3 Systems of Linear Equations, Direct Methods

We use the standard notation for systems of linear equations,

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

where A denotes the coefficient matrix, $\mathbf{b}$ the right-hand side, and $\mathbf{x}$ the solution vector. If the system consists of n equations and unknowns, then A is an $n \times n$ matrix.

Solution methods for linear systems of equations fall into two main categories: Direct and iterative methods.

- Direct methods compute an exact solution (assuming no rounding error during computation). For example, elimination, substitution, and Cramer's rule fall into this category. If you want to use paper and pencil to solve systems with two or three unknowns, they are the methods of choice. Computers can easily use direct methods for thousands of equations and unknowns.
- Iterative methods start with some initial guess and compute progressively better approximate solutions. They are only suitable for systems of equations with a specific matrix structure. This way, computers can solve huge systems of equations (several million unknowns), such as those arising in numerical flow simulations or structural analysis.

This chapter deals with direct methods and repeats (what you should know from mathematics 1) theoretical statements on the existence and uniqueness of the solution; Chapter y will treat iterative methods.

Software of recognized high quality is freely available in the LAPACK program library (`http://www.netlib.org/lapack/`). Even in commercially available software packages you will not find anything better. MATLAB also contains the LAPACK algorithms. (By the way, MATLAB was initially created as a simple user interface to this package).

3.1 Triangular matrices

If A is a lower or upper triangular matrix, one can solve the system $A\mathbf{x} = \mathbf{b}$ directly by forward or backward substitution, respectively.

Otherwise, you could transform the equations to triangular form as desdribed in Section 3.2. Alternatively, you could factorize A into a product of triangular matrices. The corresponding procedure is described in Section 3.5.

We denote triangular matrices by L and U. In the usual notation, in L only the entries in the lower left triangle are different from zero and all entries of the main diagonal are equal to one. In U, only entries in the upper right triangle, including the main diagonal are not equal to zero. Example for $n = 4$:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

If the elements of a triangular matrix L are stored on an array `a[i][j]` and the right-hand side $\mathbf{b}$ on a vector `b[i]`, the following Java program segment solves the system $L\mathbf{x} = \mathbf{b}$ stepwise by *forward substitution*.

Note that Java counts array indices from 0 to $n-1$; conventional math notation counts rows and columns from 1 to n.)

```
for (int i=0; i<n; i++) {
   x[i] = b[i];
   for (int j=0; j<i; j++) {
      x[i] -= a[i][j] * x[j];
   }
}
```

A similarly compact formulation is possible for a system of equations with an upper triangular matrix. Differences from before: the backward substitution starts in the last row and proceeds from bottom to top; dividing by the main diagonal element is necessary; the right side is already stored on `x[ ]` at the beginning; the algorithm overwrites `x[ ]` with the solution vector.

```
for (int i=n-1; i>-1; i--) {
   for (int j=i+1; j<n; j++) {
      x[i] -= a[i][j] * x[j];
   }
   x[i] /= a[i][i];
}
```

The computational effort for the forward substitution, measured by the number of multiplications and divisions, is $n^2/2 - n/2$. On the other hand, the backward substitution needs $n^2/2 + n/2$ point operations. For large n, only the quadratic term is essential. Therefore we say:

Computational effort for Forward- or backward substitution
Solving an $n \times n$ triangular system requires $O(n^2)$ floating-point operations.

Computational effort grows quadratically with the number of equations.

Question: What do you do when a system of equations is not triangular? Answer: You transform it into a triangular system. Of course, you must do this so that the original and the transformed system are equivalent (which means both systems have exactly the same solutions).

The following section shows how to do so.

3.2 Gaussian Elimination

The following box describes a basic version of Gaussian elimination.

Gaussian elimination, basic form
Given an $n \times n$ matrix A and right-hand side $\mathbf{b}$. Provided it will not divide by a zero a_{kk}, this algorithm transforms the system $A\mathbf{x} = \mathbf{b}$ to an equivalent upper triangular system $U\mathbf{x} = \mathbf{c}$.

for all columns $k = 1, \ldots n-1$
in column k eliminate all entries below the main diagonal

Elimination in column k proceeds in the following manner

for all rows $i = k+1, \ldots, n$ below the main diagonal
set $p = a_{ik}/a_{kk}$
subtract the p-th multiple of row k from row i

This subtraction is done via

for all columns $j = k, \ldots n$
$a_{ij} = a_{ij} - pa_{kj}$
for right-hand side: $b_i = b_i - pb_k$

This algorithm *overwrites* entries in A and $\mathbf{b}$ repeatedly by intermediate results and, finally, by the enties of U und $\mathbf{c}$. Entries below the main diagonal, which should be zero in U, will not be erased, however. (It will turn out later, in Section 3.5, that these entries are quite important.)

This algorithm performs $n^3/3 - n/3$ operations (counting multiplications and divisions only) to transform the matrix, and $n^2/2 - n/2$ operations to transform the right-hand side.

In JAVA source code, Gaussian elimination looks surprisingly simple. Assuming that `x[ ]` initially contains the right-hand side, the algorithm performs three nested loops so that `x[ ]` finally stores the solution vevtor.

```
for (int k=0; k<n; k++) {
   for (int i=k+1; i<n; i++) {
      double p = a[i][k] / a[k][k];
      for (int j=k+1; j<n; j++) {
         a[i][j] -= p * a[k][j];
      }
      x[i] -= p * x[k];
   }
}
```

The program does not calculate results that will be zero anyway. Thus, in step k, it will not erase the elements below the main diagonal in column k. (Actually, these entries will be necessary for the LU matrix decomposition discussed in Section 3.5.)

The stepwise backward insertion can be done with $n^2/2 + O(n)$ operations according to the program segment from the previous section. Note that te double loop in that code uses only the the upper triangle of A. Therefoer, any values $\neq 0$ below the main diagonal do not affect the computation.

These two code segments combined provide a simple equation solver.

Computational effort for basic Gaussian elimination grows cubically with the number of equations
To solve an $n \times n$ system $A\mathbf{x} = \mathbf{b}$, Gaussian elimination needs $O(n^3)$ operations.

(A precise count of multiplications and divisions results in $n^3/3 + n^2 - n/3 = n^3/3 + O(n^2)$ operations.)

3.3 Pivoting

Our basic implementation of Gaussian eliminationhas a catch: At the step $p = a_{ik}/a_{kk}$, a division by zero may occur. This is quite unlikely to happen for a matrix of randomly chosen real numbers, but Murphy's law says: *If anything can go wrong, it will.* Actually, the method fails already for systems as simple as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

because the very first operation would be a division by zero. This failure is not the fault of the equations, which have the unique solution x1 = 1, x2 = 1.

