TOPICS IN NUMERICAL LINEAR ALGEBRA

AXEL RUHE[1]

September 7, 2007

[1]Royal Institute of Technology, SE-10044 Stockholm , Sweden (ruhe@kth.se). *Material given to courses in Applied Numerical Analysis and Scientific Computation Computational Algebra.*

Preface

This booklet contains some topics in numerical linear algebra that I consider useful for any student interested in Scientific Computation but which is not included in the common beginners course.

For a more comprehensive presentation the reader is referred to the text book

James W. Demmel, *Applied Numerical Linear Algebra*, SIAM, 1997

In these notes, I give the emphasis to algorithm descriptions and some proofs that give a good understanding of how the algorithms behave in a practical situation. For the very fundamental perturbation theory, you need to consult the book. The Demmel book also gives a wealth of information on inplementation aspects collected during the long LAPACK effort.

In Chapter 1, a few introductory examples of direct methods for linear systems are given, together with a short introduction to perturbation theory and error analysis.

Chapter 2 introduces direct methods for sparse matrices. There is a large class of matrices for which these algorithms are the most effective way of solving a linear system. The theory is interesting, it is in the border area between Numerical Analysis and Computing Science.

Chapter 3 treats the Singular Value Decomposition. I could not resist giving my students access to the proof introduced in the Golub Van Loan text book. The importance of the SVD in data analysis and signal processing, motivates giving it a prominent position in any Scientific Computing curriculum.

In Chapter 4, eigenvalues are discussed. A proof of the important Schur theorem is given, together with an introduction to the Lanczos and Arnoldi algorithms for iterative solution of large sparse eigenvalues.

Finally in Chapter 5, I give an introduction to iterative algorithms for linear systems. I think time is ripe to put Krylov space based algorithms in the center here, who on earth does still use the SOR method discussed in most text books?

This is an augmented first edition. I hope to make up a more comprehensive and connected text later, but yet, remember my own motto *"When it comes to text books, the first edition is always the best!"*

Stockholm September 7, 2007

Axel Ruhe

Chapter 1

Linear Systems

In this chapter we consider linear systems of equations

$$Ax = b \tag{1.1}$$

The matrix A is small enough to be stored in the main memory of your machine. This may be rather large on the computers of today, say order n up to a few thousands. We discuss algorithms based on Gaussian elimination, this gives a direct algorithm that computes a solution x in a finite number of arithmetic operations.

1.1 Gaussian elimination

The Gaussian elimination algorithm does $n-1$ major steps that transform the matrix A into an upper triangular matrix U. A system with an upper triangular matrix can be solved by back substitution.

In major step i a multiple of the ith row is subtracted from the rows below i so that the i th column is made into zeros in these rows.

```
for i = 1 to n − 1
    for j = i + 1 to n
        l_{j,i} = a_{j,i}/a_{i,i}
        for k = i + 1 to n
            a_{j,k} = a_{j,k} − l_{j,i} a_{i,k}
        end k
    end j
end i
```

We express this in matrix language. Start with $A^{(1)} = A$. In step i, premul-

tiply by an *elimination* matrix,

$$L_i = I + l_i e_i^T = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & l_{i+1,i} & 1 & \\ & \vdots & \ddots & \\ & l_{n,i} & & 1 \end{pmatrix}$$

Its inverse is obtained by a sign change,

$$L_i^{-1} = I - l_i e_i^T$$

$$A^{(i+1)} = L_i^{-1} A^{(i)}$$

We continue until

$$A^{(n)} = L_{n-1}^{-1} A^{(n-1)} = \cdots = L_{n-1}^{-1} \dots L_1^{-1} A^{(1)}$$

The final matrix $A^{(n)}$ is upper triangular and we have gotten a LU factorization

$$A = LU \text{ with } L = L_1 L_2 \dots L_{n-1} \text{ and } U = A^{(n)}$$

This is the basic algorithm description. To make a practical code we have to observe:

Pivoting The diagonal element $a_{i,i}$, the *pivot* element has to be nonzero. We can exchange row i and a later row j such that $a_{j,i}$ is nonzero. Most often we move up the element with largest absolute value. This makes sure that all multipliers $|l_{j,i}| \leq 1$. If the matrix A is nonsingular, it is always possible to find a nonzero $a_{j,i}$ in the lower part of the ith column.

In matrix language we get

$$PA = LU, \text{ where } P = P_{n-1} \dots P_1 \text{ with } P_i \text{ giving the } i\text{th row exchange}$$

The interchanges have to be determined in each step i but we will get the same factors as if we factored a matrix PA where the rows are permuted before the elimination.

Storage We store the multipliers $l_{j,i}$ in the same place as the elements $a_{j,i}$ that are put to zero.

Operation count In each major step i we do $n - i$ divisions and then we update $(n-i)^2$ elements with a multiplication followed by a subtraction. Summing up we get

$$\sum_{i=1}^{n-1} \big((n-i)\text{Div} + (n-i)^2(\text{Add} + \text{Mul})\big) = \frac{n^3}{3}(\text{A+M}) + O(n^2)$$

In the second equality, we have used that sums of powers behave like integrals. We use the common notation $O(n^p)$ for all terms of degree p or lower. For large orders n the leading term dominates.

Execution order There are several ways of reordering the arithmetic operations, we may nest the for i, j and k statements in any order. The ijk ordering given above is natural, when you compute by hand and write rows. On a computer, it is important to keep memory references local, to take advantage of cache memory. In FORTRAN the matrices are stored by columns and then the innermost loop should keep in the same column, use ikj or kij. We also distinguish between inner product and outer product formulations, the algorithm above gives a row oriented outer product formulation. The innermost loop over j and k updates the southeast block by an outer product between the column $l(i+1:n,i)$ of multipliers and the row $a(i,i+1:n)$. The order kij gives an inner product formulation, where the inner product of the jth row and the kth column is subtracted from the element $a_{j,k}$.

1.2 Solving a linear system

We solve the linear system $Ax = b$ by the following algorithm

ALGORITHM FACTOR AND SOLVE

1. Factorize $PA = LU$ by Gaussian elimination.
2. Permute the right hand side $b' = Pb$
3. Forward substitution $b'' = L^{-1}b'$
4. Back substitution $x = U^{-1}b''$

END.

The forward and backward substitution steps need n^2 additions and multiplications.

1.3 A numerical example

Let us see what happens when we do Gaussian elimination on

$$A = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{bmatrix}$$

The first pivot is 16 which is already the largest, we get multipliers and update the rest of the matrix and get

$$A^{(2)} = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 0.3125 & 10.375 & 9.0625 & 3.9375 \\ 0.5625 & 5.875 & 4.3125 & 4.6875 \\ 0.25 & 13.5 & 14.25 & -2.25 \end{bmatrix}$$

Note that the first row is unchanged and that the first column contains the multipliers $l_{j,1}$.

In the second step, we use the second pivot 10.375, put multipliers in the second column and update the third and fourth row and column. The first two rows are left unchanged. We get:

$$A^{(3)} = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 0.3125 & 10.375 & 9.0625 & 3.9375 \\ 0.5625 & 0.56627 & -0.81928 & 2.4578 \\ 0.25 & 1.3012 & 2.4578 & -7.3735 \end{bmatrix}$$

In the third and final step, the pivot is -0.81928, and we update and get:

$$A^{(4)} = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 0.3125 & 10.375 & 9.0625 & 3.9375 \\ 0.5625 & 0.56627 & -0.81928 & 2.4578 \\ 0.25 & 1.3012 & -3 & -2.6645e-015 \end{bmatrix}$$

Note that the last element is tiny $A_{4,4} = -2.6645e-015$. The matrix is actually singular, but rounding errors give rise to a small element. It is like the machine epsilon, scaled by the size of the matrix elements.

We now get the lower triangular L and the upper triangular U as:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.3125 & 1 & 0 & 0 \\ 0.5625 & 0.56627 & 1 & 0 \\ 0.25 & 1.3012 & -3 & 1 \end{bmatrix} \text{ and } U = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 0 & 10.375 & 9.0625 & 3.9375 \\ 0 & 0 & -0.81928 & 2.4578 \\ 0 & 0 & 0 & -2.6645e-015 \end{bmatrix}$$

This was the result without pivoting. In the built in MATLAB lu factorization, row pivoting is used, let us follow that, to see the difference.

The first step is the same. In the second step we interchange the second and fourth rows and get,

$$A^{(3)} = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 0.25 & 13.5 & 14.25 & -2.25 \\ 0.3125 & 0.76852 & -1.8889 & 5.6667 \\ 0.5625 & 0.43519 & -1.8889 & 5.6667 \end{bmatrix}$$

Note that also the multipliers are permuted. In the last step, no interchange is needed, and we get:

$$A^{(4)} = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 0.25 & 13.5 & 14.25 & -2.25 \\ 0.3125 & 0.76852 & -1.8889 & 5.6667 \\ 0.5625 & 0.43519 & 1 & 3.5527e-015 \end{bmatrix}$$

Note that also now we have a tiny number in the bottom, the matrix is still singular.

1.4 Error analysis for linear systems

The error analysis of a numerical computation is most often divided into two phases, *perturbation theory* and *rounding error analysis*. In the perturbation theory, it is investigated how much the result of a computation is influenced by perturbations in the data. Such a perturbation bound depends on the data but is not dependent on how the actual computation has been done. It is also useful for all kinds of errors, not only those coming from the computation, but also uncertainties in the measurements that gave the data for an applied computation. The rounding error analysis is most often done as a *backward rounding error analysis*, saying that the result of the actual computation is the same as the result of an exact computation starting on a perturbed set of data. A bound on these perturbations can then be inserted in a perturbation theorem to bound the error in the computed result.

It may sound awkward to first go backward and then forward, but this was one of the early successes in the history of numerical computation. Before the computer age, in the 1930ies, one tried with a forward analysis, assuming some uncertainty in the data. For each operation one got a bigger uncertainty in the results and found that the numerical solution of large systems should be an risky endeavor. It was J H Wilkinson in England that showed that one could prove that the computed solution was an exact solution of a problem that was close to the original one in most cases much closer, and that the solution of the perturbed problem might be far off but most often was quite close to the original. Now systems with thousands of equations are solved reliably.

We first introduce vector and matrix norms, a way of measuring the size of a multidimensional entity. This is a special case of the rich mathematical theory of normed linear spaces. We then give a few perturbation results for the solution of a linear system. Last we indicate how large the rounding errors are that are introduced while we factorize a matrix with Gaussian elimination. Very similar methods can be used to analyze other computations, like solving a system and matrix vector multiplications. It is assumed that the reader is acquainted to how the IEEE standard floating point arithmetic works.

1.4.1 Norms of vectors and matrices

A *vector norm* is a measure of the length of a vector x. It is a nonnegative number $\|x\|$ that satisfies the following three conditions:

$$\begin{aligned} \|x\| &\geq 0 && \|x\| = 0 \text{ only when } x = 0 && (1.2)\\ \|\alpha x\| &= |\alpha|\|x\| && \text{for scalar } \alpha && (1.3)\\ \|x + y\| &\leq \|x\| + \|y\| && \text{Triangle inequality} && (1.4) \end{aligned}$$

The most common norms are the p norms defined by

$$\|x\|_p = \begin{cases} (\sum_i |x_i|^p)^{1/p} & \text{for } 1 \leq p < \infty \\ \max_i |x_i| & \text{for } p = \infty \end{cases}$$

and among those $p = 2$, the least squares or *Euclidean* norm and $p = \infty$, the maximum or *Chebyshev* norm.

