Numerical Linear Algebra

Karl Meerbergen Marc Van Barel Andrey Chesnokov
Yvette Vanberghen

June 14, 2010

Contents

1 Motivation **5**

1.1 Boundary value problem . . . 5
1.1.1 Finite differences in one dimension . . . 5
1.1.2 Finite elements in one dimension . . . 6
1.1.3 Finite differences in two dimensions . . . 7
1.1.4 Finite differences in three dimensions . . . 8
1.1.5 Integral equations in one dimension . . . 9
1.2 Partial differential equations . . . 10
1.3 System of differential equations . . . 10
1.4 Constrained Optimization . . . 12
1.5 The Laplacian of a graph . . . 13
1.6 Overview of methods . . . 13
1.6.1 Direct methods . . . 14
1.6.2 Iterative methods . . . 14
1.6.3 Splitting methods . . . 15
1.6.4 Incomplete factorization . . . 15
1.6.5 Sparse approximate inverse . . . 15
1.6.6 Multi-grid methods . . . 15
1.6.7 Algebraic multi-grid methods . . . 15
1.6.8 Structured matrices . . . 16
1.6.9 Domain decomposition . . . 16

2 Direct methods for sparse systems **17**

2.1 Gaussian elimination . . . 17
2.1.1 Transformation to upper triangular form . . . 17
2.1.2 The six forms . . . 19
2.2 Symmetric matrices . . . 20
2.3 Pivoting for numerical stability . . . 20
2.4 Banded matrices . . . 21
2.5 Variable band or skyline matrices . . . 22
2.6 Frontal methods . . . 23
2.7 Sparse matrices . . . 26
2.7.1 Storage formats . . . 26
2.7.2 Sparse matrices and graphs . . . 27

2.7.3 The clique concept . . . 28
2.7.4 The elimination tree . . . 28
2.7.5 Sparse arithmetic . . . 28
2.8 Multi-frontal methods . . . 28
2.9 The effect of renumbering the unknowns . . . 29
2.9.1 Reverse Cuthill McKee ordering . . . 30
2.9.2 Minimum degree ordering . . . 31
2.9.3 Approximate minimum degree ordering . . . 31
2.9.4 Nested bisection . . . 32
2.9.5 Partial pivoting . . . 32
2.9.6 Reordering software . . . 33

3 Krylov methods for linear systems 34
3.1 Krylov spaces and linear solvers . . . 34
3.2 Other methods for symmetric systems . . . 36
3.3 Nonsymmetric matrices . . . 37
3.4 Arnoldi based methods . . . 37
3.5 Lanczos based methods . . . 38

4 Incomplete factorization preconditioning 40
4.1 Introduction . . . 40
4.2 The classical incomplete factorization . . . 41
4.3 MILU, RILU . . . 43
4.4 Alternatives to MILU . . . 44
4.5 ILU(p) . . . 44
4.6 ILUT . . . 45

5 Multigrid 47

6 Domain decomposition 50
6.1 Nonoverlapping methods . . . 51
6.2 Overlapping methods . . . 53

7 Large scale eigenvalue problems 57
7.1 Perturbation analysis . . . 58
7.2 Simple eigenvalue solvers . . . 58
7.2.1 The power method . . . 58
7.2.2 Inverse iteration . . . 59
7.2.3 Newton's method and variations . . . 59
7.3 Subspace methods . . . 60
7.3.1 The Arnoldi method . . . 60
7.3.2 The Lanczos method . . . 61
7.3.3 Spectral transformation . . . 61
7.3.4 The Jacobi-Davidson method . . . 62

7.4 Implicit restarting for Lanczos's method 62
7.5 Example 65

8 Modelreduction methods **67**
8.1 The Padé via Lanczos method 68
8.2 Balanced truncation 68
8.3 Balanced truncation by Cholesky factorization 71
8.4 Large scale balanced truncation 71

9 Software issues for linear algebra **73**
9.1 Languages 73
9.2 Basic linear algebra 73
9.3 Software packages 76
9.3.1 LAPACK 76
9.3.2 Sparse direct solvers 76
9.3.3 Iterative linear system solvers 77
9.3.4 vector- and matrix packages 77
9.4 More about Fortran 90 78
9.5 More about Java 78
9.6 More about C++ 79
9.6.1 Mixing with FORTRAN 79

Chapter 1

Motivation

The next chapters discuss methods for the solution of linear systems of equations arising from applications including differential equations, integral equations, optimization, graph applications among others. Some of these applications are discussed in this chapter.

Most material comes from the following books: [12] [16].

1.1 Boundary value problem

Often the solution of a differential equation also has to satisfy a set of additional constraints, called boundary conditions. More generally a problem that has a set of boundary conditions is called a boundary problem. A boundary value problem is a problem, typically a differential equation, that has a set of additional restraints, called the boundary conditions. A solution to a boundary value problem is a solution to the differential equation which also satisfies the boundary conditions.

Boundary value problems arise in several branches of physics as any physical differential equation will have them. In this section some boundary value problems and their solution are discussed.

1.1.1 Finite differences in one dimension

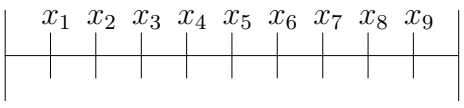
Consider the Poisson equation in one dimension:

$$-\frac{d^2u(x)}{dx^2} = f(x), \quad 0 < x < 1 \tag{1.1}$$

$$u(0) = u(1) = 0 \tag{1.2}$$

A solution is found by discretization, i.e. the continuous variable x is discretized by writing it as a sum of base functions.

The following discretization is used. Partition the interval $(0, 1)$ into $N + 1$ equal parts and define by $h = (N + 1)^{-1}$ the distance between the points x_j:



The solution is computed (approximately) in these $N = 9$ points. The derivatives can be approximated as follows:

$$\frac{du}{dx}(x_j) \simeq \frac{u(x_j) - u(x_{j-1})}{h} \tag{1.3a}$$

$$\frac{du}{dx}(x_j) \simeq \frac{u(x_{j+1}) - u(x_j)}{h} \tag{1.3b}$$

$$\frac{d^2u}{dx^2}(x_j) \simeq \frac{u(x_{j+1}) - 2u(x_j) + u(x_{j-1})}{h^2} \tag{1.3c}$$

Replacing the derivatives in (1.1) by (1.3), we obtain the following system of linear equations:

$$\frac{1}{h^2}\begin{bmatrix} 2 & -1 & & 0 \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2 \end{bmatrix}\begin{pmatrix} u_1 \\ \vdots \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} f_1 \\ \vdots \\ \vdots \\ f_N \end{pmatrix} \tag{1.4}$$

This system is sparse. It actually is a tridiagonal matrix. There is also structure in this matrix: it is symmetric, banded, and Toeplitz.

1.1.2 Finite elements in one dimension

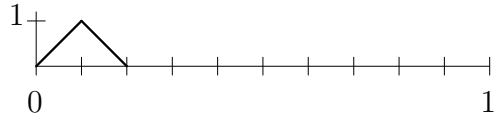
The Poisson equation of the previous section can also be solved using a finite elements approach. To this end we write the solution as a linear combination of base functions

$$u(x) = \sum_{j=1}^{N} \mu_j \phi_j(x) \tag{1.5}$$

where ϕ_j has local support, i.e. it only differs from zero in a few 'elements', and where ϕ_j has orthogonality properties, i.e.

$$\int_0^1 \phi_j(x)\phi_i(x)dx = 0$$

for most i, j combinations.



The equation is first written in variational form. Define a test function t that satisfies the boundary condition and require that

$$\int_0^1 t(x)\left(f(x) + \frac{d^2u(x)}{dx^2}\right)dx = 0\ .$$

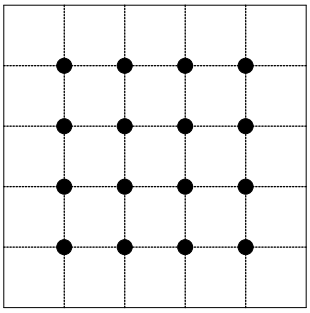


Figure 1.1: Square mesh for the Poisson equation

Using simple integration rules and taking into account the boundary conditions, we find

$$\int_0^1 \left(t(x)f(x) - \frac{dt(x)}{dx}\frac{du(x)}{dx} \right) dx = 0 \ .$$

When we plug in (1.5) and use the N test functions $t = \phi_j$, we get the system of linear equations:

$$\begin{bmatrix} \int_0^1 \phi_1'\phi_1' dx & \cdots & \int_0^1 \phi_1'\phi_j' dx & \cdots \\ \vdots & & & \vdots \\ \vdots & & & \vdots \\ \int_0^1 \phi_N'\phi_1' dx & \cdots & \int_0^1 \phi_N'\phi_j' dx & \cdots \end{bmatrix} \begin{pmatrix} \mu_1 \\ \vdots \\ \vdots \\ \mu_N \end{pmatrix} = \begin{pmatrix} \int_0^1 \phi_1 f\phi \\ \vdots \\ \vdots \\ \int_0^1 \phi_N f \end{pmatrix} \tag{1.6}$$

Thanks to the orthogonality properties, the linear system is sparse.

1.1.3 Finite differences in two dimensions

The solution is wanted for

$$-\frac{\partial^2 u(x,y)}{\partial x^2} - \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y), \quad 0 < x < 1, \quad 0 < y < 1 \tag{1.7}$$

$$u(0,y) = u(1,y) = u(x,0) = u(x,1) = 0, \quad 0 < x < 1, \quad 0 < y < 1 \ . \tag{1.8}$$

A numerical solution can be found by discretizing the interval $(0,1)$ in $N+1$ equal parts with distance $h = (N+1)^{-1}$ between the points (x_i, y_j); see Figure 1.1. The solution is computed in those $N \times N = 16$ points. The derivatives are approximated as follows:

$$\frac{\partial^2 u}{\partial x^2}(x_j, y_i) \simeq \frac{u(x_{j+1}, y_i) - 2u(x_j, y_i) + u(x_{j-1}, y_i)}{h^2} \tag{1.9a}$$

$$\frac{\partial^2 u}{\partial y^2}(x_j, y_i) \simeq \frac{u(x_j, y_{i+1}) - 2u(x_j, y_i) + u(x_j, y_{i-1})}{h^2} \tag{1.9b}$$

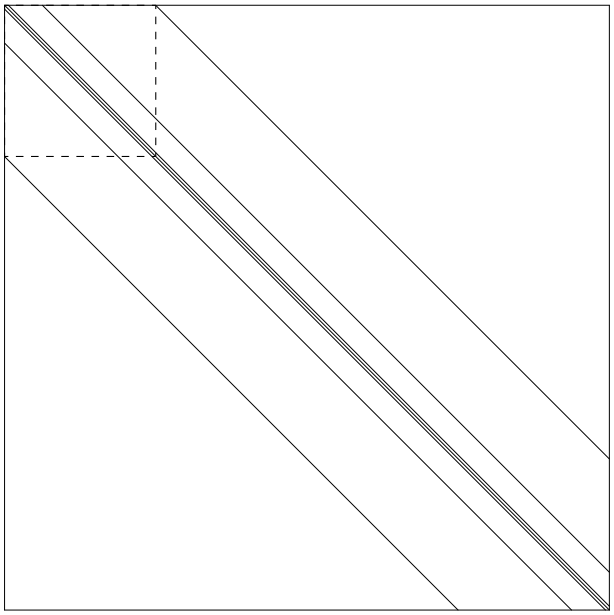


Figure 1.2: Nonzero pattern of the matrix for a 3D discretization of the Poisson equation

Replacing the derivatives in (1.7) by (1.9) we obtain

$$\frac{1}{h^2}\left[\begin{array}{cccc|cccc|cccc|cccc}
4 & -1 & & & -1 & & & & & & & & & & & \\
-1 & 4 & -1 & & & -1 & & & & & & & & & & \\
 & -1 & 4 & -1 & & & -1 & & & & & & & & & \\
 & & -1 & 4 & & & & -1 & & & & & & & & \\
\hline
-1 & & & & 4 & -1 & & & -1 & & & & & & & \\
 & -1 & & & -1 & 4 & -1 & & & -1 & & & & & & \\
 & & -1 & & & -1 & 4 & -1 & & & -1 & & & & & \\
 & & & -1 & & & -1 & 4 & & & & -1 & & & & \\
\hline
 & & & & -1 & & & & 4 & -1 & & & -1 & & & \\
 & & & & & -1 & & & -1 & 4 & -1 & & & -1 & & \\
 & & & & & & -1 & & & -1 & 4 & -1 & & & -1 & \\
 & & & & & & & -1 & & & -1 & 4 & & & & -1 \\
\hline
 & & & & & & & & -1 & & & & 4 & -1 & & \\
 & & & & & & & & & -1 & & & -1 & 4 & -1 & \\
 & & & & & & & & & & -1 & & & -1 & 4 & -1 \\
 & & & & & & & & & & & -1 & & & -1 & 4
\end{array}\right]
\left(\begin{array}{c} u_{11} \\ u_{12} \\ u_{13} \\ u_{14} \\ \hline u_{21} \\ u_{22} \\ u_{23} \\ u_{24} \\ \hline u_{31} \\ u_{32} \\ u_{33} \\ u_{34} \\ \hline u_{41} \\ u_{42} \\ u_{43} \\ u_{44} \end{array}\right)
=
\left(\begin{array}{c} f_{11} \\ f_{12} \\ f13 \\ f_{14} \\ \hline f_{21} \\ f_{22} \\ f_{23} \\ f_{24} \\ \hline f_{31} \\ f_{32} \\ f_{33} \\ f_{34} \\ \hline f_{41} \\ f_{42} \\ f_{43} \\ f_{44} \end{array}\right) \tag{1.10}$$

This system is sparse again. The halve bandwidth is $N = 4$. It is no longer a tridiagonal matrix but it has a tridiagonal block structure. The conclusion is similar for the finite element method.

1.1.4 Finite differences in three dimensions

The equation and method are similar for three dimensions. After discretization, we obtain a system whose structure is shown in Figure 1.2. This structure is again sparse, but the halve bandwidth now is N^2. This has great implications for the solution of 3D problems. Compare the results from Table 1.1.

Table 1.1: Results for the solution of Poisson's equation by a multi-frontal direct method (MUMPS)

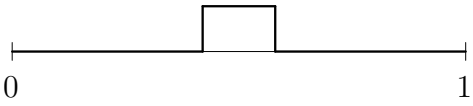
	N	order matrix	core memory (MB)	wall clock time (s)
1D	250000	250000	72	3
2D	500	250000	195	6
3D	63	250047	2020	108

1.1.5 Integral equations in one dimension

Suppose we want the solution of the Fredholm-equation

$$\int_0^1 |x-y|^{-1} u(y) dy = f(x) \ .$$

We again use a discretization. Approximate $u(x) = \sum_{j=1}^{N} \mu_j \phi_j$.



By a variational formulation, the Fredholm-equation is transformed into

$$\int_0^1 t(x) \left(f(x) - \int_0^1 |x-y|^{-1} u(y) dy \right) dx \ . \tag{1.11}$$

By choosing $t = \phi_1, \ldots, \phi_N$, (1.11) becomes a system of equations:

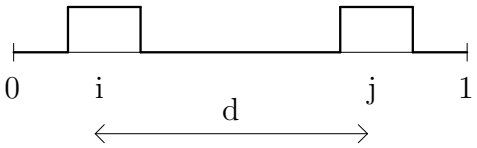
$$A \begin{pmatrix} \mu_1 \\ \vdots \\ \mu_N \end{pmatrix} = \begin{pmatrix} \int_0^1 \phi_1(x) f(x) dx \\ \vdots \\ \int_0^1 \phi_N(x) f(x) dx \end{pmatrix}$$

where the i,jth element of A is:

$$A_{ij} = \int_0^1 \phi_i(x) \left(\int_0^1 |x-y|^{-1} \phi_j(y) dy \right) dx \ .$$

When $i = j$, these are improper integrals. Usually, the kernel function is adapted to remove the singularity. The matrix A is full.

The base functions only differ from zero in a small interval, as the finite element method. When the respective intervals of ϕ_i and ϕ_j lie far away from each other,$|A_{ij}|$ is small since $A_{ij} = d^{-1} \int_0^1 \phi_i \int_0^1 \phi_j$.



As a consequence, the matrix A can be approximated by a sparse matrix which can then be used as a preconditioner for A in an iterative method.

1.2 Partial differential equations

Consider for example, the wave equation in a 2-D domain Ω

$$\begin{aligned} \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} - (\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) &= 0 \quad \text{in} \quad \Omega & (1.12)\\ \frac{\partial u}{\partial r} &= \alpha u \quad \text{on} \quad \partial\Omega & (1.13)\\ u(0,x) &= u_0(x) & (1.14) \end{aligned}$$

We only discretize the spatial variables x and y. In this way, we obtain a system of differential equations in t.

1.3 System of differential equations

The solution of the (linear) system of differential equations

$$\begin{aligned} B\dot{x} + Ax &= f & (1.15a)\\ x(0) &= x_0 & (1.15b) \end{aligned}$$

by an implicit method requires the solution of the sequence of systems of linear equations. Such a linear system, in general, takes the form

$$(B + \delta A)y = b \tag{1.16}$$

where δ is a parameter depending on the method and the time step. Higher order methods require the solution of several systems per timestep.

Differential equations are sometimes solved in the frequency domain. The equation is first transformed by the Fourier or Laplace transform into

$$(sB + A)\widehat{x} = \widehat{f} \tag{1.17}$$

where the vectors with a hat $\widehat{(\cdot)}$ represent the Laplace of Fourier transformations. Usually, $s = i\omega$ is imaginary. The advantage of this technique is that no differential equation has to be solved, but a system of algebraic equations.

Although, they are much alike, we shall see there is a big difference between the efficient solution of (1.17) and (1.16).

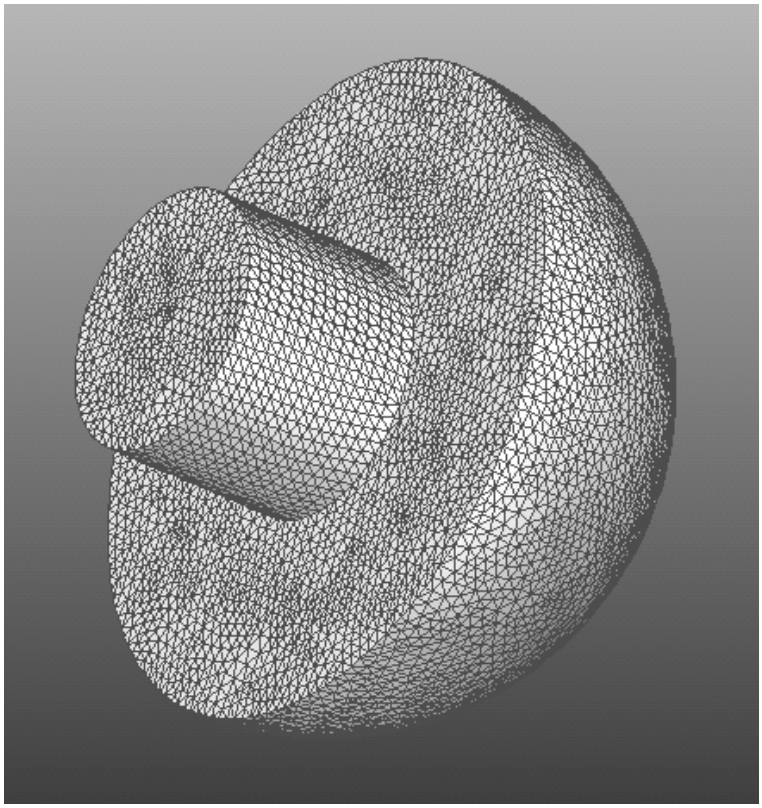


Figure 1.3: Mushroom-mesh by Airbus

Example 1.3.1 (Source: Airbus – Free Field Technologies) *The following problem originates from Airbus in a study of iterative methods for acoustics. The mesh is shown in Figure 1.3. It is a mushroom shaped volume, where the spherical part (half sphere) is covered by infinite elements. On the bottom of the mushroom is a unit acceleration boundary condition. The discrete mesh consists of 309,642 linear tetrahedral elements and 8764 infinite elements of order 5.*

We solve (1.16) using BiCGStab with ILU-1 preconditioning. Time integration by the implicit Euler method with time step $\delta = 4 \cdot 10^{-3}$ *requires 589 iterations with a CPU time of 52 seconds on a Dell Dimension 4550. The same problem in the frequency domain 1.17 with* $s = i500$ *does not converge.*

1.4 Constrained Optimization

Consider the following quadratic program with linear constraints:

$$\min \quad \frac{1}{2}x^*Ax - d^*x \tag{1.18}$$

$$Cx = b \ . \tag{1.19}$$

The introduction of a vector of Lagrange multipliers λ rewrites this problem as

$$\min \frac{1}{2}x^*Ax - d^*x + \lambda^*(Cx - b) \ .$$

Derivation to x and λ produces

$$\begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix} \begin{pmatrix} x \\ \lambda \end{pmatrix} = \begin{pmatrix} d \\ b \end{pmatrix} \ . \tag{1.20}$$

The solution of (1.20) is a so-called saddle point of the optimization problem. This kind of linear systems is considered to be one of the most difficult problems because of the zero block on the main diagonal. When A and C have full rank, the matrix in (1.20) is indefinite. Indefinite problems are usually hard to solve by an iterative method. Even direct methods for sparse matrices have to take into account this special structure in order to reduce the backward error.