If you switch the first and second equations, the procedure runs without problems. Also, for reasons of numerical accuracy, it may be beneficial to swap equations or unknowns systematically. This procedure is called pivoting.

Gaussian elimination with complete pivoting
Given an $n \times n$-Matrix A and righ-hand side $\mathbf{b}$. This algorithm transforms the system $A\mathbf{x} = \mathbf{b}$ to upper triangular form $U\mathbf{x} = \mathbf{c}$. It overwrites the matrix A with U and the vector $\mathbf{b}$ with $\mathbf{c}$.

for all columns $k = 1, \ldots, n-1$
    find the element largest in absolute value in
    the square submatrix ranging from rows and
    columns k to n.

    switch equations and unknowns so that this
    element ends up in position a_{kk}.

    if $a_{kk} = 0$
        stop.
    else
        in column k eliminate all entries
        below the main diagonal in the same way as
        in the basic version ofGaussian Elimination

The pivotal step in Gaussian elimination is selecting which equation to use to eliminate the corresponding unknown in the remaining equations. Therefore, the element a_{kk} used in the calculation $p = a_{ik}/a_{kk}$ is called the *pivot element*. To find a favorable pivot element by suitably swapping or reordering equations and unknowns is called *pivot search* or *pivot selection*.

Row and column interchanges complicate the algorithm considerably compared to the triple loop of Gaussian elimination in its basic form. We will not list code for a complete pivot search because any possible gain of additional insight is not reasonably related to the cumbersome technical details.

Usually, equation solvers do not perform complete but only row pivot search. That means they only swap rows (=equations) for simplicity[7] . As a result, this procedure is somewhat more sensitive to numerical errors than a full pivot search.

[7]For example, the entry *Gaussian elimination* in Wikipedia lists pseudocode.

3.4 Existence of Solutions

The rule of thumb, "If there are as many equations as unknowns, there is always a solution," *is wrong*. I repeat (because I hear it again and again during exams): *IS WRONG!*

For the three linear systems

$$\begin{aligned} x + y &= 2 \\ 2x + 2y &= 4 \end{aligned} \qquad \begin{aligned} x + y &= 2 \\ x + 2y &= 3 \end{aligned} \qquad \begin{aligned} x + y &= 2 \\ 2x + 2y &= 3 \end{aligned}$$

you hopefully see with the naked eye: $x = 1, y = 1$ solves the first and the second ones. The third system is unsolvable. However, for the first system, there are infinitely many more solutions. These examples illustrate the general case.

Linearer Systems—three cases
There are three possibilities for a linear system of equations. It may have

- infinitely many solutions;
- a single unique solution;
- no solution.

(You should remember this from your Mathematics introductory lecture.)

> This section deals only with systems with the same number of equations and unknowns, but the above statement also applies to linear systems with more equations than unknowns. Only two cases are possible for fewer equations than unknowns: no solution or infinitely many solutions.

An essential property of the Gaussian elimination method is that it can distinguish the three cases

3.4.1 Elimination

Gaussian elimination with complete or row pivot search transforms the original matrix A and the right-hand side $\mathbf{b}$ into a system in *row echelon form:* From each row to the next one, the number of leading zeroes (seen from the left) increases by at least one.

Possible outcomes of the elimination procedure
The system is in row echelon form.

- Zero rows occur in A, and all corresponding entries in $\mathbf{b}$ are zero as well: *infinitely many solutions.*
- Zero rows occur in A, but at least one corresponding entry in $\mathbf{b}$ is not zero: *no solution.*
- No zero rows occur in A: *a unique solution.*

When computers perform the elimination in floating-point arithmetic, it is not so easy to check whether entries are precisely equal to zero. Due to roundoff errors in the input data and during the calculation, matrix elements that should be zero might become tiny floating point numbers. It is a delicate numerical question to decide how small an entry can be considered zero. Those dubious cases when matrix rows are nearly zero so that the algorithm just barely can find the solution are called *numerically singular.*

3.4.2 Rank of matrix and augmented matrix

The *rank of an $m \times n$ matrix* is the number of its linearly independent rows or, equivalently, columns.

> Even if the matrix has different numbers of rows and columns, there are always exactly as many linearly independent rows as columns; in short: row rank = column rank.The numbers of linearly independent rows or columns are always the same; in short: row rank = column rank.

The MATLAB command `rank(A)` determines the rank of the $n \times n$ matrix A. or columns) of the n×n matrix A, and `rank([A,b])` determines the rank of the augmented coefficient matrix (this is the linear system matrix combined with the right-hand side as its last column).

Rank determines solution set

- `rank(A)` = `rank([A,b])` $< n$: *infinitely many solutions.*
- `rank(A)` $< n$ und `rank(A)` $\neq$ `rank([A,b])`: *no solution*
- `rank(A)` $= n$: *a unique solution*

There are several methods to calculate the rank of a matrix, for example, transformation to step form by Gaussian elimination: The rank is the number of non-zero rows in the matrix. However, there is no simple decision whether a value is non-zero due to rounding errors or truly non-zero. MATLAB uses a sophisticated procedure (singular value decomposition), which provides a reliable rank estimate. If a system of equations has infinitely many solutions, you may read off the general solution from the rref([A,b]) result or use the following commands:

> `pinv(A)*b` returns a particular solution
> `null(A)` returns the *null space* of A: a list of linearly independent solutions of the *homogeneous system* $A\mathbf{x} = 0$.

The general solution is the sum of a particular solution and an arbitrary linear combination from the null space.

> `null(A,'r')` You may use `null(A,'r')` if A is a small matrix with small integer elements. This variant also returns a list of linearly independent solutions, but with "nicer"numbers, i.e., ratios of small integers. However, this method is numerically less accurate than `null(A)` . (Never let yourself be blinded by external beauty if falseness lurks behind it).

