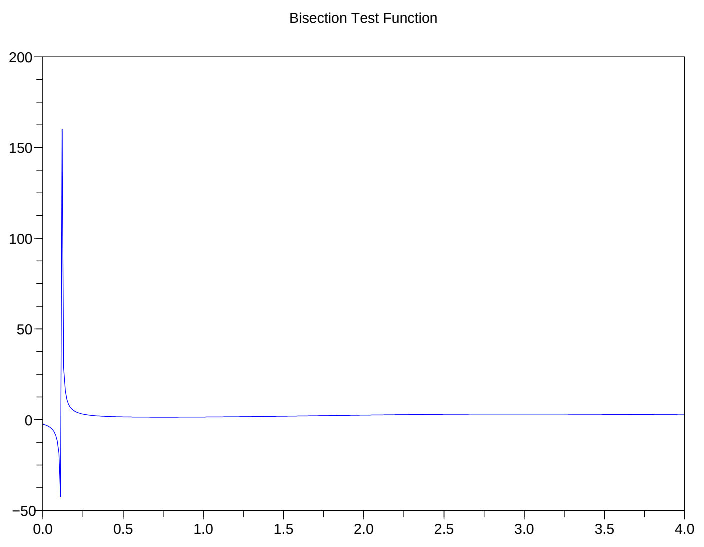


Figure 2.19: Plot of the function $f(x) = \frac{x^3+4x^2+3x+5}{2x^3-9x^2+18x-2}$ over the interval $[0, 0.5]$.

```
[0.11787656679530656, 0.11787656679530834, 0.11787656679530834, 435148019306295.44]
[0.11787656679530745, 0.11787656679530834, 0.11787656679530745, -2707587675683614.0]
[0.11787656679530745, 0.11787656679530789, 0.11787656679530789, 1015345378381355.6]
[0.11787656679530745, 0.11787656679530767, 0.11787656679530767, 3046036135144066.5]
[0.11787656679530756, 0.11787656679530767, 0.11787656679530756, -12184144540576264.0]
[0.11787656679530756, 0.11787656679530761, 0.11787656679530761, 6092072270288133.0]
[0.11787656679530756, 0.11787656679530759, 0.11787656679530759, 12184144540576266.0]

traceback (most recent call last):
  file "bisect_main.py", line 15, in <module>
    [a, b] = bisect(f, a1, b1, 1.0e-10)
  file "bisect.py", line 11, in bisect
    fval = f(m)          # evaluate the function at the midpoint
  file "bisect_main.py", line 5, in f
    z=(x**3+4.0*x**2+3.0*x+5.0)/(2.0*x**3-9.0*x**2+18.0*x-2.0)
zerodivisionerror: float division
```

other examples that may cause problems with algorithm are shown in figures 2.20, 2.21 and 2.22.

bisection algorithm: second version

Algorithm 3. *more complete bisection algorithm*

```
1: find $a$, $b$ that surround a root [$f(a)$,$f(b)$ of opposite signs], rename them if necessary so that
   $a < b$, and specify an iteration limit $m$, desired error bound $\delta$ and a "zero threshold" $\epsilon$.
2: evaluate $u = f(a)$, $v = f(b)$, $e = b - a$
3: if sign($u$) = sign($v$) then
4:     stop [failure due to bad initial interval]
5: end if
6: initialise the count of steps done, $n = 0$
7: set $e = e/2$, $m = a + e$, $w = f(m)$, $n = n + 1$
8: if $e < \delta$ then
9:     stop: output $m$, $e$ and $w$, and stop [success]
10: else if $|w| < \epsilon$ then
11:     stop: output $m$, $e$ and $w$, and stop ["probable success"]
12: else if $n = m$ then
13:     stop: output $m$, $e$ and $w$, and stop [failure to converge]
14: end if
15: if $\text{sign}(f(m)) = \text{sign}(f(a))$ then
```

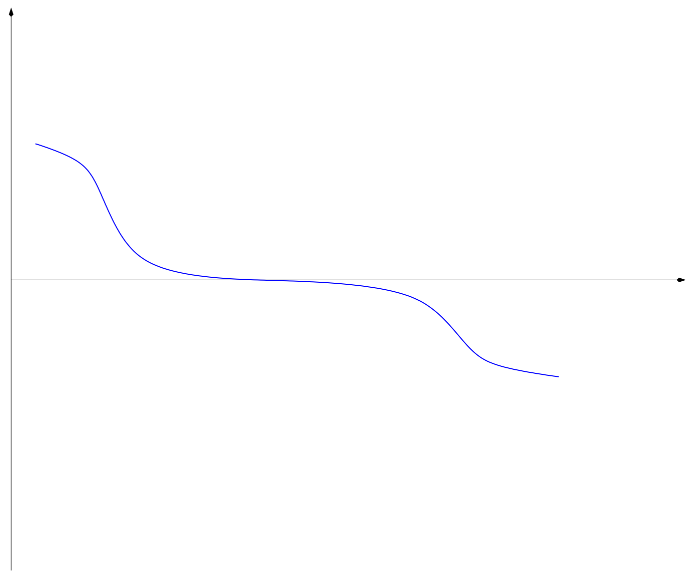


Figure 2.20: the bisection algorithm may have problems finding the roots of examples like this one.

