# Numerical Methods for Linear Algebra

Emanuela Bianchi

## Contents

# 1 Vector and Matrix Operations

Before we begin to study the solution of linear systems, let us take a look at some simpler computations.

## 1.1 Multiplying a Matrix by a Vector

Given an input matrix $A$ with $m$ rows and $n$ columns and an input vector $\vec{x}$ of dimension $n$, we compute the vector $\vec{y}$ of dimension $m$

$$\mathbf{A}\vec{x} = \vec{y} \quad \Longleftrightarrow \quad \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \tag{1}$$

We can implement the operation in two ways.
**Row-oriented operation**

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \tag{2}$$

We run $m$ threads, and each thread computes the $i$-th component of $\vec{y}$ by computing the dot product between the $i$-th row of $A$ (a vector of dimension $n$) and vector $\vec{x}$. Each entry of $A$ is read only one time, but each component of $\vec{x}$ is read $m$ times from the global memory.
A computer code to perform the matrix-vector multiplication by row looks something like this

```
do i=1,m
  do j=1,n
    y(i) = y(i) + a(i,j)*x(j)
  end do
end do
```

**Column-Oriented Operation**

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} x_1 + \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} x_2 + \cdots + \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix} x_n \tag{3}$$

A computer code to perform the matrix-vector multiplication by column looks something like this

```
do j=1,n
  do i=1,m
    y(i) = y(i) + a(i,j)*x(j)
  end do
end do
```

A simple code in fortran90 where both algorithms are implemented would show that the column-oriented implementation is faster than the row-oriented one, as shown in Figure 1.

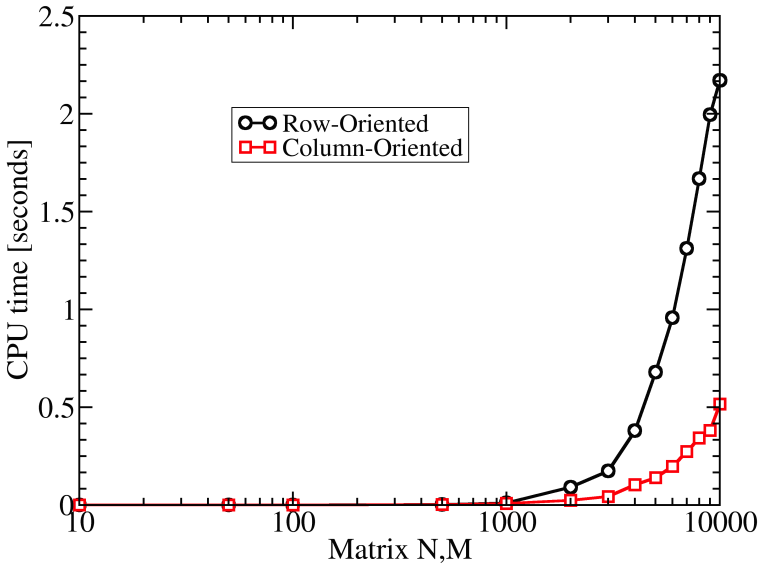



Figure 1: Comparison between the CPU-time consumption of a row-oriented and a column-oriented matrix-vector multiplication for a square matrix $A$ where $n = m$, on a 1.4 GHz Intel Core i5 Processor.

To understand Figure 1, we must introduce two concepts:

- **Number of flops.** As real numbers are normally stored in computers in a floating-point format, the arithmetic operations that a computer performs on these numbers are called floating-point operations or flops, for short.The traditional way to estimate running time is to count how many flops the computer must perform. The update $y_i = y_i + a_{ij}x_j$ involves two flops: one multiplication and one addition. As in the row-oriented implementation, the outer loop is executed $m$ times and the inner loop (that involves two flops) is executed $n$ times (vice versa for the column-oriented implementation), the number of flops is $2nm$. The flop count gives us a useful first indication of an algorithm's operation time, but the execution speed of an algorithm can be affected drastically by how the memory traffic is organized.

- **Memory organization.** Every computer has a memory hierarchy: (large and slow) memory → (small and fast) cache → registers (computations). Data stored in the main memory must first be moved to the cache and then to the registers before they can be operated on. The transfer from the main memory to the cache is much slower than that from the cache to the registers, and it is also slower than the rate at which the computer can perform arithmetic.

  Modern chips can perform floating point arithmetic operations much more quickly than they can fetch data from the memory. A matrix is a 2d array of elements, but the memory addresses are 1d. In C the default storage is by row, while in Fortran the default is by column, that is

  $$A = (a_{11}, a_{21}, ..., a_{m1}, a_{12}, a_{22}, ..., a_{m2}, ....., a_{1n}, a_{2n}, ..., a_{mn}).$$

  Caches are organized to take advantage of spatial locality (use of adjacent memory locations in a short period of program execution) and temporal locality (re-use of the same memory location in a short period of program execution).

So, even if the number of flops is $2nm$ for both row- and column-oriented matrix-vector multiplications, the column-oriented implementation runs faster because of the way the memory is accessed.

## 1.2 Multiplying a Matrix by a Matrix

Given two input matrices $A$ with $m$ rows and $r$ columns and $B$ with $r$ rows and $n$ columns, we compute the matrix $C$ of dimension $mxn$

$$C_{\mathrm{mxn}} = A_{\mathrm{mxr}} B_{\mathrm{rxn}} \tag{4}$$

$$\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1r} \\ a_{21} & a_{22} & \dots & a_{2r} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mr} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{r1} & b_{r2} & \dots & b_{rn} \end{pmatrix} \tag{5}$$

The $ij$-element of C is given by

$$c_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj} \tag{6}$$

A computer code to perform a matrix-matrix multiplication looks something like this

```
do i=1,m
  do j=1,n
    do k=1,r
      c(i,j) = c(i,j)+a(i,k)*b(k,j)
    end do
```

```
  end do
end do
```

The number of flops is $2mrn$. Thus when all matrices are square of dimension $nxn$ the flop counts is $\approx O(n^3)$. This function grows rather quickly with $n$: each time $n$ is doubled, the flops count is multiplied by eight. We know, though, that the number of flops is not the whole story. If we can move the entire arrays $A, B$, and $C$ into cache, then perform all of the operations, then move the result back to main memory, we expect the job to be done much more quickly than if we must repeatedly move data back and forth. Unfortunately, unless $n$ is quite small, the cache will not be large enough to hold the entire matrices. Then it becomes beneficial to perform the matrix-matrix multiplication by blocks.

### 1.3 Use Block Matrix Operations

The idea of partitioning matrices into blocks to decrease data movement is simple but powerful. If $A, B$ and $C$ are partitioned into blocks as in the following

$$A = \begin{array}{c} \\ m_1 \\ \vdots \\ m_p \end{array} \begin{array}{c} \begin{array}{ccc} r_1 & \dots & r_s \end{array} \\ \begin{pmatrix} A_{11} & \dots & A_{1s} \\ \vdots & \vdots & \vdots \\ A_{p1} & \dots & A_{ps} \end{pmatrix} \end{array} \tag{7}$$

where $m = m_1 + \cdots + m_p$ and $r = r_1 + \cdots + r_s$,

$$B = \begin{array}{c} \\ r_1 \\ \vdots \\ r_s \end{array} \begin{array}{c} \begin{array}{ccc} n_1 & \dots & n_q \end{array} \\ \begin{pmatrix} B_{11} & \dots & B_{1q} \\ \vdots & \vdots & \vdots \\ B_{s1} & \dots & B_{sq} \end{pmatrix} \end{array} \tag{8}$$

where $r = r_1 + \cdots + r_s$ and $n = n_1 + \cdots + n_q$,

$$C = \begin{array}{c} \\ m_1 \\ \vdots \\ m_p \end{array} \begin{array}{c} \begin{array}{ccc} n_1 & \dots & n_q \end{array} \\ \begin{pmatrix} C_{11} & \dots & C_{1q} \\ \vdots & \vdots & \vdots \\ C_{p1} & \dots & C_{pq} \end{pmatrix} \end{array} \tag{9}$$

where $m = m_1 + \cdots + m_p$ and $n = n_1 + \cdots + n_q$, then

$$C_{ij} = \sum_{k=1}^{s} A_{ik} B_{kj} \tag{10}$$

where $i = 1, ..., p$ and $j = 1, ..., q$.

For simplicity, we now assume that all matrices are square matrices of dimensions $n$x$n$. We further assume that $A$ can be partitioned into $s$ block rows and $s$ block columns, where each block is $r$x$r$

$$A = \begin{pmatrix} A_{11} & \dots & A_{1s} \\ \vdots & \vdots & \vdots \\ A_{s1} & \dots & A_{ss} \end{pmatrix} \tag{11}$$

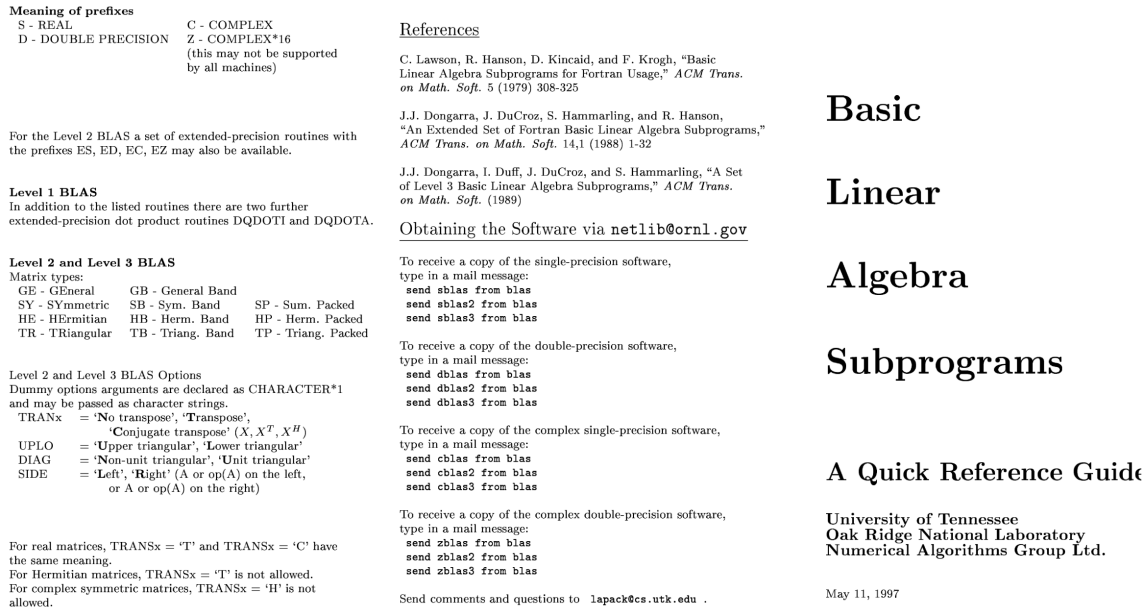
and we partition $B$ and $C$ in the same way.
A computer program based on this layout looks something like this

```
do i=1,s
  do j=1,s
    do k=1,s
      C(i,j) = C(i,j)+A(i,k)*B(k,j)
    end do
  end do
end do
```

It performs the following operations repeatedly: grabs the blocks $A_{ik}$, $B_{kj}$ and $C_{ij}$, multiplies $A_{ik}$ by $B_{kj}$ and adds the result to $C_{ij}$, then sets $C_{ij}$ aside. Varying the block size does not affect the flop counts (the block version of the algorithm has again $O(n^3)$ flops for square matrices of dimensions $n$x$n$), but it can affect the performance of the algorithm because of the way data are handled.
If we do not use blocks and the cache is big enough to hold two matrix columns or rows, the computation goes like as in the following. First, the computation of the entry $c_{ij}$ requires the $i$-th row of $A$ and the $j$-th column of $B$. The time required to move these into the cache is proportional to $2n$, that is the number of data items. Once these are in the fast memory, we can quickly perform the $2n$ flops. If we now want to calculate $c_{i,j+1}$, we can keep the $i$-th row of $A$ in the cache, but we need to bring in the column $j+1$ of B, which means we have a time delay proportional to $n$. We can then perform the $2n$ flops to get $c_{i,j+i}$. So, without blocks, the number of transfers between the main memory and the cache is proportional to the amount of arithmetic done. Now, suppose we use a block size $r$ that is small enough that the three blocks $A_{ik}$, $B_{kj}$ and $C_{ij}$ can all fit into cache at once. The time to move these blocks from the main memory to the cache is proportional to $3r^2$, the number of data items. Once they are in the fast memory, the $2r^3$ flops that the matrix-matrix multiplication requires can be performed relatively quickly. Thus, the larger the blocks are, the less significant the data transfers become. The upper limit to the value of $r$ is of course given by the cache size.
Note that if the computer has multiple processors, it can operate on several blocks in parallel.
The public-domain linear algebra subroutine libraries use block algorithms wherever they can.