For example, consider the system $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 6 \\ -1 & -2 & -2 & -2 \\ 3 & 6 & 8 & 10 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 2 \end{bmatrix}, \quad \texttt{rref([A,b])} = \begin{bmatrix} 1 & 2 & 0 & -2 & -2 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The result of the `rref([A,b])` command means: you may choose $x_2 = \lambda$ and $x_4 = \mu$ as free parameters; $x_3 = 1 - 2\mu, x_1 = -2 - 2\lambda + 2\mu$. In vector notation,

$$\mathbf{x} = \begin{bmatrix} -2 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \\ 0 \\ -2 \\ 1 \end{bmatrix}$$

MATLAB can also compute a partial solution in the form `x=A\b` and the null space via `null(A,'r')` for a matrix as simple as this one. However, these commands ares not recommended for real-world problems. Nevertheless, here they yield (apart from a warning message) the "more beautiful" result

$$\mathbf{x} = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \\ 0 \\ -2 \\ 1 \end{bmatrix}$$

`pinv(A)*b` and `null(A)` return the numerically preferred representation,

$$\mathbf{x} = \begin{bmatrix} -0.2069 \\ -0.41379 \\ 0.034483 \\ 0.48276 \end{bmatrix} + \lambda \begin{bmatrix} -0.77069 \\ 0.10421 \\ 0.56227 \\ -0.28113 \end{bmatrix} + \mu \begin{bmatrix} -0.48335 \\ 0.54725 \\ -0.61115 \\ 0.30558 \end{bmatrix}$$

MATLAB calculates this result using sophisticated and reliable numerical procedures. However, in contrast to the previous representations of the solution, substitution into the expression $A\mathbf{x}-\mathbf{b}$ does not yield precisely zero, but due to rounding errors, values in the range of 1×10^{-15}. (Sometimes, beauty signals some sort of truth indeed.)

3.4.3 Determinant

The determinant determines wether a linear system has a unique solution.

> Linear systems $A\mathbf{x} = \mathbf{b}$ with $\det A \neq 0$ have a unique solution.
>
> However, this rule is useless for numerical computation.

For example, the MATLAB command `A=rosser` creates the 8×8 matrix

$$A = \begin{bmatrix} 611 & 196 & -192 & 407 & -8 & -52 & -49 & 29 \\ 196 & 899 & 113 & -192 & -71 & -43 & -8 & -44 \\ -192 & 113 & 899 & 196 & 61 & 49 & 8 & 52 \\ 407 & -192 & 196 & 611 & 8 & 44 & 59 & -23 \\ -8 & -71 & 61 & 8 & 411 & -599 & 208 & 208 \\ -52 & -43 & 49 & 44 & -599 & 411 & 208 & 208 \\ -49 & -8 & 8 & 59 & 208 & 208 & 99 & -911 \\ 29 & -44 & 52 & -23 & 208 & 208 & -911 & 99 \end{bmatrix}$$

For this matrix, MATLAB currently[8] computes $\det A = -9480{,}580$, so you definitely would think $\det A \neq 0$. Thus, a linear system with this matrix A should have a unique solution. However, MATLAB correctly computes the rank of A as `rank(A)`=7. Because $7 < 8$, a unique solution cannot exist. MATLAB's value for the determinant is plainly wrong.

For the 6×6matrix H, a so-called Hilbert matrix,

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \end{bmatrix}$$

[8] with version 2022b. The value for version 2021b was $\det A = -10\,611$. Previous versions around 2018 gave $\det A = -9478{,}9$; the version from 2015 gives $-9448{,}8$; prior to 2013 the value was $\det A = -13\,017$. It should worry you deeply that a numerical result varies that much depending on the program version!

MATLAB computes $\det A = 5{,}3673 \times 10^{-18}$, and this you could quite reasonably round off and interpret as $\det A = 0$. However, for the rank MATLAB (correctly) calculates `rank(H)`=6. Thus, linear systems with H do have a unique solution (although, in this case, the solution is highly sensitive to roundoff errors).

These examples illustrate:

> The numerically calculated value of a determinant says nothing about the solvability of a linear system.

3.5 *LU* decomposition

The simple Gauss elimination yields (if it does not break down) more than the transformation to a triangular shape. It can, at the same time, give the decomposition

$$A = LU$$

where L is a lower triangular matrix with ones in the main diagonal and U is an upper triangular matrix.

> ***LU* decomposition**
> Gaussian elimination without pivot search factorizes (if it does not break down) a matrix A into a product $A = LU$ of a lower triangular matrix L and an upper triangular matrix U.

In case of Gaussian elimination with pivoting, the product of the lower and upper triangular parts does not give the original matrix, but a matrix with permuted rows and columns.

The elements of L are 1 along the main diagonal, and below that equal to the multipliers $p = a_{ik}/a_{kk}$ at the corresponding positions (i, k). The elements of U are exactly those that the elimination procedure writes into the upper right triangle.

The only change in the algorithm on page 30 is remembering the intermediate results `p`. Conveniently, one can store each `p` at the position of the corresponding field element `a[i][k]`; the procedure eliminates just this entry, thus generating a zero. Instead of this zero, the algorithm stores the intermediate result `p` at this position.

Computer programs usually formulate the procedure in such a way that the original matrix A is overwritten by R and U. The upper triangle of A contains, after successful completion, the non-zero entries of U. The all-ones main diagonal of L is self-evident, no values need to be stored. Below the main diagonal of A the remaining non-zero elements of L are stored. The elegance of this storage method is that it arises in the course of the procedure quasi by itself.

See the lab notes for more information!

For the LU decomposition, you don't need a right-hand side. It comes into play later. The way to solve a system $A\mathbf{x} = \mathbf{b}$ when $A = LU$ is already available is a sequence of transformations.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} & \\ (LU)\mathbf{x} &= \mathbf{b} & \\ L(U\mathbf{x}) &= \mathbf{b} & \text{set } \mathbf{y} = U\mathbf{x} \\ L\mathbf{y} &= \mathbf{b} & \text{forward substitution solves for } \mathbf{y} \\ U\mathbf{x} &= \mathbf{y} & \text{backward substitution solves for} \mathbf{x} \end{aligned}$$

The computational process and effort are completely equivalent to standard Gaussian elimination. However, the advantage of the LU decomposition becomes apparent when solving several

systems with the same matrix A and different right-hand sides $\mathbf{b}_1, \mathbf{b}_2, \ldots$ The LU decomposition is the labor-intensive part ($\sim n^3/3$ multiplications) and has to be performed only once. The individual solutions then cost only $\sim n^2$ multiplications per right-hand side.

Special variants of Gaussian elimination exist for symmetric matrices. Exploiting the symmetry saves arithmetic operations and storage space. A possible decomposition is

$$A = LDL^T \ ,$$

with a diagonal matrix D. The *Cholesky decomposition*

$$A = LL^T$$

is the method of choice for symmetric positive definite matrices.

3.6 A Numerical Example for Gaussian Elimination

Gaussian elimination is an algorithm for systematically eliminating unknowns. For small linear systems with "nice" numbers, one often deviates from the systematic way and tries to shorten the calculation (e.g., to use already existing zeros). On the other hand, there is always the danger of calculating "around in circles," not using equations, or using them twice. Therefore, while taking shortcuts for uncomplicated systems is perfectly OK, having a well-defined algorithm for the general case is essential.

