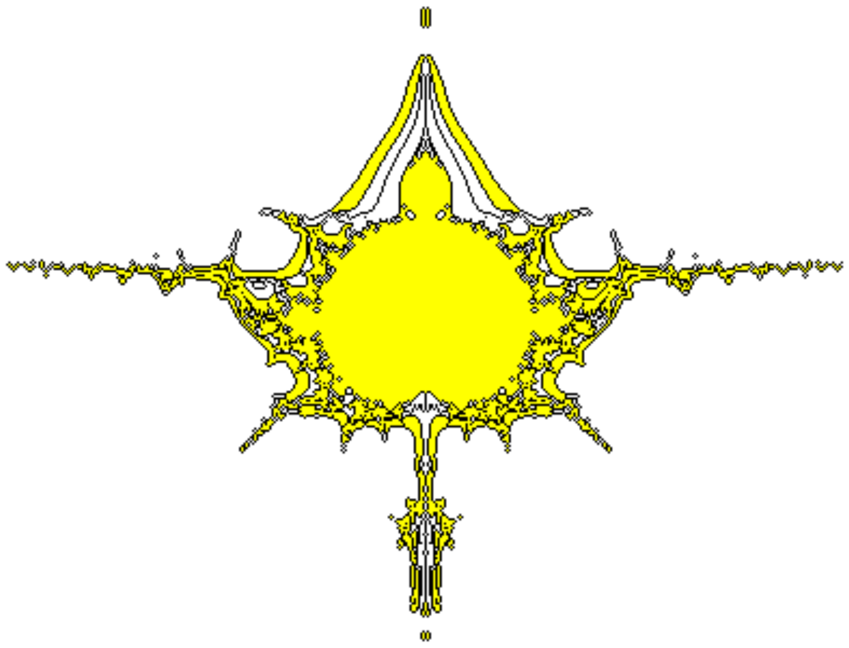
Numerical methods in computational mechanics

Edited by M. Okrouhlík



The presented publication might be of interest to students, teachers and researchers who are compelled to solve nonstandard tasks in solid and/or fluid mechanics requiring large scale computation.

The publication was prepared within the project 1ET400760509 of the Information Society and supported by the Academy of Sciences of the Czech Republic

Institute of Thermomechanics, Prague 2008

Last revision January 23, 2012

Contents

Chapter 1

Introduction

This part was written and is maintained by M. Okrouhlík. More details about the author can be found in the Chapter 16.

The presented publication might be of interest to human problem solvers in continuum mechanics, who are not professional programmers by education, but by destiny and/or by personal inquisitiveness are compelled to resolve nonstandard tasks stemming from the field of solid and/or fluid mechanics requiring large sized computation.

It is dedicated to matrix algebra, in its broadest sense, which allows for the transparent formulations of principles of continuum mechanics as well as for effective notations of algebraic operations with large data objects frequently appearing in solution of continuum mechanics problems by methods discretizing continuum in time and space.

The publication has been prepared by eight authors. Authors and the chapters the contributed to are as follows.

- Marta Čertíková – Chapter 7,
- Alexandr Damašek – Chapter 13,
- Jiří Dobiáš – Chapter 8,
- Dušan Gabriel – Chapter 9,
- Miloslav Okrouhlík – Chapters 1, 2, 3, 4, 5, and 6,
- Petr Pařík – Chapter 11,
- Svatopluk Pták – Chapter 10,
- Vítězslav Štembera – Chapter 12.

More about authors' whereabouts – their affiliations, fields of expertise, selected references and the contact addresses, can be found in the Chapter 16.

The publication is composed of contributions of two kinds.

- First, there are contributions dealing with fundamentals and the background of numerical mathematics in computational mechanics, accompanied by templates and programs, which are explaining the standard matrix operations (chapter 2) and procedures needed for the solution of basic tasks as the solution of linear algebraic systems, generalized eigenvalue problem, solution of nonlinear task and the solution of ordinary differential equations, etc. (Chapters 3, 4, 5). We focus on nonstandard storage schemes, chapter (11), allowing to tackle large scale tasks. By a template we understand a general broad term description of the algorithm using high level metastatements like *invert the matrix*, or *check the convergence*. The programs accompanying the text are written in Basic, Pascal, Fortran, Matlab and `C#`. The presented programs are rather short and trivial and are primarily intended for reading. The e-book readers (or users?) are encouraged to fiddle with these programs.

- Second, there are advanced contributions describing modern approaches to numerical algorithmization with emphasis to novelty, programming efficiency and/or to complicated mechanical tasks as the contact treatment.

  The attention is devoted to

  - domain decomposition methods, (Chapter 7),
  - BDDC methods, and FETI-DP methods , (Chapter 8),
  - the BFGS method applied to a new contact algorithm, (Chapter 9),
  - frontal solution method, (Chapter 10),
  - details of sparse storage modes, (Chapter 11),
  - intricacies of object programming approach, (Chapter 12),
  - programming the Arbitrary Lagrangian Eulerian approach to the solid-fluid problem, (Chapter 13), etc.

  Programming considerations for parallel treatment are observed. These contributions have their origin in papers that the contributing authors recently published in scientific journals and/or presented at conferences.

There are dozens of programming libraries securing matrix algebra operations. They are available in variety of programming languages.

BLAS library contains an extended set of basic linear algebra subprograms written in Fortran 77 and is at

`www.netlib.org/blas/blas2-paper.ps.`

BLAST Basic linear algebra subprograms technical standards provides extensions to Fortran 90. See

`www.netlib.org/blas/blast-forum.`

A collection of Fortran procedures for mathematical computation based on the procedures from the book *Computer Methods for Mathematical Computations*, by George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. Prentice-Hall, 1977. It can be downloaded from

`www.pdas.com/programs/fmm.f90.`

And also MATLAB matrix machine

`http://www.mathworks.com/products/matlab`

should be mentioned here, together with a nice introductorily text by one of Matlab founding fathers C. Moler available at

`www.mathworks.com/moler.`

In C programming language it is the GNU Scientific Library (or GSL) that provides the software library for numerical calculations in applied mathematics and science. Details can be found in

`http://www.gnu.org/software/gsl/`

Similar packages are available in C++, Java and other programming languages.

`http://www.ee.ucl.ac.uk/~mflanaga/java/`

Most of these are sophisticated products with a high level of programming primitives, treating matrices as building blocks and not requiring to touch individual matrix elements.

So having all these at hand, readily available at internet, one might pose a question why one should dirty his/her fine fingertips by handling matrix elements one by one.

Comparing the menial effort required to write a program for solving the system of algebraic equations in Basic, Pascal or Fortran with that demanded in Matlab, where a simple backslash operator suffices, one might come to a conclusion that to work with element level languages is a futile activity that should be avoided at all the costs.

There are at least two important reasons why it is not so.

- First, to effectively use languages with high-level programming primitives, allowing efficient processing of matrix algebra tasks, requires detailed knowledge of the matrix algebra rules and the knowledge of the rules of a particular language and/or package being used. So it is advisable to know what to choose and how to use it.

- Second, and even more important, is the size and the 'standardness' of the problem we are attempting to solve. This e-book is primarily written for non-standard problem solvers. The 'standard' and 'regular' problems could be rather easily tackled using a brute force approach using standard problem oriented packages and freely available program libraries covering a wide range of numerical problems. Non-standard problems and/or huge memory and time requirements necessitate - among others - to be familiar with efficient data storage schemes which in turn require that standard procedures providing matrix algebra operations have to be completely rewritten from the scratch which in turn requires to be familiar with matrix operations on an element level. So dirtying our hands with low, element level programming is a necessity required until the progress in computer technology provides for a new measure of what is a huge size problem and what is a non-standard problem.

We would like to conclude these introductory remarks by giving our readers a few seemingly trivial hints.

Programming the algorithm might help to clearly understand it and this respect the choice of the programming language is almost immaterial. Nevertheless to write the efficient and robust code and implement it into the existing software the choice of a 'suitable' language is of the utmost significance.

The information available on the internet is only a necessary condition for the successful path to the sought after result.

It have to be complemented by solid background knowledge of physics that stays behind the task, mathematics and the rules of nature modelling with the emphasis on the proper observation of model limits.

And last but not least, the engineering intuition and the programming craftsmanship have to fruitfully harvested.

Chapter 2

Matrix algebra background

This part was written and is maintained by M. Okrouhlík. More details about the author can be found in the Chapter 16.

2.1 Preliminary notes

Matrix algebra, together with tensor algebra, allows for the transparent formulations of principles of continuum mechanics and for effective notations of algebraic operations with large data objects frequently appearing in solution of continuum mechanics problems by discrete methods in time and space. For more details see [1], [2]. The fundamental operations are

- solution of linear algebraic equations, see the Paragraph 3.2, Chapter 4,
- matrix inversion, see the Paragraph 4.5,
- singular value decomposition, see

  `http://www.netlib.org/lapack/lug/lapack_lug.html.`
- standard and generalized eigenvalue problem, see the Paragraph 3.3,
- solution of ordinary differential equations, see the Paragraph 3.4.

The solution of algebraic equations is a basic step needed for the efficient solution of partial differential operations by difference methods as well as for the solution of static as well as of dynamic tasks of structural analysis by finite element method.

Often, the inverse matrix computation is required. It should be emphasized, however, that in most cases we try to avoid the explicit inverse computation since it destroys otherwise advantageous matrix properties as its sparsity and/or bandedness.

Assessing the vibration of mechanical structures requires solving the generalized eigenvalue problem allowing to find the eigenfrequencies and eigenmodes of structures.

The transient problems of mechanical structures, solved by discrete methods, lead to the solutions of ordinary differential equations, requiring to employ the direct time integration of ordinary differential equations and these methods often necessitate to provide repeated solution of algebraic equations.

2.2 Matrix algebra notation

The $m \times n$ matrix is defined as a set of algebraic or numeric items – matrix elements – that are arranged into m rows and n columns. A *matrix* is an array of its elements. In the text the matrices will be denoted by straight bold capital letters, say $\mathbf{A}$, sometimes emphasizing the matrix dimension as $\mathbf{A}_{m\times n}$.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & & a_{2j} & & a_{2n} \\ \vdots & & & & \vdots & & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \dots & a_{ij} & \dots & a_{1n} \\ \vdots & & & & \vdots & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mj} & \dots & a_{mn} \end{bmatrix}_{m\times n}$$

Alternatively the notation with brackets, as $[A]$, might be used. In classical textbooks a typical matrix element is denoted a_{ij}, while the notation A_{ij} can be found in those devoted to programming. See [7]. The programming details depend on the language employed – in Matlab, (www.mathworks.com), Basic [5] and Fortran [3] we write `A(i,j)`. In Pascal [4] we use `A[i,j]`, the C language notation is `A[i][j]`, etc. Notice that the programming 'image' of $\mathbf{A}$, being printed by a straight bold font, is denoted by a smaller thin Courier font – i.e. `A` – in this text.

If the matrix has the same number of rows as columns, i.e. $(m = n)$, we say that it is square and of the n-th order.

The *column vector* (a one column matrix) is

$$\mathbf{x} = \left\{ \begin{matrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{matrix} \right\}_{n\times 1} = \mathbf{x}_{n\times 1}.$$

The *row vector* (a one row matrix), can be viewed as a transpose of a column vector, i.e.

$$\mathbf{x}^{\mathrm{T}} = \{x_1\, x_2\, x_3\, \cdots\, x_n\}_{(1\times n)}.$$

To save the printing space a column vector might be written as

$$\mathbf{x} = \{x_1\, x_2\, x_3\, \cdots\, x_n\}^{\mathrm{T}}.$$

The *diagonal* and *unit matrices*

$$\mathbf{D} = \begin{bmatrix} d_{11} & & & \\ & d_{22} & 0 & \\ & & \ddots & \\ & 0 & & \\ & & & d_{nn} \end{bmatrix}, \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & & 0 \\ 0 & 0 & 1 & & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}.$$

The *lower triangular matrix*

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & & & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix}.$$

The *upper triangular matrix*

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

For elements of a *symmetric matrix* we can write $a_{ij} = a_{ji}$ for $i = 1, 2, \ldots n$; $j = 1, 2, \ldots n$.

The Matlab, see http://www.mathworks.com, as a high-performance language with high-order programming primitives suitable for technical computing, should be mentioned here. Throughout the text the Matlab will serve as a sort of matrix shorthand. It will frequently be used for explanation of programming details. It works with matrices as building blocks, circumventing thus the necessity to address individual elements. Nevertheless, to fully employ its power and efficiency it is important to fully understand the basic rules of matrix algebra and be familiar with ways how they are implemented. So dirtying our hands with element level programming is sometimes necessary for educational purposes. A few following paragraphs might clarify the authors' intentions.

Matrix determinant

The matrix determinant is defined for square matrices only. It can be viewed as an operator uniquely assigning the scalar value to a matrix. We write $d = \det \mathbf{A} = \|\mathbf{A}\|$. The algorithms for determinant evaluation are of little importance today. Classical approaches are too time demanding, furthermore the determinant value could easily be obtained as a side-product of Gauss elimination process – see

```
\cite{Breezer2008} and http://linear.ups.edu/index.html.
```

If the determinant value is equal to zero the matrix is classified as singular.

Nullity and rank of the matrix

The nullity of a matrix – nul($\mathbf{A}$) – is a scalar value characterizing the 'magnitude' of singularity, i.e. the number of linearly dependent rows or columns. The rank – rank($\mathbf{A}$) – is a complementary quantity obtained from $\mathrm{nul}(\mathbf{A}) = n - \mathrm{rank}(\mathbf{A})$, where n is the order or the 'dimension' of the matrix. For a regular matrix its nullity is equal to zero and its rank is equal to its order or dimension. In Matlab we use a built-in function allowing to write `r = rank(A)`

Trace of the matrix

The trace is defined for square matrices only. Its value is equal to the sum of its diagonal elements, i.e. $\text{trace}(\mathbf{A}) = \sum_{i=i}^{n} a_{ii}$. In Matlab we simply use the `trace` function.

Matrix equality $\mathbf{A}_{m\times n} \leftarrow \mathbf{B}_{m\times n}$

The matrix equality is defined for matrices of the same type only. Based on equality is defined the assignment, which in Matlab is naturally written as `A = B` and expresses the following sequence of statements dealing with matrix elements

```
for i = 1:m
  for j = 1:n
    A(i,j) = B(i,j)
  end
end
```

Matrix addition $\mathbf{C}_{m\times n} \leftarrow \mathbf{A}_{m\times n} + \mathbf{B}_{m\times n}$

The matrix addition in Matlab is provided by `C = A + B` and expresses the following sequence of statements dealing with matrix elements

```
for i = 1:m
  for j = 1:n
    C(i,j) = A(i,j) + B(i,j)
  end
end
```

Similarly for the substraction.

Matrix transposition $\mathbf{A}_{m\times n} \leftarrow \mathbf{B}^{\mathrm{T}}_{n\times m}$

By transposition the process of exchanging rows and columns is understood. In Matlab we write `A = B'` for real matrices, while the statement `A = B.'` is required for hermitian ones. On the element level it means

```
for i = 1:m
  for j = 1:n
    A(i,j) = B(j,i)
  end
end
```

If the elements of a matrix **A** are matrices themselves (submatrices), then the transpose of **A** requires not only the transposition of submatrices, but the transposition of submatrices elements as well.

$$\mathbf{A} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \\ \mathbf{B}_{31} & \mathbf{B}_{32} \end{bmatrix}; \qquad \mathbf{A}^{\mathrm{T}} = \begin{bmatrix} \mathbf{B}^{\mathrm{T}}_{11} & \mathbf{B}^{\mathrm{T}}_{21} & \mathbf{B}^{\mathrm{T}}_{31} \\ \\ \mathbf{B}^{\mathrm{T}}_{12} & \mathbf{B}^{\mathrm{T}}_{22} & \mathbf{B}^{\mathrm{T}}_{32} \end{bmatrix}$$

For the transposition of a sum or a difference we can write

$$(\mathbf{A}\pm\mathbf{B})^{\mathrm{T}} = \mathbf{A}^{\mathrm{T}}\pm\mathbf{B}^{\mathrm{T}}.$$

For the transposition of a matrix product we have $(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$. The double transposition does not change the matrix, i.e. $(\mathbf{A}^{\mathrm{T}})^{\mathrm{T}} = \mathbf{A}$. A symmetric matrix is insensitive to the transposition process.

The transposed row vector changes into the column one and vice versa.

Scalar matrix multiplication $\mathbf{A}_{m\times n} \leftarrow c\,\mathbf{B}_{m\times n}$

means to multiply each element of the matrix by the same scalar value, i.e.

```
for i = 1:m
  for j = 1:n
    A(i,j) = c*B(i,j)
  end
end
```

and in Matlab requires to write `A = c*B`. Similarly the division by a scalar is `A = B/c`.

Matrix multiplication $\mathbf{C}_{m\times p} \leftarrow \mathbf{A}_{m\times n}\,\mathbf{B}_{n\times p}$

In Matlab it suffices to write `C = A*B`. Two matrices could only be multiplied if the number of columns in $\mathbf{A}$ matrix is equal to the number of rows in matrix $\mathbf{B}$. This condition is often called *conformability requirement*.

The matrix multiplication is not a commutative process. For the transposition of a matrix product we have $(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$.

The matrix multiplication at an element level requires to use three embedded do-loops. By the outer loops we subsequently address all the elements, while the innermost one provides the definitoric summation, i.e.

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj} \qquad \begin{array}{lll} for & i & = 1, 2, \cdots m; \\ & j & = 1, 2, \cdots p, \end{array}$$

which could be implemented as

```
for i = 1:m
  for j = 1:p
    sum = 0;
    for k = 1:n, sum = sum + A(i,k)*B(k,j); end
    C(i,j) = sum;
  end
end
```

The matrix multiplication process is quite straightforward. Still it requires additional concern about its efficient implementation. The order of row- and column-loops is to be adjusted, according to the programming language being used. For example the Fortran stores matrix arrays columnwise, while the Pascal, Java and C-like languages store them in the row order. So the algorithm for matrix multiplication shown above is suitable for Matlab, Pascal and C-like languages. If we were programming it in Fortran, the `i`-loop should be the inner one.

The matrix multiplication requires roughly n^3, n being the matrix dimension, floating point operations. For a long time there are attempts to make this process more effective. The fastest known algorithm, devised in 1987, requires the number of operations proportional to $n^{2.38}$. More details can be found in

`http://www.siam.org/pdf/news/174.pdf.`

Fortran 77 note

The FORTRAN 77 does not allow for dynamic array dimensioning. The arrays have to be of fixed size when the main program (with proper subroutines) is being compiled. To circumvent this deficiency a few auxiliary parameters have to be introduced. In our academic case with matrix multiplication one might proceed as follows.

In `DIMENSION` statements for the matrix arrays, instead of actual values of variables `M, N, P` which might not be known in advance, we use their maximum predictable values, which might occur in the solved problem, and pass them to a proper subroutine. Assume that the values of parameters defining the actual matrix sizes are `M = 2, N = 5, P = 3` and that the maximum predictable values are `MMAX = 20, NMAX = 50, PMAX = 30`.

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

$$(m \times n)(n \times p) = (m \times p)$$
$$(2 \times 5)(5 \times 3) = (2 \times 3)$$

The matrix multiplication programming segment can then be written as indicated in the Program 1.

Program 1

```
      Program MAIN
C     DIMENSION A(MMAX, NMAX), B (NMAX, PMAX), C (MMAX, PMAX) C
      DIMENSION A(20, 50), B(50, 30), C(20, 30)
      INTEGER   P, PMAX
C     define matrices A,B and their maximum dimensions
      MMAX = 20
      NMAX = 50
      PMAX = 30
C     define dimensions for the actual task to be solved
      M = 2 ! must be .LE. MMAX  ! conformability requirements
      N = 5 ! must be .LE. NMAX  ! of matrix product
      P = 3 ! must be .LE. PMAX  ! have to be observed
      CALL MAMU2 (C, A, B, M, N, P, MMAX, NMAX, PMAX)
C     ...
```

`The corresponding procedure for matrix multiplication is`

```
      SUBROUTINE MAMU2 (C, A, B, M, N, P, MMAX, NMAX, PMAX)
      DIMENSION C(MMAX,P), A(MMAX,N), B(NMAX,P)
      INTEGER P, PMAX
C     C(M,P) = A(M,N) * B(N,P)
      DO 10 I = 1,M
        DO 10 J = 1,P
        C(I,J) = 0.
          DO 10 K = 1,N
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
10    CONTINUE
      RETURN
      END
```

☐ End of Program 1

☐ End of **Fortran 77 note**

Fortran 90 note

In FORTRAN 90 these restrictions are removed by the provision of *dynamic array structuring.* The arrays can be passed to subroutines as arguments with their actual run-time sizes. In FORTRAN 90 one could implement the matrix multiplication by a segment code indicated in the Program 2.

Program 2

```
SUBROUTINE MAT_MUL(A,B,Y,L,M,N)
! multiply A(L,M) by B(M,N) giving Y(L,N)
IMPLICIT NONE
INTEGER:: I,J,K,L,M,N
REAL, DIMENSION(L,M):: A
REAL, DIMENSION(M,N):: B
REAL, DIMENSION(L,N):: Y
Y = 0.
DO K = 1,N
  DO I = 1,M
    DO J = 1,L
      Y(J,K) = Y(J,K) + A(J,I) * B(I,K)
    END DO
  END DO
END DO
END SUBROUTINE MAT_MUL
```

☐ End of Program 2

There is a lot of publications devoted to FORTRAN 90. A nice introductory course can found in [6].

☐ End of **Fortran 90 note**

Multiplication of a matrix by a column vector from the right $\mathbf{y}_{m\times 1} \leftarrow \mathbf{A}_{m\times n}\, \mathbf{x}_{n\times 1}$

is carried out according to $\mathbf{y}_i = \sum_{j=1}^{n} A_{ij}\, x_j$ for $i = 1, 2, ...m$. Of course the conformability conditions have to observed. On the element level we write

```
for i = 1:m
  sum = 0;
  for j = 1:n, sum = sum + A(i,j)*x(j); end
  y(i) = sum;
end
```

In Matlab the element-wise approach indicated above gives as an output the row vector, regardless whether the input vector was of a row or a column character. Unless stated otherwise the Matlab rules take primarily vectors as of row nature.

When using the usual Matlab high-level statement as in `y = A*x`, the input vector have to be of column structure, while in `y = A*x'` it has to be a row one. In agreement with matrix algebra rules the output – in both cases – will be a column vector.

Other programming languages as Pascal or Fortran do not distinguish the row/column character of vectors. For these languages the set of elements, addressed by a unique name, is just an array, as it is in tensor analysis.

It should be emphasized that the use of the term 'vector' in programming languages is a little bit misleading, having nothing to do with vectors as we know them from physics – entities defined by its magnitude and direction.

Multiplication of a matrix by a vector from the left $\mathbf{y}_{1\times n} \leftarrow \mathbf{x}_{1\times m}\, \mathbf{A}_{m\times n}$

The algorithm for this multiplication is given by $y_j = \sum_{i=1}^{m} x_i\, A_{ij}$ for $j = 1, 2, ...n$.

Provided that the conformability conditions are satisfied, and the vector `x` is of row nature the Matlab high-level statement is `y = x*A`. On the output we get a row vector as well. If, by chance we start by a row `x` vector, we have to write `y = x'*A`.

Dot product $s \leftarrow (\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$

A dot product is defined for vectors of the same length by $s = (\mathbf{x}, \mathbf{y}) = \mathbf{x}^{\mathrm{T}}\mathbf{y} = \sum_{i=1}^{n} x_i\, y_i$. In Matlab, with both vectors of the column nature, we write `s = x'*y`. The alternative statement `s = dot(x,y)` takes as inputs vectors regardless of their 'columnness' or 'rowness'. On the element level we could write

```
s = 0; for i = 1:n, s = s + x(i)*y(i); end
```

Cross product $\mathbf{c} \leftarrow \mathbf{a} \times \mathbf{b}$

Strictly speaking the cross product does not belong into the menagerie of matrix algebra. It is, however, an operator frequently used in continuum mechanics and in physics generally and thus available as a high-level operator in Matlab. It can be obtained by `c = cross(a,b)`. The dimension of input vectors `a` and `b` must be just three.

Doing it by hand, the Sarrus's scheme could be used for the cross product evaluation. Assuming that in the Cartesian system of coordinates with unit vectors $\vec{i}$, $\vec{j}$, $\vec{k}$ the vectors $\vec{a}, \vec{b}$ have components a_x, a_y, a_z and b_x, b_y, b_z respectively, then the resulting vector $\vec{c} = \vec{a} \times \vec{b}$ is given by

$$\vec{c} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \left\{ \begin{array}{c} a_y\, b_z - a_z\, b_y \\ a_z\, b_x - a_x\, b_z \\ a_x\, b_y - a_y\, b_x \end{array} \right\}.$$

Still another way of expressing the i-th element of the cross product vector – this time in index notation – is

$$c_i = \sum_{j=1}^{3} \sum_{k=1}^{3} \epsilon_{ijk}\, a_j\, b_k,$$

where ϵ_{ijk} is Levi-Civita permutation symbol. Programming it this way would, however, be rather inefficient

Diadic product $\mathbf{A}_{m\times n} \leftarrow \mathbf{x}_{m\times 1}\mathbf{y}_{1\times n}$

The result of diadic product of two vectors of the same length a matrix. The algorithm for its evaluation is as follows

```
for i = 1:m
  for j = 1:n
    A(i,j) = x(i)*y(j)
  end
end
```

If both vectors are conceived as column ones, then one should write `A = x*y'` in Matlab. There is no built-in function for the diadic product evaluation in Matlab.

Matrix inversion $\mathbf{B} \leftarrow \mathbf{A}^{-1}$

If the product of two square matrices is equal to the unity matrix $\mathbf{A}\,\mathbf{B} = \mathbf{I}$, then the matrix $\mathbf{A}$ is inverse of the $\mathbf{B}$ matrix and vice versa. We write $\mathbf{B} = \mathbf{A}^{-1}$ or $\mathbf{A} = \mathbf{B}^{-1}$.

The matrix can be inverted only if $\det \mathbf{A} \neq 0$, that is for a regular matrix. If the matrix is singular, i.e. $\det \mathbf{A} = 0$, its inverse does not exist.

The inverse of the matrix product is

$$(\mathbf{A}\,\mathbf{B})^{-1} = \mathbf{B}^{-1}\,\mathbf{A}^{-1}.$$

Bibliography

[1] K.-J. Bathe. *Finite element procedures.* Prentice-Hall, New Jersey, 1996.

[2] G. Beer. *Introduction to finite and boundary methods for engineers.* John Wiley and Sons, Baffins Lane, Chichester, 1991.

[3] S.J. Chapman. *Fortran95/2003 for scientists and engineers.* McGraw Hill, Engelwood Cliffs, 2007.

[4] J.J. Gregor and A.H. Watt. *Pascal for scientists and engineers.* Pitman, London, 1983.

[5] J.G. Kemeny and T.E. Kurtz. *Back to Basic. The history, corruption, and future of the language.* Adisson-Wesley, 1985.

[6] I.M. Smith. *A Programming in Fortran 90. A first course for engineers and scientists.* John Wiley and Sons, Baffins Lane, Chichester, 1995.

[7] G. Strang. *Linear Algebra and its Applications.* Academic Press, New York, 1976.

Chapter 3

Review of numerical methods in mechanics of solids

This part was written and is maintained by M. Okrouhlík. More details about the author can be found in the Chapter 16.

A review of numerical methods based on matrix algebra is presented, especially of methods of solution of algebraic equations, generalized eigenvalue problems, solution of differential equations describing transient phenomena, and methods for the solution of nonlinear problems.

3.1 Introduction

Lot of problems in engineering practice are solved by means of commercial finite element (FE) packages. From outside, these FE packages appear to be black boxes and a chance to accustom them to user's immediate needs is rather limited. That's why standard FE users are not usually involved in studying and subsequent algorithmization and programming of modern computational methods.

Using modern finite element packages, however, requires making frequent choices between various methods that are available within a package. The principles of some of these methods are quite new and to understand them requires a detailed consulting of voluminous theoretical manuals containing numerous references embedded in various journals and modern textbooks which are not usually at hand.

The widespread availability of PC's equipped by powerful software gives the designers and stress analysts a powerful tool for the efficient use of 'classical' engineering methods allowing bigger and more complicated problems of engineering practice to be solved on condition that the user is able to grasp pertinent numerical techniques, understand them and be able to implement them in his/her computer territory using modern programming tools as Matlab, Maple, Mathematica and last but not least the good old Fortran in its new and powerful F90 version.

Different groups of users of numerical methods require 'black box' software for solving general problem classes without having to understand the details of theory and algorithmization. This kind of software is widely available in the form of mathematical libraries such as LAPACK, LINPACK, NAG, and in various books dedicated to numerical recipes.

See [32], [44], [23], [38]. Also the commercial finite element packages belong to this category. The examples are numerous. ABACUS, ANSYS, COSMOS, etc. are only a few examples from the beginning of the spectrum sorted alphabetically. Others could be found in [51]. Such a software can be reliably used within the limits of its validity, which are more or less precisely stated and declared in their manuals. Less reliable, however, are pieces of information contained in advertisement booklets distributed by their vendors. In this respect it is difficult to suppress the temptation to present a common wisdom concerning the usefulness of other people software, namely *you do not know how it works until you buy it*.

Also we would like to stress out that we are living in a quickly changing world where everything, including the terms large and small, is subjected to never ending scrutiny. Quotations from a few textbooks speak for themselves

- Ralston, 1965 ... by a very large matrix we understand a matrix of the order of 100 or higher [39].
- Parlett, 1978 ... a 200 by 200 symmetric matrix with a random pattern of zero elements is large [37].
- Papandrakakis, 1993 ... with moderately sized problems of a few tens of thousands degrees of freedom [35].

In spite of all these facts the authors are deeply convinced, that the solid knowledge of principles on which these methods are based, the ability of engineering assessment of a wide spectrum of methods which are available, is of utmost importance in everyday life of people engaged in engineering computations of complicated problems of technical practice.

3.2 Solution of algebraic equations

Lot of problems in technical practice leads to the solution of sets of n linear algebraic equations with n unknowns $\mathbf{Ax} = \mathbf{b}$.

In this paragraph only linear sets of algebraic equations will be treated without explicitly stressing out the term linear at each occurrence.

Note

The solution of linear algebraic equation is important for steady state, linear transient and generally nonlinear problems as well, since all the mentioned tasks could be formulated in such a way that at the bottom of each time step or an iteration loop the set of linear equations is solved.

The coefficient matrix $\mathbf{A}$, arising from a displacement formulation of finite element (FE) method, has a meaning of the global stiffness matrix, the $\mathbf{b}$ vector represents the external loading and the $\mathbf{x}$ vector contains the sought-after solution expressed in generalized displacements. The matrix $\mathbf{A}$ is usually *regular*, *symmetric*, *positive definite* and *sparse*.

All matrix properties mentioned above are very important for the solution of large-scale problems and should be properly exploited in algorithmization, programming and implementation of procedures providing matrix operations and handling.

3.2.1 Direct methods

These methods are characterized by the fact that, ignoring the effects of round-off errors, one can reach the exact answer after a finite number of algebraic operations.

3.2.1.1 Gauss elimination

The system of equations is usually solved by a variant of Gauss elimination. This method was known long before Carl Friedrich Gauss (1777 - 1855). Its use for a set of three equations with three unknowns was presented in a Chinese text titled *Nine Chapters on the Mathematical Art* more than 2000 years ago. See [8].

The Gauss elimination method is based on subsequent elimination of unknowns. The multiples of the first equation are being subtracted from all remaining equations in such a way that the first unknown vanishes. By this we get a smaller set of $(n-1)$ equations with $(n-1)$ unknowns. This way we continue until the last equation remains. The last equation has only one unknown whose value can be easily determined. This is the end of the first part of procedure, known by different names – *forward substitution, reduction, factorization* or *triangularization* since the process leads to an upper triangular matrix. What remains to do is a so called *back-substitution* process. We substitute the value of the last unknown into the previous equation, which allows to calculate the value of the last but one unknown. The same way we continue upward until all unknowns are calculated. It is recommended to start the study of Gauss elimination by means of a pencil and a piece of paper as shown in the Paragraph 5.2.

The previous story, expressed in an algorithmic pseudocode, is presented in Template 1. It is assumed that the initial values of matrix elements are rewritten, thus the initial matrix is lost. No pivoting is assumed.

Template 1, Gauss elimination, standard matrix storage mode

```
a) forward substitution
 for k = 1 : n-1
   for i = k + 1 : n
     for  j = k + 1 : n
       a(i,j) = a(i,j) - a(k,j) * a(i,k) / a(k,k);  % matrix factorization
     end
     b(i) = b(i) - b(k) * a(i,k) / a(k,k);          % reduction of right-hand side
   end
 end

b) backward substitution
 for i = n : 1
   sum = 0;
   for j = i + 1 : n
     sum = sum + a(i,j) * x(j);
   end
   x(i) = (b(i) - sum) / a(i,i);
 end
```

To understand how the statements appearing in the Template 1 were conceived one could study their detailed derivation in the Paragraph 5.3.

When programming the Gauss elimination process it is recommended to split it into two separate parts which could be called independently. The first part should deal with the *matrix triangularization*[1], while the second part is to be responsible for the *reduction of the right-hand side* and the back substitution. This is advantageous when solving the system for more right-hand sides with an unchanging coefficient matrix since the operation count for the matrix triangularization is much greater than that of the second part of the code.

It is obvious that the algorithm should be complemented by adding a check against the division by a 'small' or zero pivot `a(k,k)` which is not, however, the diagonal element of the original matrix, since the element values are constantly being rewritten. The appearance of a small or zero pivot at k-th step of elimination causes that the elimination process cannot go on, it does not, however, mean that the matrix is necessarily singular. In some cases this obstacle can be overcome by exchanging the order of remaining rows. As a rule we are looking for such a row - whose element under the small or zero pivot - has the largest absolute value. This process is called a *partial pivoting* and requires to find a new pivot by $newpivot = \max\limits_{i=k,k+1,\dots,n} |a_{ik}|$. The *full pivoting* means that a candidate for the new pivot is chosen from all the remaining matrix elements, i.e. $newpivot = \max|a_{ij}|$ for $i = k, k+1, \dots, n$ and $j = k, k+1, \dots, n$. If none of pivoting succeeds, then we have to conclude that the matrix is singular within the computer precision being used. It should be reminded that

- partial pivoting spoils the banded structure of the matrix,
- full pivoting spoils the symmetry of the matrix,
- if the matrix is positive definite all the pivots are positive and no pivoting is needed.

For proofs see [41], [46].

The sparseness feature is very important since it allows to deal with matrices that otherwise could not be handled and processed due to computer memory limitations. In mechanical and civil engineering applications, the most common form of sparseness is bandedness, i.e. $a_{ij} = 0$ if $|i - j| >$ `nband`, where the identifier `nband` was introduced for the halfband width (including diagonal). For programming details see the Paragraph 5.9.

The work of solving a set of algebraic equations by Gauss elimination can be measured by the number of needed arithmetic operations. Considering one multiplication, or one division, or one addition plus one subtraction as one operation then the effort needed for the factorization of a matrix can be expressed by $\frac{1}{3}(n^3 - n)$ operations. The operations count for the right-hand side reduction is $\frac{1}{2}(n^2 - n)$ while the back substitution requires $\frac{1}{2}n(n + 1)$ operations. These counts apply to full matrices, they are, however, reduced approximately by half if the matrix is symmetric. If the matrix is symmetric and banded then the operations count for the matrix factorization is proportional to $n(nband)^2$ operations instead of n^3 as in the case of factorization of a general type matrix. See [45], [16].

Both symmetry and sparseness should be fruitfully exploited not only in programming but also in storage considerations if efficient storage requirements are to be achieved.

[1]Terms factorization and reduction are used as well.

There are many storage modes available, e.g. the *rectangular storage mode* for symmetric banded matrices [44], [23] or the *skyline storage mode* (also called *variable band* or *profile mode*) [4]. Special storage modes suitable *for random sparse matrices* are described in [48], [11]. The subject is treated in more detail in the Chapter 11.

The Basic Linear Algebra Software (BLAS) subroutines for performing standard matrix operations offer a plethora of storage schemes allowing for the efficient storage of matrices of different type – general, symmetric, banded, symmetric banded, etc. For more details see

```
http://www.netlib.org/blas/blast-forum/blas-report.pdf,
www.netlib.org/blas/blast-forum.
```

The huge memory problems require a disk storage approach to be employed. One of many possibilities is the use a so called *band algorithm* whose detailed description can be found in [34]. The programming considerations are in [33]. See also the paragraph 5.15.1.

Also a so called *hypermatrix algorithm* is of interest. The method consists in subdividing the coefficient matrix into smaller submatrices, sometimes called blocks. The Gauss elimination process is then defined not for matrix entries, but for submatrices. So the intermost loop could look like $\mathbf{K}^*_{ij} = \mathbf{K}_{ij} - \mathbf{K}^{\mathrm{T}}_{is}\,\mathbf{K}^{-1}_{ss}\,\mathbf{K}_{sj}$. The method is not limited to symmetric or band matrices. Since it is based on the sequence of matrix operations, standard library routines could easily be employed for its implementation. For details see [17].

The *frontal method* is a variant of Gauss elimination It was first publicly explained by Irons [24]. Today, it bears his name and is closely related to finite element technology. J.K. Reid [40] claims, however, that the method was used in computer programs already in the early sixties. The frontal method is based on the observation that assembling the (i, j)-th element of the e-th local stiffness matrix $\mathbf{k}^{(e)}_{ij}$ into the global stiffness matrix $\mathbf{K}_{ij} = \mathbf{K}_{ij} + \mathbf{k}^{(e)}_{ij}$ need not be completed before the elimination step, i.e. $\mathbf{K}_{ij} = \mathbf{K}_{ij} - \mathbf{K}_{ik}\,\mathbf{K}_{kj}\,/\,\mathbf{K}_{kk}$. The detailed explanation and the programming considerations concerning the frontal method are in the Chapter 10.

3.2.1.2 Gauss-Jordan elimination method

This method for solving a set of algebraic equations seems to be very efficient since the elimination of equations is conceived in such a way that the process leads not to a triangular matrix but to a diagonal one. It should be noted that the method is expensive, the operation count is approximately by 50 higher than that of Gauss elimination. See [39]. Furthermore, if the coefficient matrix is banded, then – before it becomes diagonal due to the elimination process – it is subjected to a massive fill-in process in the out-of-band space. Naturally, it prevents to employ any efficient storage mode.

3.2.1.3 Cholesky decomposition

This method is based on the decomposition of the original matrix $\mathbf{A}$ into a product of lower triangular and strictly upper triangular matrices, i.e. $\mathbf{A} \to \mathbf{LU}$. The process is simplified if the matrix is symmetric, then the decomposition becomes $\mathbf{A} \to \mathbf{R}^{\mathrm{T}}\,\mathbf{R}$, where $\mathbf{R}$ matrix is upper triangular. So instead of $\mathbf{Ax} = \mathbf{b}$ we solve $\mathbf{R}^{\mathrm{T}}\mathbf{Rx} = \mathbf{b}$. Denoting $\mathbf{Rx} = \mathbf{y}$ we get two equations $\mathbf{R}^{\mathrm{T}}\mathbf{y} = \mathbf{b}$ and $\mathbf{Rx} = \mathbf{y}$ from which $\mathbf{y}$ and $\mathbf{x}$

could be successively found. In the case of a positive definite matrix, no pivoting is needed. The operation count is roughly the same as that for Gauss elimination. See [39]. The implementation details are disclosed in the Paragraph 5.13.

Scalar versus parallel approaches - programming considerations

So far we have considered the matrix types and storage modes. In the last time there is another important issue to be considered, namely a decision whether the programming module is intended to be run on a scalar, parallel or vector machine. Until recently the programming languages lacked the vocabulary and syntax for specifying parallelism allowing a coordination of flows of instructions and data with hardware resources. Presently, there are no generally valid hints available, the problem is highly language and machine dependent. The Fortran 90, a successor of a long line of Fortran dialects, seems to be an efficient tool for parallelizing the tasks which are computational intensive in matrix processing. In Template 2 there are shown the differences in Fortran 77 and 90 syntax when applied to Gauss factorization of a matrix.

Template 2, Fortran 77 and 90 codes providing the factorization of a matrix

```
Fortran 77                            Fortran 90

DO k = 1, n - 1                       DO k = 1, n-1
  DO i = k + 1, n                        FORALL (i = k + 1:n)
    a(i,k) = a(i,k) / a(k,k)               a(i,k) = a(i,k) / a(k,k)
  END DO                                 END FORALL
  DO i = k + 1, n                        FORALL (i = k+1:n, j = k+1:n)
    DO j = k + 1, n
     a(i,j) = a(i,j) - a(i,k)*a(k,j)       a(i,j) = a(i,j) - a(i,k)*a(k,j)
    END DO
  END DO                                 END FORALL
END DO                                 END DO
```

On the left-hand side of the Template 2 you can observe classical Fortran 77 syntax which is scalar (sequential) in nature. A scalar instruction is such that completes before the next one starts. If this code is submitted to Fortran F90 compiler, it remains executable but the compiler does not recognize it as parallelizable. The code runs serially with no parallel speed-up. On the right-hand side there is the same process expressed in Fortran 90 style. Although FORALL structures serve the same purpose as DO loops, they are actually *parallel assignment statements*, not *loops*, and can be run in parallel using as many processors as available. See [49]. It should be made clear that FORALL statement by itself is not enough to achieve the parallel execution. In order to ensure any improvement in performance through the parallel execution, the FORALL structure must operate on arrays that have been properly treated by the DISTRIBUTE directive. This, however, is in hands of the numerical analyst who should be able to recognize the data structure of the problem and assign a proper programming data structure to it. For details see [25].

Vector versus parallel approaches - programming considerations

Sometimes the vector computers are mistakenly considered as the opposites of parallel computers. Actually the vectorization is a form of parallel processing allowing the array elements to be processed by groups. The automatic vectorization of the code secured by vector machine compilers could result in a substantial reduction in computational time. Further reduction could be achieved by a fruitful usage of compiler directives and by a 'manual' loop unrolling. See [13]. Presently, the level of automatic vectorization is substantially higher than the level of automatic parallelization. There is no rivalry between parallel and vector approaches. The future probably belongs to multiprocessor machines (each having hundreds of processors) with huge local memories and with fast and wide communication channels between processors. Today, the impact of vectorization can be observed even on personal computers. See [20].

3.2.2 Iterative stationary methods

These methods for the solution $\mathbf{Ax} = \mathbf{b}$ of are based on the idea of solving an equivalent system of equations $\mathbf{Mx} = (\mathbf{M}-\mathbf{A})\mathbf{x}+\mathbf{b}$. The equivalent system is solved iteratively by successive substitutions using the following scheme $\mathbf{M}\,{}^{(k+1)}\mathbf{x} = (\mathbf{M}-\mathbf{A})\,{}^{(k)}\mathbf{x} + \mathbf{b}$. There are many ways how to choose the $\mathbf{M}$ matrix. The choice $\mathbf{M} = \mathbf{A}$ converges immediately and leads to direct methods. Another simple choice $\mathbf{M} = \mathbf{I}$ leads to a method sometimes called the method of *successive substitution*. See the Paragraph 5.14.1. If we take $\mathbf{M}$ as a diagonal part of $\mathbf{A}$ then we have a so called *Jacobi method*. Taking $\mathbf{M}$ as a triangular part of $\mathbf{A}$ gives a so called *Gauss-Seidel method* and taking $\mathbf{M}$ as a combination of two previous choices leads to the method of *successive overrelaxation*. Generally, all these approaches could be expressed by ${}^{(k)}\mathbf{x} = \mathbf{B}\,{}^{(k-1)}\mathbf{x}+\mathbf{c}$ where neither $\mathbf{B}$ (obtained by a suitable process from $\mathbf{A}$) nor $\mathbf{c}$ (obtained from $\mathbf{b}$) depend on the iteration count k. For more details and for discussions on convergence rates see [2], [46].

Jacobi method

The Jacobi method is a representative of one of the oldest iteration methods for the solution of algebraic equations. It is known that its rate of convergence is quite slow, compared to other methods, the method is, however, mentioned here due to its simple structure which is transparently suitable for parallelizing. For convergence considerations see [39], [45].

The method is based on examining each of the n equations, assuming that the remaining elements are fixed, i.e. $x_i = (b_i - \sum\limits_{j\neq i} a_{ij}\, x_j)/a_{ii}$. It should be noted that a regular matrix can always be rearranged in such a way that there it has no zero diagonal entries. The iterative scheme is given by ${}^{(k)}x_i = (b_i - \sum\limits_{j\neq i} a_{ij}\, {}^{(k-1)}x_j)/a_{ii}$.