### 1.4 BLAS (Basic Linear Algebra Subprograms)

The BLAS are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations (see Figure 2). Note that, the BLAS are used in LAPACK (see section 2.2.5).
A simple fortran90 code where a home-made matrix-matrix multiplication is compared to the use of, e.g., the DGEMM routine (D=double precision, GE= generic matrices, MM = matrix-matrix multiplication) from BLAS would show how the calculation speeds up with the optimization.



**Meaning of prefixes**
S - REAL
D - DOUBLE PRECISION
C - COMPLEX
Z - COMPLEX*16 (this may not be supported by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

**Level 1 BLAS**
In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

**Level 2 and Level 3 BLAS**
Matrix types:
GE - GEneral, GB - General Band
SY - SYmmetric, SB - Sym. Band, SP - Sum. Packed
HE - HErmitian, HB - Herm. Band, HP - Herm. Packed
TR - TRiangular, TB - Triang. Band, TP - Triang. Packed

Level 2 and Level 3 BLAS Options
Dummy options arguments are declared as CHARACTER*1 and may be passed as character strings.
TRANx = '**N**o transpose', '**T**ranspose', '**C**onjugate transpose' ($X, X^T, X^H$)
UPLO = '**U**pper triangular', '**L**ower triangular'
DIAG = '**N**on-unit triangular', '**U**nit triangular'
SIDE = '**L**eft', '**R**ight' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANSx = 'T' and TRANSx = 'C' have the same meaning.
For Hermitian matrices, TRANSx = 'T' is not allowed.
For complex symmetric matrices, TRANSx = 'H' is not allowed.

References

C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. on Math. Soft.* 5 (1979) 308-325

J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* 14,1 (1988) 1-32

J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* (1989)

Obtaining the Software via `netlib@ornl.gov`

To receive a copy of the single-precision software, type in a mail message:
```
send sblas from blas
send sblas2 from blas
send sblas3 from blas
```

To receive a copy of the double-precision software, type in a mail message:
```
send dblas from blas
send dblas2 from blas
send dblas3 from blas
```

To receive a copy of the complex single-precision software, type in a mail message:
```
send cblas from blas
send cblas2 from blas
send cblas3 from blas
```

To receive a copy of the complex double-precision software, type in a mail message:
```
send zblas from blas
send zblas2 from blas
send zblas3 from blas
```

Send comments and questions to `lapack@cs.utk.edu` .

**Basic**

**Linear**

**Algebra**

**Subprograms**

**A Quick Reference Guide**

**University of Tennessee**
**Oak Ridge National Laboratory**
**Numerical Algorithms Group Ltd.**

May 11, 1997

Figure 2: BLAS quick reference guide.

## 2 Systems of Linear Equations

Given the matrix $A_{m\text{x}n}$ and the vector $\vec{b}$ we want to find the vector $\vec{x}$ such that

$$A\vec{x} = \vec{b} \quad \Longleftrightarrow \quad \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \tag{12}$$

- if $m \leq n$ and $\det A = 0$, then the system is singular/undetermined and cannot be uniquely solved;
- if $m = n$ and $\det A \neq 0$, then the system is non-singular/solvable;
- if $m > n$ and $\det A \neq 0$, then the system is overdetermined (unless $m - n$ equations can be reduced by linear dependencies).

In the following, we focus on systems of $n$ linear equations for $n$ unknowns (*i.e.* we consider $m = n$) where $A$ is a $nxn$ matrix with a non-zero determinant. This system has a unique solution, whose individual values for the unknowns $\{x_i\}$ are given by Cramer's rule (we remind the reader that Cramer's rule is an explicit formula for the solution of a system of linear equations with as many equations as unknowns that is valid whenever the system has a unique solution)

$$x_i = \frac{\det(A_i)}{\det(A)} \qquad \text{with} \qquad i = 1, ..., n \tag{13}$$

where $A_i$ is the matrix formed by replacing the $i$-th column of $A$ by the column $\vec{b}$. Cramer's rule implemented in a naive way requires computation of $n + 1$ determinants and is computationally inefficient for $n \geq 3$.
Please, note that the methods to solve a system of linear equations can also be used to determine also the inverse matrix $A^{-1}$.

## 2.1 Numerical Aspects

Depending on the form of $A$, there are a large number of special methods for solving systems of linear equations, which can be divided into (i) direct and (ii) iterative processes. Direct processes deliver the exact result in a finite number of steps, apart from rounding errors. With iterative processes, the exact result can only be achieved after an infinite number of steps: starting from an approximation to the starting point, the solution sought is gradually approximated until a defined convergence criterion is reached.
Here we limit ourselves to the presentation of

- three direct methods, namely (a) the forward/backward substitution for triangular matrices, (b) the Gauss elimination algorithm for generic matrices and (c) the Thomas method for tridiagonal matrices
- two iterative methods, namely (a) the Jacobi and (b) the Gauss-Seidel methods and their generalizations.

## 2.2 Direct Methods

### 2.2.1 Triangular systems of equations

Let us start with some special cases. The simplest one would be the case where $A$ is a diagonal matrix, so that the system breaks down into $n$ independent scalar equations. Triangular systems of linear equations are somewhat more interesting. A linear system whose coefficient matrix is triangular is particularly easy to solve. For this reason, it is a common practice to reduce general systems to a triangular form, which can then be solved inexpensively.
A triangular matrix is one that is either upper or lower triangular. A matrix $A$ is lower triangular if $a_{ij} = 0$ whenever $i < j$, *i.e.*

$$A = \begin{pmatrix} a_{11} & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & 0 & \dots & 0 \\ a_{31} & a_{32} & a_{33} & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix} \tag{14}$$

Similarly, an upper triangular matrix is one for which $a_{ij} = 0$ whenever $i > j$. We recall that if $A$ is triangular, then $\det(A) = a_{11}a_{12}...a_{nn}$. Thus $\det(A) \neq 0$ if and only if $a_{ij} \neq 0$ for $i = 1, ..., n$.
Now the system $A\vec{x} = \vec{b}$, where $A$ is a non-singular, lower-triangular matrix, can be written in detail as

$$\begin{aligned} a_{11}x_1 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \\ \vdots &= \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + ... + a_{nn}x_n &= b_n \end{aligned} \tag{15}$$

The first equation involves only the unknown $x_1$, thus

$$x_1 = b_1/a_{11} \tag{16}$$

where $a_{11} \neq 0$ as $A$ is non-singular; the second equation involves only $x_1$ and $x_2$, thus

$$x_2 = (b_2 - a_{21}x_1)/a_{22} \tag{17}$$

where $a_{22} \neq 0$ as $A$ is non-singular; and so on. In general, once we have $x_1$, $x_2$, ..., $x_{i-1}$, we can solve for $x_i$ using

$$x_i = \frac{b_i - a_{i1}x_1 - a_{i2}x_2 - ... - a_{i,i-1}x_{i-1}}{a_{ii}} \tag{18}$$

or equivalently in a more compact notation

$$x_i = a_{ii}^{-1} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right) \tag{19}$$

where $a_{ii} \neq 0$ as $A$ is non-singular. This equation describes the algorithm completely; it even describes the first step as when the lower limit of a sum is greater than the upper limit, the sum is zero.
The algorithm for solving a lower-triangular system is called **forward substitution**. The one expressed in Eq. 19 is called row-oriented forward substitution because it accesses $A$ by rows: the $i$-th row is used at the $i$-th step. Before we write down the algorithm, notice that $b_1$ is only used in calculating $x_1$, $b_2$ is only used in calculating $x_2$, and so on. In general, once we have calculated $x_i$, we no longer need $b_i$. It is therefore usual for a computer program to store $\vec{x}$ over $\vec{b}$. Thus we have a single array that contains $\vec{b}$ before the program is executed and $\vec{x}$ afterward. The algorithm looks like this

```
do i=1, n
  do j=1, i-1
    b(i) = b(i)-a(i,j)*b(j)
  end do
  if(a(i,i).ne.0)then
    b(i)=b(i)/a(i,i)
  else
    write(*,*) "error: singular matrix"
  end if
end do
```

Note that (i) there are no references to $\vec{x}$, since it is stored in the array named $\vec{b}$ and (ii) the check of $a_{ii}$ is included to make the program foolproof. It is a good practice to check before each division that the divisor is not zero. In most linear algebra algorithms these checks do not contribute significantly to the time it takes to run the program, because the division operation is executed relatively infrequently.
To get an idea of the execution time of the forward substitution algorithm, let us count the number of flops: the outer loop is performed $n$ times and the inner loops is performed $n-1$ times, in the inner loop the number of flops is 2, thus the number of flops is $2n(n-1)$:

$$N_{flops} = \sum_{i=1}^{n} \sum_{j=1}^{i-1} 2 = 2 \sum_{i=1}^{n} (i-1) = 2n(n-1) \tag{20}$$

while in the if-cycle there are $n$ operations, so at large $n$ this loop is not significant. The cost of a forward substitution is thus $\approx O(n^2)$ flops.

A column-oriented implementation of the forward substitution is also possible. Which of the two versions of forward substitution is superior? The answer depends on how $A$ is stored and accessed. This, in turn, depends on the programmer's choice of the data structure and on the programming language and on the architecture of the computer.
Upper-triangular systems can be solved in much the same way as lower-triangular systems. In this case, we solve the system from bottom to top

$$x_i = a_{ii}^{-1} \left( b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right) \tag{21}$$

The process is called **backward substitution**, and it has row- and column-oriented versions as well. The cost of backward substitution is obviously the same as that of forward substitution, *i.e.* $\approx O(n^2)$ flops.
It is easy to develop block variants of both forward and backward substitution. The block version of the row-oriented forward substitution is, for instance,

```
do i=1, s
  do j=1, i-1
    b(i) = b(i)-A(i,j)*b(j)
  end do
  if(A(i,i).ne.0)then
    b(i)=b(i)/A(i,i)
  else
    write(*,*) "error: singular matrix"
  end if
end do
```

where $s$ is the maximum value for the indices of the sub-matrices $A_{ij}$.

### 2.2.2 Gauss elimination method

Gauss elimination method is an algorithm in linear algebra for solving a system of linear equations, but it can also be used to find the rank of a matrix, to calculate the determinant of a matrix and to calculate the inverse of an invertible square matrix. The method is named after Carl Friedrich Gauss (1777-1855), even though some special cases of the method – albeit presented without proof – were known to Chinese mathematicians as early as circa 179 AD.
The Gauss elimination uses a sequence of elementary row operations (1. swapping two rows, 2. multiplying a row by a nonzero number and 3. adding a multiple of one row to another row) to modify the matrix into a triangular shape. The generation of an equivalent linear system of equations with a triangular shape is an elementary process that can be easily achieved in $n-1$ steps for a square matrix $A$ of dimension $n$x$n$.

We start from

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &=& b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n &=& b_2 \\ \vdots \quad \vdots \quad \vdots \quad \vdots &=& \vdots \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n &=& b_n \end{array} \tag{22}$$

and we want to transform it into an upper triangular system. The first line does not have to be changed; all other lines must be manipulated so that the coefficient before $x_1$ disappears

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &=& b_1 \\ + \quad a'_{22}x_2 + ... + a'_{2n}x_n &=& b'_2 \\ \vdots \quad \vdots &=& \vdots \\ + \quad a'_{n2}x_2 + ... + a'_{nn}x_n &=& b'_n \end{array} \tag{23}$$

and then we recursively apply the same transformation to the last $n-1$ lines to finally get a triangular system. It is therefore sufficient to examine the first elimination step from Eq. 22 to Eq. 23 . Let $a_{11} \neq 0$, then to eliminate the term $a_{i1}x_1$ in the $i$-th line, we subtract a multiple of the (unchanged) 1st line from the $i$-th line

$$\text{new Row } i = \text{Row } i - l_{i1} \text{ Row 1} \tag{24}$$

$$\underbrace{(a_{i1} - l_{i1}a_{11})}_{=0} x_1 + \underbrace{(a_{i2} - l_{i1}a_{12})}_{=a'_{i2}} x_2 + ... + \underbrace{(a_{in} - l_{i1}a_{1n})}_{=a'_{in}} x_n = \underbrace{b_i - l_{i1}b_1}_{=b'_i} \tag{25}$$