Now, given is the system $A \cdot \mathbf{x} = \mathbf{b}$ with

$$A = \begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} \quad , \quad \mathbf{b} = \begin{bmatrix} 6 \\ 6 \\ 14 \end{bmatrix}$$

You work with the *augmented coefficient matrix*

$$[A\,\mathbf{b}] = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 10 & 20 & 23 & 6 \\ 15 & 50 & 67 & 14 \end{bmatrix}$$

Eliminating the first column

(Compare the algorithm on Page 30.)

$n = 3$; in column $k = 1$ all entries below the main diagonal will be eliminated, i.e., the entries in rows $i = 2, 3$

$k = 1, i = 2$: Elimination in first column, second row

Eliminate $a_{ik} = a_{21} = 10$ using the diagonal element $a_{kk} = 5$ from row $k = 1$. Multiply first row by $a_{ik}/a_{kk} = 2$ and subtract.

10	20	23	6	
10	12	14	12	$\vert -$
0	8	9	-6	

$k = 1, i = 3$: Elimination in first column, third row

Eliminate $a_{ik} = a_{31} = 15$ using the diagonal element $a_{kk} = 5$ from row $k = 1$. Multiply first row by $a_{ik}/a_{kk} = 3$ and subtract.

15	50	67	14	
15	18	21	18	$\vert -$
0	32	46	-4	

Transformed augmented matrix

after processing first column:

$$[A\,\mathbf{b}]^{(1)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 32 & 46 & -4 \end{bmatrix}$$

Eliminating the second column

In column $k = 2$ all entries below the main diagonal will be eliminated. Here, this is the element in row $i = 3$ only.

$k = 2, i = 3$: Elimination in second column, third row

Eliminate $a_{ik} = a_{32} = 32$ using the diagonal element $a_{kk} = 8$ from row $k = 2$. Multiply second row by $a_{ik}/a_{kk} = 4$ and subtract.

$$\begin{array}{cccc|} 0 & 32 & 46 & -4 & \\ 0 & 32 & 36 & -24 & |- \\ \hline 0 & 0 & 10 & 20 & \end{array}$$

Transformed augmented matrix

Having processed the second column, Gaussian elimination is complete.

$$[A\,\mathbf{b}]^{(2)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 0 & 10 & 20 \end{bmatrix}$$

Backsubstitution

From the third row,

$$\begin{aligned} 10x_3 &= 20 \\ x_3 &= 2 \end{aligned}$$

Substitute for x_3 in second row

$$\begin{aligned} 8x_2 + 9x_3 &= -6 \\ 8x_2 + 18 &= -6 \\ 8x_2 &= -24 \\ x_2 &= -3 \end{aligned}$$

Substitute for x_2 and x_3 in first row,

$$\begin{aligned} 5x_1 + 6x_2 + 7x_3 &= 6 \\ 5x_1 - 18 + 14 &= 6 \\ 5x_1 &= 10 \\ x_1 &= 2 \end{aligned}$$

MATLAB's command `x = A\b` basically works this way, but in addition may interchange rows when selecting the pivot element.

multiple right-hand sides

To solve for several right-hand sides with the same matrix A, you augmente the matrix by all right-hand sides as additional columns and proceed as above.

Pivoting

In this example, no reordering of equations is necessary to avoid division by zero. However, a column pivot search would exchange first and third equation before the first step. Thus, the element largest in absolute value would be in position $(1, 1)$.

LU decomposition

The transformed matrix and the corresponding pivot factors also provide the LU decomposition. A right-hand side is not necessary for the LU decomposition.

$$\begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 0 & 8 & 9 \\ 0 & 0 & 10 \end{bmatrix}$$

The MATLAB command `[L, U]=lu(A)` uses the Gaussian elimination method in principle but delivers a different LU decomposition for this numerical example. The matrix U is a genuinely upper triangular matrix, but L is a *permuted* lower triangular matrix. Reason: Column pivot search switches rows in the matrix during the elimination process. (Pivoting reduces the rounding errors.)

Decomposition obtained by `[L, U]=lu(A)`

$$\begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} = \begin{bmatrix} 1/3 & 4/5 & 1 \\ 2/3 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 15 & 50 & 67 \\ 0 & -40/3 & -65/3 \\ 0 & 0 & 2 \end{bmatrix}$$

3.7 More Applications of LU decomposition

3.7.1 Determinant

Determinant
Given a decomposition $A = LU$, the determinant of A is the product of all entries along the main diagonal of U.

Proof: The determinant of a triangular matrix is always the product of all entries along its main diagonal. Since L has an all-ones main diagonal, $\det L = 1$. Now, from the properties of the determinant,

$$\det A = \det(LR) = (\det L)(\det R) = \det R\,.$$

Counting multiplications and divisions only, computing an $n \times n$ matrix determinant this way requires $n^3/3 + 2n/3 - 1$ operations.

Compare this number of operations to the computational costs of the classical method Laplace expansion along rows or columns. For an $n \times n$ matrix, let $w(n)$ be the computational cost. For this matrix, Laplace expansion computes n subdeterminants of $(n-1) \times (n-1)$ matrices and multiplies them with corresponding matrix entries. For the computational effort thus applies the recursive relationship

$$w(n) = nw(n-1) + n = n(w(n-1) + 1) \; .$$

The function $w(n)$ grows rapidly, even stronger than the factorial $n!$. A moderately fast PC (performing 10^7 multiplications per second) would compute $\det A$ for a 10×10 matrix in less than one second. However, already for a 13×13 matrix, a quarter-hour coffee break is in order. For the result in case of a 15×15 matrix you will wait two and a half days, thirteen millennia for a 20×20 matrix, and a 25×25 matrix would not be ready after 80 billion years.
The following table illustrates the rapid growth of w(n) and the comparatively small effort of the LU decomposition. It intends to point out the importance of computationally efficient algorithms and the difference between polynomial and exponential runtime.

n	$w(n)$	$n^3/3 + 2n/3 - 1$
2	2	3
3	9	10
4	40	23
5	205	44
6	1 236	75
7	8 659	118
8	69 280	175
9	623 529	248
10	6 235 300	339
15	2 246 953 104 075	1 134
20	4 180 411 311 071 440 000	2 679
25	26 652 630 354 867 072 870 693 625	5 224

3.7.2 Inverse

You will rarely need the inverse matrix A^{-1} of a given (nonsingular) matrix A explicitly. example, if some algorithm requires the vector $\mathbf{x} = A^{-1}\mathbf{y}$, you can as well solve the linear system $A\mathbf{x} = \mathbf{y}$ with less computational effort and better numerical accuracy.