A *matrix norm* is likewize a measure of the size of the matrix, regarded as a vector in R^{mn}, satisfying (1.2)(1.3)(1.4), but now we are also interested in the matrix as an operator describing a linear mapping. We say that matrix norms are *consistent* if

$$\|AB\| \leq \|A\|\|B\| \tag{1.5}$$

for any pair of matrices that can be multiplied. Note that the matrices may be of different dimension, $m \times n$ and $n \times p$, giving a product of dimension $m \times p$, so we have three different norms.

We can let two vector norms define a matrix norm, defined by

$$\|A\| \equiv \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

the *operator norm* or induced or subordinate matrix norm. It satisfies the conditions (1.2)(1.3)(1.4) as well as (1.5). Moreover the unit matrix I has

$$\|I\| = 1$$

for any operator norm.

1.4.2 Perturbations of linear systems

Now look at a linear system $Ax = b$ (1.1), and assume that the data A and b have been perturbed into $A + \delta A$ and $b + \delta b$. Then the solution will be changed into $x + \delta x$, and now we want to bound the perturbation δx. '

We see that $x + \delta x$ is the solution of the linear system

$$(A + \delta A)(x + \delta x) = b + \delta b \tag{1.6}$$

provided that the perturbation δA does not make the matrix singular.

Subtract the original system (1.1) from this and get,

$$\begin{aligned} (A + \delta A)(x + \delta x) &= b + \delta b \\ Ax &= b \\ \hline (A + \delta A)\delta x + \delta A x &= \delta b \\ \delta x &= (A + \delta A)^{-1}(\delta b - \delta A x) \end{aligned} \tag{1.7}$$

In the simplest case, when there is no perturbation on the matrix A, we see that $\delta x = A^{-1}\delta b$ and its norm can be bounded as

$$\|\delta x\| \leq \|A^{-1}\|\|\delta b\|$$

where any pair of consistent matrix and vector norms can be used.

This is good, but we are better served by a bound that is invariant under scaling. We can obtain such a bound on the relative perturbation, by noting that

$$\|b\| \leq \|A\|\|x\|$$

Multiply those inequalities together and divide with the product $\|x\|\|b\|$ and get

$$\frac{\|\delta x\|}{\|x\|} \leq \|A^{-1}\|\|A\|\frac{\|\delta b\|}{\|b\|} \tag{1.8}$$

a bound for the relative perturbation in the solution x as a multiple of a bound for the relative pertubation to the right hand side b.

The factor multiplying the perturbation

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

is called the *condition number* of the matrix. It is always larger than one, $\kappa(A) \geq 1$ (Why?). For an operator norm, it is the quotient between the largest possible expansion and the smallest possible contraction of a vector under the mapping given by the matrix A,

$$\kappa(A) = \frac{\max_x(\|Ax\|/\|x\|)}{\min_x(\|Ax\|/\|x\|)}$$

If the matrix is singular, some vectors are mapped to zero, and the condition number $\kappa(A)$ is infinite.

It is a little more complicated to find a bound on the perturbation to the solution when the matrix is perturbed. It can then happen that the perturbation makes the matrix singular. We take norms in the original expression (1.7) for δx getting

$$\|\delta x\| \leq \|(A+\delta A)^{-1}\|(\|\delta b\| + \|\delta A\|\|x\|) \tag{1.9}$$

We now need to bound the norm of the inverse of the perturbed matrix and then we express it as

$$(A+\delta A)^{-1} = A^{-1}(I + \delta A A^{-1})^{-1}$$

and bound its norm by

$$\|(A+\delta A)^{-1}\| \leq \|A^{-1}\|(1 - \|\delta A\|\|A^{-1}\|)^{-1}\,. \tag{1.10}$$

Here we have used the fact that

$$\|(I+X)^{-1}\| \leq \frac{1}{1-\|X\|}$$

whenever $\|X\| < 1$. This is proved by taking the expansion

$$(I+X)(I - X + X^2 + \ldots X^{k-1}) = I - (-1)^k X^k$$

and noting that the the second term in the right hand side tends to the limit zero for growing k whenever $\|X\| < 1$. The series in the second factor of the left hand side is majorized by the geometric series

$$1 + \|X\| + \|X\|^2 + \cdots = 1/(1 - \|X\|)$$

We get the bound (1.10) by setting $X = \delta A A^{-1}$.

We are now ready to get the expression for the relative perturbation to x by dividing both sides of the inequality (1.9) with $\|x\|$ and noting that $\|A\|\|x\| \geq \|b\|$ we get

$$\begin{aligned} \frac{\|\delta x\|}{\|x\|} &\leq \frac{\|A^{-1}\|}{1 - \|\delta A\|\|A^{-1}\|}\left(\frac{\|\delta b\|}{\|x\|} + \|\delta A\|\right) \\ &\leq \frac{\kappa(A)}{1 - \kappa(A)\frac{\|\delta A\|}{\|A\|}}\left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|}\right) \end{aligned} \tag{1.11}$$

Note the difference to the case (1.8) when only the right hand side b was perturbed. The factor in front is larger than the condition number $\kappa(A) = \|A\|\|A^{-1}\|$ and there is a perturbation δA of size $\|\delta A\| = 1/\|A\|$ to the matrix A that makes the denominator zero and the perturbed matrix singular.

1.4.3 Rounding errors in Gaussian elimination

Let us now follow the Gaussian elimination process in detail, to see how rounding errors lead us to compute a solution to a perturbed problem and find a bound on the size of this perturbation. It is assumed that we use standard IEEE floating point arithmetic so that the result of a floating point operation, where $\odot$ stands for any one of $+ - \times /$, satisfies

$$\mathrm{fl}(a \odot b) = (a \odot b)(1 + \delta) \quad |\delta| \leq \epsilon \tag{1.12}$$

One operation can always be done with a forward relative error bounded by machine accuracy $\epsilon = 2^{-52} \approx 2.204 \times 10^{-16}$. If we do more than one operation, all we can show is that we get a correct result for perturbed data. Here we need the bound,

$$\mathrm{fl}\left(\sum_{i=1}^{d} x_i y_i\right) = \sum_{i=1}^{d} x_i y_i (1 + \delta_i) \quad \text{with } |\delta_i| \leq d\epsilon$$

We follow the Gaussian elimination for each matrix element in the algorithm. The elements a_{jk} in the upper triangle $j \leq k$ are modified the first $j - 1$ steps and are then left as u_{jk}

$$u_{jk} = a_{jk} - \sum_{i=1}^{j-1} l_{ji} u_{ik}$$

Those in the lower triangle $j > k$ are modified in the first $k - 1$ steps and are then divided by the pivot element u_{kk} to become l_{jk},

$$l_{jk} = \frac{a_{jk} - \sum_{i=1}^{k-1} l_{ji} u_{ik}}{u_{kk}}$$

Replacing all the arithmetic operations in the computation of u_{jk} by the floating point operations (1.12) we get

$$u_{jk} = \left(a_{jk} - \sum_{i=1}^{j-1} l_{ji} u_{ik} (1 + \delta_i) \right) (1 + \delta')$$

with $|\delta_i| \leq (j-1)\epsilon$ bound the backward error in the scalar product and $|\delta'| \leq \epsilon$. Then we get an expression for

$$\begin{array}{rcl} a_{jk} & = & \frac{1}{1+\delta'} u_{jk} l_{jj} + \sum_{i=1}^{j-1} l_{ji} u_{ik} (1 + \delta_i) \\ & = & \sum_{i=1}^{j} l_{ji} u_{ik} + \sum_{i=1}^{j} l_{ji} u_{ik} \delta_i \\ & \equiv & \sum_{i=1}^{j} l_{ji} u_{ik} + e_{jk} \end{array}$$

where

$$|e_{jk}| = \left| \sum_{i=1}^{j} l_{ji} u_{ik} \delta_i \right| \leq \sum_{i=1}^{j} |l_{ji}||u_{ik}|(j-1)\epsilon = (j-1)\epsilon(|L||U|)_{jk}$$

where we have assumed that δ is small enough to make $|1/(1+\delta') - 1| \leq \epsilon$. Here the absolute value notation is used to denote the matrix of absolute values,

$$(|A|)_{jk} = |a_{jk}|$$

A similar analysis for the lower triangle $j > k$ gives

$$a_{jk} \equiv \sum_{i=1}^{k} l_{ji} u_{ik} + e_{jk}$$

with

$$|e_{jk}| \leq (k-1)\epsilon(|L||U|)_{jk}$$

This way we have got bounds on the absolute values of all elements of the difference E. It is worth noting that this bound is dependent on the size of the elements of L and U. If we perform Gaussian elimination with partial pivoting, the most common variant, all elements of the lower triangular L are smaller than one. The elements of U may be larger, actually they may be as large as $2^{n-1} \max |a_{jk}|$. This kind of growth is very rare and has happened only on specially constructed examples, so in most cases it is ignored. One good precaution is to keep track on the element growth.

Chapter 2

Direct methods for Sparse Matrices

Many matrices coming from practical applications have only few nonzero elements, they are *sparse*. This happens to matrices emanating from network problems, linear programming bases, input output models in economy, and finite difference and finite element discretizations of partial differential equations.

There are quite good direct algorithms for linear systems that take sparsity into account, and some of them are implemented in MATLAB. We limit our discussion to symmetric positive definite matrices. Most of the common applications involve symmetric positive definite matrices, take e g stiffness matrices in mechanics and conductivity in electric DC networks. While factorizing positive definite matrices, it is not necessary to pivot for stability. Any elimination order is equally good, as long as symmetry is preserved.

2.1 Graphs and matrices

A *graph* G consistes of one set V of *vertices*, and one set E of *edges*, connecting pairs of vertices,

$$G = (V, E) \text{ where } V = \{1, 2, \ldots, n\}, \quad E \subset V \times V\,. \tag{2.1}$$

To each graph with n nodes, there corresponds an $n \times n$ matrix A, where $a_{i,j} \neq 0$ if and only if $(i, j) \in E$. We can think of an arrow from vertex i to vertex j. We let an *undirected* graph correspond to a symmetric matrix, in that case both of $a_{i,j}$ and $a_{j,i}$ are nonzero if one of them is. Many applied problems have a graph structure in bottom, think of networks and finite difference and -element approximations.

It the matrix A is given, we may construct a graph $G(A)$ where edges correspond to nonzero elements of the matrix.

We may renumber the nodes of a graph. This does not change its adjacency properties, so all renumberings of the same graph are in a certain respect equivalent. A renumbering of the graph corresponds to a simultaneous permutation of rows and columns of the matrix

$$A' = P^T AP \tag{2.2}$$

the multiplication from the left with P^T exchanges the rows, and the multiplication with P from the right exchanges columns of the matrix A.

2.2 Gaussian elimination of graphs

Now look at the linear system

$$Ax = b$$

where the matrix A is sparse. If we perform Gaussian elimination LU factorization, the factors L and U will in general be less sparse than the original matrix A, we say that we get *fill in* of nonzero elements. Let us see when this happens! In the innermost loop of Gaussian elimination, we compute

$$a'_{i,j} = a_{i,j} - (a_{i,k}/a_{k,k})a_{k,j} \tag{2.3}$$

on element $a_{i,j}$ in step k when variable k is eliminated. Fill in occurs whenever $a_{i,j} = 0$ (not yet filled) and both $a_{i,k}$ and $a_{k,j}$ are different form zero (filled). We assume that the pivot element $a_{k,k}$ is nonzero, this is always the case when A is symmetric positive definite.

Let us see what this means in graph terms. When we eliminate one node k we will have to introduce a fill in edge (i, j) if there is a path from node i to node j via node k.

We get a very intuitive picture of Gaussian elimination of a graph. Eliminate the nodes one by one. When one node k is eliminated, add new edges between any two nodes that are connected to node k and does not yet have an edge between them.