Systems of the form (1.20) also appear in the solution of incompressible Stokes and Navier-Stokes equations. The constraints correspond to the incompressibility condition of the fluid.

There are solution techniques that relax the constraint (1.19) by the introduction of a matrix D:

$$\begin{bmatrix} A & C^T \\ C & D \end{bmatrix} \begin{pmatrix} x \\ \lambda \end{pmatrix} = \begin{pmatrix} d \\ b \end{pmatrix} \ . \tag{1.21}$$

A good choice of D makes the iterative solution easier. We have to be careful that the nature of the problem does not disappear. The matrix of (1.21) can also be used as a preconditioner for (1.20).

model	points	triangles	boundary[2]	weights[3]	UMFPACK	BICGSTAB[4]	total[5]
VW_9K	9 121	17 842	0.69s	1.06s	0.19s	0.9s (44iter)	1.94s
HEAD_DRAGON	30 610	61 148	2.21s	4.1s	0.97s	6.8s (96iter)	7.28s
BUNNY	31 577	62 916	2.4s	4.18s	1.23s	8.9s (107iter)	7.81s
HALF_DRAGON[1]	58 260	116 396	4.2s	7.72s	2.14s	18.1s (140iter)	14.06s
FACE_STRIP	59 392	116 163	7.16s	9.45s	1.36s	7.63s (50iter)	17.97s
ISIS	185 921	371 691	18.35s	30.18s	12.64s	84.7s (223iter)	61.17s
HALF_DRAGON	255 510	510 760	19.35s	34.6s	11.66s	224s (433iter)	65.61s
HOUSE	480 852	961 194	40.13s	70s	39s	374s (380iter)	149.13s

[1] the decimated version

[2] time needed for the extraction of the boundary of the point cloud

[3] this is the time needed to construct the parameterization matrix

[4] with $ILUT(15, 1e-2)$ preconditioner

[5] total time required to compute the parameterization using UMFPACK

Table 1.2: Comparative results from point cloud processing

1.5 The Laplacian of a graph

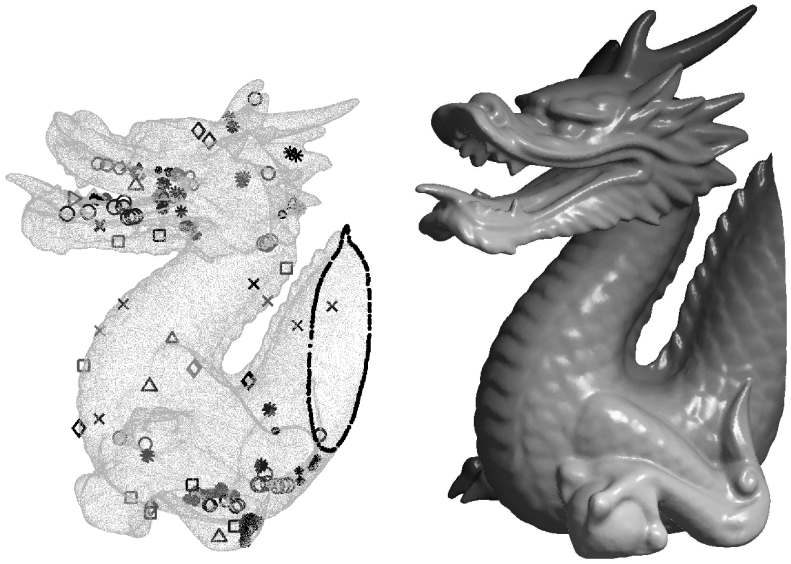
A graph is a set of vertices, connected by edges. Graphs appear in many applications, some examples are parallel computing for sparse matrices or finite element problems, point cloud processing and meshing algorithms.

The Laplacian L is a matrix where the i, jth entry is the weight of the edge from vertex i to vertex j. This weight can be a cost function, or a probability for example. The main diagonal elements are chosen such that the row sums of L are zero. So, L is a sparse matrix, unless the graph is a clique graph. For an undirected graph, the sparse structure is symmetric. Although L is a singular matrix, linear systems derived from L are solved in applications of point cloud processing.

Example 1.5.1 (Source: Tim Volodine - KU Leuven) *Laser scans of 3D objects produces point clouds in a 3D domain. Algorithms for extracting information from point clouds often solve linear systems using the Laplacian. For the computation of boundaries of point clouds or meshing point clouds large sparse linear systems need to be solved. Table 1.2 shows results for various point clouds comparing results from the iterative solver package GMM++ [31] and the direct solver UMFPACK [11]. Note the difference between the iterative and direct solvers and compare with the cost to set up the matrices.*

1.6 Overview of methods

The following overview is not complete. The details will be discussed in coming chapters.



a. b.

Figure 1.4: Dragon point cloud (a) and computed mesh (b)

1.6.1 Direct methods

For dense matrices, Gaussian elimination from LAPACK is mostly known. The complexity is $O(n^3)$. Many problems, however, have a hidden structure that can be solved quite efficiently. Typical examples are the Hankel and Toeplitz systems that can be solved with the Yule-Walker algorithm in $O(n^2)$ time.

For sparse matrices the situation is more complex. Developing efficient sparse direct methods is a complicated hobby. We will spend a chapter on this topic since they are used frequently in applications. The biggest advantage is that they always work. The disadvantage is the high memory consumption and often also the computation time.

1.6.2 Iterative methods

By iterative methods, we mean Krylov methods with a preconditioner. Krylov methods usually are significantly faster than simple iterative schemes. But the power of an iterative method usually lies in the preconditioner. The reason is that differential equations are often stiff implying that the spectrum of the matrices does not lend to an easy iterative solution. Many applications from industry cannot be solved iteratively. The reasons are often: highly ill-conditioneed indefinite problem, inappropriate mesh, or a bad preconditioner.

In the course, we will study iterative schemes (Conjugate Gradients, MINRES, Lanczos, Arnoldi, GMRES, CGS, BiCGStab, QMR, ...) and preconditioners. Finding a good preconditioner is an art: you have to find the colors and shapes that match well with the environment of the painting.

The idea of preconditioning is as follows. Suppose the system $Ax = b$ is to be solved and A is ill conditioned. With left preconditioning, the problem is transformed to another problem with the same solution:

$$P^{-1}Ax = P^{-1}b$$

but where $P^{-1}A$ is better conditioned.

Here is a short list with a first discussion.

1.6.3 Splitting methods

Splitting methods are methods where the matrix A is split into $A = M - N$ where M is the preconditioner. Examples are Jacobi, Gauss-Seidel, SOR, SSOR.

In general, these preconditioners are not efficient, but they are usually the first methods you try, since they are easy to implement.

1.6.4 Incomplete factorization

When a direct method (LU factorization/Gaussian elimination) is used for a sparse matrix, many non-zero elements are introduced. This is called fill-in. With incomplete factorization, fill-in is limited by following some rules. The factorization is not correct, but can be used as a preconditioner.

1.6.5 Sparse approximate inverse

These are preconditioners where the inverse of A is approximated by a sparse matrix [8] [1]. The application of the preconditioner is then simply a sparse matrix vector product, which is suitable for parallel computing. Naturally this type of method only works well when the matrix A^{-1} is nearly sparse. For many differential equations, this is not the case.

1.6.6 Multi-grid methods

For the solution of a boundary value problem, we use a discretization. The solution on a coarse grid (large h) can be used as an approximation of the solution of a fine grid (small h). Multi-grid is a class of methods that uses multiple grids to compute the solution.

1.6.7 Algebraic multi-grid methods

In many applications, it is hard to define multiple grids, e.g. in industrial finite element applications. The algebraic multi-grid method mimics the geometric multi-grid method by assembling matrices of smaller size as if they originate from a coarse mesh.

1.6.8 Structured matrices

In many applications, structured matrices arise, although this is not always clear at first sight. One example are finite element discretizations of regular volumes, including cubes, or pyramids. Such problems can usually be solved quickly by the application of block structured algorithms in $O(nb^2)$ time where b is the 'block' size. Most applications do have a complex geometry or boundary conditions, but the structured matrix can then sometimes be used as a preconditioner [24, 32].

1.6.9 Domain decomposition

The physical domain is split up into parts, and the problem is then solved on each part independently. Then, the problems are connected again using special boundary conditions or by transferring data from one domain to the other. This method is particularly interesting for parallel computing.

In algebraic domain decomposition, the matrix is split up into parts.

Chapter 2

Direct methods for sparse systems

2.1 Gaussian elimination

Gaussian elimination and LU factorization are the basic ideas of direct methods. An LU factorization is a decomposition of a general matrix A into $A = LU$ where U is an upper triangular matrix and L a unit lower triangular matrix[1]. The following algorithm shows how such a factorization can be performed. The matrices L and U are stored in A, so we use the symbol A for both L and U, see Figure 2.1.

Algorithm 2.1.1 (LU factorization)
1. for $k = 1, \ldots, n$ do:
   1.1. for $i = k+1, \ldots, n$ do:
      1.1.1. $A_{ik} = A_{ik}/A_{kk}$
      1.1.2. for $j = k+1, \ldots, n$ do:
         1.1.2.1. $A_{ij} = A_{ij} - A_{ik}A_{kj}$

The algorithm can be written in vector or matrix notation, which is easier to understand:

Algorithm 2.1.2 (LU factorization - matrix notation)
1. for $k = 1, \ldots, n$ do:
   1.1. for $i = k+1, \ldots, n$ do:
      1.1.1. $A_{i,k} = A_{i,k}/A_{kk}$
      1.1.2. $A_{i,k+1:n} = A_{i,k+1:n} - A_{i,k}A_{k,k+1:n}$

The computational cost of the algorithm is $\frac{4}{3}n^3 + O(n^2)$.

2.1.1 Transformation to upper triangular form

Since $U = L^{-1}A$, the LU factorization can also be viewed as a transformation of A to upper triangular form. The transformations are accumulated in L^{-1}. See Figure 2.2.

[1] Unit lower triangular means lower triangular with a unit diagonal

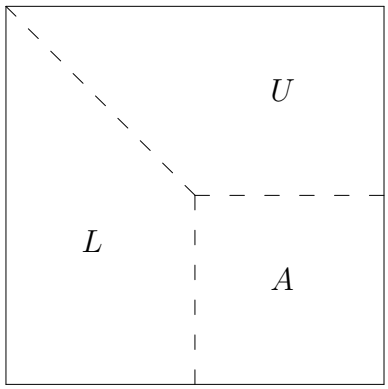


Figure 2.1: Layout of the matrix A in LU factorization at Step k. L and U are fully factorized.

$$
\begin{array}{cc}
U = A & L \\
\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ \times & 0 & 1 & 0 \\ \times & 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ \times & \times & 1 & 0 \\ \times & \times & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ \times & \times & 1 & 0 \\ \times & \times & \times & 1 \end{bmatrix}
\end{array}
$$

Figure 2.2: Transformation to upper triangular form

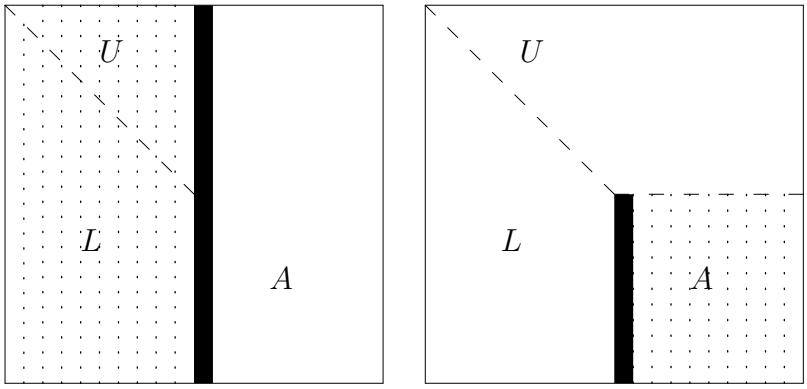


Figure 2.3: Active matrix in left- and right-looking algorithms

2.1.2 The six forms

There are six version of Algorithm 2.1.1 [19]. The version from Algorithm 2.1.1 is called the kij version, following the order of the indices. The other versions are thus kji, ikj, jki, ijk, jik.

The left-looking algorithm is the jki form: only data on the left of the jth column are accessed for the factorization ; it is a column oriented method [23].

Algorithm 2.1.3 (left-looking factorization)
1. for $j = 1, \ldots, n$ do:
   1.1. for $k = 1, \ldots, j-1$:
      1.1.1. $A_{k+1:n,j} = A_{k+1:n,j} - A_{k+1:n,k}A_{k,j}$
   1.2. $A_{j+1:n,j} = A_{j+1:n,j}/A_{jj}$

In the jth iteration column j of both L and U is computed.

The kji version is also called right-looking, since, once column k is computed, all columns of L and U on the right of column k are updated. Data on the left of column k are no longer accessed.

Algorithm 2.1.4 (right-looking factorization)
1. for $k = 1, \ldots, n$ do:
   1.1. $A_{k+1:n,k} = A_{k+1:n,k}/A_{kk}$
   1.2. $A_{k+1:n,k+1:n} = A_{k+1:n,k+1:n} - A_{k+1:n,k}A_{k,k+1:n}$

Frontal methods are related to right-looking techniques, but the update in Step 1.2 is not implemented in the same way as the original method [23]. We shall discuss frontal methods in §2.6.

Table 2.1 summarizes the six forms. The notion upward-looking is not used in the literature, only right-looking and left-looking for column oriented matrices.

type	orientation	updating
kji	column	right-looking
jki	column	left-looking
jik	column	left-looking
kij	row	right-looking
ikj	row	
ijk	row	

Table 2.1: The six forms of LU factorization

2.2 Symmetric matrices

Algorithm 2.1.1 also exists for symmetric matrices. In this case, only half of the work is required. Since A is symmetric, the factorization is usually written in the form $A = LDL^T$ where D is a diagonal matrix and L is unit upper triangular. (We still have $A = LU$ with $U = DL^T$.) For example, Algorithm 2.1.1 can be rewritten as

Algorithm 2.2.1 (LDL^T factorization)
1. for $k = 1, \ldots, n$ do:
   1.1. for $i = k + 1, \ldots, n$ do:
      1.1.1. $L_{ik} = A_{ik}/D_{kk}$
      1.1.2. for $j = k + 1, \ldots, i - 1$ do:
         1.1.2.1. $A_{ij} = A_{ij} - L_{ik}D_{kk}L_{jk}$
      1.1.3. $D_{ii} = D_{ii} - L_{ik}D_{kk}L_{ik}$

When A is Hermitian positive definite, D has positive and real elements. The Cholesky factorization can be used in this case, i.e. $A = LL^*$ where the diagonal elements of L contain the square roots of D.

2.3 Pivoting for numerical stability

Algorithm 2.1.1 is potentially numerically unstable. Only for specific classes of matrices (e.g. Hermitian positive definite), the algorithm works without numerical difficulties. Small pivots, i.e. small $|U_{ii}|$ or $|D_{ii}|$, can lead to a large backward error in the LU factorization. Partial pivoting, i.e. interchanging rows of A, improves the numerical stability enormously [19]. In the kij scheme, the pivots are chosen from the column that is to be factorized.

For symmetric matrices, pivoting is more complicated: interchanging rows also implies interchanging columns in the same way, otherwise symmetry is lost. As a result, pivots can only be chosen from the main diagonal. This has shown to pose numerical problems on linear systems of the form

$$\begin{bmatrix} A & B \\ B^T & 0 \end{bmatrix} x = f \ .$$

$$
\begin{array}{cc}
U = A & L \\
\begin{bmatrix} \times & \times & 0 & 0 \\ \times & \times & \times & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ 0 & \times & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ 0 & \times & 1 & 0 \\ 0 & 0 & \times & 1 \end{bmatrix}
\end{array}
$$

Figure 2.4: Transformation of a banded matrix to upper triangular form

Therefore, a pivoting technique is proposed that allows for 2×2 pivot blocks on the main diagonal [9] [10].

2.4 Banded matrices

Band matrices are matrices consisting of non-zero consecutive diagonals. The lower bandwidth n_l is the number of diagonals in the lower triangular part. The upper bandwidth n_u is the number of diagonals in the upper triangular part. When the lower and upper bandwidth are zero, we have a diagonal matrix.

Banded matrices play an important role in dense linear algebra: their structure is preserved by LU factorization. Tridiagonal matrices are a special case. Their structure is also preserved by QR factorization. Furthermore they play an important role in eigenvalue computations of Hermitian matrices and, to some extent, for non-Hermitian eigenvalue problems.

In the context of linear systems, let us take an other look at Algorithm 2.1.1. Consider the example in Figure 2.4. The complexity is nn_ln_u, where it is n^3 for a full matrix.

The bandwidth of the factorization is preserved. Now, we do the same exercise with pivoting, taking into account the worst case scenario. See Figure 2.5. The upper triangular part becomes larger. Generally in the presence of pivoting, the lower triangular part

$$
\begin{array}{cc}
U = A & L \\
\begin{bmatrix} \times & \times & \times & \diamond & 0 \\ \times & \times & \times & \times & 0 \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} \times & \times & \times & \diamond & 0 \\ 0 & \times & \times & \times & \diamond \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} \times & \times & \times & \diamond & 0 \\ 0 & \times & \times & \times & \diamond \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 & 0 \\ 0 & \times & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} \times & \times & \times & \diamond & 0 \\ 0 & \times & \times & \times & \diamond \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & \times \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 & 0 \\ 0 & \times & 1 & 0 & 0 \\ 0 & 0 & \times & 1 & 0 \\ 0 & 0 & 0 & \times & 1 \end{bmatrix}
\end{array}
$$

Figure 2.5: Transformation of a banded matrix with $n_l = 1$ and $n_u = 2$ to upper triangular form

loses half of its bandwidth n_l which the upper triangular part obtains. This means that the bandwidth of the upper triangular part now equals $n_l + n_u$.

Band matrix methods can be used for solving discretized differential equations. Recall the examples from Chapter 1. Banded matrices are useful for regular grids, but for realistic problems like the mushroom mesh from Figure 1.3, the band still contains many zero elements. In the following sections, we present methods that are more adequate for solving sparse linear systems than band methods.