Warning 1: Before calculating an inverse, ask yourself thrice whether you need the inverse explicitly.

Warning 2: If you still remember the formula known from linear algebra (the one with the determinants of the cofactors): forget it. It is of theoretical importance because it proves the existence of the inverse of a non-singular matrix. You should never (except in trivial examples) compute the inverse in this way. Consider: computational effort $O(n^5)$, if you calculate the individual subdeterminants employing LU decomposition; exponential computational effort, if you calculate determinants by Laplacian expansion.

If you can't avoid it, proceed this way. Call the first column of the inverse $\mathbf{x}_1$. The first column of the identity matrix I is the unit vector $\mathbf{e}_1 = (1, 0, \ldots, 0)^T$. By definition,

$$AA^{-1} = I \; .$$

The first column in this matrix equality is

$$A\mathbf{x}_1 = \mathbf{e}_1 \; .$$

Thus, you get the first column of the inverse by solving a linear system with the unit vector $\mathbf{e}_1$ on its right-hand side.

A straightforward generalization of this argument:

Inverse
The i-th column vector of A^{-1} solves the linear system

$$A\mathbf{x}_i = \mathbf{e}_i \ .$$

So you have to solve a linear system with multiple right-hand sides. Recipe:

decompose $A = LU$; (needs $(n^3 - n)/3$ operations).
for $i = 1, \ldots, n$
solve $LU\mathbf{x}_i = \mathbf{e}_i$; (needs n^2 operations per step).

Computational costs (counting multiplications and divisions) $(4n^3 - n)/3$.

The *Gauss–Jordan elimination* is an exceptionally well-organized variant of Gaussian elimination. It is well suited for pen-and-paper calculations. (You should know this algorithm from the introductory mathematics lectures.)

3.7.3 symmetric positive-definite matrices

Elementary arguments from linear algebra show: for symmetric positive-definite matrices, the basic variant of Gaussian elimination will never break down because of some $a_{kk} = 0$. Therefore, no pivoting is necessary. Conversely, symmetric positive-definite matrices are characterized by the property $a_{kk} > 0$ during all steps of an LU decomposition.

However, as mentioned on Page 36, in the case of symmetric positive-definite matrices, the decomposition is usually done in the form $A = LL^T$ (Cholesky decomposition) or$A = LDL^T$. Benefits: efficient implementations use less storage space and arithmetic operations as compared to a general LU decomposition.

3.7.4 Incomplete LU factorization

Even if most entries are zero in some matrix A, the factors L and U can have a much larger number of nonzero entries. Gaussian elimination introduces additional nonzeros during computation. These entries $\neq 0$ at positions where the initial matrix had elements $= 0$ are called *fill-in*. Simply ignoring all fill-in (or all fill-in entries smaller than some threshold) will significantly reduce the computational effort and memory size. Of course, then no longer $LU = A$, but $LU = \tilde{A}$ for an approximation $\tilde{A}$ to A. In this case, $LU = \tilde{A}$ is called an *incomplete factorization* (or *incomplete decomposition*). Many iterative solvers for linear systems work with incomplete factorizations.

A tiny change in the example program for the LU decomposition illustrates the basic idea (see the lab matrial for further information): Replace in the innermost loop

```
For j = k + 1 To n
   a(i, j) = a(i, j) - p * a(k, j)
Next
```

by

```
For j = k + 1 To n
   if a(i, j) <> 0 then
      a(i, j) = a(i, j) - p * a(k, j)
Next
```

Thus, the LR decomposition has become an incomplete factorization. However, the program does not save any memory space or computing time in this form. Particular data structures that store only the non-zero elements would be necessary for an efficient implementation (sparse data structures).

3.8 Sensitivity to Small Perturbations

Roundoff errors and noisy input data may change a matrix from A to $A + \delta A$ and the right-hand side from $\mathbf{b}$ to $\mathbf{b} + \delta\mathbf{b}$. The solution of this *perturbed* system will deviate by a (hopefully, small) $\delta\mathbf{x}$ from the true solution of the original system.

$$(A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \quad .$$

How does $\delta\mathbf{x}$ depend on δA and $\delta\mathbf{b}$?

Condition number
The *condition number* $\kappa(A)$ measures for the system $A\mathbf{x} = \mathbf{b}$, how the relative error of $\mathbf{x}$ depends on small relative changes in A and $\mathbf{b}$.

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \right)$$

In the inequality above, $\|\cdot\|$ denotes both a vector norm (for example, the 1-, 2-, or ∞-norm) and the corresponding matrix norm. A short calculation using the properties of the norm (see Section 2.3) shows

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

Sketch of the proof: Start with the perturbed system

$$(A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}\,,$$

expand the brackets,

$$A\mathbf{x} + A \cdot \delta\mathbf{x} + \delta A \cdot \mathbf{x} + \delta A \cdot \delta\mathbf{x} = \mathbf{b} + \delta\mathbf{b}\,;$$

since $A\mathbf{x} = \mathbf{b}$, we can cancel $A\mathbf{x}$ on the left and $\mathbf{b}$ on the righ-hand side. For small $\delta\mathbf{b}$ and δA, the product $\delta A \cdot \delta\mathbf{x}$ is small of higher order; we neglect this term. Thus,

$$A \cdot \delta\mathbf{x} + \delta A \cdot \mathbf{x} = \delta\mathbf{b}\ .$$

Make $\delta\mathbf{x}$ explicit,

$$\delta\mathbf{x} = A^{-1}\left(\delta\mathbf{b} - \delta A \cdot \mathbf{x}\right)\ ,$$

apply a vector norm on both sides,

$$\|\delta\mathbf{x}\| = \|A^{-1}\left(\delta\mathbf{b} - \delta A \cdot \mathbf{x}\right)\|\,,$$

use a property of matrix norms, inequality (9)

$$\|\delta\mathbf{x}\| \leq \|A^{-1}\| \cdot \|\left(\delta\mathbf{b} - \delta A \cdot \mathbf{x}\right)\|\,,$$

employ the triangle inequality

$$\|\delta\mathbf{x}\| \leq \|A^{-1}\|\left(\|\delta\mathbf{b}\| + \|\delta A \cdot \mathbf{x}\|\right)\ ,$$

expand the terms in the bracket

$$\|\delta\mathbf{x}\| \leq \|A^{-1}\|\left(\frac{\|A\mathbf{x}\|}{\|\mathbf{b}\|}\|\delta\mathbf{b}\| + \frac{\|A\|}{\|A\|}\|\delta A \cdot \mathbf{x}\|\right)\ ,$$

use again an inequality for matrix norms,

$$\|\delta\mathbf{x}\| \leq \|A^{-1}\| \cdot \|A\|\left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}\|\mathbf{x}\| + \frac{\|\delta A\|}{\|A\|}\|\mathbf{x}\|\right)\ ,$$

and finally, divide by $\|\mathbf{x}\|$,

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A^{-1}\| \cdot \|A\|\left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\delta A\|}{\|A\|}\right)\ .$$

Thus, the relative error in $\mathbf{x}$ can be $\kappa(A)$ times larger than the relative error in A and $\mathbf{b}$. Errors in the input data will strongly affect a system of equations whose matrix has a large condition number. Such a system is called *ill-conditioned*. Geometric illustration: two straight lines intersecting at a small angle.