From which it follows that $l_{i1} = a_{i1}/a_{11}$. The element $a_{11}$ is called a pivot element and the first line is called a pivot line. After this first elimination step, an $(n-1, n-1)$ sub-matrix remains in rows 2 to $n$. We are now in the same situation as when we started, but one dimension lower. So we can write

$$\text{new Row } i = \text{Row } i - \frac{a'_{i2}}{a'_{22}} \text{ Row 2} \tag{26}$$

where $i$ goes from 3 to $n$. By using this elimination method recursively we get a sequence of matrices $A^{(k)}$ with the form

$$A^{(k)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & \dots & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \dots & \dots & \dots & a_{2n}^{(2)} \\ \vdots & 0 & \ddots & & & \vdots \\ \vdots & \vdots & 0 & a_{kk}^{(k)} & \dots & a_{kn}^{(k)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & a_{nk}^{(k)} & \dots & a_{nn}^{(k)} \end{pmatrix} \tag{27}$$

with a $(n-k+1, n-k+1)$ sub-matrix, the so-called residual matrix, in the lower right corner.
To go from $A^{(k)}\vec{x} = \vec{b}^{(k)}$ to $A^{(k+1)}\vec{x} = \vec{b}^{(k+1)}$, we apply the following elimination step to the residual matrix

$$\begin{aligned} l_{ik} &= a_{ik}^{(k)}/a_{kk}^{(k)} && \text{for } i = k+1, ..., n \\ a_{ij}^{(k+1)} &= a_{ij}^{(k)} - l_{ik}a_{kj}^{(k)} && \text{for } i, j = k+1, ..., n \\ b_j^{(k+1)} &= b_i^{(k)} - l_{ik}b_k^{(k)} && \text{for } i = k+1, ..., n \end{aligned} \tag{28}$$

as the pivot element $a_{kk}^{(k)}$ is not zero.
The decomposition of the matrix $A$ in a triangular matrix can thus be written as

$$A\vec{x} = \vec{b} \Longrightarrow LU\vec{x} = \vec{b} \tag{29}$$

where the multiplication factors $l_{ik}$ form a lower-triangular matrix

$$L = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & \dots & \vdots \\ \vdots & \vdots & \vdots & \dots & 0 \\ l_{n1} & l_{n2} & \dots & l_{nn-1} & 1 \end{pmatrix} \tag{30}$$

and

$$U = \begin{pmatrix} u_{11} & u_{12} & \dots & \dots & u_{1n} \\ 0 & u_{22} & \dots & \dots & u_{2n} \\ 0 & 0 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & u_{nn} \end{pmatrix} \tag{31}$$

This is known as $LU$-factorization as it factors a matrix as the product of a lower ($L$) triangular matrix and an upper ($U$) triangular matrix. Gauss elimination process implies the following operations

- $A = LU$, $A$ is decomposed into a lower and an upper triangular matrix
- $L\vec{z} = \vec{b}$ is solved by forward substitution
- $U\vec{x} = \vec{z}$ is solved by backward substitution

A memory-saving storage method can be implemented by overwriting the original matrix A with the matrices $A^{(k)}$ where in the lower triangular part the $l_{ik}$'s are stored, as the other elements of $U$ and $L$ are known (they are either 0 or 1). The computing effort for

the triangular decomposition is $\approx O(n^3)$ flops because two operations are performed per loop over a kxk matrix with $k = 1, \ldots, n$, *i.e.*

$$N_{flops} = 2n^2 + 2(n-1)^2 + 2(n-2)^2 + \cdots = 2\sum_{k=1}^{n} k^2 \approx \frac{2}{3}n^3 \tag{32}$$

while the forward and backward substitution requires $\approx O(n^2)$ flops. The leading contribution between $\approx O(n^3)$ and $\approx O(n^2)$ thus is $N_{flops} = O(n^3)$.

### 2.2.3 Pivot search and influence of rounding errors

Difficulties may arise in the numerical implementation of the Gauss elimination method when pivot elements disappear or when they are too small with respect to the computer precision.
The following example shows that there are cases in which the $LU$-factorization cannot be carried out even though $\det A \neq 0$

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \ \det A = -1, \ a_{11} = 0 \tag{33}$$

As the matrix is not singular, we can search for a pivot element that is not zero. Exchanging the rows leads to

$$A^{(1)} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \mathbb{1} = LU \text{ where } L = U = \mathbb{1} \tag{34}$$

The following example shows that the pivot search improves the numerical accuracy

$$A = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \tag{35}$$

If within our numerical accuracy $\left(1 - \frac{1}{\epsilon}\right) \approx -\frac{1}{\epsilon}$, the $LU$-factorization of the matrix $A$ gives a wrong result

$$LU = \begin{pmatrix} 1 & 0 \\ \frac{1}{\epsilon} & 1 \end{pmatrix} \begin{pmatrix} \epsilon & 1 \\ 0 & 1 - \frac{1}{\epsilon} \end{pmatrix} \approx \begin{pmatrix} 1 & 0 \\ \frac{1}{\epsilon} & 1 \end{pmatrix} \begin{pmatrix} \epsilon & 1 \\ 0 & -\frac{1}{\epsilon} \end{pmatrix} = \begin{pmatrix} \epsilon & 1 \\ 1 & 0 \end{pmatrix} \neq A \tag{36}$$

If we first exchange the lines of $A$ and then perform the $LU$-factorization, within the same numerical accuracy we obtain the correct result

$$LU = \begin{pmatrix} 1 & 0 \\ \epsilon & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 - \epsilon \end{pmatrix} \approx \begin{pmatrix} 1 & 0 \\ \epsilon & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} \equiv A \tag{37}$$

The choice of the pivot is thus crucial for the minimization of the rounding errors. Let us consider a numeric example. We have the system

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \tag{38}$$

where $\epsilon = 10^{-4}$. The exact solution with a 4 decimals accuracy is $x_1 = 1.0$ and $x_2 = 0.9999$, while with a 3 decimals accuracy the correct result is $x_1 = 1.0$ and $x_2 = 1.0$. If we assume a 3 decimals accuracy, then $\left(1 - \frac{1}{\epsilon}\right) \approx -10^4$ and $(1 - \epsilon) \approx 1$.
If we perform the $LU$-factorization without pivoting, we obtain

$$\begin{pmatrix} \epsilon & 1 \\ 0 & 1 - \frac{1}{\epsilon} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 - \frac{1}{\epsilon} \end{pmatrix} \tag{39}$$

which gives us $x_1 = 0.0$ (wrong) and $x_2 = 1.0$ (correct).
If we now perform the $LU$-factorization with pivoting, we obtain

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 - \epsilon \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 - 2\epsilon \end{pmatrix} \tag{40}$$

which gives us $x_1 = 1.0$ (correct) and $x_2 = 1.0$ (correct).
The example above could give the impression that the selection by columns of the largest element as pivot leads to the smallest rounding error. However, this is generally not the case, since it depends on the residual matrix involved. In addition to the column pivot search as discussed above, there is also the total pivot search. In the latter, the absolutely largest matrix element of the entire residual matrix is used as the pivot element. The total pivot search is, however, much more complex in terms of programming and requires not only swapping the rows but also the columns, thus we focus here on the partial pivot search by columns.

### 2.2.4 Pivot search by columns

The Gauss elimination method with column pivoting, implies that in the elimination step from $A^{(k)}$ to $A^{(k+1)}$

- we first search by columns for a suitable pivot element , *i.e.* we look for the element with the largest absolute value from the $k$-th sub-column $a_{jk}^{(k)}$ for $j = k, ..., n$; if this element is in the $r$-th row, then the pivot element is $a_{rk}^{(k)}$ (while, if there is no $a_{jk}^{(k)} \neq 0$, then the sub-matrix $A^{(k)}$ is singular and there is no solution)
- we swap the $r$-th with the $k$-th row in $A^{(k)}$ and $\vec{b}^{(k)}$
- as the pivot element $a_{kk}^{(k)}$ is not zero, we proceed with the Gauss elimination step

The formal description of $LU$-factorization with partial pivoting makes use of the permutation matrix $P$, that is obtained by exchanging the rows or columns of the identity matrix during pivoting: for each invertible square matrix $A$ there is a permutation matrix $P$, so that a $LU$-factorization of the shape $PA = LU$ is possible.

**Example of $LU$-factorization with pivoting.**

Let us consider the matrix

$$A = \begin{pmatrix} 1 & 6 & 1 \\ 2 & 3 & 2 \\ 4 & 2 & 1 \end{pmatrix} \tag{41}$$

The pivot element of the first column is 4, so we have to swap the 1st and 3rd rows. The first elimination step delivers

$$\begin{pmatrix} 4 & 2 & 1 \\ 2 & 3 & 2 \\ 1 & 6 & 1 \end{pmatrix} \quad \rightarrow \quad \begin{pmatrix} 4 & 2 & 1 \\ 0 & 2 & \frac{3}{2} \\ 0 & \frac{11}{2} & \frac{3}{4} \end{pmatrix} \tag{42}$$

where $l_{21} = a_{21}^{(1)}/a_{11}^{(1)} = 1/2$ and $l_{31} = a_{31}^{(1)}/a_{11}^{(1)} = 1/4$. The pivot element of the second column is 2, so we have to swap the 2nd and 3rd rows. The second elimination step delivers

$$\begin{pmatrix} 4 & 2 & 1 \\ 0 & \frac{11}{2} & \frac{3}{4} \\ 0 & 2 & \frac{3}{2} \end{pmatrix} \quad \rightarrow \quad \begin{pmatrix} 4 & 2 & 1 \\ 0 & \frac{11}{2} & \frac{3}{4} \\ 0 & 0 & \frac{27}{22} \end{pmatrix} \tag{43}$$

where $l_{32} = a_{32}^{(2)}/a_{22}^{(2)} = 4/11$. Thus the resulting $LU$-factorization is

$$LU = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{4} & \frac{4}{11} & 1 \end{pmatrix} \begin{pmatrix} 4 & 2 & 1 \\ 0 & \frac{11}{2} & \frac{3}{4} \\ 0 & 0 & \frac{27}{22} \end{pmatrix} \tag{44}$$

for the matrix

$$PA = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 6 & 1 \\ 2 & 3 & 2 \\ 4 & 2 & 1 \end{pmatrix} \tag{45}$$

**Example of a linear system of equations solved by Gauss elimination.**
Let us consider the system of linear equations $A\vec{x} = \vec{b}$, where the matrix $A$ is the same as in the previous example and $\vec{b} = (1, 2, 3)^T$, namely

$$\begin{pmatrix} 1 & 6 & 1 \\ 2 & 3 & 2 \\ 4 & 2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{46}$$

If the system is solved **without pivoting**, the $LU$-factorization $A = LU$ is

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & \frac{22}{9} & 1 \end{pmatrix} \begin{pmatrix} 1 & 6 & 1 \\ 0 & -9 & 0 \\ 0 & 0 & -3 \end{pmatrix} \tag{47}$$

and the system $U\vec{x} = \vec{z}$

$$\begin{pmatrix} 1 & 6 & 1 \\ 0 & -9 & 0 \\ 0 & 0 & -3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} \tag{48}$$

is solved by backward substitution resulting into

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \frac{2}{3} \\ 0 \\ \frac{1}{3} \end{pmatrix} \tag{49}$$

Note that the system $L\vec{z} = \vec{b}$

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & \frac{22}{9} & 1 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{50}$$

is solved by forward substitution and the $\vec{z}$ is again

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} \tag{51}$$

If the system is solved **with pivoting**, the $LU$-factorization $A = LU$ is

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{4} & \frac{4}{11} & 1 \end{pmatrix} \begin{pmatrix} 4 & 2 & 1 \\ 0 & \frac{11}{2} & \frac{3}{4} \\ 0 & 0 & \frac{27}{22} \end{pmatrix} \tag{52}$$

and the system $U\vec{x} = \vec{z}$

$$\begin{pmatrix} 4 & 2 & 1 \\ 0 & \frac{11}{2} & \frac{3}{4} \\ 0 & 0 & \frac{27}{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ \frac{1}{4} \\ \frac{9}{22} \end{pmatrix} \tag{53}$$

is solved by backward substitution resulting into (49).

### 2.2.5 LAPACK (Linear Algebra PACKage)

LAPACK (see Figure 3) provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations are also provided, as are related computations. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision. LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS). The DGETRF subroutine performs, for instance, the $LU$-factorization; while the DGESV subroutine performs, for instance, the Gauss elimination.