2.5 Variable band or skyline matrices

Figure 2.6 shows the profile of a variable band or skyline matrix. The shown profile is symmetric, which is often the case in discretizations of partial differential equations. Without pivoting, the factorization remains inside the profile. The traditional variable band matrix factorization is left-looking. The order of operations is slightly different from

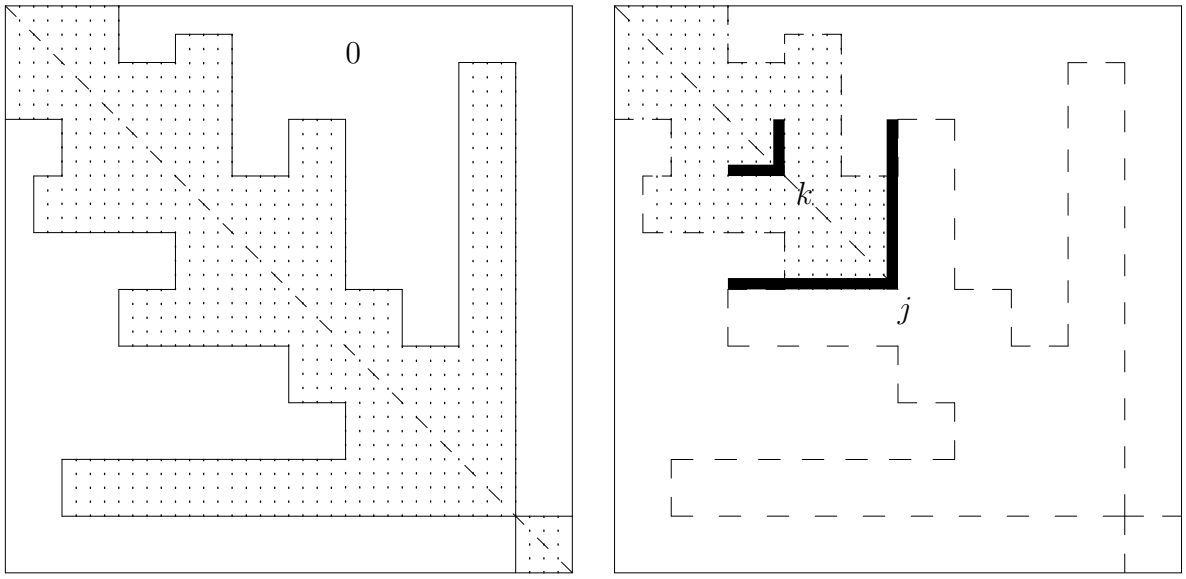


Figure 2.6: Layout of a variable band matrix A in LU factorization at Step j. Computation of A_{kj} and A_{jk}.

what we discussed before. The U part is stored column by column, while the L part is stored row by row. This format allows us to easily store L and U by a one-dimensional array.

Algorithm 2.5.1 (Left looking skyline factorization)
1. for $j = 1, \ldots, n$ do:
   1.1. for $k = u_j, \ldots, j-1$ do:
      1.1.1 $A_{k,j} = A_{k,j} - A_{k,:}A_{:,j}$
      1.1.2 $A_{j,k} = A_{j,k} - A_{j,:}A_{:,k}$
      1.1.3 $A_{j,k} = A_{j,k}/A_{kk}$
   1.2. $A_{j,j} = A_{j,j} - A_{j,:}A_{:,j}$

2.6 Frontal methods

Frontal methods were first used for the solution of finite element problems. Consider the regular mesh shown in Figure 2.7. The bullets are the unknowns of the problem. The elements are numbered from a to i, the unknowns from 1 to 16. For each unknown, the 4×4 element matrices are computed and assembled in the matrix A. The matrix is the sum of the element matrices:

$$A = \sum_{e=1}^{n_{\text{elt}}} A_e \ .$$

Unknown 2 lies on elements a and b. So, assembling the second row and column of A implies we have to add elements a and b together. The assembled matrix looks as follows:

$$\left[\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c}
a & a & & & a & a & & & & & & & & & & \\ \hline
a & a,b & b & & a & a,b & b & & & & & & & & & \\ \hline
 & b & b,c & c & & b & b,c & c & & & & & & & & \\ \hline
 & & c & c & & & c & c & & & & & & & & \\ \hline
a & a & & & a,d & a,d & & & d & d & & & & & & \\ \hline
a & a,b & b & & a,d & a,b,d,e & b,e & & d & d,e & e & & & & & \\ \hline
 & b & b,c & c & & b,e & b,c,e,f & c,f & & e & e,f & f & & & & \\ \hline
 & & c & c & & & c,f & c,f & & & f & f & & & & \\ \hline
 & & & & d & d & & & d,g & d,g & & & g & g & & \\ \hline
 & & & & d & d,e & e & & d,g & d,e,g,h & e,h & & g & g,h & h & \\ \hline
 & & & & & e & e,f & f & & e,h & e,f,h,i & f,i & & h & h,i & i \\ \hline
 & & & & & & f & f & & & f,i & f,i & & & i & i \\ \hline
 & & & & & & & & g & g & & & g & g & & \\ \hline
 & & & & & & & & g & g,h & h & & g & g,h & h & \\ \hline
 & & & & & & & & & h & h,i & i & & h & h,i & i \\ \hline
 & & & & & & & & & & i & i & & & i & i
\end{array}\right]$$

In a frontal method, the matrix A is never assembled explicitly. The computations are performed directly using element matrices. We illustrate this for an example.

The frontal method uses a dense frontal matrix in which all computations take place. We start with element a. This is a dense matrix. We store it in the frontal matrix:

$$F = \left[\begin{array}{c|ccc}
A_{11} & A_{12} & A_{15} & A_{16} \\ \hline
A_{21} & A_{22}^{(a)} & A_{25}^{(a)} & A_{26}^{(a)} \\
A_{51} & A_{52}^{(a)} & A_{55}^{(a)} & A_{56}^{(a)} \\
A_{61} & A_{62}^{(a)} & A_{65}^{(a)} & A_{66}^{(a)}
\end{array}\right]$$

Unknown 1 is fully assembled, so we can factorize row 1 and column 1. We can update the remaining rows and columns of F as for a right-looking algorithm. Since 1 is only connected to unknowns 5, 6 and 2, only the rows and columns corresponding to these unknowns are affected. These are the elements in the frontal matrix, i.e. the frontal matrix is the submatrix of A that consists of the intersections of rows and columns 1, 2, 5 and 6. Next, we load element b. The frontal matrix becomes

$$F = \left[\begin{array}{c|cccc}
A_{22} & A_{23} & 0 & A_{26} & A_{27} \\ \hline
A_{32} & A_{33}^{(b)} & 0 & A_{36}^{(b)} & A_{37}^{(b)} \\
0 & 0 & A_{55}^{(a)} & A_{55}^{(a)} & 0 \\
A_{62} & A_{63}^{(b)} & A_{65}^{(a)} & A_{66}^{(b)} & A_{67}^{(b)} \\
A_{72} & A_{73}^{(b)} & 0 & A_{76}^{(b)} & A_{77}^{(b)}
\end{array}\right]$$

As a result, unknown 2 is fully assembled and we can factorize it. Again, we update F as for a right-looking algorithm. The changes are stored in the frontal matrix. We continue

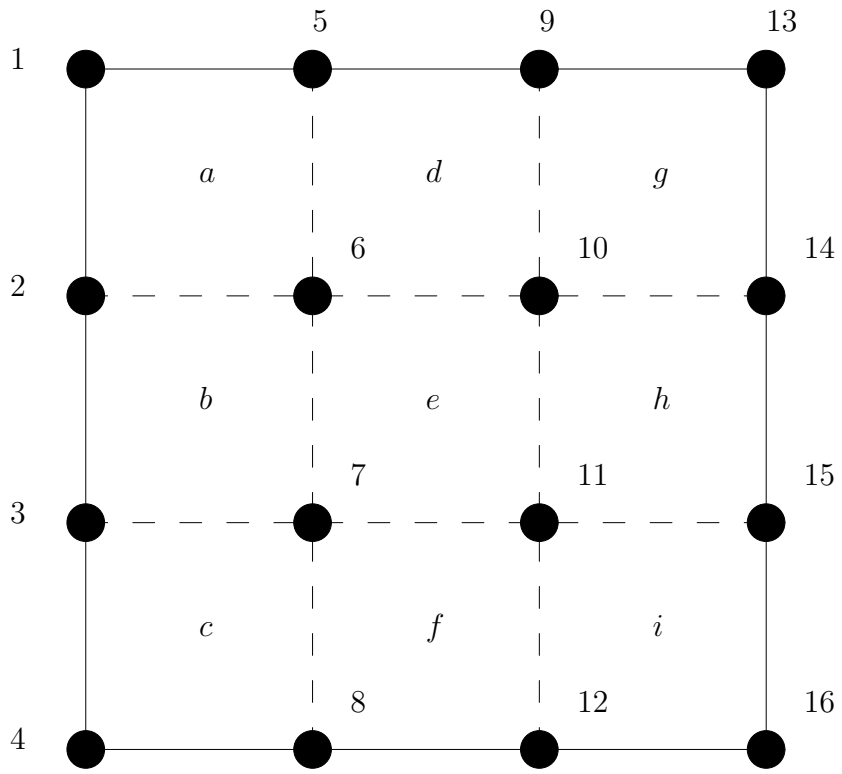


Figure 2.7: Frontal method for a regular mesh

this way. The name frontal matrix comes from the fact that it is as if a front (or a wave) travels through the mesh. The parts of the matrix that are factorized need not need to remain in the front. They are stored elsewhere.

Figure 2.8 shows the front for a tube. The front contains the elements of a cross section of the tube. As a conclusion: the thicker the tube, the more elements in the front and so, the more expensive the frontal method. This result is general and also holds for methods other than frontal methods. The size of the frontal matrix is related to the half bandwidth of a banded matrix.

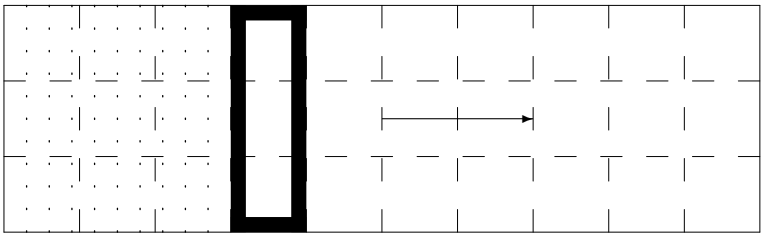


Figure 2.8: Front for a tube. The dotted part is factorized

2.7 Sparse matrices

A sparse matrix is a matrix with many non-zeroes. If a 1000×1000 matrix has two zero elements, we still treat this matrix as dense. Strictly speaking, banded and skyline matrices are sparse too, but we do not classify them as sparse formats. The banded and skyline formats are useful for sparse direct methods, but not necessarily to represent a sparse matrix.

The first frontal methods have been developed decades ago. Frontal methods are useful for finite element problems, but for other applications, they are hard to use. For this reason, sparse matrix formats and sparse direct methods have been introduced to solve more general problems.

2.7.1 Storage formats

There are many formats for sparse matrices. The most commonly used are the following two. We illustrate the storage formats for the matrix

$$\begin{bmatrix} -3.5 & 1.4 & & 8 & -2 \\ & -3 & & & \\ & & 3 & 2 & \\ & & & 1 & -5 \\ & & & & -5 \end{bmatrix}. \tag{2.1}$$

Coordinate format

By the number of non-zeroes we usually mean the number of values that are stored by the sparse matrix format. The coordinate format (COO) stores three arrays of the same length, namely, the number of non-zeros N_{nz}.

- `rows`: the row number of the non-zero
- `columns`: the column number of the non-zero
- `values`: the value of the non-zero

For (2.1), $N_{\text{nz}} = 10$. A possible COO representation is

`rows`:	1	2	3	3	1	1	4	1	4	5
`columns`:	1	2	3	4	2	4	4	5	5	5
`values`:	-3.5	-3	3	2	1.4	8	1	-2	-5	-5

The non-zeroes can be stored in any order: this is one of the great advantages of the format.

This format is used by the Matrix Market file format, which is an ASCII representation of a sparse matrix in a file. It can be used to read and write data for Matlab and other programs. Figure 2.9 is the Matrix Market file for the matrix (2.1).

```
%%MatrixMarket matrix coordinate real general
5 5 10
1 1 −3.5
1 2 1.4
2 2 −3
3 3 3
3 4 2
4 4 1
1 5 −2
1 4 8
4 5 −5
5 5 −5
```

Figure 2.9: Matrix Market file for the sparse matrix in (2.1)

Compressed sparse column format

Sorting the coordinate format by column we obtain the following data:

`rows r`:	1	1	2	3	3	4	1	1	4	5
`columns c`:	1	2	2	3	4	4	4	5	5	5
`values v`:	-3.5	1.4	-3	3	2	1	8	-2	-5	-5

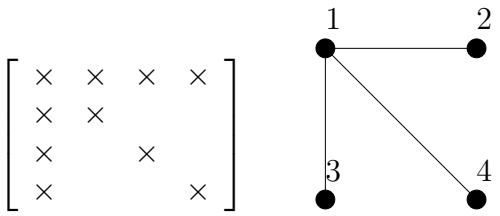
The columns array is now replaced by an array that points to the beginning of each column in the array of row numbers and values:

`columns c`:	1	2	4	5	8	11

The columns array consists of $n+1$ elements and is thus a compression of the original array. The jth element c_j tells us where the jth column starts, while $c_{j+1}-1$ is the last element in column j. If column j has no elements, $c_j = c_{j+1}$. The compressed format is sorted which makes it more difficult to construct, but is is a more efficient format for several algorithms.

2.7.2 Sparse matrices and graphs

A sparse matrix is often represented as a graph. The unknowns are vertices and the non-zeroes correspond to edges in the graph. A symmetric sparse matrix can be represented by an undirected graph. In the remainder of this chapter, we assume that matrices have symmetric non-zero structure.



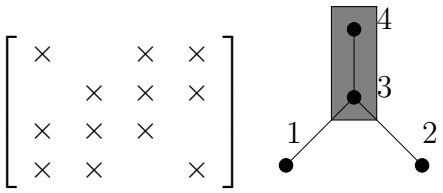


Figure 2.10: Elimination three and clique amalgamation for a 4×4 problem

2.7.3 The clique concept

A clique is a subgraph of fully interconnected vertices. In the previous example, $\{1, 2\}$ could be viewed as a clique, but also $\{1, 3\}$, etc.

2.7.4 The elimination tree

The order of elimination and the dependencies of the unknowns in the Gaussian elimination process are often expressed by the elimination tree. Analyze the small problem in Figure 2.10.

Joseph Liu [26] wrote a nice article about elimination trees and their use in sparse factorization.

2.7.5 Sparse arithmetic

Sparse arithmetic is slow because indirect addressing makes efficient use of the cache very hard: it is indeed not easy to predict in which order data come into the cache. Also, indirect addressing increases the amount of integer arithmetic, and so, slows down the code.

Sparse arithmetic is rarely used in practice. Modern codes for sparse linear systems all use dense matrix arithmetic as much as possible.

2.8 Multi-frontal methods

We illustrate the multi-frontal idea on the small example from Figure 2.10. We make the frontal matrix for node 1:

$$F_1 = \begin{bmatrix} A_{11} & A_{13} & A_{14} \\ A_{31} & & \\ A_{41} & & \end{bmatrix}$$

perform the factorization and update the lower right 2×2 corner as in the frontal method. We do the same for node 2, which can be factorized independently of node 1 as the

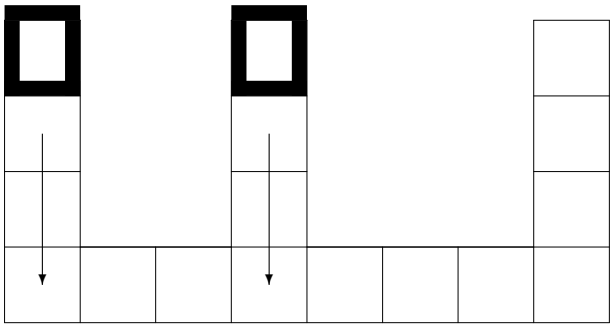


Figure 2.11: Illustration of multi-frontal method

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & & \\ \times & & \times & \\ \times & & & \times \end{bmatrix} \quad \begin{bmatrix} \times & & & \times \\ & \times & & \times \\ & & \times & \times \\ \times & \times & \times & \times \end{bmatrix}$$

Figure 2.12: Two orderings for the arrowhead profile

elimination tree suggests:

$$F_2 = \begin{bmatrix} A_{22} & A_{23} & A_{24} \\ A_{32} & & \\ A_{42} & & \end{bmatrix}$$

As a result, there are two frontal matrices, one for each branch in the elimination tree. $F_1(2:3, 2:3)$ and $F_2(2:3, 2:3)$ are then two frontal matrices which are added together to factorize nodes 3 and 4. The elimination of nodes 1 and 2, makes nodes 3 and 4 form a clique subgraph. Cliques can be amalgamated in the elimination tree.

Frontal methods do not always perform quite well on practical problems. Consider for example, the mesh in Figure 2.11. A frontal method might generate, at the cross point of the three 'legs', a front larger than the 'width' of the mesh.

2.9 The effect of renumbering the unknowns

The storage cost of the variable band method depends on the surface of the skyline profile. The computational cost also depends on the mean bandwidth. It is clear that a reduction of the bandwidth reduces storage and computational costs. Renumbering the unknowns is extremely important. Consider the matrices in Figure 2.12. Swapping the first and last rows and the first and last columns in the left hand side matrix produces the right-hand side one. For the left hand side matrix, the factorization fills the matrix completely. The right hand side matrix has a small profile, no fill-in is produced. Figure 2.13 shows the impact of renumbering on an application from graphs in the context of point clouds.

The ordering techniques for sparse matrices use graph algorithms. Therefore we use the graph connection, in the following section. The goal of the graph algorithms is to

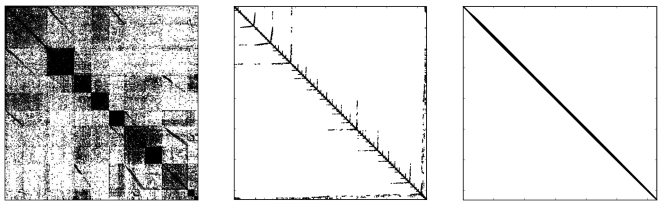


Figure 2.13: Sparse matrix from a point cloud application, renumbered following different schemes

either reduce the bandwidth of the sparse matrix, which is important for skyline and frontal methods, or to reduce the fill-in in the sparse factorization, which is important for the multi-frontal approach.

2.9.1 Reverse Cuthill McKee ordering

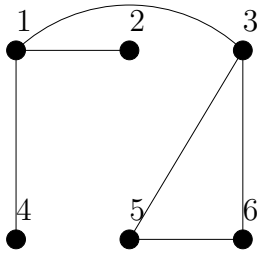
The most popular reordering method for skyline and frontal methods is undoubtfully the Reverse Cuthill McKee ordering. The Reverse Cuthill McKee (RCM) ordering is the Cuthill McKee (CM) ordering in reverse order. Recall the connection between sparse matrices and graphs. The CM ordering for a sparse matrix works as follows:

Algorithm 2.9.1 (Cuthill-McKee reordering)
1. Choose the initial set $\mathcal{I} = \{i_0\}$
2 While $\mathcal{I}$ does not yet contain all nodes, do:
  2.1 Add all vertices with edges to the vertices in $\mathcal{I}$

The nodes that are added at Step 2.1 are called level sets. The order of the nodes within a level-set could be determined by a local greedy algorithm, but this is out of the scope of the course.

Example 2.9.1 *Consider the following matrix with corresponding graph*

$$A = \begin{bmatrix} \times & \times & \times & \times & & \\ \times & \times & & & & \\ \times & & \times & & \times & \times \\ \times & & & \times & & \\ & & \times & & \times & \times \\ & & \times & & \times & \times \end{bmatrix}$$



Start CM from node 1, i.e. $\mathcal{I}_1 = \{1\}$*, then we have the following level-sets* $\mathcal{I}_2 = \{2, 3, 4\}$*,* $\mathcal{I}_3 = \{5, 6\}$*. The CM ordering becomes* $\mathcal{P}_{\text{CM}} = \{1, 2, 3, 4, 5, 6\}$ *and the RCM ordering* $\mathcal{P}_{\text{RCM}} = \{6, 5, 4, 3, 2, 1\}$*. The matrices for these renumberings are*

$$A = \begin{bmatrix} \times & \times & \times & \times & & \\ \times & \times & & & & \\ \times & & \times & & \times & \times \\ \times & & & \times & & \\ & & \times & & \times & \times \\ & & \times & & \times & \times \end{bmatrix} \qquad A = \begin{bmatrix} \times & \times & & \times & & \\ \times & \times & & \times & & \\ & & \times & & & \times \\ \times & \times & & \times & & \times \\ & & & & \times & \times \\ & & \times & \times & \times & \times \end{bmatrix}$$

Example 2.9.2 *Recall the mesh from Figure 2.8. The CM ordering creates the ordering of the traveling front.*

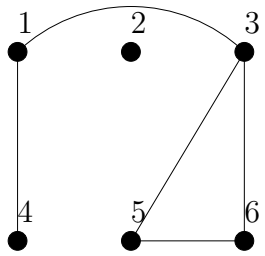
2.9.2 Minimum degree ordering

The degree of a vertex of a mesh is its number of edges. The minimum degree ordering starts with the vertex with smallest degree, since this is, intuitively, the vertex that potentially creates less fill-in. The vertex is eliminated from the graph, and the process is repeated.