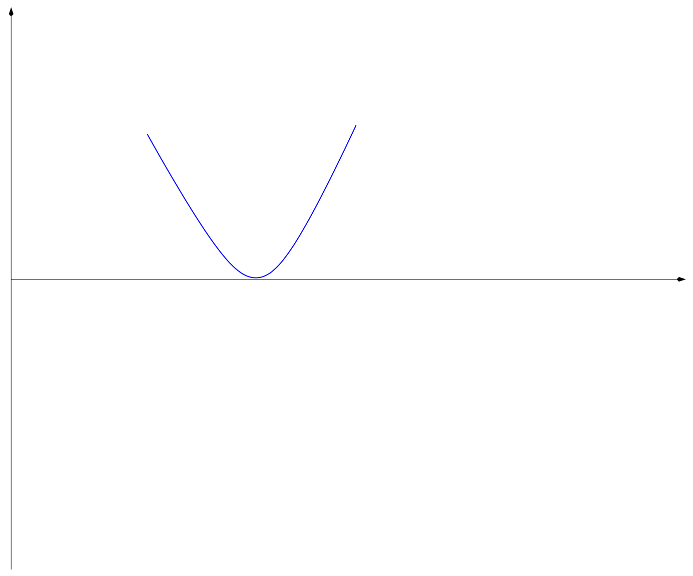


Figure 2.21: another example that may cause problems with the bisection algorithm.

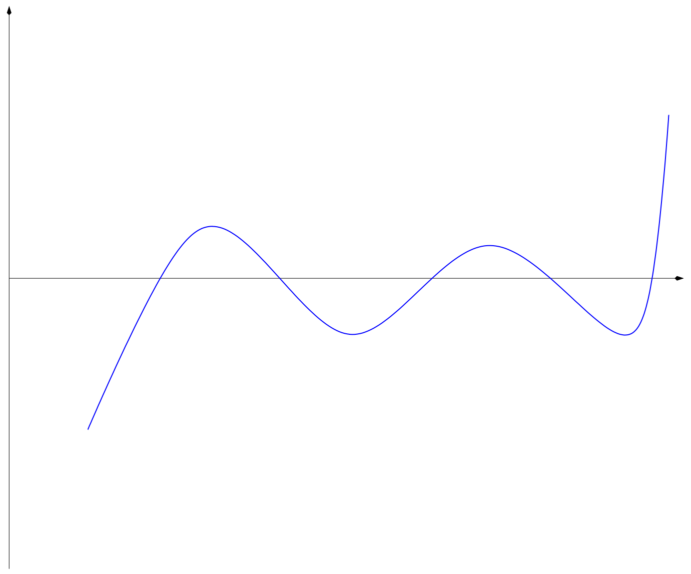


Figure 2.22: yet another example that may cause problems with the bisection algorithm.

```
16:     set $a = m$, $u = w$
17: else
18:     set $b = m$, $v = w$
19: end if
20: repeat from step 7.
```

an advantage of this method is that it is impervious to roundoff errors in the calculation of $f(x_i)$, because only the sign is used in the choice of subinterval.

however, when small roundoff errors can be made in the interval boundaries, there is the possibility of the sign changing and a wrong decision being made. there is also an interval of uncertainty about the estimated root.

convergence, error analysis and convergence rate

the bisection method gives a very simple and reliable upper bound on the error so long as $f(x)$ is known to be continuous on the search interval.

when it stops, there is a root x^* in the final interval (a, b) and its distance from the midpoint m of that interval is at most $(b - a)/2 = d$. thus the error in m as an approximation to the root is bounded by $|x^* - m| < d$.

how quickly does this error bound get down to the required value of δ?

let a_0, b_0 be the initial endpoints of the interval, $d_0 = (b_0 - a_0)/2$, and m_n the midpoint at step n.

at each step n a new interval is found of length $d_n = d_{n-1}/2$. a simple induction argument gives the error bound

$$|x^* - m_n| < d_n/2 = (b_0 - a_0)/2^{n+1}, \to 0 \text{ as } n \to \infty.$$

thus (ignoring for now problems of rounding error) the error can be made as small as one wishes by taking enough steps: the method is guaranteed to succeed (to converge to *a* root) for any continuous function. further, an error tolerance δ is achieved when $(b_0 - a_0)/2^{n+1} \leq \delta$ which is true once the number of steps reaches

$$n \geq \log_2 \frac{b_0 - a_0}{\delta} - 1 = -\log_2 \delta + \log_2(b_0 - a_0) - 1.$$

adding each extra decimal place of guaranteed accuracy involves reducing δ tenfold which increases the number of steps required by $\log_2 10 = \log 10/\log 2 \simeq 3.3$ which explains the pattern seen of needing three or four iterations per decimal place.

thus we know in advance how many iterations are required for a specified accuracy, or tolerance, in the solution.

it should be noted here that some functions may have several roots in an interval, and the bisection method will only locate one of these. also, within the interval, the function may have a singularity and this method will converge to this singularity.