# LAPACK Quick Reference Guide to the Driver Routines

**Release 3.0**

## Simple Drivers

**Simple Driver Routines for Linear Equations**

| Matrix Type | Routine | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| General | SGESV( | N, | NRHS, A, LDA, | IPIV, B, LDB, | | | | INFO ) |
| | CGESV( | N, | NRHS, A, LDA, | IPIV, B, LDB, | | | | INFO ) |
| General Band | SGBSV( | N, KL, KU, | NRHS, AB, LDAB, | IPIV, B, LDB, | | | | INFO ) |
| | CGBSV( | N, KL, KU, | NRHS, AB, LDAB, | IPIV, B, LDB, | | | | INFO ) |
| General Tridiagonal | SGTSV( | N, | NRHS, DL, D, DU, | B, LDB, | | | | INFO ) |
| | CGTSV( | N, | NRHS, DL, D, DU, | B, LDB, | | | | INFO ) |
| Symmetric/Hermitian Positive Definite | SPOSV( UPLO, | N, | NRHS, A, LDA, | B, LDB, | | | | INFO ) |
| | CPOSV( UPLO, | N, | NRHS, A, LDA, | B, LDB, | | | | INFO ) |
| Symmetric/Hermitian Positive Definite (Packed Storage) | SPPSV( UPLO, | N, | NRHS, AP, | B, LDB, | | | | INFO ) |
| | CPPSV( UPLO, | N, | NRHS, AP, | B, LDB, | | | | INFO ) |
| Symmetric/Hermitian Positive Definite Band | SPBSV( UPLO, | N, KD, | NRHS, AB, LDAB, | B, LDB, | | | | INFO ) |
| | CPBSV( UPLO, | N, KD, | NRHS, AB, LDAB, | B, LDB, | | | | INFO ) |
| Symmetric/Hermitian Positive Definite Tridiagonal | SPTSV( | N, | NRHS, D, E, | B, LDB, | | | | INFO ) |
| | CPTSV( | N, | NRHS, D, E, | B, LDB, | | | | INFO ) |
| Symmetric/Hermitian Indefinite | SSYSV( UPLO, | N, | NRHS, A, LDA, | IPIV, B, LDB, | WORK, | LWORK, | | INFO ) |
| | CSYSV( UPLO, | N, | NRHS, A, LDA, | IPIV, B, LDB, | WORK, | LWORK, | | INFO ) |
| | CHESV( UPLO, | N, | NRHS, A, LDA, | IPIV, B, LDB, | WORK, | LWORK, | | INFO ) |
| Symmetric/Hermitian Indefinite (Packed Storage) | SSPSV( UPLO, | N, | NRHS, AP, | IPIV, B, LDB, | | | | INFO ) |
| | CSPSV( UPLO, | N, | NRHS, AP, | IPIV, B, LDB, | | | | INFO ) |
| | CHPSV( UPLO, | N, | NRHS, AP, | IPIV, B, LDB, | | | | INFO ) |

**Simple Driver Routines for Standard and Generalized Linear Least Squares Problems**

| Problem Type | Routine | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Solve Using Orthogonal Factor, Assuming Full Rank | SGELS( TRANS, | M, N, NRHS, | A, LDA, B, LDB, | | | WORK, | LWORK, | INFO ) |
| | CGELS( TRANS, | M, N, NRHS, | A, LDA, B, LDB, | | | WORK, | LWORK, | INFO ) |
| Solve LSE Problem Using GRQ | SGGLSE( | M, N, P, | A, LDA, B, LDB, | C, D, X, | | WORK, | LWORK, | INFO ) |
| | CGGLSE( | M, N, P, | A, LDA, B, LDB, | C, D, X, | | WORK, | LWORK, | INFO ) |
| Solve GLM Problem Using GQR | SGGGLH( | N, M, P, | A, LDA, B, LDB, | D, X, Y, | | WORK, | LWORK, | INFO ) |
| | CGGGLH( | N, M, P, | A, LDA, B, LDB, | D, X, Y, | | WORK, | LWORK, | INFO ) |

Figure 3: LAPACK quick reference guide.

### 2.2.6 Tridiagonal systems of equations

Many problems in physics (*e.g.*, the one-dimensional Poisson problem) are described by tridiagonal systems of equations. A tridiagonal system for $n$ unknowns may be written as

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = f_i \tag{54}$$

where $a_1 = 0$ and $c_n = 0$; or equivalently in the matrix form $A\vec{x} = \vec{f}$ as

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & \dots & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & \dots & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & \dots & 0 & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix} \tag{55}$$

Tridiagonal systems of equations can be solved by the Thomas algorithm that is a simplified form of Gauss elimination and requires $\approx O(n)$ flops instead of $\approx O(n^3)$. The equations to be solved are

$$\begin{aligned} b_1 x_1 + c_1 x_2 &= f_1 \quad \text{for} \quad i = 1 \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= f_i \quad \text{for} \quad i = 2, \dots, n-1 \\ a_n x_{n-1} + b_n x_n &= f_n \quad \text{for} \quad i = n \end{aligned} \tag{56}$$

The special structure of the matrix allows for a significant reduction in computing time. First, let us assume a linear recursion rule between the unknowns

$$x_{i+1} = \alpha_i x_i + \beta_i \tag{57}$$

where $\alpha_i$ and $\beta_i$ are to be determined. We now plug Ansatz (57) in the system (56) and obtain for the $i$-th row

$$a_i x_{i-1} + b_i x_i + c_i(\alpha_i x_i + \beta_i) = a_i x_{i-1} + (b_i + c_i \alpha_i) x_i + c_i \beta_i = f_i \tag{58}$$

Solving for $x_i$

$$x_i = -\frac{a_i}{b_i + c_i \alpha_i} x_{i-1} + \frac{f_i - c_i \beta_i}{b_i + c_i \alpha_i} \tag{59}$$

If we compare the coefficients of (57) and (59) term by term, we obtain the following recursion rules

$$\alpha_{i-1} = -\frac{a_i}{b_i + c_i \alpha_i} \quad \text{and} \quad \beta_{i-1} = \frac{f_i - c_i \beta_i}{b_i + c_i \alpha_i} \tag{60}$$

The beginning of the recursion can be obtained from a closer look at the last line

$$a_n x_{n-1} + b_n x_n = f_n \quad \Longrightarrow \quad x_n = -\frac{a_n}{b_n} x_{n-1} + \frac{f_n}{b_n} = \alpha_{n-1} x_{n-1} + \beta_{n-1} \tag{61}$$

thus

$$\alpha_{n-1} = -\frac{a_n}{b_n} \quad \text{and} \quad \beta_{n-1} = \frac{f_n}{b_n} \tag{62}$$

By comparing (60) and (62), we obtain $\alpha_n = \beta_n = 0$. Starting from here, we calculate $\alpha_i$ and $\beta_i$ for $i = n-1, n-2, \ldots, 1$ with the recursion rule (60). With the knowledge of $\alpha_1$ and $\beta_1$, we calculate $x_1$

$$x_1 = \frac{f_1 - c_1 \beta_1}{b_1 + c_1 \alpha_1} \tag{63}$$

Then we can use (57) to find $x_2, x_3, \ldots, x_n$.
Solving the system of equations with a tridiagonal matrix requires $2n$ steps. Starting by $\alpha_n = \beta_n = 0$.
Note that if we have a system of equations with periodic boundary conditions (as in computer simulations of many-body systems) a slightly different form of the tridiagonal system needs to be solved

$$\begin{aligned} a_1 x_n + b_1 x_1 + c_1 x_2 &= f_1 \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= f_i \quad \text{for} \quad i = 2, \ldots, n-1 \\ a_n x_{n-1} + b_n x_n + c_n x_1 &= f_n \end{aligned} \tag{64}$$

for $i = 2, \ldots, n-1$. In this case, we can make use of the Sherman-Morrison formula to avoid the additional operations of Gauss elimination and still use the Thomas algorithm. In other situations, the system of equations may be block tridiagonal (*e.g.*, the 2D Poisson problem). Simplified forms of Gauss elimination can be developed for these situations.
Thomas algorithm is not stable in general, but it is so in several special cases, such as when the matrix is diagonally dominant (either by rows or columns) or symmetric positive definite. If stability is required, Gauss elimination with partial pivoting is recommended instead.
The $LU$-factorization is $A\vec{x} = L(U\vec{x}) = L\vec{z} = \vec{f}$ with

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ l_2 & 1 & 0 & 0 & 0 \\ 0 & l_3 & 1 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & l_n & 1 \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} d_1 & c_1 & 0 & 0 & 0 \\ 0 & d_2 & c_2 & 0 & 0 \\ 0 & 0 & d_3 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & 0 & d_n \end{pmatrix} \tag{65}$$

where $l_i = a_i / d_{i-1}$ and $d_i = b_i - l_i c_{i-1}$ and the recursion starts with $d_1 = b_1$. Thus, we must store three vectors, namely $c_i$, $l_i$ and $1/d_i$ (as multiplication is faster than division).

Note that, only row $k$ is involved in the elimination step from $A^{(k)}$ to $A^{(k+1)}$, as all others rows $j$ with $j = k+1, \ldots, n$ have already $a_{jk} = 0$. $U\vec{x} = \vec{z}$ is then solved by backward substitution, while $L\vec{z} = \vec{f}$ is solved by forward substitution.

**Example: Poisson's equation in one-dimension**
The Poisson equation is a partial differential equation of elliptic type with broad applications. In physics it is for instance used to describe the potential field of a charge or mass density distribution.
In electrostatics, the Poisson equation can be derived by the first and third Maxwell's equations (here in SI units). Let us consider Gauss's law in a linear, isotropic and homogeneous medium

$$\vec{\nabla} \cdot \vec{E} = \frac{\rho}{\epsilon} \tag{66}$$

where $\vec{\nabla}$ is the divergence operator, $\vec{E}$ is the electric field generated by the volume charge density $\rho$ and $\epsilon$ is the dielectric permittivity of the medium. In the absence of a changing magnetic field, $\vec{B}$, Faraday's law of induction gives

$$\vec{\nabla} \mathrm{x} \vec{E} = \frac{\partial \vec{B}}{\partial t} = 0 \tag{67}$$

where $\vec{\nabla}\mathrm{x}$ is the curl operator and $t$ is the time. As the curl of the electric field is zero, we can define a potential $\phi$ such that

$$\vec{E} = -\nabla\phi \tag{68}$$

By substituting (68) in (66), we obtain the Poisson equation for electrostatics

$$\nabla^2\phi = -\frac{\rho}{\epsilon} \tag{69}$$

where $\nabla^2$ is the Laplace operator. The Poisson equation in three dimensions describes the electrostatic potential $\phi(\vec{r})$ generated by a localized charge distribution $\rho(\vec{r})$. The solution of equation to be found is supplemented by initial and/or boundary conditions.
In three dimensions the Laplace operator can be expressed using cartesian coordinates as $\nabla^2 = \partial^2/\partial x + \partial^2/\partial y + \partial^2/\partial z$. We now consider the variation of potential in one of the three independent variables. Equation (69) in one-dimension takes the form

$$u''(x) = -f(x) \tag{70}$$

where $\phi(x) = u(x)$ and $\rho(x)/\epsilon = f(x)$. If we discretize equation (70), it becomes a set of linear equations, as shown in the following.
Let $\Delta x = 1/n$ be a small increment and let $x_0 = 0, x_1 = \Delta x, \ldots, x_n = n\Delta x = 1$ be equally spaced points. Since equation (70) holds at each of these points, we have

$$u''(x_i) = -f(x_i) \quad i = 1, \ldots, n-1 \tag{71}$$

with boundary conditions $u(x_0) = T_0$ and $u(x_n) = T_n$.
A good approximation for the second derivative is given by

$$u''(x_i) \approx \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{\Delta x^2} \tag{72}$$

It is not unreasonable to expect that if we treat the approximations

$$\frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{\Delta x^2} \approx -f(x_i) \tag{73}$$

as equations and solve them exactly, we will get good approximations to the true solution at the mesh points $x_i$. Thus, we let $u_1, \ldots, u_{n-1}$ denote approximations to $u(x_1), \ldots, u(x_{n-1})$ and obtain

$$-u_{i-1} + 2u_i - u_{i+1} = \Delta x^2 f(x_i) \tag{74}$$

Equation (74) is a system of $n-1$ linear equations for the $n-1$ unknowns $u_1, \ldots, u_{n-1}$, so it can be written as a matrix equation $A\vec{u} = \Delta x^2 \vec{f}$

$$\begin{pmatrix} 2 & -1 & 0 & 0 & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ldots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & -1 & 2 & -1 \\ 0 & \ldots & \ldots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \ldots \\ \ldots \\ u_{n-2} \\ u_{n-1} \end{pmatrix} = \Delta x^2 \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \ldots \\ \ldots \\ f(x_{n-2}) \\ f(x_{n-1}) \end{pmatrix} \tag{75}$$

## 2.3 Iterative Methods

In the numerical solution of partial differential equations large sparse matrices often arise. When dealing with sparse matrices, direct methods are too slow: we must thus rely on iterative methods. Also here, we assume that $A$ is non-singular, *i.e.* all its main diagonal entries $a_{ii}$ are not zero.
Iterative methods require an initial guess $x^{(0)}$, a vector in $\mathcal{R}^n$ that approximates the true solution. Once we have $x^{(0)}$, we use it to generate a new guess $x^{(1)}$ which is then used to generate yet another guess $x^{(2)}$, and so on. In this manner we generate a sequence of iterates $(x^{(k)})$ which (we hope) converges to the true solution $x$.
In practice we will not iterate forever. Once $x^{(k)}$ is sufficiently close to the solution (as indicated, *e.g.*, by the magnitude of $||b - Ax^{(k)}||$), we stop and accept $x^{(k)}$ as an adequate approximation to the solution. How soon we stop will depend on the accuracy we need.
The iterative methods that we are going to study do not require a good initial guess. If no good approximation to $x$ is known, we can take $x^{(0)} = 0$. Of course, we should take advantage of a good initial guess if we have one, for then we can get to the solution in fewer iterations than we otherwise would.