Now different variants of sparse Gaussian elimination correspond to different ways to number the nodes. There is no way to find an optimal ordering, without testing them all. We say that the problem of finding an order of a graph that has minimum fill in is *NP-complete*. We will have to use a reasonably effective heuristic, to find an order that we can use in practical cases.

2.3 Choice of algorithm

We will discuss three classes of heuristical reordering algorithms, *Variable Band*, *Nested Dissection* and *Minimum Degree*.

2.3.1 Variable band: RCM algorithm

Variable band algorithms are very easy to implement and they are still common in Structural Engineering under the name *skyline algorithm*. They follow from the observation that there is fill in in column j above the main diagonal only below the first filled element $a_{k,j}$ (2.3). Elements are filled only below the skyline of the top nonzero elements above the diagonal. A good variable band algorithm is one that makes a small skyline, and that happens when the band width of the matrix is small. This in turn happens if the graph is of an oblong shape and numbered from one end to the other.

There is an automatic way of finding a reasonably good variable band ordering, the reversed Cuthill Mc Kee algorithm (RCM). It has two phases, first it tries to find a pair of peripheral nodes, then it traverses the graph along the diameter between these hopefully peripheral nodes. Finally the ordering is reversed, that explains the peculiar name.

A *path* between two nodes i and j is a sequence of edges going from node i to node j, say $(i, i_1), (i_1, i_2), \ldots, (i_h, j)$ all belonging to the edge set E. There may be several paths between a certain pair of nodes, take one shortest path, then the *distance* between node i and node j is the number of edges in a shortest path connecting them. A *diameter* is a shortest path of maximal length, and we say that a node is *peripheral* if it is one end of a diameter.

One heuristic, that often finds a diameter, is the Gibbs Poole Stockmeyer algorithm. It builds up a *level structure* in the graph in the following way. Take one node, let it be level 0. Number its neighbors, i. e. nodes that have an edge to it. This is level 1. When we have levels $0, \ldots, k$, number all neighbors to nodes from level k, that are not yet numbered, to make up level $k + 1$. Finally all nodes are numbered (provided that the graph is connected). Those at the last level h are at a distance h from the starting node.

Now repeat the same procedure, starting at one node in the last level h. We get a new level structure that has at least h levels (Why?). Start again at the new last level and repeat the procedure until the number of levels no longer increases. This happens most often rather soon. Now the first and last nodes of the final numbering are likely to be peripheral and their shortest path be a diameter. We call them *pseudo-peripheral* to avoid claiming too much.

The Cuthill Mc Kee ordering is now taken from one of these last longest level structures. Number the nodes level by level, enumerating neighbors inside the levels. The matrix will get a cigar like shape, open in the end. Reverse the ordering, and get it open in the beginning.

2.3.2 Nested Dissection

This algorithm, also called *substructuring*, is built around finding a separating set of nodes dividing the graph into two parts, substructures. Eliminate the nodes inside each of the substructures first and take the nodes of the separating

set last. It is best illustrated by the block matrix,

$$A = \begin{bmatrix} A_{1,1} & 0 & A_{1,3} \\ 0 & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \tag{2.4}$$

Here blocks 1 and 2 are the substructures and 3 the separating set of nodes. When Gaussian elimination is performed on A, the zero blocks will remain zero.

We can repeat the division of the subgraphs recursively, and get a kind of self similar look of the matrices.

2.3.3 Minimum Degree

This is a greedy algorithm, to use a term from Computer Science. We understood from the discussion about Gauss elimination of graphs, that the largest possible fill in, when eliminating node k first is $(r_k - 1) \times (c_k - 1)$, the product of the number of nondiagonal nonzeros in row k and the number of nonzeros in column k. For a symmetric matrix this is the square of the number of edges meeting node k, this number is called the *degree* of node k. Eliminating one node with minimum degree, will give the smallest possible fill in after the first elimination step. After eliminating the first node, we have to update the graph with the fill in, and take a minimum degree node of the updated graph. Continue until all nodes are eliminated!

The surprising fact is that this very simple idea of a greedy algorithm most often is very good. It also gives rise to orderings that bear a resemblance to the much more elaborate nested dissection strategy.

2.4 Examples

Let us illustrate these algorithms on a very simple example, a finite difference Laplacian on a square grid. For a unit square and step $h = 0.25$ we get a 3×3 grid

```
G= [ 1     4     7
     2     5     8
     3     6     9]
```

with $n = 9$ nodes ordered in columns. The matrix is

```
A= [ 4    -1     0    -1     0     0     0     0     0
    -1     4    -1     0    -1     0     0     0     0
     0    -1     4     0     0    -1     0     0     0
    -1     0     0     4    -1     0    -1     0     0
     0    -1     0    -1     4    -1     0    -1     0
     0     0    -1     0    -1     4     0     0    -1
     0     0     0    -1     0     0     4    -1     0
     0     0     0     0    -1     0    -1     4    -1
     0     0     0     0     0    -1     0    -1     4]
```

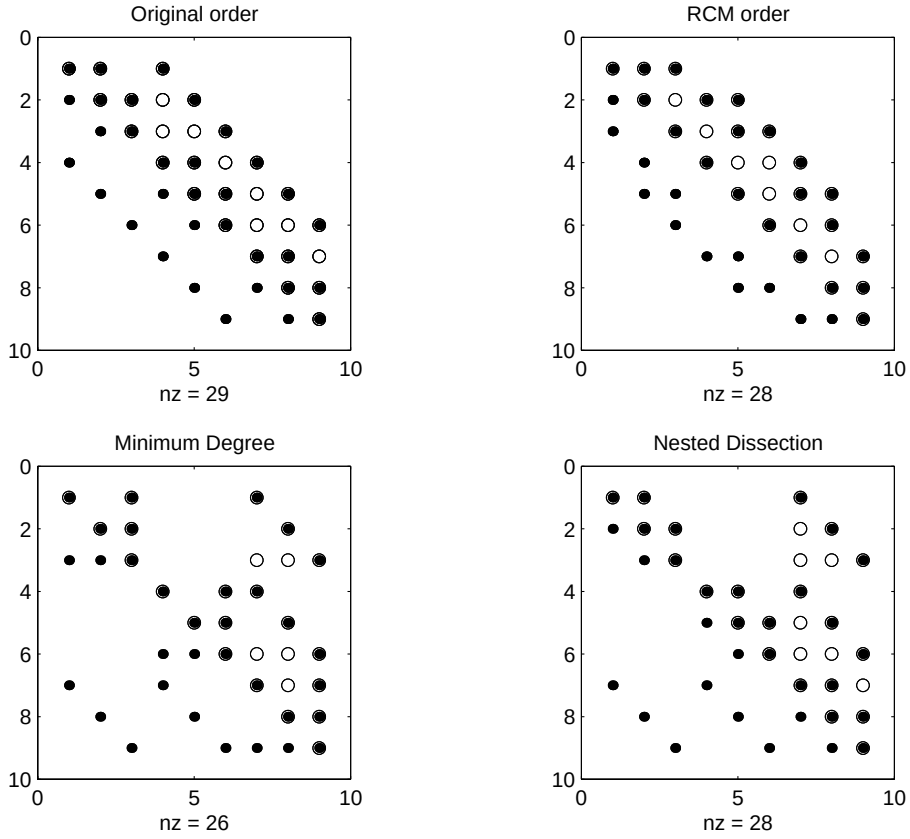

Figure 2.1: Laplace $n = 9$. Dots stand for original elements, circles fill in above the main diagonal at different orderings.

We plot the nonzero pattern as points in the upper left of figure 2.1. We do Cholesky (Gaussian elimination) and mark the filled elements of the uppertriangular factor as circles. In the first elimination step $k = 1$ there is fill in at position $2, 4$, see (2.3). In the next step $k = 2$ this recently filled element causes a fill in in position $3, 4$ while the original element at $2, 5$ causes a fill in $3, 5$. We show plots of RCM, MMD and ND reorderings in the rest of figure 2.1

This very small matrix is included mainly to make it possible to follow what happens. Take a slightly larger, $n = 25$! Here the RCM reordering of the grid is

```
G= [ 1     2     4     7    11
     3     5     8    12    16
     6     9    13    17    20
    10    14    18    21    23
    15    19    22    24    25 ]
```

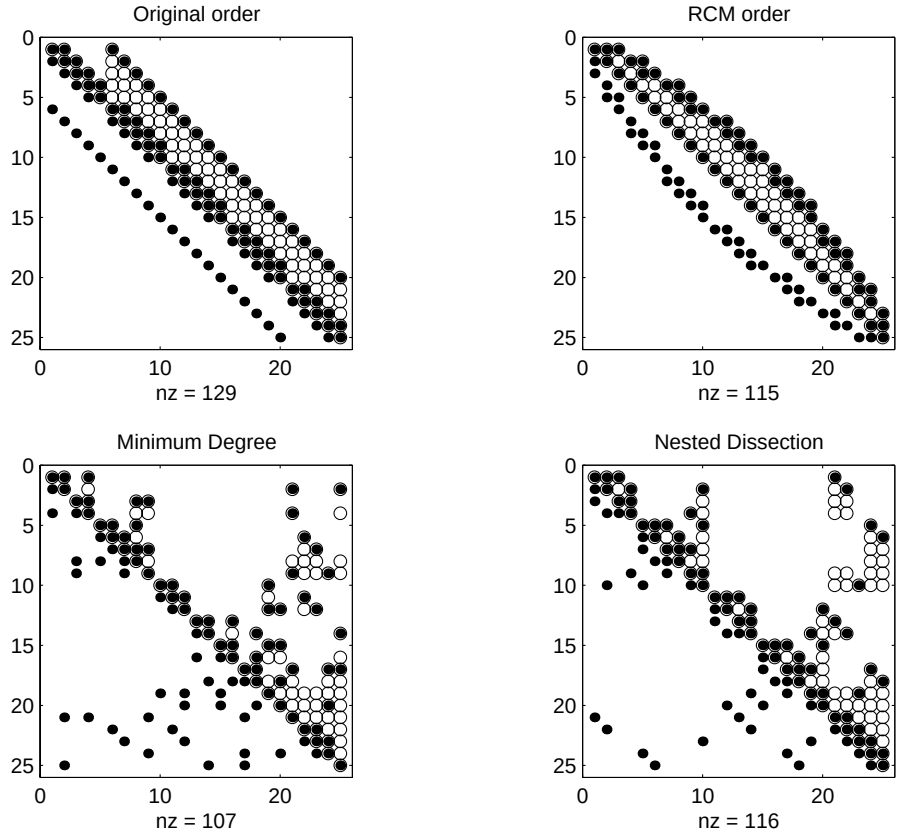

Figure 2.2: Laplace $n = 25$, fill in at different orderings given by MATLAB.

and we see in the upper right picture in figure 2.2, the characteristic cigar shape of the RCM reordering of the matrix.

In the bottom plots in figure 2.2, we see how the similarity between the MMD and ND orderings starts to show up. That is slightly surprising, Minimum Degree is a greedy, bottom up type of algorithm, while Nested Dissection is a recursive top down.