We stated that the parallelization is language dependent. Not only that, it also strongly depends of the memory model of the computer being used. The large-scale, massively parallel computers have up to thousands of processors, each having is own memory. Smaller scalable machines have only a few processors and a shared memory. The problem, that has not changed much in the last years, is that it takes far longer to distribute data than it does to do the computation. For more details see the Paragraph 5.14.2.

Gauss-Seidel method

The Gauss-Seidel method is similar to Jacobi method, it uses, however, the updated values of unknowns as soon as they are available. G. Strang claims that the method was apparently unknown to Gauss and not recommended by Seidel. [45]. The algorithm is in [2]. The rate of convergence of the Gauss-Seidel method is better than that of Jacobi, still it is relatively slow. It also strongly depends on the ordering of equations, and on the 'closeness' of an initial guess. See [2]. The Gauss-Seidel seems to be a fully sequential method. A careful analysis have shown that a high degree of parallelism is available if the method is applied to sparse matrices arising from the discretized partial differential equations. See [1] and the Paragraph 5.14.3.

Successive overrelaxation method (SOR)

This method is derived from the Gauss-Seidel method by introducing a relaxation parameter for increasing the rate of convergence. For the optimum choice of the relaxation parameter the method is faster than Gauss-Seidel by an order of magnitude. For details see [2], [33].

3.2.3 Iterative nonstationary methods

These methods are characterized by the fact that the computations strongly depend on information based on values at the current iteration step.

3.2.3.1 Conjugate gradient (CG) method

The CG method generates a sequence of conjugate (orthogonal) vectors which are residuals of iterates. They are constructed in such a way that they minimize the quadratic potential which is an equivalent to the solving the system of algebraic equations. The idea is appealing to mechanical engineers who are deeply aware of the fact that the nature acts so as to minimize energy, i.e. that the potential energy $\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{b}$ of a structure assumes a minimum at a configuration defined by $\mathbf{x}$, where the structure is in equilibrium. By other words, the minimization of potential energy leads to equilibrium $\mathbf{A}\mathbf{x} = \mathbf{b}$. The method is especially effective for symmetric, positive definite matrices. The iterates are updated at each step along a direction ${}^{(i)}\mathbf{d}$ at a distance ${}^{(i)}\alpha$ by ${}^{(i)}\mathbf{x} = {}^{(i-1)}\mathbf{x} + {}^{(i)}\alpha\,{}^{(i)}\mathbf{d}$. The direction and the distance are computed to minimize the residuals of ${}^{(i)}\mathbf{r} = \mathbf{b} - \mathbf{A}\,{}^{(i)}\mathbf{x}$. The direction vectors are mutually orthogonal and satisfy the relation ${}^{(i)}\mathbf{d}^{\mathrm{T}}\,\mathbf{A}\,{}^{(j)}\mathbf{d} = 0$ for $i \neq j$ and ${}^{(i)}\mathbf{d} = {}^{(i)}\mathbf{r} + {}^{(i-1)}\beta\,{}^{(i-1)}\mathbf{d}$, The vector of residuals ${}^{(i)}\mathbf{r}$ is updated by ${}^{(i)}\mathbf{r} = {}^{(i-1)}\mathbf{r} - {}^{(i)}\alpha\,{}^{(i)}\mathbf{q}$, where ${}^{(i)}\mathbf{q} = \mathbf{A}\,{}^{(i)}\mathbf{d}$. The step length parameter is determined by ${}^{(i)}\alpha = {}^{(i)}\mathbf{r}^{\mathrm{T}}\,{}^{(i)}\mathbf{r}/({}^{(i)}\mathbf{d}^{\mathrm{T}}\,\mathbf{A}\,{}^{(i)}\mathbf{d})$ which minimizes the expression ${}^{(i)}\mathbf{r}^{\mathrm{T}}\,\mathbf{A}^{-1}\,{}^{(i)}\mathbf{r}$. The parameter ${}^{(i)}\beta = {}^{(i)}\mathbf{r}^{\mathrm{T}}\,{}^{(i)}\mathbf{r}/({}^{(i-1)}\mathbf{r}^{\mathrm{T}}\,{}^{(i-1)}\mathbf{r})$ guaranties that ${}^{(i)}\mathbf{r}$ and ${}^{(i-1)}\mathbf{r}$ are orthogonal. The method is often combined with a suitable preconditioning as described in the text that follows. The excellent introduction to Conjugate gradient method, titled *An introduction to conjugate gradient method without agonizing pain* can be downloaded from

```
http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf
```

The Preconditioned Conjugate Gradient Method for solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ is adapted from

```
netlib2.es.utk.edu in linalg/templates
```

shown in Template 3.

Template 3, Preconditioned Conjugate Gradient Method

```
Compute ${}^{(0)}\mathbf{r} = \mathbf{b} - \mathbf{A}\,{}^{(0)}\mathbf{x}$ for an initial guess ${}^{(0)}\mathbf{x}$.
Assume that a symmetric, positive definite preconditioner $\mathbf{M}$ is known.
for i = 1,2, ...
  solve $\mathbf{M}\,{}^{(i-1)}\mathbf{z} = {}^{(i-1)}\,\mathbf{r}$
  ${}^{(i-1)}\rho = {}^{(i-1)}\mathbf{r}^{\mathrm{T}}\,{}^{(i-1)}\mathbf{z}$
  if i = 1 then
    ${}^{(1)}\mathbf{d} = {}^{(0)}\mathbf{z}$
    else
    ${}^{(i-1)}\beta = {}^{(i-1)}\rho\,/\,{}^{(i-2)}\rho$
    ${}^{(i)}\mathbf{d} = {}^{(i-1)}\mathbf{z} + {}^{(i-1)}\beta\,{}^{(i-1)}\mathbf{d}$
  endif
  ${}^{(i)}\mathbf{q} = \mathbf{A}\,{}^{(i)}\mathbf{d}$
  ${}^{(i)}\alpha = {}^{(i-1)}\rho\,/\,({}^{(i)}\mathbf{d}^{\mathrm{T}}\,{}^{(i)}\mathbf{q})$
  ${}^{(i)}\mathbf{x} = {}^{(i-1)}\mathbf{x} + {}^{(i)}\alpha\,{}^{(i)}\mathbf{d}$
  ${}^{(i)}\mathbf{r} = {}^{(i-1)}\mathbf{r} + {}^{(i)}\alpha\,{}^{(i)}\mathbf{q}$
  check convergence; continue if necessary
end
```

The method of conjugate gradients is a direct (finite) method rather than iterative, it can however be stopped part way. In linear cases its operation count exceeds that of direct solvers unless a good estimation of the solution as a starting guess is known. Its strength lies in solutions of large sparse systems stemming from nonliner cases where we proceed by iterations and where the result of a previous step is a good starting estimate for the next one. Convergence rates of iterative methods are difficult to estimate. As in the case of other methods they can be related to the condition number. See [46].

There are other representatives of nonstationary methods to be mentioned here. The Minimum Residual (MINRES) and Symmetric LQ (LQMMLQ) methods are variants of conjugate gradient method for symmetric indefinite matrices. Another approach is a so called Generalized Minimal Residual (GMRES) method which combines the computation of a sequence of orthogonal vectors with least square solve and update. This method can be applied to general nonsymmetric matrices. [2], [42].

A few words about preconditioners

The convergence rate of iterative methods depends on spectral properties of the coefficient matrix. One way to improve it, is to transform the initial system $\mathbf{Ax} = \mathbf{b}$ into another system, say $\mathbf{M}^{-1}\,\mathbf{Ax} = \mathbf{M}^{-1}\,\mathbf{b}$, where the matrix $\mathbf{M}$ is a so-called *preconditioner*. The matrix $\mathbf{M}$ is chosen in such a way that the spectral properties of $\mathbf{M}^{-1}\,\mathbf{A}$ are better than those of $\mathbf{A}$ and, at the same time, the inverse of $\mathbf{M}$ is not expensive and does not spoil the symmetry and positive definiteness of the original coefficient matrix. The Jacobi preconditioner, the simplest of other known approaches, consists of diagonal elements of the original coefficient matrix and is thus easy to implement and calculate. For more details see [35]. An incomplete Cholesky elimination is another known precoditioner. See [2], [42].

3.2.3.2 Multigrid method

The trick is based on the idea of solving the problem on a fine and coarse grids simultaneously. Starting from a fine grid with an approximation, say ${}^{(\mathrm{F})}\mathbf{x}$, we can proceed the following way, indicated in Template 4.

Template 4, The multigrid idea

```
Assume that a starting guess ${}^{(F)}\mathbf{x}$ is known
while not satisfied do

  ${}^{(F)}\mathbf{r} = \mathbf{b} - {}^{(F)}\mathbf{A}\,{}^{(F)}\mathbf{x}$           residual error
  ${}^{(C)}\mathbf{r} = \mathbf{C}\,{}^{(F)}\mathbf{x}$                    transfer to the coarse grid
  ${}^{(C)}\mathbf{A}\,{}^{(C)}\mathbf{y} = {}^{(C)}\mathbf{x}$                solve for ${}^{(C)}\mathbf{y}$ on the coarse grid data
  ${}^{(F)}\mathbf{y} = \mathbf{F}\,{}^{(C)}\mathbf{y}$                    transfer to the fine grid
  ${}^{(F)}\mathbf{x}^{\mathrm{new}} = {}^{(F)}\mathbf{x} + {}^{(F)}\mathbf{y}$           new approximation
  ${}^{(F)}\mathbf{x} = {}^{(F)}\mathbf{x}^{\mathrm{new}}$                    update the guess
  are you satisfied?        check convergence

end
```

The matrix $\mathbf{C}$ is a coarsening operator which combines values on a fine mesh and gives those on a coarse one. Conversely, the refining operator $\mathbf{F}$ yields values on a fine grid from those on a coarse one. See [29], [9]. The full text of the last book could be downloaded from

`https://computation.llnl.gov/casc/people/henson/mgtut/welcome.html.`

3.2.4 Convergence criteria for iterative methods

Denoting the iteration counter by `it`, the maximum permissible number of iterations by `itmax`, the required tolerance by `ep`, the current residuum by $\mathbf{r}$ and the right-hand side vector by $\mathbf{b}$, the criteria of convergence could be implemented, as in Template 5, by a `do while` or `repeat until` programming structures followingly.

Template 5, Convergence criteria

```
it = 0;  itmax = ...  ;  ep = ...  ;

One can use do while          or          repeat until programming structure
while $\|\mathbf{r}\|/\|\mathbf{b}\|$ > ep or it < itmax do     repeat
  it = it + 1;                              it = it + 1;
  iterate;                                  iterate;
end                                       until $\|\mathbf{r}\|/\|\mathbf{b}\|$ < ep and it > itmax
```

Instead of the ratio of norms indicated above, which is a convergence measure expressed in "forces" acting on the considered structure, we could also use the ratio of "displacement" norms $(|\| \mathbf{x}^{\mathrm{new}} \| - \| \mathbf{x}^{\mathrm{old}} \||)/ \| \mathbf{x}^{\mathrm{old}} \|$. The value of `ep` must be reasonably small with respect to the level of values appearing in the problem and with respect to the unit round-off error (machine epsilon) that is described in following paragraphs.

3.2.5 Precision, or do we believe in our results?

The smallest positive *real* number eps (real in the computer jargon) the computer can distinguish from the unit, i.e. for which `1 + eps > 1` is known as the *unit round-off error* or *machine epsilon* and is computer and language dependent. The value of machine epsilon plays crucial role in the assessment of numerical accuracy of the computational process. It can be easily determined by a piece of code shown in the Template 6.

Template 6, Machine epsilon and how to get it

```
eps = 1; i = 0;
while eps + 1 > 1 do
  i = i + 1;
  eps = 0.5 * eps;
end
```

The final `eps` reveals the value of machine epsilon while the final `i` tells the number of bits for the binary representation of a mantissa of the real number for a particular computer since the division by two is equivalent to one binary shift to the right. See [16].

It is known [41] that when solving $\mathbf{Ax} = \mathbf{b}$, the relative error of the solution is bounded by the relative error of the coefficient matrix multiplied by the value of the condition number, i.e. $\| \Delta\mathbf{x} \| \,/\, \| \mathbf{x} \| \leq c(\mathbf{A}) \| \Delta\mathbf{A} \| \,/\, \| \mathbf{A} \|$, where $c(\mathbf{A})$, the condition number of the matrix, is defined by $c(\mathbf{A}) = \| \mathbf{A^{-1}} \| \| \mathbf{A} \|$ or by a ratio of maximum and minimum eigenvalues of the coefficient matrix, i.e. $c(\mathbf{A}) = \lambda_{\max}/\lambda_{\min}$ in case that the coefficient matrix is positive definite. Assuming that the computer works with t decimal digits, i.e. $\| \Delta\mathbf{A} \| \,/\, \| \mathbf{A} \| = 10^{-t}$ and that the solution is known with s decimal digits, i.e. $\| \Delta\mathbf{x} \| \,/\, \| \mathbf{x} \| = 10^{-s}$, we can easily derive the approximate formula for the number of significant decimal digits of the solution in the form $s \geq t - \log_{10} \mathrm{c}(\mathbf{A}))$. By other words we are losing $\log_{10}(c(\mathbf{A}))$ decimal digits due to round-off errors. The formula is easy, the computation of the value of the condition number by definitions given above, is expensive. There are many approaches leading to a cheaper evaluation of the condition number. One of them is almost free of charge and can be explained easily. Let $\overline{\mathbf{x}}$ is a computer solution of $\mathbf{Ax} = \mathbf{b}$. It is obvious that $\overline{\mathbf{x}} = \mathbf{A^{-1}}\,\mathbf{b}$. For norms we have $\| \overline{\mathbf{x}} \| \leq \| \mathbf{A^{-1}} \| \| \mathbf{b} \|$ which gives a rough estimate of the condition number in the form $c(\mathbf{A}) \leq \| \overline{\mathbf{x}} \| \| \mathbf{A} \| \,/\, \| \mathbf{b} \|$. For more details see [41], [16].

It is the large value of the condition number which declares that the coefficient matrix is ill-conditioned and signals the problems we will have solving the problem and interpreting results. Generally, the result of the solution of a system of algebraic equation with an ill-conditioned matrix is extremely sensitive to small changes of input coefficients. On the other hand a small value of the determinant of the coefficient matrix does not necessarily mean that the matrix is 'nearly' singular. See [46].

3.2.6 Finite versus iterative solvers

The finite elimination methods gained their good reputation in the history of solid mechanics computing by their robustness and stability without pivoting. They are burdened by fill-in problems that cannot be avoided but only minimized by a suitable numbering of variables. Finite (direct) methods are the backbone of today's standard commercial finite element calculation.

The iterative methods do not suffer from fill-in, they can survive in almost the same storage as original data but their performance depends on the numerical values of data of the problem to be solved. Presently, the iterative methods are subjected to intensive study with the intention to develop adaptive and robust procedures which could be applied commercially.

The present state of affairs is summarized in the following table, which is adapted from [35].

Finite vs. iterative methods	**Finite**	**Iterative**
Solution of algebraic equations	exact	approximate
Performance depends on	non-zero structure	numerical values
Additional right-hand side	cheap	repeat all
Irregular geometries	no problem	slows down
Storage for large problems	fill-in problems	only original data
Current usage	commercial, standard use	academic, special projects

3.3 Generalized eigenvalue problem

There are many engineering tasks which lead to standard $\mathbf{Ax} = \lambda\mathbf{x}$ or to a generalized $(\mathbf{K} - \lambda\,\mathbf{M})\,\mathbf{x} = 0$ eigenvalue problem. As examples the elastic stability or vibration problems could be mentioned here. In latter case $\mathbf{K}$ is the stiffness matrix, $\mathbf{M}$ denotes the mass matrix. These matrices are usually symmetric and positive definite. The stiffness matrix is positive semidefinite if there are any rigid body modes present. The stiffness matrix could lose its symmetry if hydrodynamic forces in journal bearings are modelled. The mass matrix could lose its positive semidefiniteness if there is a missing mass in a node of the structure. The eigenvalue problem could also become semidefinite of indefinite also due to incorrectly applied underintegration process [4], [7].

3.3.1 Transformation methods

Transformation methods are based on a sequence of orthogonal transformations that lead to a diagonal or tridiagonal matrix. The resulting diagonal matrix has the eigenvalues stored directly on the diagonal, from a tridiagonal matrix they can be easily 'extracted'.

3.3.1.1 Jacobi method

Jacobi method, based on subsequent rotations, is schematically described in Template 7. It is shown here for a standard eigenvalue problem $(\mathbf{A} - \lambda\,\mathbf{I})\,\mathbf{x} = \mathbf{0}$. The transformation leads to so called *diagonalization* of the matrix defined by $\mathbf{A} = \mathbf{X}\,\mathbf{\Lambda}\,\mathbf{X}^{-1}$, where Λ is a diagonal matrix containing individual eigenvalues λ_i and $\mathbf{X}$ is a modal matrix containing corresponding eigenvectors. They are stored columnwise in the same order as eigenvalues in Λ.

Template 7, Jacobi method for a standard eigenvalue problem

```
$^{(0)}\mathbf{A} = \mathbf{A}$
Diagonalization of $\mathbf{A}$
for k = 1,2, ...
  $^{(k)}\mathbf{A} = {}^{(k)}\mathbf{T}^{\mathrm{T}}\ {}^{(k-1)}\mathbf{A}\ {}^{(k)}\mathbf{T}$
  check convergence
end
Eigenvectors
$^{(k)}\mathbf{X} = {}^{(1)}\mathbf{T}\ {}^{(2)}\mathbf{T} \ldots {}^{(k)}\mathbf{T}$
Note
For $k \to \infty$ the sequence leads to $^{(k)}\mathbf{A} \to \Lambda$ and $^{(k)}\mathbf{X} \to \mathbf{X}$
```

Notes

- Matrix Λ is diagonal, it contains all eigenvalues – generally, they are sorted neither in ascending nor in descending order.
- Matrix $\mathbf{X}$ is a so-called modal matrix. It contains all eigenvectors – columnwise. It is not symmetric. It is, however, orthogonal, if $\mathbf{A}$ matrix is symmetric.

The bandedness of matrix $\mathbf{A}$ cannot be exploited. There is a massive fill-in before the coefficient matrix finally becomes diagonal. See [43]. At each iteration the Jacobi method uses the transformation matrix $\mathbf{T}$ in the form shown in Template 8.

Template 8, The Jacobi transformation matrix for k-th iteration

```
The angle $\varphi$, obtained from the condition of zeroing the largest non-diagonal entry and its indices $p, q$, are known
The non-zero entries for this rotation are as follows
for i = 1 to n
  T(i,i) = 1
end
$c = \cos(\varphi); s = \sin(\varphi)$
T(p,p) = c; T(p,q) = s; T(q,p) = -s; T(q,q) = c;
```

The trigonometrical entries are in p-th and q-th columns and rows only, with ones on the diagonal and zeros everywhere else. In each iteration the procedure is based on such a choice of rotations that leads to zeroing the largest out of out-of-diagonal element. For details see [43]. The generalized Jacobi method could be conceived in such a way that the transformation of generalized eigenvalue problem into a standard form could be avoided. See [4].

3.3.1.2 The Givens method

is another possible variant of rotation approaches. It uses the same rotation matrix $\mathbf{T}$ as Jacobi. The rotation pairs p and q, needed for zeroing of element a_{jk}, are picked up sequentially, i.e. $p = j + 1$; $q = k$. In this case the transformation process leads to a symmetric tridiagonal matrix that offers an easy way to eigenvalue extraction. The procedure is carried out in a finite number of steps, unlike the Jacobi method which requires an iteration process to convergence.

3.3.1.3 Left-right transformation

is based on a sequence of similarity transformations leading to a diagonal matrix, i.e. ${}^{(0)}\mathbf{A} \leftarrow \mathbf{A}$; ${}^{(k)}\mathbf{A} \rightarrow {}^{(k)}\mathbf{R}^{\mathrm{T}}\,{}^{(k)}\mathbf{R}$; ${}^{(k+1)}\mathbf{A} \leftarrow {}^{(k)}\mathbf{R}\,{}^{(k)}\mathbf{R}^{\mathrm{T}}$; k=1,2,.... The indicated decomposition is due to Cholesky. The matrix $\mathbf{R}$ is upper triangular.

3.3.1.4 QR transformation

is based on the Gramm-Schmidt orthogonalization decomposition $\mathbf{A} \rightarrow \mathbf{QR}$. The matrix $\mathbf{Q}$ is orthogonal, the matrix $\mathbf{R}$ is upper triangular. With a starting guess ${}^{(0)}\mathbf{A} \leftarrow \mathbf{A}$ the procedure is given by the sequence of similarity transformations ${}^{(k)}\mathbf{A} \rightarrow {}^{(k)}\mathbf{Q}\,{}^{(k)}\mathbf{R}$; ${}^{(k+1)}\mathbf{A} \leftarrow {}^{(k)}\mathbf{R}\,{}^{(k)}\mathbf{Q}$ for $k = 0, 1, 2, ...$ As before for $k \rightarrow \infty$ the matrix ${}^{(k)}\mathbf{A}$ approaches the lower triangular matrix with eigenvalues on its diagonal.

3.3.1.5 QL method

is a fast efficient method for the solution of eigenvalue problems with tridiagonal matrices. The method is similar to the QR method, however the lower triangular $\mathbf{L}$ is used instead of $\mathbf{R}$. It is proved that QL method has a smaller round-off error than that of QR algorithm. See [38]. The routines based on QL method are usually available in standard computer libraries.

The above mentioned methods are sometimes combined together. For example in Nastran [27] the Givens method is used for making the matrix tridiagonal and QR transformation for extraction of chosen modes.

3.3.2 Forward and inverse power methods

These methods are iterative. Their principles could best be explained on a standard eigenvalue problem.

3.3.2.1 Forward iteration method

In this paragraph the upper left hand side index in brackets denotes the iteration counter, while the the upper right hand side index (in brackets again) points to a particular column of the matrix. In *forward iteration method* we proceed from an initial choice of ${}^{(0)}\mathbf{x}$ by ${}^{(k+1)}\mathbf{x} = \mathbf{A}\,{}^{(k)}\mathbf{x}$, for $k \rightarrow 1, 2, ...$ If the matrix $\mathbf{A}$ has distinct eigenvalues $|\lambda_1| < |\lambda_2| < \ldots < |\lambda_n|$ then it also has a full set of eigenvectors $\mathbf{u}^{(i)}$, $i = 1, 2...n$. The upper left-hand index is the iteration counter, the upper right-hand index points to a particular eigenvector. These vectors could be considered as base vectors and each vector (including the starting guess) could be expressed as their linear combination ${}^{(0)}\mathbf{x} = \sum_{i=1}^{n} c_i \mathbf{u}^{(i)}$. Assuming that $c_n \neq 0$ we we could express the k-th iteration in the form ${}^{(k)}\mathbf{x} = \lambda_n^k \left[\sum_{i=1}^{n-1} c_i (\lambda_i/\lambda_n)^k\, \mathbf{u}^{(i)} + c_n \mathbf{u}^{(n)}\right]$. Since λ_n is by assumption the largest eigenvalue, then $(\lambda_i/\lambda_n)^k \rightarrow 0$ for $k \rightarrow \infty$. The k-th iteration of vector ${}^{(k)}\mathbf{x}$ thus approaches the eigenvector $\mathbf{u}^{(n)}$ belonging to λ_n, i.e. ${}^{(k)}\mathbf{x} \rightarrow c_n \lambda_n^k \mathbf{u}^{(n)}$. Finally, the largest eigenvalue can be calculated by means of Rayleigh quotient. The method fails if $\lambda_i/\lambda_n = \pm 1$, or if the initial guess does not contain the direction corresponding to the eigenvector belonging to the largest eigenvalue, i.e. if $c_n = 0$. The convergence to the eigenvector is often very slow and depends on the ratio $|\lambda_{n-1}|/|\lambda_n|$. The next eigenvectors can be found from orthogonality conditions. One has to exclude from further considerations the directions corresponding to eigenvectors already found. To circumvent the numerical overflow the working vectors are to be normalized after each iteration. For details see [45].

3.3.2.2 Inverse iteration method

In *inverse iteration method* we are searching for the eigenvector belonging to largest eigenvalue of $\mathbf{A}^{-1}$ matrix, which is, however, the smallest eigenvalue of matrix $\mathbf{A}$. From the standard eigenvalue problem we easily get $\mathbf{A}^{-1}\,\mathbf{u}^{(i)} = \frac{1}{\lambda_i}\mathbf{u}^{(i)}$ with $\mathbf{B} = \mathbf{A}^{\mathbf{-1}}$ and $\kappa_i = \frac{1}{\lambda_i}$. Then we have $\mathbf{B}\,\mathbf{u}^{(i)} = \kappa_i\,\mathbf{u}^{(i)}$. The iteration process goes by ${}^{(k+1)}\mathbf{x} = \mathbf{A}^{-1}\,{}^{(k)}\mathbf{x}$ for $k = 0, 1, 2, ...$ The matrix $\mathbf{A}$ is not inverted, the set of algebraic equations $\mathbf{A}\,{}^{(k+1)}\mathbf{x} = {}^{(k)}\mathbf{x}$ for ${}^{(k+1)}\mathbf{x}$ is solved instead.

3.3.2.3 The shifted inverse power method

The shifted inverse power method for a given starting guess ${}^{(k)}\mathbf{x}$ solves consecutively the system of equations $(\mathbf{A} - \sigma\mathbf{I})\,{}^{(k+1)}\mathbf{x} = {}^{(k)}\mathbf{x}$ with a shift σ suitably chosen close to λ_1. All the spectrum is shifted by the same amount σ and the convergence factor is $|\lambda_1 - \sigma|/|\lambda_2 - \sigma|$. The better the choice of σ the more ill-conditioned is the coefficient matrix $(\mathbf{A} - \sigma\mathbf{I})$. Fortunately the error is mostly confined to 'length' of the eigenvector which is not dangerous since any multiple of an eigenvector is still an eigenvector.

3.3.2.4 The block power method

The block power method works with several vectors at once. An initial choice consists of, say s, orthogonal vectors instead of one as before. For keeping them orthogonal throughout the iteration process the Gramm-Schmidt orthogonalization process have to applied repeatedly. As a result we get the s largest eigenvalues and their eigenvectors.

3.3.2.5 The block inverse power method

The block inverse power method is a similar method. It works with the $\mathbf{A}^{-1}$ instead of $\mathbf{A}$ and yields the s smallest eigenpairs.

3.3.2.6 Subspace iteration method

Instead of solving the full n by n generalized eigenvalue problem this method aims at finding the lowest s eigenvalues and corresponding eigenvectors of a reduced problem with $\mathbf{K}_{n\times n}\,\mathbf{V}_{n\times s} = \mathbf{M}_{n\times n}\,\mathbf{V}_{n\times s}\,\mathbf{\Lambda}_{s\times s}$ where the matrix $\mathbf{V}$ contains s orthogonal vectors satisfying $\mathbf{V}^{\mathrm{T}}\,\mathbf{K}\,\mathbf{V} = \Lambda$ and $\mathbf{V}^{\mathrm{T}}\,\mathbf{M}\,\mathbf{V} = \mathbf{I}$. More details, including the programming considerations and the Fortran code can be found in [4].

3.3.3 Determinant search method

The standard eigenvalue problem $\mathbf{A}\mathbf{x} = \lambda\,\mathbf{x}$ can be written as $(\mathbf{A} - \lambda\,\mathbf{I})\mathbf{x} = \mathbf{0}$, which is a system of homogeneous equations having a nontrivial solution only if $\det(\mathbf{A} - \lambda\,\mathbf{I}) = \mathbf{0}$. Evaluating this determinant we get a so-called characteristic polynomial $p(\lambda) = c_n\lambda^n + c_{n-1}\lambda^{n-1} + \ldots + c_1\lambda + c_0 = 0$, whose order is the same as the order of the coefficient matrix. It is known that the roots of this polynomial are the eigenvalues we are looking for. A long time ago, in 1831, Évariste Galois established a general theorem stating the impossibility of finite algebraic formulation of solutions for polynomials of all degrees greater then 4. See [26]. In practice the eigenvalues are obtained not by numeric root finders but utilizing the Gauss decomposition $\mathbf{A} \rightarrow \mathbf{LU}$ and the fact that $p(\lambda) = \det\mathbf{A} = \Pi_{i=1}^{n}\,\lambda_i$. Details can be found in [4].

3.3.4 Lanczos method

The Lanczos method was introduced in 1950 [3]. The algorithm solves the standard n by n eigenvalue problem $\mathbf{A}\mathbf{x} = \lambda\,\mathbf{x}$ using a recursion leading to a similar eigenvalue problem $\mathbf{T}\mathbf{q} = \lambda\,\mathbf{q}$ with a tridiagonal matrix $\mathbf{T}$. Vectors ${}^{(i)}\mathbf{q}$ are columns of an orthonormal matrix $\mathbf{Q}$ defined in such a way that $\mathbf{Q}^{\mathrm{T}}\,\mathbf{Q} = \mathbf{I}$ and a similarity transformation $\mathbf{Q}^{\mathrm{T}}\,\mathbf{A}\,\mathbf{Q} = \mathbf{T}$ holds.

The recurrence formula, generating the sequence of vectors ${}^{(i)}\mathbf{q}$, could be derived by writing out the individual elements of equation $\mathbf{AQ} = \mathbf{QT}$ in full and yields $\mathbf{A}\,{}^{(i)}\mathbf{q} = \beta_{i-1}\,{}^{(i-1)}\mathbf{q} + \alpha_i\,{}^{(i)}\mathbf{q} + \beta_i\,{}^{(i+1)}\mathbf{q}$. The coefficients $\alpha_i = {}^{(i)}\mathbf{q}^{\mathrm{T}}\,\mathbf{A}\,{}^{(i)}\mathbf{q}$ are due to required orthogonality conditions. Then, the coefficients α_i and β_i are diagonal and codiagonal entries of the matrix $\mathbf{T}$ as indicated in Template 9.

Template 9, Appearance of Lanczos matrix

```
T(1,1) = $\alpha_1$; T(1,2) = $\beta_2$
for i = 1 to m - 1
  T(i,i-1) = $\beta_i$; T(i,i) = $\alpha_i$; T(i,i+1) = $\beta_{i+1}$
end
T(m,m-1) = $\beta_m$; T(m,m) = $\alpha_m$
```

The procedure was initially conceived for $m = n$ and it was seen as a direct method. The idea was that a tridiagonal eigenvalue problem is easier to solve than that with a full matrix. The Lanczos algorithm for a standard problem $\mathbf{Ax} = \lambda\mathbf{x}$ could formulated as indicated in Template 10.

Template 10, Lanczos method for a standard eigenvalue problem

```
Choose a normalized starting vector ${}^{(1)}\mathbf{q}$. Its norm is equal to one
Set $\beta_1 = 1$; ${}^{(0)}\mathbf{q} = \mathbf{0}$; $k = 0$
while not satisfied do
  k = k + 1
  ${}^{(k)}\alpha = {}^{(k)}\mathbf{q}^{\mathrm{T}}\ \mathbf{A}\ {}^{(k)}\mathbf{q}$
  ${}^{(k+1)}\mathbf{r} = \mathbf{A}\ {}^{(k)}\mathbf{q} - {}^{(k)}\alpha\ {}^{(k)}\mathbf{q} - {}^{(k)}\beta\ {}^{(k-1)}\mathbf{q}$
  ${}^{(k+1)}\beta = \|{}^{(k+1)}\mathbf{r}\|$
  ${}^{(k+1)}\mathbf{q} = {}^{(k+1)}\mathbf{r}\ /\ {}^{(k+1)}\beta$
  are you satisfied?
end of while
```

If the algorithm were carried out without round-off errors, all the generated vectors $\mathbf{q}$ would be orthonormal. In the finite precision arithmetics the orthogonality of vectors is seriously degraded due to round-off errors, so the presented mathematical approach is not sufficient for the practical implementation. More details for preserving orthogonality can be found in [35], [46].

Now, the method is considered to be semi-direct. It means that only a part of the matrix $\mathbf{T}$ and only a few $\mathbf{q}$'s are computed. In typical large-scale applications the order of the matrix $\mathbf{A}$ may be in tens of thousands while the order of $\mathbf{T}$ is of about twice the number of required eigenvalues, usually 20 to 30.

For the vibration related generalized eigenvalue problem the Lanczos algorithm must be modified. If the $\mathbf{Kx} = \lambda\,\mathbf{Mx}$ is transformed to a standard form $\mathbf{Ax} = \mathbf{b}$ by a process indicated above, then the Lanczos algorithm would yield approximations of the largest eigenvalues and eigenvectors which are uninteresting from the engineering point of view. Using a suitable shift σ (which is close to the eigenvalues of interest) and solving the inverted and shifted eigenvalue problem instead, we could proceed from the initial formulation $\mathbf{K}\,\mathbf{x} = \lambda\,\mathbf{M}\,\mathbf{x}$ to $\mathbf{K}\,\mathbf{x} = (\lambda - \sigma)\mathbf{M}\,\mathbf{x} + \sigma\mathbf{M}\,\mathbf{x}$. After some easy manipulation we get $(\mathbf{K} - \sigma\mathbf{M})^{-1}\,\mathbf{Mx} = \kappa\,\mathbf{x}$ with $\kappa = 1/(\lambda - \sigma)$. Applying the Lanczos approach we get $\mathbf{Tq} = \kappa\,\mathbf{q}$. The eigenvalues of the original problem are $\lambda = \sigma - 1/\kappa$. We could proceed the way outlined in Template 11.

Template 11, Lanczos method for a generalized inverted problem with a shift

```
$\beta_1 = 0; {}^{(0)}\mathbf{q} = \mathbf{0};$
i = 0;

${}^{(0)}\mathbf{r} = ...$                a starting vector
$\sigma = ...$                  a starting shift
$\mathbf{A} = (\mathbf{K} - \sigma\mathbf{M})^{-1}$   of course, we do not solve it by inversion
${}^{(1)}\mathbf{q} = \mathbf{AM}\,{}^{(0)}\mathbf{r}$         we use Gauss elimination instead
satisfied = false
while not satisfied do
   i = i + 1
   $\alpha_i = {}^{(i)}\mathbf{q}^{\mathrm{T}}\,\mathbf{MAM}\,{}^{(i)}\mathbf{q}$
   ${}^{(i+1)}\mathbf{r} = \mathbf{AM}\,{}^{(i)}\mathbf{q} - \alpha_i\,{}^{(i)}\mathbf{q} - \beta_i\,{}^{(i-1)}\mathbf{q}$
   $\beta_{i+1} = ({}^{(i)}\mathbf{r}^{\mathrm{T}}\,\mathbf{M}\,{}^{(i)}\mathbf{r})^{1/2}$
   ${}^{(i+1)}\mathbf{q} = {}^{(i+1)}\mathbf{r}/\beta_{i+1}$
   are you satisfied?
end
```

Each of these methods has advantages for certain tasks and disadvantages for others. The Jacobi or QR methods solve the complete system of equations. For very large systems these methods prove to be inefficient and the Lanczos method is gaining the growing acceptance as a basic tool for solving large eigenvalue problems especially when only a few eigenvalues are needed.

3.4 Solution of ordinary differential equations

For an undamped system in the current configuration C at time t, we get the equations of motion in the form resulting from the finite element semidiscretization $\mathbf{M}\,\ddot{\mathbf{q}} = \mathbf{F}^{\mathrm{res}}$ where $\mathbf{M}$ is the mass matrix, $\ddot{\mathbf{q}}$ represents nodal acceleration vector, the residual vector is $\mathbf{F}^{\mathrm{res}} = \mathbf{F}^{\mathrm{ext}} - \mathbf{F}^{\mathrm{int}}$, $\mathbf{F}^{\mathrm{ext}}$ is vector of externally applied nodal forces.

The internal nodal forces corresponding to element stresses are $\mathbf{F}^{\mathrm{int}} = \sum_e \int_V \mathbf{B}^{\mathrm{T}}\,\boldsymbol{\sigma}\,\mathrm{d}V$, where $\sum_e$ represents assemble operations taken over all the elements, the strain-displacement matrix is denoted $\mathbf{B}$ and $\boldsymbol{\sigma} = \{\sigma_{11}\,\sigma_{22}\,\sigma_{33}\,\sigma_{12}\,\sigma_{23}\,\sigma_{31}\}^{\mathrm{T}}$ is a vector representation of stress tensor. Finally, the variable V stands for the element volume at current configuration.

Details can be found in many finite element textbooks e.g. [3], [21], [12], [52].

The aim of computational procedures used for the solution of transient problems is to satisfy the equation of motion, not continually, but at discrete time intervals only. It is assumed that in the considered time span $< 0, t_{\max} >$ all the discretized quantities at times 0, Δt, $2\Delta t$, $3\Delta t$, ..., t are known, while the quantities at times $t + \Delta t$, ... $t_{\max}$ are to be found. The quantity Δt, being the time step, need not necessarily be constant throughout the integration process.

Time integration methods can be broadly characterized as explicit or implicit.

3.4.1 Explicit time integration algorithms

Explicit formulations follow from the equations of motion written at time t. Corresponding quantities are denoted with a left superscript as in $\mathbf{M}\,{}^t\ddot{\mathbf{q}} = {}^t\mathbf{F}^{\mathrm{res}}$.

The internal forces can alternatively be written as ${}^t\mathbf{F}^{\mathrm{int}} = {}^t\mathbf{D}\,{}^t\dot{\mathbf{q}} + {}^t\mathbf{K}\,{}^t\mathbf{q}$; $\mathbf{M}$, ${}^t\mathbf{D}$ and ${}^t\mathbf{K}$ are the mass, viscous damping and stiffness matrices, respectively and ${}^t\ddot{\mathbf{q}}$, ${}^t\dot{\mathbf{q}}$ and ${}^t\mathbf{q}$ are nodal accelerations, velocities and displacements. In structural solid mechanics problems the mass matrix $\mathbf{M}$ does not change with time.

A classical representative of the explicit time integration algorithm is the central difference scheme.

If the transient problem is linear, then the stiffness and damping matrices are constant as well. Substituting the central difference approximations for velocities and accelerations ${}^t\dot{\mathbf{q}} = \left({}^{t+\Delta t}\mathbf{q} - {}^{t-\Delta t}\mathbf{q}\right)/(2\Delta t)$, ${}^t\ddot{\mathbf{q}} = \left({}^{t+\Delta t}\mathbf{q} - 2\,{}^t\mathbf{q} + {}^{t-\Delta t}\mathbf{q}\right)/\Delta t^2$, into the equations of motion written at time t we get a system of algebraic equations from which we can solve for displacements at time $t + \Delta t$ namely $\mathbf{M}^{\mathrm{eff}}\,{}^{t+\Delta t}\mathbf{q} = \mathbf{F}^{\mathrm{eff}}$ where so called effective quantities are

$$\mathbf{M}^{\mathrm{eff}} = \mathbf{M}/\Delta t^2 + \mathbf{D}/(2\,\Delta t),$$
$$\mathbf{F}^{\mathrm{eff}} = {}^t\mathbf{F}^{\mathrm{ext}} - \left(\mathbf{K} - 2\mathbf{M}/\Delta t^2\right)\,{}^t\mathbf{q} - \left(\mathbf{M}/\Delta t^2 - \mathbf{D}/(2\Delta t)\right)\,{}^{t-\Delta t}\mathbf{q}.$$

The last term in the expression for the effective force indicates that the process is not self-starting.

The process is explicit only if the mass matrix is made diagonal by a suitable lumping process. The damping matrix needs to be diagonal as well. The inversion of $\mathbf{M}^{\mathrm{eff}}$ is then trivial and instead of a matrix equation we simply have the set of individual equations for each degree of freedom and no matrix solver is needed.

A comprehensive survey of explicit time integration methods for dynamic analysis of linear and nonlinear structures can be found in [15]. A thorough analysis of transient algorithms together with a rich source of references was presented by Hughes in [6].

Stability analysis of explicit and implicit schemes has been studied for a long time. Park [36] has investigated stability limits and stability regions for both linear and nonlinear systems. A comprehensive survey showing a variety of approaches is presented by Hughes in [6]. See also [3].

The explicit methods are only conditionally stable; the stability limit is approximately equal to the time for an elastic wave to transverse the smallest element. The critical time step securing the stability of the central difference method for a linear undamped system is (see [26]) $\Delta t_{cr} = 2/\omega_{\max}$, $\omega_{\max}$ being the maximum eigenfrequency, related to the maximum eigenvalue $\lambda_{\max}$ of the generalized eigenvalue problem $\mathbf{K}\,\mathbf{q} = \lambda\,\mathbf{M}\,\mathbf{q}$ by $\omega^2 = \lambda$. Practical calculations show that the result is also applicable to nonlinear cases, since each time step in nonlinear response can roughly be considered as a linear increment of the whole solution.

Explicit time integration methods are employed mostly for the solution of nonlinear problems, since the implementing of complex physical phenomena and constitutive equations is then relatively easy. The stiffness matrix need not be assembled so that no matrix solver is required, which saves computer storage and time. The main disadvantage is the conditional stability which clearly manifests itself in linear problems, where the solution quickly blows up if the time step is larger than the critical one. In nonlinear problems results calculated with a ’wrong’ step could contain a significant error and may not show immediate instability. See Program 3.

Program 3

```
function [disn,veln,accn] = cedif(dis,diss,vel,acc,xm,xmt,xk,xc,p,h)
% central difference method
% dis,vel,acc .............. displacements, velocities, accelerations
%                            at the beginning of time step ... at time t
% diss ..................... displacements at time t - h
% disn,veln,accn ........... corresponding quantities at the end
%                            of time step ... at tine t + h
% xm ....................... mass matrix
% xmt ...................... effective mass matrix
% xk ....................... stiffness matrix
% xc ....................... damping matrix
% p ........................ loading vector at the end of time step
% h ........................ time step

% constants
a0 = 1/(h*h);
a1 = 1/(2*h);
a2 = 2*a0;
a3 = 1/a2;

% effective loading vector
r = p - (xk - a2*xm)*dis - (a0*xm - a1*xc)*diss;
% solve system of equations for displacements using lu decomposition of xmt
disn=xmt\r;

% new velocities and accelerations
accn = a0*(diss - 2*dis + disn);
veln = a1*(-diss + disn);
% end of cedif
```

□ End of Program 3.

3.4.2 Implicit time integration algorithms

The implicit formulations stem from equations of motion written at time $t + \Delta t$; unknown quantities are implicitly embedded in the formulation and the system of algebraic equations must be solved to 'free' them. In structural dynamic problems implicit integration schemes give acceptable solutions with time steps usually one or two orders of magnitude larger than the stability limit of explicit methods.

Perhaps the most frequently used implicit methods belong to the so called Newmark family. The Newmark integration scheme is based upon an extension of the linear acceleration method, in which it is assumed that the accelerations vary linearly within a time step.

The Newmark method consists of the following equations [30]

$$\mathbf{M}\,{}^{t+\Delta t}\ddot{\mathbf{q}} + {}^{t+\Delta t}\mathbf{D}\,{}^{t+\Delta t}\dot{\mathbf{q}} + {}^{t+\Delta t}\mathbf{K}\,{}^{t+\Delta t}\mathbf{q} = {}^{t+\Delta t}\mathbf{F}^{\mathrm{ext}},$$

$${}^{t+\Delta t}\mathbf{q} = {}^{t}\mathbf{q} + \Delta t\,{}^{t}\dot{\mathbf{q}} + \frac{1}{2}\,\Delta t^2\,\left((1-2\beta)\,{}^{t}\ddot{\mathbf{q}} + 2\beta\,{}^{t+\Delta t}\ddot{\mathbf{q}}\right),\; {}^{t+\Delta t}\dot{\mathbf{q}} = {}^{t}\dot{\mathbf{q}} + \Delta t\left((1-\gamma)\,{}^{t}\ddot{\mathbf{q}} + \gamma\,{}^{t+\Delta t}\ddot{\mathbf{q}}\right),$$

which are used for the determination of three unknowns ${}^{t+\Delta t}\mathbf{q}$, ${}^{t+\Delta t}\dot{\mathbf{q}}$ and ${}^{t+\Delta t}\ddot{\mathbf{q}}$. The parameters β and γ determine the stability and accuracy of the algorithm and were initially proposed by Newmark as $\beta = 1/4$ and $\gamma = 1/2$ thus securing the unconditional stability of the method, which means that the solution, for any set of initial conditions, does not grow without bounds regardless of the time step. Unconditional stability itself does not secure accurate and physically sound results, however. See [6], [21], [47], [22].

With the values of β and γ mentioned above, the method is sometimes referred to as the constant-average acceleration version of the Newmark method, and is widely used for structural dynamic problems. In this case the method conserves energy.