The concept of iterative methods can be demonstrated by considering the simple linear equation $(1-a)x = b$. We try to solve this equation with the iteration rule

$$x^{(k+1)} = ax^{(k)} + b \quad \text{where} \quad k = 0, 1, \ldots \tag{76}$$

where $x^{(k)}$ is the value of the solution after the $k$-th step. If you start from any starting value $x^{(0)}$, you get the iteration sequence

$$\begin{aligned} x^{(1)} &= ax^{(0)} + b \\ x^{(2)} &= ax^{(1)} + b = a^2x^{(0)} + ab + b \\ \ldots &= \ldots \\ x^{(k+1)} &= ax^{(k)} + b = a^{k+1}x^{(0)} + (a^k + a^{k-1} + \cdots + a + 1)b \end{aligned} \tag{77}$$

If $|a| < 1$, then

$$\lim_{k\to\infty} a^{k+1} = 0 \quad \text{and} \quad \sum_{k=0}^{\infty} a^k = \frac{1}{1-a} \tag{78}$$

so that by iteration, the solution of the linear equation converges to

$$x = \frac{b}{1-a} \tag{79}$$

In contrast, for $|a| \geq 1$ the iterative procedure given by the rule (76) diverges.
In order to enlarge the convergence range, we modify the iteration rule (76) by carrying out a weighting with the previous iteration step

$$x^{(k+1)} = \omega(ax^{(k)} + b) + (1-\omega)x^{(k)} = (1 - \omega(1-a))x^{(k)} + \omega b \quad \text{where} \quad k = 0, 1, \ldots \tag{80}$$

Assuming any starting value $x^{(0)}$, this iteration sequence converges to

$$x = \frac{\omega b}{1 - [1 - \omega(1-a)]} = \frac{b}{1-a} \tag{81}$$

if $|1 - \omega(1-a)| < 1$, *i.e.*, we have a convergent iteration method if $0 < (1-a)\omega < 2$. The best convergence is obtained if we choose

$$\omega = \frac{1}{1-a} \tag{82}$$

because then $1 - \omega(1-a) = 0$ and the first iteration step already leads to the exact value of the solution.
The transfer of the iteration concept to linear systems of equations is simple. Let us consider a system with three unknowns

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \tag{83}$$

In order to arrive at an iteration rule, one separates an unknown in each equation from (83)

$$\begin{aligned} x_1 &= c_{12}x_2 + c_{13}x_3 + d_1 \\ x_2 &= c_{21}x_1 + c_{23}x_3 + d_2 \\ x_3 &= c_{31}x_1 + c_{32}x_2 + d_3 \end{aligned} \tag{84}$$

where $c_{ii} = 0$, $c_{ij} = -a_{ij}/a_{ii}$ for $i \neq j$ and $d_i = b_i/a_{ii}$ for $i, j = 1, 2, 3$. The system of equations can be thus divided into two parts, one with a lower left matrix $L$ and one with a right upper matrix $U$

$$L = \begin{pmatrix} 0 & c_{12} & c_{13} \\ 0 & 0 & c_{23} \\ 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} 0 & 0 & 0 \\ c_{21} & 0 & 0 \\ c_{31} & c_{32} & 0 \end{pmatrix} \tag{85}$$

resulting into $\vec{x} = U\vec{x} + L\vec{x} + \vec{d}$.
On this bases, we can define four iteration methods for solving a linear system of equations. Each of the methods is described completely by specifying how a given iterate $x^{(k)}$ is used to generate the next iterate $x^{(k+1)}$.

- the overall-step method or Jacobi method (J-method)

$$\vec{x}^{(k+1)} = U\vec{x}^{(k)} + L\vec{x}^{(k)} + \vec{d} \tag{86}$$

- the single-step method or Gauss and Seidel method (GS-method)

$$\vec{x}^{(k+1)} = U\vec{x}^{(k)} + L\vec{x}^{(k+1)} + \vec{d} \tag{87}$$

- the single-step method with over-relaxation, often referred to as successive over-relaxation method (SOR-method)

$$\vec{x}^{(k+1)} = \omega(U\vec{x}^{(k)} + L\vec{x}^{(k+1)} + \vec{d}) + (1-\omega)\vec{x}^{(k)} \tag{88}$$

- the overall-step method with over-relaxation, that is a generalization of the Jacobi method with over-relaxation (often referred to as JOR-method)

$$\vec{x}^{(k+1)} = \omega(U\vec{x}^{(k)} + L\vec{x}^{(k)} + \vec{d}) + (1-\omega)\vec{x}^{(k)} \tag{89}$$

The GS- and SOR-methods are more convenient for numerical implementation as they do not require the solution to be temporarily stored; however, this is usually not a decisive criterion for the selection of a specific method. For the discussion of convergence properties, reference is made to the standard literature from numerical mathematics.

### 2.3.1 J-method

The idea is to use the $i$-th equation to correct the $i$-th unknown. The $i$-th equation in the system $A\vec{x} = \vec{b}$ is

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \tag{90}$$

which can be rewritten (solved for $x_i$) as

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j\right) \tag{91}$$

since $a_{ii} \neq 0$. Let us define $x_i^{(k+1)}$ to be the adjusted value that would make the $i$-th equation true

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^{(k)}\right) \tag{92}$$

Making this adjustment for $i = 1, \ldots, n$, we obtain our next iterate $x^{(k+1)}$.
If the main-diagonal entries of the coefficient matrix are large relative to the the off-diagonal entries, convergence is aided, as elucidated here in the following. Let us write $A = D + R$, where $D$ is the diagonal matrix whose main diagonal entries are the same as those of $A$ and $R$ contains the off-diagonal entries

$$D = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{pmatrix} \quad \text{and} \quad R = \begin{pmatrix} 0 & a_{12} & \ldots & a_{1n} \\ a_{21} & 0 & \ldots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \ldots & 0 \end{pmatrix} \tag{93}$$

then equation (103) can be written as

$$x^{(k+1)} = D^{-1}\left(b - Rx_j^{(k)}\right) \tag{94}$$

If we define the residual after $k$ iterations as $r^{(k)} = b - Ax^{(k)}$, then

$$x^{(k+1)} = x^{(k)} + D^{-1}r^{(k)} \tag{95}$$

### 2.3.2 GS-Method

As before, we use the $i$-th equation to modify the $i$-th unknown, but now we consider doing the process sequentially. First we use the first equation to compute $x_1^{(k+1)}$, then we use the second equation to compute $x_2^{(k+1)}$, and so on. By the time we get to the $i$-th equation, we

have already computed $x_1^{(k+1)}, \ldots x_{i-1}^{(k+1)}$. In computing $x_i^{(k+1)}$, we could use these newly calculated $x_1^{(k+1)}, \ldots x_{i-1}^{(k+1)}$ or we could use the old values. The Jacobi method uses the old values; Gauss-Seidel uses the new. This is the only difference. Thus

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^{(*)} \right) \tag{96}$$

where $x_j^{(*)}$ denotes the most up-to-date value for the unknown $x_j$. More precisely, we can write the Gauss-Seidel iteration as follows

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \tag{97}$$

Note that in practice, there is no need to maintain separate storage locations for $x_i^{(k)}$ and $x_i^{(k+1)}$, everything can be done in a single $x$ array.

Note that the order in which the corrections are made is important. If they were made in, say, the reverse order $i = n, \ldots, 1$, the iteration would have a different outcome. The standard order is $i = 1, \ldots, n$.

The fact that we can store each new $x_i$ value immediately in place of the old one is an advantage of the Gauss-Seidel method over Jacobi. For one thing, it makes the programming easier. It also saves storage space; Jacobi's method needs to store two copies of $x$, since $x^{(k)}$ needs to be kept until the computation of $x^{(k+1)}$ is complete. If the system we are solving has millions of unknowns, each copy of $x$ will occupy several megabytes of storage space.

On the other hand, Jacobi's method has the advantage that all of the corrections can be performed simultaneously; the method is inherently parallel. Gauss- Seidel, naively implemented, is inherently sequential.

Let us write $A = D + L + U$, where $D$ is the diagonal matrix whose main diagonal entries are the same as those of $A$, $L$ is the strictly lower triangular part of $A$ and $U$ is the strictly upper triangular part of $A$, then equation (101) can be written as

$$x^{(k+1)} = D^{-1} \left( b - L x^{(k+1)} - U x^{(k)} \right) \tag{98}$$

or equivalently

$$x^{(k+1)} = (D + L)^{-1} \left( b - U x^{(k)} \right) \tag{99}$$

If we define the residual after $k$ iterations as $r^{(k)} = b - A x^{(k)}$, then

$$x^{(k+1)} = x^{(k)} + (D + L)^{-1} r^{(k)} \tag{100}$$

### 2.3.3 SOR-method

The process of correcting an equation by modifying one unknown is sometimes called relaxation. Before the correction, the equation is not quite true; like an assemblage of parts that does not fit together quite right, it is in a state of tension. The correction of one variable relaxes the tension. The GS-method performs successive relaxation. That is, it moves from equation to equation, relaxing one after the other.
In many cases convergence can be accelerated substantially by over-relaxing. This means that rather than making a correction for which the equation is satisfied exactly, we make a somewhat bigger correction. In the simplest case one chooses a relaxation factor $\omega > 1$ and overcorrects by that factor at each step

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) + (1-\omega) x_i^{(k)} \tag{101}$$

This is referred to as successive over-relaxation (SOR). The SOR-method reduces to the GS-method for $\omega = 1$

$$x^{(k+1)} = \omega x_{\mathrm{GS}}^{(k+1)} + (1-\omega) x^{(k)} \tag{102}$$

Note that, since SOR is successive (like GS) rather than simultaneous (like J), it needs to keep only one copy of the vector $x$.

### 2.3.4 JOR-method

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) + (1-\omega) x^{(k)} \tag{103}$$

The JOR-method reduces to the J-method for $\omega = 1$

$$x^{(k+1)} = \omega x_{\mathrm{J}}^{(k+1)} + (1-\omega) x^{(k)} \tag{104}$$

# 3 Computation of eigenvalues and eigenvectors

Eigenvalue problems occur in many places in physics. Examples are the determination of the normal modes of an oscillating system or the search for the solution of the time-independent Schrödinger equation or the determination of the shape parameters characterizing large macromolecules such as polymers. This means that eigenvalue problems are perhaps the most important problems in both classic and quantum mechanics.
The solution of eigensystems is a fairly complicated business, thus you should usually rely on publicly available, canned routines.The purpose of this section is to give you some appreciation of what is going on inside such canned routines, so that you can make intelligent choices about using them, and intelligent diagnoses when something goes wrong.

## 3.1 Some basic facts and definitions

### 3.1.1 Eigensystems

An $n$x$n$ matrix $A$ is said to have an *eigenvector* $\vec{x}$ if

$$A\vec{x} = \lambda\vec{x} \tag{105}$$

where $\lambda$ is the *eigenvalue* associated to $\vec{x}$. Obviously any multiple of $\vec{x}$ is also an eigenvector, but we will not consider such multiples as being distinct eigenvectors. Equation (105) implies that the associated *characteristic polynomial* $p(\lambda) = \det|A - \lambda\mathbb{1}|$ must be zero, *i.e.*,

$$\det|A - \lambda\mathbb{1}| = 0 \tag{106}$$

which, if expanded out, is an $n$-th degree polynomial in $\lambda$ whose roots are the eigenvalues. This proves that there are always $n$ (not necessarily distinct) eigenvalues for an $n$x$n$ matrix. Equal eigenvalues coming from multiple roots are called *degenerate*.
The above two equations also prove that every one of the $n$ eigenvalues has a (not necessarily distinct) corresponding eigenvector. For each eigenvalue $\lambda$ a corresponding eigenvector can be found by solving the equation

$$(A - \lambda\mathbb{1})\vec{x} = 0 \tag{107}$$

The *algebraic multiplicity* of an eigenvalue $\lambda$ of $A$ is the number of times it appears as a root of the characteristic polynomials. The *geometric multiplicity* of an eigenvalue $\lambda$ is the maximum number of linearly independent eigenvectors associated to $\lambda$. In general, the algebraic multiplicity and geometric multiplicity of an eigenvalue can differ. However, the geometric multiplicity can never exceed the algebraic multiplicity.
Except for low-dimensional problems ($n = 1, 2$, or 3), analytical solutions for the eigenvalues and eigenvectors are rarely known, and therefore one is almost always dependent on numerical methods. The eigenvalue problem could be solved by searching the roots of the characteristic polynomial. Unfortunately, this calculation method is very poor and can only be recommended for $n = 2$. We will learn much better ways, as well as efficient ways for finding corresponding eigenvectors.