ND does not come out at its best compared to RCM, a refined version of ND gives the reordering

```
G= [ 1     3    21    13    11
     2     4    22    14    12
     9    10    23    20    19
     6     8    24    18    16
     5     7    25    17    15 ]
```

Here we see how the corners are taken first, then the separators between two corner blocks and finally the large separator in the middle. Look at the plot in

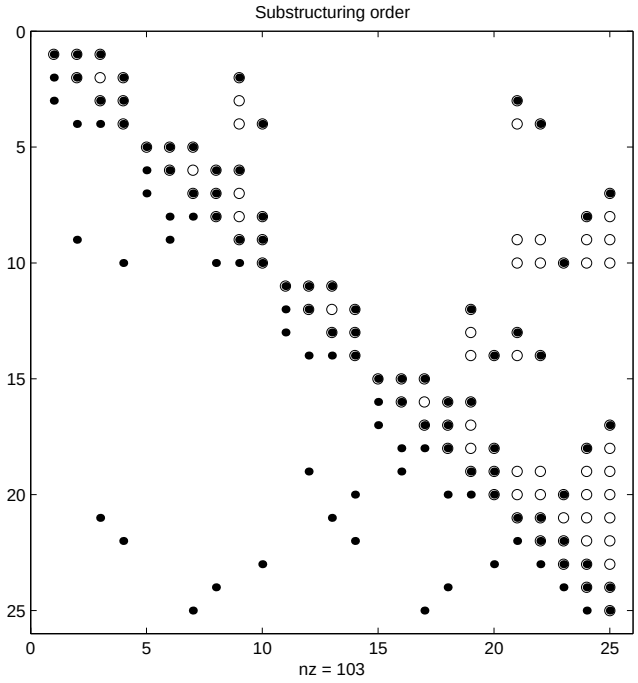

Figure 2.3: Laplace $n = 25$, optimal nested dissection.

figure 2.3! First comes a 4×4 block, then another 4×4 and then the 2 node separator. Then the same pattern is repeated and last come the 5 nodes in the second order separator. The fill in is postponed until the latest possible columns, and we see that the bottom 7×7 block is nearly full after the factorization is finished.

In practical codes for large matrices, one switches to full matrix code at that stage when a substantial part of the elements are filled. A full matrix code will then be faster, the sparse code needs quite a bit of bookkeeping and indirect addressing.

Let us take a still larger example, a 31×31 grid that gives $n = 969$. We plot the matrix and its Cholesky factor after MMD reorder in figure 2.4. The Cholesky factor now has 11023 filled elements, there is a fill in of 8198 new elements above the diagonal. This is still quite a bit better than the original ordering for which the Cholesky factor has 29821 filled elements, the RCM ordering where it has 21266, and ND where it has 15032.

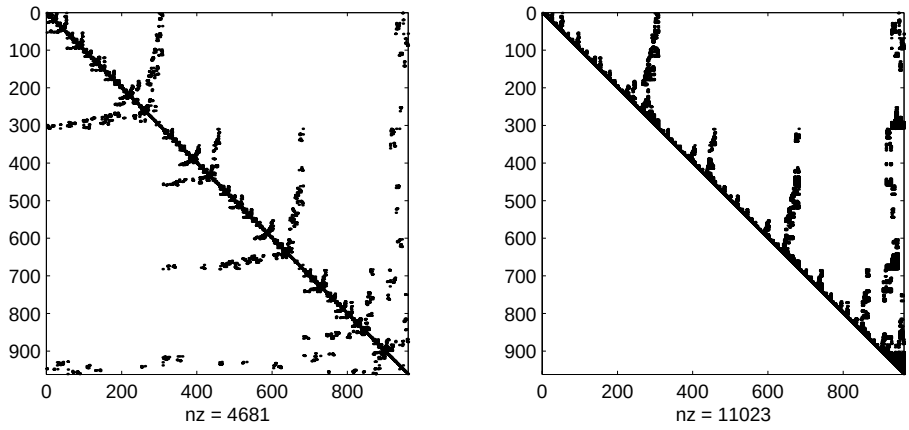

Figure 2.4: Laplace $n = 969$, Minimum Degree. Left before factorization, right upper triangular factor. Fill in upper triangle is $8198 = 11023 - (4681 + 969)/2$ elements.

Chapter 3

The Singular Value Decomposition

3.1 Overdetermined systems, least squares

One of the most common computations in practice is fitting a mathematical model to a set of observations. Let us say that we observe the activity of a sample that is undergoing radioactive decay. We have measured a series of n intensities $(t_i, y_i), i = 1, \ldots, n$. The intensity is modeled by

$$y = \sum_{j=1}^{p} \alpha_j \exp(-\lambda_j t)$$

where, $\alpha_j \geq 0$, is the amount and $\lambda_j \geq 0$ is the decay rate of element j. Inserting our observations in the model we need to minimize the norm of the residual

$$r = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} - \begin{bmatrix} \exp(-\lambda_1 t_1) & \ldots & \exp(-\lambda_p t_1) \\ \exp(-\lambda_1 t_2) & \ldots & \exp(-\lambda_p t_2) \\ \vdots & & \vdots \\ \exp(-\lambda_1 t_n) & \ldots & \exp(-\lambda_p t_n) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_p \end{bmatrix}$$

$$r = y - A(\lambda)x$$

The residual vector r is the difference between the *observation* vector y and the product of the *design* matrix A and the *parameter* vector x. Most often n ,the number of observations, is considerably larger than p, the number of elements in the model, we have an *overdetermined* system. The masses α_j occur linearly in the model, while the decay rates λ_j occur nonlinearly. In the simplest case, the decay rates are known, we have to determine only the linear parameters α_j.

Our task is now to compute parameters α_j and λ_j so that the residual r is minimized in an appropriate norm. The choice of norm depends on the situation. If the errors in the model are caused by many independent sources,

we choose the Euclidean norm, $\|r\|_2 = (r^T r)^{1/2} = (\sum r_i^2)^{1/2}$. This is the most common situation in experimental or observational science, it is also the case that is simplest to handle numerically, we talk about a *least squares* problem.

In some cases other norms are used. If we want to minimize the maximal residual, we use the maximum norm, $\|r\|_\infty = \max_i |r_i|$. This is the situation when we need to approximate an elementary function by a polynomial or rational function in a computer. There are also cases when the sum norm, $\|r\|_1 = \sum |r_i|$, is of interest, this happens when there are outliers in the set of observations, i. e. that some of the observations are far away from the model.

Let us now consider a linear least squares problem

$$\min_x \|r\|_2 = \|y - Ax\|_2$$

We see that we should choose the p vector x so that the distance between the image Ax and the observation vector y is as small as possible. This happens when the difference $r = y - Ax$, the residual, is perpendicular to the range space $R(A)$ of all possible Ax for the given matrix A. Especially the residual r should be orthogonal to each of the columns of the matrix A, that is

$$A^T r = A^T(y - Ax) = A^T y - A^T Ax = 0$$

which is a $p \times p$ system with a symmetric matrix $A^T A$ that x should satisfy. These are the *normal equations*.

If the design matrix A has linearly independent columns, the normal equation matrix $A^T A$ is nonsingular, and one can compute a solution to the linear least squares problem by solving the normal equations.

The normal equations do not give the most accurate method to solve a linear least squares problem, we get better algorithms by observing that we may transform the matrix A itself by *orthogonal* transformations. Replacing the matrix A by $Q^T A$ does not change the norm $\|r\|_2$ if

$$\|Q^T r\|_2 = ((Q^T r)^T Q^T r)^{1/2} = (r^T Q Q^T r)^{1/2} = \|r\|_2$$

which happens if $QQ^T = I$, that is if the transformation Q is orthogonal.

We may compute an orthogonal matrix Q that transforms the rectangular matrix A into upper triangular form R as a product of p elementary reflections $Q = H_1 H_2 \dots H_p$ each H_k putting zeros into column k of A.

Let us not discuss this algorithm in detail yet, we will get an even more powerful algorithm by using the singular value decomposition SVD.

3.2 The SVD theorem

Theorem 1 *Any $m \times n$ matrix A can be factorized*

$$A = U\Sigma V^T = U_r \Sigma_{rr} V_r^T \tag{3.1}$$

where

$$U = \begin{bmatrix} U_r, U_{m-r} \end{bmatrix} \text{ is } m \times m \text{ orthogonal(unitary), left singular vectors}$$
$$V = \begin{bmatrix} V_r, V_{n-r} \end{bmatrix} \text{ is } n \times n \text{ orthogonal(unitary), right singular vectors}$$
$$\Sigma = \begin{bmatrix} \Sigma_{rr} & 0 \\ 0 & 0 \end{bmatrix} \text{ is } m \times n \text{ with}$$
$$\Sigma_{rr} = \mathrm{diag}(\sigma_k),\ r \times r \text{ diagonal with } \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0, \text{ singular values}$$

Proof: We prove the theorem by finding the first singular value and its pair of left and right singular vectors and reduce the matrix to one with one row and column less. The same procedure is then repeated.

Take a unit length vector x for which the maximum in the definition of the Euclidean matrix norm is attained,

$$\|A\|_2 = \max \frac{\|Ax\|_2}{\|x\|_2} \tag{3.2}$$

Set $\sigma y = Ax$, where $\sigma = \|Ax\|_2 = \|A\|_2$ is a normalization factor chosen to make y a unit length vector. We can now form two orthogonal matrices, $V_1 = [x, X_{n-1}]$ and $U_1 = [y, Y_{m-1}]$, by adding orthogonal columns to the vectors x and y.

Multiply the matrix A with these and get,

$$\begin{aligned} U_1^T A V_1 &= U_1^T \begin{bmatrix} Ax & AX_{n-1} \end{bmatrix} \\ &= U_1^T \begin{bmatrix} \sigma y & AX_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} y^T \\ Y_{m-1}^T \end{bmatrix} \begin{bmatrix} \sigma y & AX_{n-1} \end{bmatrix} = \begin{bmatrix} \sigma & w^T \\ 0 & B \end{bmatrix} = A_1 \end{aligned} \tag{3.3}$$

The zeros in the first column of A_1 are a consequence of that U_1 is chosen to be orthogonal. Its first column is then orthogonal to the remaining columns $Y_{m-1}^T y = 0$.

Now let us prove that also the row vector w^T in the upper right hand corner of A_1 is zero. We know that the orthogonal transformation (3.3) does not change the norm of A, $\|A_1\|_2 = \|A\|_2 = \sigma$. Let A_1 operate on the vector $z = (\sigma,\ w^T)^T$ and note that

$$A_1 z = \begin{bmatrix} \sigma & w^T \\ 0 & B \end{bmatrix} \begin{bmatrix} \sigma \\ w \end{bmatrix} = \begin{bmatrix} \sigma^2 + w^T w \\ Bw \end{bmatrix}$$

which means that

$$\sigma = \|A\|_2 = \|A_1\|_2 \geq \frac{\|A_1 z\|_2}{\|z\|_2} \geq \frac{((\sigma^2 + w^T w)^2 + \|Bw\|_2^2)^{1/2}}{(\sigma^2 + w^T w)^{1/2}} \geq (\sigma^2 + w^T w)^{1/2} \geq \sigma\,.$$

This means that all these inequalities are equalities, which can be true only if $w = 0$.

We have now proved that

$$U_1^T A V_1 = \begin{bmatrix} \sigma & 0 \\ 0 & B \end{bmatrix}$$

and can continue to apply the same reduction on the lower right block B getting

$$U_2^T B V_2 = \begin{bmatrix} \sigma_2 & 0 \\ 0 & C \end{bmatrix}$$

or after multiplying together

$$\begin{bmatrix} 1 & 0 \\ 0 & U_2 \end{bmatrix}^T U_1^T A V_1 \begin{bmatrix} 1 & 0 \\ 0 & V_2 \end{bmatrix} = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & C \end{bmatrix} .$$

We have now written σ_1 for the number σ used previously. Note that $\sigma_1 \geq \sigma_2$ because σ was chosen as the maximum of $\|Ax\|$ over all unit vectors (3.2), while σ_2 is the maximum over vectors restricted to the range of X_{m-1} of vectors orthogonal to the original x.