orders of convergence

when a method converges as a factor (less than 1) times the previous uncertainty to the first power, it is said to converge *linearly.* those that converge as a higher power are said to converge *superlinearly.* that is, in mathematical terms,

$$d_{n+1} = constant \times (d_n)^m$$

where $m = 1$ for linear convergence and $m > 1$ for superlinear convergence.

for the bisection method then, with the error bound[1] d_n, the new error bound in terms of the old was

$$d_{n+1} = d_n/2$$

and so the convergence is linear.

suppose $x_1, x_2, x_3, ...$ is a sequence of real numbers approaching a limit x^*

- the rate of convergence is at least *linear* if there is a constant $c < 1$ and an integer n such that
$$|x_{n+1} - x^*| \leq c|x_n - x^*| \qquad (n \geq n),$$
- the rate of convergence is at least *superlinear* if there exists a sequence ϵ_n approaching 0 and an integer n such that
$$|x_{n+1} - x^*| \leq \epsilon_n |x_n - x^*| \qquad (n \geq n),$$
- if c (not necessarily < 1) and an integer n exist such that
$$|x_{n+1} - x^*| \leq c|x_n - x^*|^2 \qquad (n \geq n)$$
then the rate of convergence is at least *quadratic.*

for example, the sequence $x_{i+1} = x_i/2 + 1/x_i$ approaches $\sqrt{2}$. according to figure 2.23, what is its convergence rate? the y axis shows the values of $|x_{n+1} - \sqrt{2}|/|x_n - \sqrt{2}|^\alpha$ for $\alpha = 1$, 2 and 3.

using a diagram such as the one given in figure 2.23 does not replace a proof, but it is a good way to get an idea of what the answer should be.

Definition 30 (orders of convergence)**.** *in general, if constants α and c and an integer n exist such that*

$$|x_{n+1} - x^*| \leq c|x_n - x^*|^\alpha \qquad (n \geq n)$$

then the rate of convergence is order α.

[1]error $e_n = x^* - x_n$ and $x_n = m_n$

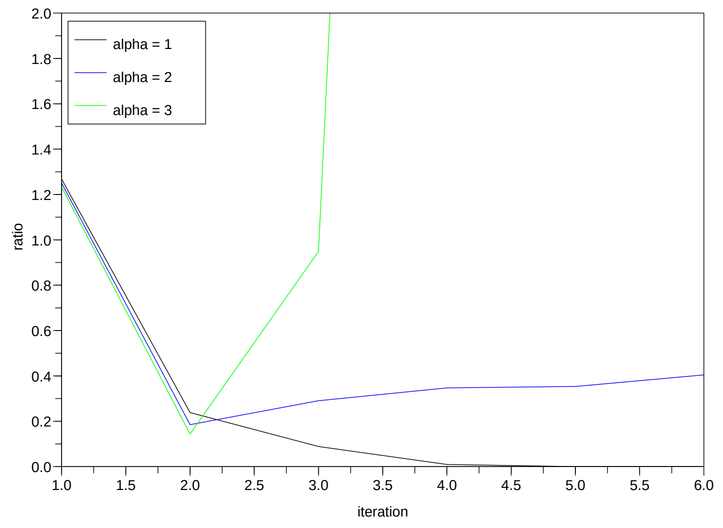


Figure 2.23: plots of $|x_{n+1} - \sqrt{2}|/|x_n - \sqrt{2}|^\alpha$ for $\alpha = 1$, 2 and 3

2.7.3 second order methods

square roots revisited

returning to our earlier problem we would like to construct an algorithm for calculating $\sqrt{s}$ which is more efficient.

we will first look at a method that works specifically for the calculation of square roots, and then we will study a general method (Newton's method) which provides an automatic way to construct such higher order methods.

what we would like to construct is an iterative scheme for x_n which satisfies

$$|x_{n+1} - \sqrt{s}| \leq |x_n - \sqrt{s}|^2. \tag{2.10}$$

with reference to the example above $f(x) = x^2 - 3$, on the interval $[1, 2]$ and with an initial value $x_0 = 1.5$, we will have,

$$\begin{array}{rll} & |x_0 - \sqrt{s}| & \leq \frac{1}{2} \\ |x_1 - \sqrt{s}| & \leq |x_0 - \sqrt{s}|^2 & \leq \left(\frac{1}{2}\right)^2 \\ |x_2 - \sqrt{s}| & \leq |x_1 - \sqrt{s}|^2 & \leq \left(\frac{1}{2}\right)^{2^2} \\ |x_3 - \sqrt{s}| & \leq |x_2 - \sqrt{s}|^2 & \leq \left(\frac{1}{2}\right)^{2^3} \end{array} \tag{2.11}$$

and in general

$$|x_n - \sqrt{s}| \leq \left(\frac{1}{2}\right)^{2^n}. \tag{2.12}$$

from table **??** (page **??**) we see that 28 iterations are necessary to ensure eight correct digits for the bisection method, whereas according to equation (2.12) five iterations suffices for a method satisfying expression (2.10).