### 3.1.2 Properties of matrices

A matrix is called *symmetric* if it is equal to its transpose

$$A = A^T \qquad \text{or} \qquad a_{ij} = a_{ji} \tag{108}$$

It is called *Hermitian* if it equals the complex-conjugate of its transpose (its Hermitian conjugate, denoted by $*$)

$$A = A^* \qquad \text{or} \qquad a_{ij} = a^*_{ji} \tag{109}$$

It is termed *orthogonal* if its transpose equals its inverse

$$A^T \cdot A = A \cdot A^T = \mathbb{1} \tag{110}$$

and *unitary* if its Hermitian conjugate equals its inverse

$$A^* \cdot A = A \cdot A^* = \mathbb{1} \tag{111}$$

Finally, a matrix is called *normal* if it commutes with its Hermitian conjugate

$$A^* \cdot A = A \cdot A^* \tag{112}$$

Note that for real matrices, Hermitian means the same as symmetric, unitary means the same as orthogonal, and both of these distinct classes are normal. The reason why "Hermitian" is an important concept has to do with eigenvalues: the eigenvalues of a Hermitian matrix are all real. The reason why "normal" is an important concept has to do with the eigenvectors: the eigenvectors of a normal matrix with non-degenerate (*i.e.* distinct) eigenvalues are complete and orthogonal, spanning the $n$-dimensional vector space. For a normal matrix with degenerate eigenvalues, we have the additional freedom of replacing the eigenvectors corresponding to a degenerate eigenvalue (also referred to as associated eigenvectors) by linear combinations of themselves. Using this freedom, we can always perform Gram-Schmidt orthogonalization and find a set of eigenvectors that are complete and orthogonal, just as in the non-degenerate case. The normal matrices contain both Hermitian matrices and real symmetric matrices, and cover the most relevant cases for physics.

### 3.1.3 Useful transformations

Note that, some transformations can improve the eigenvalues calculation, so here is a short overview of how the eigenvalues are affected by a given transformation.

$$\begin{array}{lrcll}
\text{Shift} & (A-\sigma\mathbb{1})\vec{x} & = & (\lambda-\sigma)\vec{x} & \\
\text{Inversion} & A^{-1}\vec{x} & = & \lambda^{-1}\vec{x} & \\
\text{Power} & A^2\vec{x} & = & \lambda^2\vec{x} & \\
\text{Polynomial expansion} & p_n(A)\vec{x} & = & p_n(\lambda)\vec{x} \quad \text{where} & p_n(A) = c_o\mathbb{1} + c_1 A + c_2 A^2 + \cdots + c_n A^n
\end{array} \tag{113}$$

Note that shifting is an important part of many algorithms for computing eigenvalues: as there is no special significance to a zero eigenvalue, any eigenvalue can be shifted to zero, or any zero eigenvalue can be shifted away from zero.
In the following, we will often rely on similarity transformations

$$A \to A' = Z^{-1} A Z \tag{114}$$

as such transformation leaves the eigenvalues of a matrix unchanged, *i.e.*,

$$\begin{aligned} \det|Z^{-1}AZ - \lambda\mathbb{1}| &= \det|Z^{-1}(A - \lambda\mathbb{1})Z| \\ &= \det|Z|\det|(A - \lambda\mathbb{1})|\det|Z^{-1}| \\ &= \det|(A - \lambda\mathbb{1})| \end{aligned} \tag{115}$$

### 3.2 Condition number of the eigenvalues problem

In numerics, the condition of a problem describes how much the solution of the problem changes with the disturbance of the input data. Given a problem $f$ with an input $x$ and an algorithm $\tilde{f}$ with the disturbed input data $\tilde{x}$, the absolute error $||f(x) - \tilde{f}(\tilde{x})||$ by applying the triangle inequality is

$$||f(x) - \tilde{f}(\tilde{x})|| = ||f(x) - f(\tilde{x}) + f(\tilde{x}) - \tilde{f}(\tilde{x})|| \leq ||f(x) - f(\tilde{x})|| + ||f(\tilde{x}) - \tilde{f}(\tilde{x})|| \tag{116}$$

where $||f(x) - f(\tilde{x})||$ is the condition of the problem and $||f(\tilde{x}) - \tilde{f}(\tilde{x})||$ is referred to as stability. Note that the condition is a property of the problem (not of the algorithm nor of floating-point accuracy), while the stability is a property of the algorithm.
The relative condition number of the problem $f(x)$ is the smallest number $K \geq 0$, such that

$$\frac{||f(x) - f(\tilde{x})||}{||f(x)||} \leq K \frac{||\tilde{x} - x||}{||x||} \tag{117}$$

The condition number associated with the linear equation $Ax = b$, where $A \neq 0$, is defined as the maximum of the ratio between the relative error in $x$

$$\frac{||A^{-1}e||}{||A^{-1}b||} \tag{118}$$

and the relative error in $b$

$$\frac{||e||}{||b||} \tag{119}$$

where $e$ is the error in $b$; namely

$$\begin{aligned} K(A) = \max_{e,b\neq 0}\left\{\frac{||A^{-1}e||}{||e||}\frac{||b||}{||A^{-1}b||}\right\} &= \max_{e\neq 0}\left\{\frac{||A^{-1}e||}{||e||}\right\}\max_{b\neq 0}\left\{\frac{||b||}{||A^{-1}b||}\right\} = \\ &= \max_{e\neq 0}\left\{\frac{||A^{-1}e||}{||e||}\right\}\max_{x\neq 0}\left\{\frac{||Ax||}{||x||}\right\} = \\ &= ||A^{-1}||\,||A|| \end{aligned} \tag{120}$$

The same definition is used for any consistent norm, *i.e.* one that satisfies

$$K(A) = ||A||||A^{-1}|| \geq ||AA^{-1}|| = 1 \tag{121}$$

When the condition number is exactly one, then a solution algorithm can find (in principle, meaning if the algorithm introduces no errors of its own) an approximation of the solution whose precision is no worse than that of the data. However, it does not mean that the algorithm will converge rapidly to this solution, just that it will not diverge arbitrarily because of inaccuracy on the source data, provided that the error introduced by the algorithm does not diverge as well because of accumulating intermediate rounding errors. The condition number may also be very large, then the matrix is said to be ill-conditioned: practically, such a matrix is almost singular and the computation of its inverse, or the solution of a linear system of equations associated to it, is prone to large numerical errors; a matrix that is not invertible has condition number equal to infinity. If the condition number is not too much larger than one, the matrix is well-conditioned, which means that its inverse can be computed with good accuracy.
The eigenvalue problem is well-conditioned for normal matrices. If $A$ has only real eigenvalues $\lambda_1, \ldots, \lambda_n$ and if there is an orthonormal basis of eigenvectors $x_1, \ldots, x_n$, then $Ax_i = \lambda_i x_i$, *i.e.*

$$AQ = Q\Lambda \quad \text{where} \quad \Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n) \quad \text{and} \quad Q = [x_1, \ldots, x_n] \tag{122}$$

Multiplying by $Q^T$ from the left gives the relationship $Q^T AQ = \Lambda$, and multiplying by $Q^T$ from the right gives $A = Q\Lambda Q^T$. Thus, the inverse of matrix $A$ is

$$A^{-1} = Q\Lambda^{-1}Q^T \tag{123}$$

As we previously learned, the condition number of a matrix $A$ is given by

$$K(A) \equiv ||A||||A^{-1}|| \tag{124}$$

where the induced norm of matrix $A$ is defined as

$$||A|| \equiv \sup_{x \neq 0} \frac{||Ax||}{||x||} = \max_{||x||=1} ||Ax|| \tag{125}$$

The $\max_{||x||=1}||Ax||$ is given the largest eigenvalue of $A$, *i.e.* $\max_{||x||=1}||Ax|| = \max|\lambda_i|$. Similarly, $\max_{||x||=1}||A^{-1}x||$ is given by the largest eigenvalue of the inverse matrix $A^{-1}$, that is the smallest eigenvalue of $A$, *i.e.* $\sup_{||x||=1}||A^{-1}x|| = 1/\min|\lambda_i|$. Thus, the condition number of $A$ is

$$K(A) = \frac{\max|\lambda_i|}{\min|\lambda_i|} \tag{126}$$

This relationship says that the handling and calculation with matrices becomes more difficult the "broader" the eigenvalue spectrum is. Note that similarity transformations do not change the condition number of a matrix as they do not change the eigenvalue spectrum of the matrix.
A first estimate for eigenvalues can be achieved by using the Gershgorin circles theorem. This theorem bounds the spectrum of a complex square matrix $C \, \epsilon \, \mathcal{C}^{nxn}$ as it states that all

eigenvalues of $C$ are in circles in the complex plane around the matrix diagonal elements, *i.e.*

$$|\lambda_i - c_{ii}| \leq \sum_{j \neq i} |c_{ij}| \quad \text{by rows} \tag{127}$$

$$|\lambda_i - c_{ii}| \leq \sum_{j \neq i} |c_{ji}| \quad \text{by columns} \tag{128}$$

where the estimate is given by the union of the circles by rows and by columns.

### 3.3 The grand strategy to solve eigensystem problems

In order to determine the eigenvalues of a matrix, we look for similarity transformations that convert $A$ into a diagonal matrix. A matrix $A$ is diagonalizable if there exists an invertible matrix $X$ such that $X^{-1}AX = \Lambda$, where $\Lambda$ is a diagonal matrix. If $X_R$ is the matrix formed by columns from the right eigenvectors (remember: right eigenvectors are column vectors that are multiplied to the right of a matrix A and satisfy $A\vec{x} = \lambda\vec{x}$) and $X_L$ is the matrix formed by rows from the left eigenvectors (remember: left eigenvectors are row vectors that multiply $A$ to the left and satisfy $\vec{x}A = \lambda\vec{x}$), then

$$AX_R = X_R\Lambda \tag{129}$$

$$X_LA = \Lambda X_L \tag{130}$$

where $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$. Note that if the matrix A is symmetric, then the left and right eigenvectors are just transposes of each other, that is, have the same numerical values as components (likewise, if the matrix is Hermitian, the left and right eigenvectors are Hermitian conjugates of each other).
Multiplying equation (129) by $X_R^{-1}$, we get

$$X_R^{-1}AX_R = \text{diag}(\lambda_1, \ldots, \lambda_n) \tag{131}$$

which is a particular case of a similarity transformation of the matrix $A \to Z^{-1}AZ$ for some matrix $Z$. Note that for real symmetric matrices, the eigenvectors are real and orthonormal, so the transformation matrix is orthogonal. The similarity transformation is then also an *orthogonal transformation* of the form $A \to Z^TAZ$.
Equation (131) shows that any matrix with complete eigenvectors (which includes all normal matrices) can be diagonalized by a similarity transformation, where the columns of the transformation matrix are the right eigenvectors (and the rows of its inverse are the left eigenvectors).
Almost all eigensystem routines bring $A$ towards the diagonal form by a sequence of similarity transformations

$$A \to P_1^{-1}AP_1 \to P_2^{-1}P_1^{-1}AP_1P_2 \to P_3^{-1}P_2^{-1}P_1^{-1}AP_1P_2P_3 \to \ldots \tag{132}$$

If we get all the way to the diagonal form, then the eigenvectors are the columns of the accumulated transformations

$$X_R = P_1 P_2 P_3 \dots \tag{133}$$

Sometimes we do not want to go all the way to the diagonal form. For example, if we are interested only in eigenvalues, not eigenvectors, it is enough to transform the matrix $A$ to be tridiagonal or triangular.

The grand strategy (132) thus can be either

- a reduction of the matrix to a diagonal form (Jacobi method 3.4); in this case, we iterate the finite sequence of transformations over and over until the deviation of the matrix from diagonal is negligibly small

or

- a reduction of the matrix to a tridiagonal or triangular form (Givens 3.5.1 and Householder 3.5.2 methods); in this case, we use the finite sequence of transformations to go most of the way and then we calculate the eigenvalues either by a recursion relation (3.5.3) or by a factorization method ($QR$-method 3.6.2)

Note that if we want only the maximum (or minimum) eigenvalue of a matrix and possibly its corresponding eigenvector then we consider a slightly different strategy (power method 3.7).