We may repeat the same procedure until, after r reductions, we finally get a zero bottom right block and have

$$U^T A V = \begin{bmatrix} \sigma_1 & 0 & \dots & \dots & 0 \\ 0 & \sigma_2 & \dots & \dots & 0 \\ \vdots & & & \ddots & \\ 0 & & & \sigma_r & 0 \\ 0 & 0 & & \ddots & 0 \end{bmatrix}$$

where the orthogonal matrix U is the product $U_1 U_2 \dots U_r$ with appropriate rows and columns from the unit matrices added in the beginning of each factor U_k, $k = 2, \dots, r$. Likewize V is $V_1 V_2 \dots V_r$. *This ends the proof.*

This proof is not actually constructive. It depends on the existence of a maximizing vector x for the quotient $\|Ax\|_2 / \|x\|_2$, (3.2).

The traditional way to prove the SVD is to use the relation between eigenvalues and singular values. Then the singular values are the square roots of the eigenvalues of the symmetric and positive semidefinite matrix $A^T A$. The right singular vectors V are the eigenvectors of $A^T A$. The left singular vectors U are the eigenvectors of AA^T which has the same nonzero eigenvalues as $A^T A$.

Note that we have made no assumptions on the number of rows m and columns n . We use to talk about *overdetermined* systems when $m > n$ and *underdetermined* if $m < n$.

The number $r \leq \min(m, n)$ is called the *rank* of the matrix and is a very important number. It gives the dimensions of the *range space* of A,

$$R(A) = \{y = Ax, x \in R^n\} \subset R^m$$

as well as the range space of A^T

$$R(A^T) = \{x = A^T y, y \in R^m\} \subset R^n .$$

The leading r columns U_r of U give an orthonormal basis of $R(A)$ and those V_r of V an orthonormal basis of $R(A^T)$ the range of its transpose. The remaining columns V_{n-r} of V form a basis of the *nullspace* of A

$$N(A) = \{x \in R^n, \quad Ax = 0\} \subset R^n$$

and those of U, U_{m-r} form a basis of the nullspace of A^T

$$N(A^T) = \{y \in R^m, \quad A^T y = 0\} \subset R^m \,.$$

These are the four fundamental subspaces of the linear mapping given by the matrix A,

$$R^n = R(A^T) \oplus N(A) = \text{span}(V_r) \oplus \text{span}(V_{n-r})$$

$$R^m = R(A) \oplus N(A^T) = \text{span}(U_r) \oplus \text{span}(U_{m-r})$$

If $r = n$, the nullspace of A is zero and A has linearly independent columns, we say that A has *full column rank*. If $r = m$, the nullspace of A^T is zero and now A has linearly independent rows, *full row rank*. A *nonsingular* matrix is a square matrix with full rank in both directions, $r = m = n$. If the rank r is smaller $r < \min(m, n)$ we say that the matrix is *rank deficient*. A *singular* matrix is a square matrix that is rank deficient, $r < m = n$.

3.3 Using SVD to solve linear least squares problems

With the singular value decomposition available, it is an easy matter to find solutions to any linear least squares problem,

$$\min_x \|Ax - b\|_2 \,.$$

Use the SVD (3.1) and the orthogonal invariance of the Euclidean vector norm to observe that,

$$\|Ax - b\|_2 = \|U\Sigma V^T x - b\|_2 = \|\Sigma(V^T x) - U^T b\|_2$$

that gives the algorithm

Algorithm Solve Least Squares Problem

1. Compute the SVD of the matrix, $A = U\Sigma V^T$.
2. Multiply the right hand side $U^T b = \begin{bmatrix} b'_r \\ b'_{m-r} \end{bmatrix}$.
3. Divide $y = (V_r^T x) = \Sigma_{rr}^{-1} b'_r$.

4. Multiply $x = V_r y$.

END.

We note that the residual is

$$r = b - Ax = U_{m-r} b'_{m-r} \in N(A)$$

and is of minimal norm $\|r\| = \|b'_{m-r}\|$ which can be computed from the bottom part of the transformed right hand side $b' = U^T b$. There is no way of making the residual smaller. If the columns are linearly dependent, $r < n$ there are $n - r$ last elements in the transformed solution vector $y = Vx$ that can be chosen arbitrarily without changing the norm of the residual. Our choice is to take them as zero, it gives the solution of minimum norm.

We notice that the solution x is a linear function of the right hand side b and we can write it as

$$x = A^+ b = V_r \Sigma_{rr}^{-1} U_r^T b$$

We call the matrix A^+ the *pseudoinverse* of A. It is the inverse of A regarded as a one to one mapping from $R(A^T)$ to $R(A)$, the two r-dimensional range spaces spanned by V_r and U_r.

3.4 Finding low rank approximations

We may write the SVD (3.1) as a sum of rank one matrices,

$$\begin{aligned} A =& U_r \Sigma_{rr} V_r^T \\ =& \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_r u_r v_r^T \\ =& \sigma_1 E_1 + \sigma_2 E_2 + \cdots + \sigma_r E_r \end{aligned}$$

The interesting thing is now that the sum of the leading $k \leq r$ terms gives the best rank k approximation to A measured in the Euclidean, or more properly Frobenius, norm.

We call this *Principal component* analysis. The first principal component $\sigma_1 E_1$ contains the main variation of the data material collected in the matrix A. This property of the SVD is widely used in signal processing, data analysis and psychometrics.

3.5 Computing the SVD

Algorithms to compute the SVD make use of the close relation between the SVD and the symmetric eigenvalue problems of the matrices $A^T A$ and AA^T. These matrices are not formed explicitly, instead one multiplies the original matrix A by different orthogonal matrices from left and right, which corresponds to orthogonal similarities on AA^T and $A^T A$ respectively.

Chapter 4

Eigenvalues

In many scientific and engineering contexts one needs to solve an algebraic eigenvalue problem,

$$(A - \lambda B)x = 0\,, \tag{4.1}$$

for eigenvalues λ_k and eigenvectors x_k. Its solution is used as the first step on the way from a static to a dynamic description of a system of interacting entities. Think of a network of electric components or a mechanical construction of masses and springs! Small perturbations of a static, equilibrium, position will be formed as eigenvectors and move as the eigenvalues indicate.

The foremost mathematical modeling tool is the Laplace transform, where we map a description in time and space to one involving decay rates and oscillation frequences, these are the real and imaginary parts of the eigenvalues. In this realm we use eigenvalues to get the solution of a linear system of ordinary differential equations,

$$\dot{x} = Ax\,,\ x(0) = a\,,$$

over time as a linear combination of eigensolutions,

$$x(t) = \Sigma_k \alpha_k x_k e^{\lambda_k t}\,.$$

Complex eigenvalues correspond to oscillating solutions,

$$\exp(a + ib)t = \exp(at)(\cos(bt) + i\sin(bt))\,.$$

4.1 Transformation algorithms

For matrices of moderate size, essentially those that can be stored as a full array in the computer, the standard way to compute eigenvalues is to apply similarity transformations, until we reach a form of the matrix where it is easy to read off the eigenvalues. If there are n linearly independent eigenvectors, $x_1, x_2, \ldots, x_n$, building up the nonsingular matrix X, then

$$AX = XD, \text{ giving } A = XDX^{-1}, \text{ where } D = \text{diag}(\lambda_k) \tag{4.2}$$

and in this case we say that A is *diagonalizable*.

4.1.1 Theory

Not all matrices are diagonalizable, but we can transform any square matrix to *triangular* form by means of an unitary (or orthogonal) similarity. This is the consequence of the Schur theorem,

Theorem 2 *(Schur theorem) Every square matrix A can be transformed*

$$A = UTU^H$$

where U is unitary and T is upper triangular with eigenvalues in the diagonal in any chosen order.

Proof: The proof is very similar to the proof of the SVD theorem 1. We find an eigenvale eigenvector pair, transform the first row and column to the form wanted, and repeat the procedure on a matrix of smaller size.

Take λ_1 an eigenvalue and x an eigenvector of unit norm, $\|x\|_2 = 1$. Add $n-1$ column vectors X_{n-1} to make a unitary matrix, $U_1 = [x, X_{n-1}]$.

Now

$$\begin{aligned} Ax &= x\lambda_1 \\ AU_1 &= [x\lambda_1, AX_{n-1}] \\ U_1^H AU_1 &= \begin{bmatrix} \lambda_1 & A'_{1,n-1} \\ 0 & A'_{n-1,n-1} \end{bmatrix} \end{aligned}$$

The zeros in the lower left are there because U is unitary, its first column x is orthogonal to the remaining columns in X_{n-1}.

We have now made the first step, repeat the same procedure on the lower right block $A'_{n-1,n-1}$ giving λ_2 and U_2, and continue down to the right, until we have an upper triangular matrix T and a unitary $U = U_1 U_2 \dots U_{n-1}$, with parts of the unit matrix added to each U_k, to make it $n \times n$. *This ends the proof.*

Note that, unlike Theorem 1 there is no specified order between the eigenvalues. Any of the eigenvalues can be chosen as λ_1, and algorithms for eigenvalue computation may give their results in surprising orders.

There is no way to transform the matrix into triangular form by a finite number of elementary transformations. Any practical transformation algorithm is divided into two phases, one initial reduction, that zeroes out all subdiagonal elements except one diagonal, and one final iterative phase where the remaining subdiagonal elements are made smaller and smaller in magnitude.

4.1.2 Initial reduction: Householder algorithm

We use a finite systematic algorithm to make an orthogonal (unitary in the complex case) similarity transformation into Hessenberg form. A Hessenberg matrix is upper triangular with one nonzero subdiagonal. If the original matrix A is Hermitian (real symmetric), the Hessenberg matrix H will be symmetric and tridiagonal.

In the real nonsymmetric case we get the factorization,

$$A = WHW^T$$

where the result H is of Hessenberg form and the orthogonal transformation matrix W is a product $W = H_1 H_2 \dots H_{n-2}$ of Householder transformations or *elementary reflections*.

Elementary reflections: An elementary reflection is a matrix $H = I - 2uu^T$, where the vector u is of unit norm $\|u\|_2 = 1$. An elementary reflection is both orthogonal and symmetric. To check symmetry, note that,

$$(I - 2uu^T)^T = I^T - 2(u^T)^T u^T = I - 2uu^T \,,$$

using the facts that $I^T = I$, $(AB)^T = B^T A^T$ and $(A^T)^T = A$ for all pairs of matrices A and B of the appropriate orders. To verify orthogonality, multiply

$$(I - 2uu^T)(I - 2uu^T) = I - 2uu^T - 2uu^T + 4uu^T uu^T = I \,,$$

since $uu^T uu^T = u(u^T u)u^T = uu^T$ when the vector u is of unit Euclidean length, $u^T u = 1$.

Progress of transformations: We now choose $n-2$ elementary reflections $H_k, k = 1, \dots, n-2$, where H_k makes all elements except the $k+1$ first elements in column k of A equal to zero. The vector u_k in H_k is zero in the leading k positions.