For linear problems the mass, damping and stiffness matrices are constant and the method leads to the repeated solution of the system of linear algebraic equations at each time step giving the displacements at time $t+\Delta t$ by solving the system $\mathbf{K}^{\text{eff}}\, {}^{t+\Delta t}\mathbf{q} = \mathbf{F}^{\text{eff}}$, where so called effective quantities are

$$\mathbf{K}^{\text{eff}} = \mathbf{K} + a_0\,\mathbf{M} + a_1\,\mathbf{D},$$
$$\mathbf{F}^{\text{eff}} = {}^{t+\Delta t}\mathbf{F}^{\text{ext}} + \mathbf{M}\left(a_0\,{}^{t}\mathbf{q} + a_2\,{}^{t}\dot{\mathbf{q}} + a_3\,{}^{t}\ddot{\mathbf{q}}\right) + \mathbf{D}\left(a_1\,{}^{t}\mathbf{q} + a_4\,{}^{t}\dot{\mathbf{q}} + a_5\,{}^{t}\ddot{\mathbf{q}}\right),$$

where

$$a_0 = 1/(\beta\,\Delta t^2),\ a_1 = \gamma/(\beta\,\Delta t),\ a_2 = 1/(\beta\,\Delta t),\ a_3 = 1/(2\beta) - 1,\ a_4 = \gamma/\beta - 1,$$

$$a_5 = \frac{1}{2}\Delta t(\gamma/\beta - 2),\ a_6 = \Delta t(1-\gamma),\ a_7 = \Delta t\,\gamma.$$

The last two parameters are used for calculating of the accelerations and velocities at time $t + \Delta t$

$${}^{t+\Delta t}\ddot{\mathbf{q}} = a_0\left({}^{t+\Delta t}\mathbf{q} - {}^{t}\mathbf{q}\right) - a_2\,{}^{t}\dot{\mathbf{q}} - a_3\,{}^{t}\ddot{\mathbf{q}},\ {}^{t+\Delta t}\dot{\mathbf{q}} = {}^{t}\dot{\mathbf{q}} + a_6\,{}^{t}\ddot{\mathbf{q}} + a_7\,{}^{t+\Delta t}\ddot{\mathbf{q}}.$$

The Matlab implementation of the Newmark algorithm is in the Program 4.

Program 4

```
function [disn,veln,accn] = newmd(beta,gama,dis,vel,acc,xm,xd,xk,p,h)
% Newmark integration method for [XM]{acc} + [XD]{vel} + [XK]{dis} = {p}
%
% beta, gama ................ coefficients
% dis,vel,acc ............... displacements, velocities, accelerations
%                             at the beginning of time step
% disn,veln,accn ............ corresponding quantities at the end
%                             of time step
% xm,xd ..................... mass and damping matrices
% xk ........................ effective stiffness matrix (NOT the STIFFNESS matrix)
% p ......................... loading vector at the end of time step
% h ......................... time step
%
% constants
a1 = 1/(beta*h*h);
a2 = 1/(beta*h);
a3 = 1/(2*beta) - 1;
a4 = (1 - gama)*h;
a5 = gama*h;
a1d = gama/(beta*h);
a2d = gama/beta - 1;
a3d = 0.5*h*(gama/beta - 2);

% effective loading vector
r = p + xm*(a1*dis + a2*vel + a3*acc) + xd*(a1d*dis + a2d*vel + a3d*acc);
```

```
% solve system of equations for displacements using lu decomposition of xk
disn = xk\r;

% new velocities and accelerations
accn = a1*(disn - dis) - a2*vel - a3*acc;
veln = vel + a4*acc + a5*accn;
% end of newmd
```

□ End of Program 4.

The implementation shown in the Program 4 works well and might be useful for many medium sized tasks. Notice that a rather inefficient the equation solving – using the Matlab backslash operator – is provided at each integration step.

An efficient implementation of the Newmark methods for linear problems requires that the direct methods (e.g. Gauss elimination) are used for the solution of the system of algebraic equations. The effective stiffness matrix is positive definite, which allows to proceed without a search for the maximum pivot. Furthermore the effective stiffness matrix is constant and thus can be factorized only once, before the actual time marching, and at each step only factorization of the right hand side and backward substitution is carried out. This makes the Newmark method very efficient; the treatment of a problem with a consistent mass matrix requires even less floating point operations than that using the central difference method.

The Fortran implementation of the Newmark algorithm, taking the efficient storage considerations into account, is in the Program 5.

Program 5

```
      SUBROUTINE DNEWMD(BETA,GAMA,DIS,VEL,ACC,F,NF,CKR,CMR,CDR,NDIM,
     1                NBAND,H,EPS,IER,DISS,VELS,ACCS,P,G,G1)
C
      DOUBLE PRECISION BETA,GAMA,DIS,VEL,ACC,F,CKR,CMR,CDR,H,EPS,
     /                 DISS,VELS,ACCS,P,G,A1,A2,A3,A4,A5,AD1,AD2,AD3,G1
      DIMENSION DIS(NF),VEL(NF),ACC(NF),F(NF),CKR(NDIM,NBAND),
     1          CMR(NDIM,NBAND),DISS(NF),VELS(NF),ACCS(NF),
     2         P(NF),G(NF),G1(NF),CDR(NDIM,NBAND)
C
C     *** Newmark time integration ***
C
C     The system of ordinary differential equations
C       [M]{ACC}+[C]{VEL}+[K]{DIS}={P}
C     with symmetric banded positive definite matrices,
C     i.e. the mass, damping and the reduced effective stiffness matrices
C     are efficiently stored in rectangular arrays CMR, CDR, CKR
C     with dimensions NDIM*NBAND. Only the upper part of the matrix
C     band (including diagonal) is stored in the memory.
C     Double precision version
C     Required subroutines:
C       DMAVB       ... Matrix vector multiplication for the rectangular storage mode
C       DGRE        ... Solution of linear algebraic equations for the rectangular storage mode
C
C     Parameters
C     BETA,GAMA......... Newmark parameters
C     DIS(NF)  ......... On input ... displacements at time T
C                        On output .. displacements at time T+H
```

```
C     VEL(NF)  .......... On input ... velocities at time T
C                         On output .. velocities at time T+H
C     ACC(NF)  .......... On input ... accelerations at time T
C                         on output .. accelerations at time T+H
C     F(NF)    .......... Loading at time (T+H)
C     NF       .......... Number of unknowns
C     CKR(NDIM,NBAND) ....Upper part of the band of the reduced effective
C                         stiffness matrix, i.e. the matrix
C                         [K]+A1*[M]+AD1*[C] after being processed by
C                         the DGRE subroutine with parameter KEY = 1)
C     CMR(NDIM,NBAND) ....Upper part of the band of the mass matrix
C     CDR(NDIM,NBAND) ....Upper part of the band of the damping matrix
C     NDIM     ...........Row dimension of CKR, CMR, CDR matrices
C                         in the main program
C     NBAND    .......... Half band size (including diagonal)
C     H        .......... Integration step
C     EPS      .......... Pivot tolerance for DGRE subroutine
C     IER      .......... Error parameter (See DGRE)
C     DISS(NF),VELS(NF),ACCS(NF) ..... Displacements, velocities and
C                         accelerations from the previous step
C     P(NF),G(NF),G1(NF). auxiliary arrays
C
C     *********************************************************
C
C     Constants
C
      A1 = 1.D0/(BETA*H*H)
      A2 = 1.D0/(BETA*H)
      A3 = 1.D0/(BETA*2.D0)-1.D0
      A4 = (1.D0-GAMA)*H
      A5 = GAMA*H
      AD1 = GAMA/(BETA*H)
      AD2 = (GAMA/BETA)-1.D0
      AD3 = (H/2.D0)*(GAMA/BETA-2.D0)
C
      DO 10 I=1,NF
      DISS(I) = DIS(I)
      VELS(I) = VEL(I)
10    ACCS(I) = ACC(I)
C
C     Vector of effective loading forces at time T + H
C
      DO 40 I = 1,NF
      G(I) = A1*DISS(I) + A2*VELS(I) + A3*ACCS(I)
40    G1(I) = AD1*DISS(I) + AD2*VELS(I) + AD3*ACCS(I)
      CALL DMAVB(CMR,G,P,NF,NDIM,NBAND)
      DO 50 I = 1,NF
50    G(I) = F(I) + P(I)
      CALL DMAVB(CDR,G1,P,NF,NDIM,NBAND)
      DO 57 I = 1,NF
        G(I) = G(I) + P(I)
57    CONTINUE
C
C     Displacements at time T + H
C
      IER = 0
      DET = 0.D0
      CALL DGRE(CKR,G,DIS,NF,NDIM,NBAND,DET,EPS,IER,2,kerpiv)
C
C     Accelerations at time T + H
C
     DO 60 I = 1,NF
60    ACC(I) = A1*(DIS(I) - DISS(I)) - A2*VELS(I) - A3*ACCS(I)
C
C     Velocities at time T+H
C
     DO 70 I = 1,NF
70   VEL(I) = VELS(I) + A4*ACCS(I) + A5*ACC(I)
C
      RETURN
      END
%
```

□ End of Program 5.

The DGRE procedure needed for the DNEWMD subroutine is in the Program 6. The subroutine DMAVBA is in the Program 7.

Program 6

```
      SUBROUTINE DGRE(A,B,Q,N,NDIM,NBAND,DET,EPS,IER,KEY,KERPIV)
      DIMENSION A(NDIM,NBAND),B(N),Q(N)
      DOUBLE PRECISION SUM,A,B,Q,AKK,AKI,AIJ,AKJ,ANN,T,AL1,AL2,AII
     1                ,DET,EPS
C
C     ***   Solution of R*Q=B                ***
C     ***   by Gauss elimination for banded, ***
C     ***   symmetric, positive definite     ***
C     ***   matrix                           ***
C     ***   DOUBLE PRECISION version         ***
C     It is assumed that the upper part
C     of the band of R matrix is stored in
C     a rectangular array A
C
C
C     Parameters
C     A      On input - rectangular array containing
C                       the upper band of R matrix
C            On output  triangularized matrix
C     B      RHS vector
C     Q      result
C     N      Number of unknowns
C     NDIM   Row dimension of A array declared in main
C     NBAND  half-band size (including diagonal)
C     DET    Matrix determinant
C     EPS    Minimum acceptable pivot value
C     KERPIV pointer to the pivot, where the error occurred
C     IER    error parameter
C            = 0 ... O.K.
C            =-1 ... absolute value of pivot
C                    smaller than EPS
C                    The matrix not positive definite
C                    The computation is stopped
C     KEY    key
C            = 1 ... reduction (triangularization) of R
```

```
C             = 2 ... reduction of the RHS and
C                     the back substitution
      NM1 = N-1
      IER = 0
      II = KEY
      GO TO (1000,2000), II
C
C     reduction (triangularization) part
C
1000  DET = 1.
      IRED = 0
      DO 9 K = 1,NM1
      AKK = A(K,1)
      KERPIV = K
      IMAX = K + NBAND - 1
      IF(IMAX .GT. N) IMAX=N
      JMAX = IMAX
      IF(AKK .GT. EPS) GO TO 5
      IER = -1
      RETURN
C     DET = DET*AKK
5     CONTINUE
      KP1 = K+1
      DO 9 I = KP1,IMAX
      AKI=A(K,I-K+1)
      IF(ABS(AKI) .LT. EPS) GO TO 9
      T=AKI/AKK
      DO 8 J = I,JMAX
      AIJ=A(I,J-I+1)
      AKJ=A(K,J-K+1)
8     A(I,J-I+1) = AIJ - AKJ*T
9     CONTINUE
      ANN = A(N,1)
      DET = DET*ANN
C
C     reduction is successfully finished
C
      IRED = 1
      RETURN
C
C     RHS reduction
C
2000  DO 90 K = 1,NM1
      KP1  =K + 1
      AKK = A(K,1)
      IMAX = K+NBAND-1
      IF(IMAX .GT. N) IMAX=N
      DO 90 I = KP1,IMAX
      AKI=A(K,I-K+1)
      T = AKI/AKK
90    B(I) = B(I) - T*B(K)
C
C     Back substitution
C
      Q(N) = B(N)/A(N,1)
      AL1 = A(N-1,2)
      AL2 = A(N-1,1)
      Q(N-1) = (B(N-1) - AL1*Q(N))/AL2
      DO 10 IL = 3,N
      I = N - IL + 1
      AII = A(I,1)
      SUM = 0.D0
      J1 = I + 1
      JMAX = MIN0(I+NBAND-1,N)
      DO 20 J = J1,JMAX
      AIJ = A(I,J-I+1)
20    SUM = SUM+AIJ*Q(J)
10    Q(I) = (B(I) - SUM)/AII
      RETURN
      END
```

□ End of Program 6.

If $\gamma \geq \frac{1}{2}$ and $\beta = \frac{1}{4}(\frac{1}{2}+\gamma)^2$ the method is still unconditionally stable but a positive algorithmic damping is introduced into the process. With $\gamma < \frac{1}{2}$ a negative damping is introduced, which eventually leads to an unbounded response. With different values of parameters β, γ, the Newmark scheme describes a whole series of time integration methods, which are sometimes called the Newmark family.

For example if $\beta = 1/12$ and $\gamma = \frac{1}{2}$, it is a well known Fox-Goodwin formula, which is implicit and conditionally stable, else if $\gamma = \frac{1}{2}$ and $\beta = 0$, then the Newmark's method becomes a central difference method, which is conditionally stable and explicit (if mass and damping matrices are diagonal).

The algorithmic damping introduced into the Newmark method, by setting the parameter $\gamma > \frac{1}{2}$ and calculating the other parameter as $\beta = \frac{1}{4}(\frac{1}{2}+\gamma)^2$, is frequently used in practical computations, since it filters out the high-frequency components of the mechanical system's response. This damping is generally viewed as desirable, since the high-frequency components are very often mere artifacts of finite element modelling, they have no physical meaning and are consequences of the discrete nature of the finite element model and its dispersive properties. See [22].

It is known that algorithmic damping adversely influences the lower modes in the solution. To compensate for the negative influence of algorithmic damping on the lower modes behaviour Hilber [18], [19] modified Newmark method with the intention of ensuring adequate dissipation in the higher modes and at the same time guaranteeing that the lower modes are not affected too strongly.

In a nonlinear case the displacements are not small, the strains are finite, constitutive equations are nonlinear and the boundary conditions can change with time. This means that the equation of motion written for an undamped structure at time $t + \Delta t$ is

$$\mathbf{M}\,{}^{t+\Delta t}\ddot{\mathbf{q}} + {}^{t+\Delta t}\mathbf{F}^{\mathrm{int}} = {}^{t+\Delta t}\mathbf{F}^{\mathrm{ext}},$$

where

$${}^{t+\Delta t}\mathbf{F}^{\mathrm{int}} = \int_{{}^{t+\Delta t}V} {}^{t+\Delta t}\mathbf{B}^{\mathrm{T}}\,{}^{t+\Delta t}\boldsymbol{\sigma}\,\mathrm{d}^{t+\Delta t}V$$

.

The strain-displacement matrix $\mathbf{B}$ and stresses $\boldsymbol{\sigma}$ depend on a current configuration which is generally unknown.

The equations of motion should be satisfied at any moment. The so called vector of residual forces, which can be expressed as a function of current displacements, is

$\mathbf{R}({}^{t+\Delta t}\mathbf{q}) = \mathbf{M}\,{}^{t+\Delta t}\ddot{\mathbf{q}} + {}^{t+\Delta t}\mathbf{F}^{\mathrm{int}} - {}^{t+\Delta t}\mathbf{F}^{\mathrm{ext}}$ and should be identically equal to zero if dynamic equilibrium is achieved.

If we employ Newmark relations for ${}^{t+\Delta t}\mathbf{q}$ and ${}^{t+\Delta t}\dot{\mathbf{q}}$, then ${}^{t+\Delta t}\ddot{\mathbf{q}}$ can be expressed and substituted into the previous equation. Using Taylor series and neglecting the nonlinear terms we have $\mathbf{R}({}^{t+\Delta t}\mathbf{q}) = \mathbf{R}({}^{t}\mathbf{q}) + \mathbf{J}({}^{t}\mathbf{q})\Delta\mathbf{q} + \ldots$ where the increment of displacements is $\Delta\mathbf{q} = {}^{\mathrm{t}+\Delta\mathrm{t}}\mathbf{q} - {}^{t}\mathbf{q}$ and $\mathbf{J}$ is the so called Jacobi matrix, which, in the case of Newmark's approach for kinematic quantities, can be expressed as $\mathrm{J}_{ij} = \partial \mathrm{R}_i/\partial \mathrm{q}_j$ or $\mathbf{J} = a_0\,\mathbf{M} + a_1\,{}^{t}\mathbf{D} + {}^{t}\mathbf{K}$.

Setting the right-hand side of Taylor expansion to the expected zero value, we get the first estimate of the increment of displacements $\Delta\mathbf{q}$ which can be obtained by solving the system of equations $\mathbf{J}({}^{t}\mathbf{q})\,\Delta\mathbf{q} = \mathbf{R}({}^{t}\mathbf{q})$.

The increments must be sequentially refined in an iterative manner, until the convergence criteria are satisfied, i.e. the norms of the residual forces and those of the increments must be substantially less than the norms of the external forces and of current displacements respectively. Tolerance parameters are typically 10^{-2} to 10^{-3} or smaller. See [3].

The suggested approach uses actually the Newton's method for iteration within each time step and is thus very expensive, since it requires the recalculation of the Jacobi matrix in each iteration, which in turn requires reassembly of new ${}^{t}\mathbf{K}$ and ${}^{t}\mathbf{D}$ for the current iteration. That's why explicit methods are almost exclusively used here.

Today many implicit finite element codes use a combination of line search and quasi-Newton methods. See [10], [14], [28].

Some recent developments in the field of time integration procedures, more efficient elements, adaptive meshes and the exploitation of parallel computers appear in [5].

References on other algorithms used in structural analysis (Houbolt, collocation methods, operator splitting methods and others) can be found in [22].

The stress wave problem methodologies, briefly described here, would be unthinkable of a swift progress which is constantly being reported in so called finite element technology area. A bibliography of books and monographs on finite element technology was recently presented by Noor in [31].

3.5 Solution of nonlinear tasks

Virtually everything in the nature is nonlinear. There are, however, no universally available hints and approaches to the solution of nonlinear problems of continuum mechanics regardless of sources of nonlinearity. There are no direct solvers which would lead to the sought-after result by a finite number of steps, all nonlinear solvers are iterative and lead to the solution by linearization. On the other hand significant advances in the development and implementation of nonlinear procedures have been reported in the last decade. See [35].

In mechanics of solids the sources of nonlinearity stem from nonlinear constitutive equations, from considering large displacements and large strains, from change of boundary conditions during the solution as in contact impact problems, etc. [3], [53].

Generally, the task to be solved is to find an equilibrium state at the current configuration ${}^{t}\mathrm{C}$ due to a prescribed loading ${}^{t}\mathbf{Q}$. We assume that the "state" quantities at the reference configuration ${}^{0}\mathrm{C}$, i.e. generalized displacements ${}^{0}\mathbf{u}$, generalized displacements an nodes ${}^0\mathbf{q}$, strains ${}^0\boldsymbol{\varepsilon}$, stresses ${}^0\boldsymbol{\sigma}$ and the initial loads ${}^0\mathbf{Q}$ are known. We are looking for corresponding quantities at configuration ${}^t\mathrm{C}$, i.e. ${}^t\mathbf{u}$, ${}^t\mathbf{q}$, ${}^t\boldsymbol{\varepsilon}$, ${}^t\boldsymbol{\sigma}$.

The external forces $\mathbf{Q}$ consisting of body, surface and applied forces at nodes and the internal forces $\mathbf{F} = \int \mathbf{B}^{\mathrm{T}}\boldsymbol{\sigma}\,\mathrm{d}V$, where $\mathbf{B}$ is an operator relating the strains to generalized displacements at nodes, have to be in equilibrium at any moment, i.e. ${}^0\mathbf{Q} = {}^0\mathbf{F}$ and ${}^t\mathbf{Q} = {}^t\mathbf{F}$. The internal forces at ${}^t\mathrm{C}$ could be approximated by ${}^t\mathbf{F} = {}^0\mathbf{F} + \Delta\mathbf{F}$ while increments of internal forces by $\Delta\mathbf{F} = {}^0\mathbf{K}\,\Delta\mathbf{q}$, with tangent stiffness matrix is ${}^0\mathbf{K} = \mathbf{K}({}^0\mathbf{q})$ and the increments of generalized displacements at nodes are $\Delta\mathbf{q} = {}^t\mathbf{q} - {}^0\mathbf{q}$.

Putting it all together we could calculate the increments of displacements solving the linear system of equations ${}^0\mathbf{K}\,\Delta\mathbf{q} = \mathbf{Z}$, where $\mathbf{Z} = {}^t\mathbf{Q} - {}^0\mathbf{F}$ is a so called residual (sometimes out-of-balance) force. This gives the first estimation of 'new' generalized displacements at nodes ${}^t\mathbf{q} = {}^0\mathbf{q} + \Delta\mathbf{q}$ which, however, should be refined in a suitable iteration process.

The iteration process should be stopped when the increments of displacements (in a suitable norm) are small with respect to a norm of current displacements and, at the same time, when the out-of-balance force is small with respect to the applied load. With these two conditions satisfied we should not be far from equilibrium conditions.

Generally, the system of n nonlinear equations with n unknowns $f_i(x_j) = 0$ or $\mathbf{f} = \mathbf{f}(\mathrm{x}) = 0$ could be treated by many methods, the most known of them being so-called Newton-like and quasi-Newton methods.

3.5.1 Newton-like methods

The classical representative of the former group is the Newton-Raphson (sometimes called Newton only) method. It is based on the Taylor theorem applied to $\mathbf{f}(\mathrm{x})$ at $\mathbf{x}+\Delta\mathbf{x}$, which yields $\mathbf{f}(\mathrm{x}+\Delta\mathbf{x}) = \mathbf{f}(\mathrm{x}) + \mathbf{J}\,\Delta\mathbf{x}$ + higher other terms, where a generic term of the Jacobi matrix $\mathbf{J}$ has the form $J_{ij} = \partial f_i(x_k)/\partial x_j$. Neglecting the higher order terms we could find such a $\Delta\mathbf{x}$, which satisfies the condition $0 = \mathbf{f}(\mathrm{x}) + \mathbf{J}\,\Delta\mathbf{x}$. This leads to a system of algebraic equations in the form $\mathbf{J}\,\Delta\mathbf{x} = -\mathbf{f}(\mathrm{x})$. Solving it for $\Delta\mathbf{x}$ gives the first estimation of the solution which, however, must be refined in an iterative process.

Notes to the Newton-Raphson method

- We are satisfied with the test of convergence if the system of equations is satisfied, i.e. $\|\mathbf{f}\|$ is "small" and if the last norm of increment $\|\Delta\mathbf{x}\|$ is small with respect $\|\mathbf{x}\|$. At the same time the number of iterations should not be excessive.
- Tolerances should not be to severe. Their reasonable setting could be tricky.
- The norms being used in calculations should be cheap.
- If the Jacobi matrix is positive definite then a quadratic convergence is guaranteed.
- The method is expensive and requires the Jacobi matrix calculation at each iteration.
- In finite element calculations the Jacobi matrix is usually taken as a tangent stiffness matrix evaluated at $\mathbf{x}$.

- The modified Newton-Raphson method is a variant of Newton Raphson method in which the initial Jacobi matrix is used throughout the entire iteration process. For the simplification of the computational process we pay by slower convergence resulting in the higher number of iteration steps needed for convergence.
- The modified Newton-Raphson method exists in many variants. Sometimes the Jacobi matrix is recalculated after several iteration steps, sometimes it is combined with the division of the total loading into several steps which are applied consecutively. See Template 12.

For proofs and details see [46], [3], [2], [35].

Template 12, Newton-Raphson process with incremental loading

Assume that from `t = 0` to `t = tmax` there are `kmax` same loading steps. The maximum loading force corresponding to to final configuration at time `t` is ${}^{tmax}\mathbf{R} = {}^{kmax}\mathbf{R}$. Linear increase of the loading between consecutive steps is assumed.The force corresponding to the `k`-th loading step is ${}^{k}\mathbf{R} = {}^{kmax}\mathbf{R} * k/kmax$.

Notation

${}^0\mathbf{K}$	tangent stiffness matrix at ^{0}C
${}^0\mathbf{F}$	internal forces at ^{0}C
${}^0\mathbf{q}$	generalized displacements in nodes at ^{0}C
k	as an upper left-hand index is the loading pointer
(i)	as an upper right-hand index is the iteration counter
${}^k\mathbf{K}^{(i)}$	the i-th iteration of $\mathbf{K}$ at the k-th loading step

```
for k = 1 to kmax                    incremental-loading loop
  i = 0;                             iteration counter
  if k = 1 then
      ${}^k\mathbf{K}^{(0)} = {}^0\mathbf{K}; {}^k\mathbf{q}^0 = \mathbf{q}^{(0)}; {}^k\mathbf{F}^{(0)} = {}^0\mathbf{F} = \mathbf{0};$
    else
      ${}^k\mathbf{K}^{(0)} = {}^{k-1}\mathbf{K}^{(ilast)}; {}^k\mathbf{q}^{(0)} = {}^{k-1}\mathbf{q}^{(ilast)};$
  end                                of if branch
  ${}^k\mathbf{R} = {}^{kmax}\mathbf{R} * k/kmax;$
  satisfied = false;
  while not satisfied do
    i = 1 + 1;
    solve ${}^k\mathbf{K}^{(i+1)}\,\Delta\mathbf{q}^{(i)} = {}^k\mathbf{R} - {}^k\mathbf{F}^{(i-1)}$ for $\Delta\mathbf{q}^{(i)}$
    ${}^k\mathbf{q}^{(i)} = {}^k\mathbf{q}^{(i-1)} + \Delta\mathbf{q}^{(i)};$        update displacements
    assemble ${}^k\mathbf{K}^{(i)}$ for new ${}^k\mathbf{q}^{(i)}$
    calculate new internal forces ${}^k\mathbf{F}^{(i)}$
    ilast = i
    satisfied =  $(\|\Delta\mathbf{q}^{(i)}\|/\|{}^k\mathbf{q}^{(ilast)}\| < \varepsilon_1$ and $\|{}^k\mathbf{R} - {}^k\mathbf{F}^{(ilast)}\|/\|{}^k\mathbf{R}\| < \varepsilon_2)$
  end                                of while loop
end                                  of for loop
```

3.5.2 Quasi-Newton methods

The idea behind these methods can be outlined as follows. The Taylor theorem applied to equilibrium conditions at the k-th iteration gives ${}^{(k+1)}\mathbf{f} = {}^{(k)}\mathbf{f} + {}^{(k)}\mathbf{J}\,({}^{(k+1)}\mathbf{x} - {}^{(k)}\mathbf{x})$. Denoting increments $\Delta\mathbf{f} = {}^{(k+1)}\mathbf{f} - {}^{(k)}\mathbf{f}$ and $\Delta\mathbf{x} = {}^{(k+1)}\mathbf{x} - {}^{(k)}\mathbf{x}$ we have $\Delta\mathbf{f} = {}^{(k)}\mathbf{J}\,\Delta\mathbf{x}$. The following approach, allowing to calculate ${}^{(k)}\mathbf{J}$ out of the previous iteration, and not by recomputing all derivatives as before, is due to Broyden [10]. Actually, the matrix $\mathbf{J}$ is not the Jacobi matrix but a matrix which is replacing it successfully. One would not surely comply against the identity ${}^{(k-1)}\mathbf{J}\,\Delta\mathbf{x} = {}^{(k-1)}\mathbf{J}\,\Delta\mathbf{x}$. Combining the last two equations and multiplying the result by $\Delta\mathbf{x}^{\mathrm{T}}$ from the right we get $(\Delta\mathbf{f} - {}^{(k-1)}\mathbf{J}\,\Delta\mathbf{x})\,\Delta\mathbf{x}^{\mathrm{T}} = ({}^{(k)}\mathbf{J} - {}^{(k-1)}\mathbf{J})\,\Delta\mathbf{x}\,\Delta\mathbf{x}^{\mathrm{T}}$. Denoting by $\mathbf{X}$ the dyadic product $\Delta\mathbf{x}\,\Delta\mathbf{x}^{\mathrm{T}}$ we get the relation for the k-th iteration of ${}^{(k)}\mathbf{J}$ in the form ${}^{(k)}\mathbf{J} = {}^{(k-1)}\mathbf{J} + ((\Delta\mathbf{f} - {}^{(k-1)}\mathbf{J}\,\Delta\mathbf{x})\,\Delta\mathbf{x}^{\mathrm{T}})\mathbf{X}^{-1}$ containing, however, the inversion of $\mathbf{X}$ matrix, which spoils the whole process a little bit. At this point Broyden suggested a brilliant trick. He has shown that one can always find another vector, say $\mathbf{z}$, satisfying both orthogonality conditions $\Delta\mathbf{x}^{\mathrm{T}}\,\mathbf{z} = \mathbf{z}^{\mathrm{T}}\,\Delta\mathbf{x}$ and the equation ${}^{(k-1)}\mathbf{J}\,\Delta\mathbf{z} = {}^{(k)}\mathbf{J}\,\Delta\mathbf{z}$. It can be easily proved that under these conditions the dyadic product $\Delta\mathbf{x}\,\Delta\mathbf{x}^{\mathrm{T}}$ can be 'replaced' by the scalar one, i.e. by $\Delta\mathbf{x}^{\mathrm{T}}\,\Delta\mathbf{x}$ which miraculously simplifies the formula for the calculation of ${}^{(k)}\mathbf{J}$ out of ${}^{(k-1)}\mathbf{J}$. The **Broyden formula** then reads

$$
{}^{(k)}\mathbf{J} = ({}^{(k-1)}\mathbf{J} + ((\Delta\mathbf{f} - {}^{(k-1)}\mathbf{J}\,\Delta\mathbf{x})\,\Delta\mathbf{x}^{\mathrm{T}}))\,/\,\Delta\mathbf{x}^{\mathrm{T}}\,\Delta\mathbf{x}
$$

.

It is proved [46] that the convergence is faster than linear. As a starting guess, we usually take ${}^{(0)}\mathbf{J} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. At each iteration the system of algebraic equations is solved as in the case of Newton-Raphson method. This computationally expensive step could be circumvented if the iteration process would yield ${}^{(k)}\mathbf{J}^{-1}$ instead of ${}^{(k)}\mathbf{J}$.

Denoting ${}^{(k)}\mathbf{H} = {}^{(k)}\mathbf{A}^{-1}$ at the k-th iteration the **improved Broyden formula** reads

$$
{}^{(k+1)}\mathbf{H} = {}^{(k)}\mathbf{H} - \frac{({}^{(k)}\mathbf{H}\,\Delta\mathbf{f} - \Delta\mathbf{x})\,\Delta\mathbf{x}^{\mathrm{T}}}{\Delta\mathbf{x}^{\mathrm{T}}\,{}^{(k)}\mathbf{H}\,\Delta\mathbf{f}}\,{}^{(k)}\mathbf{H}
$$

.

The process, explaining this, is derived in [50].

Today, variants of these approaches are standardly implemented in finite element codes under a cryptic acronym BFGS which originated from initial letters of names of respective gentlemen, i.e. Broyden, Fletcher, Goldfarb and Shanno. See [14], [35].

The methods mentioned so far are sometimes classified as implicit since they involve the computation of system of algebraic equations with stiffness matrices.

There are other approaches to the solution of nonlinear problems, called explicit, which generally do not require assembling of stiffness matrices and their handling. The representatives of these methods are conjugate gradient method, the method of the steepest descent, the dynamic overrelaxation methods and others.

3.5.3 The method of the steepest descent

This is another method which could be used if the derivatives appearing in the Jacobi matrix are hard to be found. We could take $\mathbf{J}$ as an identity matrix or its multiple $\mathbf{I}/\alpha$.

In this case the iteration we get $0 = {}^{(k)}\mathbf{f} + (\mathbf{I}/\alpha)\left({}^{(k+1)}\mathbf{x} - {}^{(k)}\mathbf{x}\right)$ which makes the iteration steps very simple, i.e. ${}^{(k+1)}\mathbf{x} = {}^{(k)}\mathbf{x} - \alpha\,{}^{(k)}\mathbf{f}$. The name of the method stems from its geometrical interpretation. The function $\mathbf{f} = \mathbf{f}(\mathbf{x})$ can be viewed as a derivative of a scalar function $U = U(\mathbf{x}) = U(x_j)$, i.e. $f_i = \partial U/\partial x_j$ and also as a gradient of this function. The gradient of a scalar function is a vector pointing in the direction of the steepest slope. Moving on the 'surface' U in the direction, where the surface is the steepest, i.e. in the direction of a negative gradient of U, we aim at the minimum of U, for which $f_i = 0$ holds. It should be reminded that a new direction ${}^{(k)}\mathbf{f}$ have to be calculated at each iteration. There is a problem with the determination of parameter α which tells us how far we should go in the direction of gradient at each iteration. Its exact determination could be found from $\partial U/\partial\alpha = 0$, but in practice it is being determined experimentally. See [46], [35].

In solid mechanics the scalar function U represents the potential energy of the system, i.e. $\frac{1}{2}\mathbf{x}^{\mathrm{T}}\,\mathbf{A}\,\mathbf{x}$, the function $\mathbf{f} = \mathbf{f}(\mathbf{x})$ corresponds to components of $\mathbf{A}\mathbf{x} - \mathbf{b}$ and the elements of Jacobi matrix $J_{ij} = \partial f_i/\partial x_j = \partial^2 U/\partial x_i\,\partial x_j$ are entries of $\mathbf{A}$.

3.5.4 The conjugate gradient method

could be formulated similarly. The iteration process goes by the formula ${}^{(k+1)}\mathbf{x} = {}^{(k)}\mathbf{x} + \alpha\,{}^{(k)}\mathbf{f}$ with direction ${}^{(k)}\mathbf{d} = -{}^{(k)}\mathbf{f} + \beta_k\,{}^{(k-1)}\mathbf{d}$. The direction is chosen in such a way that it is orthogonal to all previous directions from the condition $\beta_k = {}^{(k)}\mathbf{f}^{\mathrm{T}}\left({}^{(k)}\mathbf{f} - {}^{(k-1)}\mathbf{f}\right)/\left({}^{(k-1)}\mathbf{f}^{\mathrm{T}}\,{}^{(k-1)}\mathbf{f}\right)$. The implementation details are in Chapter 6. A nice introductory titled text *An introduction to the conjugate gradient method without agonizing pain* is in

`http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.`

3.5.5 Tracing equilibrium paths methods

There are cases in structural mechanics where load-displacement relations exhibit unstable branches as in snap-through and snap-back problems. The classical representative of these method is a so-called arc-length method. See [35].

The choice a proper approach to the nonlinear solution of a particular task is not easy, it is still a matter of judicious engineering judgment foresight and experience.

3.6 Conclusions

Computational methods in mechanics of solids are subjected to intensive research. The methods based on classical matrix algebra are being reconsidered from the point of view of new parallel hardware available. Scientists become programmers with the intention to accelerate commercial codes to be able to solve the large-scale problems of fundamental research and technical practice. New methods are quickly becoming engineering standards even if they did not yet succeed to find their way into today's engineering textbooks. The authors believe that the fascinating subject computational mechanics is to be studied intensively in order to be able to utilize the hardware tools that are available now and to tackle the tasks that, until recently, seemed unsolvable.

Bibliography

[1] L. Adams and H. Jordan. Is sor color-blind? *SIAM, Journal of Sci. Stat. Comp.*, (7):490–506, 1986.

[2] R. Barrett et al. *Templates for the Solution of Linear Systems.* SIAM, Philadelphia, 1994.

[3] K.J. Bathe. *Finite element procedures in engineering analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

[4] K.J. Bathe and E. L. Wilson. *Numerical Methods in Finite Element Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[5] T. Belytschko. On computational methods for crashworthiness. *Computers and Structures*, (2):271–279, 1992.

[6] T. Belytschko and T.R.J. Hughes. *Computational methods for transient analysis.* North Holland, Amsterdam, 1986.

[7] T. Belytschko and J. S-J. Ong. Hourglass control in linear and nonlinear problems. *Computer Methods in Applied Mechanics and Engineering*, (43):251–276, 1984.

[8] C.B. Boyer. *A history of Mathematics.* John Wiley and Sons, New York,, 1969.

[9] A.L. Briggs. *A Multigrid Tutorial.* SIAM, Philadelphia, 1987.

[10] C.G. Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematical Computations*, (19):577–593, 1965.

[11] T.S. Chow and J.S. Kowalik. Computing with sparse matrices. *International Journal for Numerical Methods in Engineering*, 7:211–223, 1973.

[12] R.D. Cook. *Concepts and applications of finite element method.* John Wiley and Sons, New York, 1991.

[13] Cray Research, Inc., Mendota Heights, MN, U.S.A., 2360 Pilot Knot Road. *CF77, Compiling System*, 1991.

[14] J.E. Dennis and Moré J. Quasi-newton methods, motivation and theory. *SIAM Review*, (19):46–89, 1977.

[15] M.A. Dokainish and K. Subbaraj. A survey of direct time integration methods in computational structural dynamics - i. explicit methods. *Computers and Structures*, (6):1371–1386, 1989.

[16] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

[17] J. Herzbergen. Iteractionverfahren hoeheren ordnung zur einschliessung des inversen einer matrix. *ZAMM*, 69:115–120, 1989.

[18] H. M. Hilber and et al. Improved numerical dissipation for time integration algorithms in structural dynamics. *Earthquake Engineering and Structural Dynamics*, (5):283–292, 1977.

[19] H. M. Hilber and T.R.J. Hughes. Collocation, dissipation and 'overshoot' for time integration schemes in structural dynamics. *Earthquake Engineering and Structural Dynamics*, (5):99–117, 1977.

[20] Y.P. Huang et al. Vectorization using a personal computer. *Developments in Computational Techniques for Structural Techniques Engineering, ed.: Topping, B.H.V., Edinburgh UK, CIVIL-COMP PRESS*, pages 427–436, 1995.

[21] T.J.R. Hughes. *The finite element method.* Prentice-Hall, Englewood Cliffs, N.J., 1987.

[22] T.J.R. Hughes and T. Belytschko. A precis development in computational methods for transient analysis. *Journal of Applied Mechanics*, (50):1033–1041, 1983.

[23] IMLS, Inc.,, GNB Building, 7500 Bellair BVLD, Housto TX 77036. *EISPACK*, 1989.

[24] B. M. Irons. A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, 2:5–32, 1970.

[25] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele, Jr., and M.E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, Mass., 1994.

[26] E.E. Kramer. *The Nature and growth of Modern Mathematics.* Hawthorn Books, Inc., New York, 1970.

[27] The MacNeal Swendler Corporation, 815 Colorado Blvd, Los Angeles, CA 90041-1777, U.S.A. *MSC/Nastran, Numerical Methods*, 1993.

[28] H. Matthies and G. Strang. The solution of nonlinear finite element equations. *International Journal for Numerical Methods in Engineering*, (14):1613–1626, 1979.

[29] S.F. McCormick. *Multigrids methods.* SIAM, Philadelphia, 1987.

[30] N.M. Newmark. A method for computation of structural dynamics. *Journal of the Engineering Mechanics Division, ASCE*, pages 67–94, 1959.

[31] A.K. Noor. Bibliography of books and monographs on finite element technology. *Applied Mechanics Review*, (6):307–317, 1991.

[32] The Numerical Algoritms Group, Ltd., Wilkinson House, Jordan Hill Road, OXFORD, U.K., OX2 8DR. *NAG Library Manual*, 1992.

[33] M. Okrouhlík, I. Huněk, and Loucký K. *Personal Computers in Technical Practice.* Institute of Thermomechanics, 1990.

[34] Y. C. Pao. Algorithm for direct access gaussian solution of structural stiffness natrix equation. *International Journal for Numerical Methods in Engineering*, (12):751–764, 1978.

[35] M. Papandrakakis. *Solving Large-Scale Problems in Mechanics.* John Wiley and Sons Ltd, Baffins Lane, Chichester, 1993.

[36] K.C. Park. Practical aspects of numerical time integration. *Computers and Structures*, (7):343–353, 1977.

[37] B. N. Parlett. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[38] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes. The Art of Scientific Computing.* Cambridge University Press, New York, 1986.

[39] A. Ralston. *A First Course in Numerical Analysis.* McGraw-Hill, Inc., New York, 1965.

[40] J.K. Reid. Algebraic aspects of finite-element solutions. *Computer Physiscs Reports*, (6):385–413, 1987.

[41] J. R. Rice. *Matrix Computations and Mathematical Software.* McGraw-Hill, Inc., Auckland,, 1983.

[42] M. Rozložník. *Numerical Stability of the GMRES Method.* PhD thesis, Institute of Computer Science, Academy of Sciences of the Czech Republic, 1996.

[43] H.R. Schwarz, H. Rutishauser, and B. Stiefel. *Numerical Analysis of Symmetrical Matrices.* Pretice-Hall, Englewood Cliffs, N.J., 1973.

[44] Society for Industrial and Applied Mathematics, Philadelphia. *LINPACK User's Guide*, 1990.

[45] G. Strang. *Linear Algebra and its Applications.* Academic Press, New York, 1976.

[46] G. Strang. *Introduction to Applied Mathematics.* Cambridge Press, Welleseley, MA, U.S.A., 1986.

[47] K. Subbaraj and M.A. Dokainish. A survey of direct time integration methods in computational structural dynamics - ii. implicit methods. *Computers and Structures*, (6):1387–1401, 1989.

[48] R.P. Tewarson. *Sparse matrices.* Academic Press, New York, 1973.

[49] B.H.V. Topping and A.I. Khan. *Parallel Finite Element Computations.* Saxe Coburg Publications, Edinburgh, U.K.,, 1995.

[50] R.E. White. *An introduction to the finite element method with applications to nonlinear problems.* John Wiley and Sons, New York, 1987.

[51] Z.H. Zhong and J. Mackerle. Contact-impact problems: A review with bibliography. *Applied Mechanics Review*, 47(2), 1994.

[52] O.C. Zienkiewicz. *The finite element method in engineering science.* McGraw-Hill, London, 1971.

[53] J. A. Zukas. *High Velocity Impact Dynamics.* John Wiley and Sons, Inc., New York, 1990.

Chapter 4

Implementation remarks to equation solvers

This part was written and is maintained by M. Okrouhlík. More details about the author can be found in the Chapter 16.

4.1 Storage modes

The *symmetry* of $\mathbf{A}$ matrix in FE analysis can be deduced from the fact that elementary matrices - out of which the global stiffness matrix is assembled - are also symmetric. They are defined by an integral over the element volume $\mathbf{k} = \int_V \mathbf{B}^{\mathrm{T}}\mathbf{E}\mathbf{B}\,\mathrm{d}V$. It is obvious that such a matrix is symmetric for any $\mathbf{B}$ on condition that the $\mathbf{E}$, i.e. that matrix of elastic moduli, is symmetric. For discussions concerning the symmetry of stress tensors in statics and dynamics see [20].

The *positive definiteness* of the matrix stems from energy minimizing principles underlying the FE model. For any permissible generalized displacement $\mathbf{x}$, the stiffness matrix $\mathbf{A}$ constitutes a quadratic form $Q = \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} > 0$, unless $\mathbf{x} = 0$. It is easy to show that Q is proportional to the deformation energy stored in the considered structure. If the structure is not properly constrained then the $\mathbf{A}$ matrix is positive semidefinite.

All the matrix properties mentioned above are very important for the solution of large-scale problems and should be properly exploited in algorithmization, programming and implementation of procedures providing matrix operations and handling.

The sparseness feature is very important since it allows to deal with matrices that otherwise could not be handled and processed due to computer memory limitations. In mechanical and civil engineering applications, the most common form of sparseness is bandedness, i.e. $a_{ij} = 0$ if $|i - j| >$ `nband`, where the identifier `nband` was introduced as a measure of the halfband width (including diagonal). How both symmetry and bandedness could be systematically employed in the algorithmization is shown in the Template 13.