an iterative method with property (2.10) is said to be *second order* or to possess *quadratic behaviour*. in view of the expected payoff, it is very sensible to search for a quadratic iterative scheme.

towards this goal, let us search for a p_n such that

$$x_{n+1} - \sqrt{s} = p_n(x_n - \sqrt{s})^2.$$

rewriting this expression we see that

$$x_{n+1} = p_n \left(x_n^2 + s\right) + (1 - 2x_n p_n)\sqrt{s}.$$

we rid ourselves of the $\sqrt{s}$ term if we set $p_n = 1/(2x_n)$.
hence we are led to the iterative scheme

$$x_{n+1} = \frac{x_n^2 + s}{2x_n} = \frac{1}{2}\left(x_n + \frac{s}{x_n}\right)$$

which by construction is *second order accurate.*

example application

Example 37. *let us see how our calculation of $\sqrt{3}$ fares using this new method. (our example was $f(x) = x^2 - 3$ on an interval $[a, b] = [1, 2]$, and with the initial condition $x_0 = 1.5$.)*

n	x_n	$x_n - \sqrt{3}$
0	1.50000	-0.232050807569
1	1.75000	0.017949192431
2	1.73214	0.000092049574
3	1.73205	0.000000002446

Table 2.8: calculation of $\sqrt{3} = 1.73205080756888$ using the second order method

as can be seen in table 2.8 (page 77) the required accuracy of eight correct digits is found in four iterations.

2.7.4 Newton's method for solving equations

Newton's method v's bisection method

Newton's method provides a way to generate second order iterative schemes for the solution of fairly general nonlinear equations $f(x) = 0$.

one reason that the bisection method is rather slow is that it only uses the sign of $f(x)$ at each point where it is evaluated, ignoring the magnitude as a hint as to how close the root is.

Newton's method uses the value of $f(x)$ and also its derivative to make a better estimate of how far away the root is. it is a first example of a common paradigm in numerical methods for producing a succession $x_0, x_1, \ldots$ of approximations to the answer x^*, with errors $e_n = x_n - x^*$ that hopefully converge to zero as n increases.

iterative approximations

1. find an initial approximation x_0 to the solution, x^*.
2. at the n'th step, compute an *estimate* $\hat{e}_n$ of the (unknown) error e_n: that is one would like $\hat{e}_n \approx e_n$
3. define the new approximation by adding this error estimate to the previous one: $x_{n+1} = x_n - \hat{e}_n$. if the error estimate is in fact exactly right, x_{n+1} will be the exact answer.
4. repeat from step 2 until the approximation is judged to be sufficiently accurate: usually the size of the error estimate $\hat{e}_n$ will be one factor in the decision about whether to stop.

Newton's method

consider the taylor remainder formula for the value of $f(x^*)$ at $x = x_n$:

$$0 = f(x^*) = f(x_n) + (x^* - x_n)f'(\xi_n)$$

for some (unknown) intermediate value ξ_n between x^* and x_n. the error is then

$$e_n = x_n - x^* = f(x_n)/f'(\xi_n).$$

an approximation is obtained by substituting the unknown ξ_n by x_n:

$$\hat{e}_n = f(x_n)/f'(x_n).$$

consequently, the next iterate is then

$$x_{n+1} = x_n - \hat{e}_n = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{2.13}$$

geometrically, x_{n+1} is just the x-coordinate of the intersection between the x-axis and the tangent to the graph of f through the point $(x_n, f(x_n))$. if x_n is close to the solution then $|x - x_n|$ is small and the approximation is good. on the other hand, if x_n is not close then the truncated terms are significant and the approximation is not accurate.

the simplest algorithm iterates the formula (2.13) until some stopping criterion is met. see the following fragment of a Python program.

```
# first implementation of Newton algorithm
def newton(f, df, x0, delta):
    x = x0              # initial guess
    err = delta+1.0     # set an upper bound on the error
    print('The inital guess is', x);
    print('f(x)= ',f(x));

    # while the error estimate is large
    while err > delta :
        y = f(x);         # evaluate the function
        dydx = df(x);     # evaluate the derivative
        dx =  - y/dydx;   # find the step size
        err = abs(dx);    # estimate the error
        x = x + dx;       # take a step
        print('The new approximate root is', x);
        print('f(x)= ',y);
        print('The error estimate is', err);
    return x
```

example application

Example 38. *using Newton's method, employing double precision computation, find the negative zero of the function $f(x) = \exp(x) - 1.5 - \arctan(x)$*[2]

with an initial value of $x_0 = -7$ the results are seen in table 2.9 (page 79).

about three steps are needed to get one decimal place of accuracy, but then it "accelerates", roughly doubling the number of decimal places of accuracy at each step.

convergence and error analysis of Newton's method

the standard questions should be asked: when does this method succeed, giving approximations that converge to a root? how fast is it? and what precautions should be taken to be sure that it will work in a particular case, or at least to detect failure?

Example 39. *using Newton's method find the root of the function $f(x) = x/(1 + x^2)$.*

with an initial guess of 0.1, Newton's method takes four iterations to converge to the root. observe that this function only has one root at $x = 0$. if instead the initial guess is set to 0.75, the method diverges and PYTHON crashes with an overflow error.