The calculation of the condition number directly according to the definition would require the computation of the inverse and would be nonsensically expensive. Many equation solvers provide estimates of $\kappa(A)$ as a byproduct. For example, it holds

$$\kappa(A) \geq \frac{\max|\lambda|}{\min|\lambda|}$$

(Ratio of largest to smallest magnitude of eigenvalues; Section **??** treats eigenvalues.)

4 Iterative Solvers for Linear Systems

Let a linear system in n equations and unknowns be given.

$$\begin{array}{lllll} a_{11}x_1 + a_{12}x_2 + & \dots & +a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + & \dots & +a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + & \dots & +a_{nn}x_n & = & b_n \end{array} \tag{10}$$

In matrix notation,

$$A\mathbf{x} = \mathbf{b}\,. \tag{11}$$

Gaussian elimination is the classical solution method, at least for systems with up to several thousand equations. Chapter 3 treats it in detail. However, many applications (flow simulation, seismics, tomography, structural analysis...) produce systems with hundred thousands or millions of unknowns. Iterative solvers work well for large systems of this kind. Here you will get to know some basic methods only. They are fundamental in building more powerful iterative solvers.

4.1 Basic iterative solvers: Jacobi, Gauss-Seidel, SOR

Suppose the diagonal elements a_{ii} of an $n \times n$ matrix A are all nonzero. Then the following recipe for solving $A\mathbf{x} = \mathbf{b}$ (a fixed point method) would be possible:

Jacobi mehtod for $A\mathrm{x} = \mathrm{b}$, loosely formulated
Solve each equation for its diagonal element, set initial values, and iterate.

In more detail, using the component-wise notation (10): In row i, bring all terms except for the i-th to the right-hand side, and solve for x_i.

A correspondingly transformed 3×3 system then looks like this:

$$\begin{aligned} x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \end{aligned}$$

Suppose $\mathbf{x}^{(k)}$ is some approximate solution. The Jacobi method computes a new approximation by

$$\begin{aligned} x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11} \\ x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22} \\ x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33} \end{aligned}$$

You may find matrix notation clearer. For this, we define a matrix $D = [d_{ij}]$ with the same diagonal elements as A and zero in all off-diagonal elements. The remaining elements of A we write into a matrix E.

$$A = D + E \text{ with } D = [d_{ij}], \quad d_{ij} = \begin{cases} a_{ii} & \text{if } i = j, \\ 0 & \text{else.} \end{cases} \qquad E = A - D\,. \tag{12}$$

The linear system (11) then may be transformed,

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (D+E)\mathbf{x} &= \mathbf{b} \\ D\mathbf{x} &= \mathbf{b} - E\mathbf{x} \\ \mathbf{x} &= D^{-1}(\mathbf{b} - E\mathbf{x}) \ . \end{aligned}$$

The last equation is in fixed-point form. The corresponding fixed-point iteration

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{13}$$

is called *Jacobi method*.

Iteration step of the Jacobi method
In matrix notation for the splitting $A = D + E$:

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - E\mathbf{x}^{(k)})$$

Component-wise notation for $i = 1, \ldots, n$

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

The Jacobi method does not take advantage of the most recent information to calculate $x_i^{(k+1)}$. For example, it uses $x_1^{(k)}$ when calculating $x_2^{(k+1)}$, even though the more recent approximation $x_1^{(k+1)}$ is already available. If we formulate the method in such a way that it always uses the most current values of the x_i, we get the *Gauss-Seidel method*.

For the matrix notation of the Gauss-Seidel method, we define a matrix $C = [c_{ij}]$ with the same elements as A in and below the main diagonal, and zero above the main diagonal. We write the remaining elements of A into a matrix E:

$$A = C + E \text{ with } C = [c_{ij}], \quad c_{ij} = \begin{cases} a_{ij} & \text{if } i \geq j, \\ 0 & \text{else.} \end{cases} \qquad E = A - C \ . \tag{14}$$

The same steps that led to the fixed point equation 13 for the Jacobi method, we can repeat with the matrix C instead of D and obtain the iteration rule for the Gauss-Seidel method:

$$\mathbf{x}^{(k+1)} = C^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{15}$$

Iteration step of the Gauss-Seidel method
loosely formulated
Solve each equation for its diagonal element, set initial values, iterate using the most recently calculated approximations.

Matrix notation for splitting $A = C + E$

$$\mathbf{x}^{(k+1)} = C^{-1}(\mathbf{b} - E\mathbf{x}^{(k)})$$

Component-wise notation
for $i = 1, \ldots, n$

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

It is possible to accelerate the Gauss-Seidel method considerably if the new value $x_i^{(k+1)}$ is not used directly, but in combinatio with the old value in the form $\omega x_i^{(k+1)} + (1-\omega) x_i^{(k)}$ with some extrapolation factor $\omega > 1$. This iterative procedure is called the *SOR method* (SOR stands for successive overrelaxation). However, it is difficult to give a suitable value for ω. The theory says $1 \leq \omega < 2$ with values relatively close to 2. For $\omega = 1$, SOR reduces to Gauss-Seidel.