Template 13, Gauss elimination – influence of symmetry and bandedness

```
three types of matrix storage mode
a ...  general matrix, b ...  symmetric, c ...  symmetric, banded
nband ...  halfband width (including diagonal)
for k = 1 to n-1
  case (a,b,c)
    a), b)   imax = n
    c)   imax = min(k+nband-1, n)
  end
  for i = k+1 to imax
    case (a,b,c)
      a)   jmax = n
        t = a(i,k) / a(k,k)
        jmin = k + 1
      b)   jmax = n
        t = a(k,i) / a(k,k)
        jmin = i
      c)   jmax = imax
        t = a(k,i) / a(k,k)
        jmin = i
    end
    for j = jmin to jmax
      a(i,j) = a(i,j) - a(k,j) * t
    end
    b(i) = b(i) - b(k) * t
  end
end
```

The template 13 shows the imprint of matrix type (general, symmetric, symmetric banded) into a process needed for the matrix factorization. The effort required for solving a system of algebraic equations by Gauss elimination can be measured by the number of needed arithmetic operations. Considering one multiplication, or one division, or one addition plus one subtraction as one operation then the the factorization of a full, standardly stored matrix can be secured by $\frac{1}{3}(n^3 - n)$ operations. The operations count for the right-hand side reduction is $\frac{1}{2}(n^2-n)$ while that for the back substitution is $\frac{1}{2}n(n+1)$ operations. The count for matrix factorization is, however, reduced approximately by half if the matrix is symmetric. If the matrix is symmetric and banded then the operations count for the matrix factorization is proportional to $n(nband)^2$ operations instead of n^3 as in the case of factorization of a general type matrix. See [19]. Both symmetry and sparseness should be fruitfully exploited not only in programming but also in storage considerations if efficient storage requirements are to be achieved. There are many storage modes available. Here we mention only a few. A more detail information about the storage modes is in the Chapter 11.

4.1.1 Symmetric matrix storage mode

If the matrix is 'only' symmetric we could store the elements of its upper triangular part (including the diagonal) in a one-dimensional array columnwise, starting with element a_{11}. Of course, the accepted storage mode influences all the subsequent coding of matrix operations. In this case a one-dimensional pointer, say `l`, corresponding to matrix indices `i,j` in its upper triangular part can be easily calculated by `l = i + (j*j - j) / 2`. It is obvious that the memory requirements `nmem = n*(n + 1)/2` are roughly reduced by a factor of two. A similar storage mode is used in IBM programming packages for symmetric matrices. See [1].

4.1.2 Rectangular storage mode – symmetric banded matrices

Also called a compressed diagonal storage mode.

$$\mathbf{R} = \begin{bmatrix} \star & \star & \star & \star & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \star & \star & \star & \star & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \star & \star & \star & \star & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \star & \star & \star & \star & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \star & \star & R_{ij} & \star & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \star & \star & \star & \star & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \star & \star & \star & \star \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \star & \star & \star \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \star & \star \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \star \end{bmatrix}_{n\times n}$$

$$\mathbf{A} = \begin{bmatrix} \star & \star & \star & \star \\ \star & \star & \star & \star \\ \star & \star & \star & \star \\ \star & \star & \star & \star \\ \star & \star & A_{kl} & \star \\ \star & \star & \star & \star \\ \star & \star & \star & \star \\ \star & \star & \star & \cdot \\ \star & \star & \cdot & \cdot \\ \star & \cdot & \cdot & \cdot \end{bmatrix}_{n\times nband}$$

One of possible approaches to an efficient storage mode for symmetric banded matrices, whose bandwidth is fairly constant, is graphically depicted above.

The indices of a general matrix element `R(i,j)` are related to its efficiently stored partner `A(k,l)` by simple relations, i.e. by

`k = i` and `l = j - i + 1`.

If we have a relatively narrow bandwidth then the savings in terms of memory requirements could be substantial, since `nmem = n*nband`. A similar storage mode is used in LINPACK and EISPACK packages. See [17], [10].

One has to realize, that accepting a special storage mode influences all the programming steps dealing with matrix algebra. As an example a procedure for the matrix-vector multiplication taking into account the rectangular storage mode could be as follows. It is written in an old-fashioned programming style, but the procedure works and the reliable procedures should not be changed for formal reasons only.

Program 7

```
      SUBROUTINE DMAVB(A,B,C,N,NDIM,NBAND)
      DOUBLE PRECISION A,B,C,SUM1,SUM2
      DIMENSION A(NDIM,NBAND),B(NDIM),C(NDIM)
C     Multiply a matrix [A] by a vector {b} from the right.
C     It is assumed that the initial square band symmetric matrix is
C     efficiently stored in an rectangular array A(N,NBAND)
C
C     A(N,NBAND)      input matric
C     B(N)            input vector
C     C(N)            output vector C=A*B
C     NDIM            row dimension of [A] matrix in main
C     N               other dimension of [A], dimensions of {B} and {C}
C     NBAND           half-band size (including diagonal)
C
      DO 101 I = 1,N
      SUM1 = 0.D0
      SUM2 = 0.D0
      IF(I .LT. NBAND) GO TO 100
      IF(I .LT. (N-NBAND+2)) GO TO 200
C     Region 3 - I = N-NBAND+2,N
      DO 310 J = I,N
310   SUM1 = SUM1 + B(J)*A(I,J-I+1)
      J1 = I-NBAND+1
      J2 = I-1
      DO 320 J = J1,J2
320   SUM2 = SUM2 + B(J)*A(J,I-J+1)
      GO TO 10
C     Region 1 - I = 1,NBAND-1
100   J2 = I + NBAND - 1
      DO 110 J = I,J2
110   SUM1 = SUM1 + B(J)*A(I,J-I+1)
      IF(I .EQ. 1) GO TO 10
      J2 = I-1
      DO 120 J = 1,J2
120   SUM2 = SUM2 + B(J)*A(J,I-J+1)
      GO TO 10
C     Region 2 - I = NBAND,N-NBAND-1
200   J2 = I + NBAND - 1
      DO 210 J = I,J2
210   SUM1 = SUM1 + B(J)*A(I,J-I+1)
      J1 = I - NBAND + 1
      J2 = I - 1
      DO 220 J = J1,J2
220   SUM2 = SUM2 + B(J)*A(J,I-J+1)
10    C(I) = SUM1 + SUM2
101   CONTINUE
      RETURN
      END
```

□ End of Program 7.

4.1.3 Skyline storage mode

Also called a variable band or profile mode.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 & 0 & 0 & 0 \\ \cdot & a_{22} & a_{23} & 0 & 0 & 0 & 0 & 0 \\ \cdot & \cdot & a_{33} & a_{34} & 0 & a_{36} & 0 & 0 \\ \cdot & \cdot & \cdot & a_{44} & a_{45} & a_{46} & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & a_{55} & a_{56} & 0 & a_{58} \\ \cdot & \cdot & \cdot & \cdot & \cdot & a_{66} & a_{67} & 0 \\ \cdot & \cdot & \text{symm.} & \cdot & \cdot & \cdot & a_{77} & a_{78} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a_{88} \end{bmatrix}$$

If the matrix bandwidth varies dramatically with respect the column index, then – to a creative mind – the columns of nonzero elements could remind a *skyline* of a city which gives the name to the following storage mode. We could have a situation which is illustrated by a small matrix example above. In the computer memory there are stored all zero and non-zero elements between the diagonal and the skyline. The elements are stored in a one-dimensional array, say `w`, columnwise, starting from the diagonal and going upward. Each column is thus stored 'backward' as seen in the following table. Since the individual columns are of different lengths, for a unique one-to-one correspondence of individual elements in both storage modes we need another piece of information related to column lengths. It could be stored in another array, say **m**, containing the pointers to positions of diagonal elements in array **w**.

diag.elem	index k of `w`	index i of **A**	index j of **A**
*	1	1	1
*	2	2	2
	3	1	2
*	4	3	3
	5	2	3
*	6	4	4
	7	3	4
	8	2	4
	9	1	4
*	10	5	5
	11	4	5
*	12	6	6
	13	5	6
	14	4	6
	15	3	6
*	16	7	7
	17	6	7
*	18	8	8
	19	7	7
	20	6	8
	21	5	8

The above table shows the correspondence of indices of vector and matrix representations. The asterisks point to diagonal elements and denote the positions of those elements in **w** array. The auxiliary array is `m = {1 2 4 6 10 12 16 18 22}`. It should be mentioned that the last element of **m** array contains the value corresponding to the number of 'under - skyline' elements plus one. So the position of the `j`-th diagonal element of original matrix in **w** could be found easily by `k1 = m(j)` and the 'height' `ih` of `j`-th column is `ih = m(j+1) - m(j)` and thus `w(k1) = a(j,j)`.

So, for indices `i,j` of **A** matrix, the corresponding `k` index in **w** is `k = m(j) + j - i`. The memory requirements are given by the number of elements between skyline and diagonal plus a small space required for mentioned pointers. For more details see [5].

4.2 Fill-in problem

In many engineering applications the Gauss elimination process starts with a symmetric, banded matrix; that's why we have discussed a few storage modes here. During the elimination process, that leads to upper triangular matrix, the coefficient matrix is subjected to a so-called *fill-in* process. It can be shown [7] that during the elimination process most of zero entries in the coefficient matrix, appearing between the diagonal and the bandwidth (or skyline) boundary, are being *filled in* (overwritten) by nonzero values. There is, however, no fill-in outside the bandwidth or skyline boundary which makes various storage schemes attractive. This also explains why any under-skyline zero element must be kept in memory. It also resolves a general attitude of finite element programmers to special storage modes, typical for sparse matrices which are not densely populated and have no regular pattern of sparseness. There are many storage modes suitable for these matrices [21], however, the computational overhead due to the additional memory management required by the fill-in and fill-out processes during the Gauss elimination is prohibitive. For treatment of randomly sparse matrices without any particular pattern or numerical characteristics see [6].

4.3 Disk storage modes – out-of-core solvers

Today, a user of standard medium-performance workstations could handle his/her matrices in memory sizes of the order of a few gigabytes per processor. But remember that a standard double precision floating point number requires eight bytes to be stored.

These sizes, as impressive as they are compared to memory sizes we were used to a few years ago, still does not cover the memory requirements of the most complicated problems of engineering practice which are measured in millions of 16-decimal-digit unknowns.

Thus, the huge memory problems require a disk storage approach to be employed. The idea is simple, the coefficient matrix is stored on a disk and only its part is being processed in the internal memory at the moment. The efficient implementation is rather difficult, since it is constrained by many factors. One has to take into account the fact that the access and retrieval time of data in a disk storage is of several orders of magnitude greater than that for data in the internal memory. From it follows that the disk algorithm is to be conceived in such a way that the number of data transfers is minimized. The effectiveness of a disk elimination algorithm also strongly depends on the proper exploitation of sparseness of the coefficient matrix.

As an example a so-called *disk band algorithm*, graphically depicted the Paragraph 4.1.2 could be mentioned here. The upper-banded part of the coefficient matrix is stored in a disk file columnwise. The algorithm is based on the fact that at each elimination step only the under-pivot elements are influenced. At each elimination step the algorithm requires to read one column from the disk and to write one row back on the disk storage. The method is very inefficient due to the excessive number of data transfers. For the same storage mode the idea could be substantially improved by treating the larger chunks of data between disk transfers. The working space having the size `nband` by `nband` could be increased to `kk` by `nband`, where `kk` >> `nband` and set generally as large as possible. The detailed description of the algorithm is in [14], the programming considerations and the Fortran program are in [13]. See the Program 8 and the Paragraph 5.15.1.

Program 8

```
      SUBROUTINE DBANGZ(N,K,KK,A,DS,B,EPS,KEY,LKN,FNAM,IRECL,MAXR,NZ)
C
C     THE CATALOG NAME OF THIS SUBROUTINE IS 'S.DBANGZ'
C
C***********************************************************
C*                                                         *
C*   SOLUTION OF A SYSTEM OF LINEAR ALGEBRAIC EQUATIONS    *
C*   WITH POSITIVE DEFINITE SYMMETRIC AND BANDED MATRIX    *
C*   BY DIRECT-ACCESS GAUSS ELIMINATION METHOD.            *
C*   (SUITABLE FOR LARGE SYSTEMS OF EQUATIONS IN FEA)      *
C*                                                         *
C***********************************************************
C
C     * DOUBLE PRECISION VERSION *
C
C     AUXILIARY ARRAY NZ(K) MUST BE DECLARED IN the CALLING PROGRAM
C
C     BEFORE CALLING PROCEDURE THE FILE MUST EXIST ON THE DISC
C     WITH DIRECT ACCESS, LINK NAME 'LKN' AND WITH N RECORDS,
C     WHICH CONTAINS BANDED MATRIX, EFFICIENTLY STORED
C     IN RECTANGULAR FORM.
C -----------------------------------------------------------------
C      DESCRIPTION OF PARAMETERS:
C
C     N.........NUMBER OF EQUATIONS=NUMBER OF RECORDS IN INPUT DISC
C               FILE
C     K.........HALF WIDTH OF BAND OF MATRIX (WITH DIAGONAL)
C     KK........NUMBER OF ROWS OF SYSTEM'S MATRIX,
C               WHICH MAY BE IN MEMORY AT THE SAME TIME
C               K<=KK<=N
C     A(KK,K)...WORKING MATRIX,IN WHICH ROWS OF SYSTEM'S MATRIX
C               ARE HANDLED (IT MUST BE DECLARED IN MAIN)
C     DS(N).....WORKING VECTOR
C               ON INPUT: ARBITRARY
C               ON OUTPUT: VECTOR OF SOLUTION
C     B(N)......THE RIGHT SIDE VECTOR
C     EPS.......IF ELEMENT OF MATRIX IS >= EPS,THEN
C               IT IS TAKEN AS NON-ZERO *** DOUBLE PRECISION ***
C     KEY.......SOLUTION KEY
C                  = 1       REDUCTION OF MATRIX
C                  = 2       REDUCTION OF THE RIGHT SIDE VECTOR
C                            AND BACK SUBSTITUTION
C     LKN.......LINK NAME OF FILE,IN WHICH A MATRIX OF SYSTEM IS STORED
C     FNAM......NAME OF DISC FILE (TYPE CHARACTER - IT MUST BE
C               DECLARED IN MAIN, MAXIMUM IS 12 CHARACTERS)
```

```
C     IRECL.....RECORD LENGHT (IN BYTES)
C     MAXR......MAXIMUM NUMBER OF RECORDS
C     NZ(K).....WORKING VECTOR
C -----------------------------------------------------------------
C
      CHARACTER*(12,V) FNAM
      OPEN(LKN,FILE=FNAM,ACCESS='DIRECT',STATUS='OLD',
     /     RECL=IRECL,MAXREC=MAXR)
C
      DIMENSION A(KK,K),DS(N),B(N),NZ(K)
      DOUBLE PRECISION A,DS,B,RATIO,EPS
C
C     VECTOR OF RIGHT SIDES B INTO WORKING VECTOR DS
      DO 2 I=1,N
2     DS(I)=B(I)
C
      II=KEY
      GO TO (1000,2000), II
C
C     READ FIRST PART OF MATRIX

1000  DO 1 I=1,KK
1     READ(LKN'I) (A(I,J),J=1,K)
C
      JUMP=0
C
C     IEQ...STEP IN GAUSS ELIMINATION
      DO 6 IEQ=1,N
C     WRITE(6,21)
C21   FORMAT(1X,' Intermediate results A(KK,K)'/)
C     DO 22 ID=1,KK
C22   WRITE(6,24) (A(ID,JD),JD=1,K)
C24   FORMAT(1X,10E12.6)
      JUMP=JUMP+1
      IF(JUMP.GT.KK) JUMP=JUMP-KK
      I=0
      DO 29 J=2,K
      IF (ABS(A(JUMP,J)).LT.EPS) GO TO 29
      I=I+1
      NZ(I)=J
29    CONTINUE
      IF (I.EQ.0) GO TO 4
      JUMP1=JUMP-1
      DO 5 L=1,I
      M=NZ(L)
      ITMP=JUMP1+M
      IF(ITMP.GT.KK) ITMP=ITMP-KK
      IF(ABS(A(JUMP,1)).LT.EPS) GO TO 300
      RATIO=A(JUMP,M)/A(JUMP,1)
      IR1=M-1
      DO 3 JC=L,I
      MM=NZ(JC)
      JTMP=MM-IR1
3     A(ITMP,JTMP)=A(ITMP,JTMP)-RATIO*A(JUMP,MM)
5     CONTINUE
C
4     KT=IEQ+KK
```

```
      IF(KT.GT.N) GO TO 6
      KQ=IEQ
      WRITE(LKN'KQ) (A(JUMP,J),J=1,K)
      READ(LKN'KT)(A(JUMP,J),J=1,K)
6     CONTINUE
C
C     RECORD LAST BLOCK OF MATRIX ON DISC
      IND1=(N/KK)*KK+1
      IND2=N
      M=1
      DO 14 I=IND1,IND2
      WRITE(LKN'I) (A(M,J),J=1,K)
      M=M+1
14    CONTINUE
C
      IND1=N-KK+1
      IND2=(N/KK)*KK
      DO 16 I=IND1,IND2
      WRITE(LKN'I) (A(M,J),J=1,K)
      M=M+1
16    CONTINUE
C
C     REDUCTION SUCCESSFULLY ENDED
C
      RETURN
C
C ----------------------------------------------------------------
C
C     REDUCTION OF VECTOR DS
C     READ FIRST PART OF A MATRIX
2000  DO 100 I=1,KK
100   READ(LKN'I)(A(I,J),J=1,K)
C
      JUMP=0
C
      DO 160 IEQ=1,N
      JUMP=JUMP+1
      IF(JUMP.GT.KK) JUMP=JUMP-KK
C
      DO 150 IR=2,K
      IF(ABS(A(JUMP,IR)).LT.EPS) GO TO 150
      IR1=IR-1
      RATIO=A(JUMP,IR)/A(JUMP,1)
      DS(IEQ+IR1)=DS(IEQ+IR1)-RATIO*DS(IEQ)
     IF(ABS(DS(IEQ+IR1)).LT.1.D-30) DS(IEQ+IR1)=0.D0
150   CONTINUE
C
      KT=IEQ+KK
      IF(KT.GT.N) GO TO 160
      READ(LKN'KT)(A(JUMP,J),J=1,K)
160   CONTINUE
C
C     BACK SUBSTITUTION
C
      DS(N)=DS(N)/A(JUMP,1)
      I=N
C
```

```
      DO 9 M=2,KK
      JUMP=JUMP-1
      IF(JUMP.EQ.0) JUMP=KK
      I=I-1
      ITMP=I-1
C
      DO 10 J=2,K
      IF(ABS(A(JUMP,J)).GT.EPS) DS(I)=DS(I)-A(JUMP,J)*DS(ITMP+J)
10    CONTINUE
C
9     DS(I)=DS(I)/A(JUMP,1)
C
      IF(I.EQ.1) RETURN
12    I=I-1
      READ(LKN'I) (A(1,M),M=1,K)
      ITMP=I-1
C
      DO 8 J=2,K
      IF(ABS(A(1,J)).GT.EPS) DS(I)=DS(I)-A(1,J)*DS(ITMP+J)
8     CONTINUE
C
      DS(I)=DS(I)/A(1,1)
      IF(I.GT.1) GO TO 12
      GO TO 200
C
C
300  WRITE(6,310) JUMP,A(JUMP,1)
310  FORMAT(1X,'ERROR-RIGIDITY MATRIX NOT POSITIVE DEFINITE'//
    /       1X,' JUMP=',I6,5X,'A(JUMP,1)=',E14.6)
     STOP 01
C
200  RETURN
     END
```

□ End of Program 8.

Also a so called *hypermatrix algorithm* is of interest. The method consists in subdividing the coefficient matrix into smaller rectangular matrices, sometimes called blocks. The Gauss elimination process is then defined not for matrix entries, but for blocks. So the intermost loop could look like

$\mathbf{K}^{\star}_{ij} = \mathbf{K}_{ij} - \mathbf{K}^{\mathrm{T}}_{is}\mathbf{K}^{-1}_{ss}\mathbf{K}_{sj}$. The method is not limited to symmetric or band matrices. Since it is based on the sequence of matrix operations, standard library routines could easily be employed for its implementation. For details see [8].

4.4 Frontal method

The frontal method is a variant of Gauss elimination. It was first publicly explained by Irons [11]. Today, it bears his name and is closely related to finite element technology. J.K. Reid [16] claims, however, that the method was used in computer programs already in the early sixties.

Its main advantage consists in the fact that the elimination process is based on element matrices and leads to the solution of the system of equations without a necessity to assemble the global coefficient matrix.

Particularly, it is based on following observations

- at a given moment the elimination process is applied only to those local element matrices that are in the same *front* which means that the particular unknowns are 'interrelated'. The partial results of factorization and of right-hand side reduction are temporarily stored in small auxiliary arrays whose dimensions depends on the size of the front, called *frontwidth*.
- the final elimination of the i-th unknown can be achieved as soon as we know that no other, so far untouched, element matrix will contribute to the result.

From it follows that

- the elimination process does not proceed sequentially, starting from the first equation to the last. Instead, the first fully eliminated unknown is that which already does not have any contribution from other unknowns. The similar way we proceed for the remaining equations,
- the i-the unknown being eliminated, it is necessary to remove the corresponding row and the corresponding right-hand side element from auxiliary arrays and store them somewhere else,
- entries related to new unknowns, i.e. those belonging to element matrices that have not yet been taken into account, are immediately being moved to locations that were freed by unknowns eliminated in previous steps.

The effectivity of the frontal algorithm depends on the element numbering. The notions as band and/or profile widths play no role here. There is no decrease of effectivity due to the variable band and/or profile widths as it is in case of band and skyline solvers. In these respects the frontal method outperform other elimination approaches

The detailed explanation and the programming considerations are in [3] and in the Chapter 10.

4.5 Solving Ax = b by inversion

Solving the system of algebraic equation by inversion seems to algorithmically simple, but brings many practical obstacles, namely the greater number of operations nad the fact that the matrices, frequently appearing in FE analysis, lose their initially banded structure. This might be memory prohibiting when solving large systems.

One of the method

There are many methods leading to inverse matrix calculation. Usually there are based on a sort of elimination.

The following variant, capable of inverting regular matrices in place, is shown in the template 14. For efficiency all traditional if statements were removed by explicit definition of limits of loop variables. No spectacular advantage could be achieved by this trick on scalar mode machines, there was, however, a significant improvement by a factor of ten, reported when this code run on a vector machine [9].

Template 14, Inversion of a matrix in place - all if's were removed

```
for i=1 to n
  d = 1/a(i,i); tt = -d
  for j=1 to n
    a(i,j) = a(i,j)*tt
  end
  for k=1 to i-1
    tt = a(k,i)
    for j=1 to i-1
      a(k,j) = a(k,j) + tt*a(i,j)
    end
    for i+1 to n
      a(k,j) = a(k,j) + tt*a(i,j)
    end
    a(k,i)= tt*d
  end
  for k = i+1 to n
    tt = a(k,i)
      for j=1 to i-1
        a(k,j) = a(k,j) + tt*a(i,j)
      end
      for j = i+1 to n
        a(k,j) = a(k,j) + tt*a(i,j)
      end
      a(k,i) = tt*d
  end
  a(i,i) = d
end
```

Program 9

```
function b = inverse(n,s)
% a simple inversion program with all if's removed
% c:\prog_all_backup_zaloha_280105\prog_c\prog\mtl\inverse.m
for i = 1:n
  d = 1/s(i,i);
  tt = -d;
  for j = 1:n
    s(i,j) = s(i,j)*tt;
  end
  for k = 1:i-1
    tt = s(k,i);
    for j = 1:i-1
      s(k,j) = s(k,j) + tt*s(i,j);
    end
    for j = i+1:n
      s(k,j) = s(k,j) + tt*s(i,j);
    end
    s(k,i) = tt*d;
  end
  for k = i+1:n
    tt = s(k,i);
```

```
      for j = 1:i-1
        s(k,j) = s(k,j) + tt*s(i,j);
      end
      for j = i+1:n
        s(k,j)  =  s(k,j) + tt*s(i,j);
      end
      s(k,i) = tt*d;
  end
  s(i,i) = d;
end b = s;
% end of inverse
```

□ End of Program 9.

Solving the set of algebraic equations, the vector of unknowns can be easily achieved by $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. This approach is not recommended but could be accepted if the coefficient matrix is small and if memory and time requirements do not play an important role in our considerations. Using it, however, for large matrices it dramatically spoils the chances to succeed computationally in the task to be solved. The process is more expensive, the operation count is proportional to n^3, which is three times more than that of Gauss elimination, furthermore the inverse matrix loses its bandedness so that no efficient storage scheme could be meaningfully employed. A few years ago, when finite element programmers were mentally withered with limited available memories of computers at hands, they were prone a slightly exaggerated statement that solving the set of algebraic equations by inversion is a computational crime. Today, with huge memories available, one has to admit that for one-shot solutions, made quickly by brute force, the above-mentioned crime should not be punished. Nevertheless, if the inversion is not explicitly needed it should not be computed.

4.6 Jacobi iteration method

The Jacobi method is a representative of one of the oldest iteration methods for the solution of algebraic equations. It is known that its rate of convergence is quite slow, compared to other methods, the method is, however, mentioned here due to its simple structure which is transparently suitable for parallelizing. For convergence considerations see [15], [18].

The method is based on examining each of the n equations, assuming that the remaining elements (unknowns) are fixed, i.e. $x_i = (b_i - \sum_{j \neq i} a_{ij} x_{ij})/a_{ii}$. It should be noted that a regular matrix can always be rearranged in such a way that there it has no zero diagonal entries. The iterative scheme is given by ${}^{(k)}x_j = (b_i - \sum_{j \neq i} a_{ij}{}^{(k-1)}x_j / a_{ii}$, the corresponding template is 15.

Template 15, The Jacobi Method

```
Choose an initial guess $^{(0)}$x to the solution x.
for k=1,2,...
  for i=1,2,...,n
    s(i) = 0   !  initialize the elements of summation array
    for j = 1,2,...,i-1,   i+1,...,n
      s(i) = s(i) + a(i,j)*(k-1)x(j)
    end
    s(i) = (b(i) - s(i)) / a(i,i)
  end
  (k) x = s   !  bold identifiers indicate array data structures check convergence;
  continue if necessary
end
```

One could observe that the order in which the equations are examined is irrelevant and that the equations can be treated independently. The iterative scheme in the Template 15, programmed by means of DO loops in Fortran 77, would run serially. Using FORALL structure of Fortran 90 for the innermost loop as in the Template 2 would allow that these statements could be parallelized and a parallel speed-up be achieved. Sometimes it is claimed [12] that the speed-up is in near direct proportion to the number of processors. The reader should take such a statement with extreme caution.

We have stated that the parallezation is language dependent. Not only that, it also strongly depends on the memory model of the computer being used. The large-scale, massively parallel computers have up to thousands of processors, each having its own memory. Smaller scalable machines have only a few processors and a shared memory. The problem, that has not changed much in the last years, is that it takes far longer to distribute data than it does to do the computation. See the Paragraph 5.14.2.

4.7 Gauss-Seidel method

The Gauss-Seidel method is similar to Jacobi method, it uses, however, the updated values of unknowns as soon as they are available. G. Strang claims that the method was apparently unknown to Gauss and not recommended by Seidel. [18]. The algorithm is in the Template 16.

Template 16, The Gauss-Siedel Method

```
Choose an initial guess $^{(0)}$x to the solution x.
for k=1,2,...
  for i=1,2,...,n
    sum = 0
    for j=1,2,..., i-1
      sum = sum + a(i,j) * (k)x(j)
    end
    for j=i+1,...,n
      sum = sum + a(i,j) * (k-1)x(j)
    end
      (k)x(i) = (b(i) - sum) / a(i,i)
  end
    check convergence; continue if necessary
end
```

The rate of convergence of the Gauss-Seidel method is ussually better than that of Jacobi, still it is relatively slow. It also strongly depends on the ordering of equations, and on the 'closeness' of an initial guess. See [4]. The Gauss-Seidel seems to be a fully sequential method. A careful analysis has shown that a high degree of parallelism is available if the method is applied to sparse matrices arising from the discretized partial differential equations. See [2] and the paragraph 5.14.3.

4.8 Successive overrelaxation method (SOR)

This method is derived from the Gauss-Seidel method by introducing an relaxation parameter for increasing the rate of convergence. For the optimum choice of the relaxation parameter the method is faster than Gauss-Seidel by an order of magnitude. For details see [2], [13] and the paragraph 5.14.4.

Bibliography

[1] *IBM Scientic Subroutine Package.* 112 East Post Road, White Plains, N. Y, 1968.

[2] L. Adams and H. Jordan. Is sor color-blind? *SIAM, Journal of Sci. Stat. Comp.*, (7):490–506, 1986.

[3] D.G. Ashwell and R.H. Gallagher. *Finite Elements for Thin Shells and Curved Members*, chapter Semi Loof Shell Element by Irons, B. M. John Wiley and Sons, New York, 1976.

[4] R. Barrett et al. *Templates for the Solution of Linear Systems.* SIAM, Philadelphia, 1994.

[5] K.J. Bathe and E. L. Wilson. *Numerical Methods in Finite Element Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[6] T.S. Chow and J.S. Kowalik. Computing with sparse matrices. *International Journal for Numerical Methods in Engineering*, 7:211–223, 1973.

[7] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

[8] J. Herzbergen. Iterationverfahren hoeheren ordnung zur einschliessung des inversen einer matrix. *ZAMM*, 69:115–120, 1989.

[9] Y.P. Huang et al. Vectorization using a personal computer. *Developments in Computational Techniques for Structural Techniques Engineering, ed.: Topping, B.H.V., Edinburgh UK, CIVIL-COMP PRESS*, pages 427–436, 1995.

[10] IMLS, Inc, GNB Building, 7500 Bellair BVLD, Houston TX 77036. *EISPACK*, 1989.

[11] B. M. Irons. A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, 2:5–32, 1970.

[12] C.H. Koelbel, D.B. Loveman, R.S Schreiber, G.L. Steele, and M.E. Zosel. *The High Performance Fortran Hanbook.* MIT Press, Cambridge Massachusetts, London England, 1994.

[13] M. Okrouhlík, I. Huněk, and Loucký K. *Personal Computers in Technical Practice.* Institute of Thermomechanics, 1990.

[14] Y. C. Pao. Algorithm for direct access gaussian solution of structural stiffness natrix equation. *International Journal for Numerical Methods in Engineering*, (12):751–764, 1978.

[15] A. Ralston. *A First Course in Numerical Analysis.* McGraw-Hill, Inc., New York, 1965.

[16] J.K. Reid. Algebraic aspects of finite-element solutions. *Computer Physiscs Reports*, (6):385–413, 1987.

[17] Society for Industrial and Applied Mathematics, Philadelphia. *LINPACK User's Guide*, 1990.

[18] G. Strang. *Linear Algebra and its Applications.* Academic Press, New York, 1976.

[19] G. Strang and G. J. Fix. *An analysis of the finite element method.* Prentice Hall, 1973.

[20] I. Szabó. *Einfuhrung in die Technische Mechanik.* Berlin, Springer-Verlag, 1963.

[21] R.P. Tewarson. *Sparse matrices.* Academic Press, New York, 1973.

Chapter 5

How to dirty your hands

This part was written and is maintained by M. Okrouhlík. More details about the author can be found in the Chapter 16.
The solution of

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{5.1}$$

cannot be achieved if the **A** matrix is singular, i.e. if $\det \mathbf{A} = 0$. It is known that a solution for a system with singular matrix still exists if the right hand side vector has a special 'position' in the column space of **A** as shown in [15]. Such a solution is not unique and will be excluded from further considerations – in the text we will mainly deal with systems that are regular, i.e. having the property of $\det \mathbf{A} \neq 0$.

Actually, the boundary between singularity and regularity is not sharp, since we are solving our equations on computers with finite number of significant digits – with computations subjected to round-off errors. See [18] and a nice paper titled *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic* by Prof. W. Kahan provides a tour of some under-appreciated features of IEEE 754 and includes many examples where they clarify numerical algorithms. The Kahan's Notes could downloaded from

`http://grouper.ieee.org/groups/754/.`

It will be shown that it is the condition number of the **A** matrix, defined by

$$c = \| \mathbf{A} \| \, \| \mathbf{A}^{-1} \|,$$

that resolves the solvability of (5.1) rather than the zero or a small value of the matrix determinant.

The matrix singularity has a nice physical interpretation when solving static problems of continuum mechanics by means of finite element method (FEM). Employing the deformation variant of FEM the **A** matrix represents the stiffness matrix of the solved mechanical system, the vector **x** corresponds to unknown generalized displacements, while the components of vector **b** are prescribed generalized forces. The solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$, representing the static conditions of equilibrium, constitutive equations and prescribed boundary conditions, could only be found if the mechanical system is properly fixed to the ground – otherwise stated, if it cannot move as a rigid body. If the system has rigid-body degrees of freedom and is subjected to applied forces, then – according to the Newton's second law – it starts to accelerate which, however, is the task beyond the scope of statics. In mathematics such a situation is dutifully signalized by the matrix singularity. It is worth noticing that nullity of a stiffness matrix is equal to the number of rigid-mode degrees of freedom of the mechanical system.

5.1 Gauss elimination method

The classical Gauss elimination process for the solution of the system of algebraic equations (5.1) is explained in many textbooks [28], [6], [30]. Before trying to program the whole process it is expedient to play with a trivial example with a small number of equations and do all the calculations by hand. See [15].

5.2 Gauss elimination by hand

Gauss elimination method was known long before Gauss. Its usage for three equations with three unknowns was found in Chinese scripts *The Nine Chapters on the Mathematical Art* more than 2000 years ago.

The Gauss elimination consists of two basic steps. The *elimination*, during which the initial matrix is put in an equivalent triangular form, i.e. $\mathbf{A} \rightarrow \mathbf{U}$ and the right hand side vector $\mathbf{b}$ is transformed to $\mathbf{c}$, and the *backsubstitution* by which the individual unknowns are extracted.

☐ **Example**

Solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ for $n = 4$ with

$$\mathbf{A} = \begin{bmatrix} 5 & -4 & 1 & 0 \\ -4 & 6 & -4 & 1 \\ 1 & -4 & 6 & -4 \\ 0 & 1 & -4 & 5 \end{bmatrix}, \qquad \mathbf{b} = \begin{Bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{Bmatrix}.$$

The elimination

is carried out in $(n-1)$ steps.

For $k = 1$ – eliminate the elements under the first diagonal element

The $\mathbf{A}$ matrix is concatenated with the $\mathbf{b}$ vector and the elimination process is carried out simultaneously for both entries.

$$[\mathbf{A}\,\mathbf{b}] = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ -4 & 6 & -4 & 1 & 1 \\ 1 & -4 & 6 & -4 & 0 \\ 0 & 1 & -4 & 5 & 0 \end{bmatrix} \begin{array}{l} \\ \text{second row} - (-\frac{4}{5}) \times \text{first row} \\ \text{third row} - (\frac{1}{5}) \times \text{first row} \\ \text{fourth row} - (0) \times \text{first row} \end{array}$$

and carry out the indicated row operations resulting in

$$[\mathbf{A}^{(1)}\,\mathbf{b}^{(1)}] = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & 1 \\ 0 & -\frac{16}{5} & \frac{29}{5} & -4 & 0 \\ 0 & 1 & -4 & 5 & 0 \end{bmatrix}$$

The element $a_{11}^{(1)} = 5$ (generally $a_{kk}^{(k)}, k = 1, 2, \dots n-1$) is called *pivot*. The multipliers in the k-th elimination step are $n^{(k)} = a_{ik}^{(k-1)}/a_{kk}^{(k-1)}, i = k+1, k+2, \dots n$. The upper right-hand side index is the elimination step counter.

$$\mathbf{N}^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{4}{5} & 1 & 0 & 0 \\ -\frac{1}{5} & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The $\mathbf{A}$ matrix and the $\mathbf{b}$ vector after the first step become $\mathbf{A}^{(1)}$ and $\mathbf{b}^{(1)}$ respectively. In matrix form we have

$$\mathbf{A}^{(1)} = \mathbf{N}^{(1)}\mathbf{A}, \qquad \mathbf{b}^{(1)} = \mathbf{N}^{(1)}\,\mathbf{b}.$$

For $k = 2$ – eliminate elements under the second diagonal element

The pivot for $k = 2$ is $a_{22}^{(2)} = \frac{14}{5}$. The corresponding multipliers, obtained from the condition that the second row elements under the diagonal should become zero, are stored with an opposite sign into second row of $\mathbf{N}^{(2)}$. The zero elements are not printed.

$$\mathbf{N}^{(2)} = \begin{bmatrix} 1 & & & \\ 0 & 1 & & \\ 0 & \frac{16}{14} & 1 & \\ 0 & -\frac{5}{14} & 0 & 1 \end{bmatrix}.$$

The second elimination step could be expressed in the matrix form as

$$\mathbf{A}^{(2)} = \mathbf{N}^{(2)}\mathbf{A}^{(1)} = \begin{bmatrix} 5 & -4 & 1 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} \\ 0 & 0 & -\frac{20}{7} & \frac{65}{14} \end{bmatrix}; \qquad \mathbf{b}^{(2)} = \mathbf{N}^{(2)}\mathbf{b}^{(1)} = \begin{Bmatrix} 0 \\ 1 \\ \frac{8}{7} \\ -\frac{5}{14} \end{Bmatrix}.$$

For $k = 3$ – eliminate elements under the third diagonal element

The pivot for $k = 3$ is $a_{33}^{(3)} = \frac{15}{7}$, the multiplier is $-\frac{20}{15}$ and the matrix of multipliers is

$$\mathbf{N}^{(3)} = \begin{bmatrix} 1 & & & \\ 0 & 1 & & \\ 0 & 0 & 1 & \\ 0 & 0 & \frac{20}{15} & 1 \end{bmatrix}.$$

The sought-after upper triangular matrix $\mathbf{U}$, the result of the elimination process, is

$$\mathbf{U} = \mathbf{A}^{(3)} = \mathbf{N}^{(3)}\mathbf{A}^{(2)} = \begin{bmatrix} 5 & -4 & 1 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} \\ 0 & 0 & 0 & \frac{5}{6} \end{bmatrix},$$

while the right hand side vector $\mathbf{b} \to \mathbf{c}$, where

$$\mathbf{c} = \mathbf{b}^{(3)} = \mathbf{N}^{(3)}\mathbf{b}^{(2)} = \begin{Bmatrix} 0 & 1 & \frac{8}{7} & \frac{7}{6} \end{Bmatrix}^{\mathrm{T}}.$$

The equation $\mathbf{Ux} = \mathbf{c}$, obtained by the elimination process applied to $\mathbf{Ax} = \mathbf{b}$, has the same solution as the original problem. The elimination process, written in matrix form, is

$$\mathbf{U} = \mathbf{N}^{(3)}\mathbf{A}^{(2)} = \mathbf{N}^{(3)}\mathbf{N}^{(2)}\mathbf{A}^{(1)} = \mathbf{N}^{(3)}\mathbf{N}^{(2)}\mathbf{N}^{(1)}\mathbf{A} \tag{5.2}$$

$$\mathbf{c} = \mathbf{N}^{(3)}\mathbf{N}^{(2)}\mathbf{N}^{(1)}\mathbf{b}. \tag{5.3}$$

Back substitution

is – due to the triangular structure of the **U** matrix, simple. The system of equations, after the elimination process is

$$\begin{aligned} 5x_1 - 4x_2 - 1x_3 + 0x_4 &= 0 \\ \frac{14}{5}x_2 - \frac{16}{5}x_3 + 1x_4 &= 1 \\ \frac{15}{7}x_3 - \frac{20}{7}x_4 &= \frac{8}{7} \\ \frac{5}{6}x_4 &= \frac{7}{6} \end{aligned}$$

The last unknown is obtained from the last equation. Being substituted into the last but one equation allows to get the last but one unknown. That way – backwards – we proceed to the first equation. That's why the term back substitution.

$$x_4 = \frac{7}{5}$$

$$x_3 = \frac{\frac{8}{7} - (-\frac{20}{7})x_4}{\frac{15}{7}} = \frac{12}{5}$$

$$x_2 = \frac{1 - (-\frac{16}{15})x_3 - 1x_4}{\frac{14}{5}} = \frac{13}{5}$$

$$x_1 = \frac{0 - (-4)x_2 - 1x_3 - 0x_4}{5} = \frac{8}{5}$$

The **Gauss elimination process**, shown above, could be generalized for the system $\mathbf{Ax} = \mathbf{b}$ with n equations as follows

Triangularization of the matrix

$$\mathbf{A} \to \mathbf{U}; \qquad \mathbf{U} = \mathbf{NA}. \tag{5.4}$$

Reduction of the right hand side vector

$$\mathbf{b} \to \mathbf{c}; \qquad \mathbf{c} = \mathbf{Nb}, \tag{5.5}$$

where the matrix of multipliers $\mathbf{N}$ is

$$\mathbf{N} = \mathbf{N}^{(n-1)}\,\mathbf{N}^{(n-2)} \quad \ldots \quad \mathbf{N}^{(2)}\,\mathbf{N}^{(1)}. \tag{5.6}$$

For the k-th elimination step the matrix $\mathbf{N}^{(k)}$ has the form

$$\mathbf{N}^{(k)} = \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & -n^{(k)}_{k+1,k} & & & \\ & & & -n^{(k)}_{k+2,k} & & & \\ & & & \vdots & \ddots & & \\ & & & & & 1 & \\ & & & -n^{(k)}_{n,k} & & & 1 \end{bmatrix}, \tag{5.7}$$

where the multipliers of the k-th step are

$$n^{(k)}_{i,k} = a^{(k-1)}_{ik} \, / \, a^{(k-1)}_{kk}, \quad i = k+1, k+2, \ldots n. \tag{5.8}$$

It can be shown that the $\mathbf{N}^{(k)}$ matrix has an interesting property – its inverse, say $\mathbf{L}^{(k)}$, is simply obtained by changing the signs of out-of diagonal elements. namely

$$\mathbf{L}^{(k)} = \mathbf{N}^{(k)^{-1}} = \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & n^{(k)}_{k+1,k} & & & \\ & & & n^{(k)}_{k+2,k} & & & \\ & & & \vdots & \ddots & & \\ & & & & & 1 & \\ & & & n^{(k)}_{n,k} & & & 1 \end{bmatrix}. \tag{5.9}$$

In practice the product (5.6) is not explicitly carried out, instead the intermediate partial products $\mathbf{A}^{(k)} = \mathbf{N}^{(k)}\mathbf{A}^{(k-1)}$ and $\mathbf{b}^{(k)} = \mathbf{N}^{(k)}\mathbf{b}^{(k-1)}$ subsequently evaluated and stored. Since the values of the treated arrays are being constantly rewritten, the upper right side index is not actually needed, and the the first two steps of the Gauss process could be written in a simpler way as follows

$$\begin{aligned} &\mathbf{A} \to \mathbf{U} \\ &a_{ij} \leftarrow a_{ij} - a_{kj}a_{ik} \, / \, a_{kk}\,; \\ &j = k+1, k+2, \ldots n;\ i = k+1, k+2, \ldots n;\ k = 1, 2, \ldots n-1. \end{aligned} \tag{5.10}$$

$$\begin{aligned} &\mathbf{b} \to \mathbf{c} \\ &b_i \leftarrow b_i - b_k a_{ik}/a_{kk} \\ &i = k+1, k+2, \ldots n;\ k = 1, 2, \ldots n-1. \end{aligned} \tag{5.11}$$

The memory locations, initially reserved for the upper triangular part of $\mathbf{A}$ matrix, are filled by elements of the matrix $\mathbf{U}$ after the elimination process. They are however are denoted by a_{ij}. The under-diagonal elements could be nulled or the multiplier values could be stored there for later use. It can be shown that

$$\mathbf{L} = \mathbf{N}^{-1} = (\mathbf{N}^{(n-1)}\,\mathbf{N}^{(n-2)} \ldots \mathbf{N}^{(2)}\,\mathbf{N}^{(1)})^{-1} \tag{5.12}$$

and

$$\mathbf{L} = \mathbf{N}^{(1)^{-1}}\,\mathbf{N}^{(2)^{-1}} \ldots \mathbf{N}^{(n-1)^{-1}}. \tag{5.13}$$

It is of interest that the **L** matrix need not be obtained by explicit multiplication (5.13) – due to its special structure it could be assembled by inserting individual columns of $\mathbf{N}^{(k)^{-1}}$ instead. The **L** matrix written in full is

$$\mathbf{L} = \begin{bmatrix} 1 & & & & & \\ n_{21}^{(1)} & 1 & & & & \\ n_{31}^{(1)} & n_{31}^{(2)} & & & & \\ & & \ddots & & & \\ \vdots & & & 1 & & \\ & & & & \ddots & \\ n_{n,1}^{(1)} & n_{n,2}^{(2)} & \ldots & & n_{n,n-1}^{(n-1)} & 1 \end{bmatrix}. \tag{5.14}$$

The Gauss elimination process could thus be viewed as a matrix decomposition written in the form

$$\mathbf{A} = \mathbf{N}^{-1}\,\mathbf{U} = \mathbf{L}\mathbf{U}. \tag{5.15}$$

Then the **U** matrix could be decomposed into $\mathbf{U} = \mathbf{D}\overline{\mathbf{U}}$, where **D** is the diagonal matrix, so $\mathbf{A} = \mathbf{L}\mathbf{D}\overline{\mathbf{U}}$. If the **A** matrix is symmetric, then $\overline{\mathbf{U}} = \mathbf{L}^{\mathrm{T}}$ and we finally have

$$\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^{\mathrm{T}}, \quad \mathbf{U} = \mathbf{D}\mathbf{L}^{\mathrm{T}}. \tag{5.16}$$

Coming back to our numerical example we get

$$\mathbf{L} = \begin{bmatrix} 1 & & & \\ -\frac{4}{5} & 1 & & \\ \frac{1}{5} & 0 & 1 & \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ 0 & 1 & & \\ 0 & -\frac{16}{14} & 1 & \\ 0 & \frac{5}{14} & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ 0 & 1 & & \\ 0 & 0 & 1 & \\ 0 & 0 & -\frac{20}{15} & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & \\ -\frac{4}{5} & 1 & & \\ \frac{1}{5} & -\frac{16}{14} & 1 & \\ 0 & \frac{5}{14} & -\frac{20}{15} & 1 \end{bmatrix},$$

$$\mathbf{D} = \begin{bmatrix} 5 & & & \\ & \frac{14}{5} & & \\ & & \frac{15}{7} & \\ & & & \frac{5}{6} \end{bmatrix},$$

and finally

$$\mathbf{U} = \mathbf{D}\mathbf{L}^{\mathrm{T}}, \qquad \mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^{\mathrm{T}},$$

It could be shown that $\det \mathbf{A} = \det \mathbf{U} = \det \mathbf{D} = \prod_{i=1}^{n} d_{ii} = 5\,\frac{14}{5}\,\frac{15}{7}\,\frac{5}{6} = 25.$

Backsubstitution

could be expressed in the matrix form by

$$\mathbf{U}\mathbf{x} = \mathbf{c}, \quad \mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{c}, \quad \mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{D}^{-1}\mathbf{c}. \tag{5.17}$$

or, on an element level, by

$$x_i = \frac{1}{a_{ii}}(b_i - \sum_{j=i+1}^{n} a_{ij}x_j), \qquad i = n, n-1, \ldots 1. \tag{5.18}$$

End of **Example** □

5.3 Programming Gauss elimination

Two main steps of the Gauss elimination process, i.e. the *elimination* and the *back-substitution* are described here, taking into account the details of programming considerations. Two fundamental steps of this method are formally indicated in the Template 17, where a sort of Pascal-like programming style is simulated.