[2] $\tan^{-1}(x)$ is used sometimes to denote $\arctan(x)$

n	x	$f(x)$
0	-7.00000000000000000000000000	-0.702×10^{-1}
1	-10.67709617664001399296984386	-0.226×10^{-1}
2	-13.27916737563271290859786319	-0.437×10^{-2}
3	-14.05365585426923873474831753	-0.239×10^{-3}
4	-14.10110995686641347616312706	-0.800×10^{-6}
5	-14.10126977093941594621579506	-0.901×10^{-11}
6	-14.10126977273996842508300314	-0.114×10^{-20}
7	-14.10126977273996842531155122	0.000

Table 2.9: solving $\exp(x) - 1.5 - \arctan(x) = 0$ with Newton's method

error analysis of Newton's method

how does the size of the error change from one step to the next.

with $e_n = x_n - x^*$ and $\hat{e}_n = x_n - x_{n+1}$

$$\begin{aligned} e_{n+1} &= x_{n+1} - x^* = e_n - \hat{e}_n = e_n - \frac{f(x_n)}{f'(x_n)} \\ &= \frac{e_n f'(x_n) - f(x_n)}{f'(x_n)} \end{aligned} \tag{2.14}$$

and then using taylor's formula with an error term

$$0 = f(x^*) = f(x_n - e_n) = f(x_n) - e_n f'(x_n) + \frac{1}{2} e_n^2 f''(\xi_n)$$

where ξ_n is between x_n and x^*. from this equation

$$-e_n f'(x_n) + f(x_n) = -\frac{1}{2} e_n^2 f''(\xi_n)$$

so

$$e_{n+1} = \frac{f''(\xi_n)}{2f'(x_n)} e_n^2 \approx \frac{f''(x^*)}{2f'(x^*)} e_n^2 = c e_n^2$$

if the approximations are reasonably close to the root x^*.

thus each error is roughly proportional to the square of the previous one: this is *quadratic convergence* and means that the number of decimal places of accuracy roughly doubles at each step once the approximations are near the root.

the obvious way to force convergence is to make sure that the error reduces at each step. this requires

$$\left| \frac{f''(\xi_n)}{2f'(x_n)} e_n \right| < 1$$

and this can be guaranteed by first ensuring that $f''(\xi_n)/2f'(x_n)$ is bounded on some interval near the root, and then getting a good enough initial approximation (so that e_n is small).

to make this precise, consider the interval containing the points within some distance δ of the root, $|x - r| \leq \delta$. the coefficient of e_n^2 above is bounded on that interval by

$$c(\delta) = \frac{1}{2} \max_{|x-r| \leq \delta} |f''(x)| / \min_{|x-r| \leq \delta} |f'(x)|.$$

then when $|e_n| \leq \delta$, $|e_{n+1}| \leq c(\delta)\delta|e_n|$ so the error decreases to the next step if we choose δ small enough such that $\rho = c(\delta)\delta < 1$. so as long as $f'(r) \neq 0$ and f' and f'' are continuous near r, this will hold for a small enough value δ. then the result $|e_{n+1}| < |e_n|$ guarantees that the next approximation is also within this distance of r, so the same argument holds repeatedly at subsequent steps. in fact $|e_{n+1}| \leq \rho|e_n|$, so starting with $|e_0| \leq \delta$, $|e_n| \leq \rho^n|e_0|$, $\to 0$ as n increases.

recalling that a *simple root* of a function is one where the derivative is non-zero, we have established the following theorem

Theorem 19 (Newtons method)**.** *if f'' exists and is continuous near a simple root r of $f(x) = 0$ then there is a neighbourhood $|x - r| < \delta$ of r such that when Newton's method is started with initial approximation x_0 in this neighbourhood, the successive approximations converge to the root with*

$$|x_{n+1} - r| \leq c(x_n - r)^2$$

for some constant c.

this in itself is not very practical as the problem of finding a suitable δ is often intractable. however a couple of useful things do follow from it.

firstly,

Theorem 20. *Newton's method always works for a function that is increasing (or decreasing) and convex (or concave), and has a continuous second derivative.*

secondly, it suggests an empirical method to at least detect whether Newton's method is failing: that is, once e_n is small, e_{n+1} should become far smaller than e_n, so to a good approximation $e_n = x_n - x^* \approx x_n - x_{n+1}$ and so

$$(x_n - x_{n+1})/(x_{n-1} - x_n)^2 \approx c$$

it can be checked numerically whether the quantity on the left does converge to a constant c. also, this says that the error estimates $\hat{e}_n = x_n - x_{n+1}$ should decrease rapidly once the approximation is reasonably close. thus as a rule of thumb, if this error estimate actually increases from one step to the next (after the first few), it is a warning that something is probably wrong.

global v's local convergence

when successful, Newton's method is powerful and converges quickly. it is a good choice when the derivative can be evaluated easily and is continuous and nonzero in the neighbourhood of the root.

however, it also has drawbacks in that it may display poor global convergence. it is used at times to 'polish' a root (solution) by applying a few steps of the procedure, to gaining twice or four times the number of significant figures. it is typically combined with a slower method which has better convergence properties.

calculating the derivative

in order to evaluate the derivative at each step, a numerical difference can be used from the definition of the derivative which we know as

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

and which is most accurate when h is small.

but here two function evaluations are required at each step, which typically doubles the amount of work required to reach a certain accuracy.

in addition, if h is too small, roundoff errors may dominate, while if h is too large the convergence order may only be linear.

the secant method is able to avoid some of these pitfalls while not having to make explicit use of the derivative $f'(x)$.