We start with the full matrix

$$A^{(1)} = \begin{bmatrix} x & x & x & \dots & x \\ x & x & x & \dots & x \\ x & x & x & \dots & x \\ \vdots & \vdots & \vdots & & \vdots \\ x & x & x & \dots & x \end{bmatrix}$$

Multiply by H_1 from the left and get $A^{(1.5)} = H_1 A^{(1)}$, where the first column has zeros below position 2,

$$A^{(1.5)} = \begin{bmatrix} x & x & x & \dots & x \\ r & y & y & \dots & y \\ 0 & y & y & \dots & y \\ 0 & \vdots & \vdots & & \vdots \\ 0 & y & y & \dots & y \end{bmatrix}$$

We note that the first row is unchanged. Complete the similarity transformation by multiplying from the right, giving $A^{(2)} = A^{(1.5)} \times H_1 = H_1 A^{(1)} H_1$, now with

the first column untouched

$$A^{(2)} = \begin{bmatrix} x & z & z & \dots & z \\ r & z & z & \dots & z \\ 0 & z & z & \dots & z \\ 0 & \vdots & \vdots & & \vdots \\ 0 & z & z & \dots & z \end{bmatrix}$$

This was the transformation with the first Householder matrix, H_1. The next H_2 does not change the first two rows when multiplied from the left. This means that the zeros introduced during the first transformation will remain zero, and the vector u_2 can be chosen to annihilate all elements in the second column except the 3 first,

$$A^{(2.5)} = \begin{bmatrix} x & z & z & \dots & z \\ r & z & z & \dots & z \\ 0 & r & w & \dots & w \\ 0 & \vdots & \vdots & & \vdots \\ 0 & 0 & w & \dots & w \end{bmatrix}$$

The transformation from the right

$$A^{(3)} = A^{(2.5)}H_2 = H_2A^{(2)}H_2 = H_2H_1A^{(1)}H_1H_2$$

does not destroy any of the zeros in the first two columns. Continue with the transformations H_3 etc until H_{n-2} gives the product $W = H_1H_2 \dots H_{n-2}$. The matrix A will have the form,

$$A^{(n-1)} = \begin{bmatrix} x & z & v & \dots & v \\ r & z & v & \dots & v \\ 0 & r & v & \dots & v \\ & & \ddots & & \vdots \\ 0 & & & r & v \end{bmatrix}$$

.

A numerical example Let us look at an example. Take the matrix,

$$A = magic(4) = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{bmatrix}$$

It is of order $n = 4$, and we perform $n - 2 = 2$ reflections on it.

The first reflection is determined by the last 3 elements of the first column. It is

$$H_1 = I - 2u_1u_1^T = \begin{bmatrix} 1 & & & \\ & -0.45268 & -0.81482 & -0.36214 \\ & -0.81482 & 0.54296 & -0.20313 \\ & -0.36214 & -0.20313 & 0.90972 \end{bmatrix}$$

Note that it is both symmetric and orthogonal and equal to the unit matrix in the first row and column. We multiply A from the left and get

$$A^{(1.5)} = H_1 A = \begin{bmatrix} 16 & 2 & 3 & 13 \\ -11.0454 & -15.7532 & -14.8479 & -13.7614 \\ & -8.0061 & -7.9374 & -0.2062 \\ & 7.3306 & 8.8056 & -4.425 \end{bmatrix}$$

The first row is unchanged, the norms of the columns are also left invariant. Then multiply from the right and get,

$$A^{(2)} = H_1 A H_1 = \begin{bmatrix} 16 & -8.0577 & -2.6415 & 10.4927 \\ -11.0454 & 24.2131 & 7.5696 & -3.7981 \\ & 10.1665 & 2.2558 & 4.3241 \\ & -8.8909 & -0.29322 & -8.4689 \end{bmatrix}$$

Now the three last columns are affected and the row norms are left invariant.

This concludes the first orthogonal similarity. Apply the next reflection H_2 from left and right and get,

$$A^{(3)} = W^T A W = \begin{bmatrix} 16 & -8.0577 & 8.8958 & 6.1595 \\ -11.0454 & 24.2131 & -8.1984 & 2.1241 \\ & -13.5058 & -4.3894 & -7.8918 \\ & & -3.2744 & -1.8237 \end{bmatrix}$$

which is of Hessenberg form.

The final transformation matrix is,

$$W = H_1 H_2 = \begin{bmatrix} 1 & & & \\ & -0.45268 & 0.37496 & -0.80901 \\ & -0.81482 & -0.54243 & 0.20453 \\ & -0.36214 & 0.75178 & 0.55107 \end{bmatrix}$$

It is orthogonal but not symmetric.

Complexity In this illustration, we have written the transformation matrices H_1 and H_2 as full matrices. In any serious implementation, one takes advantage of that H_i is an elementary transformation matrix, that is a rank 1 modification of the unit matrix, and computes each column $H_1 a$ as

$$H_1 a = (I - 2uu^T)a = A - 2uu^T a = a - u(2u^T a)$$

first evaluating the scalar product in parentheses $u^T a$, this needs n additions and multiplications. Then multiply by 2, this is one multiplication, and finally subtract a multiple of the u vector from the a vector, this is what in linear algebra slang is called an axpy operation, needing n additions and multiplications. In all we need $2n$ flops, if we count one addition plus one multiplication as one flop, floating point operation. This is considerably less than the n^2 flops needed to multiply a full matrix with a vector.

Also in the matrix matrix operation $A' = H_i A$ one saves a factor n in operation count, it takes $2n^2$ flops. The complete reduction needs $\mathcal{O}(n^3)$ flops, which is of the same order of magnitude as the Gaussian elimination factorization for solving a linear system or as a matrix matrix multiply, $C = AB$.

4.1.3 Final iteration: QR algorithm

The first phase of the transformation algorithm was finite, and needed $n-2$ steps to get the matrix into Hessenberg form. Now in the second phase, the Hessenberg form will be retained, while we will make the subdiagonal elements smaller and smaller. The standard algorithm is the QR algorithm, which does the similarities by doing a sequence of QR factorizations of shifted matrices.

The basic QR algorithm is

ALGORITHM QR, UNSHIFTED

Start $A_1 = H$ (Hessenberg matrix), $U_1 = W$ (transformation).
For $k = 1, \ldots,$ do

1. Factorize $A_k = Q_k R_k$ giving Q_k, orthogonal, and R_k, upper triangular.
2. Multiply $A_{k+1} = R_k Q_k$
3. Accumulate $U_{k+1} = U_k Q_k$

END.

We see that

$$A_{k+1} = R_k Q_k = Q_k^T A_k Q_k$$

an orthogonal similarity. The accumulation makes the final

$$A_{k+1} = U_k^T A U_k$$

where A is the original matrix that we started the whole process on.

This simple QR algorithm with no shifts will converge, but very slowly. We will get a faster convergence if we introduce *shifts*. This depends on the observation that if A_k is singular, one diagonal element in R_k will be zero. The determinant of R_k is namely the product of its diagonal elements. Most often it is the last element $r_{nn} = 0$. Then the whole last row of R_k is zero, and when we multiply from the right with Q_k in the second step, the entire last row of A_{k+1} will be zero. We then know that we have a zero eigenvalue and that the rest of the eigenvalues are eigenvalues of the leading $(n-1) \times (n-1)$ submatrix. Then we can continue to compute eigenvalues of this smaller matrix, this is called *deflation*.

In practice we do not know any eigenvalue, we have to choose an approximation as shift. In LAPACK and other mathematical software, one takes an eigenvalue of the last 2×2 submatrix, this *Wilkinson* shift is easy to compute, and has the advantage that one may get complex shifts for a real matrix. A nonsymmetric real matrix most often has complex eigenvalues.

Let us formulate a simple variant of the shifted algorithm.

ALGORITHM QR,WITH EXPLICIT SHIFTS

Start $A_1 = H$ (Hessenberg matrix), $U_1 = W$ (transformation).
For $k = 1, \ldots,$ do

1. Choose shift σ_k
2. Factorize $A_k - \sigma_k I = Q_k R_k$ (Shifted matrix)
3. Multiply $A_{k+1} = R_k Q_k + \sigma_k I$ (Restore shift)
4. Accumulate $U_{k+1} = U_k Q_k$

END.

The new A_{k+1} is still an orthogonal similarity of A_k

$$A_{k+1} = R_k Q_k + \sigma_k I = (Q_k^T (A_k - \sigma_k I)) Q_k + \sigma_k I = Q_k^T A_k Q_k$$

Numerical example, continued We will continue with the 4×4 example and use the simple *Newton* shift. The Newton shift is the last diagonal element, eigenvalue of the last 1×1 matrix.

We start the QR algorithm on the Hessenberg matrix,

$$A_1 = \begin{bmatrix} 16 & -8.0577 & 8.8958 & 6.1595 \\ -11.0454 & 24.2131 & -8.1984 & 2.1241 \\ & -13.5058 & -4.3894 & -7.8918 \\ & & -3.2744 & -1.8237 \end{bmatrix}$$

The Newton shift $\sigma_1 = a_{4,4}^{(1)} = -1.8237$ which gives,

$$A_1 - \sigma_1 I = \begin{bmatrix} 17.8237 & -8.0577 & 8.8958 & 6.1595 \\ -11.0454 & 26.0368 & -8.1984 & 2.1241 \\ & -13.5058 & -2.5657 & -7.8918 \\ & & -3.2744 & \end{bmatrix}$$

$$= \begin{bmatrix} -0.85002 & -0.42038 & -0.22937 & -0.2194 \\ 0.52676 & -0.67837 & -0.37013 & -0.35405 \\ & 0.60257 & -0.57671 & -0.55165 \\ & & -0.69123 & 0.72263 \end{bmatrix} \begin{bmatrix} -20.9687 & 20.5642 & -11.8801 & -4.1168 \\ & -22.4134 & 0.27583 & -8.7857 \\ & & 4.7371 & 2.3522 \\ & & & 2.25 \end{bmatrix}$$

Note that the orthogonal factor Q is of Hessenberg form, it is actually computed as a product of $n - 1$ elementary rotations. The upper triangular R factor is already quite heavy at the top, even if the bottom element is not yet really small.

Multiply and add back the shift and get,

$$A_2 = \begin{bmatrix} 26.8323 & -12.2938 & 6.8951 & 0.89857 \\ -11.8064 & 13.547 & 14.2098 & 1.4345 \\ & 2.8545 & -6.1815 & -0.91342 \\ & & -1.5553 & -0.19778 \end{bmatrix}$$

We take the new shift $\sigma_2 = a_{4,4}^{(2)} = -0.19778$ and get the next

$$R_2 = \begin{bmatrix} -29.4961 & 16.7676 & -0.63089 & -0.24928 \\ & -8.1885 & -12.7059 & -1.2508 \\ & & 11.2182 & 1.4258 \\ & & & 0.1996 \end{bmatrix}$$

Now the bottom element is quite small which will make the last subdiagonal element even smaller in

$$A_3 = \begin{bmatrix} 33.5439 & -3.1162 & 1.849 & 0.0071313 \\ -3.2776 & 11.2646 & 9.3767 & 0.049679 \\ & -3.9106 & -10.8084 & -0.045647 \\ & & -0.027673 & -0.00010351 \end{bmatrix}$$

The next shift is very small, we actually have one zero eigenvalue. In the next R_3 we will get the last element $r_{4,4} = 0.0001$ and the matrix

$$A_4 = \begin{bmatrix} 33.952 & -1.16 & 0.57038 & 1.9624e-008 \\ -1.1269 & 6.6039 & 15.7545 & 4.5179e-007 \\ & 2.3502 & -6.5559 & -2.4933e-007 \\ & & -4.113e-007 & -1.3074e-014 \end{bmatrix}$$

which is ready to deflate.

We now take the shift from the leading 3×3 as $\sigma = a_{3,3}^{(4)} = -6.5559$. The next transformed matrix is

$$A_5 = \begin{bmatrix} 33.9945 & -0.36846 & -0.20014 & -7.0527e-009 \\ -0.37072 & 9.2971 & -12.925 & -4.0113e-007 \\ & 0.48995 & -9.2915 & -3.2514e-007 \\ & & -4.113e-007 & -1.3074e-014 \end{bmatrix}$$

Now we look at the element in position $(3, 2)$ and see that it is getting smaller. Another step on the leading 3×3 gives

$$A_6 = \begin{bmatrix} 33.999 & -0.1592 & 0.085271 & 3.6171e-009 \\ -0.15922 & 8.952 & 13.4072 & 4.096e-007 \\ & -0.0089776 & -8.951 & -3.1446e-007 \\ & & -4.113e-007 & -1.3074e-014 \end{bmatrix}$$

and now the interesting subdiagonal element is substantially smaller. It can actually be proved that this element is converging to zero quadratically.