Template 17, Gauss elimination

```
BEGIN
  ELIMINATION;
  BACKSUBSTITUTION
END;
```

Assuming n by n matrix let's elaborate the `ELIMINATION` 'procedure' at first.

Template 18, Elimination – the first attempt to grasp the problem

```
Use the first equation to eliminate the variable $x_1$ from equations 2 to $n$;
Use the second equation to eliminate the variable $x_2$ from equations 3 to $n$;
Use the third equation to eliminate the variable $x_3$ from equations 4 to $n$;
...
etc.
```

Let's add a few programming details – they are elaborated in three subsequent levels. So far, the declaration details are disregarded.

Template 19, Elimination – the three-level elaboration

The first – individual unknowns are addressed

```
FOR k := 1 TO n-1
    use the k-th equation to eliminate the variable $x_k$ from equations $k + 1$ to $n$;
```

The second – individual equations are addressed

```
FOR k := 1 TO n-1 DO
    FOR i := k+1 TO n DO
        Use the k-th equation to eliminate the variable $x_k$ from i-th equation;
```

The third. What do we mean by elimination of x_k from the i-th equation? Actually, the following:

```
    ($i$-th equation) $\leftarrow$ ($i$-th equation) – $a[i,k]/a[k,k]*$($k$-th equation);
```

The last 'megastatement' in Template 19, containing the term 'equation', requires actually addressing the coefficients of the **A** matrix from (5.1). So far we used k counter for addressing unknown variables, i counter for equations. Let's add another counter, say j, to address individual elements 'alongside' the i-th equation.

The next logical step is sketched in the Program 10, this time using proper Pascal statements. It should be noticed that the right hand side elements are processed at the same time as well. As usual in programming practice the elimination is carried out 'in place', destroying (overwriting) thus the original values of matrix **A** and vector **b** elements.

Program 10

```
Procedure ELIMINATION;
 BEGIN FOR  k := 1  TO  n-1  DO
  FOR i := k+1  TO  n  DO
    BEGIN
      FOR j := k+1  TO  n  DO  a[i,j] := a[i,j] - a[i,k]/a[k,k]*a[k,j];
      b[i] := b[i] - a[i,k]/a[k,k]*b[k]
    END
 END;
```

End of Program 10. □

Let's improve the preceding reasoning a little bit.

First, the repeated multiplication by $a[i,k]/a[k,k]$, being carried out within the j-loop, actually does not depend on the j-counter and could thus be computed in advance – outside that loop. An auxiliary variable, say t, might serve the purpose.

Second, there is a repeated division by the $a[k,k]$ within the inner-most loop. This diagonal element is often called *pivot*. It should be emphasized that the current $a[k,k]$ does not contain the value of the original matrix element. The division by the pivot which has zero or a 'small' value would result in erroneous overflow condition. To prevent this we should stop the execution of the program if this might happen. Let's define a suitable threshold variable, say ε – or `eps` in the program, and change our procedure as follows.

Program 11

```
Procedure ELIMINATION;
 BEGIN
 ier := 0;
 FOR  k := 1  TO  n-1  DO
   IF ABS(a[k,k]) >  eps
    THEN
       BEGIN
       FOR i := k+1  TO  n  DO
         IF a[i,k] <> 0
           THEN
           BEGIN
             t := a[i,k]/a[k,k];
             FOR j := k+1  TO  n  DO  a[i,j] := a[i,j]-t*a[k,j];
             b[i] := b[i] - t*b[k]
           END
       END
    ELSE ier := -1
 END;
```

End of Program 11. □

The error parameter `ier`, being initially set to zero, was introduced. If the execution of statement `IF ABS(a[k,k])> eps` gives the true value, everything is O.K. If it gives the false value, the procedure sets the value of the error parameter to -1, indicating thus that something is wrong. The execution of the elimination process is interrupted, the remaining computation skipped and it is up to the user to decide what to do next. This is not an approach with which we could be be satisfied. But we will improve it later.

Notice another statement, i.e. `IF a[i,k]<>0`, preventing the useless multiplication by zero, saving thus the computing time.

Having finished the elimination part, the processed matrix has a triangular structure. The elimination process is often called *triangularization*, *factorization* or *reduction*.

$$\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}\,x_2 & + & \cdots & & + & a_{1n}\,x_n & = & b_1 \\
 & & a_{22}\,x_2 & + & \cdots & & + & a_{2n}\,x_n & = & b_2 \\
 & & & & & & & & \ldots & \\
 & & & & & a_{n-1,n-1}\,x_{n-1} & + & a_{n-1,n}\,x_n & = & b_{n-1} \\
 & & & & & & & a_{nn}\,x_n & = & b_n.
\end{array} \tag{5.19}$$

The best way to evaluate the unknowns, contained in $\mathbf{x}$, is to proceed backwards – from the last equation to the the first. That's why the term *backsubstitution* is used. It should also be reminded that the elimination process – as it is conceived above – is processed 'in place'. The initial values of the upper part of $\mathbf{A}$ are lost, being replaced by new ones, while the those of its lower part are unchanged. The original values of the right-hand side are lost as well, being replaced by new ones. Denoting the upper triangular part of $\mathbf{A}$ by $\mathbf{U}$ and the changed elements of $\mathbf{b}$ by $\mathbf{c}$ one can symbolically write

$$\mathbf{A} \rightarrow \mathbf{U}, \quad \mathbf{c} \rightarrow \mathbf{b}. \tag{5.20}$$

The beauty and efficiency of the elimination and back substitution processes is based upon the fact that the equations $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{c}$ have the same solutions – the latter one being solvable more easily. We start with the last equation where there is just one unknown, i.e. x_n. After being extracted, it could be substituted into the last but one equation for solving the unknown x_{n-1}. Similarly, we could proceed 'up' to the first equation. The back substitution process could be expressed by the relation

$$x_i = (b_i - \sum_{j=i+1}^{n} a_{ij}\, x_j)/a_{ii}, \quad i = n, n-1, \cdots 1, \tag{5.21}$$

that could be programmed as follows.

Program 12

```
Procedure BACKSUBSTITUTION;
 BEGIN
   FOR i := n  DOWNTO  1  DO
     BEGIN
       sum := 0;
       FOR j := i+1  TO  n  DO  sum := sum + a[i,j]*x[j];
       x[i] := (b[i] - sum)/a[i,i]
   END
 END;
```

End of Program 12. □

If there appears a zero pivot during the elimination process we cannot proceed the way indicated so far. There is, however, a way. If the equation (5.1) has a unique and nontrivial solution we could circumvent the division by zero by reordering the remaining equations. Usually, the equation with zero pivot is replaced by that having the element - in the column under the faulty pivot – with the greatest absolute value.

$$newpivot \leftarrow \max_{i=k,k+1,\cdots n} |a_{ik}|. \tag{5.22}$$

This is called the *partial pivoting.* This process does not change the order of variables in contradistinction to the process called the *full pivoting* where the search for a suitable pivot is carried out within all the remaining matrix elements.

$$newpivot \leftarrow \max_{i=k,k+1,\cdots n,\, j=k,k+1,\cdots n} |a_{ij}|. \tag{5.23}$$

In other cases, when the system of equations does not have a solution or has infinitely many of them, the elimination process necessarily fails and cannot be resuscitated by any, whatever ingenious, pivoting. Sources of troubles should then be found. They usually comes from erroneous input data and/or from the physical nature of the solved problem.

Solving the systems of equations, having their origin in continuum mechanics tasks by finite element method, we are lucky since in most cases our matrices are symmetric and banded. Furthermore they often positive definite – this property guarantees that all the pivots are positive and – in most cases – no pivoting is required. See [30].

One has to proceed carefully during the pivoting process. Let's show it by means of the following trivial example.

□ **Example.**

$$\begin{aligned} 0.0001x_1 + x_2 &= 1 \\ x_1 + x_2 &= 2. \end{aligned} \tag{5.24}$$

The exact solution is

$$\begin{aligned} x_1 &= 1.0001 \\ x_2 &= 0.9999. \end{aligned} \tag{5.25}$$

Assume that we have a computer working with a three-digit arithmetics. The numerical results are rounded, not cut. Multiplying the first equation by −10000 and adding it to the second we get (exactly)

$$-9999x_2 = -9998. \tag{5.26}$$

From this, in three-digit arithmetics, we get

$$x_2 = 1.00 \tag{5.27}$$

and substituting this into the first one

$$x_1 = 0.00, \tag{5.28}$$

we get a completely wrong result. Reordering the equations (pivoting) we get

$$\begin{aligned} x_1 + x_2 &= 2 \\ 0.0001x_1 + x_2 &= 1 \end{aligned} \tag{5.29}$$

and solving it we obtain

$$\begin{aligned} x_1 &= 1.00 \\ x_2 &= 1.00, \end{aligned} \tag{5.30}$$

which, considering the employed three-digit arithmetics, is the acceptable result. The moral of the story is that one has to avoid not only zero pivots, but the small ones as well.

End of Example. □

The influence of pivoting on the propagation of errors during the Gauss elimination was studied by John von Neumann a H.H. Goldstine [14] back in 1947. Their à-priory approach to the error analysis lead to very pessimistic conclusions. They predicted that elimination methods would fail when attempting to solve large systems of equations.

This topics was treated later – à-posteriori approaches prevailed – and it was found that that the round-off errors do not sum up during the elimination process, rather they have a tendency to cancel out themselves. See [34].

A few notes to remember

- The *full pivoting* spoils the matrix symmetry and quadruples the number of necessary floating point operations needed for the elimination process.
- The *partial pivoting* does not change the order of equations and requires only a slightly greater number of operations than that with no pivoting at all.
- Solving the system with a positive definite matrix requires no pivoting. All the pivots are positive.

More details about the partial pivoting

If, during the k-the step of the elimination process, there is a zero pivot, then we start to look for a new candidate in the column immediately bellow the a_{kk} element, i.e. from rows $k+1, k+2, \cdots n$. We will opt for an element having in absolute value the greatest magnitude. It might happen that all the potential candidates will have a value zero or be in absolute value smaller than the chosen threshold ε. In such a case the system has no solution, or has infinitely many solutions – the matrix is singular (actually or computationally) and the execution of the program has to be stopped.

Let's define a new boolean variable, say `singularity`, having the `true` value in case the matrix is singular and modify our previous program as follows

Template 20, Improved elimination

```
BEGIN
   ELIMINATION;
   IF singularity THEN writeln (Error – matrix is singular);
                  ELSE BACKSUBSTITUTION
END.
```

Now, insert the matrix reordering process together with singularity checks into the Elimination procedure.

Template 21, Reorder equations and check the singularity condition

```
k := 1
REPEAT
   Reorder equations and check for the singularity;
   IF NOT singularity THEN
      eliminate $x_k$ from equations $k+1$ to $n$;
      k := k+1
UNTIL $(k = n)$ OR singularity;
```

Now, we are looking for a suitable pivot. If the bellow-the-pivot element, having in absolute value the greatest magnitude, is located in the l-th row, then we could proceed followingly

Template 22, Search for a suitable pivot

```
Start with l := k;
FOR i := k+1 TO n DO
     IF ABS (a[i,k]) > ABS (a[l,k]) THEN l := i;
```

After the cycle is finished the variable l points to the row, containing the new pivot. If, after the cycle is finished, the variable a_{lk} contains zero or a value (in absolute value) smaller than the threshold ε, we conclude that there is no suitable pivot available and that the matrix is singular.

Knowing where the best pivot candidate resides, we can exchange equations.

Template 23, Exchange equations

```
IF ABS (a[l,k]) < $\varepsilon$ THEN singularity := true
                    ELSE IF $k$ <> $l$ THEN exchange equations $k$ and $l$;
```

By exchanging equations k and l is actually understood exchanging their elements. In the k-th step of elimination the unknowns $x_1, x_2, \cdots x_{k-1}$ has already been eliminated – what remains to exchange are elements from the k-th to n-th position.

Template 24, Exchange matrix and vector elements

```
BEGIN
   FOR j := k TO n DO exchange matrix elements (a[k,j], a[l,j]);
   exchange right-hand side elements (b[k], b[l])
END;
```

The Exchange procedure could be as follows

```
Procedure EXCHANGE (VAR x,y:  real);
  VAR t:real;
  BEGIN t := x; x := y; y := t; END;
```

Having come to this point there is still a possibility that the zero or small pivot resides in the last equation, i.e. in $a_{nn}\, x_n = b_n$. This might be treated by the statement appearing in the Template 25.

Template 25, The last equation singularity treatment

```
IF NOT singularity THEN singularity := ABS (a[n,n]) < $\varepsilon$;
```

The last statement secures that the pivot a_{nn} is tested after all the previous $n-1$ equations have been successfully treated, i.e. with no singularity detected.

In the Program 13 all pieces of the Pascal puzzle are put together.

Program 13

```
PROGRAM PIVOT;
 CONST  N=4; EPS=1E-6;
 TYPE   RMAT= ARRAY[1..N,1..N] OF REAL;
        RVEC= ARRAY[1..N] OF REAL;
 VAR    A,AC         : RMAT;
        X,B,BC,R     : RVEC;
        I,J,K        : INTEGER;
        SINGULARITY  : BOOLEAN;

 PROCEDURE EXCHANGE(VAR X,Y:REAL);
 VAR T : REAL;
 BEGIN T := X; X := Y; Y := T END;

 PROCEDURE REORDER;
 VAR    I,J,L        : INTEGER;
 BEGIN
  L:=K;
  FOR I:=K+1 TO N DO   (* Find the greatest sub-pivot element *)
    IF(ABS(A[I,K]) > ABS(A[L,K])) THEN L:=I;
    IF ABS(A[L,K]) < EPS   (* and check whether it is not too small *)
     THEN SINGULARITY := TRUE
     ELSE IF K<>L THEN BEGIN (* REORDER *)
                       FOR J := K TO N DO EXCHANGE(A[K,J],A[L,J]);
```

```
                      EXCHANGE(B[K],B[L])
                      END
END;

PROCEDURE ELIMINATION;
VAR I,J        : INTEGER;
    T          : REAL;
BEGIN
K := 1;
SINGULARITY := FALSE;
REPEAT
 REORDER;
 IF NOT SINGULARITY THEN (* ELIMINATION itself *)
    FOR I := K+1 TO N DO
    BEGIN
      T := A[I,K]/A[K,K];
      FOR J := K+1 TO N DO
        A[I,J] := A[I,J] - T*A[K,J];
        B[I] := B[I] - T*B[K];
    END;
K := K+1
UNTIL (K = N) OR SINGULARITY;
END;

PROCEDURE BACKSUBSTITUTION;
VAR SUM    : REAL;
    I,J    : INTEGER;
BEGIN
 FOR I := N DOWNTO 1 DO
   BEGIN
     SUM := 0;
     FOR J := I+1 TO N DO SUM := SUM + A[I,J]*X[J];
     X[I] := (B[I] - SUM)/A[I,I]
   END
END;

BEGIN (* main *)
 (* Matrix *)
 A[1,1]:=2; A[1,2]:=4; A[1,3]:=1; A[1,4]:=2;
 A[2,1]:=1; A[2,2]:=2; A[2,3]:=2; A[2,4]:=3;
 A[3,1]:=2; A[3,2]:=1; A[3,3]:=3; A[3,4]:=3;
 A[4,1]:=1; A[4,2]:=3; A[4,3]:=1; A[4,4]:=2;
 (* Right-hand side vector *)
 B[1]:=17; B[2]:=16; B[3]:=16; B[4]:=14;
 (* make a copy *)
 AC := A; BC := B;
 (* start *)
 ELIMINATION;
 IF SINGULARITY
    THEN WRITELN (* matrix is singular *)
    ELSE BACKSUBSTITUTION;
 (* if the matrix is not singular, print the result *)
 (* and compute the residuum *)
 IF NOT SINGULARITY THEN
    BEGIN
      WRITELN ('result');
      FOR I := 1 TO N DO WRITELN(I:2,X[I]:12:6)
```

```
        ;
        (* residuum (R)=[A](X)-(B) *)
        FOR I := 1 TO N DO
          BEGIN
            R[I] :=0;
            FOR J := 1 TO N DO R[I] := R[I] + AC[I,J]*X[J];
            R[I] := R[I] - BC[I]
          END;
        (*print *)
        WRITELN ('residuum');
        FOR I:=1 TO N DO WRITELN (I:2,R[I]:12:6)
      END
END. (* of main *)
```

End of Program 13. □

Before executing the Program 13 one has to choose a suitable value for the threshold variable `EPS`. Evidently, this can be neither 'pure' zero nor a value of the order of the *unit roundoff* or of the *machine epsilon*.

The unit roundoff is the smallest positive number, say `u`, for which `1 + u` is different from 1. There is a similarly defined quantity, the *machine epsilon*, often denoted `machep = a - 1`, where `a` is the smallest representable number greater than 1. For binary computers with rounding we usually have `machep = 2*u`. In many texts this distinction is not observed, since both values are very small and furthermore of the same order. For more details see a nice book, written by one of the founding fathers of the numerical analysis and Matlab, namely C. Moler, where – among others – the floating point arithmetics is explained in a way that nobody could misunderstand it[1]. It could be downloaded from

`www.mathworks.com/moler.`

The previous paragraph should be complemented by a few words about the representation of real numbers on the computer together with a reminder that the term *real numbers* is currently being used in two different and rather contradictory meanings. First, in mathematics, the term denotes the positive and negative numbers, together with zero, but excluding the imaginary numbers. They range from $-\infty$ to $+\infty$ and completely fill the number axis. Second, in the computer jargon, the same term actually denotes the *numbers of the real type*, that could be represented on a computer using the finite number of bits for their internal representation. There is a finite number of numbers that could be represented this way, they are unequally spaced on the number axis and furthermore their range is limited as schematically indicated in Fig. 5.1. Today, the internal storage of integer and floating point numbers is standardized by IEEE standards. For more details see

`www.psc.edu/general/software/packages/ieee/ieee.php`

An amusing paper about floating point intricacies titled *How Futile are Mindless Assessments of Roundoff in Floating point computation?* could be downloaded from

`www.cs.berkeley.edu/~wkahan/Mindless.pdf.`

[1]Murphy law: If you explain something so clearly that nobody could misunderstand it, then there is always somebody who will.

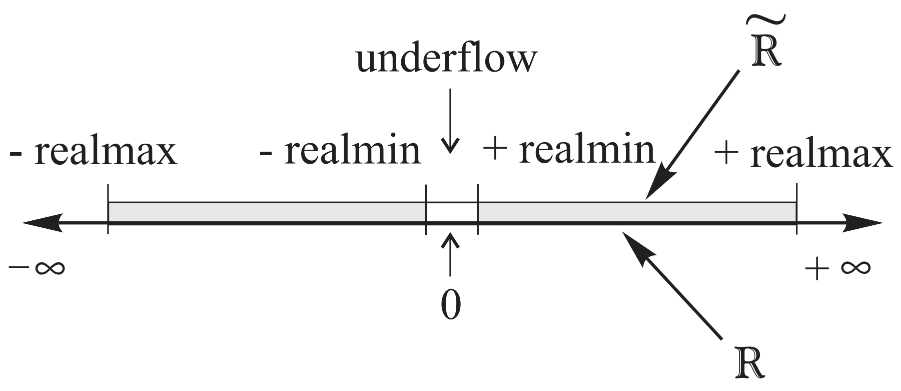


Figure 5.1: Real numbers $\mathcal{R}$ and numbers of real type $\tilde{\mathcal{R}}$

The values of unit roundoffs and those of maximum represented numbers for a single and double precision are listed in the following table.

precision	**unit roundoff**	**max number**
single	6×10^{-8}	10^{38}
double	2×10^{-16}	10^{308}

The author of [28] recommends to set the elimination threshold as `EPS = n*machep`, where `n` is the order of the matrix being solved.

5.4 Error estimation for $\mathbf{Ax} = \mathbf{b}$

The solution of Eq. (5.1) could formally be expressed as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Generally, the input values, i.e. the values of $\mathbf{A}$ and of $\mathbf{b}$ are known with a limited precision. Our goal is to find out how the error of the output, i.e. of the solution $\mathbf{x}$, is influenced by uncertainties of input data.

At first, let's assume that it is only the right-hand side vector which is subjected to errors, say $\Delta\mathbf{b}$. Then the solution of Eq. (5.1), instead of $\mathbf{x}$ will be $\mathbf{x} + \Delta\mathbf{x}$

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

Substituting the last equation from (5.1) we get

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

Assuming that $\mathbf{A}$ is regular we can write

$$\Delta\mathbf{x} = \mathbf{A}^{-1}\,\Delta\mathbf{b}.$$

Applying norms we get

$$\begin{aligned} \|\Delta\mathbf{x}\| &= \|\mathbf{A}^{-1}\Delta\mathbf{b}\|, \\ \|\Delta\mathbf{x}\| &\le \|\mathbf{A}^{-1}\|\|\Delta\mathbf{b}\|. \end{aligned} \tag{5.31}$$

Since $\mathbf{b} = \mathbf{Ax}$ we can also write

$$\|\mathbf{b}\| \le \|\mathbf{A}\|\|\mathbf{x}\|. \tag{5.32}$$

Multiplying the second equation of (5.31) by (5.32) we get

$$\|\Delta\mathbf{x}\|\,\|\mathbf{b} \leq \|\mathbf{A}\|\,\|\mathbf{A}^{-1}\|\,\|\mathbf{x}\|\,\|\Delta\mathbf{b}\|.$$

Assuming a nonzero vector $\mathbf{b}$, whose norm is naturally $\|\mathbf{b}\| \neq 0$, the previous equation could be rearranged into

$$\frac{\|\Delta\mathbf{x}\|}{|\mathbf{x}\|} \leq \|\mathbf{A}\|\,\|\mathbf{A}^{-1}\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}. \tag{5.33}$$

The matrix norm product in (5.33) is called the *condition number* and could be denoted by

$$c(\mathbf{A}) = \|\mathbf{A}\|\,\|\mathbf{A}^{-\mathbf{1}}\|. \tag{5.34}$$

The condition number is defined for square matrices only and depends on the type of norm utilized. Since we are looking for an error estimation the type of the norm being used does not play a significant role. Rather, we should be interested in obtaining the norm of the inverse matrix without inverting it. The inequality (5.32) could be rewritten into

$$\frac{\Delta\|\mathbf{x}\|}{\|\mathbf{x}\|} \leq c(\mathbf{A})\,\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}. \tag{5.35}$$

The relation $\|\Delta\mathbf{b}\| \,/\, \|\mathbf{b}\|$ from (5.32) represents the relative error of the right hand side and similarly, the term $\|\Delta\mathbf{x}\| \,/\, \|\mathbf{x}\|$ corresponds to the relative error of the result. The condition number $c(\mathbf{A})$ could then be interpreted as the amplification factor of the right hand side errors. Since $c(\mathbf{A}) \geq \mathbf{1}$, it is obvious that the accuracy of the result can never be better than the accuracy with which we know the right hand-side-element values.

Now, let's analyze what happens if the matrix elements are subjected to errors. Starting with

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \text{and} \quad (\mathbf{A} + \boldsymbol{\Delta}\mathbf{A})\,(\mathbf{x} + \boldsymbol{\Delta}\mathbf{x}) = \mathbf{b},$$

then by subtracting and rearrangement we get

$$\Delta\mathbf{x} = -\mathbf{A}^{-1}\,\Delta\mathbf{A}\,(\mathbf{x} + \Delta\mathbf{x}).$$

Applying the norms

$$\|\Delta\mathbf{x}\| \leq \|\mathbf{A}^{-1}\|\,\|\Delta\mathbf{A}\|\,\|\mathbf{x} + \Delta\mathbf{x}\|$$

and rearranging we get

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x} + \Delta\mathbf{x}\|} \leq \|\mathbf{A}^{-1}\|\,\|\Delta\mathbf{A}\| = c(\mathbf{A})\frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\;|}. \tag{5.36}$$

One can state that the relative error of the result is bounded by the relative error of matrix elements multiplied by the condition number of the matrix.

Large value of the condition number signifies that the matrix is badly conditioned and that the results of (5.1) will be subjected to significant errors. At the same time, the determinant value need not, however, be 'small'. It should be emphasized that a small determinant value does not indicate that the matrix is 'nearly' singular.

□ **Example**

The solution of $\mathbf{Ax} = \mathbf{b}$ with

$$\mathbf{A} = \begin{bmatrix} 5 & 1 \\ 4 & 6 \end{bmatrix}, \qquad \mathbf{b} = \left\{ \begin{array}{c} 1 \\ 32 \end{array} \right\}$$

is

$$\left\{ \begin{array}{c} x_1 \\ x_2 \end{array} \right\} = \left\{ \begin{array}{c} -1 \\ 6 \end{array} \right\}.$$

The matrix determinant $\det \mathbf{A} = 26$, while the matrix column norm is $\|\mathbf{A}\| = 9$. The inverse matrix, computed by hand, is

$$\mathbf{A}^{-1} = \frac{1}{\det \mathbf{A}} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} = \frac{1}{26} \begin{bmatrix} 6 & -1 \\ -4 & 5 \end{bmatrix}.$$

The column norm of the inverse matrix is

$$\|\mathbf{A}^{-1}\| = 10/26.$$

The condition number, computed from (5.31), is $c(\mathbf{A}) = 90/26$. It is known that the solution of (5.1) does not change if all the matrix elements, as well as the elements of the right hand side, are multiplied by a constant, say 10^{-6}. From the physical point of view it only means that the variables and constants, we are we using for the computation are expressed in different physical units. Now, our equation has the form

$$\begin{bmatrix} 5 \times 10^{-6} & 1 \times 10^{-6} \\ 4 \times 10^{-6} & 6 \times 10^{-6} \end{bmatrix} \left\{ \begin{array}{c} x_1 \\ x_2 \end{array} \right\} = \left\{ \begin{array}{c} 1 \times 10^{-6} \\ 32 \times 10^{-6} \end{array} \right\}.$$

The matrix determinant is then 10^{-12} times smaller, while the condition number, computed from (5.31), does not change. Nor does change the result, i.e. $\{x\} = \{-1, 6\}^{\mathrm{T}}$.

Let's continue in our example by changing the value of an element, say a_{11}, by 1%. The solution of (5.1) with

$$\mathbf{A} = \begin{bmatrix} 5.05 \times 10^{-6} & 1 \times 10^{-6} \\ 4 \times 10^{-6} & 6 \times 10^{-6} \end{bmatrix}, \qquad b = \left\{ \begin{array}{c} 1 \times 10^{-6} \\ 32 \times 10^{-6} \end{array} \right\}.$$

is

$$\left\{ \begin{array}{c} x_1 \\ x_2 \end{array} \right\} = \mathbf{A}^{-1} \left\{ \begin{array}{c} b_1 \\ b_2 \end{array} \right\} = \left\{ \begin{array}{c} -0.9885931558935361\ldots \\ 5.992395437262357\ldots \end{array} \right\}.$$

A small change of input data values caused a similarly small change of result values. We say that such a system (matrix) is well conditioned. An obvious conclusion is that a small determinant value is immaterial[2] provided that the matrix is *well conditioned.*

It remains to show what is a *badly conditioned* matrix. It is a matrix whose columns are 'nearly' linearly dependent. Remaining with trivial examples, we could take a sort of insight from finding the intersection of two 'nearly' parallel lines depicted in Fig. 5.2. The equations are

[2]We are tacitly assuming that the floating point computation is employed. It should be reminded that the above conclusion would not be valid for a system working with fixed point representation of real numbers, which is typical for real time operating systems

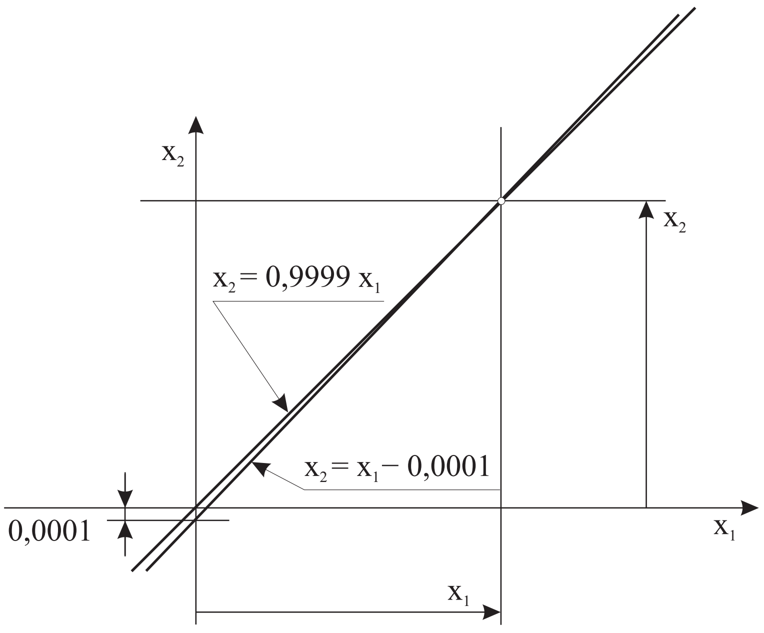


Figure 5.2: Two nearly parallel lines

$$\begin{aligned} x_1 - x_2 &= 0.0001, \\ 0.9999x_1 - x_2 &= 0. \end{aligned}$$

The matrix and its determinant are

$$\mathbf{A} = \begin{bmatrix} 1 & -1 \\ 0.9999 & -1 \end{bmatrix}, \quad \det \mathbf{A} = 0.0001.$$

The inverse matrix is

$$\mathbf{A}^{-1} = -10000 \begin{bmatrix} -1 & 1 \\ -0.9999 & 1 \end{bmatrix}.$$

The column norms and the condition number are

$$\|\mathbf{A}\| = 2, \qquad \|\mathbf{A}^{-1}\| = 20000, \qquad c(\mathbf{A}) = 40000.$$

The solution is

$$\left\{ \begin{array}{c} x_1 \\ x_2 \end{array} \right\} = \mathbf{A}^{-1} \left\{ \begin{array}{c} b_1 \\ b_2 \end{array} \right\} = \begin{bmatrix} 10000 & -\ 10000 \\ 9999 & -\ 10000 \end{bmatrix} \times \left\{ \begin{array}{c} 0.0001 \\ 0 \end{array} \right\} = \left\{ \begin{array}{c} 1 \\ 0.9999 \end{array} \right\}.$$

Again, let's observe the influence of a small change of matrix element on the solution. In this case

$$\mathbf{A} = \begin{bmatrix} 1.01 & -1 \\ 0.9999 & -1 \end{bmatrix}, \quad \mathbf{b} = \left\{ \begin{array}{c} 0.0001 \\ 0 \end{array} \right\},$$

then $\det \mathbf{A} = -0.0101$ and

$$\mathbf{A}^{-1} = -99.00990099 \begin{bmatrix} -1 & 1 \\ -0.9999 & 1.01 \end{bmatrix}.$$

The solution is

$$\left\{ \begin{array}{c} \bar{x}_1 \\ \bar{x}_2 \end{array} \right\} = \mathbf{A}^{-1} \left\{ \begin{array}{c} b_1 \\ b_2 \end{array} \right\} = \left\{ \begin{array}{c} 0.00990099\cdots \\ 0.00990000\cdots \end{array} \right\}.$$

Now, the 1% change of input data leads to hundredfold change of the result. Geometrical explanation is obvious – a small change of the gradient of one line, being closely parallel to the other line, significantly relocates their intersection.

End of Example. □

5.5 Condition number computation

Using the definition of the condition number by (5.31) for its computation requires to invert the original matrix and then compute its norm. For n by n matrix additional $n^3 + 2n^2$ floating point operations are approximately needed. To circumvent this hindrance we are looking for alternative approaches.

An approximate computation of the condition number could be based upon the following reasoning – see [28].

Let $\mathbf{w}$ and $\mathbf{y}$ are n-element vectors for which the following relation is satisfied

$$\mathbf{w} = \mathbf{A}^{-1}\mathbf{y}.$$

Applying norms we get

$$\|\mathbf{w}\| \leq \|\mathbf{A}^{-1}\|\,\|\mathbf{y}\|,$$

from which we get

$$\|\mathbf{A}^{-1}\| \geq \|\mathbf{w}\|/\|\mathbf{y}\|.$$

We can take an arbitrary sequence of vectors $\mathbf{y}^{(i)}, i = 1, 2, \cdots k, k < n$ and apply the Gauss elimination process to

$$\mathbf{A}\,\mathbf{w}^{(i)} = \mathbf{y}^{(i)}$$

for solving vectors $\mathbf{w}^{(i)}$. The approximate norm of the inverse matrix, as shown in [6], could than be taken as

$$\|\mathbf{A}^{-1}\| = \max(\|\mathbf{w}^{(i)}\|/\|\mathbf{y}^{(i)}\|). \tag{5.37}$$

Practical experience shows that the approximate ratio of arbitrary vectors $\|\mathbf{w}\|/\|\mathbf{y}\|$ processed by (5.34) converges to $\frac{1}{2}\|\mathbf{A}^{-1}\|$.

This approach is advocated in the classical book by Forsythe, Malcolm and Moler, see [6]. A collection of Fortran90 procedures for mathematical computation, based on the ideas from the book, could be downloaded from

`http://www.pdas.com/programs/fmm.f90.`

A cheep estimate of the norm of the inverse matrix is based on the following reasoning. Denoting by $\bar{\mathbf{x}}$ the computed solution of $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ then

$$\bar{\mathbf{x}} = \mathbf{A}^{-1}\mathbf{b},$$

then

$$\|\bar{\mathbf{x}}\| \leq \|\mathbf{A}^{-1}\|\,\|\mathbf{b}\|$$

and the estimate of the inverse matrix norm is

$$\|\mathbf{A}^{-1}\| \leq \|\bar{\mathbf{x}}\|/\|\mathbf{b}\|. \tag{5.38}$$

5.6 Other ways to estimate the condition number

Assume that solving $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ by Gauss elimination we got the solution $\bar{\mathbf{x}}$. We could thus compute a residual vector

$$\bar{\mathbf{r}} = \mathbf{b} - \mathbf{A}\bar{\mathbf{x}}, \tag{5.39}$$

which would be zero provided that there were no round-off errors. Rearranging we get

$$\bar{\mathbf{r}} = \mathbf{A}(\mathbf{x} - \bar{\mathbf{x}}) = \mathbf{A}\bar{\mathbf{e}}, \tag{5.40}$$

where we defined the absolute error by

$$\bar{\mathbf{e}} = \mathbf{x} - \bar{\mathbf{x}}. \tag{5.41}$$

The equation $\mathbf{A}\bar{\mathbf{e}} = \bar{\mathbf{r}}$ might subsequently be solved for

$$\bar{\mathbf{e}} = \mathbf{A}^{-1}\bar{\mathbf{r}} \tag{5.42}$$

leading to an 'exact' solution in the form

$$\mathbf{x} = \bar{\mathbf{x}} + \bar{\mathbf{e}}. \tag{5.43}$$

Before accepting this tempting solution, let's analyze the the propagation of round-off errors. Assuming that the magnitudes of elements of $\mathbf{A}$ matrix are not far from unity and that the computation is provided with 8 significant digits and that 5 digits are lost due to round-off errors during the elimination process. From it follows that the result is known with precision of three digits only. Let's denote the valid digits by `V` and the lost ones by `L`.

$$\begin{array}{lcl} \mathbf{A}, \mathbf{b} & : & \texttt{0.VVVVVVVV} \\ \bar{\mathbf{x}} = \mathbf{A}^{-1}\mathbf{b} & : & \texttt{0.VVVLLLLL} \\ \bar{\mathbf{r}} & : & \texttt{0.000VVVVVLLL} \end{array}$$

Using the mentioned eight-digit arithmetics we actually do not see the last three digits of the residual vector $\bar{\mathbf{r}}$. Solving (5.39) for $\bar{\mathbf{e}}$ we would lose five digits again and the vector $\bar{\mathbf{e}}$

$$\bar{\mathbf{e}} = \texttt{0.000LLLLLLLL}$$

would contain pure numerical garbage. So trying to estimate error $\bar{\mathbf{e}}$ by solving (5.42) fails. The 'computing reality' is not usually be as black as in the presented case where we have lost five significant digits out of eight. To get a reasonable answer using this approach requires to compute the residuum with the precision which is higher than that for the rest of computation. The idea might be implemented as indicated in the Program 14.

Program 14

```
      DOUBLE PRECISION SUM,AIJ,XJ,BI
      ...
      DO 10 I = 1,N
        SUM = 0.D0
        DO 20 J = 1,N
          AIJ = A(I,J)
          XJ = X(J)
          SUM = SUM + AIJ*XJ
```

```
20        CONTINUE
        BI = B(I)
        SUM = BI - SUM
        R(I) = SUM
10      CONTINUE
```

End of Program 14. □

Another estimate of the condition number might be obtained by applying norms to (5.39), which gives

$$\|\bar{\mathbf{e}}\| \leq \|\mathbf{A}^{-\mathbf{1}}\| \, \|\bar{\mathbf{r}}\|. \tag{5.44}$$

In most cases, a generic element of $\bar{\mathbf{r}}$ will have a magnitude equal to a few of last significant digits of $a_{ij}\,\bar{x}_j$, i.e. to b^{-t} $\max_{i=1,2,\ldots n}\left(\|a_{ij}\|\,\|x_j\|\right)$ where b is the digit basis for the internal representation of 'real' numbers on the computer being employed and t is the number of significant digits representing the mantissa part of the number representation. The residual norm could thus be expressed in the form

$$\|\bar{\mathbf{r}}\| \doteq b^{-t}\|\mathbf{A}\|\,\|\mathbf{x}\|.$$

Substituting the last equation into (5.41) we get

$$\|\bar{\mathbf{e}}\| \leq b^{-t}\|\mathbf{A}^{-1}\|\,\|\mathbf{A}\|\,\|\bar{\mathbf{x}}\|,$$

or

$$\|\bar{\mathbf{e}}\| \leq b^{-t}p\,(\mathbf{A})\bar{\mathbf{x}}.$$

So another estimate of the condition number is

$$c(\mathbf{A}) \geq \frac{\|\bar{\mathbf{e}}\|}{\|\bar{\mathbf{x}}\|}\, b^{t}. \tag{5.45}$$

Since the order of b^{-t} is equal to the order of the unit round-off error ε_{M} we can use the previous relation for the relative error estimate of $c(\mathbf{A})$ followingly

$$\frac{\|\bar{\mathbf{e}}\|}{\|\bar{\mathbf{x}}\|} \leq c(\mathbf{A})\, b^{-t} = c(\mathbf{A})\,\varepsilon_{\mathrm{M}}. \tag{5.46}$$

In technical practice we often assess the precision by number of significant digits in the result. Assuming that we have at our disposal a computer with t significant decimal digits then the order of relative errors of the matrix input data could be expressed as

$$\frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} \doteq 10^{-t}. \tag{5.47}$$

Obtaining the result with s significant digits actually means

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}+\Delta\mathbf{x}\|} \doteq 10^{-s}. \tag{5.48}$$

Substituting Eqs. (5.44) and (5.45) into do Eq. (5.33) and rearranging we get a nice and simple relation

$$s \geq t - \log_{10}(c(\mathbf{A})) \tag{5.49}$$

for the estimate of the number of significant digits or the result obtained when solving $\mathbf{Ax} = \mathbf{b}$ on a computer working with t significant decimal digits for a matrix whose condition number is $c(\mathbf{A})$.

5.7 Another partial pivoting example

How to proceed when a partial pivoting is required is shown in the Program 15, where a Basic[3] program, adapted from [6], is presented. The Fortran version of this program, together with many others, can be downloaded from

`www.pdas.com/programs/fmm.f90.`

The Program 15 solves the system of n algebraic equations by means of two procedures, namely `DECOMP` and `SOLVE`. The former assumes that the input matrix is stored in `A(N,N)` array and creates – in the same place – the upper triangular matrix resulting from the Gauss elimination process. During this process the condition number `COND`, using Eq. (5.34), is computed and the matrix singularity is being checked. The procedure employs two auxiliary vectors `P(N)` and `W(N)`. The right hand side vector `B(N)` has to be introduced only after the triangularization process has been finished, since the `DECOMP` procedure employs it for the condition number evaluation. The `SOLVE` procedure provides the right hand side reduction and the back substitution. The result is stored in the `B(N)` vector and thus its initial contents is destroyed.