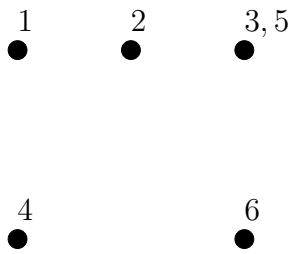
2.9.3 Approximate minimum degree ordering

The minimum degree ordering is rather complicated, since the elimination of a vertex is similar to the elimination of an unknown from a linear system. It creates additional edges between the vertices that are connected to the removed vertex. The approximate minimum degree ordering is much more efficient. When a vertex is eliminated, the connected vertices are amalgamated into a clique or super-node. The degrees are recomputed and the process restarted.

Example 2.9.3 *We recall the problem from Example 2.9.1. The vertex with the smallest degree is 2 or 4. Let us take 2. We put all vertices connected to 2 into a super-node. Since this is only vertex 1, the super-node only contains vertex 1. The elimination of node 2, creates the following graph:*



Now, we select node 4 as next node with smallest degree. Then the node with smallest degree is 1. The remaining nodes are 3, 5, and 6. If we continue with node 6, its elimination creates a clique or super-node consisting of nodes 3 *and* 5.

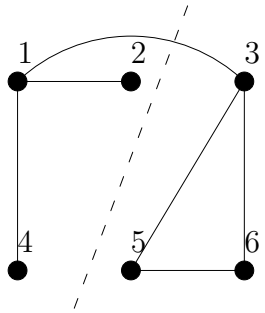


We obtain the ordering $\{2, 4, 1, 6, 3, 5\}$ *The reordered matrix is as follows:*

$$A = \begin{bmatrix} \times & & \times & & & \\ & \times & \times & & & \\ \times & \times & \times & & \times & \\ & & & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \end{bmatrix}$$

2.9.4 Nested bisection

The idea of nested bisection is to cut a graph in two parts of equal number of vertices where the edge cut (the number of cut edges) is minimal. For our example, the optimal edge cut is one, namely, the edge between vertices 1 and 3 is cut:



With nested bisection, this process is repeated for the subgraphs. The method is mainly used for parallel processing to optimize the communication overhead. We do not discuss this in this chapter.

2.9.5 Partial pivoting

The reordering schemes do not take numerical pivoting into account. Sparse LU factorization software usually performs a pre-ordering using one of the techniques discussed in this section. The pre-ordering only uses the non-zero pattern of the matrix.

During the numerical factorization, numerical pivoting is used to improve numerical stability. In order to reduce the fill-in due to pivoting (recall banded matrices), the pivots are usually chosen close to the main diagonal.

Another idea is to modify the values on the main diagonal so that the pivots do not introduce large rounding errors. This changes the matrix, but the solution of the original system can be found by iterative refinement.

We do not discuss this any further.

2.9.6 Reordering software

- PORD
- Metis
- SCOTCH

Chapter 3

Krylov methods for linear systems

In this chapter, we review 'classical' Krylov methods for solving linear systems. Recall from Demmel's book [12], the notion Krylov space, Lanczos and Arnoldi process.

3.1 Krylov spaces and linear solvers

The idea is to solve the linear system

$$Ax = b \tag{3.1}$$

where $A \in \mathbb{C}^{n\times n}$ with n large.

In this chapter, we discuss a few Krylov methods. The following methods are most important in practice:

- Symmetric positive definite matrices: Conjugate Gradients (CG)
- Symmetric matrices: MINRES
- Non-symmetric matrices: GMRES and BiCGStab.

The Krylov space $\mathcal{K}_k$ is defined by the space spanned by the Krylov sequence

$$\{b, Ab, A^2, \ldots, A^{k-1}b\}$$

It is assumed that the dimension of $\mathcal{K}_k = k$.

Arnoldi's algorithm produces an orthogonal basis $Q_k = [q_1, \ldots, q_k]$, where the relation with A is given by a recurrence relation. This relation can be written in the following ways:

$$AQ_k = Q_{k+1}\underline{H}_k \tag{3.2}$$
$$AQ_k = Q_k H_k + \beta_k q_{k+1} e_k^* \tag{3.3}$$

where

$$\underline{H}_k = \begin{bmatrix} H_k \\ \beta_k e_k^* \end{bmatrix} = \left[\begin{array}{cccc} h_{1,1} & \cdots & \cdots & h_{1k} \\ \beta_1 & h_{2,2} & & \vdots \\ & \ddots & \ddots & \vdots \\ & & \beta_{k-1} & h_{k,k} \\ \hline & & & \beta_k \end{array}\right]$$

The matrix H_k is upper Hessenberg. It is called the Hessenberg matrix of the Arnoldi algorithm. From $Q_{k+1}^* Q_{k+1} = I$ and (3.2), we derive that $\underline{H}_k = Q_{k+1}^* A Q_k$.

If $\beta_k = 0$, then

$$AQ_k = Q_k H_k$$

which implies that eigenvalues of H_k are eigenvalues of A. The Krylov space $\mathcal{K}_k$ is called an invariant subspace of A, i.e.

$$A\mathcal{K}_k \subset \mathcal{K}_k \ .$$

When A is symmetric (Hermitian), H_k is also symmetric, so it is tridiagonal. The Arnoldi method becomes the Lanczos method.

The solution of (3.1) by a Krylov method can be done as follows. Assume that the initial vector $x_0 = 0$, and let x_k (the k th iterate) be chosen in the Krylov subspace $\mathcal{K}_k$. In mathematical notation, this is

$$x_k = Q_k z = \sum_{j=1}^{k} q_j z_j \quad \text{with} \quad z = [z_1, \ldots, z_k]^T \ .$$

The residual at iteration k is then given by

$$\begin{aligned} r_k &= b - Ax_k = b - AQ_k z \qquad (3.4) \\ &= \|b\|_2 q_1 - Q_k H_k z - q_{k+1}\beta_k e_k^* z \\ &= Q_{k+1}(\|b\|_2 e_1 - \underline{H}_k z) \qquad (3.5) \end{aligned}$$

There are several ways for computing x_k:

1. $\|x - x_k\|_2$ minimal: perhaps ideal but this is not possible in practice
2. $\|b - r_k\|_2$ minimal: residual minimizing methods (MINRES, GMRES)
3. $r_k \perp \mathcal{K}_k$: residual orthogonal to the Krylov space. This is the Galerkin projection method. From (3.5) we have that

$$\begin{aligned} Q_k^T r_k &= 0 \\ &= \|b\|_2 Q_k^T q_1 - Q_k^T Q_k H_k z - Q_k^T q_{k+1}\beta_k e_k^* z \\ &= \|b\|_2 e_1 - H_k z \end{aligned}$$

So, choosing $z = \|b\|_2 H_k^{-1} e_1$, does the job. When k is not too large, the computation of z is cheap.

4. When A is symmetric positive definite (SPD), the Galerkin projection also corresponds to minimizing

$$\|r_k\|_{A^{-1}}^2 = r_k^T A^{-1} r_k = (x - x_k)^T A (x - x_k) \ .$$

This is the case for the conjugate gradient method.

The various iterative methods try and optimize the memory consumption. For example, the Conjugate Gradient and MINRES method do not require the storage of Q_k and T_k. They work with a small number of vectors and the last column of T_k only. See §6.6.3 in Demmels' book [12].

3.2 Other methods for symmetric systems

The conjugate gradient method can be used for symmetric matrices that are not positive definite. However, a division by zero is possible. The method solves at each iteration the linear system

$$T_k z = e_1 \|b\|_2$$

If A is SPD, so is $T_k = Q_k^T A Q_k$. If A has positive and negative eigenvalues with λ_1 the smallest and λ_N the largest, then T_k also has eigenvalues between λ_1 and λ_N. If λ_1 and λ_N have different sign, T_k may be singular or close to singular. The SYMMLQ method [30] performs the factorization $LQ = T_k$ with L lower triangular and Q unitary. It is more stable than the Cholesky or LDL^T factorization that is used by Conjugate Gradients.

The MINRES method minimizes r_k instead of making r_k orthogonal to $\mathcal{K}_k$. In addition, MINRES can be used for any non-singular symmetric linear system.

The idea is to have the kth iterate $x_k = Q_k z$ so that $\|r_k\|_2$ is minimal. from (3.5), we have

$$\begin{aligned} \|r_k\|_2 &= \|Q_{k+1}(\|b\|_2 e_1 - \underline{T}_k z)\|_2 \\ &= \|\|b\|_2 e_1 - \underline{T}_k z\|_2 \end{aligned}$$

So, z is the solution of the $k+1 \times k$ tridiagonal least squares problem

$$\min \|\|b\|_2 e_1 - \underline{T}_k z\|_2 \tag{3.6}$$

This problem is solved by using a QR factorization of $\underline{T}_k$. Similar to the CG method, this can be performed column by column, and there is no need to store all Q_k's. The solution x_k is updated along the way.

Note that MINRES never breaks down, since the solution of (3.6) always exists for any k. The reason is that the tridiagonal matrix $\underline{T}_k$ has non-zero off-diagonal elements, so that it is of full rank. To see this, consider the $k \times k$ submatrix of the last k rows. This is a triangular matrix with nonzero diagonal elements and so it is non-singular. Hence, T_k has rank at least k.

3.3 Nonsymmetric matrices

Any system (3.1) can be solved by the conjugate gradient method by first transforming it into

$$A^T Ax = A^T b \qquad \text{or} \qquad AA^T y = b, x = A^T y$$

Least squares problems of the form

$$\min \|b - Ax\|_2$$

can also be solved in a similar way, provided A has full rank. The condition is that A is not too badly conditioned, since $\text{cond}(A^T A) = \text{cond}(A)^2$.

Usually, methods for nonsymmetric matrices are preferred.

There are two groups of Krylov methods:

- Methods based on the Arnoldi algorithm, e.g. GMRES, FOM. These methods are robust, have monotonic convergence, but require a large amount of memory, since all vectors Q_k need to be stored.
- Methods based on the nonsymmetric Lanczos algorithm, e.g. CGS, QMR, BiCGStab. These methods do not have monotonic convergence, but do not require a large amount of memory.

3.4 Arnoldi based methods

The theory for symmetric systems still remains, but now we have to work with a Hessenberg matrix rather than a tridiagonal. The LU factorization of H_k produces a full upper triangular U and a bidiagonal L, so, at iteration k,

$$x_k = Q_k H_k^{-1} e_1 \|b\|_2 = Q_k U^{-1} L^{-1} e_1 \|b\|_2$$

requires all columns in Q_k. It is no longer possible to write $x_k = x_{k-1}+$ a small number of terms. This makes the solution of a nonsymmetric system by Arnoldi's method less attractive than symmetric systems. The method, we just described is called FOM (Full Orthogonalization Method) or Arnoldi. A practical implementation performs the factorization of H_k iteration by iteration. In each iteration, we add a vector (q_{k+1}) to the basis: the memory grows with the iteration count.

As for the SYMMLQ method, H_k can be singular, even if A is not singular, which leads to breakdown in the method.

The GMRES method (Generalized Minimal RESidual method) minimizes $\|r_k\|_2$. As for MINRES, this method never breaks down. On each iteration, the GMRES method reduces the residual, i.e. convergence is monotonic. The theory is similar as for MINRES.

The convergence of GMRES is also explained in terms of polynomials and eigenvalues as for Conjugate Gradients. Let $A = U\Lambda U^{-1}$ be the eigendecomposition. Then

$$\frac{\|r_k\|_2}{\|b\|_2} \leq \text{cond}(U) \min_{p_k, p_k(0)=1} \max_{i=1,\ldots,n} |p_k(\lambda_i)| \ .$$

If cond(U) is not too large, the convergence theory is similar to conjugate gradients. Eigenvalues can be complex, and this makes it a little more complicated to understand. The following rules can help to understand the convergence of GMRES:

- if A is well conditioned, the method converges quickly
- if the eigenvalues of A lie in a cluster away from zero, the method converges quickly.
- If only a few eigenvalues lie near zero and/or a few far away from zero, convergence is still fast.

The major difficulty with GMRES is that $k+1$ basis vectors need to be stored. This is no longer feasible when k becomes large. Therefore, restarted GMRES is proposed: if $\|r_k\|_2$ is large, and the storage of additional basis vectors is no longer an option, the method is restarted on the residual equation

$$Ay = b - Ax_k$$

The solution of $Ax = b$ is then obtained as $y_k + x_k$.

3.5 Lanczos based methods

There is a variant of the Lanczos method for nonsymmetric matrices. It is also called the Lanczos method. As for the symmetric case, a three term recurrence relation is built:

$$Av_j = \beta_{j-1}v_{j-1} + \alpha_j v_j + \gamma_j v_{j+1}$$

Putting together the equations for $j = 1, \ldots, k$, we have in matrix form

$$AV_k = V_k T_k + v_{k+1}\gamma_k e_k^T$$

with

$$T_k = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \gamma_1 & \alpha_2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \alpha k-1 & \beta_{k-1} \\ & & & \gamma_{k-1} & \alpha_k \end{bmatrix}$$

Note that T_k can be non-symmetric. The choice of the elements in T_k is not unique. Therefore, a second equation is added to fix the constants

$$A^T W_k = W_k T_k^T + w_{k+1}\beta_k e_k^T$$

such that $W_{k+1}^T V_{k+1} = I$. This is called bi-orthogonality. We thus have two Krylov subspaces:

- $\mathcal{K}_k(A, b)$ the same Krylov space as we had before in Arnoldi's method, but with an oblique basis $v_1, \ldots, v_k$.
- $\mathcal{K}_k(A^T, w_1)$ a Krylov space for A^T, where the basis vectors W_{k+1} are biorthogonal to V_{k+1}.

From the biorthogonality we derive that

$$T_k = W_k^T A V_k$$

We now can again develop similar solvers as for the symmetric case or the Arnoldi type methods.

For example, the 'bi-Lanczos' method makes r_k biorthogonal to V_k, i.e. we choose $x_k = V_k z$ so that $W_k^T r_k = 0$:

$$\begin{aligned} W_k^T r_k &= 0 \\ &= W_k^T b - W_k^T A V_k z \\ &= e_1 (w_1^T b) - T_k z \end{aligned}$$

So, z is computed from a $k \times k$ nonsymmetric tridiagonal linear system. Similar to the symmetric case, the solution follows from the LU factorization of T_k. Since L and U are bidiagonal matrices, the storage of the last v's and w's suffices for computing x_k. We can reuse x_{k-1} to compute x_k, where the storage of all columns of V_k and W_k is not required.

The fact that the basis is not orthogonal has important consequences. If $\text{cond}(V_k)$ is large, numerical difficulties can arise. It may even happen that $v_{k+1}^T w_{k+1} = 0$. This situation is called breakdown. Cures to breakdown have been proposed in the literature (so-called look-ahead Lanczos).

Also, $\|r_k\|_2$ does not at all converge monotonically. The behaviour is often called erratic, and hard to predict. In the literature of Krylov solvers, this errative behaviour is studied and explained.

Another disadvantage is that the transpose of A is required, which is not always feasible.

Developing a Lanczos method that minimizes $\|r_k\|_2$ is not so easy since $V_k^T V_k \neq I$. The QMR method (Quasi minimum residual method) [18] approximates this by assuming that $V_k^T V_k = I$, i.e. similar to the MINRES method, a small least squares problem is solved in iteration k:

$$\min \|e_1 (w_1^T b) - \underline{T}_k z\|_2$$

The solution $x_k = V_k z$ does not minimize $\|r_k\|_2$, but if V_k is not too ill-conditioned, it is quasi minimizing $\|r_k\|_2$.

The BiCGStab method [40] is a Lanczos method that does not require the transpose. In addition, it stabilizes the unpredictable behaviour of the convergence by combining one step of Lanczos with one step of GMRES. The details of this method are out of the scope of this course.

Chapter 4

Incomplete factorization preconditioning

4.1 Introduction

Krylov methods for linear systems only work well when there is a polynomial p_k so that

- $p_k(0) = 1$
- $|p_k(\lambda_j)| \ll 1$ where λ_j is an eigenvalue of A

This implies that there should be few eigenvalues near zero and/or far away from zero. Most applications do not have matrices that satisfy this condition, e.g. differential equations are often stiff, which typically produces a wide spectrum.

Example 4.1.1 *The eigenvalues of the discretized 1D-Poisson equation*

$$\frac{1}{h^2}\begin{bmatrix} 2 & -1 & & 0 \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2 \end{bmatrix}$$

with $h = (n+1)^{-1}$ are

$$\lambda_j = 2h^{-2}(1 - \cos(\pi h j)) \quad , \quad j = 1, \ldots, n.$$

For small j, $\lambda_j \simeq \pi^2 j^2$, while for large j, we have $\lambda_j \simeq 4h^{-2}$. The spectrum is spread out between π^2 and $4h^{-2}$. This is hard for a Krylov method.

In other words, preconditioning is important indeed. In this section, we discuss a preconditioning technique for sparse matrices. A direct method is the perfect preconditioner, since convergence is obtained in less than one iteration. The practical difficulty with a direct method is the memory consumption. The large memory consumption is a result of fill-in, i.e. transformation of zero into non-zero elements. Incomplete factorizations do not perform the factorization completely in order to save memory.

4.2 The classical incomplete factorization

Suppose we are given a non-singular matrix A, for which we can compute the LU factorization $A = LU$ with L lower triangular with ones on the main diagonal and U upper triangular. In general, $L + U$ is much denser than A. If A is symmetric positive definite, we use the Cholesky factorization $A = LL^T$.

Let us now perform the LU factorization so that $L + U$ has the same structure as A, i.e. if an element has to be filled-in in L or U, we only do this when A also has a non-zero in that position. For example, consider the matrix with the pattern

Example 4.2.1

$$\begin{bmatrix} \times & & \times & \times & & \times \\ & \times & & \times & & \\ \times & & \times & & \times & \\ \times & \times & & \times & & \times \\ & & \times & & \times & \times \\ \times & & & \times & \times & \times \end{bmatrix}$$

The LU factorization produces a matrix with the following pattern

$$\begin{bmatrix} \times & & \times & \times & & \times \\ & \times & & \times & & \\ \times & & \times & \circ & \times & \circ \\ \times & \times & \circ & \times & \circ & \times \\ & & \times & \circ & \times & \times \\ \times & & \circ & \times & \times & \times \end{bmatrix}$$

If we do not compute the elements indicated by $\circ$, we have an incomplete factorization.

Denote by S the sparsity pattern of A: $S = \{(i, j) : a_{i,j} \neq 0\}$. The sparsity pattern of $L + U$ lies in S. In principal S does not have to match the sparsity pattern of A. Incomplete factorizations may be defined for any S. The algorithm for incomplete LU factorization is as follows:

Algorithm 4.2.1 (ILU)
1. For $k = 1, \ldots, n$
   1.1. For all $(i, j) \in S$ for which $a_{i,k}a_{k,j} \neq 0$:
      1.1.1. Compute $f_{i,j} = -a_{i,k}a_{k,j}/a_{k,k}$.
      1.1.2. $a_{i,j} = a_{i,j} + f_{i,j}$.

Note that if A is positive definite, the incomplete Cholesky factorization may not exist. It is possible that a diagonal element becomes negative so that the square root cannot be computed.

If A is a symmetric positive definite matrix, the incomplete Cholesky factorization (IC) follows the same ideas.

Definition 4.2.1 (M-matrix) *[20] An $n \times n$ matrix A is called an M-matrix if*

1. $a_{ii} > 0$, *for* $i = 1, \ldots, n$,
2. $a_{ij} \leq 0$, $i, j = 1, \ldots, n$, $i \neq j$, *and*
3. A *is nonsingular and* $A^{-1} \geq 0$.