2.7.5 the secant method for solving equations

the secant method – another error approximation

we start by expanding $f(x^*)$ – like in Newton's method.

$$0 = f(x^*) = f(x_n) + \frac{f(x^*) - f(x_n)}{x^* - x_n}(x^* - x_n).$$

from this we get for the exact solution

$$x^* = x_n - f(x_n)\frac{x^* - x_n}{f(x^*) - f(x_n)}$$

and thus the (exact) value of the error is

$$e_n = x_n - x^* = f(x_n)\,\frac{x^* - x_n}{f(x^*) - f(x_n)}.$$

an approximation is obtained by replacing the unknown x^* by x_{n-1}:

$$\hat{e}_n = f(x_n)\,\frac{x_{n-1} - x_n}{f(x_{n-1}) - f(x_n)}.$$

this leads directly to the iteration used by the secant method:

$$x_{n+1} = x_n - \hat{e}_n = x_n - f(x_n)\,\frac{x_{n-1} - x_n}{f(x_{n-1}) - f(x_n)}. \tag{2.15}$$

as already mentioned, several circumstances can make it difficult or impossible to use Newton's method ($x_{n+1} = x_n - f(x_n)/f'(x_n)$). one is that the derivative of the function $f(x)$, whose roots are sought, is not directly available: for example when $f(x)$ is given only as the output of another computer program. another disadvantage can be that there is a formula available for the derivative, but its evaluation is very time consuming compared with the evaluation of $f(x)$.

this then, is called the secant method, and the following is a geometrical interpretation of the formula (2.15).

given a sequence of approximations $x_0, x_1, \ldots, x_{n-1}, x_n$ to a root of $f(x) = 0$, take the two most recent (and hopefully most accurate) approximations, and approximate the function $f(x)$ by the secant line through the two points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$. this gives the linear function

$$l(x) = f(x_n) + \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x - x_n).$$

then take the root of this function as the next approximation x_{n+1} of the root of $f(x)$: this gives (2.15) again.

this suggests other variants in the choice of two points (from the collection $x_0, \ldots, x_n$) to use in constructing the linear approximation. one idea would be to choose the two where the value of $|f(x)|$ is smallest, using this as a natural indication of closeness to a root. however once the secant method is reasonably close to the root, the two most recent approximations will be the two most accurate in this sense so the methods are more or less equivalent. this will be verified below when it is shown that in general the errors in the secant method approximations decrease rapidly at each step (faster than first order convergence).

a second idea would be to keep the root "bracketed" by the pair of points: insist on having a sign change between the two points kept as with the bisection method. this is known as *regula falsi*, but turns out to be a bad idea: from some stage on, all the new approximations are on the same side of the actual root (as with Newton's method) so the closest point on the other side is one of the earlier, less accurate approximations. thus the interval does not even shrink down to zero length, and the linear approximation is not improving much in accuracy, giving slow convergence. in fact the convergence, or regula falsi, is first order and so again the error analysis below shows the superiority of the secant method.

thus we will concentrate on the secant method as given by the above formula.

secant algorithm

```
# first implementation of Secant algorithm
def secant(f, x1, x2, delta, epsilon, N):
    f1 = f(x1); f2 = f(x2)      # initial guess
    for step in range (1,N):      # upper bound on interations
        if f1 == f2:              # avoid devision by zero
            print('error: division by zero')
            return x1
        dx = -f2*(x2-x1)/(f2-f1)  # find step szie
        x3 = x2+dx;               # take a step
        err = abs(dx);            # approximate the error
        f3 = f(x3);               # evaluate function at point
        if ((err < delta) | (abs(f3) < epsilon)):
            root = x3;            # return if error is small or
            return root           # x3 close to root
        x1 = x2; x2 = x3;         # next iteration
        f1 = f2; f2 = f3;

    # if maximum number of iterations reached print message
    print('error: iteration limit was reached without convergence')
    return x1
```

a complete algorithm is given above in Python form. note that it needs *two* initial approximations of the root, `x1` and `x2`. these might be chosen to “bracket” the root as with the bisection method, but there is no guarantee that the root found will be in this interval (the algorithm could be modified to ensure this). it also requires the familiar inputs: the name of a Python function to evaluate $f(x)$ and the standard three parameters for stopping conditions, `delta`, `epsilon` and `n`. the final approximation to the root is returned through the variable `root`.

note the recycling of the function values, so as to evaluate $f(x)$ only once per step rather than twice. also the reuse of variable names must be done in the correct order (think what happens if you do `x2=x3` before `x1=x2`, for example).