Iteration step of the SOR method
loosely formulated
For each i, calculate first an intermediate result $y_i^{(k+1)}$ via a Gauss-Seidel step; get the new value by extrapolation (overrelaxation)from old value and intermediate result: $x_i^{(k+1)} = \omega y_i^{(k+1)} + (1 - \omega) x_i^{(k)}$

The component-wise notation already looks a bit confusing here.

for $i = 1, \ldots, n$

$$y_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}$$

$$x_i^{(k+1)} = \omega y_i^{(k+1)} + (1 - \omega) x_i^{(k)}$$

This method can also be written with a decomposition $A = B + E$ similar to equations 13 and 15,

$$\mathbf{x}^{(k+1)} = B^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{16}$$

For SOR, B is a combination of the matrices C and D from before (12, 14) in the form

$$B = C + \left(\frac{1}{\omega} - 1 \right) D$$

4.2 Convergence Criteria for the Jacobi and Gauss-Seidel Methods

The three methods presented above will not necessarily converge for an arbitrary Matrix A. However, it is possible to prove the convergence of the Jacobi method by showing that the fixed point iteration is a contraction mapping. For this purpose, we define

An $n \times n$matrix $A = [a_{ij}]$ is called *strictly diagonally dominant*, if

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad \text{for} \quad i = 1, 2, \ldots, n$$

Thus, in each row the sum of the absolute values of the non-diagonal elements must be smaller than the absolute value of the diagonal element.

Convergence of Jacobi Method
For linear systems with strictly diagonally dominant matrices the Jacobi method converges to the unique solution.

Proof: We show that the function $\mathbf{\Phi}(\mathbf{x}) = D^{-1}(\mathbf{b} - E\mathbf{x})$, which defines the iteration (13), is a contration mapping in the maximum norm for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. According to section 2.4, this ensures convergence.

$$\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y}) = D^{-1}(\mathbf{b} - E\mathbf{x}) - D^{-1}(\mathbf{b} - E\mathbf{y}) = D^{-1}E(\mathbf{y} - \mathbf{x})$$

Row i of the matrix $D^{-1}E$ is

$$\frac{a_{i1}}{a_{ii}} \quad \frac{a_{i2}}{a_{ii}} \quad \cdots \quad \frac{a_{i,i-1}}{a_{ii}} \quad 0 \quad \frac{a_{i,i+1}}{a_{ii}} \quad \cdots \quad \frac{a_{in}}{a_{ii}}$$

The sum of the absolute values in this row is < 1 (diagonal dominance ensures that the nominator is smaller than the denominator). Since this holds for all rows of $D^{-1}E$, the maximum norm (infiniyty norm) fulfills

$$\|D^{-1}E\|_\infty < 1 \ .$$

A property of the norm, inequality (9), immediately gives the contraction property.

$$\|\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y})\|_\infty = \|D^{-1}E(\mathbf{y} - \mathbf{x})\|_\infty \leq \|D^{-1}E\|_\infty \cdot \|\mathbf{y} - \mathbf{x}\|_\infty \leq C\|\mathbf{y} - \mathbf{x}\|_\infty$$

mit $C = \|D^{-1}E\|_\infty < 1$.

With considerably more effort convergence of the Jacobi method can be shown also for a larger class of matrices (weakly diagonally dominant, irreducible matrices). This statement is important, because many real-world problems yield exactly such matrices. For the sake of completeness here are the definitions:
An $n \times n$ matrix $A = [a_{ij}]$ is *weakly diagonally dominant*, if

$$|a_{ii}| \geq \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad \text{for} \quad i = 1, 2, \ldots, n,$$

and at least for one i strict inequality holds. To check for irreducibility, you draw a point for each i. For each matrix entry $a_{ij} \neq 0$ in A you join points i and j by an arrow $i \to j$. This drawing represents a *directed graph* If you can reach, following the arrows, any point starting from any other point, then this graph is *connected*, and the corresponding matrix A is *irreducible*.

As a rule, the Gauss-Seidel method converges more rapidly than the Jacobi method. It typically needs only half as many iterations for the same accuracy. The SOR method with optimally chosen relaxation parameter ω needs $O(\sqrt{N})$ iterations, where the Jacobi method needs N iterations. However, there are also matrices for which one method converges, but the other does not. We quote here without proof two more theorems formulating convergence conditions.

If A has positive elements in the main diagonal and all other elements are ≤ 0, then the Gauss-Seidel method converges if and only if the Jacobi method converges. If both methods converge, then the Gauss-Seidel method is asymptotically faster (*theorem of Stein and Rosenberg*).

If A is symmetric positive definite, then the Gauss-Seidel method converges.

4.3 Modern Iterative Solvers

Linear systems from flow simulation, structural analysis, financial mathematics, and many other fields may easily reach a size of several million unknowns. However, only a few elements per matrix row are different from zero. (Such a matrix is called *sparse*). Today, almost exclusively iterative methods are used to solve such systems. The classical methods (Jacobi, Gauss-Seidel) converge too slowly and therefore require too much computational effort.

These notes can only give an introductory overlook to some ideas used by modern iterative solvers.

4.3.1 Matrix Splitting, Preconditioning

Let us assume you want to solve the system

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

A clever idea: You replace the matrix A by another matrix $\tilde{A}$ for which you can the linear system more quickly. You can make it easy for yourself and choose for $\tilde{A}$ the unit matrix I, or the diagonal part of A, or selectively only certain matrix elements out of A.

Write $A = \tilde{A}+E$. This is called a *splitting* of A into an approximation, called the *preconditioner*, and a residual part E. You then reformulate the original system as a fixed point problem.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (\tilde{A}+E)\mathbf{x} &= \mathbf{b} \\ \tilde{A}\mathbf{x}+E\mathbf{x} &= \mathbf{b} \\ \tilde{A}\mathbf{x} &= \mathbf{b}-E\mathbf{x} \\ \mathbf{x} &= \tilde{A}^{-1}(\mathbf{b}-E\mathbf{x}) \end{aligned}$$

The Jacobi method uses this idea with $\tilde{A} = D$, the diagonal part. Alternatively, you will get $\tilde{A}$ for the Gauss-Seidel method if you set in A all elements above the main diagonal to zero.

In general, the better $\tilde{A}$ approximates the original matrix, the faster such an iterative procedure converges.

Particularly good splittings result from incomplete LU decomposition. These methods are discussed in section 3.7.4.

However, you should not implement the method in the fixed-point form above since one should explicitly compute the matrix $\tilde{A}^{-1}$ in the simplest cases only (such as $\tilde{A} = D$). An algebraically equivalent form but suitable for computers is

Basic iterative solver with $A = \tilde{A} + E$
For a suitable splitting $A = \tilde{A} + E$, an arbitrary initial vektor $\mathbf{x}^{(0)}$ and some given accuracy threshold $\epsilon > 0$ this algorithm approximately solves $A\mathbf{x} = \mathbf{b}$.

start with initial vector $\mathbf{x}^{(0)}$
set $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$
for $k = 0, 1, 2, \ldots$
  solve $\tilde{A}\mathbf{d}^{(k+1)} = \mathbf{r}^{(k)}$
  set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{d}^{(k+1)}$
  set $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - A\mathbf{d}^{(k+1)}$
until $\|\mathbf{r}^{(k+1)}\| < \epsilon$
result: approximate solution $\mathbf{x}^{(k+1)}$

In an iterative solver of this type, $\tilde{A}$ is called the *preconditioner*.