Program 15

```
4 REM NO2EB last ver. 2008 12 23, ok
5 REM Program FORSYTE41
6 REM
7 OPEN "NO2EB.txt" FOR OUTPUT AS #1
10 REM test decomp a solve
17 DEFDBL A-H,O-Z  : DEFINT I-N
20 N=10
30 DIM A(N,N):DIM B(N)
40 DIM W(N): DIM P(N)
45 DIM C(N)
50 FOR I=1 TO N
51   FOR J=1 TO N
52   A(I,J)=0
54   NEXT J
55 NEXT I
56 FOR I=2 TO N-1
58   A(I,I-1)=-.5
60   A(I,I)=1
62   A(I,I+1)=-.5
64   XJ=2*N+2
65 NEXT I
66 A(1,1)=(N+2)/XJ
68 A(N,N)=A(1,1)
70 A(1,2)=-.5
72 A(N,N-1)=-.5
74 A(1,N)=1/XJ
76 A(N,1)=A(1,N)
210 REM
220 FOR I=1 TO N
222   B(I)=0
224 NEXT I
226 B(N)=1
```

[3]An old fashioned GwBasic style is used. The Czech reference could be found at http://www.dmaster.wz.cz/kurzy/basic/basic.htm. The interpreter itself could be downloaded from http://www.geocities.com/rhinorc/gwbasic.html.

```
250 REM make a copy
260 FOR I=1 TO N
261   C(I)=B(I)
262 NEXT I
270 PRINT #1, "Right hand side vector": PRINT
280 FOR I=1 TO N
290   PRINT #1, B(I);"  ";
300 NEXT I
310 PRINT #1, " ";
320 REM Call DECOMP(n,a,cond,b,p)
330 GOSUB 1000
340 PRINT #1, "cond=";COND
350 CONDP=COND+1
360 IF CONDP=COND THEN PRINT "Matrix is singular":STOP
362 REM Define the right hand side vector here - not sooner!!)
363 REM ***********************
365 FOR I=1 TO N: B(I)=C(I):NEXT I
366 REM ***********************
370 REM Call SOLVE(n,a,b,p)
380 GOSUB 3000
390 PRINT #1, "          Results                Rel. errrors [%]"
392 PRINT #1, "-----------------------------------------------------"
400 FOR I=1 TO N
410   PRINT #1, I TAB(10) B(I) TAB(30) (I-B(I))*100/I
420 NEXT I
425 CLOSE #1
430 STOP
1000 REM DECOMP(n,a,cond,p,w)
1002 REM
1004 REM n      matrix order
1006 REM a      input matrix on input
1008 REM a      triangugularized matrix on output
1010 REM cond   condition number
1012 REM p,w        pivot and auxiliary vectors
1014 REM
1016 P(N)=1
1018 IF N=1 THEN GOTO 1192
1020 NM1=N-1
1022 REM a-norm
1024 ANORM=0
1026 FOR J=1 TO N
1028   T=0
1030   FOR I=1 TO N
1032     T=T+ABS(A(I,J))
1034   NEXT I
1036   IF T>ANORM THEN ANORM=T
1038 NEXT J
1040 REM gaus. elim. - part. piv.
1042 FOR K=1 TO NM1
1044   KP1=K+1
1046   REM find a pivot
1048   M=K
1050   FOR I=KP1 TO N
1052     IF (ABS(A(I,K))>ABS(A(M,K)))THEN M=I
1054   NEXT I
1056   P(K)=M
1058   IF M<>K THEN P(N)=-P(N)
```

```
1060   T=A(M,K)
1062   A(M,K)=A(K,K)
1064   A(K,K)=T
1066   REM skip if pivot=0
1068   IF T=0 THEN GOTO 1098
1070   REM multipliers
1072   FOR I=KP1 TO N
1074     A(I,K)=-A(I,K)/T
1076   NEXT I
1078   REM exchange and eliminate
1080   FOR J=KP1 TO N
1082     T=A(M,J)
1084     A(M,J)=A(K,J)
1086     A(K,J)=T
1088     IF T=0 THEN GOTO 1096
1090     FOR I=KP1 TO N
1092       A(I,J)=A(I,J)+A(I,K)*T
1094     NEXT I
1096   NEXT J
1098 NEXT K
1100 REM estimate cond
1102 REM res. (a-trans)*y=e
1104 FOR K=1 TO N
1105   PRINT "k=";K
1106   T=0
1108   IF K=1 THEN GOTO 1118
1110   KM1=K-1
1112   FOR I=1 TO KM1
1114     T=T+A(I,K)*W(I)
1116   NEXT I
1118   EK=1
1120   IF T<0 THEN EK=-1
1122   IF A(K,K)=0 THEN GOTO 1196
1124   W(K)=-(EK+T)/A(K,K)
1126 NEXT K
1128 FOR L=1 TO NM1
1130   K=N-1
1132   T=0
1134   KP1=K+1
1136   FOR I=KP1 TO N
1138     T=T+A(I,K)*W(K)
1140   NEXT I
1142   W(K)=T
1144   M=P(K)
1146   IF M=K THEN GOTO 1154
1148   T=W(M)
1150   W(M)=W(K)
1152   W(K)=T
1154 NEXT L
1156 REM
1158 YNORM=0
1160 FOR I=1 TO N
1162   YNORM=YNORM+ABS(W(I))
1164 NEXT I
1166 REM solution A*Z=Y
1168 REM Call SOLVE(n,a,w,p)
1170 FOR I=1 TO N
```

```
1171   B(I)=W(I)
1172 NEXT I
1174 GOSUB 3000
1175 FOR I=1 TO N: W(I)=B(I): NEXT I
1176 ZNORM=0
1178 FOR I=1 TO N
1180   ZNORM=ZNORM+ABS(W(I))
1182 NEXT I
1184 REM estimate cond.
1186 COND=ANORM*ZNORM/YNORM
1188 IF COND<1 THEN COND=1
1190 RETURN
1192 REM 1 by 1
1194 IF A(1,1)<>0 THEN RETURN
1196 REM exact singularity
1198 COND=1E+32
1200 RETURN
1202 REM end of DECOMP
3000 REM SOLVE(n,a,b,p)
3002 REM
3004 REM input
3006 REM n ..  matrix order
3008 REM a ..  triang. matrix from DECOMP
3010 REM b ..  RHS vector
3012 REM p ..  pivot vector from DECOMP
3014 REM output
3016 REM b .. vector of results
3018 REM
3020 REM forward elimination
3022 IF N=1 THEN GOTO 3064
3024 NM1=N-1
3026 FOR K=1 TO NM1
3028   KP1=K+1
3030   M=P(K)
3032   T=B(M)
3034   B(M)=B(K)
3036   B(K)=T
3038   FOR I=KP1 TO N
3040     B(I)=B(I)+A(I,K)*T
3042   NEXT I
3044 NEXT K
3046 REM back substitution
3048 FOR L=1 TO NM1
3049   KM1=N-L
3050   K=KM1+1
3052   B(K)=B(K)/A(K,K)
3054   T=-B(K)
3056   FOR I=1 TO KM1
3058     B(I)=B(I)+A(I,K)*T
3060   NEXT I
3062 NEXT L
3064 B(1)=B(1)/A(1,1)
3066 RETURN
3068 REM end
3070 STOP
```

End of Program 15. □

The `DECOMP (1000 - 1202)` subroutine has the following input parameters

`N`	number of unknowns, dimension of matrix, vectors,
`A(N,N)`	array containing the matrix - generally unsymmetric.

The `DECOMP` subroutine has the following output parameters

`A(N,N)`	triangularized input matrix,
`COND`	estimate of condition number,
`P(N)`	vector containing pivots.

The `SOLVE (3000 - 3066)` procedure has the following input parameters

`N`	number of unknowns,
`A(N,N)`	triangularized matrix from `DECOMP`,
`B(N)`	right hand side vector,
`P(N)`	vector containing pivots from `DECOMP`.

The `SOLVE (3000 - 3066)` procedure has the only parameter, i.e. `B(N)` vector, containing the solution.

For testing purposes the following input matrix is used

$$\mathbf{A} = \begin{bmatrix} (n+2)/(2n+2) & -1/2 & 0 & 0 & \cdots & 0 & 1/(2n+2) \\ -1/2 & 1 & -1/2 & 0 & \cdots & 0 & 0 \\ 0 & -1/2 & 1 & -1/2 & \cdots & 0 & 0 \\ \vdots & & & & & & \\ 0 & 0 & \cdots & & -1/2 & 1 & -1/2 \\ 1/(2n+2) & 0 & \cdots & & 0 & -1/2 & (n+2)/(2n+2) \end{bmatrix}. \tag{5.50}$$

It is a nice matrix, because its inverse is known in a general algebraic form, see [25], namely

$$\mathbf{A}^{-1} = \begin{bmatrix} n & n-1 & n-2 & \cdots & 2 & 1 \\ n-1 & n & n-1 & \cdots & 3 & 2 \\ n-2 & n-1 & n & \cdots & 4 & 3 \\ \vdots & & & & & \\ 2 & 3 & 4 & \cdots & n & n-1 \\ 1 & 2 & 3 & \cdots & n-1 & n \end{bmatrix}. \tag{5.51}$$

Assuming the right hand side vector in the form

$$\mathbf{b}^{\mathrm{T}} = \{\ 0 \quad 0 \quad 0 \quad \cdots \quad 0 \quad 1\ \},$$

the solution of (5.25) is evidently just the last column of $\mathbf{A}$, i.e.

$$\mathbf{x}^{\mathrm{T}} = \{\ 1 \quad 2 \quad 3 \quad \cdots \quad n-1 \quad n\ \}.$$

The results of the Program 15 for $n = 10$ are presented in the following table

```
cond = 126.6239077610022

Counter        Results           Rel. errrors [%]
1        0.999999999999999   1.040834085586084D-13
2        1.999999999999999   5.828670879282072D-14
3        2.999999999999999   4.440892098500626D-14
4        3.999999999999999   3.608224830031759D-14
5        4.999999999999999   3.108624468950438D-14
6        5.999999999999999   2.405483220021173D-14
```

```
 7        6.999999999999999   1.903239470785983D-14
 8        7.999999999999999   1.249000902703301D-14
 9        8.999999999999999   7.401486830834377D-15
 10       10                  4.440892098500626D-15
```

In Matlab, after the input data for the matrix and the right hand side vector have been loaded, it suffices to write

Program 16

```
 ......
 cond(A), x = A\b;
 ......
```

End of Program 16. □

Executing the Program 16, written in Matlab, we get

```
cond = 1.323604518682353e+002

Counter       Results                 Rel. errrors [%]
1        9.999999999999889e-001    1.110223024625157e-012
2        1.999999999999987e+000    6.328271240363392e-013
3        2.999999999999986e+000    4.588921835117313e-013
4        3.999999999999985e+000    3.774758283725532e-013
5        4.999999999999983e+000    3.375077994860476e-013
6        5.999999999999982e+000    2.960594732333751e-013
7        6.999999999999985e+000    2.157004733557447e-013
8        7.999999999999987e+000    1.665334536937735e-013
9        8.999999999999989e+000    1.184237892933500e-013
10       9.999999999999993e+000    7.105427357601002e-014
```

Evaluating condition numbers by the Program 16 for different dimensions of $\mathbf{A}$ matrix, defined by (5.50), and computing the estimated number of significant digits in the result of $\mathbf{Ax} = \mathbf{b}$, using the formula (5.46) with $t = 16$, we get

```
matrix dimension    condition number    number of significant
                                           digits (rounded)
      10            132.360451868235              14
     100            13507.5820929118              12
    1000            1351031.12977951              10
    5000            33775844.3395144               8
```

The Fig. 5.2 shows how the condition number – in this case – depends on the order of the matrix. It also shows that whatever you plot in logarithmic coordinates closely resembles a line.

Comparing the menial effort required to write a program for solving the system of algebraic equations in Basic (Program 15) and in Matlab (Program 16) one might come to conclusion – evidently a naive one, as the reader is hoped to discover soon – that to work with element level languages (Basic, Pascal, Fortran77) is a futile activity that should be avoided at all costs.

There are at least two important reasons why it is not so.

First, to effectively use languages with high-level programming primitives, allowing the efficient processing of matrix algebra tasks, requires detailed knowledge of the matrix algebra rules and of their particular implementations. One has to know what to choose and from where ....

Second, and even more important, is the size and the 'standardness' of the problem we are trying to solve. This e-book is primarily written for human non-standard problem solvers. Standard and 'regular' problems could easily be tackled using a brute force approach using standard problem oriented packages and freely available programming libraries covering a wide range of numerical problems. Non-standard problems and/or huge memory and time requirements necessitate – among others – in non-standard data storage schemes which in turn require that standard procedures providing matrix algebra operations have to be completely rewritten from the scratch which requires to be familiar with matrix operations on an element level. So dirtying our hands with low, element level programming is a necessity until the progress in computer technology provides for a new measure of what is a huge size problem and what is a non-standard problem.

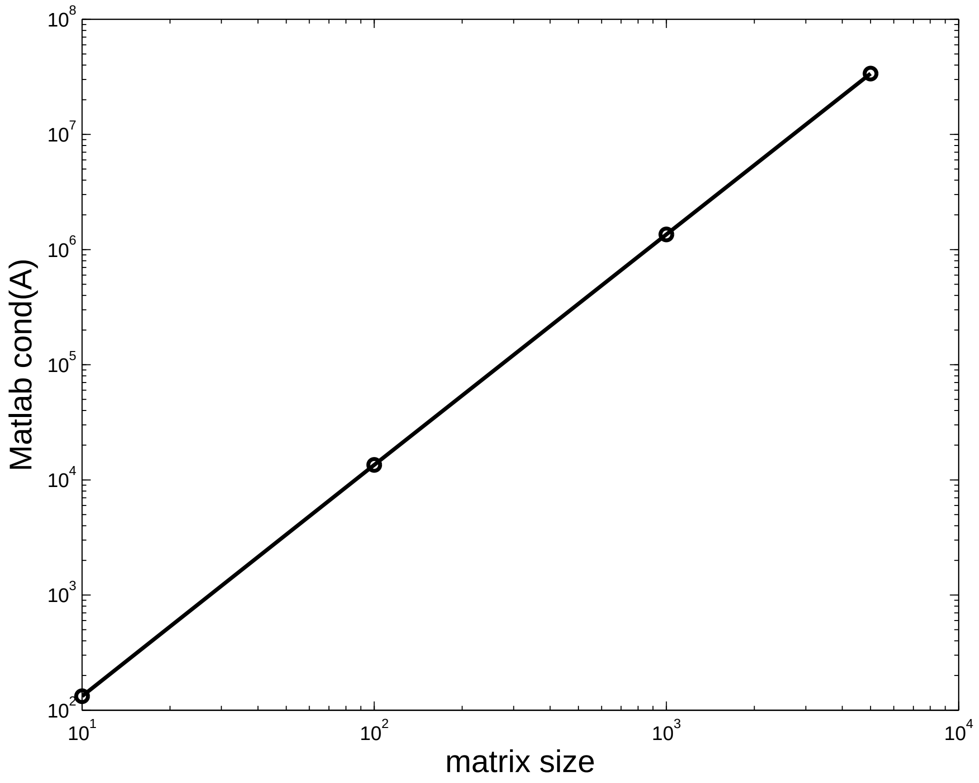


Figure 5.3: Condition number for the matrix defined by (5.47) as a function of the number of equations

5.8 Gauss elimination with a symmetric matrix

Symmetric matrices play an important role in computational mechanics since their properties can advantageously be exploited for saving both the memory storage and the computing time requirements.

Storing the elements, residing in the upper triangular part of the **A** matrix, in a vector **w** columnwise – this storage approach, initially introduced by IBM in their programming libraries, is known under the name *symmetric matrix storage mode*, is explained in the Chapter 4 – requires $n(n-1)/2$ memory locations instead of n for a regular 'full' or 'general' storage approach characterized by the statement of `DIMENSION A(n,n)` type. Of course n is the order of the matrix.

The number of floating point operations – computing time – can be saved by realizing that the property of symmetry is being maintained during the elimination process[4].

From it follows that eliminating the `k`-th column of a symmetric matrix it suffices to work with a part of the `i`-th row which is 'behind' the diagonal. So the `j`-loop limits are set to from `i` to `n`, instead of `k+1` to `n` as it is in a 'full' storage case.

One has also to realize that employing the *symmetric matrix storage mode* we not have at our disposal the the under-diagonal element a_{ik} needed for the multiplier $t = a_{ik}/a_{kk}$. This element has to be replaced by its symmetric counterpart, i.e. by a_{ki}. The considerations, explaining the programming differences for a general, symmetric and *symmetric banded*[5] storage modes are summarized in Fig. 5.4.

5.9 Gauss elimination with a symmetric, banded matrix

The rectangular storage mode suitable for symmetric banded matrices is described in 4.

Additional savings could be achieved by employing the matrix bandedness. To eliminate the k-th column elements it suffices to address the elements of the the i-th row from the diagonal to the end of the band, since the elimination process does not create new elements outside the band.

Thus the `i`-loop ranges from `j` to do `JMAX` where kde `JMAX` is the smaller value from `k + NBAND - 1`, and `n`, where `NBAND` is the half-band width including the diagonal. Furthermore, there is no need to eliminate the under-diagonal elements so the `i` counter ranges from `IMAX = JMAX` only. See Fig. 5.3 again.

If, in addition to that, the matrix is positive definite, the pivoting process could be avoided entirely, since all the pivots are to be positive. Of course, they still have to checked. See [28]. An implementation example of the Gauss elimination with rectangular-storage-mode ideas, written in Pascal, is in the Program 17. Procedure is named `gre` and for the solution of the system of algebraic equations has to be called twice. First, with `key = 1` the matrix triangularization is obtained, then, with `key = 2` the reduction of he right hand side and the back substitution is carried out. This way allows for efficient solution of cases with multiple right hand sides.

As a side product of triangularization we obtain the determinant value which is known to be the product of diagonal elements of the upper triangular matrix, see [30]. If any pivot becomes 'small' or negative, the execution of the procedure is stopped with a an error flag `ier = -1`.

[4]So far no pivoting is assumed

[5]See the following paragraph – 5.9.

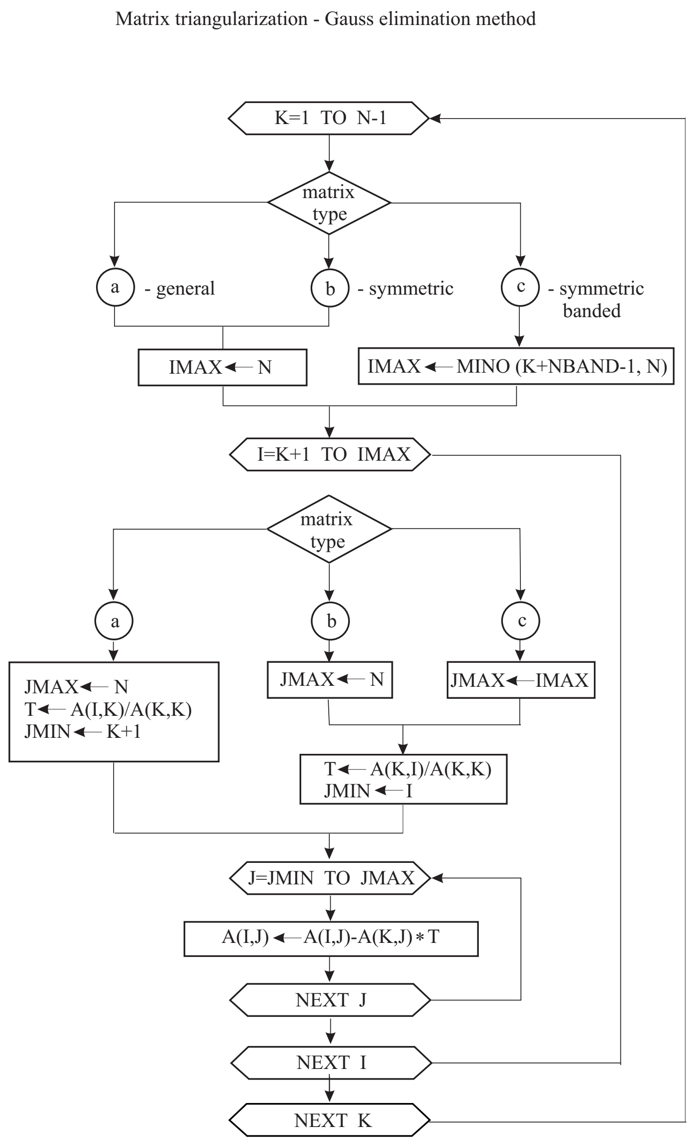


Figure 5.4: Employing symmetry and bandednes of a matrix during the Gauss elimination process

Program 17

```
program tegrem1;

{test of gre and mavba procedures}

const
      ndim=6;mdim=3;
type
      rmat=array[1..ndim,1..mdim] of real;
      rvec=array[1..ndim] of real;
var
      a,acopy                : rmat;
      b,c,q,bcopy,r          : rvec;
      i,j,jm,n,ier,key,nband : integer;
      eps, det               : real;
```

```
procedure gre(var a:rmat; var b,q:rvec;
              var n,nband,ier,key:integer;
              var det,eps:real);

 { Solution of [A](q)=(b)}
 { for a symmetric positive definite matrix }
 { efficiently stored in a rectangular array}

label 9,99;
 var i,j,k,imax,jmax               : integer;
    akk,aki,aij,akj,aii,ann,t,sum  : real;

begin
 ier:=0; det:=1;
 case key of
1: begin {triangular decomposition ... key=1}
   for k:=1 to n-1 do
    begin
     akk:=a[k,1]; imax:=k+nband-1;
     if imax > n then imax:=n;
     jmax:=imax;
     if akk < eps then begin ier:=-1; goto 99 end;
     det:=det*akk;
     for i:=k+1 to imax do
      begin
       aki:=a[k,i-k+1];
       if abs(aki) < eps then goto 9;
       t:=aki/akk;
       for j:=i to jmax do
        begin
         aij:=a[i,j-i+1];
         akj:=a[k,j-k+1];
         a[i,j-i+1]:=aij-t*akj
        end;
9:    end;
    end;
    det:=det*a[n,1];
   end; {the end of triangularization}

2:begin {reduction of the right hand side ... pro key=2}
   for k:=1 to n-1 do
    begin
     akk:=a[k,1]; imax:=k+nband-1;
     if imax > n then imax:=n;
     for i:=k+1 to imax do
      begin
       aki:=a[k,i-k+1]; t:=aki/akk; b[i]:=b[i]-t*b[k];
      end;
    end;
{  back substitution  }
   for i:=n downto 1 do
    begin
     aii:=a[i,1]; sum:=0; jmax:=i+nband-1;
     if jmax > n then jmax:=n;
     for j:=i+1 to jmax do
      begin
       aij:=a[i,j-i+1]; sum:=sum+aij*q[j];
      end;
     q[i]:=(b[i]-sum)/aii;
    end;
  end;  {end of the reduction of the RHS and of the backsubstitution}
 end;   {of case}
99:    {error return} end;   {of GRE}

procedure mavba(var a:rmat; var b,c:rvec; var n,nband:integer);

 { matrix - vector multiplication (c) <- [A](b)}
 { matrix is stored in a rectangular array n*nband}

label 4; var i,j,key   : integer;
             sum1,sum2 : real;
begin
  for i:=1 to n do
    begin
      sum1:=0; sum2:=0;
      if i < nband then key:=1
                   else if i < n-nband+2 then key:=2
                                         else key:=3;
      case key of
1:      begin
          for j:=i to i+nband-1 do sum1:=sum1+b[j]*a[i,j-i+1];
          if i=1 then goto 4;
          for j:=1 to i-1 do sum2:=sum2+b[j]*a[j,i-j+1];
        end;
2:      begin
          for j:=i to i+nband-1 do sum1:=sum1+b[j]*a[i,j-i+1];
          for j:=i-nband+1 to i-1 do sum2:=sum2+b[j]*a[j,i-j+1];
        end;
3:      begin
          for j:=i to n do sum1:=sum1+b[j]*a[i,j-i+1];
          for j:=i-nband+1 to i-1 do sum2:=sum2+b[j]*a[j,i-j+1];
        end;
      end; {end of case}
4:    c[i]:=sum1+sum2;
  end;     {of i-loop}
end;       {of MAVBA}

{main}
 begin
 n:=6; nband:=3; eps:=1e-6;
 {test matrix}
 a[1,1]:=1;  a[1,2]:=2;  a[1,3]:=-1;
 a[2,1]:=8;  a[2,2]:=4;  a[2,3]:=-2;
 a[3,1]:=19; a[3,2]:=9;  a[3,3]:=-3;
 a[4,1]:=26; a[4,2]:=5;  a[4,3]:=-3;
 a[5,1]:=14; a[5,2]:=1;  a[5,3]:=0;
 a[6,1]:=6;  a[6,2]:=0;  a[6,3]:=0;
 {make a copy}
 acopy:=a;
 {RHS vector}
 b[1]:=-2; b[2]:=-10; b[3]:=35; b[4]:=47; b[5]:=-34; b[6]:=-3;
 {mae a copy}
 bcopy:=b;
 write('matrix'); writeln;
 for i:=1 to n do
  begin
   writeln;
   for j:=1 to nband do write(a[i,j]:7:2);
  end;
 writeln;
 writeln('RHS vector');
 for i:=1 to n do write(b[i]:7:2);
 writeln;
 key:=1;
 gre(a,b,q,n,nband,ier,key,det,eps); {triangularization}
 writeln('ier,det:',ier,det);
 if ier=0 then
  begin
   key:=2;
   gre(a,b,q,n,nband,ier,key,det,eps); {RHS reduction and backsubstitution}
   writeln('result');
   for i:=1 to n do write(q[i]:7:2);
   writeln;
  end
          else writeln('solution failed');
 {check (c) <- [A](q)}
 mavba(acopy,q,c,n,nband);
 writeln('check (c) <- [A](q)');
 for i:=1 to n do write(c[i]:7:2);
 {reziduum}
 for i:=1 to n do r[i]:=c[i]-bcopy[i];
 writeln; writeln('rezidual vector');
 for i:=1 to n do writeln(r[i]);
end.
```

End of Program 17. □

The Program contains a trivial test example. Also the `mavba` procedure for the matrix vector multiplication is presented, explaining the need for dirtying our fingertips with low level element programming in cases when a special storage schemes are employed.

The Fortran77 equivalent of the procedure for solving the system of algebraic equations, whose matrix is stored in the rectangular storage mode, is in the Program 18.

Program 18

```
      SUBROUTINE GRE(A,B,Q,N,NDIM,NBAND,DET,EPS,IER,KEY)
      DIMENSION A(NDIM,NBAND),B(N),Q(N)
      DOUBLE PRECISION SUM

C      ***   Solution of [R]{Q} = {B}
C      ***   by Gauss elimination for a symmetric,
C      ***   banded, positive definite matric [R]
C
C      It is assumed that the upper triangular part
C      of the band [R] is stored in a rectangular array A(NBAND,N)

C      Parameters
C       A   on input         contains the upper triangular band of [R] matrix
C           on output        triangularized part of [R]
C
C       B                    RHS vector
C       Q                    result vector
```

```
 C      N                    number of unknowns
 C      NDIM                 row dimension of A array declared in main
 C      NBAND                half bandwidth (including diagonal)
 C      DET                  matrix determinant
 C      EPS                  smallest acceptable pivit value
 C      IER                  error parameter
 C                              = 0 ... O.K.
 C                              = -1    matrix is singular or not positive definite
 C                                      computation is interupted
 C
 C     KEY                   key of the solution
 C                              = 1 ... triangularization of the input matrix
 C                              = 2 ... RHS reduction and back substitution
      NM1=N-1
      IER=0
      II=KEY
      GO TO (1000,2000), II
C     triangularization
1000  DET=1.
      DO 9 K=1,NM1
        AKK=A(K,1)
        IMAX=K+NBAND-1
        IF(IMAX .GT. N) IMAX=N
        JMAX=IMAX
        IF(AKK .GT. EPS) GO TO 5
        IER=-1
        RETURN
5       DET=DET*AKK
        KP1=K+1
        DO 9 I=KP1,IMAX
          AKI=A(K,I-K+1)
            IF(ABS(AKI) .LT. EPS) GO TO 9
        T=AKI/AKK
        DO 8 J=I,JMAX
          AIJ=A(I,J-I+1)
            AKJ=A(K,J-K+1)
8           A(I,J-I+1)=AIJ-AKJ*T
9     CONTINUE
      ANN=A(N,1)
      DET=DET*ANN
C     success
      RETURN
C     reduction of B
2000  DO 90 K=1,NM1
        KP1=K+1
        AKK=A(K,1)
        IMAX=K+NBAND-1
        IF(IMAX .GT. N) IMAX=N
        DO 90 I=KP1,IMAX
          AKI=A(K,I-K+1)
          T=AKI/AKK
          B(I)=B(I)-T*B(K)
          if(abs(b(i)) .lt. 1.e-30) b(i)=0.0
90    continue
      back substitution
      Q(N)=B(N)/A(N,1)
      AL1=A(N-1,2)
      AL2=A(N-1,1)
      Q(N-1)=(B(N-1)-AL1*Q(N))/AL2
      DO 10 IL=3,N
        I=N-IL+1
        AII=A(I,1)
        SUM=0.D0
        J1=I+1
        JMAX=MIN0(I+NBAND-1,N)
        DO 20 J=J1,JMAX
          AIJ=A(I,J-I+1)
20        SUM=SUM+AIJ*Q(J)
10    Q(I)=(B(I)-SUM)/AII
      RETURN
      END
```

End of Program 18. □

5.10 Gauss elimination with a matrix stored by means of skyline scheme

This storage scheme is also known as *variable band* or *variable profile* mode. It is described in the Chapter 4.

It is known – see [30] and [28] – that there appear no fill-in elements outside the skyline of the matrix during the elimination process. That's why it is suitable to store only under-skyline elements in the memory.

The Fortran implementation of the Gauss elimination process with a matrix stored in the skyline storage mode is listed in [2].

5.11 Gauss elimination with a tri-diagonal matrix

The elements of a tri-diagonal matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & & \cdots & & 0 \\ a_{21} & a_{22} & a_{23} & 0 & & \cdots & & 0 \\ 0 & a_{32} & a_{33} & a_{34} & & & & \\ & & & & \ddots & & & \\ & & & & & a_{n-2,n-1} & a_{n-1,n-1} & a_{n,n-1} \\ 0 & & \cdots & & & & a_{n,n-1} & a_{n,n} \end{bmatrix}$$

could be associated with three vectors $\mathbf{e}$, $\mathbf{f}$, $\mathbf{g}$ as follows

$$\mathbf{A} = \begin{bmatrix} e_1 & f_1 & 0 & 0 & & & & \\ d_2 & e_2 & f_2 & 0 & & & & \\ 0 & d_3 & e_3 & f_3 & & & & \\ 0 & 0 & d_4 & e_4 & & & & \\ & & & & \ddots & & & \\ & & & & & d_{n-1} & e_{n-1} & f_{n-1} \\ & & & & & & d_n & e_n \end{bmatrix}$$

Then, we can could proceed as follows

$$\mathbf{A} = \mathbf{T}\,\mathbf{U},$$

where

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & & & \\ t_2 & 1 & 0 & & 0 & \\ 0 & t_3 & 1 & & & \\ \vdots & & & \ddots & & \\ 0 & \cdots & & & t_n & 1 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} u_1 & f_1 & 0 & & \cdots & 0 \\ & u_2 & f_2 & 0 & & \\ & & u_3 & f_3 & & \vdots \\ & 0 & & \ddots & & \\ & & & & u_{n-1} & f_{n-1} \\ & & & & & u_n \end{bmatrix}.$$

Assuming the matrix regularity we could triangularize the matrix as indicated in the Template 26.

Template 26, Triangularization A → U

```
$u_1 = e_1$
FOR $i = 2$ TO $n$
$t_i = d_i/u_{i-1}$
$e_i = e_i - t_i\, f_{i-1}$
NEXT $i$ ,
```

Template 27, Right hand side reduction b → c

```
$c_1 = b_1$
FOR $i = 2$ TO $n$
$c_i = b_i - t_i\, c_{i-1}$
NEXT $i$
```

Template 28, Back substitution

```
$x_n = c_n/, u_n$
FOR $i = n - 1$ TO $1$
$x_i = c_i - f_i\, x_{i+1}/u_i$
NEXT $i$
```

The Basic implementation of the Gauss elimination process with a tridiagonal matrix is indicated in the Program 19.

Program 19

```
4 REM NUMAT21, last rev. 010392, ok
5 REM Program TRIDIAG
6 REM Gauss elimination for a system of algebraic equations
7 REM with a tridiagonal matrix
10 N=10
20 DIM A(N,N)
30 DIM E(N): DIM F(N): DIM D(N)
40 DIM B(N): REM right hand side
50 DIM X(N): REM solution
60 DIM U(N)
70 DIM R(N): DIM T(N)
```

```
100 REM matrix
105 REM diagonal
110 FOR I=1 TO N
120   A(I,I)=I
130   E(I)=I
140 NEXT I
150 REM above diagonal elements
160 FOR I=1 TO N-1
170   A(I,I+1)=I+1
180   F(I)=A(I,I+1)
190 NEXT I
200 REM under diagonal elements
210 FOR I=2 TO N
220   A(I,I-1)=N-I
225   D(I)=A(I,I-1)
230 NEXT I
240 REM print
250 FOR I=1 TO N
260   FOR J=1 TO N
270     PRINT A(I,J);" ";
280   NEXT J
290   PRINT
300 NEXT I
305 PRINT "Right Hand Side"
310 REM RHS
320 FOR I=1 TO N
330   B(I)=2*I-N
340   PRINT B(I);" ";
350 NEXT I
360 PRINT
370 GOSUB 1000
380 REM print result
390 PRINT "Solution"
400 FOR I=1 TO N
410   PRINT I,X(I)
420 NEXT I
430 REM Residual vector
435 PRINT "Residual vector"
440 FOR I=1 TO N
450   R(I)=0
460   FOR J=1 TO N
470     R(I)=R(I)+A(I,J)*X(J)
480   NEXT J
490   R(I)=R(I)-B(I)
500   PRINT I,R(I)
510 NEXT I
520 STOP
1000 REM procedure TRIDIAG
1010 REM Solution of [A]{x}={b} with a tridiagonal matrix
1020 REM matrix diagonal is stored in e(n)
1030 REM under diagonal elements are in d(n)
1040 REM above diagonal elements are in f(n)
1050 REM u(n),t(n) auxiliary fields
1055 REM b(n) RHS
1060 REM x(n) solution
1070 REM
1080 REM matrix triangularization
```

```
1090 U(1)=E(1)
1100 FOR I=2 TO N
1110   T(I)=D(I)/U(I-1)
1120   U(I)=E(I)-T(I)*F(I-1)
1130 NEXT I
1140 REM reduction of RHS
1150 X(1)=B(1)
1160 FOR I=2 TO N
1170   X(I)=B(I)-T(I)*X(I-1)
1180 NEXT I
1190 REM back substitution
1200 X(N)=X(N)/U(N)
1210 FOR I=N-1 TO 1 STEP -1
1220   X(I)=(X(I)-F(I)*X(I+1))/U(I)
1230 NEXT I
1240 RETURN
1250 REM end of TRIDIAG
```

End of Program 19. □

5.12 Using Gauss elimination for the inverse matrix computation

One way how to solve $\mathbf{Ax} = \mathbf{b}$ consists in computation of $\mathbf{A}^{-1}$ complemented by its multiplication by the right hand side vector $\mathbf{b}$.

Such a process, however, is highly inefficient, since the inverse matrix computation generally requires n^3 operations, where n the matrix order. The consequent matrix-vector multiplication adds additional n^2 operations. The solution of the system of algebraic equations with one right hand side requires only $n^3/3 + n^2 - n/3$ operations as shown in [30].

There is another, even more significant reason against matrix inversion, namely the loss of symmetry and banddedness of the matrix being inverted. This prevents usage of any efficient matrix storage modes.

A few decades ago solving a system of equations by the matrix inversion was considered to be a sort of computational crime. Today, when a 1000×1000 matrix can be inverted in a fraction of second executing the simple Matlab statement `inv(A)`, one can proceed unpunished for doing this. But, as before, it is the problem size and its 'standardness' which influences the decision of an acceptable brute force solvability. The limits are higher than before, as well as the consequences and fines we have pay for our eventual failures.

So if we insist on the computing the matrix inversion we still could do in a memory-efficient way by solving the system of equations n times

$$\mathbf{A}\mathbf{x}^{(i)} = \mathbf{e}^{(i)}, \qquad i = 1, 2, \cdots n,$$

with right hand side vectors $\mathbf{e}^{(i)}$ containing zeros with the exception of a single '1' at its i-th location. Each solution gives a single vector $\mathbf{x}^{(i)}$ which is just the i-th column of the inverse matrix $\mathbf{A}^{-1}$, i.e.

$$\mathbf{A}^{-1} = [\mathbf{x}^{(1)} \quad \mathbf{x}^{(2)} \quad \cdots \quad \mathbf{x}^{(n)}].$$

This way we can carry out the triangularization of the matrix only once employing thus the suitable efficient matrix storage schemes. Obtaining the inverse this way requires the $4n^3/3$ operations, which is still more than using classical Gauss-Jordan reduction process. It might be acceptable if the memory limitations are more severe than those of time efficiency.

5.13 Cholesky decomposition

There are other methods based on the matrix decomposition into two or more matrices. One of them, based on the decomposition of the input matrix into the product of two matrices, bears the name after its 'invertor' André-Luis Cholesky. See [27]. The Cholesky decomposition is a product of the lower tridiagonal matrix **L** and the upper tridiagonal one **U** with 1's on its diagonal, i.e.

$$\mathbf{A} = \mathbf{L}\,\mathbf{U}, \tag{5.52}$$

where

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & & \\ \vdots & & \ddots & 0 \\ l_{n1} & & \cdots & l_{nn} \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 1 & u_{12} & & \cdots & u_{1n} \\ 0 & 1 & & \cdots & u_{2n} \\ & & \ddots & & \vdots \\ 0 & \cdots & & 0 & 1 \end{bmatrix}. \tag{5.53}$$

Writing the Eq. (5.52) element wise, considering the explicit forms of the **L** and **U** matrices we get

$$a_{ij} = \sum_{k=1}^{n} l_{ik}u_{kj} = \sum_{k=1}^{j-1} l_{ik}u_{kj} + l_{ij}. \tag{5.54}$$

From Eq. (5.54) all the elements of **U** and **L** could be expressed. For the first row elements we have

$$\begin{array}{l} l_{11} = a_{11} \\ u_{11} = 1; \quad u_{1j} = a_{1j}/l_{11}, \quad j = 2, 3, \cdots, n \end{array} \tag{5.55}$$

and for the remaining ones, i.e. for $i = 2, 3, \cdots, n$

$$\begin{array}{ll} l_{i1} = a_{i1}, & \\ l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}, & j = 2, 3, \cdots, i \\ u_{ij} = (a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj})/l_{ii} & j = i+1, i+2, \cdots, n. \end{array} \tag{5.56}$$

You should notice that each element of **A** matrix is used only once, so the resulting elements of **L** and of **U** matrices could be stored in the matrix being decomposed. Evidently the diagonal elements need not be stored at all – they are already in place.

The initial problem

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{5.57}$$

is thus expressed by the equivalent form

$$\mathbf{L}\,\mathbf{U}\,\mathbf{x} = \mathbf{b}, \tag{5.58}$$

that can alternatively be written as

$$\mathbf{L}\,\mathbf{y} = \mathbf{b} \qquad \mathbf{U}\,\mathbf{x} = \mathbf{y}. \tag{5.59}$$

Also the following relations are valid

$$\mathbf{U} = \mathbf{L}^{-1}\,\mathbf{A}; \qquad \mathbf{y} = \mathbf{L}^{-1}\mathbf{b}. \tag{5.60}$$

So triangularizing the input matrix we, at the same time, are reducing the right hand side vector. The vector of unknowns is then obtained by the back substitution process from the second equation of (5.59).

The use of the Cholesky decomposition is advantageous mainly for symmetric matrices. In this case, instead of (5.59), the matrix is decomposed of two matrices that are mutually transposed

$$\mathbf{A} = \mathbf{R}^{\mathrm{T}}\,\mathbf{R}. \tag{5.61}$$

It has been proved, see [5], that this decomposition is unique. Writing the product on the element level we get

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ & a_{22} & & a_{2n} \\ & & \ddots & \vdots \\ & & & a_{nn} \end{bmatrix} = \begin{bmatrix} r_{11} & 0 & \cdots & 0 \\ r_{12} & r_{22} & & \\ & & \ddots & \vdots \\ r_{1n} & & \cdots & r_{nn} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & & \\ & & \ddots & \vdots \\ 0 & \cdots & 0 & r_{nn} \end{bmatrix}. \tag{5.62}$$

We express the individual elements from Eq. (5.62). The first row of $\mathbf{R}$ is

$$\begin{aligned} r_{11} &= \sqrt{a_{11}} \\ r_{1j} &= a_{1j}/r_{11}, \qquad j = 2, 3, \cdots, n. \end{aligned} \tag{5.63}$$

For the remaining rows, i.e. for $i = 2, 3, \cdots, n$, we have

$$\begin{aligned} r_{ii} &= \sqrt{a_{ii} - \sum_{k=1}^{i-1} r_{ki}^2}, \\ r_{ij} &= a_{ij} - \sum_{k=1}^{i-1} r_{ki} r_{kj}, \qquad j = i+1,\, i+2, \cdots, n. \end{aligned} \tag{5.64}$$

Similarly to (5.59) we write

$$\mathbf{R}^{\mathrm{T}}\mathbf{y} = \mathbf{b}, \quad \mathbf{R}\mathbf{x} = \mathbf{y}. \tag{5.65}$$

Writing out explicitly the element products in previous equations we get the following formulas

$$\begin{aligned} y_i &= (b_i - \sum_{k=1}^{i-1} r_{ki}\, y_k)/r_{ii}, \qquad i = 1, 2, \cdots, n, \\ x_i &= (y_i - \sum_{k=1+i}^{n} r_{ik}\, x_k)/r_{ii}, \quad i = n, n-1, \cdots, 1. \end{aligned} \tag{5.66}$$

The sums are executed only if their upper limits are not smaller then the lower ones. Programs in Basic implementing the above ideas, commented in Czech, are available in [17] and [16].

Another approach for the Cholesky decomposition is based on the decomposition of the quadratic form, belonging to the symmetric positive $\mathbf{A}$ matrix, into the sum of squares of linear terms. Let's rewrite the input matrix in the form

$$\mathbf{A} = \begin{bmatrix} a_{11} & \mathbf{b}^{\mathrm{T}} \\ \mathbf{b} & \mathbf{B} \end{bmatrix}. \tag{5.67}$$

The **B** matrix is symmetric and positive definite as well. Now the **A** can be decomposed as follows

$$^{(0)}\mathbf{A} = \mathbf{A} = \begin{bmatrix} \sqrt{a_{11}} & \mathbf{0}^{\mathrm{T}} \\ \mathbf{b}/\sqrt{a_{11}} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0}^{\mathrm{T}} \\ 0 & ^{(1)}\mathbf{A} \end{bmatrix} \begin{bmatrix} \sqrt{a_{11}} & \mathbf{b}^{\mathrm{T}}/\sqrt{a_{11}} \\ 0 & \mathbf{I} \end{bmatrix}. \tag{5.68}$$

Comparing (5.67) with (5.68) we get

$$^{(1)}\mathbf{A} = \mathbf{B} - \frac{\mathbf{b}\,\mathbf{b}^{\mathrm{T}}}{a_{11}}. \tag{5.69}$$

Repeating this process we obtain the matrix of the first order and meanwhile the decomposition (5.61) will yield the product of the lower triangular and upper triangular matrices.

Gauss and Cholesky factorizations require roughly the same number of operations. See [30].

The incomplete Cholesky factorizations became popular as a precondition matrix in iterative methods for the solution of algebraic equations. See [1]. To carry out the incomplete Cholesky factorizations one should use the standard Cholesky algorithm modified in such a way that any resulting entry is set to zero if the corresponding entry in the original matrix is also zero. This forcefully prevents the fill-in, so the incompletely factorized matrix has the same sparsity as the original one. See [7].

5.14 Iterative methods

The systems of algebraic equations could alternatively be solved by iterative methods. In general the number of iterations needed for reaching the result cannot be predicted in contradistinction to 'direct' methods (Gauss, Cholesky, Gauss-Jordan, etc) where the result is obtained by the finite number of computing operations, known in advance.

In this paragraph a few iterative methods as *simple iteration*, *Jacobi*, *Gauss-Seidel* will be briefly mentioned.

5.14.1 Simple (successive substitution) iteration method

The system of algebraic equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{5.70}$$

with a regular, square matrix **A** of the n-th order and with the right hand side vector **b** is to be solved for unknown vector **x**.

The iterative process for solving (5.70) might start with

$$\mathbf{x} = \mathbf{U}\mathbf{x} + \mathbf{v}, \tag{5.71}$$

where the **U** matrix is square and of the order n, and the **v** vector are known.

Choosing $^{(0)}\mathbf{x}$ as a suitable initial approximation we construct the sequence

$$^{(k+1)}\mathbf{x} = \mathbf{U}\,^{(k)}\mathbf{x} + \mathbf{v}, \tag{5.72}$$

where k is the iteration counter. The procedure defined by (5.72) will be called as the basic stationary iterative scheme of the first order (see [8]). It is called stationary, since the **U** matrix does not change during the the process. The first order implies that the new approximation ${}^{(k+1)}\mathbf{x}$ depends on the preceding one ${}^{(k)}\mathbf{x}$ only.

The method (5.72) converges to the solution of (5.70) if the spectral radius of **U** is $\rho(\mathbf{U}) < 1$. The convergence does not depend on the choice of the initial approximation of ${}^{(0)}\mathbf{x}$.. The proof can be found in [5]. The spectral radius is defined as in absolute value the greatest eigenvalue of the matrix, i.e.

$$\rho(\mathbf{U}) = \max_{i=1\cdots n} |\lambda_i|,$$

where λ_i are the eigenvalues of **U**. For the exact solution $\mathbf{x}^*$, using (5.71), we can write

$$\mathbf{x}^* = \mathbf{U}\,\mathbf{x}^* + \mathbf{v}. \tag{5.73}$$

Subtracting (5.73) from (5.72), we get

$${}^{(k+1)}\mathbf{e} = \mathbf{U}\,{}^{(k)}\mathbf{e}, \quad \text{where} \quad {}^{(k)}\mathbf{e} = {}^{(k)}\mathbf{x} - \mathbf{x}^*,$$

which is the k-th step error. Back substituting we get

$${}^{(k+1)}\mathbf{e} = \mathbf{U}\,{}^{(0)}\mathbf{e}, \quad {}^{(0)}\mathbf{e} = {}^{(0)}\mathbf{x} - \mathbf{x}^*. \tag{5.74}$$

The sufficient condition for the convergence of the simple iteration method is following. If for the **U** matrix of (5.72) its norm $\|\mathbf{U}\| < 1$, then the iterative process is independent of the choice of the initial approximation and the absolute error estimate is

$$\|\mathbf{x}^* - {}^{(k)}\mathbf{x}\| \leq \frac{\|\mathbf{U}\|}{1 - \|\mathbf{U}\|}\, \|{}^{(k)}\mathbf{x} - {}^{(k-1)}\mathbf{x}\|. \tag{5.75}$$

The proof is in [33].