## 3.4 From a symmetric matrix to a diagonal form

### 3.4.1 Jacobi Transformations

A Jacobi transformation is an orthogonal similarity transformation designed to reduce a symmetric matrix to a diagonal form. More specifically, a Jacobi transformation is a plane rotation designed to annihilate one of the off-diagonal elements of the matrix $A$

$$A = \begin{pmatrix} * & & & \dots & & & * \\ & \ddots & & & & & \\ & & a_{pp} & \dots & a_{pq} & & \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ & & a_{qp} & \dots & a_{qq} & & \\ & & & & & \ddots & \\ * & & & \dots & & & * \end{pmatrix} \rightarrow A' = \begin{pmatrix} * & & & \dots & & & * \\ & \ddots & & & & & \\ & & a'_{pp} & \dots & 0 & & \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ & & 0 & \dots & a'_{qq} & & \\ & & & & & \ddots & \\ * & & & \dots & & & * \end{pmatrix} \tag{134}$$

*i.e.*

$$A \rightarrow A' = P_{pq}^T A P_{pq} \tag{135}$$

where $P_{pq}$ is the matrix of the Jacobi rotation that transforms the matrix $A$ into $A'$ and has the form

$$P_{pq} = \begin{pmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \cdots & \\ & & & & & & 1 \end{pmatrix} \tag{136}$$

Here all the diagonal elements are unity except for the two elements $c$ in rows (and columns) $p$ and $q$. All off-diagonal elements are zero except the two elements $s$ and $-s$. The numbers $c$ and $s$ are the cosine and sine of a rotation angle $\phi$, thus $c^2 + s^2 = 1$. So geometrically, the matrix describes a rotation with angle $\phi$ in the $(p, q)$-plane. Notice that the subscripts $p$ and $q$ do not denote components of $P_{pq}$, but rather label which kind of rotation the matrix is, *i.e.* which rows and columns it affects.
In the aforementioned Jacobi rotation (135), $P_{pq}^T A$ changes only rows $p$ and $q$ of $A$, while $AP_{pq}$ changes only columns $p$ and $q$. Thus the changed elements of $A$ in (135) are only in the $p$ and $q$ rows and columns indicated below

$$A' = \begin{pmatrix} & \cdots & a'_{1p} & \cdots & a'_{1q} & \cdots & \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{p1} & \cdots & a'_{pp} & \cdots & a'_{pq} & \cdots & a'_{pn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{q1} & \cdots & a'_{qp} & \cdots & a'_{qq} & \cdots & a'_{qn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ & \cdots & a'_{np} & \cdots & a'_{nq} & \cdots & \end{pmatrix} \tag{137}$$

where

$$a'_{rp} = ca_{rp} - sa_{rq} \quad r \neq p, r \neq q \tag{138}$$
$$a'_{rq} = ca_{rq} + sa_{rp} \quad r \neq p, r \neq q \tag{139}$$
$$a'_{pp} = c^2 a_{pp} + s^2 a_{qq} - 2sca_{pq} \tag{140}$$
$$a'_{qq} = s^2 a_{pp} + c^2 a_{qq} + 2sca_{pq} \tag{141}$$
$$a'_{pq} = (c^2 - s^2)a_{pq} + sc(a_{pp} - a_{qq}) \tag{142}$$

As the idea of the Jacobi method is to put to zero the off-diagonal elements by a series of plane rotations, we set $a'_{pq} = 0$. Equation (142) thus gives the following expression for the rotation angle $\phi$

$$\cot 2\phi \equiv \frac{c^2 - s^2}{2sc} = \frac{a_{qq} - a_{pp}}{2a_{pq}} \tag{143}$$

Note that a sequence of successive transformations of the form of equation (132) undo previously set zeros, but the off-diagonal elements nevertheless get smaller and smaller, until the matrix is diagonal to machine precision. Eventually one obtains a diagonal matrix $D$ where the diagonal elements are the eigenvalues of the original matrix $A$.

### 3.4.2 Eigenvalues and eigenvectors of a diagonal matrix

Accumulating the product of the transformations gives the matrix of eigenvectors $V$, *i.e.*

$$D = V^T A V \tag{144}$$

where

$$V = P_1 P_2 P_3 \ldots \tag{145}$$

the $P_i$'s being the successive Jacobi rotation matrices. Since $AV = VD$, the columns of $V$ are the eigenvectors. They can be computed by applying $V' = VP_i$ at each stage of calculation, where initially $V$ is the identity matrix, *i.e.*

$$v'_{rs} = v_{rs} \quad s \neq p, s \neq q \tag{146}$$

$$v'_{rp} = cv_{rp} - sv_{rq} \tag{147}$$

$$v'_{rq} = sv_{rp} + cv_{rq} \tag{148}$$

The only remaining question is the strategy one should adopt for the order in which the elements are to be annihilated. Jacobi's original algorithm of 1846 searched the whole upper triangle at each stage and set the largest off-diagonal element to zero. A better strategy for our purposes is the cyclic Jacobi method, where one annihilates elements in strict order. For example, one can simply proceed down the rows: $P_{12}, P_{13}, ..., P_{1n}$; then $P_{23}, P_{24}$, etc.
The Jacobi method is conceptually simple and foolproof for all symmetric matrices; however, for $n \gtrsim 10$, it is computationally inefficient, it is thus recommend only for matrices of moderate order.

## 3.5 From a symmetric matrix to a tridiagonal form

Instead of trying to reduce the matrix all the way to diagonal form, we stop when the matrix is tridiagonal. This allows the procedure to be carried out in a finite number of steps, unlike the Jacobi method, which requires iteration to convergence. We describe here two methods to reduce a symmetric matrix to a tridiagonal form: Givens and Householder method. The latter is generally more used as it is more efficient than the first; the first has the advantage that it can be easily parallelized and it works very well for sparse matrices.

### 3.5.1 Givens Method

Givens reduction is a modification of the Jacobi method. In this case, we choose the rotation angle in (136) such that it puts to zero an element that is not at one of the four "corners" in (137), *i.e.* it is not $a_{pp}$, $a_{pq} = a_{qp}$, or $a_{qq}$. We choose the sequence where $P_{pq}$ annihilates for instance $a_{q-1,p}$ and (by symmetry) $a_{p,q-1}$. Using (138), we can define the angle $\phi$

$$a'_{q-1,p} = 0 = ca_{q-1,p} - sa_{q-1,q} \rightarrow \tan\phi = \frac{a_{q-1,p}}{a_{q-1,q}} \tag{149}$$

The method works because elements $a'_{rp}$ and $a'_{rq}$, with $r \neq p$ and $r \neq q$, are linear combinations of the old quantities $a_{rp}$ and $a_{rq}$, thus if $a_{rp}$ and $a_{rq}$ have been already set to zero, they remain zero as the reduction proceeds.

### 3.5.2 Householder method

The Householder matrix $H$ is an orthogonal matrix ($H^T H = H^2 = \mathbb{1}$) defined as

$$H = \mathbb{1} - 2\frac{\vec{u} \cdot \vec{u}^T}{|\vec{u}|^2} \tag{150}$$

where $\vec{u}$ can be any vector. Choose $\vec{u}$ such that

$$\vec{u} = \vec{a_i} \mp |\vec{a_i}|\vec{e}_1 \tag{151}$$

where $\vec{a_i}$ is the vector composed of the first column of $A$, $\vec{e}_1 = (1, 0, \ldots, 0)^T$ is the unit vector and the choice of the sign is made later. Then

$$H\vec{a_i} = \pm|\vec{a_i}|\vec{e}_1 \tag{152}$$

as

$$\begin{aligned} H\vec{a_i} &= \vec{a_i} - 2\frac{\vec{u}}{|\vec{u^2}|}(\vec{a_i} \mp |\vec{a_i}|\vec{e}_1)^T\vec{a_i} \\ &= \vec{a_i} - \frac{2\vec{u}(|\vec{a_i}|^2 \mp |\vec{a_i}|a_1)}{(|2\vec{a_i}|^2 \mp 2|\vec{a_i}|a_1)} \\ &= \vec{a_i} - \vec{u} \\ &= \pm|\vec{a_i}|\vec{e}_1 \end{aligned}$$

Equation (152) shows that the Householder matrix $H$ sets to zero all elements of a given vector except the first one.
To reduce a symmetric matrix $A$ to a tridiagonal form, we choose the vector for the first Householder matrix to be the lower $n-1$ elements of the first column of $A$, *i.e.*

$$\vec{u}_1 = \begin{pmatrix} a_{21} \\ a_{31} \\ \vdots \\ a_{n1} \end{pmatrix} \mp \left| \begin{pmatrix} a_{21} \\ a_{31} \\ \vdots \\ a_{n1} \end{pmatrix} \right| \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \rightarrow H_1^{(n-1)} = \mathbb{1} - \frac{2\vec{u}_1 \cdot \vec{u}_1^T}{|\vec{u}_1|^2} \tag{153}$$

where $H_1^{(n-1)}$ denotes a Householder matrix with dimensions $(n-1)$x$(n-1)$. Then the lower $n-2$ elements of column one will be zeroed

$$H_1 A = \left(\begin{array}{c|ccc} 1 & 0 & \dots & 0 \\ \hline 0 & & & \\ 0 & & & \\ \vdots & & H_1^{(n-1)} & \\ 0 & & & \end{array}\right) \left(\begin{array}{c|ccc} a_{11} & a_{12} & \dots & a_{1n} \\ \hline a_{21} & & & \\ a_{31} & & & \\ \vdots & & a_{ij} & \\ a_{n1} & & & \end{array}\right) = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \dots & a_{1n} \\ \hline k & & & \\ 0 & & & \\ \vdots & & a'_{ij} & \\ 0 & & & \end{array}\right) \tag{154}$$

The quantity $k$ is simply plus or minus the magnitude of the vector $(a_{21}, \dots, a_{n1})^T$. Using the fact that $H_1 H_1^T = \mathbb{1}$, the complete orthogonal transformation is

$$A' = H_1 A H_1 = \left(\begin{array}{c|ccc} a_{11} & k & 0 & \dots & 0 \\ \hline k & a'_{22} & a'_{23} & \dots & a'_{2n} \\ 0 & a'_{32} & a'_{33} & \dots & a'_{3n} \\ \vdots & \vdots & & & \vdots \\ 0 & a'_{n2} & a'_{n3} & \dots & a'_{nn} \end{array}\right) \tag{155}$$

Now choose the vector for the second Householder matrix to be the bottom $n-2$ elements of the second column of $A$, *.i.e*

$$\vec{u}_2 = \begin{pmatrix} a_{32} \\ a_{42} \\ \vdots \\ a_{n2} \end{pmatrix} \mp \left|\begin{pmatrix} a_{32} \\ a_{42} \\ \vdots \\ a_{n2} \end{pmatrix}\right| \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \rightarrow H_2^{(n-2)} = \mathbb{1} - \frac{2\vec{u}_2 \cdot \vec{u}_2^T}{|\vec{u}_2|^2} \tag{156}$$

and we construct

$$H_2 = \left(\begin{array}{cc|ccc} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \hline 0 & 0 & & & \\ \vdots & & & H_2^{(n-2)} & \\ 0 & 0 & & & \end{array}\right) \tag{157}$$

The complete orthogonal transformation thus is $A'' = H_2 A' H_2 = H_2 H_1 A H_1 H_2$.
The identity block in the upper left corner guarantees that the tridiagonalization achieved in the first step will not be spoiled by this one, while the $(n-2)$-dimensional Householder matrix $H_1^{(n-2)}$ creates one additional row and column of the tridiagonal output. The Householder algorithm reduces an $n$x$n$ symmetric matrix $A$ to a tridiagonal form by $(n-2)$ orthogonal transformations. Each transformation annihilates the required part of a whole column and whole corresponding row.

### 3.5.3 Eigenvalues of a tridiagonal matrix

If our symmetric matrix can be reduced to a tridiagonal form, one possible way to determine its eigenvalues is to find the roots of the characteristic polynomial

$$p(\lambda) = \det \begin{pmatrix} a_{11}-\lambda & a_{21} & 0 & 0 & \dots & \dots & 0 \\ a_{21} & a_{22}-\lambda & a_{23} & 0 & \dots & \dots & 0 \\ 0 & a_{32} & a_{33}-\lambda & a_{34} & 0 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & \dots & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & a_{n-1n-2} & a_{n-1n-1}-\lambda & a_{n-1n} \\ 0 & \dots & \dots & \dots & 0 & a_{nn-1} & a_{nn}-\lambda \end{pmatrix} \tag{158}$$

The characteristic polynomial of a tridiagonal matrix can be evaluated for any trial value of $\lambda$ by the recursion relation (obtained by exploiting the properties of the determinant of a tridiagonal matrix)

$$p_i(\lambda) = (a_{ii}-\lambda)p_{i-1}(\lambda) - a_{ii-1}^2 p_{i-2}(\lambda) \tag{159}$$

The trial values of $\lambda$ can be searched for in the interval defined by the Gershgorin circles theorem

$$\min\left\{a_{ii} - \sum_{j\neq i}|a_{ij}|\right\} \leq \lambda \leq \max\left\{a_{ii} + \sum_{j\neq i}|a_{ij}|\right\} \tag{160}$$

The polynomials of lower degree produced during the recurrence form a sequence, namely $\{1, p_1(\lambda_i), p_2(\lambda_i), \dots, p_n(\lambda_i)\}$, that can be used to localize the eigenvalues to intervals on the real axis. A root-finding method (such as Newton's algorithm) can then be employed to refine the intervals.