Let us do a last step on the 3×3 and get

$$A_7 = \begin{bmatrix} 33.9998 & -0.066365 & -0.035536 & -2.0987e-009 \\ -0.066365 & 8.9445 & -13.4164 & -4.0977e-007 \\ & 3.3717e-006 & -8.9443 & -3.1425e-007 \\ & & -4.113e-007 & -1.3074e-014 \end{bmatrix}$$

Now we are ready in all essential. We know that the leading eigenvalue is $\lambda = 34$, the common row and column sum of the magic square. (Why?)

4.1.4 Computing eigenvectors

After the QR algorithm with deflation, A_k is in triangular Schur form T and its eigenvalues are in the diagonal. In the symmetric (Hermitian) case, it is diagonal and the transformation matrix WU will have eigenvectors as columns.

In the nonsymmetric case we solve a triangular linear system by back substitution for each eigenvector.

$$(T - \lambda_k I)x = 0$$

Note that λ_k is one diagonal element of T,

$$\begin{bmatrix} \lambda_1 - \lambda_k & t_{1,2} & \dots & t_{1,k} & \dots & t_{1,n} \\ & \lambda_2 - \lambda_k & \dots & t_{2,k} & \dots & t_{2,n} \\ & & \ddots & \vdots & & \vdots \\ & & & 0 & & \vdots \\ & & & & \ddots & \vdots \\ 0 & & & & & \lambda_n - \lambda_k \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ \vdots \\ x_n \end{bmatrix} = 0$$

This system is singular and we can choose the kth element $x_k = 1$. Then the first $k-1$ elements are computed by back substitution on the upper $(k-1) \times (k-1)$ submatrix $T_{k-1,k-1} - \lambda_k I$. The trailing $n-k$ elements are chosen as zeros.

The back substitution operation involves divisions with differences $\lambda_i - \lambda_k$, and if two eigenvalues have the same value it may fail. In that case, we say that the matrix is *defective* and does not have a full set of n linearly independent eigenvectors. Not all matrices with multiple eigenvalues are defective, if the relevant part of T is diagonal, we have a full set of eigenvectors.

Numerical example, conclusion The resulting triangular matrix in our example is

$$T = \begin{bmatrix} 34 & 0 & 0 & 0 \\ & 8.9443 & 13.4164 & 0 \\ & & -8.9443 & 0 \\ & & & 0 \end{bmatrix}$$

The first eigenvector is chosen as $x_1 = e_1 = (1, 0, 0, 0)^T$, the next as $x_2 = e_2$ since the leading 2×2 is diagonal. Then the third vector is obtained by back

substitution in the leading 2×2 which is just a division by $\lambda_3 - \lambda_2 = -17.8885$, giving the second element $x_{2,3} = t(2,3)/(\lambda_3 - \lambda_2) = -0.75$, and the eigenvector $x_3 = (0, -0.75, 1, 0)^T$. The fourth vector $x_4 = e_4$ since now we have a block of zeros again.

Normally we normalize the eigenvectors to unit Euclidean length, this gives the eigenvector matrix,

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -0.6 & 0 \\ 0 & 0 & 0.8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

to the triangular T and

$$WUX = \begin{bmatrix} 0.5 & 0.82361 & -0.37639 & -0.22361 \\ 0.5 & -0.42361 & -0.023607 & -0.67082 \\ 0.5 & -0.023607 & -0.42361 & 0.67082 \\ 0.5 & -0.37639 & 0.82361 & 0.22361 \end{bmatrix}$$

gives eigenvectors of the original matrix A. Note that the eigenvectors of a nonsymmetric matrix do not form an orthogonal matrix, but that in this case the first and last columns are orthogonal to each other.

4.2 Iterative eigenvalue algorithms

Iterative algorithms compute a sequence of vectors that hopefully converges to an eigenvector. The most basic iteration is the power method, where the x_0 is a starting guess and then a sequence x_k is computed by

$$x_k = Ax_{k-1} \tag{4.3}$$

After many iterations x_k will tend to an eigenvector, corresponding to the eigenvalue λ_1 that is largest in absolute value, provided there is only one such eigenvalue.

The power method is very slow to converge, most often the absolutely largest λ_1 is not very much larger than the others in absoute value. It is also rather wasteful, we just forget the vectors x_i already computed, and keep looking at the newest one x_k.

4.2.1 Arnoldi algorithm

We will get more interesting algorithms, if we save all vectors in the sequence (4.3), and get the *Krylov subspace*

$$K_k(A, x_0) = \{x_0, Ax_0, \ldots, A^{k-1}x_0\} \tag{4.4}$$

(Here the brackets $\{\ldots\}$ mean linear span of the columns given)

The vectors in the basis (4.4) will be close to linearly dependent, if we just compute them by the power method recursion (4.3). We may compute an orthonormal basis of the Krylov space (4.4) by means of the Arnoldi algorithm:

ALGORITHM ARNOLDI

Start with $q_1 = x/\|x\|_2$ where x is a starting vector.
For $k = 1, 2, \ldots$

1. $u = Aq_k$
2. For $j = 1, \ldots, k$
   (1) $h_{j,k} = q_j^H u$
   (2) $u = u - q_j h_{j,k}$
3. $h_{k+1,k} = \|u\|_2$
4. $q_{k+1} = u/h_{k+1,k}$

END.

Let us follow what happens. In the first step in the loop, we multiply with the matrix A, which brings us one step forward in the Krylov sequence (4.4). The inner loop in step 2 then makes the computed vector u orthogonal to each of the previous vectors q_j. This is one step of the Modified Gram Schmidt algorithm for orthogonalization. In step 3, the vector u is ortogonal to all the previous q_j, $j = 1, \ldots, k$. It is normalized to unit length and put in as the next basis vector q_{k+1}.

Eliminate the intermediate vector u used in the algorithm, and get the *basic recursion*

$$\begin{aligned} Aq_k &= q_1 h_{1,k} + q_2 h_{2,k} + \cdots + q_k h_{k,k} + && h_{k+1,k} q_{k+1} \\ &= Q_k h_k + && h_{k+1,k} q_{k+1} \\ AQ_k &= Q_k H_{k,k} + && h_{k+1,k} q_{k+1} e_k^T \end{aligned} \tag{4.5}$$

In the last line, we have added the basic recursions from previous steps, and got a matrix $H_{k,k}$ of Hessenberg form,

$$H_{k,k} = \begin{bmatrix} h_{1,1} & h_{1,2} & \cdots & & h_{1,k} \\ h_{2,1} & h_{2,2} & \cdots & & h_{2,k} \\ 0 & h_{3,2} & \cdots & & h_{3,k} \\ & & \ddots & & \vdots \\ 0 & 0 & & h_{k,k-1} & h_{k,k} \end{bmatrix}$$

The residual term to the right is orthogonal to the Krylov space (4.4), and we see that

$$H_{k,k} = Q_k^H A Q_k \tag{4.6}$$

the restriction of the operator A to the Krylov space spanned by Q_k. Its eigenvalues approximate those of A, they are called *Ritz* values.

Let us take an eigenpair (θ, s) of $H_{k,k}$

$$H_{k,k}s = s\theta \tag{4.7}$$

then the vector

$$y_k = Q_k s \tag{4.8}$$

is an approximate eigenvector of the original matrix A. Compute its residual

$$\begin{aligned} Ay_k - y_k\theta &= AQ_k s - Q_k s\theta && \text{(From definition(4.8))} \\ &= (Q_k H_{k,k} + h_{k+1,k}q_{k+1}e_k^T)s - Q_k s\theta && \text{(From basic recursion (4.5))} \\ &= h_{k+1,k}q_{k+1}e_k^T s && (s \text{ is eigenvector (4.7)}) \end{aligned}$$

The residual is in the direction of q_{k+1}, the next basis vector, which is orthogonal to the Krylov space (4.4) spanned by Q_k, we say that we have a *Galerkin* approximation. We note that $\|q_{k+1}\|_2 = 1$ so the norm of the residual is

$$\|Ay_k - y_k\theta\|_2 = |h_{k+1,k}s_k| \tag{4.9}$$

the product of the next subdiagonal element and the last element of the eigenvector of $H_{k,k}$. If this is very small, we have an indication of convergence of θ towards one eigenvalue λ of A.

This relation (4.9) is the basis for how Arnoldi is used as an iterative eigenvalue algorithm for matrices that are large and sparse, too large to perform similarity transformations. The original matrix A is accessed only to perform matrix vector operations in step 1 of ALGORITHM ARNOLDI, and any type of sparsity structure can be used. We need to compute and store the vectors q_k, but the computation of Ritz values (4.7) and estimate of residual norms (4.9) can be performed using only the small $k \times k$ matrix $H_{k,k}$. We can repeat this computation at regular intervals, until we reach a value of k, for which the residual estimate (4.9) is small enough. Then we have to compute the Ritz vector y (4.8), which is a major operation, since it involves k long n vectors. In normal practice, the number of steps k is much smaller than the order of the matrix n, say that $k = 20 - 50$ while $n = 10000$.

The Arnoldi algorithm is getting very heavy computationally if a large number of steps k are needed. The orthogonalization in step 2 of ALGORITHM ARNOLDI will involve the order of nk flops in step k and for large k it will be very time consuming and start to dominate.

4.2.2 Lanczos algorithm

If the matrix A is real symmetric or complex Hermitian, then the Hessenberg matrix $H_{k,k}$ (4.6) is real symmetric and tridiagonal. To see this, note that its conjugate transpose, $H_{k,k}^H = (Q_k^H A Q_k)^H = Q_k^H A^H (Q_k^H)^H = Q_k^H A Q_k = H_{k,k}$ which is following from $A^H = A$ and $(Q^H)^H = Q$. Moreover the subdiagonal

element $h_{k+1,k}$ is computed as a norm in ALGORITHM ARNOLDI step 3, and the diagonal element is always real for a Hermitian matrix. We can replace $H_{k,k}$ by the tridiagonal matrix

$$T_k = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ 0 & \beta_2 & \alpha_3 & & & \\ & & \ddots & \ddots & \ddots & \\ & & & \beta_{k-2} & \alpha_{k-1} & \beta_{k-1} \\ & & & & \beta_{k-1} & \alpha_k \end{bmatrix} \tag{4.10}$$

The orthogonalization in step 2 of ALGORITHM ARNOLDI will be replaced by two vector subtractions and we get the Lanczos algorithm:

ALGORITHM LANCZOS

Start with $q_1 = x/\|x\|_2$ where x is a starting vector.
For $k = 1, 2, \ldots$

1. $u = Aq_k - q_{k-1}\beta_{k-1}$
2. $\alpha_k = q_k^H u$
3. $u = u - q_k\alpha_k$
4. $\beta_k = \|u\|_2$
5. $q_{k+1} = u/\beta_k$

END.

In the step 1 we assume that $q_0 = 0$ and $\beta_0 = 0$ the first time $k = 1$. For later steps $k > 1$, we already know the orthogonalization coefficient β_{k-1} from symmetry.

The Lanczos algorithm needs only a few scalar products and vector additions, together with the matrix vector multiplication, in each step k, so it can be run for substantially more steps k than Arnoldi. Moreover we need not save all the basis vectors q_k, only the last two vectors need to be available, which makes it possible to use Lanczos for considerably larger matrices than Arnoldi.