Implementation of the simple iteration method

To get (5.71) out of (5.70) we proceed as follows. To each side of $\mathbf{Ax} = \mathbf{b}$ we add the **x** vector

$$\mathbf{x} + \mathbf{Ax} = \mathbf{x} + \mathbf{b}$$

and then

$$\mathbf{x} = (\mathbf{I} - \mathbf{A})\mathbf{x} + \mathbf{b}, \qquad \text{where}\,\mathbf{I}\,\text{is the identity matrix.}$$

The iterative scheme is thus conceived as

$${}^{(k+1)}\mathbf{x} = (\mathbf{I} - \mathbf{A})\,{}^{(k)}\mathbf{x} + \mathbf{b}, \tag{5.76}$$

with $\mathbf{U} = \mathbf{I} - \mathbf{A}$, and $\mathbf{v} = \mathbf{b}$. It goes under the name the *simple iterative method* or the *Richardson's methods*. See [8]. The above method could be modified as follows.

The initial equation (5.70) is multiplied by a scalar c.

$$c\,\mathbf{Ax} = c\,\mathbf{b}.$$

Now, we will proceed as before, i.e. the vector **x** will be added and the iteration scheme could be conceived as

$${}^{(k+1)}\mathbf{x} = (\mathbf{I} - c\mathbf{A})\,{}^{(k)}\mathbf{x} + c\,\mathbf{b}. \tag{5.77}$$

It can be shown (see [8], [29], [12]), that using $c = \frac{2}{M(\mathbf{A})+m(\mathbf{A})}$ the method (5.77) converges for any choice of an initial approximation. It is assumed the **A** matrix is positive definite and $M(\mathbf{A})$ and $m(\mathbf{A})$ are their maximum and minimum eigenvalues respectively.

Implementation considerations are in the Template 29.

Template 29, Simple iteration

Preliminaries

```
    Chose a nonzero scalar $c$
    Compute $\mathbf{U}$ matrix:
        $\mathbf{U} \leftarrow \mathbf{I} - c\,\mathbf{A}$
    Compute $\mathbf{v}$ vector
        $\mathbf{v} \leftarrow c\,\mathbf{b}$
```

Parameters

`ier = 0`	error parameter
`k = 0`	iteration counter
ε	precision threshold
`kmax`	admissible number of iterations
$\mathbf{x}$	initial estimation, might be a zero vector

Iteration (5.72)

```
A : $k \leftarrow k + 1$
    if  k > kmax  then  ier=-1;  RETURN
    $\mathbf{x} \leftarrow \mathbf{U}\mathbf{s} + \mathbf{v}$            a new iteration
    $\mathbf{r} \leftarrow \|\mathbf{x} - \mathbf{s}\|$          difference of two consecutive iterations
    rnorm = $\|\mathbf{r}\|$          norm of the difference
```

Test convergence

```
        if  rnom < $\varepsilon$  then  $\mathbf{x}$  is solution;  RETURN
                         else  $\mathbf{s} \leftarrow \mathbf{x}$;          GO TO  A
```

On the output the error parameter `ier` has the following meaning

```
ier  = 0    converged,
     = −1   failed, i.e. kmax steps not enough for reaching the required precision.
```

The Basic implementation of previous considerations is in the Program 20.

Program 20

```
4 REM NUMAT17 last rev. 100109, ok
 5 REM Program SimpleIter
 6 REM solution of [A]{x} = {b} by the simple iteration
 10 N=4
 20 DIM A(N,N), V(N), D(N), B(N), X(N), S(N), R(N)
 60 REM parameters
 70 C=1
 80 EPS=.0000001
 90 KMAX=30
 100 PRINT "Solution of [A]{x} = {b}"
 120 REM matrix
 130 FOR I=1 TO N: FOR J=1 TO N
```

```
140   A(I,J)=1
150 NEXT J: NEXT I
160 REM its diagonal
170 FOR I=1 TO N
180   A(I,I)=-9
190 NEXT I
200 REM RHS vector
210 FOR I= 1 TO N
220   B(I)=-I
230 NEXT I
240 PRINT
250 REM
260 PRINT "Matrix"
280 FOR I=1 TO N
290   FOR J=1 TO N
300     PRINT A(I,J);" ";
320   NEXT J
330 PRINT
340 NEXT I
350 REM
360 PRINT "RHS vector"
380 FOR I=1 TO N
390   PRINT B(I);" ";
410 NEXT I
420 REM
440 GOSUB 1000: REM simple iteration
450 IF IER=-1 THEN PRINT "Failed": STOP
460 IF IER=-2 THEN PRINT "norma [U] >= 1, solution will not converge": STOP
470 REM
480 PRINT "Solution for a given eps=";EPS;" c=";C
500 PRINT "was obtained with ";K;" iterations:"
520 FOR I=1 TO N
530   PRINT X(I);" ";
550 NEXT I
560 STOP
1000 REM simple iteration
1002 REM
1004 REM input data
1006 REM c ...... a nonzero constant
1008 REM eps .... prescribed threshold
1010 REM kmax ... admissible number of iterations
1012 REM a(n,n) . matrix (its contents is destroyed on the way)
1014 REM b(n) ... RHS vector
1016 REM output data
1018 REM x(n) ... solution
1020 REM ier .... error parameter
1022 REM     = 0  .... O.K.
1024 REM     = -1 .... failed for a given kmax and eps
1026 REM     = -2 .... norm [U]>= 1, no convergence possible
1028 REM v(n),d(n),s(n) ... auxiliary vectors
1030 REM
1032 IER=0
1034 REM compute d(i)
1036 FOR I=1 TO N
1038   D(I)=1/(A(I,I)+C)
1040 NEXT I
1042 REM compute [E] in place of [A]
```

```
1044 FOR I=1 TO N: FOR J=1 TO N
1046   A(I,J)=-A(I,J)
1048 NEXT J: NEXT I
1050 REM diagonal
1052 FOR I=1 TO N
1054   A(I,I)=C
1056 NEXT I
1058 REM product [D]*[E] in an efficient way
1060 FOR I=1 TO N: FOR J=1 TO N
1062   A(I,J)=D(I)*A(I,J)
1064 NEXT J: NEXT I
1066 REM compute norm of [U]
1068 SUM=0
1070 FOR I=1 TO N: FOR J=1 TO  N
1072   SUM=SUM+A(I,J)*A(I,J)
1074 NEXT J: NEXT I
1076 UNORM=SQR (SUM)
1078 PRINT "norma [U]=";UNORM
1080 IF UNORM>=1 THEN LET IER=-2: RETURN
1082 REM vector {v}
1084 FOR I=1 TO N
1086   V(I)=D(I)*B(I)
1088 NEXT I
1090 K=0
1092 REM initial approximation
1094 FOR I=1 TO N
1096   S(I)=0
1098 NEXT I
1100 REM iteration loop
1102 K=K+1
1104 IF K>KMAX THEN IER=-1: RETURN
1106 REM [U]*{x}+{v}
1108 FOR I=1 TO N
1110   S1=0
1112    FOR J=1 TO N
1114      S1=S1+A(I,J)*S(J)
1116    NEXT J
1118 REM sum [U]*{x}+{v}
1120 X(I)=S1+V(I)
1122 NEXT I
1124 REM difference of old and new iterations
1126 FOR I=1 TO N
1128   R(I)=ABS (X(I)-S(I))
1130 NEXT I
1132 REM norm of the difference
1134 SUM=0
1136 FOR I=1 TO N
1138   SUM=SUM+R(I)*R(I)
1140 NEXT I
1142 RNORM=SQR (SUM)
1144 IF RNORM<EPS THEN RETURN
1146 REM current iteration
1148 PRINT"k=";K
1152 FOR I=1 TO N
1154   PRINT X(I);" ";
1158 NEXT I
1160 PRINT
```

```
1162 REM do it again
1164 FOR I=1 TO N
1166   S(I)=X(I)
1168 NEXT I
1170 GOTO 1100
1182 REM end of simple iteration procedure
```

End of Program 20. □

5.14.2 Jacobi method

Assuming the matrix **A** of (5.70) is regular, one can – using pivoting – avoid zero elements on the diagonal. Then the Eq. (5.70) could be rewritten into the form

$$\begin{aligned} x_1 &= -\tfrac{1}{a_{11}}(0 + a_{12}x_2 + \quad \cdots \quad + a_{1n}x_n - b_1), \\ x_2 &= -\tfrac{1}{a_{22}}(a_{21}x_1 + 0 + \quad \cdots \quad + a_{2n}x_n - b_2), \\ &\vdots \\ x_n &= -\tfrac{1}{a_{nn}}(a_{n1}x_1 + a_{n2}x_2 + \quad \cdots \quad + 0 - b_n). \end{aligned} \tag{5.78}$$

or

$$x_i = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^{n} a_{ij}x_j) \qquad i = 1, \cdots, n. \tag{5.79}$$

The Jacobi iteration is defined as

$$^{(k+1)}x_i = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}\,{}^{(k)}x_j - \sum_{j=i+1}^{n} a_{ij}\,{}^{(k)}x_j) \qquad i = 1, \cdots, n. \tag{5.80}$$

where for $i = 1$

$$\sum_{j=1}^{i-1} a_{ij}x_j = 0,$$

while for $i = n$

$$\sum_{j=i+1}^{n} a_{ij}x_j = 0.$$

To express the Jacobi method in the matrix form we write

$$\mathbf{A} = \mathbf{A}_\mathrm{L} + \mathbf{D} + \mathbf{A}_\mathrm{U},$$

where

$$\mathbf{D} = \begin{bmatrix} a_{11} & & & \\ & a_{22} & & 0 \\ & & \ddots & \\ & 0 & & a_{nn} \end{bmatrix} \tag{5.81}$$

$$\mathbf{A}_\mathrm{L} = \begin{bmatrix} 0 & & & \\ a_{21} & 0 & 0 & \\ \vdots & & \ddots & \\ a_{n1} & \cdots & a_{n,n-1} & 0 \end{bmatrix}, \quad \mathbf{A}_\mathrm{U} = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ & 0 & \cdots & a_{2n} \\ & & \ddots & \\ & 0 & 0 & a_{n-1,n} \\ & & & 0 \end{bmatrix}.$$

The Jacobi iteration (5.80) in matrix form is

$$^{k+1}\mathbf{x} = -\mathbf{D}^{-1}\,(\mathbf{A}_\mathrm{L} + \mathbf{A}_\mathrm{U})\,^{(k)}\mathbf{x} + \mathbf{D}^{-1}\,\mathbf{b}. \tag{5.82}$$

Comparing (5.82) with (5.72) we find that the Jacobi method could be classified as the basic iteration method of the fist order with

$$\begin{aligned} \mathbf{U} &= -\mathbf{D}^{-1}\,(\mathbf{A}_\mathrm{L} + \mathbf{A}_\mathrm{U}), \\ \mathbf{v} &= \mathbf{D}^{-1}\,\mathbf{b}. \end{aligned} \tag{5.83}$$

Convergence of Jacobi method

A sufficient condition for the Jacobi method to converge is

$$|a_{ii}| > \sum_{j=1}^{n} |a_{ij}|, \qquad \text{pro} \qquad i = 1, 2, \cdots, n \tag{5.84}$$

This is also the condition of the diagonal dominance of the matrix.

Comparing the simple and Jacobi iteration methods

Elements of the Jacobi iteration matrix $\mathbf{D}^{-1}(\mathbf{A}_\mathrm{L} + \mathbf{A}_\mathrm{U})$ will be in absolute value smaller than those of $\mathbf{I} - \mathbf{A}$ when the diagonal elements of $\mathbf{A}$ are in absolute value > 1. Furthermore the diagonal elements of the Jacobi iteration matrix are zero. Thus its norm is smaller than that of the iteration norm of the simple iteration and the former method iterates faster.

From it follows that solving the system by Jacobi method is identical to that of simple iteration provided that the initial equation is divided by diagonal elements, i.e.

$$\mathbf{D}^{-1}\,\mathbf{A}\mathbf{x} = \mathbf{D}^{-1}\mathbf{b},$$

since

$$\mathbf{U} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A} = \mathbf{I} - \mathbf{D}^{-1}(\mathbf{A}_\mathrm{L} + \mathbf{D} + \mathbf{A}_\mathrm{U}) = -\mathbf{D}^{-1}(\mathbf{A}_\mathrm{L} + \mathbf{A}_\mathrm{U}).$$

The programming considerations could be followed reading the listing of the Program 21.

Program 21

```
4 REM NUMAT18 last rev. 010392, ok
5 REM Program JACOITER
6 REM solution of [A]{x} = {b} by the Jacobi method
10 N=4
20 DIM A(N,N), D(N), B(N), X(N), S(N), R(N)
60 REM parameters
70 EPSZ=.000001: EPS=.000001: KMAX=30
100 PRINT "Solution of [A]{x} = {b} by the Jacobi method"
120 REM [A] matrix
130 FOR I=1 TO N: FOR J=1 TO N
140   A(I,J)=1
150 NEXT J: NEXT I
160 REM diagonal
170 FOR I=1 TO N
180   A(I,I)=-9
```

```
190 NEXT I
200 REM RHS vector
210 FOR I= 1 TO N
220   B(I)=-I
230 NEXT I
240 PRINT
250 REM
260 PRINT "[A] matrix"
280 FOR I=1 TO N
290   FOR J=1 TO N
300     PRINT A(I,J);" ";
320   NEXT J
330   PRINT
340 NEXT I
350 REM
360 PRINT "RHS vector"
380 FOR I=1 TO N
390   PRINT B(I);" ";
410 NEXT I
420 REM
430 PRINT
440 GOSUB 1000: REM Jacobi iteration
450 IF IER=-1 THEN PRINT "Failed": STOP
460 IF IER=-2 THEN PRINT "Convergence condition not satisfied": STOP
465 IF IER=-3 THEN PRINT "There are zeros on diagonal": STOP
470 REM
480 PRINT "The solution for the prescribed eps=";EPS
500 PRINT "was obtained with  ";K;" iterations:"
520 FOR I=1 TO N
530   PRINT X(I);" ";
550 NEXT I
560 STOP
1000 REM Jacobi iteration
1002 REM Solution od [A]{x} = {b} by Jacobi iteration method
1004 REM input data
1006 REM eps .... prescribed precision
1008 REM epsz ... test for the nonzero value of the diagonal element
1010 REM kmax ... permissible number of iterations
1012 REM a(n,n) . [A] matrix (destroyed)
1014 REM b(n) ... RHS vector
1016 REM output data
1018 REM x(n) ... solution
1020 REM ier .... error parameter
1022 REM     =0  .... O.K.
1024 REM     =-1 .... no convergence for given kmax and eps
1026 REM     =-2 .... convergence condition not satisfied
1027 REM     =-3 .... zero diagonals
1028 REM v(n),d(n),s(n) ... auxiliary vectors
1030 REM
1032 IER=0
1034 REM check the convergence conditions
1036 REM diagonal elements
1038 S1=0
1040 FOR I=1 TO N
1042 S1=S1+ABS(A(I,I))
1044 NEXT I
1046 REM out-of diagonal elements
```

```
1048 S2=0: S3=0
1050 FOR I=1 TO N
1052   FOR J=1 TO I-1
1054     S2=S2+ABS(A(I,J))
1056   NEXT J
1058   FOR J=I+1 TO N
1060     S3=S3+ABS(A(I,J))
1062   NEXT J
1064 NEXT I
1066 IF S1<(S2+S3) THEN IER=-2: RETURN
1068 REM check zeros on the diagonal
1070 FOR I=1 TO N
1072   IF (ABS(A(I,I)) < EPSZ) THEN IER=-3: RETURN
1074 NEXT I
1076 REM zeroth iteration
1078 FOR I=1 TO N
1080   S(I)=0
1082 NEXT I
1084 REM
1086 K=0
1088 REM iteration
1090 K=K+1
1092 IF K>KMAX THEN IER=-1: RETURN
1094 FOR I=1 TO N
1096   S1=0: S2=0
1098   FOR J=1 TO I-1
1100     S1=S1+A(I,J)*S(J)
1102   NEXT J
1104   FOR J=I+1 TO N
1106     S2=S2+A(I,J)*S(J)
1108   NEXT J
1110   X(I)=(B(I)-S1-S2)/A(I,I)
1112 NEXT I
1114 REM difference of old and new iterations
1116 FOR I=1 TO N
1118   D(I)=ABS (X(I)-S(I))
1120 NEXT I
1122 REM its norm
1124 SUM=0
1126 FOR I=1 TO N
1128   SUM=SUM+D(I)*D(I)
1130 NEXT I
1132 RNORM=SQR (SUM)
1134 IF RNORM<EPS THEN RETURN
1136 REM current iteration
1138 PRINT "k=";K
1142 FOR I=1 TO N
1144   PRINT X(I);" ";
1148 NEXT I
1150 PRINT
1152 REM do it again
1154 FOR I=1 TO N
1156   S(I)=X(I)
1158 NEXT I
1160 GOTO 1090
1162 REM end of Jacobi
```

End of Program 21. □

A contemporary way, based on high programming primitives, dealing with matrices as building blocks, that are available in Matlab, is shown in Programs 22 and in 23. The function procedures are taken from [1]. Its electronic version is available at

`http://www.netlib.org/linalg/html_templates/Templates.html.`

Program 22

```
function [x, error, iter, flag]  = jacobi(A, x, b, max_it, tol)

%  -- Iterative template routine --
%     Univ. of Tennessee and Oak Ridge National Laboratory
%     October 1, 1993
%     Details of this algorithm are described in "Templates for the
%     Solution of Linear Systems: Building Blocks for Iterative
%     Methods", Barrett, Berry, Chan, Demmel, Donato, Dongarra,
%     Eijkhout, Pozo, Romine, and van der Vorst, SIAM Publications,
%     1993. (ftp netlib2.cs.utk.edu; cd linalg; get templates.ps).
%
% [x, error, iter, flag]  = jacobi(A, x, b, max_it, tol)
%
% jacobi.m solves the linear system Ax=b using the Jacobi Method.
%
% input   A        REAL matrix
%         x        REAL initial guess vector
%         b        REAL right hand side vector
%         max_it   INTEGER maximum number of iterations
%         tol      REAL error tolerance
%
% output  x        REAL solution vector
%         error    REAL error norm
%         iter     INTEGER number of iterations performed
%         flag     INTEGER: 0 = solution found to tolerance
%                           1 = no convergence given max_it

  iter = 0;                                       % initialization
  flag = 0;

  bnrm2 = norm( b );
  if  ( bnrm2 == 0.0 ), bnrm2 = 1.0; end

  r = b - A*x;
  error = norm( r ) / bnrm2;
  if ( error < tol ) return, end

  [m,n]=size(A);
  [ M, N ] = split( A , b, 1.0, 1 );              % matrix splitting

  for iter = 1:max_it,                            % begin iteration

     x_1 = x;
     x   = M \ (N*x + b);                         % update approximation

     error = norm( x - x_1 ) / norm( x );         % compute error
     if ( error <= tol ), break, end              % check convergence

  end
  if ( error > tol ) flag = 1; end                % no convergence

% END jacobi.m
%
```

End of Program 22. □

You may notice that the `split` function is required. It is as follows.

Program 23

```
function [ M, N, b ] = split( A, b, w, flag )
%
% function [ M, N, b ] = split( A, b, w, flag )
%
% split.m sets up the matrix splitting for the stationary
% iterative methods: jacobi and sor (gauss-seidel when w = 1.0 )
%
% input   A        DOUBLE PRECISION matrix
%         b        DOUBLE PRECISION right hand side vector (for SOR)
%         w        DOUBLE PRECISION relaxation scalar
%         flag     INTEGER flag for method: 1 = jacobi
%                                           2 = sor
%
% output  M        DOUBLE PRECISION matrix
%         N        DOUBLE PRECISION matrix such that A = M - N
%         b        DOUBLE PRECISION rhs vector ( altered for SOR )

  [m,n] = size( A );

  if ( flag == 1 ),                  % jacobi splitting

     M = diag(diag(A));
     N = diag(diag(A)) - A;

  elseif ( flag == 2 ),              % sor/gauss-seidel splitting

     b = w * b;
     M =  w * tril( A, -1 ) + diag(diag( A ));
     N = -w * triu( A,  1 ) + ( 1.0 - w ) * diag(diag( A ));

  end;

% END split.m
```

End of Program 23. □

5.14.3 Gauss-Seidel method

The convergence of the Jacobi method could naturally be accelerated by using the newly computed iteration matrix elements immediately after they have been obtained. It means that when computing ${}^{(k+1)}\mathbf{x}$ according to (5.80) we use ${}^{(k+1)}x_2$ instead of ${}^{(k)}x_2$ as before.

The Gauss-Seidel iteration could then be described by

$$ {}^{(k+1)}x_i = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}\,{}^{(k+1)}x_j - \sum_{j=i+1}^{n} a_{ij}\,{}^{(k)}x_j), \qquad i = 1, \cdots, n. \tag{5.85} $$

In matrix form we have

$$(\mathbf{A}_{\mathrm{L}} + \mathbf{D})\,{}^{(k+1)}\mathbf{x} = -\mathbf{A}_{\mathrm{U}}\,{}^{(k)}\mathbf{x} + \mathbf{b},$$

i.e.

$$^{(k+1)}\mathbf{x} = (\mathbf{A}_{\mathrm{L}} + \mathbf{D})^{-1}\,(-\mathbf{A}_{\mathrm{U}}\quad {}^{(k)}\mathbf{x} + \mathbf{b}). \tag{5.86}$$

Gauss-Seidel iteration matrix and the corresponding vector are

$$\mathbf{U} = -(\mathbf{A}_{\mathrm{L}} + \mathbf{D})^{-1}\,\mathbf{A}_{\mathrm{U}},$$

$$\mathbf{v} = (\mathbf{A}_{\mathrm{L}} + \mathbf{D})^{-1}\,\mathbf{b}.$$

Convergence of Gauss-Seidel method

A sufficient condition for the convergence is the same as that for the Jacobi method, i.e. the matrix diagonal dominance of the **A** matrix

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|, \qquad i = 1, 2, \cdots, n.$$

The Gauss-Seidel method converges for any initial approximation, provided the matrix **A** is symmetric, positive definite. See [8], [23]. All the needed steps for implementing the Gauss-Seidel algorithm on the element level are listed in the Program 24.

Program 24

```
4 REM NUMAT19 last rev. 010392, ok
5 REM Program GSITER
6 REM Solution of [A]{x} = {b} by Gauss Seidel
10 N=4
20 DIM A(N,N), D(N), B(N), X(N), S(N), R(N)
60 REM parameters
70 EPSZ=.000001: EPS=.000001: KMAX=30
100 PRINT "solution of [A]{x} = {b} by Gauss Seidel"
120 REM [A] matrix
130 FOR I=1 TO N: FOR J=1 TO N
140   A(I,J)=1
150 NEXT J: NEXT I
160 REM diagonal
170 FOR I=1 TO N
180   A(I,I)=-9
190 NEXT I
200 REM RHS vector
210 FOR I=1 TO N
220   B(I)=-I
230 NEXT I
240 PRINT
250 REM
260 PRINT "[A] matrix"
280 FOR I=1 TO N
290   FOR J=1 TO N
300     PRINT A(I,J);" ";
```

```
320   NEXT J
330   PRINT
340 NEXT I
350 REM
360 PRINT "RHS vector"
380 FOR I=1 TO N
390   PRINT B(I);" ";
410 NEXT I
420 REM
430 PRINT
440 GOSUB 1000: REM Gauss-Seidel method
450 IF IER=-1 THEN PRINT "Failed": STOP
460 IF IER=-2 THEN PRINT "Convergence condition not satisfied": STOP
465 IF IER=-3 THEN PRINT "The are zeros on the diagonal": STOP
470 REM
480 PRINT "The solution for a given eps= ";EPS
500 PRINT "was obtained with ";K;" iterations:"
520 FOR I=1 TO N
530   PRINT X(I);" ";
550 NEXT I
560 STOP
1000 REM Gauss-Seidel iteraction
1002 REM Solution of [A]{x} = {b} by Gauss-Seidel method
1004 REM input parameters
1006 REM epsz ... zero threshold for diagonal elements
1008 REM eps .... prescribed precision
1010 REM kmax ... permissible number of iterations
1012 REM a(n,n) . [A] matrix (destroyed)
1014 REM b(n) ... RHS vector
1016 REM output parameters
1018 REM x(n) ... solution
1020 REM ier .... error parameter
1022 REM     = 0 .... O.K.
1024 REM     =-1 .... no convergence for given kmax and eps
1026 REM     =-2 .... convergence condition not satisfied
1027 REM     =-3 .... zero diagonals
1028 REM v(n),d(n),s(n) ... auxiliary vectors
1030 REM
1032 IER=0
1034 REM check the convergence conditions
1036 REM diagonal elements
1038 S1=0
1040 FOR I=1 TO N
1042   S1=S1+ABS(A(I,I))
1044 NEXT I
1046 REM out-of diagonal elements
1048 S2=0: S3=0
1050 FOR I=1 TO N
1052   FOR J=1 TO I-1
1054     S2=S2+ABS (A(I,J))
1056   NEXT J
1058   FOR J=I+1 TO N
1060     S3=S3+ABS(A(I,J))
1062   NEXT J
1064 NEXT I
1066 IF S1<(S2+S3) THEN IER=-2: RETURN
1068 REM check zeros on the diagonal
```

```
1070 FOR I=1 TO N
1072   IF (ABS(A(I,I))<EPSZ) THEN IER=-3: RETURN
1074 NEXT I
1076 REM zero-th iteration
1078 FOR I=1 TO N
1080   S(I)=0
1082 NEXT I
1084 REM
1086 K=0
1088 REM iteration
1090 K=K+1
1092 IF K>KMAX THEN IER=-1: RETURN
1094 FOR I=1 TO N
1096   S1=0: S2=0
1098   FOR J=1 TO I-1
1100     S1=S1+A(I,J)*X(J)
1102   NEXT J
1104   FOR J=I+1 TO N
1106     S2=S2+A(I,J)*S(J)
1108   NEXT J
1110   X(I)=(B(I)-S1-S2)/A(I,I)
1112 NEXT I
1114 REM difference of old and new iterations
1116 FOR I=1 TO N
1118   D(I)=ABS (X(I)-S(I))
1120 NEXT I
1122 REM its norma
1124 SUM=0
1126 FOR I=1 TO N
1128   SUM=SUM+D(I)*D(I)
1130 NEXT I
1132 RNORM=SQR (SUM)
1134 IF RNORM<EPS THEN RETURN
1136 REM current iteration
1138 PRINT "k=";K
1142 FOR I=1 TO N
1144   PRINT X(I);" ";
1148 NEXT I
1150 PRINT
1152 REM update and do it again
1154 FOR I=1 TO N
1156   S(I)=X(I)
1158 NEXT I
1160 GOTO 1090
1162 REM end of Gauss-Seidel
```

End of Program 24. □

5.14.4 Successive overrelaxation (SOR) method

The Gauss-Seidel method could be accelerated by introducing so called *relaxation parameter*, which is denoted ω here.

The new approximation is taken as

$$^{(k+1)}x_i = {}^{(k)}x_i + \omega \, ({}^{(k+1)}x_i^{\mathrm{GS}} - {}^{(k)}x_i), \tag{5.87}$$

where ${}^{(k+1)}x_i^{\mathrm{GS}}$ is the approximation taken from the Gauss-Seidel method. For $\omega = 1$ the method is identical with the Gauss-Seidel method. Substituting (5.85) into (5.87) we get

$$ {}^{(k+1)}x_i = (1-\omega)\ {}^{(k)}x_i + \frac{\omega}{a_{ii}}\,(b_i - \sum_{j=1}^{i-1} a_{ij}\ {}^{(k+1)}x_j - \sum_{j=i+1}^{n} a_{ij}\ {}^{(k)}x_j), \qquad i = 1, \cdots, n \quad (5.88)$$

or

$$ {}^{(k+1)}x_i = {}^{(k)}x_i + \frac{\omega}{a_{ii}}\,(b_i - \sum_{j=1}^{i-1} a_{ij}\ {}^{(k+1)}x_j - \sum_{j=i}^{n} a_{ij}\ {}^{(k)}x_j). \quad (5.89)$$

The matrix formulation of (5.88) is

$$ {}^{(k+1)}\mathbf{x} = (\mathbf{D} + \omega\mathbf{A}_{\mathrm{L}})^{-1}(-\omega\mathbf{A}_{\mathrm{U}} + (1-\omega)\mathbf{D})\ {}^{(k)}\mathbf{x} + \omega\mathbf{b}. \quad (5.90)$$

The iteration matrix of the successive overrelaxation method is

$$\mathbf{U} = (\mathbf{D} + \omega\mathbf{A}_{\mathrm{L}})^{-1}\,(-\omega\mathbf{A}_{\mathrm{U}} + (1-\omega)\mathbf{D}).$$

Convergence of the successive overrelaxation method

A necessary condition for the SOR method to converge is that the relaxation parameter ω is bounded by $0 < \omega < 2$. This is the condition necessary but not sufficient.

If, however the **A** matrix is symmetric and positive definite then the SOR method converges for every $\omega \in (0, 2)$ and for every initial approximation. For more details see [32]. The Basic program in can be found in [17].

5.15 Solution of large systems

Modelling the continuum mechanics problems by discretization methods often requires solving the system of large algebraic equations.

What is a large scale problem in computational mechanics can be characterized by the order of a matrix which is subjected to solution of linear algebraic equations. The notion of 'largeness' of a matrix size dramatically changed during our lifetime. In 1965 it was the order of 100 by 100 which was considered large by A. Ralston ([26]), B.N. Parlett ([22]) in 1978 considered a symmetric matrix 200 by 200 being large. In 1993 M. Papadrakakis ([21]) discusses the medium sized problems with a few hundred thousands equations. The largest matrix used for testing the high performance computers in 2007 was of the order 2 456 063. The matrix was full, nonsymmetric. See

`www.top500.org`.

Today, (written at the end of 2008) one can expect troubles with storing more than a million of equations in the operational memory of currently available computers. A definition of what is a currently available computer is politely declined here.

So for large problems, approximately defined above, the methods using auxiliary disk storage have to be used. Suddenly the problem cannot be solved by a brute force and all the special matrix features (symmetry, bandedness, positive definiteness) should be efficiently employed in order to minimize the transfer of data between the internal memory and the disk medium.

Modelling physical and engineering tasks in continuum mechanics by *methods based on the discretization* requires to solve large systems of algebraic equations. The memory requirements often exceeds the available internal memory of a computer available for the task. The memory problems could be circumvented either

- by *domain decomposition methods* described in the Chapter 7,
- or by using the famous *frontal method*, invented by B. Irons see [11] and [9]. The frontal method does not carry out the global stiffness assembly process explicitly – instead it provides the Gauss elimination on neighboring finite elements and stores the intermediary results on a disk medium. As such the frontal method is capable of solving the systems with millions of equations. For the programming details see the Chapter 10,
- or by *disk oriented algorithms* based on storing the working matrices and/or their parts on an external memory medium, usually on a disk. See the paragraph 5.15.1.

In this paragraph we will concentrate on the implementation of Gauss elimination process with a matrix stored in the rectangular storage mode in a disk file. The multiple right-hand sides are allowed.

Of course the communication with external memory is substantially slower than with the internal one, so when designing software based on these considerations, one has to try to minimize the the amount of data transfers. And of course to employ all the nice matrix properties as their symmetry, sparsity, positive definiteness, etc.

In this respect the nature is kind to us since our matrices are frequently endowed with the above mentioned properties.

5.15.1 Disk band algorithm

Let the half-band width, including the diagonal, is denoted `NBAND`. It is assumed that the upper part of matrix is stored columnwise on the disk file the as indicated in Fig. 5.5. One should notice that only the triangular part of the matrix under the row containing the current pivot is influenced by the corresponding elimination operations at each elimination step. After the elimination step with the row containing the pivot has been finished and stored on the disk, the next consecutive column is being read from the disk. The size of internal memory needed for this kind of elimination algorithm is proportional to the hatched triangle which moves down during the elimination process.

A similar algorithm, together with Fortran listing, is presented in [31].

A more effective algorithm, coming from the workshop of the Institute of Thermomechanics, is listed in the Program 25.

Program 25

```
      SUBROUTINE DBANGZ(N,K,KK,A,DS,B,EPS,KEY,LKN,FNAM,IRECL,MAXR,NZ)
C
C     THE CATALOG NAME OF THIS SUBROUTINE IS 'S.DBANGZ'
C     Programmed and tested by Ivo Hunek
C     ***********************************************************
C     *                                                         *
C     *   SOLUTION OF A SYSTEM OF LINEAR ALGEBRAIC EQUATIONS    *
```

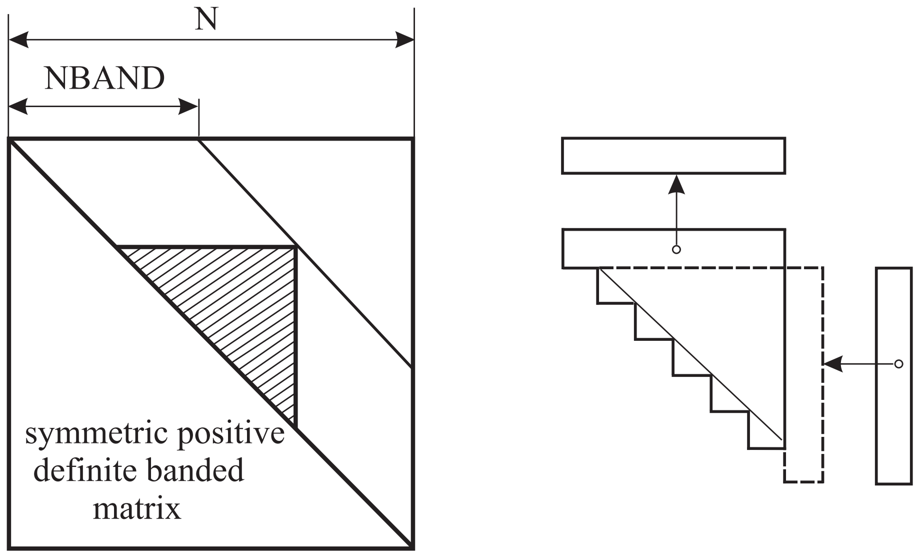


Figure 5.5: Rectangular band storage on the disk

```
C      *   WITH POSITIVE DEFINITE SYMMETRIC AND BANDED MATRIX    *
C      *       BY DIRECT-ACCESS GAUSS ELIMINATION METHOD.        *
C      *    (SUITABLE FOR LARGE SYSTEMS OF EQUATIONS IN FEM)     *
C      *                                                         *
C      ***********************************************************
C
C      * DOUBLE PRECISION VERSION *
C      IN THE CALLING PROGRAM THE ARRAY NZ(K) MUST BE DECLARED
C
C      BEFORE CALLING PROCEDURE THE FILE MUST EXIST ON THE DISC
C      WITH DIRECT ACCESS,LINK NAME 'LKN' AND WITH N RECORDS,
C      WHICH CONTAINS BANDED MATRIX,EFFICIENTLY STORED
C      IN RECTANGULAR FORM.
C
C      -------------------------------------------------------------------
C      DESCRIPTION OF PARAMETERS:
C
C      N.........NUMBER OF EQUATIONS=NUMBER OF RECORDS IN INPUT DISC
C                FILE
C      K.........HALF WIDTH OF BAND OF MATRIX (WITH DIAGONAL)
C      KK........NUMBER OF ROWS OF SYSTEM'S MATRIX,
C                WHICH MAY BE IN MEMORY AT THE SAME TIME
C                K<=KK<=N
C      A(KK,K)...WORKING MATRIX,IN WHICH ROWS OF SYSTEM'S MATRIX
C                ARE HANDLED (IT MUST BE DECLARED IN MAIN)
C      DS(N).....WORKING VECTOR
C                 ON INPUT: ARBITRARY
C                 ON OUTPUT: VECTOR OF SOLUTION
C      B(N)......THE RIGHT SIDE VECTOR
C      EPS.......IF ELEMENT OF MATRIX IS >= EPS,THEN
C                IT IS TAKEN AS NON-ZERO *** DOUBLE PRECISION ***
C      KEY.......SOLUTION KEY
C                 =1       REDUCTION OF MATRIX
C                 =2       REDUCTION OF THE RIGHT SIDE VECTOR
C                          AND BACK SUBSTITUTION
```

```
C     LKN.......LINK NAME OF FILE,IN WHICH A MATRIX OF SYSTEM IS STORED
C     FNAM......NAME OF DISC FILE (TYPE CHARACTER - IT MUST BE
C               DECLARED IN MAIN, MAXIMUM IS 12 CHARACTERS)
C     IRECL.....RECORD LENGHT (IN BYTES)
C     MAXR......MAXIMUM NUMBER OF RECORDS
C     NZ(K).....WORKING VECTOR
C     ------------------------------------------------------------
C
      CHARACTER*(12,V) FNAM
      OPEN(LKN,FILE=FNAM,ACCESS='DIRECT',STATUS='OLD',
     /     RECL=IRECL,MAXREC=MAXR)
C
      DIMENSION A(KK,K),DS(N),B(N),NZ(K)
      DOUBLE PRECISION A,DS,B,RATIO,EPS
C
C     VECTOR OF RIGHT SIDES B INTO WORKING VECTOR DS
      DO 2 I=1,N
2     DS(I)=B(I)
C
      II=KEY
      GO TO (1000,2000), II
C
C     READ FIRST PART OF MATRIX
1000  DO 1 I=1,KK
1     READ(LKN'I) (A(I,J),J=1,K)
C
     JUMP=0
C
C     IEQ...STEP IN GAUSS ELIMINATION
     DO 6 IEQ=1,N
C     WRITE(6,21)
C21   FORMAT(1X,' MEZIVYSLEDKY: MATICE A(KK,K)'/)
C     DO 22 ID=1,KK
C22   WRITE(6,24) (A(ID,JD),JD=1,K)
C24   FORMAT(1X,10E12.6)
      JUMP=JUMP+1
      IF(JUMP.GT.KK) JUMP=JUMP-KK
      I=0
      DO 29 J=2,K
      IF (ABS(A(JUMP,J)).LT.EPS) GO TO 29
      I=I+1
      NZ(I)=J
29    CONTINUE
      IF (I.EQ.0) GO TO 4
      JUMP1=JUMP-1
      DO 5 L=1,I
      M=NZ(L)
      ITMP=JUMP1+M
      IF(ITMP.GT.KK) ITMP=ITMP-KK
      IF(ABS(A(JUMP,1)).LT.EPS) GO TO 300
      RATIO=A(JUMP,M)/A(JUMP,1)
      IR1=M-1
      DO 3 JC=L,I
      MM=NZ(JC)
      JTMP=MM-IR1
3     A(ITMP,JTMP)=A(ITMP,JTMP)-RATIO*A(JUMP,MM)
5     CONTINUE
```

```
C
4     KT=IEQ+KK
      IF(KT.GT.N) GO TO 6
      KQ=IEQ
      WRITE(LKN'KQ) (A(JUMP,J),J=1,K)
      READ(LKN'KT)(A(JUMP,J),J=1,K)
6     CONTINUE
C
C     RECORD LAST BLOCK OF MATRIX ON DISC
      IND1=(N/KK)*KK+1
      IND2=N
      M=1
      DO 14 I=IND1,IND2
      WRITE(LKN'I) (A(M,J),J=1,K)
      M=M+1
14    CONTINUE
C
      IND1=N-KK+1
      IND2=(N/KK)*KK
      DO 16 I=IND1,IND2
      WRITE(LKN'I) (A(M,J),J=1,K)
      M=M+1
16    CONTINUE
C
C     REDUCTION SUCCESSFULLY ENDED
C
      RETURN
C
C     ----------------------------------------------------------------
C
C     REDUCTION OF VECTOR DS
C     READ FIRST PART OF A MATRIX
2000  DO 100 I=1,KK
100   READ(LKN'I)(A(I,J),J=1,K)
C
      JUMP=0
C
      DO 160 IEQ=1,N
      JUMP=JUMP+1
      IF(JUMP.GT.KK) JUMP=JUMP-KK
C
      DO 150 IR=2,K
      IF(ABS(A(JUMP,IR)).LT.EPS) GO TO 150
      IR1=IR-1
      RATIO=A(JUMP,IR)/A(JUMP,1)
      DS(IEQ+IR1)=DS(IEQ+IR1)-RATIO*DS(IEQ)
      IF(ABS(DS(IEQ+IR1)).LT.1.D-30) DS(IEQ+IR1)=0.D0
150   CONTINUE
C
      KT=IEQ+KK
      IF(KT.GT.N) GO TO 160
      READ(LKN'KT)(A(JUMP,J),J=1,K)
160   CONTINUE
C
C     BACK SUBSTITUTION
C
      DS(N)=DS(N)/A(JUMP,1)
      I=N
C
      DO 9 M=2,KK
     JUMP=JUMP-1
      IF(JUMP.EQ.0) JUMP=KK
      I=I-1
      ITMP=I-1
C
      DO 10 J=2,K
      IF(ABS(A(JUMP,J)).GT.EPS) DS(I)=DS(I)-A(JUMP,J)*DS(ITMP+J)
10    CONTINUE
C
9     DS(I)=DS(I)/A(JUMP,1)
C
      IF(I.EQ.1) RETURN
12    I=I-1
      READ(LKN'I) (A(1,M),M=1,K)
      ITMP=I-1
C
      DO 8 J=2,K
      IF(ABS(A(1,J)).GT.EPS) DS(I)=DS(I)-A(1,J)*DS(ITMP+J)
8     CONTINUE
C
      DS(I)=DS(I)/A(1,1)
      IF(I.GT.1) GO TO 12
     GO TO 200
C
C
300   WRITE(6,310) JUMP,A(JUMP,1)
310   FORMAT(1X,' ERROR-RIGIDITY MATRIX NOT POSITIVE DEFINITE'//
     /       1X,' JUMP=',I6,5X,'A(JUMP,1)=',E14.6)
      STOP 01
C
200   RETURN
      END
```

End of Program 25. □

The algorithm was designed for symmetric, positive definite matrices with a constant band width. Only the upper symmetric part, from the diagonal to the band boundary, is stored (rowwise) in the disk memory as indicated in the Fig. 5.6. The intermediate working array for the Gauss elimination, which is kept in the operational memory, is rectangular and has dimensions `KK` $\times$ `K`, where the number of working rows is `KK` $\geq$ `K`. The row dimension of the working array `KK` should be as large as possible, constrained by the available memory size, to minimize the number of data transfers between the disk and the operational memory. In the fist step, see Fig. 5.7, the rows `1` to `KK` read from the disk file, proper elimination steps within the working array are executed and then the first row is written to the disk file and is replaced by the Then the first row is replaced the (`KK + 1`)-th row, the second one by the (`KK + 2`)-th row, etc.

Detailed description of the `DBANGZ` procedure

The system of algebraic equations $\mathbf{Ax} = \mathbf{b}$ is solved. It is assumed that the matrix is symmetric, banded, positive definite. The classical Gauss elimination without pivoting

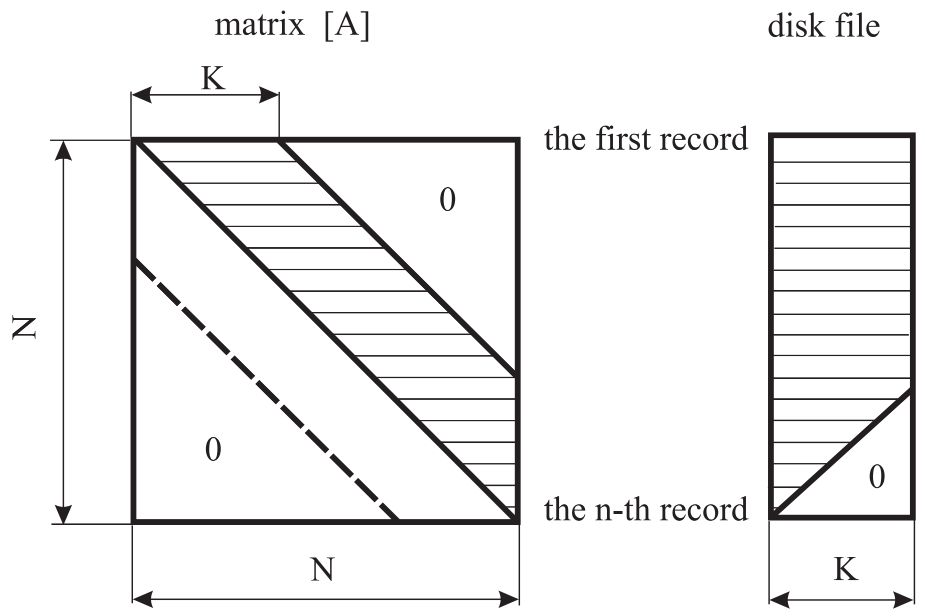


Figure 5.6: Rectangular band storage scheme. Caution: Identifier `K` is used here instead of `NBAND`

is used. The number of records of the disk file, see Fig. 5.6, is equal to the number of equations `N`. In this case the half-band width is denoted `K`, instead of `NBAND` as it was before. Number of rows kept in the memory, i.e. `KK`, is stored in an auxiliary vector `A`. The minimum number of rows of `A` is the bandwidth size, i.e. `K`.