When A is a matrix with non-positive off-diagonal entries, this is equivalent with any of the following statements:

1. The eigenvalues of A have positive real parts.
2. $A + A^T$ is positive definite.
3. All principal minors of A are M-matrices.

Theorem 4.2.1 *[20]*

1. *For any M-matrix A and for any sparsity pattern S that contains the main diagonal elements, there exist unique L and U (L lower triangular with ones on the main diagonal and U upper triangular) where $L + U$ have the sparsity pattern S, so that*

$$A - LU = R$$

where $r_{ij} = 0$ for all $(i, j) \in S$.

2. *For any symmetric M matrix and (symmetric) sparsity pattern S containing the main diagonal, there exists a unique lower triangular matrix L so that*

$$A - LL^T = R$$

where $r_{ij} = 0$ for all $(i, j) \in S$.

Example 4.2.2

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

has eigenvalues 0.2246, 1.5249, 2.5493, 3.2019, 4.1355, 5.3638 *and has non-positive off-diagonal elements, so it is an M matrix. The (full) Cholesky factorization produces*

$$L = \begin{bmatrix} 1.7321 & & & & & \\ 0 & 1.4142 & & & & \\ -0.5774 & 0 & 1.6330 & & & \\ -0.5774 & -0.7071 & -0.2041 & 1.0607 & & \\ 0 & 0 & -0.6124 & -0.1179 & 1.6159 & \\ -0.5774 & 0 & -0.2041 & -1.2964 & -0.7908 & 1.1485 \end{bmatrix}$$

The positions in the full factorization that result from fill-in are marked in italics. The incomplete Cholesky factorization produces

$$L = \begin{bmatrix} 1.7321 & & & & & \\ 0 & 1.4142 & & & & \\ -0.5774 & 0 & 1.6330 & & & \\ -0.5774 & -0.7071 & 0 & 1.0801 & & \\ 0 & 0 & -0.6124 & 0 & 1.6202 & \\ -0.5774 & 0 & 0 & -1.2344 & -0.6172 & 1.3274 \end{bmatrix}$$

with

$$R = A - LL^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -0.3333 & 0 & -0.3333 \\ 0 & 0 & -0.3333 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.3333 & 0 & 0 & 0 \end{bmatrix} .$$

The eigenvalues of $L^{-1}AL^{-T}$ *are* 1.0000, 0.5305, 1.3536, 1.0000, 1.0000, 1.0000. *The spectrum is pretty well clustered around one.*

4.3 MILU, RILU

The convergence of ILU (incomplete LU) is not always satisfactory. A modification was proposed that significantly improves convergence in some cases.

It was shown that for second order elliptic differential equations (e.g. the Laplacian) the number of iterations with preconditioned CG (PCG) is of the order h^{-1} with and without preconditioning, since $\text{cond}((LU)^{-1}A) = O(h^{-2})$ and $\text{cond}(A) = O(h^{-2})$. A modification to the ILU preconditioner was proposed so that $\text{cond}((LU)^{-1}A) = O(h^{-1})$.

The idea is to ensure that the rowsum of A and $M = LU$ are the same. There are often physical reasons for this: the rowsum corresponds to the mass related to a node in the mesh, or the preconditioner should have the same rigid body mode as the physical problem. Preserving such special characteristics of the physical problem are sometimes advantageous to the iterative solution.

Let e be the n-vector of all ones. This is the condition

$$Ae = Me \ .$$

Since $A = M + R$, $Ae = Me$ implies $Re = 0$. In other words, if $e_j^T Re$ is not zero, it is sufficient to subtract $e_j^T Re$ from m_{jj}.

This can be done in practice as follows: during the sparse factorization, the term $a_{i,k}a_{k,k}^{-1}a_{k,j}$ is subtracted from $a_{i,j}$. If $(i,j) \notin S$, the term $a_{i,k}a_{k,k}^{-1}a_{k,j}$ is lost. It is actually added to r_{ij}. The MILU method (modified ILU) subtracts this term from the main diagonal.

The relaxed ILU (RILU) is a variation on MILU: the terms that are subtracted from the main diagonal are first multiplied by a scalar α with $0 < \alpha < 1$.

Algorithm 4.3.1 (RILU)
1. For $k = 1, \ldots, n$
   1.1. For all (i, j) for which $a_{i,k}a_{k,j} \neq 0$:
   1.2. Compute $f_{i,j} = -a_{i,k}a_{k,j}/a_{k,k}$.
   1.3. If $(i, j) \in S$:
      1.3.1. $a_{i,j} = a_{i,j} + f_{i,j}$.
   1.4. Else:
      1.4.1. $a_{i,i} = a_{i,i} + \alpha f_{i,j}$

4.4 Alternatives to MILU

Some applications are not suitable for ILU preconditioning. Most matrices in applications are no M-matrices. Several authors have proposed preconditioning techniques that first construct an M-matrix B which is 'close' to A.

This is for example the case for indefinite linear systems. The Helmholtz equation is a shifted Laplace equation, i.e.

$$\Delta - k^2$$

Let $(K - k^2M)x = b$ be the discretization of the Helmholtz equation where K is the stiffness matrix and M the mass matrix, i.e. M is the discretization of the variational formulation of the operator 1, where K is the discretization of the Laplacian. The matrix $A = K - k^2M$ is symmetric but indefinite. In order to use ILU preconditioning, one can apply ILU or MILU to $B = K + k^2M$, which is a positive definite matrix. The convergence depends on the spectrum of $M^{-1}A = (M^{-1}B)(B^{-1}A)$ Most eigenvalues of $B^{-1}A$ are close to one. If M is a good preconditioner to B, M will also be a good preconditioner to A. The choice of B is still an active research theme for many classes of applications.

4.5 ILU(p)

ILU and MILU preconditioning are often not strong enough to cluster the spectrum. Therefore, one often enlarges S by not only including the non-zero pattern of A. The standard ILU factorization is also called ILU(0).

We introduce the notion of level of fill in the following algorithm:

Algorithm 4.5.1 (LevFill)
1. $\text{LevFill}(a_{i,j}) = 0$ for all $(i, j) \in S$ and $\text{LevFill}(a_{i,j}) = \infty$ for all $(i, j) \notin S$.
2. For $k = 1, \ldots, n$
   2.1. For all (i, j):
      2.1.1. $\text{Levfill}(a_{i,j}) = \min(\text{Levfill}(a_{i,j}), \max(\text{Levfill}(a_{i,k}), \text{Levfill}(a_{k,j})) + 1)$

Saad [33] defines the level of fill as

$$\text{Levfill}(a_{i,j}) = \min(\text{Levfill}(a_{i,j}), \text{Levfill}(a_{i,k}) + \text{Levfill}(a_{k,j})) + 1)$$

For Example 4.2.1, the levels of fill are given by the following matrix:

$$\begin{bmatrix} 0 & & 0 & 0 & & 0 \\ & 0 & & 0 & & \\ 0 & & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 2 & 0 \\ & & 0 & 2 & 0 & 0 \\ 0 & & 1 & 0 & 0 & 0 \end{bmatrix}$$

When ILU(0) does not work well, we can try ILU(p), i.e. we perform the ILU factorization, and allow a level of fill up to p.

Algorithm 4.5.2 (ILU(p))
1. Levfill($a_{i,j}$ = for all (i, j).
2. For $k = 1, \ldots, n$
   2.1. For all (i, j):
      2.1.1. Compute Levfill($a_{i,k}$).
      2.3.1. If Levfill($a_{i,j}$) $\leq p$, then $a_{i,j} = a_{i,j} - a_{i,k}a_{k,j}/a_{k,k}$.

Increasing the level of fill usually results in more accurate ILU and fewer iterations in the iterative method.

For Example 4.2.2, ILU(1) produces:

$$L = \begin{bmatrix} 1.7321 & & & & & \\ 0 & 1.4142 & & & & \\ -0.5774 & 0 & 1.6330 & & & \\ -0.5774 & -0.7071 & \textit{-0.2041} & 1.0607 & & \\ 0 & 0 & -0.6124 & 0 & 1.6202 & \\ -0.5774 & 0 & \textit{-0.2041} & -1.2964 & -0.6944 & 1.2093 \end{bmatrix}$$

The positions in the full factorization that result from fill-in are marked in italics. The eigenvalues of $L^{-1}AL^{-T}$ are 1.0000, 0.8323, 1.0000, 1.0782, 1.0000, and 1.0000. The spectrum is pretty well clustered around one. Note that

$$A - LL^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.1250 & 0 \\ 0 & 0 & 0 & -0.1250 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

4.6 ILUT

ILU, MILU, and ILU(p) work well for M matrices. Many matrices are no M matrices, and ILU does not work well.

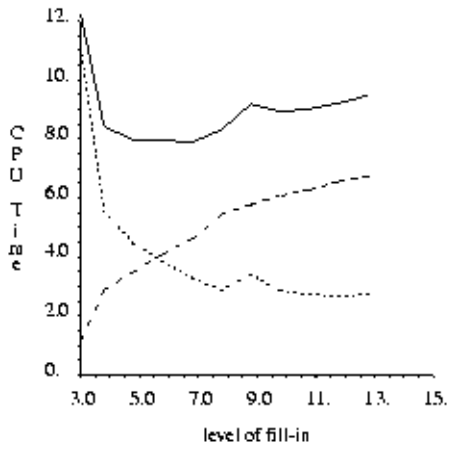


Figure 4.1: Comparison for GMRES with ILUT-preconditioning. The dashed line is the time for the ILUT factorization, the dotted line is for GMRES. The solid line is the sum of both.

Some people proposed to throw away the small elements of A and use a sparse direct method. This appeared to work for some electronic circuit simulations, where the idea was that elements with large resistance can be removed from the system since they have little impact. In general, however, this is not such a good idea.

ILUT is an incomplete factorization method with a threshold drop strategy. The following algorithm illustrates this idea.

Algorithm 4.6.1 (ILUT(p, τ))
1. For $k = 1, \ldots, n$
   1.1. Compute the elements in row k.
   1.2. Throw away all elements smaller than $\tau \|a_{k,:}\|$.
   1.2. Keep the largest p elements.

A variation on ILUT is ILUTP, which is ILUT with partial pivoting.

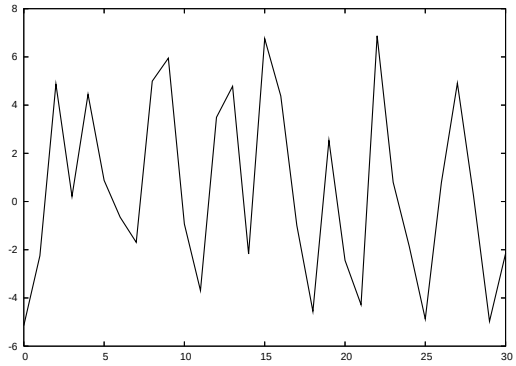
Chapter 5

Multigrid

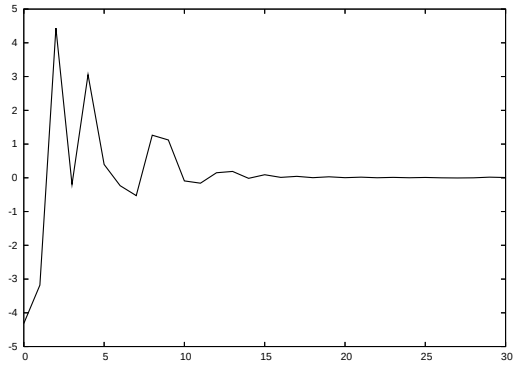
We follow the book van Demmel [12]. We just give an example to illustrate the effects of

- Smoothing
- Restriction
- Prolongation

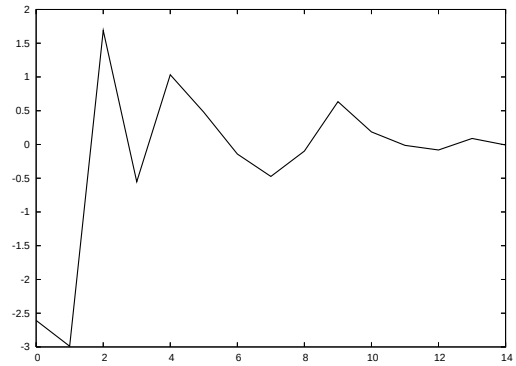
Fine grid before Jacobi iteration: the residual has components in all eigenvalues.



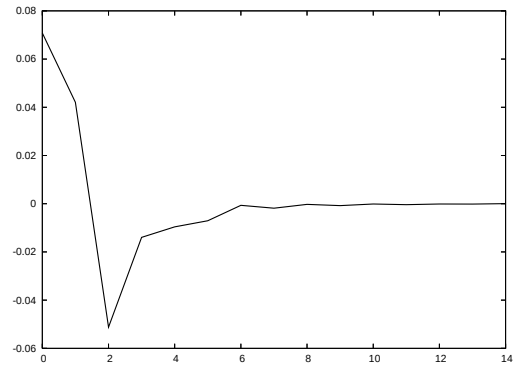
Fine grid after Jacobi iteration: the high frequencies have disappeared from the residual.



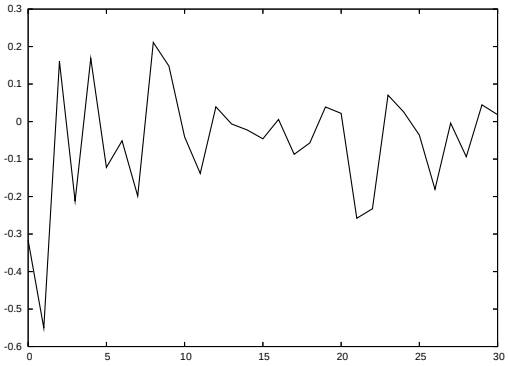
Restriction into coarse grid: the residual has components in all eigenvectors.



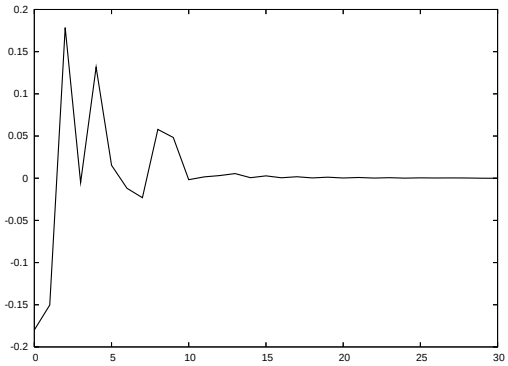
Coarse grid after Jacobi iteration: the high frequencies have disappeared from the residual.



Prolongation from coarse to fine grid: high frequencies return into the residual.



Final Jacobi iteration: high frequencies arre again damped out.



Chapter 6

Domain decomposition

Domain decomposition was in the ninetienth century viewed as a method for solving differential equations on irregular domains that can be decomposed into a small number of regular domains such as squares, rectangles, or disks [35]. See Figure 6.1. Suppose that analytic solutions are available for a disk and a square domain, then domain decomposition is a method that solves the equation on the domain that consists of the sum of the two domains, where the method uses the analytic solutions for each of the two (regular) subdomains. Note that there is significant overlap between the two domains.

The book [37] gives a good overview of some domain decomposition methods. Books on matrix algorithms such as [20] and [12] give a short introduction only. The reason is that domain decomposition is more related to the solution of PDE's than matrix algorithms.

Whereas multigrid is very well suited for regular problems with a uniform grid or when different PDE's are coupled e.g. in multi-physics applications, the development of multigrid is perhaps not so easy. In addition, domain decomposition is a method that can easily be used in parallel computing since each subdomain can be solved independently in parallel.

In this chapter, we subdive domain decomposition methods in two classes: nonoverlapping methods and overlapping methods.

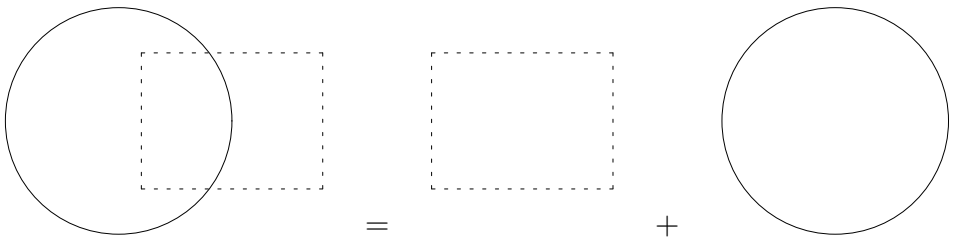


Figure 6.1: Domain decomposition as a method for solving PDE's on irregular domains

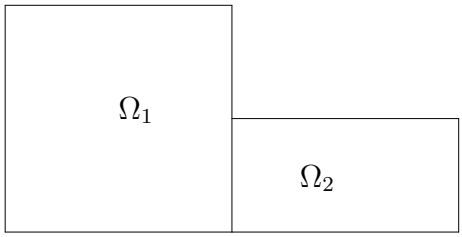


Figure 6.2: L-shaped domain for non-overlapping methods.

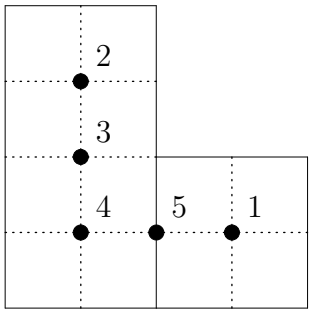


Figure 6.3: Discretization for the Laplace equation with homogeneous Dirichlet boundary conditions.

6.1 Nonoverlapping methods

Consider the L-shaped domain in Figure 6.2. The degrees of freedom that do not lie on the border of the domain are called internal degrees of freedom. We split the domain into two rectangles Ω_1 and Ω_2. Note that domains Ω_1 and Ω_2 only have the interface in common: there is no overlap of internal degrees of freedom.

When we discretize the equation and renumber the unknowns so that the internal degrees of freedom of Ω_1 come first, then the unknowns of Ω_2 and then the interface degrees of freedom, we obtain a linear system as follows:

$$\begin{bmatrix} A_{11} & & A_{13} \\ & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \tag{6.1}$$

where x_1 and x_2 are the unknowns for domain Ω_1 and Ω_2 resp. and x_3 are the interface degrees of freedom. Note that the sparse matrix A has zero blocks in the first row and column since the internal degrees of freedom have no connection with the internal degrees of freedom of the other domain.

This is quite clear when we look at the Laplacian for Figure 6.3

The discretized operator with this numbering becomes the matrix:

$$\left[\begin{array}{c|ccc|c} 4 & & & & -1 \\ \hline & 4 & -1 & & \\ & -1 & 4 & -1 & \\ & & -1 & 4 & -1 \\ \hline -1 & & & -1 & 4 \end{array}\right]$$

which has the structure of (6.1). The LU factorization for (6.1) takes the following form:

$$\begin{bmatrix} A_{11} & & A_{13} \\ & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} I & & \\ & I & \\ A_{31}A_{11}^{-1} & A_{32}A_{22}^{-1} & I \end{bmatrix} \cdot \begin{bmatrix} A_{11} & & A_{13} \\ & A_{22} & A_{23} \\ & & S \end{bmatrix}$$

where

$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23} \ .$$

The inverse of A is given by the product

$$\begin{bmatrix} I & & \\ & I & \\ -A_{31}A_{11}^{-1} & -A_{32}A_{22}^{-1} & I \end{bmatrix} \cdot \begin{bmatrix} A_{11}^{-1} & & -A_{11}^{-1}A_{13}S^{-1} \\ & A_{22}^{-1} & -A_{22}^{-1}A_{23}S^{-1} \\ & & S^{-1} \end{bmatrix}$$

This formula is used for solving a linear system with A. Matrix inversions should be interpreted by linear solves.

The blockwise backward substitution needed for the solution of (6.1) only requires linear solves with A_{11} and A_{22} and S. The matrix S is called the *Schur complement*. The solution with S is tricky. The matrix A_{33} is the descritization of a 1D problem, in our case, a 1D Laplace equation and is thus a tridiagonal matrix. The Schur complement is in general a full matrix. Therefore, this matrix is in general not computed explicitly. Instead, an iterative solver can be used for the solution of a linear system of S. For the Laplace equation, S is a symmetric positive definite matrix and so we can use the conjugate gradient method. For the Laplace equation, the condition number of A grows like $O(h^{-2})$ where the condition number of S grows like $O(h^{-1})$ [].