a disadvantage in this method can occur when the root does not remain bracketed, and the algorithm need not converge branching off to to infinity. alternatively, for certain functions the method may take ages to converge.

convergence and error analysis of the secant method

to compare the efficiency of the secant method with others, we need to analyse its convergence rate. in particular this will help to judge whether it or Newton's method is likely to be the faster in any particular case.

error analysis

defining $e_n = x_n - x^*$ where x^* is a root as before, the new error can be related to the errors in the two previous approximations, due to the appearance of both x_n and x_{n-1} on the right-hand side of (2.15). omitting the calculations, one gets the result

$$e_{n+1} \approx \frac{1}{2}\frac{f''(x^*)}{f'(x^*)} e_n e_{n-1} = c e_n \, e_{n-1}$$

when the errors are small, so long as f is twice continuously differentiable and $f'(x^*) \neq 0$.

here c is the same constant as in the corresponding result for Newton's method.

since we expect the errors to be reducing, the quantity $ce_n e_{n-1}$ will typically be bigger than ce_n^2 appearing for Newton's method, so the reduction in the error is not as fast from one step to the next, and this suggests that it is slower than second order.

on the other hand, the ratio of consecutive errors $e_{n+1}/e_n \simeq ce_{n-1}$ which goes to zero, so the convergence is faster than first order.

in fact it can be proven that

$$|e_{n+1}| \simeq a|e_n|^{\phi}, \qquad \text{where } \phi = \frac{1+\sqrt{5}}{2} \simeq 1.62$$

which is the golden mean ratio. so indeed the convergence is between first and second order in speed.

the proof of this is difficult, but once you *assume* that the errors satisfy the relation

$$|e_{n+1}| \simeq a|e_n|^{\alpha}$$

for some power α, it is easy to verify that the power must be the golden ratio ϕ.

again the computation is omitted. more important is the consequence for comparing the speeds of different methods. on the one hand, when computing to high enough accuracy, Newton's method will converge more quickly in that it requires fewer steps to reach a given degree of accuracy. however the time taken for each step will be different and this can favour the secant method.

in general the time involved will largely be the time required for the evaluations of $f(x)$ and $f'(x)$. if the latter is at least as time consuming as the former, two steps of the secant method take less time (or not much more) than one of Newton's method. what is the order of convergence of the new "method" in which each step is equivalent to two in the secant method? for the secant method,

$$|e_{n+2}| \simeq a|e_{n+1}|^{\phi} \simeq a(ae_n^{\phi})^{\phi} = a^{1+\phi}.|e_n|^{\phi^2} = a^{1+\phi}|e_n|^{(3+\sqrt{5})/2}.$$

thus this is a method of order $(3+\sqrt{5})/2 \simeq 2.62 > 2$, which says that it should converge faster than Newton's method when a small enough error is demanded. in other words the secant method will typically converge in fewer than twice as many steps as in Newton's method, and in the common case that the cost of each step is not much more than half of that for Newton's method, the secant method will be the more efficient.

these claims can be tested (at least the observation about the relative number of steps needed) by using PYTHON to compute roots of various test functions, starting with comparable initial guesses.

overall then, there is a tradeoff between speed and reliability in the bisection method, Newton's method and the secant method

2.7.6 solving equations by function iteration

function iteration formulation

Newton's method can be written in the form

$$x_{n+1} = f(x_n) \tag{2.16}$$

with $f(x) = x - f(x)/f'(x)$. [it is important to keep track of the distinction between the the functions f and f!]

other methods like the secant method can be put in the form (2.16) as well, by considering each x_n to be a vector whose components are each approximations of the desired root, but we will keep to the basic case where each x_n is a number. Newton's method and the secant method are two of many applications of a useful strategy in solving various sorts of equations:

1. start with one (or several) approximations of the solution,
2. use it (or them) to estimate the error in the latest approximation, and then subtract this error estimate to get a new, hopefully better, approximation,
3. then repeat the process computing a sequence of approximations which hopefully converges to the exact solution.

fixed point iteration

Definition 31 (fixed point iteration)**.** *suppose the sequence of approximations given by (2.16),*

$$x_0,\, x_1 = f(x_0),\, x_2 = f(x_1), \ldots$$

do approach some limit,

$$\lim_{n\to\infty} x_n = s.$$

then, if f is continuous at s, $f(x_n) \to f(s)$ while also $f(x_n) = x_{n+1} \to s$. thus

$$f(s) = s. \tag{2.17}$$

when this condition occurs, s is called a fixed point of the function f.

we will study various ways of putting mathematical problems in the form where the solution s is a fixed point of a suitable function f, and the sequence of *iterations* defined by (2.16) can be guaranteed to converge to a suitable fixed point, at least for some choices of the starting point. this is so that we have a procedure for computing arbitrarily accurate approximations of the fixed point.

convergence of fixed point iterations

how must f behave near s in order for this convergence to occur?