For a vector $\mathbf{x}$ and given A and $\mathbf{b}$, the expression $\mathbf{b} - A\mathbf{x}$ is called the *residual*. Thus, solving a linear system $A\mathbf{x} = \mathbf{b}$ is equivalent to finding an $\mathbf{x}$ with a vanishing residual. One can easily verify that the vectors $\mathbf{r}^{(k)}$ in the above basic scheme are indeed the respective residuals $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$. Thus, the stopping criterion of the method requires the residual norm to be smaller than a given bound.

Caution! A small residual does not automatically mean that also the error $\mathbf{x}_{\text{exc}} - \mathbf{x}^{(k)}$ between exact and approximate solution is small. For example, if A is symmetric, the following bounds hold.

$$\frac{\|\mathbf{r}^{(k)}\|_2}{|\lambda_{\max}|} \leq \|\mathbf{x}_{\text{exc}} - \mathbf{x}^{(k)}\|_2 \leq \frac{\|\mathbf{r}^{(k)}\|_2}{|\lambda_{\min}|}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ denote A's largest and smallest eigenvalues, respectively. Therefore, if $\lambda_{\min}$ is close to zero, a small residual does not say much about the size of the error.

Likewise, small corrections $\mathbf{d}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ will not automatically guarantee the smallness of the error. However, modern iterative methods approximate the eigenvalues with little additional effort and thus provide reliable bounds for the error.

4.3.2 Minimizing the residual to accelerate convergence

One often observes with the above basic scheme that the vectors $\mathbf{d}^{(k)}$, by which the approximate solution vectors change per iteration, point in the right direction but not with the correct magnitude. Instead of changing the vector $\mathbf{x}^{(k)}$ by only the vector $\mathbf{d}^{(k+1)}$ per iteration step, one can therefore try to apply a multiple ω of this correction. (The SOR method already used a similar approach.)

If the approximation vector changes by $\omega\mathbf{d}^{(k+1)}$ at the step from k to $k+1$, then it is easy to see that the residual vector changes by $-\omega A\mathbf{d}^{(k+1)}$. Accordingly, one changes the basic scheme and sets

$$\begin{aligned} \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \omega\mathbf{d}^{(k+1)} \\ \mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)} \end{aligned}$$

At each step, one chooses ω so that the magnitude $\|\mathbf{r}^{(k+1)}\|$ becomes as small as possible. How does this work? The usual procedure when looking for an extreme value is to differentiate and

set the derivative to zero. (Here, $\|\cdot\|$ always denotes the 2-norm, that is, the Euclidean length of a vector.)

$$\begin{aligned}\|\mathbf{r}^{(k+1)}\|^2 &= (\mathbf{r}^{(k+1)} \cdot \mathbf{r}^{(k+1)}) \\ &= \Big((\mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}) \cdot (\mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)})\Big) \\ &= \Big(\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)} - 2\omega(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + \omega^2(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)})\Big)\end{aligned}$$

The individual inner products are all just scalar constants. Differentiation with respect to ω and setting the derivative to zero yields

$$\begin{aligned}0 &= \frac{d}{d\omega}\|\mathbf{r}^{(k+1)}\|^2 \\ &= \frac{d}{d\omega}\Big(\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)} - 2\omega(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + \omega^2(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)})\Big) \\ &= -2(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + 2\omega(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)}) \quad \text{, daraus} \\ \omega &= \frac{\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}}{A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)}}\end{aligned}$$

4.3.3 Orthogonalizing search directions to accelerate convergence

We have set

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}$$

where ω is chosen optimally so that it minimizes the norm $\|\mathbf{r}^{(k+1)}\|$. Thus, any further change of the residual vector in direction $A\mathbf{d}^{(k+1)}$ will increase its norm. Now, if in the next iteration

$$\mathbf{r}^{(k+2)} = \mathbf{r}^{(k+1)} - \omega A\mathbf{d}^{(k+2)}$$

the correction $A\mathbf{d}^{(k+2)}$ contains a component in direction $A\mathbf{d}^{(k+1)}$, any amount of correction in that direction will deteriorate the approximation.

Therefore: If the residual is already minimized along some direction, it should not be changed in this direction any more. So we need a method that removes the undesired component from the new search direction $A\mathbf{d}^{(k+2)}$. This can be achieved by *orthogonalization.*

Let $\mathbf{p}$ and $\mathbf{q}$ be two vectors $\neq 0$. The component of $\mathbf{p}$ in direction $\mathbf{q}$ is given by

$$\left(\frac{\mathbf{p} \cdot \mathbf{q}}{\mathbf{q} \cdot \mathbf{q}}\right) \mathbf{q}$$

Thus, the vector

$$\mathbf{p} - \left(\frac{\mathbf{p} \cdot \mathbf{q}}{\mathbf{q} \cdot \mathbf{q}}\right) \mathbf{q}$$

no longer contains a component in direction $\mathbf{q}$, i.e., it is orthogonal to $\mathbf{q}$. (Special case: if $\mathbf{p}$ is a scalar multiple of $\mathbf{q}$, this calculation gives the zero vector.)

This procedure can successively remove from $\mathbf{p}$ components of several vectors.

Orthogonalization
Given m nonzero vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m \in \mathbb{R}^n$ and a vector $\mathbf{p} \in \mathbb{R}^n$. This algorithm removes from $\mathbf{p}$ all components in directions $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m$.

for $i = 1 \ldots m$
  compute inner product $r_i = \mathbf{p} \cdot \mathbf{q}_i / \mathbf{q}_i \cdot \mathbf{q}_i$
  subtract component by $\mathbf{p} = \mathbf{p} - r_i \mathbf{q}_i$

An early application of these ideas (preconditioning, minimization, orthogonalization) is the ORTHOMIN algorithm (published in 1976 by P. K. W. Vinsome, then employed by a petroleum company). Since then, many methods based on similar principles have been developed. They all belong to the class of what is now called *Krylov-subspace methods*.

A very elegant and powerful method exists for symmetric positive definite matrices, the conjugate gradient method (Hestenes and Stiefel, 1952). Combined with suitable preconditioning, it is no routinely used for large linear systems.

For unsymmetric matrices, GMRES (for *generalized minimal residual method*), BiCG (for *biconjugate gradients*) or CGS (for *conjugate gradient squared*) are common methods.