Another class of non-overlapping domain decomposition methods are the FETI methods [17]. FETI stands for Finite Element Tearing and Interconnecting. It is a method where the interface is not described in terms of the unknowns of the system but where Lagrange multipliers λ are introduced and an additional equation is added that enforces the coupling between the domains.

Assume we have two domains, then for Ω_s we have

$$\Delta_s u_s = f_s \tag{6.2}$$

where we impose that

$$u_1 = u_2 \tag{6.3}$$

$$-\frac{\partial u_1}{\partial n} = \frac{\partial u_2}{\partial n} \tag{6.4}$$

on $\Omega_1 \cap \Omega_2$.

In the dual Schur complement method, we impose on both subdomains

$$-\frac{\partial u_1}{\partial n} = \frac{\partial u_2}{\partial n} = \lambda$$

where λ are called the Lagrange multipliers, and we solve (6.3) iteratively. This is also called the FETI method. After discretization of (6.2) we obtain the following linear system for domain Ω_s:

$$\begin{bmatrix} A_{ss} & A_{s3} \\ A_{3s} & A_{33}^{(s)} \end{bmatrix} \begin{pmatrix} x_s \\ x_3^{(s)} \end{pmatrix} = \begin{pmatrix} f_s \\ (-1)^s \lambda \end{pmatrix} \tag{6.5}$$

$$A_s \mathbf{x}_s = \mathbf{f}_s + B_s^T \lambda \tag{6.6}$$

where $B_s = [0\ I]$.

The interface system is derived from (6.3):

$$\begin{aligned} x_3^{(1)} - x_s^{(2)} &= 0 \\ B_1 \mathbf{x}_1 - B_2 \mathbf{x}_2 &= 0 \\ (B_1 A_1^{-1} B_1^T + B_2 A_2^{-1} B_2^T)\lambda &= B_1 A_1^{-1} \mathbf{f}_1 - B_2 A_2^{-1} \mathbf{f}_2 \ . \end{aligned}$$

This is a linear system in λ, that can be solved iteratively.

The vector λ is the Lagrange multiplier of the constraint (6.3) for the optimization problem

$$\min \int_\Omega x \cdot \Delta x - x \cdot f$$

6.2 Overlapping methods

Overlapping methods means that subdomains overlap as in Figures 6.1 or 6.5. We have two methods: additive Schwarz and multiplicative Schwarz.

We start with a small example in order to explain some properties. Consider the equation

$$\frac{d^2 u}{dx^2} = 0$$

with boundary conditions $u(-1) = u(1) = 1$. The solution is $u \equiv 1$. Let us divide the interval $(-1, 1)$ in two overlapping intervals $\Omega_1 = (-1, \alpha)$ and $\Omega_2 = (-\alpha, 1)$ with $\alpha > 0$. The alternating or multiplicative Schwarz method works as follows:

1. Given initial conditions on the boundary points α and $-\alpha$, i.e. $u(\alpha) = 0$ on Ω_1 and $u(-\alpha) = 0$ on Ω_2.

2. Then compute the solution on Ω_1 using boundary conditions $u_1(-1) = 1$ and $u_1(\alpha) = u_2(\alpha)$

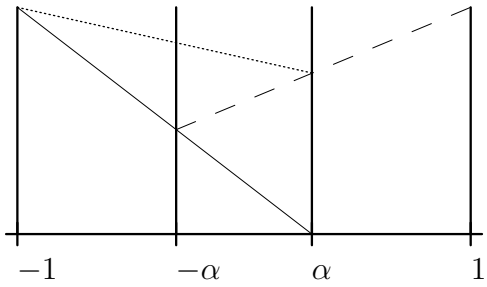


Figure 6.4: Illustration of the alternating or multiplicative Schwarz method. Full line: iteration one, dashed line: second iteration, dotted line: third iteration

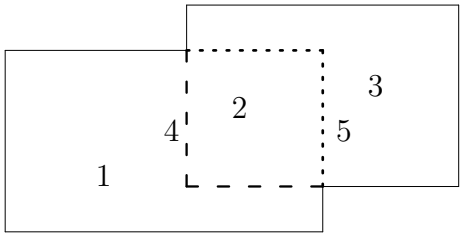


Figure 6.5: Overlapping domain decomposition: ordering of the unknowns

3. Then compute the solution on Ω_2 using boundary conditions $u_2(1) = 1$ and $u_2(-\alpha) = u_1(-\alpha)$.

Repeat this process. The result is shown in Figure 6.4.

Note that if the overlap region $[-\alpha, \alpha]$ becomes larger, the convergence is faster.

For a generic problem we have the following situation. After discretization, we obtain the following linear system for domain Ω_s

$$\begin{bmatrix} A_{11} & & & A_{14} & \\ & A_{22} & & A_{24} & A_{25} \\ & & A_{33} & & A_{35} \\ A_{41} & A_{42} & & A_{44} & \\ & A_{52} & A_{53} & & A_{55} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{pmatrix},$$

where the indices have the following meaning: (1) are the internal unknowns for Ω_1, (3) are the internal unknowns for Ω_2, (2) are the internal unknowns for the overlap $\Omega_1 \cap \Omega_2$, (4) and (5) are the interfaces between Ω_1, $\Omega_1 \cap \Omega_2$ and Ω_2, see Figure 6.5. Then the multiplicative Schwarz algorithm works as follows:

Algorithm 6.2.1 (Multiplicative Schwarz)

1. Given starting vector $x_1, \ldots, x_5$.
2. Until convergence do:
   2.1. Solve

$$\begin{bmatrix} A_{11} & & A_{14} \\ & A_{22} & A_{24} \\ A_{41} & A_{42} & A_{44} \end{bmatrix} \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \tilde{x}_4 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 - A_{25}x_5 \\ f_4 \end{pmatrix}$$

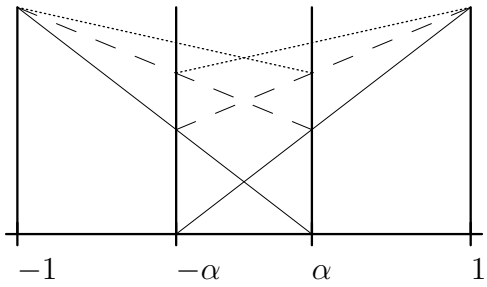


Figure 6.6: Illustration of the additive Schwarz method. Full line: iteration one, dashed line: second iteration, dotted line: third iteration

2.2. Solve

$$\begin{bmatrix} A_{22} & & A_{25} \\ & A_{33} & A_{35} \\ A_{52} & A_{53} & A_{55} \end{bmatrix} \begin{pmatrix} \hat{x}_2 \\ \hat{x}_3 \\ \hat{x}_5 \end{pmatrix} = \begin{pmatrix} f_2 - A_{24}x_4 \\ f_3 \\ f_5 \end{pmatrix}$$

2.3. Set $x = \hat{x}$.

Note that each domain requires interface information from the other domain. This looks like a block-Gauss-Seidel method since the first block is solved using old data and the second block is solved with the data from the first block. The difference with block Gauss-Seidel is that the blocks overlap.

Define the matrix E_s so that it takes a vector from Ω_s and expands it with zeros so that it becomes a vector of the size of the entire domain. Then we can write Step 2.1 as

$$\begin{aligned} \tilde{x} &= x + (E_1^T A E_1)^{-1} E_1^T (f - Ax) \\ \hat{x} &= \tilde{x} + (E_2^T A E_2)^{-1} E_2^T (f - A\tilde{x}) \end{aligned}$$

With $B_s = (E_s^T A E_s)^{-1} E_s^T$, elimination of $\tilde{x}$ leads to

$$\hat{x} = x + \{B_1 + B_2 - B_2 A B_1\}(f - Ax)$$

This is a simple iteration method with preconditioner

$$B_1 + B_2 - B_2 A B_1 \ .$$

The multiplicative Schwarz method is a sequential method, since each subdomain uses the latest information from the other subdomains. The additive Schwarz method only uses the information from the last iteration. In that respect, the additive Schwarz method is related to the block Jacobi method. Figure 6.6 shows the convergence of the additive Schwarz method for our small model problem.

The method can be summarized with the following equations:

$$\begin{aligned} \tilde{x} &= x + (E_1^T A E_1)^{-1} E_1^T (f - Ax) \\ \hat{x} &= \tilde{x} + (E_2^T A E_2)^{-1} E_2^T (f - Ax) \end{aligned}$$

which can be computed fully in parallel. In the additive Schwarz method the solution in the overlap region is the sum of the contributions from both domains. As a result, we have that one iteration can be written as

$$x \leftarrow x + (B_1 + B_2)(f - Ax) \ .$$

Important convergence properties of Schwarz methods are:

- the number of iterations is independent of the mesh size,
- the overlap region can be quite small without affecting the convergence rate,
- and the number of iterations for the multiplicative variant is about half that for additive Schwarz.

Chapter 7

Large scale eigenvalue problems

In this chapter, we discuss the solution of the eigenvalue problems

$$\begin{aligned} Ax &= \lambda x && (7.1)\\ Ax &= \lambda Bx && (7.2)\\ Ax + \lambda B &= \lambda^2 Cx \end{aligned}$$

where A, B and C are matrices of large dimension. Such problems can all be written as a problem of the form

$$A = \lambda x$$

which is the problem we want to discuss.

Often one is interested in one or a few eigenvalues of a large matrix:

- In the stability analysis of systems of differential equations, the sign of the right-most eigenvalue determines stability.
- The computation of the low frequency eigenmodes of a vibrating system is an important problem in the analysis of structures, acoustics, and other vibration problems.
- The computation of energy levels of an atom or molecule can give rise to problems of millions of degrees of freedom.
- Eigenvalue problems also arise in graph theory.
- The google-eigenvector problem is a more recently introduced problem, that is related to graphs and networks.

For small eigenvalue problems, the QR method usually is the method of choice [19]. The complexity is of the order n^3, so this method is not suitable for large scale problems.

In this chapter, we discuss a general framework of large scale eigenvalue computations in practice, with focus on the Arnoldi method. The goal is to compute a small number of eigenvalues:

- s eigenvalues with largest real part,
- s eigenvalues with largest modulus,
- s eigenvalues nearest zero,
- etc.

7.1 Perturbation analysis

Perturbation analysis is the tool for determining bounds to the error of a computed eigenvalue. As for linear systems, the accuracy is usually measured by its residual or backward error.

Let $(\hat{\lambda}, \hat{x})$ be a computed eigenpair, then the residual is defined by

$$r = A\hat{x} - \hat{\lambda}\hat{x} \ .$$

The Bauer-Fike theorem shows a relation between the error on the eigenvalue and the residual norm.

Theorem 7.1.1 *Let A be a non-symmetric matrix and let $AX = X\Lambda$ be the full eigendecomposition. Consider $(\hat{\lambda}, \hat{x})$ and define the residual norm*

$$\rho = \|A\hat{x} - \hat{\lambda}\hat{x}\|_2 \ .$$

Then there is an eigenvalue λ of A so that

$$|\lambda - \hat{\lambda}| \leq \text{cond}(X)\rho \ .$$

For normal matrices the error on the eigenvalue is bounded by the residual norm.

7.2 Simple eigenvalue solvers

In this section, we discuss a few simple eigenvalue solvers that form the basis of modern eigenvalue methods. They are usually not employed in practice.

7.2.1 The power method

The power method computes the dominant eigenvector from the following iterative process:

Algorithm 7.2.1 (Power method)
1. Given x_0 with $\|x_0\| = 1$.
2. For $k = 1, 2, \ldots$:
   2.1. Compute $y_k = Ax_{k-1}$
   2.2. Compute $\lambda_k = x_{k-1}^* y_k$
   2.3. Normalize: $x_k = y_k / \|y_k\|$

Under certain conditions, the algorithm converges to the most dominant eigenvalue and corresponding eigenvector.

7.2.2 Inverse iteration

If one is interested in the eigenvalue nearest zero, the power method cannot be used. The inverse iteration method is the power method applied to A^{-1}. Since the eigenvalues of A^{-1} are λ_j^{-1} where λ_j is an eigenvalue of A, the dominant eigenvalues of A^{-1} correspond to the eigenvalues of A nearest zero.

Often, one is not interested in the eigenvalues near zero, but another point, called shift or pole. The power method is then applied to $(A - \sigma I)^{-1}$.

For the generalized eigenvalue problem (7.2), the shift-invert operator is

$$(A - \sigma B)^{-1} B \ .$$

Its eigenvalues are $\theta_j = (\lambda_j - \sigma)^{-1}$. The dominant θ's correspond to λ's nearest σ.

7.2.3 Newton's method and variations

The eigenpair (λ, x) is the solution of the non-linear system

$$\begin{aligned} (A - \lambda I)x &= 0 \\ 1 - x^* x &= 0 \end{aligned} \tag{7.3}$$

where the second equation is a normalization term. The Newton-Raphson method can be used for solving this equation. Here, we compute a new

$$(\widetilde{\lambda}, \widetilde{x}) = (\lambda, x) + (\eta, z)$$

where (η, z) is the solution of

$$\begin{bmatrix} A - \lambda I & -x \\ -x^* & 0 \end{bmatrix} \begin{pmatrix} z \\ \eta \end{pmatrix} = \begin{pmatrix} -(A - \lambda I)x \\ 0 \end{pmatrix} \ .$$

Writing out the first row, we have

$$(A - \lambda I)z - \eta x = -r \tag{7.4}$$

where r is the residual $r = Ax - \lambda x$.

Let us know assume that $\lambda = x^* A x$, then $x^* r = 0$. Note that $I - xx^*$ is the orthogonal projector onto the space orthogonal to x. Applying $I - xx^*$ to the left of (7.4) produces

$$(I - xx^*)(A - \lambda I)z = r \ .$$

This is a singular linear system with a non-unique solution. If we look for z so that $x^* z = 0$, then the solution is unique, since imposing $x^* z = 0$ on (7.4), produces

$$\begin{aligned} z &= -x + \eta (A - \lambda I)^{-1} x \\ \eta &= (x^* (A - \lambda I)^{-1} x)^{-1} \ . \end{aligned}$$

Since $(I - xx^*)z = z$, we can write z as the solution of

$$(I - xx^*)(A - \lambda I)(I - xx^*)z = -r \ . \tag{7.5}$$

This equation has a unique solution for z orthogonal to x. This is called the correction equation of the Jacobi-Davidson method, which we will discuss in §7.3.4. We have from (7.4) that $\eta = -x^*(A - \lambda I)z$. The solution of (7.5) produces one step of the Newton method where λ is computed from the Rayleigh quotient.

Newton's method for the eigenvalue problem is therefore also called Rayleigh quotient iteration.

7.3 Subspace methods

The previously described methods do not allow the computation of more than one eigenvalue. The methods are not reliable: often, they do not find the desired eigenvalues. Last, but not least, the methods are relatively slow.

Subspace methods have been used for a long time, but during the nineties, the existing methods have obtained a facelift and new methods have been developed that are more efficient and more reliable than existing techniques. See [3] for an overview.

7.3.1 The Arnoldi method

The most popular subspace methods are Krylov methods. Recall the Krylov methods discussed in the context of linear systems.

The Arnoldi process builds an orthonormal basis for the Krylov space

$$\mathcal{K}_k(A, v_1) = \{v_1, Av_1, A^2v_1, \ldots, A^{k-1}v_1\} \ .$$

The Arnoldi process for the matrix A is as follows:

Algorithm 7.3.1 (Arnoldi process)
1. Given v_1 with $\|v_1\| = 1$.
2. For $j = 1, \ldots, k$:
   2.1. Compute $w_j = Av_j$.
   2.2. Compute $h_j = V_j^* w_j$.
   2.3. Compute $\tilde{w}_j = w_j - V_j h_j$.
   2.4. Compute $\beta_j = \|\tilde{w}_j\|$.
   2.5. Compute $v_{j+1} = \tilde{w}_j / \beta_j$.

Collecting the v_j in the $n \times k$ matrix V_k and introducing the Hessenberg matrix

$$H_k = \begin{bmatrix} h_{1,1} & \cdots & \cdots & h_{1k} \\ \beta_1 & h_{2,2} & & \vdots \\ & \ddots & \ddots & \vdots \\ & & \beta_{k-1} & h_{k,k} \end{bmatrix}$$

we find the recurrence relation

$$AV_k = V_k H_k + \beta_k v_{k+1} e_k^* \ . \tag{7.6}$$

We now perform a Galerkin projection of the eigenvalue problem on the Krylov subspace, i.e. find $\hat{\lambda}, \hat{x} = V_k z$ so that $V_k^* r = 0$. From (7.6), we find

$$\begin{aligned} V_k^*(AV_k z - \hat{\lambda} V_k z) &= 0 \\ H_k z - \hat{\lambda} z &= 0 \end{aligned}$$

This is a $k \times k$ Hessenberg eigenvalue problem, which can quickly be solved by the QR method. If $k \ll n$, this is not expensive relative to the cost of the construction of the Krylov vectors. The eigenvalues of H_k are called Ritz values en the approximate eigenvectors $V_k z$ are the Ritz vectors.

The cost of the Arnoldi method is thus

- k matrix vector products with A
- k Gram-Schmidt orthogonalizations (Steps 2.2–2.5).
- The QR method for a $k \times k$ Hessenberg matrix

The Arnoldi method computes k eigenvalues. Typically, the well-separated extremal eigenvalues converge first. Note that this is quite different from the power method: the Arnoldi method does not converge to the dominant eigenvalues. Conversely, the dominant eigenvalues that are well separated converge quickly. Clustered dominant eigenvalues converge slowly.

In contrast with iterative methods for linear systems, is it important that the Krylov vectors are orthogonal. Eigenvalue software performs extra steps to ensure orthogonality.

7.3.2 The Lanczos method

The Lanczos method is the Arnoldi method for Hermitian matrices. The Hessenberg matrix becomes a tridiagonal matrix. Developing Lanczos methods for real life problems is rather technical. See [21] for details.

7.3.3 Spectral transformation

Whereas the Arnoldi and Lanczos methods can be seen as subspace versions for the power method, one could also apply the Arnoldi method to the shift-invert transformation $(A-\sigma I)^{-1}$. This method computes the well-separated eigenvalues of $(A-\sigma I)^{-1}$. Usually these correspond to the eigenvalues of A nearest σ, but this is not always the case.

For the spectral transformation, usually, a direct method is used, since one can exploit the expensive factorization as much as possible. When an iterative method is used, $(A-\sigma I)^{-1}$ is computed up to a tolerance, which does not guarantee the wanted eigenvalues are computed to the desired accuracy. The Jacobi-Davidson method is more appropriate in this case.

7.3.4 The Jacobi-Davidson method

Whereas the Arnoldi method can be seen as an extension of the power method, the Jacobi-Davidson method is a subspace version of the Newton method where in each iteration the correction equation (7.5) is solved by an iterative method. The method works as follows:

Algorithm 7.3.2 (Jacobi-Davidson)
1. Given $(\hat{\lambda}, \hat{x})$.
2. Let $V_1 = [\hat{x}]$.
3. For $j = 1, \ldots, k$ do:
   3.1. Compute the residual $r = A\hat{x} - \hat{\lambda}\hat{x}$
   3.1. Solve the correction equation (7.5).
   3.2. Add the solution y to the subspace: $V_k = \text{orth}([V_{k-1}, y])$.
   3.3. Compute $H_k = V_k^* A V_k$
   3.4. Solve $H_k z = \lambda z$
   3.5. Select a λ and compute $\hat{x} = V_k z$

The difference with Krylov methods is that the Jacobi-Davidson method focuses on one eigenpair. This eigenpair is selected at each iteration. If we are for example interested in the eigenvalue with the largest real part, we select the λ with the largest real part in Step 3.5.
1. Select p columns Z_k: $Z_{k,p}$.
2. Compute $H_p = D_p$.
2. Compute $V_p = V_k Z_{k,p}$.
2. Compute $R_p = AV_p - V_p T_p = R_k Z_p$. In practice, we do not use eigenvectors, but Schur vectors for reasons of numerical stability.

7.4 Implicit restarting for Lanczos's method

The problem with subspace methods is that the subspace dimension k grows as long as the desired eigenvalues have not been computed. In addition, the computation of the eigenvalues of H_k also becomes more expensive.