intuitively you would expect the errors to decrease at each step once the approximations are good: that is

$$|x_{n+1} - s| < |x_n - s|.$$

this is equivalent to $|f(x_n) - f(s)| < |x_n - s|$, and since we want a condition that does not refer to the as yet unknown solution s, it perhaps makes sense to require that

$$|f(x) - f(y)| < |x - y| \tag{2.18}$$

whenever x and y are in some region where we expect to find a fixed point.

this turns out to be almost, but not quite, enough. for example the function

$$f(x) = |x| + e^{-|x|}$$

can be shown to have this property but clearly has no fixed point ($f(x) > x$ always).

contractive mapping

one extra condition which will make things work is that the errors decrease at some guaranteed rate, like the halving seen with the bisection method. thus consider the condition

$$|f(x) - f(y)| < \lambda|x - y|, \qquad \lambda < 1. \tag{2.19}$$

when this holds for all x and y in some set, f is called *contractive* on that set. this is useful because of the following fact:

Theorem 21 (the contractive mapping theorem)**.** *let f be contractive for all x in a closed bounded interval $i = [a, b]$ with $f(x) \in i$ for all $x \in i$. then f has a unique fixed point in that interval. further, for any $x_0 \in i$, the iteration defined by (2.16) will converge to this fixed point.*

higher order convergence

Theorem 22 (higher order convergence)**.** *suppose that the conditions of theorem 21 hold. let the unique fixed point in i be denoted s. suppose $f'(s) = f''(s) = \cdots = f^{(m)}(s) = 0$. then for any $x_0 \in i$, the iteration defined by (2.16) will converge to s with order $m+1$. in particular, there will exist a positive constant c such that*

$$|e_{n+1}| \le c\,|e_n|^{m+1}.$$

a simple taylor series expansion about the fixed point should convince you of this result. indeed if we start "close" enough to a fixed point where all the derivatives up to order m are zero, then the fixed point iteration algorithm will converge with an order of $m+1$. an advantage of the iterative procedures is that they are not vulnerable to accumulative roundoff errors.

example of finding the roots

Example 40. *find the roots of the function $f(x) = x^2 - 3x + 1$. (we know they are $x = 1.5 \pm \sqrt{1.25} = 2.618034, 0.381966$.) define f as $f_1(x) = \frac{1}{3}(x^2+1)$ and try $x_0 = 1$. iterating produces a sequence $x_1 = 0.667$, $x_2 = 0.481$, $x_3 = 0.411$, $x_4 = 0.390$ which is converging to 0.381966. commencing with $x_0 = 3$ produces $x_1 = 3.333$, $x_2 = 4.037$, $x_3 = 5.766$, $x_4 = 11.414$ which is diverging.*

Example 41. *having another try, we can define f as $f_2(x)$ as $3 - \frac{1}{x}$. then, with the same starting conditions as before we get, $x_0 = 1$, $x_1 = 2$,$x_2 = 2.5$, $x_3 = 2.6$, $x_4 = 2.615$ and also $x_0 = 3$, $x_1 = 2.667$, $x_2 = 2.625$, $x_3 = 2.619$, $x_4 = 2.618$, both of which converge, but to the same solution.*

it is clear from the example above that the convergence in these methods is critically dependent on the choice of f and the initial value x_0. also the *derivative* of f in the neighbourhood of the critical point is crucial. it appears that when the slope of $f(x)$ is less steep than the slope of $x = f(x) = 1$ then we have convergence, and indeed this is easy to prove.

convergence of fixed point iteration

Theorem 23 (convergence of fixed point iteration)**.** *let $x = s$ be a solution of $f(x) = x$ and suppose that f has a continuous derivative on some interval i containing s. then if $|f'| \le \alpha < 1$ in i the iteration process defined by $x_{n+1} = f(x_n)$ converges for any x_0 in i.*

proof: by the mean value theorem of calculus there is some t between x and s such that

$$f(x) - f(s) = f'(t)(x-s).$$

so

$$\begin{aligned}
|x_n - s| &= |f(x_{n-1}) - f(s)| \\
&= |f'(t)||x_{n-1} - s| \\
&\le \alpha|x_{n-1} - s| \\
&\le \alpha^2|x_{n-2} - s| \\
&\le \alpha^n|x_0 - s|
\end{aligned}$$

since $\alpha < 1$ the $\alpha^n \to 0$ and so $|x_n - s| \to 0$ as n approaches infinity. so for arbitrary $x_0 \in i$, the sequence $\{x_n\}$ defined by $x_{n+1} = f(x_n)$ converges to the fixed point s of f.

lets reconsider the previous examples where $f_1(x) = \frac{1}{3}(x^2+1)$ and $f_2(x) = 3 - \frac{1}{x}$.

now $f_1' = 2/3x := \alpha_1$. the mapping is contractive if $|\alpha_1| < 1$ or $|x| < 3/2$ and thus $i = (-3/2, 3/2)$. observe that the example converged with the initial guess $x_0 = 1$ but not when the initial guess was $x_0 = 3$.

with f_2, $f_2' = x^{-2} := \alpha_2$. this time $|\alpha_2| < 1$ when $|x| > 1$ so $i = (-\infty, -1] \cup [1, \infty)$. this example converged for both initial guess $x_0 = 1$ and $x_0 = 3$.

Bibliography