### 3.5.4 Eigenvectors by inverse iteration

Once the eigenvalues $\lambda_i$ of a $n$x$n$ matrix $A$ are known, the eigenvectors $\vec{x}_i$, with $i = 1, \dots, n$ can be computed by inverse iteration. Let $\vec{x'}_i^{(1)}$ be the solution of

$$(A - \tau\mathbb{1})\vec{x'}_i^{(1)} = \vec{x}_i^{(0)} \tag{161}$$

where $\vec{x}_i^{(0)}$ is any starting vector that is properly normalized $|\vec{x}_i^{(0)}| = 1$. If $\tau = (\lambda_i + \epsilon)$, then $\vec{x'}_i^{(1)}$ is relatively close to $\vec{x}_i$. If we now normalize the solution

$$\vec{x}_i^{(1)} = \frac{\vec{x'}_i^{(1)}}{|\vec{x'}_i^{(1)}|} \tag{162}$$

and iterate the approach, we can refine more and more our estimate of $\vec{x}_i$. Namely, at the $j$-th stage of iteration we are solving the equation

$$(A - (\lambda_i + \epsilon)\mathbb{1})\vec{x'}_i^{(j+1)} = \vec{x}_i^{(j)} \tag{163}$$

where $\vec{x}_i^{(j)}$ is our current guess for some eigenvector $\vec{x}_i$ and $\vec{x'}_i^{(j+1)}$ is an improved approximation of it, that is successively normalized

$$\vec{x}_i^{(j+1)} = \frac{\vec{x'}_i^{(j+1)}}{|\vec{x'}_i^{(j+1)}|} \tag{164}$$

and used to iterate further. We can see why this approach works by expanding both $\vec{x'}_i^{(j+1)}$ and $\vec{x}_i^{(j)}$ as linear combinations of the eigenvectors of $A$

$$\vec{x'}_i^{(j+1)} = \sum_k \alpha_k^{(j)} \vec{x}_k \quad \text{and} \quad \vec{x}_i^{(j)} = \sum_k \beta_k^{(j)} \vec{x}_k \tag{165}$$

then equation (163) gives

$$\sum_k \alpha_k^{(j)} (\lambda_k - \lambda_i - \epsilon)\vec{x}_k = \sum_k \beta_k^{(j)} \vec{x}_k \tag{166}$$

so that

$$\alpha_k = \frac{\beta_k^{(j)}}{\lambda_k - \lambda_i - \epsilon} \tag{167}$$

and

$$\vec{x'}_i^{(j+1)} = \sum_k \frac{\beta_k^{(j)} \vec{x}_k}{\lambda_k - \lambda_i - \epsilon} \tag{168}$$

When $\epsilon \to 0$, then the sum is dominated by the term $\lambda_i = \lambda_k$ (provided that $\beta_k^{(j)}$ is not accidentally too small).

While the above formulas look simple enough, in practice the implementation can be quite tricky. The first question to be resolved is when to use inverse iteration. Most of the computational load occurs in solving the linear system (163). Thus a possible strategy is first to reduce the matrix $A$ to a special form that allows easy solution of (163). Then apply inverse iteration to generate all the eigenvectors. Inverse iteration is generally used when one already has good eigenvalues and wants only a few selected eigenvectors. You can write a simple inverse iteration routine yourself using $LU$-decomposition to solve (163). Note though, that, since the linear system (163) is nearly singular, you must be careful.

## 3.6 From a normal matrix to a triangular form

### 3.6.1 Reduction to a triangular form

A Givens rotation is designed to zero a single off-diagonal element in the matrix, while Householder rotation sets to zero a whole particular row or column; they both reduce a symmetric matrix to a tridiagonal form, but the latter is more efficient. Both can be used to reduce normal $A$ matrix into *e.g.* an upper triangular matrix by a series of orthogonal transformations

$$A = \begin{pmatrix} * & * & * & \dots & * \\ * & * & * & \dots & * \\ * & * & * & \dots & * \\ \vdots & & \ddots & & \vdots \\ * & \dots & * & * & * \end{pmatrix} \to \dots \to A' = \begin{pmatrix} * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ 0 & 0 & * & \dots & * \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & 0 & 0 & * \end{pmatrix} \tag{169}$$

### 3.6.2 The $QR$- and $QL$-algorithms

The basic idea behind the $QR$ ($QL$)-algorithm is that any real matrix $A$ can be decomposed in the product of an orthogonal matrix $Q$ and an upper (lower) triangular matrix $R$ ($L$)

$$A = QR \tag{170}$$

The standard, but confusing, nomenclature $R$ (and $L$) stands for whether the right or left of the matrix is nonzero (also the $LU$-factorization was initially called $LR$-factorization). There is nothing special about $A$ to be upper triangular; one could equally well make it lower triangular; thus we use $QR$ henceforth.
As $Q$ is orthogonal, then $R = Q^T A$ and thus we have

$$A' = RQ = Q^T AQ \tag{171}$$

We see that $A'$ is an orthogonal transformation of $A$. You can verify that a $QR$-transformation preserves the following properties of a matrix: symmetry, tridiagonal form, and Hessenberg form.
The $QR$-algorithm consists of a sequence of orthogonal transformations

$$\begin{aligned} A^{(j)} &= Q^{(j)} R^{(j)} \\ A^{(j+1)} &= R^{(j)} Q^{(j)} = Q^{(j)T} A^{(j)} Q^{(j)} \end{aligned} \tag{172}$$

For a general matrix, the decomposition is constructed by applying orthogonal similarity transformations (such as the Householder reduction) to annihilate successive columns of $A$ below the diagonal.

- if $A$ has non-degenerate eigenvalues, then $A^{(j)}$ converges to an upper triangular form as $j \to \infty$, and the eigenvalues appear on the diagonal in increasing order of absolute magnitude

$$A^{(j\to\infty)} = \begin{pmatrix} \lambda_1 & * & * & & \dots & * \\ 0 & \lambda_2 & * & & \dots & \vdots \\ 0 & 0 & \lambda_3 & & & \\ \vdots & & \ddots & \ddots & & \\ \vdots & & & & & \\ 0 & \dots & & & 0 & \lambda_n \end{pmatrix} \tag{173}$$

- if $A$ has a degenerate eigenvalue of multiplicity $p$, then $A^{(j)}$ converges to an upper triangular form as $j \to \infty$ with a block matrix along diagonal of order $p$ whose eigenvalue is $p$-degenerate

$$A^{(j\to\infty)} = \begin{pmatrix} \lambda_1 & * & * & \dots & & & * \\ 0 & \lambda_2 & * & \dots & & & \vdots \\ 0 & 0 & \lambda_3 & \dots & & & \\ \vdots & & \ddots & & & & \\ & & & a_{ii} & \dots & a_{ii+p} & \\ & & & \vdots & \ddots & \vdots & * \\ & & & a_{i+pi} & \dots & a_{i+pi+p} & \\ \vdots & & & & & & \ddots \\ 0 & \dots & & & & 0 & \lambda_n \end{pmatrix} \tag{174}$$

You recall the corresponding eigenvectors as the columns of $\Pi_i Q_i$.
The major limitation of the $QR$-algorithm is that already the first stage generates usually complete fill-in in general sparse matrices. It can therefore not be applied to large sparse matrices, simply because of excessive memory requirements. On the other hand, the $QR$-algorithm computes all eigenvalues which is rarely desired in sparse matrix computations anyway.

### 3.7 The power method for the dominant eigenvalue and eigenvector

The power method is used to estimate the extremal eigenvalues of a matrix as well as their associated eigenvectors and it is particularly useful for sparse matrices.
We consider a real, symmetric $nxn$ matrix $A$ with eigenvalues $\lambda_i$ and eigenvectors $\vec{x}_i$, with $i = 1, \dots, n$. We assume that the eigenvalues of $A$ are real, distinct, and ordered by decreasing magnitude, thus $\lambda_1$ and $\lambda_\mathrm{n}$ are respectively the largest $\lambda_1 = \lambda_{\max}$ and the

smallest $\lambda_n = \lambda_{\min}$ eigenvalue of $A$. Further, we assume that $A$ has a dominant eigenvalue $|\lambda_1| \gg |\lambda_i|$ for $i = 2, \ldots, n$.
The Rayleigh quotient of a vector $\vec{z}$ for our real symmetric matrix $A$ is defined as

$$\rho(\vec{z}, A) = \frac{\vec{z}^T A \vec{z}}{\vec{z}^T \vec{z}} \tag{175}$$

where $\vec{z}\ \epsilon\ \mathcal{R}^n$. It can be shown that

$$\max\rho(\vec{z}, A) = \lambda_1 \qquad \min\rho(\vec{z}, A) = \lambda_n \tag{176}$$

or equivalently

$$\lambda_{\rm n} \leq \rho(\vec{z}, A) \leq \lambda_1 \tag{177}$$

where $\lambda_1$ and $\lambda_{\rm n}$ are respectively the largest and the smallest eigenvalue of $A$, as stated before.
Note that if $\vec{z}$ is an eigenvector of $A$ corresponding to the eigenvalue $\lambda$, then the Rayleigh quotient of $\vec{z}$ is exactly $\lambda$

$$\rho(\vec{z}, A) = \frac{\vec{z}^T A \vec{z}}{\vec{z}^T \vec{z}} = \frac{\lambda \vec{z}^T \vec{z}}{\vec{z}^T \vec{z}} = \lambda \tag{178}$$

while if $\vec{z}$ is as an estimate for an eigenvector of $A$, then $\rho(\vec{z}, A)$ is the best estimate for the corresponding eigenvalue $\lambda$. This statement can be understood by observing that the Rayleigh quotient is a weighted average of the eigenvalues of $A$: if we (i) use the orthonormal eigensystem, namely we define the matrix whose columns are the eigenvectors $V = (\vec{x}_1, \ldots, \vec{x}_n)$, then $V^T A V = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$, and (ii) write $\vec{z}$ as a linear combination of eigenvectors $\vec{z} = \sum_i c_i \vec{x}_i$, then

$$\rho(\vec{z}, A) = \frac{\vec{z}^T V^T A V \vec{z}}{\vec{z}^T \vec{z}} = \frac{\sum_i c_i^2 \lambda_i}{\sum_i c_i^2} \tag{179}$$

Since in a weighted average elements with a high weight contribute more than do elements with a low weight, if $\vec{z}$ is an estimate of the larger (smaller) eigenvalue $\lambda_1$ ($\lambda_n$), then the dominant $c_i$ is $\lambda_1$ ($\lambda_n$) and thus $\rho(\vec{z}, A)$ approaches $\lambda_1$ ($\lambda_n$) from below (above).
We now choose an initial approximation for corresponding dominant eigenvector $\vec{x}_1$: it must be a nonzero vector in $\mathcal{R}^n$ and we write it as a linear combination of the eigenvectors of $A$

$$\vec{z}^{(0)} = \sum_i c_i \vec{x}_i \tag{180}$$

Then we write the iteration sequence

$$\begin{aligned} \vec{z}^{(j)} &= A\vec{z}^{(j-1)} = A^{(j)}\vec{z}^{(0)} = \sum_i \lambda_i^{(j)} c_i \vec{x}_i \qquad (181) \\ &= \lambda_1^{(j)} \left( c_1 x_1 + c_2 x_2 \left(\frac{\lambda_2}{\lambda_1}\right)^{(j)} + \cdots + c_n x_n \left(\frac{\lambda_n}{\lambda_1}\right)^{(j)} \right) \end{aligned}$$

Since the eigenvalues are ordered by decreasing magnitude, it follows that for all $i = 2, \ldots, n$

$$\lim_{j\to\infty} \left(\frac{\lambda_i}{\lambda_1}\right)^{(j)} = 0 \tag{182}$$

implying that, as $j$ increases,

$$\vec{z}^{(j)} = \lambda_1^{(j)} c_1 \vec{x}_1 \tag{183}$$

Note that, as the most time-consuming operation of the algorithm is the multiplication of matrix $A$ by a vector, the power method is effective for large sparse matrices. In cases for which the power method generates a good approximation of a dominant eigenvector, the Rayleigh quotient provides a correspondingly good approximation of the dominant eigenvalue: at the $j$-th iteration of the power method, we have

$$\rho(\vec{z}, A) = \frac{\vec{z}^{(j)T} A \vec{z}^{(j)}}{\vec{z}^{(j)T} \vec{z}^{(j)}} = \frac{\vec{z}^{(j)T} \vec{z}^{(j+1)}}{\vec{z}^{(j)T} \vec{z}^{(j)}} = \lambda_1 + O\left(\left|\frac{\lambda_i}{\lambda_1}\right|^{2j}\right) \tag{184}$$

A computer code to apply such a method looks something like this

Define $z_0$
Calculate the unit vector $y_0 = z_0/|z_0|$
Calculate the new vector $z_1 = Ay_0$
Calculate the first estimate of $\lambda = y_0 z_1/(y_0 y_0)$
Define a convergence criteria $tol = 10^{-6}$

do while (err.gt.(tol*abs(lambda)).and.(k.lt.100))

    Calculate the new unit vector $y_k = z_k/|z_k|$ with $k = 1, ...kmax$
    Calculate the new vector $z_{k+1} = Ay_k$
    Calculate the new estimate of $\lambda = y_k * z_{k+1}/(y_k y_k)$

    err=$|\lambda' - \lambda|$
    $\lambda = \lambda'$
    k=k+1

end do

The maxmimum eigenvalue is $\lambda'$ after $k$ iterations
The associated eigenvector is $y_k$ after $k$ iterations

Finally, we note that the inverse power method allows you to determine the minimum eigenvalue and its corresponding eigenvector: you simply apply the power method to $A^{-1}$.

# 4 References

Please, also refer to [1, 2].

# References

[1] William H. Press, William T. Vetterling, and Saul Teukolsky. *Numerical Recipes: The Art of Scientific Computing, third edition.* Cambridge University Press, 1986.

[2] David Watkins. *Fundamental of Matrix Computations, second edition.* Wiley Interscience, 2002.