Therefore, we work with a matrix H_k of maximum size k. As for the restarted GMRES method, we could restart the method with a new initial vector v_1. This idea was explored during the end of the eighties and beginning of the nineties. Suppose we want to compute the eigenvalue with largest real part. Then we select v_1 as the Ritz vector associated with the Ritz value with largest real part.

The best idea so far was proposed by [38] and independently by [28]. It is the idea of implicit restarting.

The k Ritz values computed by the Lanczos method can be divided in the following groups:

- p wanted Ritz values

- $k - p$ unwanted Ritz values

The idea is to discard the unwanted Ritz values. In doing so the subspace is reduced from dimension k to p.

We have the following relation:

$$AV_k - V_k T_k = \beta_k v_{k+1} e_k^* \ .$$

Replacing T_k by its eigendecomposition produces

$$AX_k - X_k D_k = \beta_k v_{k+1} e_k^* Z_k \ .$$

The residual term has the rank-1 form

$$\beta_k v_{k+1} e_k^* Z_k$$

where column j is the residual corresponding to the jth column of X_k. Now select p columns of Z_k. Then, we define

$$\begin{aligned} T_p &= D_k \\ V_p &= V_k Z_p \\ R_p &= \beta_k v_{k+1} e_k^* Z_p \end{aligned}$$

The vector v_{k+1} becomes the $p+1$st column of V_{p+1}. If we look at the new Lanczos recurrence relation, we have the following:

$$AV_p - V_p T_p = \beta_k v_{p+1} e_k^* Z_p \ .$$

The residual term does not have the form $\beta_p v_{p+1} e_p^*$. Therefore, we first have to apply an orthogonal transformation on the right of the recurrence relation. Define a $p \times p$ unitary matrix Q such that $e_k^* Z_p Q = e_p^*$. Q is a sequence of Givens rotations:

$$Q = G_{p-1,p} \cdots G_{2,3} \cdot G_{1,2} \ .$$

The $G_{1,2}$ makes the first element in $e_k^* Z_p$ zero, $G_{2,3}$ makes the second element zero etc. Now, Q has to be applied also on V_p and T_p:

- $T_p^+ = Q^* T_p Q$
- $V_p^+ = V_p Q$
- $\beta_p^+ = \beta_k \| e_k^* Z_p \|_2$

We also need to apply orthogonal transformations on the left and the right of T_p^+ to make it tridiagonal. As a conclusion,

$$AV_p^+ - V_p^+ T_p^+ = \beta_p^+ v_{p+1} e_k^* \ ,$$

which is again a Lanczos recurrence relation. The matrix T_p^+ has the same eigenvalues as T_p. We can now perform $k-p$ additional Lanczos steps to obtain a new recurrence relation of order k. This method is called implicitly restarted Lanczos since it is as if the method is restarted with a new initial vector v_1^+ in an implicit way, i.e. v_j^+ for $j = 2, \ldots, p+1$ are not computed.

We now show several properties in order to better understand the implicitly restarted Lanczos method. Since $v_1^+ = V_k Z_p e_1$, we have

$$v_1^+ = \sum_{j=1}^{p} \alpha_j \hat{x}_j$$

where $\hat{x}_j = V_k z_j$ is the jth Ritz vector. The vector v_1^+ is the starting vector of the Lanczos process of p steps that produces V_p^+ and T_p^+. It is a linear combination of the first p Ritz vectors. This is a good choice, according to the following Theorem:

Theorem 7.4.1 *If v_1 is a linear combination of k eigenvectors of A with distinct eigenvalues, the Krylov space $\mathcal{K}_k(A, v_1)$ is an invariant subspace of A spanned by the eigenvectors that are used in the linear combination.*

Proof. The Krylov subspace is spanned by the sequence $\{A^j v_1\}_{j=0,\ldots,k-1}$. Let $v_1 = \sum_{i=1}^{k} \alpha_i u_i$, then $A^j v_1 = \sum_{i=1}^{k} \lambda_i^j \alpha_i u_i$, so

$$[v_1, Av_1, \ldots, A^j v_1] = [u_1\alpha_1, \ldots, u_k\alpha_k] \begin{bmatrix} 1 & \lambda_1 & \cdots & \lambda_1^j \\ \vdots & & & \vdots \\ 1 & \lambda_k & \cdots & \lambda_k^j \end{bmatrix}$$

The matrix in the right-hand side is a Vandermonde matrix which is of full rank when $j < k$. For $j = k-1$, we can thus choose $u_1, \ldots, u_k$ as basis of the Krylov space. □
So, the hope is that if $\hat{x}_1, \ldots, \hat{x}_p$ are good approximations to eigenvectors, the new Krylov space is almost an invariant subspace, so that it takes only a few more iterations to get the corresponding eigenvectors.

Since Krylov spaces can be regarded as polynomial spaces, we can explain the implicitly restarted Lanczos method by polynomials.

Theorem 7.4.2 *$V_p^+ = \psi_{k-p}(A) V_p$ where*

$$\psi_{k-p}(z) = \gamma \Pi_{i=p+1}^{k} (z - \hat{\lambda}_i) \ .$$

Proof. Recall that the Krylov vectors (columns of V_k) and the Ritz vectors (columns of $X_k = V_k Z_k$) can be used as bases of the Krylov subspace. We give the proof by induction on p. Let $p = 1$, so that $v_1^+ = (A - \hat{\lambda}_k I) v_1$ and the eigenvalues of T_{k-1}^+ are the eigenvalues of T_k except $\hat{\lambda}_k$.

From the recurrence relation, we have that

$$(A - \hat{\lambda}_k I) V_k = V_k (T_k - \hat{\lambda}_k I) + \beta_k v_{k+1} e_k^* \ .$$

Since $(T_k - \hat{\lambda}_k I)$ is singular but the submatrix consistng of the $k-1$ first columns is non-singular, the $k-1$ first columns of

$$V_k(T_k - \hat{\lambda}_k I)$$

span $\{\hat{x}_1, \ldots, \hat{x}_k\}$, i.e.

$$\text{Range}(V_k(T_k - \hat{\lambda}_k I)) = \text{Range}([\hat{x}_1, \ldots, \hat{x}_k]) = \text{Range}(V_{k-1}^+) \ .$$

In other words,

$$(A - \hat{\lambda}_k I)V_{k-1} = V_{k-1}^+ T_{k-1}$$

where T_{k-1} is a full rank $k-1 \times k-1$ matrix. This proves the theorem. □
So, the new initial vector v_1^+ is the old v_1 to which a polynomial filter is applied.

7.5 Example

We show a numerical example to illustrate the properties of implicitly restarted Lanczos. Consider the symmetric matrix

$$A = \text{diag}(1, \ldots, n)$$

with $n = 1000$. We apply the method with $k = 10$ and $p = 5$ starting with the initial vector $v_1 = [1, \ldots, 1]^*/\sqrt{n}$. (Note that $\|v_1\| = 1$.) After the first $k = 10$ Lanczos steps, we have the following triangular matrix

$$T_k = \begin{bmatrix}
500.5000 & 288.6750 & 0.0000 & 0.0000 & 0.0000 & -0.0000 & -0.0000 & 0.0000 - 0.0000 & -0.0000 \\
288.6750 & 500.5000 & 258.1984 & -0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.00000.0000 & 0.0000 \\
0 & 258.1984 & 500.5000 & 253.5451 & 0.0000 & -0.0000 & 0.0000 & 0.00000.0000 & -0.0000 \\
0 & 0 & 253.5451 & 500.5000 & 251.9743 & -0.0000 & -0.0000 & -0.0000 - 0.0000 & -0.0000 \\
0 & 0 & 0 & 251.9743 & 500.5000 & 251.2563 & 0.0000 & -0.00000.0000 & 0.0000 \\
0 & 0 & 0 & 0 & 251.2563 & 500.5000 & 250.8681 & -0.0000 - 0.0000 & -0.0000 \\
0 & 0 & 0 & 0 & 0 & 250.8681 & 500.5000 & 250.63410.0000 & 0.0000 \\
0 & 0 & 0 & 0 & 0 & 0 & 250.6341 & 500.5000250.4817 & 0.0000 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 250.4817500.5000 & 250.3766 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0250.3766 & 500.5000
\end{bmatrix}$$

The Ritz values and residual norms are

λ	:	13.5525	67.9757	160.8012	283.8062	426.0642	574.9358	987.4475	933.0243	840.1988	717.1938
$\|r\|$	:	24.7560	54.7311	80.0556	98.3254	107.8891	107.8891	24.7560	54.7311	80.0556	98.3254

Suppose we want the smallest eigenvalues, select the first $p = 5$ Ritz vectors. After $k - p = 5$ additional Lanczos steps, we obtain:

λ	:	6.6121	32.3857	77.6614	140.6522	218.8932	309.4573	409.5949	528.0484	803.3954	977.5553
$\|r\|$	:	14.2031	32.0945	48.8484	63.7571	76.3210	86.3523	94.7842	135.7242	121.7046	46.9226

And so on:

λ	:	4.0842	19.2154	46.0806	84.1311	132.6274	190.8660	260.6769	551.1126	814.8709	979.956
$\|r\|$	:	9.2906	21.1867	32.7197	43.6579	53.9946	64.5227	87.9593	162.5460	154.0785	61.3495

λ :	2.8964	12.9693	30.9511	56.6823	90.0004	131.0784	241.6878	507.3128	792.7108	974.0993
$\|r\|$:	6.8407	15.6783	24.3051	32.5117	40.0214	45.9465	154.4340	185.1511	162.5817	66.0783
λ :	2.2369	9.4621	22.3835	40.9594	65.1492	95.0430	235.6612	535.7867	809.0614	978.5301
$\|r\|$:	5.0400	11.6291	18.1104	24.3774	30.3422	35.9114	159.9014	195.4625	169.0786	68.8642
λ :	1.8339	7.2891	17.0439	31.1089	49.5118	72.5488	236.1783	500.3414	791.2735	973.7052
$\|r\|$:	4.3508	10.1597	15.9871	21.8758	28.1326	37.2574	172.3348	208.3122	168.4177	68.8307
λ :	1.5779	5.8827	13.5772	24.7255	39.4876	59.4557	228.6763	529.5759	807.1299	978.2626
$\|r\|$:	3.6662	8.6931	13.8378	19.2822	25.7858	43.4956	156.6449	201.4525	169.5652	68.8179
λ :	1.4095	4.9350	11.2323	20.4197	32.8077	53.4525	228.0451	494.7558	789.7077	973.5432
$\|r\|$:	3.2735	7.8904	12.6465	17.7407	23.9441	57.2007	171.1978	210.5856	167.4886	68.1905
λ :	1.2938	4.2618	9.5523	17.3200	27.9546	49.0961	223.1674	526.2630	806.0225	978.1726
$\|r\|$:	2.6938	6.6177	10.6604	14.9837	20.1981	59.7478	160.0980	204.1970	170.2249	68.5365
λ :	1.2122	3.7670	8.3033	15.0008	24.2874	45.6792	223.0127	492.1591	788.9813	973.5106
$\|r\|$:	2.4147	6.0728	9.8459	13.8752	18.6989	65.1306	175.8371	212.1219	167.8324	67.9026

Skipping 15 restarts, we obtain

λ :	1.0010	2.0502	3.4047	5.3980	8.3854	32.1524	211.2473	520.4809	803.9197	978.0821
$\|r\|$:	0.1978	1.1418	2.6241	4.1403	5.7574	70.7821	169.4968	209.9560	174.2699	67.9826

Skipping 10 restarts, we obtain

λ :	1.0000	2.0025	3.0586	4.3960	6.3911	30.7951	210.4412	519.9291	803.7002	978.0608
$\|r\|$:	0.0249	0.2816	1.1177	2.3111	3.5550	71.0246	169.5268	211.4472	175.6345	68.0446

Chapter 8

Modelreduction methods

Consider the linear time-invariant (LTI) system

$$\begin{aligned} \dot{x}(t) &= Ax(t) + bu(t) \\ y(t) &= c^*x(t) + \delta u(t) \end{aligned} \tag{8.1}$$

where u is a function in time, called the input and y is called the output. The vector $x \in \mathbb{C}^n$ is called the state and can be large in size. We assume that this system is stable, i.e. the eigenvalues of A lie in the left half plane (i.e. they have negative real parts).

The goal of modelreduction is to develop another model

$$\begin{aligned} \dot{\hat{x}}(t) &= \hat{A}\hat{x}(t) + \hat{b}u(t) \\ \hat{y}(t) &= \hat{c}^*\hat{x}(t) + \delta u(t) \end{aligned}$$

where $\hat{A}$ is an $m \times m$ matrix with $m \ll n$, such that $\|y - \hat{y}\|$ is small in some sense.

The motivation is to obtain a small size model which is then cheaper to use. Such model can also be represented in the Laplace domain. (We use the same notation for x, y and u as in the time domain.)

$$\begin{aligned} sx &= Ax + bu \\ y &= c^*x + \delta u \end{aligned}$$

The solution is $y = Tu$ where the transfer function T is

$$T = c^*(sI - A)^{-1}b + \delta$$

So, the goal is to find a reduced model

$$\begin{aligned} s\hat{x} &= \hat{A}\hat{x} + \hat{b}u \\ \hat{y} &= \hat{c}^*\hat{x} + \delta u \end{aligned}$$

with transfer function

$$\hat{T} = \hat{c}^*(sI - \hat{A})^{-1}\hat{b} + \hat{\delta}$$

so that $|T - \hat{T}|$ is small in some sense. Usually, the H-infinity norm is used, i.e.

$$\|T - \hat{T}\|_{\mathcal{H}\infty} = \sup_{\omega} |T(i\omega) - \hat{T}(i\omega)|$$

Note that the error is taken for imaginary values of s. The H-infinity norm is the maximum amount of energy increase from the input to the output.

An LTI system is sometimes represented by

$$\left| \begin{array}{cc} A & b \\ c^* & \delta \end{array} \right| \tag{8.2}$$

which is $(n+1) \times (n+1)$ matrix and the reduced system is represented by the $(m+1) \times (m+1)$ matrix

$$\left| \begin{array}{cc} \hat{A} & \hat{b} \\ \hat{c}^* & \hat{\delta} \end{array} \right|$$

In the rest of this chapter, we assume that $\delta = \hat{\delta} = 0$.

8.1 The Padé via Lanczos method

8.2 Balanced truncation

Instead of using the H-infinity norm, we can use the Hankel norm: it is still the maximum energy increase from input to output, but where it is assumed that the input is zero from time $t = 0$ on and where we only consider the output for $t > 0$, i.e. future output.

For the time invariant system, we then have

$$\begin{aligned} y(t) &= \int_{-\infty}^{0} c^* e^{(t-\tau)A} b u(\tau) d\tau \\ &= c^* e^{tA} \int_{0}^{+\infty} e^{\tau A} b u(-\tau) d\tau \\ &= c^* e^{tA} x(0) \end{aligned}$$

So, the output consists of two factors: $x(0)$ which is computed from $u(t)$ and the time integration with initial solution $x(0)$ and zero input.

We define the Hankel map $\mathcal{H}$ as the map of $u((-\infty, 0])$ to $y([0, \infty))$.

Let us now consider the discrete time system

$$\begin{aligned} x_{k+1} &= Ax_k + bu_k \\ y_k &= c^* x_k + \delta u_k \end{aligned}$$

Then, we have

$$y_k = \sum_{-\infty}^{0} c^* A^{k-j} b u_j$$

$$
\begin{aligned}
&= c^* A^k \sum_{0}^{+\infty} A^j b u_{-j} \\
&= c^* A^k x_0 \ .
\end{aligned}
$$

If $u_j = 0$ for $j > 0$, then it is quite easy to see that $x_k = A^k x_0$ and so $y_k = c^* A^k x_0$.

Define the infinite dimensional matrices $\mathcal{O}$ and $\mathcal{C}$

$$
\begin{aligned}
\mathcal{C} &= [b \ Ab \ A^2 b \ldots] \\
\mathcal{O} &= \begin{bmatrix} c^* \\ c^* A \\ c^* A^2 \\ \vdots \end{bmatrix}
\end{aligned}
$$

$\mathcal{C}$ has an infinite number of columns and $\mathcal{O}$ an infinite number of rows. Recall that $x_0 = \mathcal{C}[u_{-1}, \ldots, u_{-\infty}]^T$ and $[y_1, \ldots, y_{+\infty}]^T = \mathcal{C} x_0$. So, we have that

$$
\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{+\infty} \end{pmatrix} = \mathcal{O}\mathcal{C} \begin{pmatrix} u_{-1} \\ y_{-2} \\ \vdots \\ u_{-\infty} \end{pmatrix} \ .
$$

or, in other words, $\mathcal{H} = \mathcal{O}\mathcal{C}$.

Note that $\mathcal{O}$ is related to y, so it is what we observe and that $\mathcal{C}$ is related to u, the input, which is the control of the system.

We introduce the matrices

$$
\begin{aligned}
G_o = \mathcal{O}^*\mathcal{O} &= \sum_{j=0}^{+\infty} (c^* A^j)^T (c^* A^j) \\
G_c = \mathcal{C}\mathcal{C}^* &= \sum_{j=0}^{+\infty} (A^j b)(A^j b)^T
\end{aligned}
$$

We call G_o the observability Gramian and G_c the controllability Gramian.

For the time invariant system, we have the following Gramians:

$$
\begin{aligned}
G_o = \mathcal{O}^*\mathcal{O} &= \int_{j=0}^{+\infty} (c^* e^{At})^T (c^* e^{(}At)) dt \\
G_c = \mathcal{C}\mathcal{C}^* &= \int_{j=0}^{+\infty} (e^{At} b)(e^{At} b)^T dt
\end{aligned}
$$

A linear system can be represented by different realizations. For example, if we define a new state variable in (8.1), $z = Tx$, then (8.1) becomes

$$
\begin{aligned}
\dot{z}(t) &= TAT^{-1} z(t) + Tbu(t) \\
y(t) &= c^* T^{-1} z(t) + \delta u(t)
\end{aligned} \tag{8.3}
$$

The matrix representation of this system is

$$\left|\begin{array}{cc} TAT^{-1} & Tb \\ c^*T^{-1} & \delta \end{array}\right|$$

Note that A and TAT^{-1} have the same eigenvalues.

The Gramians for this realization are

$$\begin{aligned} G_{T,o} &= T^{-*}G_oT^{-1} \\ G_{T,c} &= TG_cT^* \end{aligned}$$

We shall see further that for any positive definite matrices G_o and G_c there is a T so that $G_{T,o} = G_{T,c} = \Lambda$, where Λ is diagonal. A realization whose Gramians are equal and diagonal is called a balanced realization. The name balanced comes from the fact that the observability and controllability are in balance with each other: states that are easily controllable are also easily observable. Note that the eigenvalues of A and TAT^{-1} are the same, but the eigenvalues of G_o and $G_{T,o}$ are different.

Recall the Hankel map $\mathcal{H} = \mathcal{OC}$. The singular values of $\mathcal{H}$ are the square roots of the eigenvalues of $\mathcal{C}^*\mathcal{O}^*\mathcal{OC}$ which has the same *non-zero* eigenvalues as $\mathcal{O}^*\mathcal{OCC}^* = G_oG_c$. If we thus want a low rank approximation of the Hankel map, we have to compute the eigenvalues of G_oG_c. These singular values are called the Hankel singular values. A reduced model is then obtained by dropping the small singular values.

The idea of modelreduction by balanced truncation is to remove the states that correspond to low energy that can be controlled and is hard to observe, i.e. by removing the small Hankel singular values from the balanced realization.