However, orthogonality of the basis vectors q_k depends on an assumption on symmetry, and gets lost after some time. It was proved by Christopher Paige in his thesis 1970, that this happens precisely when the first eigenvalue converges. The safest way to avoid trouble with this is to reorthogonalize the vectors, but then we need to do as much work as we do with Arnoldi.

4.2.3 Spectral Transformation

A standard practice to find eigenvalues of large matrix pencils (4.1) is to apply a Krylov space algorithm, Lanczos or Arnoldi, to a shift invert spectral transformation,

$$C = (A - \sigma B)^{-1} B \text{ with eigenvalues } \theta_j = 1/(\lambda_j - \sigma)\,. \tag{4.11}$$

It will compute eigenvalues λ_j close to the shift point σ in the complex plane[1].

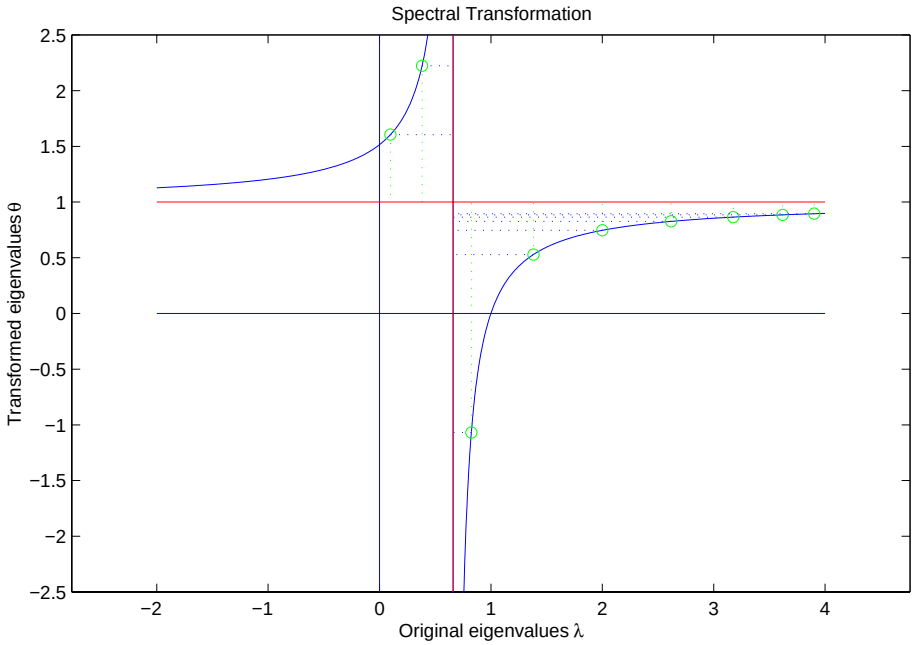

Figure 4.1: Shift invert spectral transformation $\theta = \frac{\lambda-\mu}{\lambda-\sigma}$ *Slightly different from the one described in text!*

Chapter 5

Linear systems: Iterative algorithms

Now we return to algorithms for numerical solution of a linear system

$$Ax = b \tag{5.1}$$

and seek an iteration that generates a sequence of vectors, $x_1, x_2, \ldots, x_k, \cdots \rightarrow x^*$, where $Ax^* = b$ the solution of the system (5.1)

5.1 Krylov sequence methods

We can look in the Krylov space spanned by the vectors obtained by successively premultiplying a starting vector x_0 by the matrix A,

$$K_k(A, x_0) = \{x_0, Ax_0, \ldots, A^{k-1}x_0\} \tag{5.2}$$

In this section let us assume that the matrix A is symmetric positive definite. This happens in many important practical cases. Let us choose the normalized right hand side b of the system (5.1) as starting vector $q_1 = b/\|b\|_2$ and compute an orthogonal basis Q_k of the Krylov space (5.2) by means of the Lanczos algorithm

$$AQ_k = Q_k T_k + \beta_k q_{k+1} e_k^T \tag{5.3}$$

with a tridiagonal matrix T_k (4.10).

Let us take an approximate solution $x_k = Q_k z_k$ from the Krylov space (5.2). If we choose $z_k = T_k^{-1} e_1 \|b\|_2$, we get a residual

$$\begin{aligned}
r_k =& b - Ax_k \\
=& Q_k e_1 \|b\|_2 - AQ_k z_k && \text{(b is starting vector)} \\
=& Q_k e_1 \|b\|_2 - Q_k T_k z_k - \beta_k q_{k+1} e_k^T z_k && \text{(basic recursion (5.3))} \\
=& Q_k (e_1 \|b\|_2 - T_k z_k) - q_{k+1} \beta_k \zeta_k \\
=& -q_{k+1} \beta_k \zeta_k && \text{(by choice of } z_k\text{)}
\end{aligned}$$

The residual r_k is orthogonal to the Krylov space (5.2) and its norm is getting small if the last component ζ_k of the vector z_k is small. The vector z_k is proportional to the first column of the inverse T^{-1}.

5.2 Conjugate gradient algorithm

We can derive a short recursion to compute the entire sequence of approximate solutions x_k without needing to factorize T_k anew each step k. We assumed that the matrix A was symmetric positive definite. Then that is true also for the tridiagonals T_k, let us write its Gaussian factorization,

$$T_k = L_k D_k L_k^T = \begin{bmatrix} 1 & & & \\ l_1 & 1 & & \\ & \ddots & \ddots & \\ & & l_{k-1} & 1 \end{bmatrix} \begin{bmatrix} d_1 & & \\ & \ddots & \\ & & d_k \end{bmatrix} \begin{bmatrix} 1 & l_1 & & \\ & \ddots & \ddots & \\ & & 1 & l_{k-1} \\ & & & 1 \end{bmatrix} \tag{5.4}$$

Now the approximation at step k is

$$\begin{aligned} x_k &= Q_k T_k^{-1} e_1 \|b\|_2 \\ &= Q_k (L_k^{-T} D_k^{-1} L_k^{-1}) e_1 \|b\|_2 \\ &= (Q_k L_k^{-T})(D_k^{-1} L_k^{-1} e_1 \|b\|_2) \\ &= \quad P_k \quad y_k \end{aligned}$$

defining the new basis P_k and the vector y_k of length k. The basis P_k is *A-conjugate* which means that

$$\begin{aligned} P_k^T A P_k &= (Q_k L_k^{-T})^T A (Q_k L_k^{-T}) \\ &= L^{-1} (Q_k^T A Q_k) L_k^{-T} \\ &= L^{-1} T_k L_k^{-T} \\ &= L^{-1} (L_k D_k L_k^T) L_k^{-T} = D_k \end{aligned}$$

a diagonal matrix, this is what has given this algorithm its name. The vectors p_k are search directions leading from x_{k-1} to x_k because

$$x_k = P_k y_k = [P_{k-1}\, p_k] \begin{bmatrix} y_{k-1} \\ \eta_k \end{bmatrix} = P_{k-1} y_{k-1} + p_k \eta_k = x_{k-1} + p_k \eta_k \tag{5.5}$$

which gives the recursion to update x_{k-1}.

Now we need a recursion for the p_k vectors. Note that $P_k = Q_k L_k^{-T}$ which means that $Q_k = P_k L_k^T$ and note that the last column of L_k^T has only the two last elements nonzero giving,

$$p_k = q_k - l_{k-1} p_{k-1} \tag{5.6}$$

We started this discussion with computing the orthogonal basis Q_k by means of the Lanczos algorithm. We saw that the residuals r_k had the directions of the q_k. It is now possible to update r_k directly by the recursion

$$r_k = r_{k-1} - Ap_k\eta_k \tag{5.7}$$

which is a direct consequence of the update of x_k (5.5) and the fact that $r_k = b - Ax_k$ so that $r_k - r_{k-1} = -A(x_k - x_{k-1}) = -Ap_k\eta_k$. We can then dispose of the actual computation of Q_k and T_k in the Lanczos algorithm and get the coefficients η_k and l_{k-1} of the updates directly from the vectors p_k, conjugate search directions, and r_k orthogonal residuals.

Gathering all this together, we get:

ALGORITHM CONJUGATE GRADIENT

Start with solution $x_0 = 0$, residual $r_0 = b$, search $p_1 = b$.
For $k = 1, 2, \ldots$

1. $z = Ap_k$
2. $\nu_k = (r_{k-1}^T r_{k-1})/(p_k^T z)$
3. $x_k = x_{k-1} + \nu_k p_k$ (Next iterate (5.5))
4. $r_k = r_{k-1} - \nu_k z$ (Recursive residual (5.7))
5. $\mu_{k+1} = (r_k^T r_k)/(r_{k-1}^T r_{k-1})$
6. $p_{k+1} = r_k + \mu_{k+1} p_k$ (New search (5.6))

END.

5.3 Preconditioning

We noted that the conjugate algorithm is actually Lanczos applied to the eigenvalue problem

$$(A - \lambda I)x = 0$$

We remember that convergence depends on the distribution of eigenvalues, if they are close together towards the ends of the spectrum, convergence will be slow. If we find a matrix M that is a good approximation to the matrix A but is cheap to invert, or more properly solve systems, we can device an algorithm for the *preconditioned* system

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

The preconditioned algorithm can be derived from the Lanzcos algorithm applied to the generalized eigenvalue problem

$$(A - \lambda M)x = 0$$

Inserting the multiplication with M^{-1} at the appropriate place, and using M scalar products, we get:

ALGORITHM PRECONDITIONED CONJUGATE GRADIENT

Start with solution $x_0 = 0$, residual $r_0 = b$, search $p_1 = M^{-1}b$, $y = M^{-1}r_0$.
For $k = 1, 2, \ldots$

1. $z = Ap_k$
2. $\nu_k = (y_{k-1}^T r_{k-1})/(p_k^T z)$ *(M-scalar product)*
3. $x_k = x_{k-1} + \nu_k p_k$ (Next iterate (5.5))
4. $r_k = r_{k-1} - \nu_k z$ (Recursive residual (5.7))
5. $y_k = M^{-1}r_k$ *(Preconditioner)*
6. $\mu_{k+1} = (y_k^T r_k)/(y_{k-1}^T r_{k-1})$ *(M-scalar product)*
7. $p_{k+1} = y_k + \mu_{k+1} p_k$ (New preconditioned search (5.6))

END.

There is a trade off when choosing preconditioner M. A very crude M adds little to the work of each iteration, but needs many iterations to converge. A very accurate M, on the other hand, will need few iterations to converge but will need quite some time to factorize and the operation with the factors $M^{-1}r$ will be heavy in each iteration.

The simplest choice is to take the diagonal elements of A, *diagonal preconditioning*. Another simple minded approach is to do a Cholesky factorization and stop after a limited amount of fill in has occurred. One talks about ICCG(p) for an *Incomplete Cholesky*, filling p diagonals, adding to those alreday full. If we have an incomplete factorization

$$A = LL^T + R = M + R$$

where R contains those elements we did not eliminate, the operation with the preconditioning is simply

$$y = M^{-1}r = L^{-T}(L^{-1}r)$$

effected by a sparse forward substitution followed by a sparse back substitution, as indicated by the parentheses. We may also let a *drop tolerance* determine which elements to leave in R, this variant is implemented in MATLAB .

If we know something more about the matrix, there are more sophisticated variants. If A is a finite difference or finite element approximation to a partial differential equation, we may use a coarse level approximation to give M, *Multigrid*. That will be discussed in connection with algorithms for Partial Differential Equations. M may also be the exact solution over a simple region, for instance a rectangle.

Bibliography

[1] T. Ericsson and A. Ruhe, *The spectral transformation Lanczos method for the numerical solution of large sparse generalized symmetric eigenvalue problems*, Mathematics of Computation, 35 (1980), pp. 1251–1268.