Suppose we have a balanced realization. Then the Hankel singular values are the square roots of the diagonal elements of Λ. Partition

$$\Lambda = \left[\begin{array}{cc} \Lambda_1 & \\ & \Lambda_2 \end{array}\right]$$

so that Λ_1 contains the m largest eigenvalues of Λ. Then define $P = [I\ 0] \in \mathbb{R}^{m\times n}$ and

$$\left|\begin{array}{cc} \hat{A} & \hat{b} \\ \hat{c}^* & 0 \end{array}\right| = \left|\begin{array}{cc} PTAT^{-1}P^* & PTb \\ c^*T^{-*}P^* & 0 \end{array}\right| \tag{8.4}$$

The following is taken from [7, Proposition 1.4.1]

Proposition 8.2.1 *Let (8.2) be a realization of a stable linear time-independent system with transfer function $T(s)$ and let $\hat{T}(s)$ be the transfer function associated with (8.4).*

Then the Gramians for $\hat{T}(s)$ are

$$\hat{G}_o = \hat{G}_c = \Lambda_m$$

where Λ_m consists of the m first rows and columns of Λ, and we have the absolute error bound

$$\|T - \hat{T}\|_{\mathcal{H}\infty} \leq 2 \sum_{j=m+1}^{n} \lambda_j \ .$$

Of particular interest is the error bound as it allows an adaptive choice of the order of the reduced-order model based on a prescribed tolerance, an advantage that the PVL method does not have.

So far, we have not discussed how we compute the balanced truncation.

Lemma 8.2.1 *[22, Lemma 2.2.3] Let $G_c = UU^*$ be the Cholesky factorization of G_c and let $U^*G_oU = R\Sigma^2R^*$ be the eigendecomposition of U^*G_oU. Let $T = \Sigma^{1/2}R^*U^{-1}$, then*

$$TG_cT^* = T^{-*}G_oT^{-1} = \Sigma \ .$$

8.3 Balanced truncation by Cholesky factorization

The Gramians, G_o and G_c satisfy the following Lyapunov equations

$$\begin{aligned} AG_c + G_cA^* + bb^* &= 0 \\ A^*G_o + G_oA + cc^* &= 0 \end{aligned}$$

The solutions of these equations are unique if the eigenvalues of A lie in the left half plane. Since the eigenvalues of A lie in the left half plane, G_o and G_c are positive definite.

Now compute the Cholesky factors U and L so that $G_c = UU^*$ and $G_o = LL^*$. The singular values of U^*L are the Hankel singular values. Let $U^*L = Z\Sigma Y^*$ be the singular value decomposition of U^*L.

Let Σ_1 be the first m rows and columns of Σ and Z_1 and Y_1 be the m leading columns of Z and Y respectively. Define

$$\begin{aligned} W_1 &= LY_1\Sigma_1^{-1/2} \\ V_1 &= UZ_1\Sigma_1^{-1/2} \end{aligned}$$

Note that $W_1^*V_1 = I$, so that $V_1W_1^*$ is an oblique projector.

A reduced model of order m is obtained from

$$\left| \begin{array}{cc} W_1^*AV_1 & W_1^*b \\ c^*v_1 & 0 \end{array} \right|$$

The eigenvalues of G_o and G_c usually decay very rapidly [39] [2]. Therefore the Gramians are ill-conditioned matrices, which may make the Cholesky-factorization sensitive to rounding errors.

8.4 Large scale balanced truncation

The previously explained method is not useful when A is a large sparse matrix. First, the solution of the Lyapunov equation would be too costly and the storage of G_c and G_o infeasible.

For the large scale situation, we exploit the fact that G_c and G_o are close to low rank matrices. The last decade, solution methods for large scale Lyapunov equations have been an important research topic, especially in the context of model reduction.

The Bartels and Stewart [6] algorithm is only suitable for small matrices.

We refer to the work on the ADI and Smith methods [22] and the solution of Lyapunov equations [36].

Chapter 9

Software issues for linear algebra

Linear algebra is used as a black box in many applications over many years. This was only possible thanks to the availability of high quality software. In this chapter, we briefly discuss some software issues and their impact on linear algebra.

9.1 Languages

Linear algebra programming probably started a long time ago using machine instructions. The first portable software, however, was written in Algol and later in Fortran, of which FORTRAN 77 is still most used for linear algebra packages.

In the nineties Fortran 90 was proposed as a modern extension of FORTRAN 77. The language is suitable for linear algebra since it has built-in vector and matrix operations. However, symmetric and sparse matrices are missing. But, most importantly, linking with existing FORTRAN 77 software is not without danger, which we explain later.

The language C is poor for numerical computations. The C99 standard added complex numbers, which was a great step forward. Programming linear algebra in C is not straightforward since there are no vector and matrix types.

The C++ language does not have vector and matrix types either, but packages are developed that offer vector and matrix types for dense and sparse data, with or without structure (bandedness, symmetry, etc). Many of these packages are still under development. We will discuss this in more detail further.

The website `http://www.netlib.org/utk/people/JackDongarra/la-sw.html` contains a list of freely available software for linear algebra.

9.2 Basic linear algebra

In the early days, FORTRAN quickly became the software standard for numerical programming. Since the language had no vector or matrix concepts, libraries were developed. There were various reasons:

- Portability of code

- Efficiency
- Easy of programming in FORTRAN.

The Basic Linear Algebra Subprograms (BLAS) became a standard for vector operations, which were optimized on the available platforms. Initially, the BLAS only contained operations on vectors (BLAS-1) [25]. There were four versions: real single and double precision and complex single and double precision, and there also were mixed precision routines. Here are the routines for real double precision:

- daxpy: $y \leftarrow y + \alpha x$
- ddot: $\alpha = y^T x$
- dscal: $x \leftarrow \alpha x$
- dswap: $x \leftrightarrow y$
- dnrm2: Two-norm
- dasum: one-norm.
- idamax: Index of largest entry in absolute value (can be used for the infinity norm)
- dcopy: $y \leftarrow x$

Here are the routines for complex double precision:

- zaxpy: $y \leftarrow y + \alpha x$
- zdotu: $\alpha = y^T x$
- zdotc: $\alpha = y^* x$
- zscal: $x \leftarrow \alpha x$
- zdscal: $x \leftarrow \alpha x$
- zswap: $x \leftrightarrow y$
- znrm2: Two-norm
- zasum: one-norm of real part plus one-norm of the imaginary part.
- izamax: Index of largest entry in modulus
- zcopy: $y \leftarrow x$

The asum corresponds to the one-norm for real vectors, while for complex vectors it is the sum of the one-norms of the real and imaginary parts. vectors are stored as follows: the first position is indicated by a FORTRAN array type (which is a pointer), the size (i.e. number of elements) and the stride (i.e. the distance between two successive elements). E.g. define the vector x as

```
DOUBLE PRECISION X( 5 )
```

then x_1 is the first element, the size is 5 and the stride is one. so, we can compute the two-norm as:

```
EXTERNAL DNRM2
DOUBLE PRECISION DNRM2
...
V = DNRM2( 5, X, 1 )
```

Let the vector y be the same vector as x, but only the even numbered elements, then we can compute the two norm of y as:

```
V = DNRM2( 2, X(2), 2 )
```

The BLAS-1 were easily optimized on vector machines. All dense linear algebra routines for eigenvalues (EISPACK) and linear systems (LINPACK) were developed using BLAS-1.

The vector machines went out of fashion and vector operations were no longer efficient, especially on machine with a cache memory. The cache is a fast memory that lies between the main memory and the CPU. Often there are several layers of cache. The idea is to keep data as long as possible in cache to reduce the access of the slow main memory. Memory access have more and more become a bottleneck in computations. The BLAS-2 and later the BLAS-3 were introduced [15] [14] for operations on matrices, including matrix times vector and matrix times matrix products, triangular solves, low rank updates, triangular matrix vector products, etc. These routines were (and still are) optimized on machines with cache memory. The routine that is most popular to optimize is the matrix matrix product, the so-called gemm. In FORTRAN, a matrix is stored column by column. The leading dimension is the distance of two successive elements in a row. Consider, for example:

```
DOUBLE PRECISION X( 10, 5 )
```

The leading dimension of X is 10. If we take the submatrix $X(1:5, 2:4)$, the leading dimension is still 10, if the matrix is not copied.

Here are a few interfaces to the FORTRAN routine DGEMM:

```
INTEGER M, N, K
PARAMETER (M=5, N=3, K=4)
DOUBLE PRECISION X( M, K ), Y(K, N), Z( M, N )
DOUBLE PRECISION ALPHA, BETA
EXTERNAL DGEMM
  ..
CALL DGEMM( 'N', 'N', M, N, K, ALPHA, X, M, Y, K, BETA, Z, M )
```

```
CALL DGEMM( 'N', 'N', M, N, 2, ALPHA, X, M, Y, K, BETA, Z, M )
...
CALL DGEMM( 'T', 'N', K, N, M, ALPHA, X, M, Z, M, BETA, Y, K )
```

The first call computes $Z = \alpha XY + \beta Z$. The second call does the same, but only uses the first two columns of X and the first two rows of Y. Note that the leading dimensions do not change (arguments 8 and 10). The last call, computes $Y = \alpha X^T Z + \beta Y$.

9.3 Software packages

There is a large amount of numerical software. In this section, we discuss some software for linear algebra.

9.3.1 LAPACK

LAPACK is the modern version of LINPACK and EISPACK, using BLAS-3 routines for efficiently using the cache. There also is a parallel (MPI) version (ScaLAPACK) and a multi-core version is under development.

LAPACK is the core of linear algebra. Not knowing LAPACK is the same for linear algebra as not knowing that a bicyle needs wheels to ride.

LAPACK contains routines for solving dense and banded linear systems, eigenvalue problems, and singular value decompositions. The package also contains auxiliary routines, such as condition number estimators. The following is the interface for the solution of the $n \times n$ linear system $Ax = b$ with NRHS right-hand sides (columns of b) using LU factorization partial pivoting:

```
DOUBLE PRECISION A(LDA,N), B(LDB,NRHS)
INTEGER IPIV(N)
INTEGER INFO
EXTERNAL DGESV
  ...
CALL DGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )
IF (INFO.NE.0) THEN
  PRINT*,'Solution failed for row ', INFO
  STOP
ENDIF
```

Note that B contains the solution after the call to DGESV. The integer INFO tells whether the solution is computed. When INFO=0 after the call to DGESV, the solution is computed, otherwise $A_{\mathsf{INFO},\mathsf{INFO}} = 0$ was encountered, i.e. A is a singular matrix.

9.3.2 Sparse direct solvers

The package SuperLU is written in C and consists of a sequential code and a parallel MPI code [13]. It is an implementation of a sparse direct linear system solver. The sequential and parallel versions have been developed independently. The sequential version

is outdated. The parallel version is still competitive with other packages, although the memory consumption is high.

MUMPS [29] stands for MUltifrontal Massively Parallel Solver and started its development in the mid-nineties. It is currently one of the best freely available software packages for parallel sparse direct methods. It is written in Fortran 90 and exists in four flavours (real and complex, single and double precision). There is a C-interface. A C++ interface is available from the Boost-Sandbox library [27].

Another software package for sparse direct solves that is used quite often is UMFPACK [11]. It is a multi-frontal method, written in C. An interface to C++ exists [27].

9.3.3 Iterative linear system solvers

There really is a large amount of software for iterative methods. The basic iterative schemes are relatively easy to develop, although parallel (MPI) versions are a little bit trickier. Also, the development of numerically stable Lanczos based methods is not straightforward.

With the book *Templates for the solution of linear systems* [5] software is associated via the website `http://www.netlib.org/templates/`. There is software for Matlab, FORTRAN and a C++ interface. The software is excellent for teaching purposes, but does not have a good reputation for industrial use.

PETSc [4] stands for Portable, Extensible Toolkit for Scientific computation and is an MPI based packages for parallel iterative methods. It is an interface to a range of packages for sparse iterative and direct linear system solvers including Krylov methods and a range of preconditioners such as multigrid, algebraic multigrid, splitting methods, incomplete factorizations. There are interfaces for C, FORTRAN and Fortran 90.

Trilinos [34] is a similar collection of packages written in C++.

9.3.4 vector- and matrix packages

C++ does not have built-in vector and matrix packages. The last ten years, an enormous effort has been done to develop libraries using the power of the language to create an easy syntax and efficient code.

Existing packages are Blitz++ and MTL (Matrix Template Library), and uBLAS. A recently started up project, that is still under development is GLAS. These packages use expression templates for efficiency. GLAS also provides interfaces with existing software, and tries to reconcile common practice of software development in numerical linear algebra with C++.

Here is a small example:

```
#inlude <glas/toolbox/la/la.hpp>
#inlude <glas/glas.hpp>

glas::dense_vector< double > x( 7 ), y( 5 ) ;
glas::dense_matrix< double > a( 7, 5 ) ;
...
```

```
y += trans(a) * x ;

double sum = dot( row(a,2), y ) ;
```

9.4 More about Fortran 90

Fortran 90 (95) was a new language based on the FORTRAN 77 syntax that supports a modular approach to software development. Developing software in Fortran became much more reliable. The addition of true vector and matrix concepts and algorithms made the language in those days suitable for linear algebra.

```
REAL, DIMENSION(:,:) :: X, Y, Z
...
Z = matmul( X, Y )
```

Nevertheless, Fortran 90 never became a popular language, especially not in industry for the following reasons:

- The first compilers were not reliable, the language took a slow start ;
- FORTRAN was no longer used in many newer codes, but rather C++ ;
- modern language concepts (object orientation, true generic programming) are missing.

Currently, Fortran 95 is mainly used in universities in Europe.

A more important disadvantage for linear algebra is that it is not clear how to use BLAS and LAPACK subprograms in Fortran 90, since the stride and leading dimension cannot be computed from Fortran 90 arrays. This has lead to the development of heavy interfaces with BLAS and LAPACK, which have never become popular. In addition, Fortran 90 does not have nice interfaces to sparse and structured matrices, e.g. banded, symmetric, ...

The Fortran 2008 standard would solve this problem and would also provide generic programming mechanisms (parameterized modules and subprograms).

9.5 More about Java

Java struggled with similar problems as Fortran 90 in its early days. The object code was slow, and in addition, numerics were not supported well. The idea of Java is to produce machine independent computational results, i.e. floating point computations were performed by software and not necessarily by hardware, which again led to significant reductions in performance.

Last but not least was a bug in the floating point arithmetic operations with a risk of significant loss of precision in the results.

The only project of an important size that we are aware of is the JAMA project (JAva MAtrix package), which is a basic linear algebra package for Java. It provides user-level classes for constructing and manipulating real, dense matrices. It is intended to serve as the standard matrix class for Java.

Java is not popular for numerics and rarely used in industry.

9.6 More about C++

C++ is a powerful language with a steep learning curve. A large amount of useful software is available:

- STL (Standard Template Library): for operations on vectors, list, maps, associative data structures, etc.
- Boost: graphs, multi-threading, file operations, error handling, vector and matrix operations (MTL and uBLAS)

More and more software is being written in C++. The first compilers produced slow code, but the last five years a tremendous effort was done in improving performance.

9.6.1 Mixing with FORTRAN

Linking with BLAS and LAPACK is technical, but relatively easy. The size and stride can easily be computed for vectors and the reference to the first element is basically a pointer. FORTRAN routines pass all arguments by pointers. The linker for FORTRAN and C++ is the same. First, an interface to C should be made, e.g.

```
external "C" {
  void daxpy_( int* n, double* alpha, double* x, int* stride_x,
               double* y, int* stride_y ) ;
}
```

The function daxpy_ can now be used in the C++ code. The underscore is needed when we use the GCC compilers, e.g. It is not needed with Windows Visual Studio.

Bibliography

[1] G. Alléon, M. Benzi, and L. Giraud. Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. Technical Report TR/PA/97/05, CERFACS, Toulouse, France, 1997.

[2] A.C. Antoulas, D.C. Sorensen, and Y. Zhou. On the decay rate of hankel singular values and related issues. *Systems & Control Letters*, 46:323–342, 2002.

[3] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, PA, USA, 2000.

[4] S. Baley, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, C. McInnes, B. Smith, and H. Zhang. Petsc users manual. Technical Report ANL-95/11 -Revision 2.3.3, 1995. `http://acts.nersc.gov/petsc/`.

[5] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, Philadelphia, PA, USA, 1994.

[6] R.H. Bartels and G.W. Stewart. Algorithm 432: Solution of the matrix equation ax + xb = c. *Comm. ACM*, 15:820–826, 1972.

[7] P. Benner and S Quintana-Ortí. *Model reduction based on spectral projection methods*, pages 5–48. Springer, Heidelberg, 2005.

[8] M. Benzi and M. Tuma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 19(3):968–994, 1998.

[9] J. Bunch. Partial pivoting strategies for symmetric matrices. *SIAM Journal on Numerical Analysis*, 11(3):521–528, 1971.

[10] J. Bunch and Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31:163–179, 1977.

[11] T.A. Davis. Users' guide for the Unsymmetric-pattern MultiFrontal Package (UMFPACK). Technical Report TR-95-004, Computer and Information Sciences Department, University of Florida, Gainesville, FL, USA, 1995.

[12] J. W. Demmel. *Applied numerical linear algebra*. SIAM, Philadelphia, PA, USA, 1997.

[13] J. W. Demmel, J. R. Gilbert, and X. S. Li. SuperLU users' guide. Available through `http://www.cs.berkeley.edu/~demmel/SuperLU.html` or `http://www.nersc.gov/~xiaoye/SuperLU/`, 1999.

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14:18–32, 1988.

[15] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14:1–17, 1988.

[16] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. Software - Environments - Tools. SIAM, Philadelphia, PA, USA, 1998.

[17] C. Farhat and F.-X. Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal of Numerical Methods in Engineering*, 32:1205–1227, 1991.

[18] R.W. Freund and N.M. Nachtigal. QMR : a quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematic*, 60:315–339, 1991.

[19] G. Golub and C. Van Loan. *Matrix computations*. The Johns Hopkins University Press, 3rd edition, 1996.

[20] A. Greenbaum. *Iterative methods for solving linear systems*. SIAM, Philadelphia, PA, USA, 1997.

[21] R.G. Grimes, J.G. Lewis, and H.D. Simon. A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM Journal on Matrix Analysis and Applications*, 15:228–272, 1994.

[22] S. Gugercin and Jing-Rebecca Li. *Smith-type Methods for Balanced Truncation of Large Sparse Systems*. Springer-Verlag, Berlin, Heidelberg, 2005.

[23] M. Heath, E. Ng, and B. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–460, 1991.

[24] E Heikkola and T. Rossi. A parallel fictitious domain method for the three-dimensional helmholtz equation. *SIAM Journal on Scientific Computing*, 24(5):1567–1588, 2003.

[25] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[26] J.W.H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990.

[27] K. Meerbergen, K. Fresl, and T. Knapen. C++ bindings for external software libraries. Technical Report TW506, K.U.Leuven, Department of Computer Science, 2007.

[28] R.B. Morgan. On restarting the Arnoldi method for large nonsymmetric eigenvalue problems. *Mathematics of Computation*, 65:1213–1230, 1996.

[29] MUMPS. MUlfrontal Massively Parallel Solver, 2001. `http://graal.ens-lyon.fr/MUMPS/`.

[30] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12:617–629, 1975.

[31] Y. Renard, J. Pommier, and C. Lecalvez, 2003. http://www.gmm.insa-tlse.fr/getfem/gmm.html.

[32] T. Rossi and J. Toivanen. A parallel fast direct solver for block tridiagonal systems with separable matrices of arbitrary dimension. Report 21/1996, University of Jyväskylä, Department of Mathematics, Laboratory of Scientific Computing, Finland, 1996.

[33] Y. Saad. *Iterative Methods for Linear Systems*. SIAM, Philadelphia, 2003.

[34] M. Sala, M. Heroux, and D. Day. Trilinos. `http://trilinos.sandia.gov/`.

[35] H.A. Schwarz. *Gesammelte Mathematische Abhandlungen*, volume 2. Springer, Berlin, 1890. (first published in Viertaljahrschrift Naturforsch. Ges. Zürich, 15 (1870), pp. 272–286).

[36] V. Simoncini and V. Druskin. Convergence analysis of projection methods for the numerical solution of large Lyapunov equations. *??*, ??:18, 2007. Revised version, April 2008.

[37] B. Smith, P. Bjorstad, and W. Gropp. *Domain decomposition. Parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press, London, U.K., 2996.

[38] D.C. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM Journal on Matrix Analysis and Applications*, 13:357–385, 1992.

[39] Penzl T. Eigenvalue decay bounds for solutions of Lyapunov equations: the symmetric case. *Systems Control Lett.*, 40:139–144, 2000.

[40] H.A. van der Vorst. BiCGStab: a fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.