The subroutine `DBANGZ` is a modification of that initially described in[19], which in turn is based on the program published in [20]. Now the subroutine `DBANGZ` allows for multiple right-hand sides.

The input and output parameters of the procedure are at the beginning of the listing in the Program 25.

At first, the right-hand side (RHS) vector `B` is copied into the auxiliary vector `DS`, just in case it is needed again. Then, depending on the `KEY` value, i.e. `1` or `2`, the triangular factorization or the RHS reduction and the back substitution is carried out.

Triangular factorization – `KEY = 1`

The first part of the matrix, defined by `KK` $\times$ `K`, is copied into the auxiliary array `A`. The pointer `IEQ` denotes the row, containing the current pivot. The `IEQ`-th row of the matrix, also the `IEQ`-th record of the disk file, is in the auxiliary array stored in its `JUMP`-th row. So the the integer variable `JUMP` varies from `1`, to `KK`. See the do-loop with label 6.

The pivot for the `IEQ`-th elimination step is – due to the assumed rectangular storage scheme – located at `A(JUMP,1)`. Having the general storage scheme for `N` $\times$ `N` in mind, then at each step the nonzero elements in the column under the pivot are to be eliminated. Due to the matrix symmetry, and the rectangular storage scheme being used, these elements are identical with `A(JUMP,J)` for `J = 2` to `K`.

See Fig. 5.8, where a hypothetical situation at the third elimination step is indicated. Now, one has to eliminate the elements with indices `(5,3)` and `(6,3)`, in the classical general storage scheme – on the left in Fig. 5.8 – whose position in the 'rectangular'

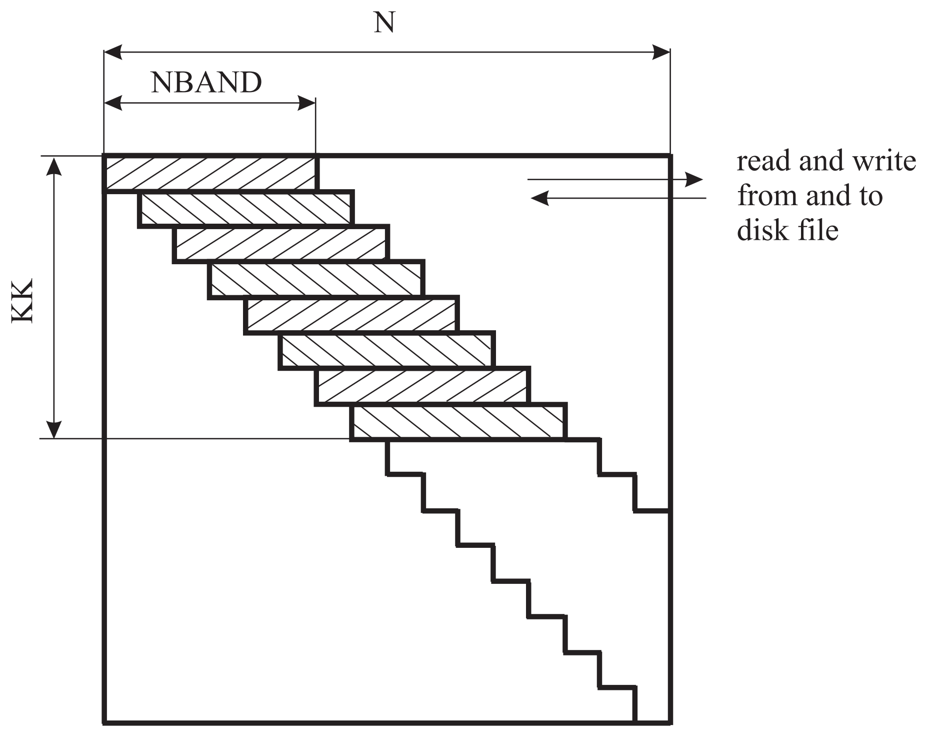


Figure 5.7: Data transfer

storage scheme (on the right) is `(JUMP,3)` and `(JUMP,4)`, with `JUMP=3` in the `A` array. The `J` indices of non-zero elements in `A(JUMP,J)` are being stored in the auxiliary array `NZ`. which efficiently controls the subsequent processing of following rows. See do-loop with label 29.

If there is at least one pointer in the `NZ` array, the under-pivot elements in subsequent rows are nulled. The pointers to these rows in the `A` array are given by `ITMP = (JUMP+NZ(L)-1) modulo`[6] `KK`, where `L = 1 ... I` with `I` being the number of non-zero elements in `A(JUMP,J)`.

The multiplier for the `ITMP`-th row is obtained from `RATIO = A(JUMP,NZ(L)) / A(JUMP,1)` with `L = 1, ... I`. Next, the elements in the `ITMP`-th row are processed. The process is limited only to those elements `A(ITMP, JTMP)`, whose column index is `JTMP = NZ(JC) - NZ(L) + 1`, where `JC = L, ... ,I`, i.e. only for such elements having a non-zero element in the pivot row 'above' them (the term 'above' is expressed 'optically' in the general storage mode). This way the repeated check of non-zero elements is avoided.

Having finished the `IEQ`-th step of elimination, the row – pointed to by `JUMP` value appearing in `A`) – is not needed any longer and is written into the `IEQ`-th record of the disk file. Then the `IEQ+KK`-th row from the disk (the next one) is read from the disk file.

The subsequent over-writing the rows of the auxiliary `A` array requires to compute the row pointer modulo `KK`.

[6]Modulo function provides the remainder after the integer division.

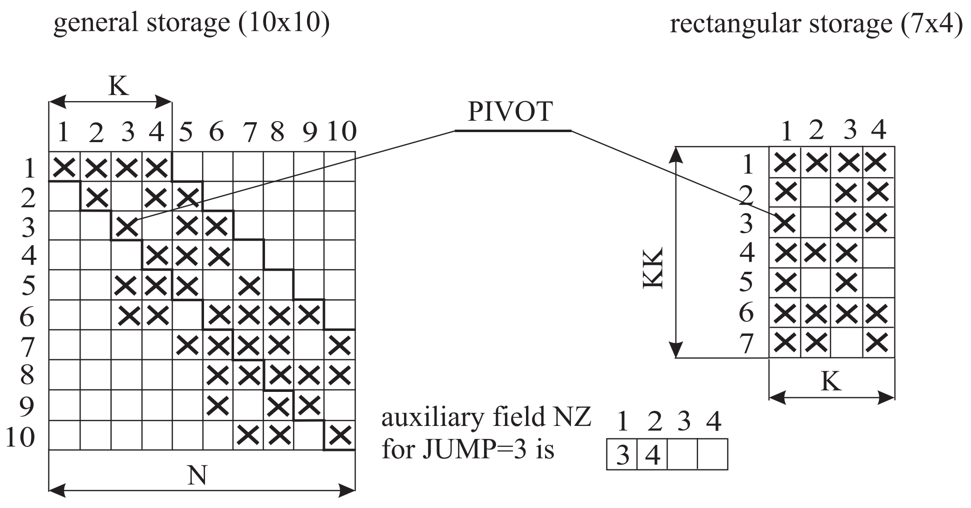


Figure 5.8: General and rectangular storage compared. Crosses indicate non-zero elements

Reduction of RHS and backsubstitution – `KEY = 2`

- Reduction of the right hand side

  The first part of the triangularized matrix (rows `1` ··· `KK`) is read from the disk and stored into the auxiliary array `A`. The multiplier is evaluated the same way as before, i.e. by `RATIO = A(JUMP,IR)/A (JUMP,1)`, where `IQ` os the pointer to nonzero elements of the `JUMP`-th row, i.e. `2` $\leq$ `IR` $\leq$ `K`.

- Backsubstitution

  The contents of the auxiliary array `DS`, is being filed by newly computed components of the solution backwards. The last `KK` components is computed directly, since the elements of the last part of the triangularized matrix is already available in the CPU memory.

5.15.2 Block Gauss elimination

This method is limited neither by the halfband size, nor by the number of equation assuming that the available disk memory is sufficient.

The principle of this approach consists in division of the original matrix into square submatrices of the same size and expressing the elimination process with submatrices. (See [4] and [31])

$$\mathbf{K}^*_{ij} = \mathbf{K}_{ij} - \mathbf{K}^{\mathrm{T}}_{is}\,\mathbf{K}^{-1}_{ss}\,\mathbf{K}_{sj}.$$

If the number of equation is a power of 2, the recursive approach could be implemented. See [24], [10].

5.16 Solution of overdetermined systems

Sometimes we could have more equation than unknowns – then the system of equations has the form

$$\mathbf{A}_{m\times n}\mathbf{x}_{n\times 1} - \mathbf{b}_{m\times 1} = \mathbf{0}_{m\times 1} \tag{5.91}$$

with $\mathbf{A}$ matrix being rectangular. The Eq. (5.91) is classified as *overdetermined* and generally is not solvable, i.e. one cannot find the $\mathbf{x}$ vector satisfying the Eq. (5.91) exactly. The overdetermined system usually arise as the result of repeated experimental measurements and it would be unwise to disregard any of $m - n$ equations to get a solution.

Instead, assuming that all the equation have the same statistical significance, we will try to get a 'closest possible' solution. We could proceed as follows. First, we compute a residuum of the overdetermined system

$$\mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}, \tag{5.92}$$

and try to find conditions for its minimization. A good measure of the 'best' solution might be the minimum of the Euclidian norm of the residuum. The square of that norm will satisfy the required condition as well

$$\|\mathbf{r}\|^2 = (\mathbf{r}, \mathbf{r}) = \mathbf{r}^{\mathrm{T}}\mathbf{r}. \tag{5.93}$$

Substituting $\mathbf{r}$ from (5.92) into (5.93) we get

$$\|\mathbf{r}\|^2 = \mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{b} - \mathbf{b}^{\mathrm{T}}\mathbf{b}. \tag{5.94}$$

The middle scalar terms in 5.94 are identical, so

$$\|\mathbf{r}\|^2 = \mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} - 2\mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{b} - \mathbf{b}^{\mathrm{T}}\mathbf{b}. \tag{5.95}$$

The extremal value is obtained from the condition of a zero derivative. Since it holds

$$\frac{\partial(\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x})}{\partial\mathbf{x}} = 2\mathbf{A}\mathbf{x}, \qquad \frac{\partial(\mathbf{x}^{\mathrm{T}}\mathbf{a})}{\partial\mathbf{x}} = \mathbf{a}, \tag{5.96}$$

we can write

$$\frac{\partial\|\mathbf{r}\|^2}{\partial\mathbf{x}} = 2\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} - 2\mathbf{A}^{\mathrm{T}}\mathbf{b} = \mathbf{0} \tag{5.97}$$

or

$$\mathbf{A}^{\mathrm{T}}(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{0}. \tag{5.98}$$

The extremum $\mathbf{x}$ obtained from (5.98) is really the sought-after minimum. For any other vector, say $\mathbf{x}'$, we would have

$$\begin{aligned}
&\|\mathbf{A}\,\mathbf{x}' - \mathbf{b}\|^2 = (\mathbf{A}\,\mathbf{x}' - \mathbf{b}, \mathbf{A}\,\mathbf{x}' - \mathbf{b}) = \\
&= (\mathbf{A}(\mathbf{x}' - \mathbf{x}) + \mathbf{A}\,\mathbf{x} - \mathbf{b}, \mathbf{A}(\mathbf{x}' - \mathbf{x}) + \mathbf{A}\,\mathbf{x} - \mathbf{b}) = \\
&= (\mathbf{A}(\mathbf{x}' - \mathbf{x}), \mathbf{A}\,(\mathbf{x}' - \mathbf{x})) + 2(\mathbf{A}(\mathbf{x}' - \mathbf{x}), \mathbf{A}\,\mathbf{x} - \mathbf{b}) + \\
&+ (\mathbf{A}\,\mathbf{x} - \mathbf{b}, \mathbf{A}\,\mathbf{x} - \mathbf{b}) = \\
&= \|\mathbf{A}\,(\mathbf{x}' - \mathbf{x})\|^2 + \|\mathbf{A}\,\mathbf{x} - \mathbf{b}\|^2 \geq \\
&\geq \|\mathbf{A}\,\mathbf{x} - \mathbf{b}\|^2,
\end{aligned} \tag{5.99}$$

since

$$\begin{aligned}&(\mathbf{A}\,(\mathbf{x}'-\mathbf{x}, \mathbf{A}\,\mathbf{x}-\mathbf{b})) = \\ &= (\mathbf{A}\,(\mathbf{x}'-\mathbf{x}))^{\mathrm{T}}\,(\mathbf{A}\,\mathbf{x}-\mathbf{b}) = \\ &= (\mathbf{x}'-\mathbf{x})^{\mathrm{T}}\,\mathbf{A}^{\mathrm{T}}\,(\mathbf{A}\,\mathbf{x}-\mathbf{b}) = 0\end{aligned} \tag{5.100}$$

the last term is equal to zero. The 'best possible' solution of (5.98) can be obtained from

$$\mathbf{A}^{\mathrm{T}}\,\mathbf{A}\,\mathbf{x} = \mathbf{A}^{\mathrm{T}}\,\mathbf{b} \tag{5.101}$$

provided that $\mathbf{A}^{\mathrm{T}}\,\mathbf{A}$ matrix is regular.

It should be emphasized that individual equation of (5.101) are usually 'almost' linearly dependent since they were obtained by measuring the same phenomenon under the 'same' conditions. If they were 'fully' linearly dependent, we could proclaim the measurement as contradictory and not of a great significance. From it follows that the condition number of $\mathbf{A}^{\mathrm{T}}, \mathbf{A}$ matrix will be large and that the obtained result will be subjected to significant round-off errors. See [13].

5.17 Solution with some of unknowns prescribed

Solving the problems of technical practice we are often required to solve a system of algebraic equation with a few of unknowns known in advance. Using the deformation variant of the finite element method, see [3], the solution of a static problem is prescribed by

$$\mathbf{K}\mathbf{q} = \mathbf{F},$$

where $\mathbf{K}$ is the stiffness matrix, the $\mathbf{q}$ and $\mathbf{F}$ vectors contain the unknown displacements and the prescribed external forces respectively.

Now, let's assume that some displacements are known, while the corresponding forces are not, so the overall number of equations is the same, as before.

The stiffness matrix could be reordered in such a way that the unknown displacements and forces are separated, i.e.

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \end{Bmatrix}, \tag{5.102}$$

where K_{ij} and q_L, F_k are submatrices and subvectors respectively. In (5.102) we denoted

$$\begin{array}{lll lll} q_1 & - & \text{unknown displacements}, & F_1 & - & \text{prescribed forces}, \\ q_2 & - & \text{prescribed displacements}, & F_2 & - & \text{unknown forces – reactions.} \end{array}$$

Rewriting (5.102) we get two matrix equations

$$\begin{aligned} K_{11}\,q_1 \;&+\; K_{12}\,q_2 = \;F_1 \\ K_{21}\,q_1 \;&+\; K_{22}\,q_2 = \;F_2. \end{aligned} \tag{5.103}$$

The unknown displacements could found by solving the first equation of (5.103) for q_1

$$K_{11}\,q_1 = F_1 - K_{12}\,q_2,$$

while the unknown reactions F_2 we get after substituting the q_1 displacements into the second equation.

This straightforward approach is easy to implement, but becomes difficult for large systems of equations, especially in cases when special storage modes are employed – in these case the above mentioned reordering should be avoided.

A possible approach is shown here by means of a simple example. At first we make a formal changes in the system matrix, then we solve it by classical equation solvers.

□ **Example**

Starting with Eq. (5.102) for $n = 8$ and $m = 3$ we assume that the prescribed variables reside at the third, fifth and seventh rows. The are renamed as u_1, u_2, u_3. We thus could create a pointer array `IP` and write

$$\begin{array}{lcl} \texttt{IP(1)} = 3 & & u_1 \\ \texttt{IP(2)} = 5 & \Longleftrightarrow & u_2 \\ \texttt{IP(3)} = 7 & & u_3 \end{array}$$

So, $x_3 = u_1, x_5 = u_2, x_2 = u_3$, while the unknown elements of the right hand side are b_3, b_5, b_7. The initial system of equations

$$\begin{array}{lllllllll} a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & + \cdots + & a_{18}x_8 & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & a_{23}x_3 & + \cdots + & a_{28}x_8 & = & b_2 \\ \vdots \\ a_{81}x_1 & + & a_{82}x_2 & + & a_{83}x_3 & + \cdots + & a_{88}x_8 & = & b_8 \end{array}$$

could thus be rewritten into the form

$$\begin{array}{lcll} a_{11}x_1 + a_{12}x_2 + 0 + a_{14}x_4 + 0 + a_{16}x_6 + 0 + a_{18}x_8 & = & b_1 - a_{13}u_1 - a_{15}u_2 - a_{17}u_3 & (1) \\ a_{21}x_1 + a_{22}x_2 + 0 + a_{24}x_4 + 0 + a_{26}x_6 + 0 + a_{28}x_8 & = & b_2 - a_{23}u_1 - a_{25}u_2 - a_{27}u_3 & (2) \\ 0 + 0 + 1 \cdot x_3 + 0 + 0 + 0 + 0 + 0 & = & u_1 & (3) \\ a_{41}x_1 + a_{42}x_2 + 0 + a_{44}x_4 + 0 + a_{46}x_6 + 0 + a_{48}x_8 & = & b_4 - a_{413}u_1 - a_{45}u_2 - a_{47}u_3 & (4) \\ 0 + 0 + 0 + 0 + 1 \cdot x_5 + 0 + 0 + 0 & = & u_2 & (5) \\ a_{61}x_1 + a_{62}x_2 + 0 + a_{64}x_4 + 0 + a_{66}x_6 + 0 + a_{68}x_8 & = & b_6 - a_{63}u_1 - a_{65}u_2 - a_{67}u_3 & (6) \\ 0 + 0 + 0 + 0 + 0 + 0 + 1 \cdot x_7 + 0 & = & u_3 & (7) \\ a_{81}x_1 + a_{82}x_2 + 0 + a_{84}x_4 + 0 + a_{86}x_6 + 0 + a_{88}x_8 & = & b_8 - a_{83}u_1 - a_{85}u_2 - a_{87}u_3 & (8) \end{array} \tag{5.104}$$

To preserve the original structure of the system, the third, fifth and seventh equations were replaced by identities $x_3 = u_1, x_5 = u_2$ a $x_7 = u_3$.

The unknown elements b_3, b_5 a b_7 of the right hand side could evaluated from

Template 30, RHS unknowns

```
for i = 1 to m
  $b_{\texttt{IP(i)}} = \sum_{j=1}^{n} a_{\texttt{IP(i)},j}\, x_j$
next i.
```

The same in more detail is.

Template 31, RHS unknowns in detail

```
for i = 1 to m
  $b_{\mathtt{IP(i)}} = 0$
  for j = 1 to n
    $b_{\mathtt{IP(i)}} = b_{IP(i)} + a_{\mathtt{IP(i),j}}\, x_j$
  next j
next i.
```

□ **End of Example**

Summarizing the steps shown in the previous example we see that the following changes were made.

- The right hand side.

  In rows pointed to by `IP(L), L=1,` $\cdots$ `,m` the right hand side element was replaced by the u_L value.

  In the remaining rows the the right hand side elements are evaluated from

$$b_i - \sum_{L=1}^{m} a_{i,\mathtt{IP(L)}} \cdot u_{\mathtt{L}}.$$

- The matrix. The rows and columns pointed to by `IP(L), L=1,` $\cdots$ `,m` are replaced by zeros, with the exception of diagonal places where ones are inserted.

The system of equation being processed this way could be consequently solved by a suitable equation solver.

For a symmetric, positive definite matrix whose upper band is stored in a rectangular array, see the paragraph 4.1.2, the Fortran subroutine `DPREGRE` preparing the system matrix for the standard 'equation' processing is listed in the Program 26. The procedure to be called after `DPREGRE` is called `DGRE` and is listed in the Program 18.

Program 26

```
      SUBROUTINE DPREGRE(XK,B,N,NDIM,NBAND,U,IP,M,IER)
C     This is a pre-processor to DGRE subroutine.
C     It reformulates the matrix and the RHS vector for a case with
C     M prescribed displacements U, whose pointers are stored in IP array.
C     Stiffness matrix is stored in a rectangular array.
c     must be called before DGRE.
C
C     Parameters
C     XK(N,NBAND) ...... Stiffness matrix
C     B(N) ............. RHS vector
C     N ................ Number of equations
C     NDIM ............. Row dimension of XK array in main
C     NBAND ............ half-band size
C     U(M) ............. vector of prescribed displacements
C     IP(M) ............ pointer to the position of prescribed displacements
C     M ................ Number of prescribed displacements (unknowns)
C     IER .............. Error parameter
C                        IER = 0 ... O. K.
C                            =-1 ... M .GE. N
```

```
C                            =-2 ... M .EQ. 0
C     *****************************************************************
C
      DIMENSION XK(NDIM,NBAND),B(N),U(M),IP(M)
      DOUBLE PRECISION XK,B,U,T,SUM
C
      IF (M .GE. N) GO TO 9000
      IF (M .EQ. 0) GO TO 9010
C
C     Rearrange the RHS vector B
C
C     Loop over rows
      DO 100 I=1,N
C     Does the row correspond to prescribed displacements?
      DO 20 L=1,M
      IF(IP(L) .NE. I) GO TO 20
C     Yes it does
C     the L-th prescribed displacement corresponds to the I-th row
      B(I)=U(L)
      GO TO 100
20    CONTINUE
C     SUM
      SUM=0.D0
      DO 40 L=1,M
      J=IP(L)
      IF (I .GT. J) THEN
C     The under-diagonal elements are replaced
C     by their above-diagonal equivalents
       IS=J
       JS=I
      ELSE
C     we are above the diagonal
       IS=I
       JS=J
      END IF
C     out-off band elements are equal to zero
C     and are not kept in the memory
      IF (JS .GT. IS+NBAND-1) THEN
      T=0.D0
      ELSE
      T=XK(IS,JS-IS+1)
      END IF
C
      SUM=SUM+T*U(L)
40    CONTINUE
C     New RHS is
      B(I)=B(I)-SUM
100   CONTINUE
C
C     Rearrangement of the stiffness matrix
C
C     insert zeros into the corresponding rows
      DO 200 L=1,M
      DO 200 J=1,NBAND
      XK(IP(L),J)=0.D0
200   CONTINUE
C     insert zeros into the corresponding columns'
```

```
      DO 210 L=1,M
      DO 220 I=1,N
      J=IP(L)
C     under-diagonal elements are not kept in the memory
      IF (I .GT. J) GO TO 220
C     the same for the out-off band elements, do nothing
      IF(J .GT. I+NBAND-1) GO TO 220
C     compute the correct pointer
      JR=J-I+1
      XK(I,JR)=0.D0
220   CONTINUE
210   CONTINUE
C     insert ones into the diagonal places
      DO 230 L=1,M
      XK(IP(L),1)=1.D0
230   CONTINUE
      RETURN
C     ERROR RETURN
9000  IER=-1
      RETURN
9010  IER=-2
      RETURN
      END
```

End of Program 26. □

Bibliography

[1] R. Barrett, M. Berry, T.F. 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, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[2] K.J. Bathe. *Numerical methods in finite element analysis*. Prentice-Hall, Inc., Englewood Cliffs, 1976.

[3] K.J. Bathe and E.L. Wilson. *Numerical methods in finite element analysis*. Prentice-Hall, Englewood Cliffs, 1976.

[4] G. Cantin. An equation solver of very large capacity. *International Journal for Numerical Methods in Engineering*, 3:379–388, 1971.

[5] M. Fiedler. *Specialni matice a jejich pouziti v numericke matematice*. SNTL Praha, 1981.

[6] G.E. Forsythe, M.A. Malcolm, and C.B. Moler. *Computer methods for mathematical computations*. Prentice-Hall, Englewood Cliffs, 1977.

[7] G.H. Golub and Ch.F VanLoan. *Matrix computation*. John Hopkins, ISBN 978-0-8018-5414-9, New York, 1996.

[8] L.A. Hageman and D.M. Young. *Applied Iterative Methods*. Academic, in Russian, Moskva Mir, 1986.

[9] Owen D.R.J. Hinton, E. *Finite Element Programming.* Academic Press, London, 1977.

[10] R.A. Horne and C.A. Johnson. *Matrix analysis.* Cambridge University Press, 1993.

[11] B.M. Irons. A frontal solution program for finite elemenent analysis. *International Journal for Numerical Methods in Engineering*, 2:5–32, 1970.

[12] G. Marčuk. *Metody numericke matematiky.* Praha, Academia, 1987.

[13] C. Moler, J. Little, and S. Bangert. *PC-Matlab.* The MathWorks, Inc., Sherborn, MA, 1987.

[14] J. Neumann and H.H. von Goldstine. Numerical inverting of matrices of high order. *Bull. Amer. Math. Soc.*

[15] M. Okrouhlík. *Technicka mechanika II: Reseni uloh mechaniky pevnych teles metodou konecnych prvku.* CVUT Praha, 1984.

[16] M. Okrouhlík. *Personal computers in computational mechanics (in Czech).* Vydavatelstvi CVUT, Prague, 1992.

[17] M. Okrouhlík, I. Huněk, and K. Loucký. *Personal computers in technical practice, part VI (in Czech).* Dům techniky, Prague, 1990.

[18] S. Oliviera and D. Steward. *Writing scientific software.* Cambridge University Press, New York, 2006.

[19] L. Ondris. Jednoduchy a ucinny podprogram pro reseni velkych soustav linearnich rovnic v mkp. In *Teorie a praxe vypocetnch metod v mechanice kontinua*, pages 32–37.

[20] Y.C. Pao. Algorithms for direct-access gaussian solution of structural stiffness matrix equation. *International Journal for Numerical Methods in Engineering*, 12:751–764, 1978.

[21] M. Papadrakakis. *Solving Large-Scale Problems in Mechanics.* John Wiley and Sons Ltd, Chichester, Baffins Lane, 1993.

[22] B.N. Parlett. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, N. J., 1978.

[23] S.R. Phansalkar. *Matrix Iterative Methods for Structural Reanalysis, Vol. 4.* Computers & Structures, 1974.

[24] H. Prokop. Cache oblivious algorithm. Master's thesis, MIT, 1999.

[25] J.S. Przemiecki. *Theory of matrix structural analysis.* McGraw-Hill, New York, 1968.

[26] A. Ralston. *A first course in numerical analysis.* McGraw-Hill, New York, 1965.

[27] K. Rektorys. *Prehled uzite matematiky.* SNTL Praha, 1981.

[28] J.R. Rice. *Matrix Computation and mathematical software*. McCraw-Hill, Auckland, 1983.

[29] A.A. Samarskij and A.V. Gulin. *Cislennyje metody*. Moskva Nauka, 1989.

[30] G. Strang. *Linear algebra and its applications*. Academic Press, Englewood Cliffs, 1976.

[31] J. Valenta, J. Němec, and E. Ulrych. *Novodobe metody vypoctu tuhosti a pevnosti ve strojirenstvi*. SNTL Praha, 1975.

[32] R.S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, New Jersey, 1965.

[33] E. Volkov. *Cislennyje metody*. Moskva Nauka, 1987.

[34] J.H. Wilkinson. Modern error analysis. *SIAM Review*, 13:751–764, 1971.

Chapter 6

Implementation remarks to nonlinear tasks

This part was written and is maintained by M. Okrouhlík. More details about the author can be found in the Chapter 16.

In this chapter we will concentrate on numerical solution of systems of nonlinear real algebraic equations. A few iterative approaches will be described and explained using simple examples.

6.1 Preliminaries, terminology and notation

We will devote our attention to two types of problems. They will be confined to n-dimensional Euclidian space.

- Solution of n nonlinear equations with n unknowns described by a vector function of a vector variable. The equations have the form $\mathbf{g}(\mathbf{x}) = \mathbf{0}$, or $g_i(x_k) = 0$, or

$$\begin{aligned} g_1(x_1, x_2, \ldots, x_n) &= 0, \\ g_2(x_1, x_2, \ldots, x_n) &= 0, \\ &\vdots \\ g_n(x_1, x_2, \ldots, x_n) &= 0. \end{aligned} \tag{6.1}$$

  We are looking for such a value of $\mathbf{x}$ that satisfies Eq. (6.1). The found solution, sometimes called a root, is denoted $\mathbf{x}^*$.

- Minimization of a scalar function of a vector variable $F(\mathbf{x})$, i.e.

$$F(x_1, x_2, \ldots, x_n). \tag{6.2}$$

  We are looking for such a value of $\mathbf{x}$ for which the function (6.2) reaches its minimum. Sometimes the minimum value is required as well, i.e. $F_{\min} = F(\mathbf{x}^*)$.

If the function $F(\mathbf{x})$, tohether with their first derivatives, is continuous, then its gradient $\mathbf{g}(\mathbf{x})$ could be expressed in the form

$$\mathbf{g}(\mathbf{x}) = \boldsymbol{\nabla} F(\mathbf{x}) = \left\{ \begin{array}{c} \dfrac{\partial F}{\partial x_1} \\ \dfrac{\partial F}{\partial x_2} \\ \vdots \\ \dfrac{\partial F}{\partial x_n}. \end{array} \right\}. \tag{6.3}$$

Under these conditions the vector $\mathbf{x}^*$, satisfying $\mathbf{g}(\mathbf{x}^*) = \mathbf{0}$ – the solution of (6.1), 'points' to the location where the function $F(\mathbf{x})$ has its minimum – in other words $F_{\min} = F(\mathbf{x}^*)$. Proof can be found in [4].

The first derivatives of the $\mathbf{g}(\mathbf{x})$ function can be assembled into the matrix which goes under the name of Jacobi matrix. Generally, it is defined by the relation

$$J_{ij}(x_1, x_2, \ldots, x_n) = \frac{\partial g_i(x_1, x_2, \ldots, x_n)}{\partial x_j}, \tag{6.4}$$

whose matrix representation is

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{g}(\mathbf{x})}{\partial \mathbf{x}^{\mathrm{T}}} = \begin{bmatrix} \dfrac{\partial g_1}{\partial x_1} & \dfrac{\partial g_1}{\partial x_2} & \cdots & \dfrac{\partial g_1}{\partial x_n} \\ \dfrac{\partial g_2}{\partial x_1} & \dfrac{\partial g_2}{\partial x_2} & \cdots & \dfrac{\partial g_2}{\partial x_n} \\ & & \ddots & \\ \dfrac{\partial g_n}{\partial x_1} & \dfrac{\partial g_n}{\partial x_2} & \cdots & \dfrac{\partial g_n}{\partial x_n} \end{bmatrix}. \tag{6.5}$$

Notes

- To save the printing space in relations above, we have used F instead of $F(x_1, x_2, \cdots, x_n)$ and g_i instead of $g_i(x_1,\, x_2,\, \ldots,\, x_n)$. In the text to follow we will similarly write $\mathbf{g} = \mathbf{g}(\mathbf{x})$, $\mathbf{J} = \mathbf{J}(\mathbf{x})$, etc.

- The transposition of the Jacobi matrix is denoted

$$\mathbf{J}^{\mathrm{T}} = \frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \begin{bmatrix} \dfrac{\partial g_1}{\partial x_1} & \dfrac{\partial g_1}{\partial x_1} & \cdots & \dfrac{\partial g_1}{\partial x_1} \\ \dfrac{\partial g_2}{\partial x_2} & \dfrac{\partial g_2}{\partial x_2} & \cdots & \dfrac{\partial g_2}{\partial x_2} \\ & & \ddots & \\ \dfrac{\partial g_n}{\partial x_n} & \dfrac{\partial g_n}{\partial x_n} & \cdots & \dfrac{\partial g_n}{\partial x_n} \end{bmatrix}. \tag{6.6}$$

- The determinant of Jacobi matrix is called Jacobian.

- The Hess matrix contains the second derivatives of the function $\mathbf{g}(\mathbf{x})$, i.e.

$$H_{ij} = \frac{\partial^2 g_i}{\partial x_i \, \partial x_j}. \tag{6.7}$$

End of notes □

Example 1.
A scalar function of a vector variable

$$F(\mathbf{x}) = x_1^4 + x_1^2 x_2^2 + 2x_2^4 - 0.904x_1 - 6.12x_2, \tag{6.8}$$

depicted in Fig. 6.1, could be used as a simple benchmark for different iteration schemes. The function has a minimum in (0.4, 0.9). The value of the minimum is $F_{\min} = -4.4022$.

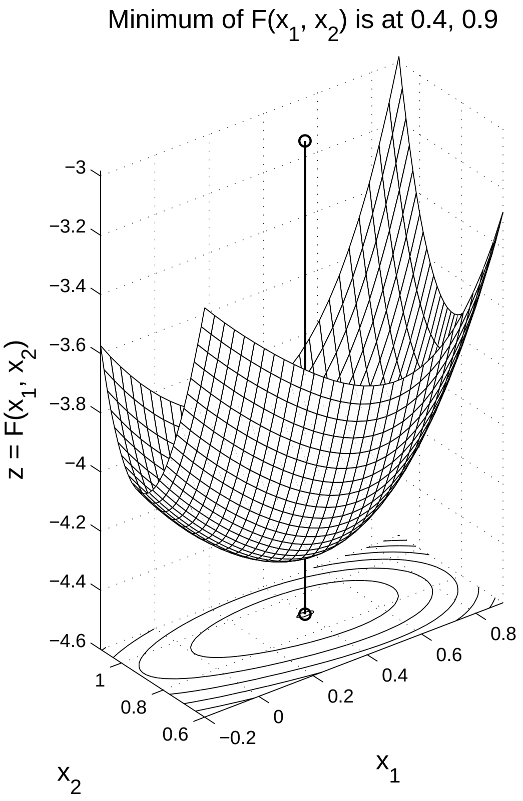


Figure 6.1: Minimum of $z = F(x_1, x_2)$ and its location

According to(6.8) the gradient of $F(\mathbf{x})$ is

$$\boldsymbol{\nabla} F = \left\{ \begin{array}{c} \dfrac{\partial F}{\partial x_1} \\ \dfrac{\partial F}{\partial x_2} \end{array} \right\} = \mathbf{g} = \left\{ \begin{array}{c} g_1 \\ g_2 \end{array} \right\} = \left\{ \begin{array}{c} 4x_1^3 + 2x_1 x_2^2 - 0.904 \\ 2x_1^2 x_2 + 8x_2^3 - 6.12 \end{array} \right\}. \tag{6.9}$$

A note for finite element readers

A nonlinear function $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ could be alternatively written in the form

$$\mathbf{K}(\mathbf{x})\,\mathbf{x} = \mathbf{b}. \tag{6.10}$$

In this case we have

$$\mathbf{K} = \begin{bmatrix} 4x_1^2 & 2x_1\,x_2 \\ 2x_1\,x_2 & 8x_2^2 \end{bmatrix}, \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{Bmatrix} 0.904 \\ 6.12 \end{Bmatrix}. \tag{6.11}$$

In the deformation variant of the finite element method the Eq. (6.10) represents the formulation of a static equilibrium which takes the constitutive relations into account as well. The matrix $\mathbf{K}(\mathbf{x})$ then represents the secant stiffness matrix – generally, it is a function of the unknown vector of displacements, i.e. of $\mathbf{x}$. The $\mathbf{b}$ vector contains the components of loading forces. If the function $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ is linear, then the matrix $\mathbf{K}$ is constant and the equation $\mathbf{K}\,\mathbf{x} = \mathbf{b}$ leads a system of linear algebraic equations that could be solved by finite methods.

End of note □

In this example the problem of finding the minimum of $F(\mathbf{x})$, defined by (6.8), is identical with the solution of $\boldsymbol{\nabla}F(\mathbf{x}) = 0$ – or in another notation $\mathbf{g}(\mathbf{x}) = \mathbf{0}$, where $\mathbf{g}(\mathbf{x})$ is given by (6.9). Obviously, the extremum appears in a location where the function derivative equals zero. The graphical approach to the solution of this task is depicted in

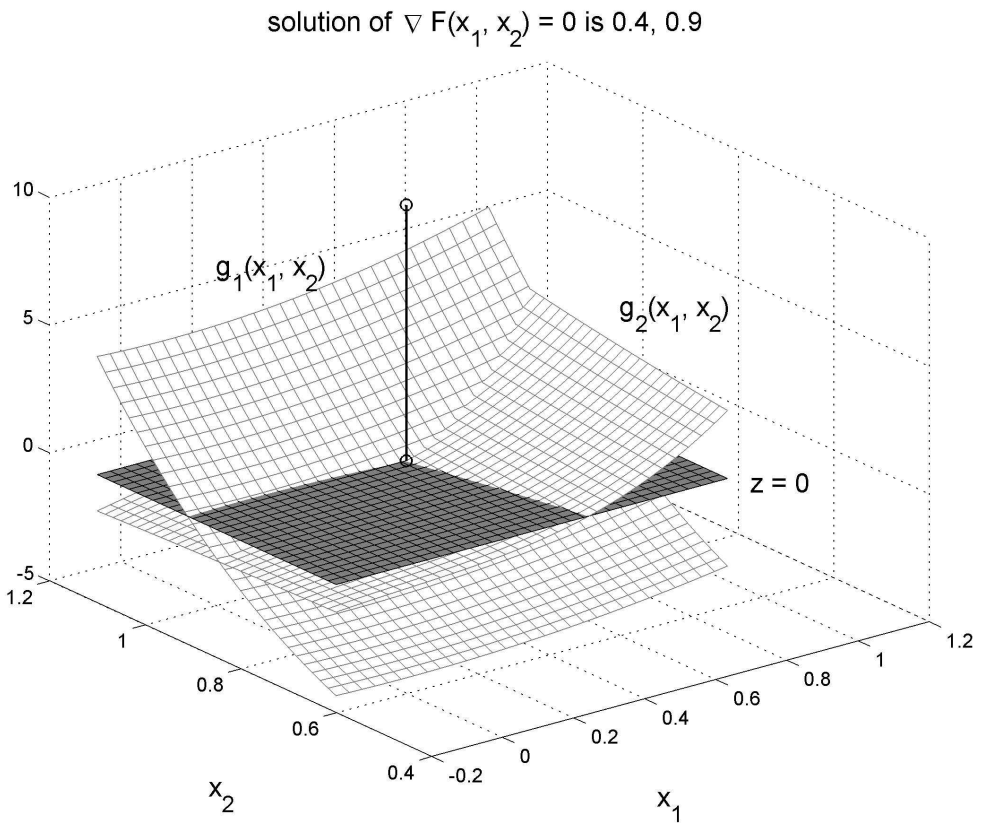


Figure 6.2: Graphical solution of $\boldsymbol{\nabla}F(x_1, x_2) = \mathbf{0}$

Fig. 6.2. Notice the intersection of 'gradient' surfaces $g_1(x_1, x_2)$ and $g_2(x_1, x_2)$ with the horizontal plane $z = 0$. The vertical line points to the sought after solution, i.e. to (0.4, 0.9).

In matrix notation the Jacobi matrix for $\mathbf{g}(\mathbf{x})$, see (6.9), is

$$\mathbf{J} = \mathbf{g}'(\mathbf{x}) = \frac{\partial \mathbf{g}(\mathbf{x})}{\partial \mathbf{x}^{\mathrm{T}}} = \begin{bmatrix} \dfrac{\partial g_1}{\partial x_1} & \dfrac{\partial g_1}{\partial x_2} \\ \dfrac{\partial g_2}{\partial x_1} & \dfrac{\partial g_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 12x_1^2 + 2x_2^2 & 4x_1\,x_2 \\ 4x_1\,x_2 & 2x_1^2 + 24x_2^2 \end{bmatrix}. \tag{6.12}$$

It is the Jacobi matrix which plays the role of the secant matrix in finite element method. Compare with (6.10). Numerically, the solution can be found by iterating the increments from the equation

$$\mathbf{J}(\mathbf{x})\,\Delta\mathbf{x} = \Delta\mathbf{b}. \tag{6.13}$$

The Hess matrix contains the second derivatives of the function $\mathbf{g}(\mathbf{x})$, see (6.9), i.e.

$$H_{ij} = \frac{\partial^2 g_i}{\partial x_i\,\partial x_j}. \tag{6.14}$$

In our example we get

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 g_1}{\partial x_1\,\partial x_1} & \dfrac{\partial^2 g_1}{\partial x_2\,\partial x_1} \\ \dfrac{\partial^2 g_2}{\partial x_1\,\partial x_2} & \dfrac{\partial^2 g_2}{\partial x_2\,\partial x_2} \end{bmatrix} = \begin{bmatrix} 24x_1 & 4x_2 \\ 4x_1 & 48x_2 \end{bmatrix}. \tag{6.15}$$

End of example 1. □

6.1.1 Newton-Raphson method

The method often goes under alternative names as the *Newton* or *Newton-Gauss* method and could be used for the solution of a system of nonlinear algebraic equations. The method is based on the Taylor series expansion of a vector function of a vector variable $\mathbf{g}(\mathbf{x})$, at the location $\mathbf{x} + \Delta\mathbf{x}$, in the n-dimensional space and can be written in the form

$$\mathbf{g}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{g}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\,\Delta\mathbf{x} + \text{higher order terms}, \tag{6.16}$$

where

$$\mathbf{g}(\mathbf{x} + \Delta\mathbf{x}) = \begin{Bmatrix} g_1(x_1 + \Delta x_1,\, x_2 + \Delta x_2,\, \ldots,\, x_n + \Delta x_n) \\ g_2(x_1 + \Delta x_1,\, x_2 + \Delta x_2,\, \ldots,\, x_n + \Delta x_n) \\ \vdots \\ g_n(x_1 + \Delta x_1,\, x_2 + \Delta x_2,\, \ldots,\, x_n + \Delta x_n) \end{Bmatrix}.$$

The vector of increments is $\Delta\mathbf{x} = \{\Delta x_1 \Delta x_2 \cdots \Delta x_n\}^{\mathrm{T}}$, the function $\mathbf{g}(\mathbf{x})$ is defined by Eq. (6.1) while the Jacobi matrix $\mathbf{J}(\mathbf{x})$ by Eq. (6.5).

Neglecting the higher order terms and realizing that we are looking for such a value of $\Delta\mathbf{x}$ for which the left-hand side of Eq. (6.16) is equal to zero, we get the system of linear algebraic equations for unknown increments $\Delta\mathbf{x}$ in the form

$$\mathbf{J}(\mathbf{x})\,\Delta\mathbf{x} = -\mathbf{g}(\mathbf{x}). \tag{6.17}$$

The increment $\Delta\mathbf{x}$, obtained by solving Eq. (6.17), is used for evaluating the next approximation of the solution

$$\mathbf{x}^{(1)} = \mathbf{x} + \Delta\mathbf{x}, \tag{6.18}$$

which, however, has to be be improved by a subsequent iteration process. We might proceed as follows

$$\begin{aligned} &\mathbf{J}(\mathbf{x}^{(k)})\,\Delta\mathbf{x}^{(k)} = -\mathbf{g}(\mathbf{x}^{(k)}), \quad \Rightarrow \Delta\mathbf{x}^{(k)}, \\ &\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}. \end{aligned} \tag{6.19}$$

The upper right-hand side index denotes the iteration counter for which $k = 0, 1, 2, \ldots$
.

Let's assume that i) the function $\mathbf{g}(\mathbf{x})$ and its derivatives, defined in an open convex region D, are continuous, and ii) there is such a $\mathbf{x}^*$ for which $\mathbf{g}(\mathbf{x}^*) = \mathbf{0}$ in this region and iii) the matrix $\mathbf{J}(\mathbf{x}^*)$ is regular.

If the function $\mathbf{g}(\mathbf{x})$ in the vicinity of $(\mathbf{x}^*)$ has at least two derivatives, then the so called Lipschitz condition having the form

$$\|\mathbf{J}(\mathbf{x}) - \mathbf{J}(\mathbf{x}^*)\| \leq \gamma \|(\mathbf{x}) - (\mathbf{x}^*)\|, \tag{6.20}$$

is satisfied for a suitably chosen value of γ constant.

If the above conditions are satisfied then the iterative process, described by (6.19) for $k = 0, 1, 2, \ldots,$ converges to the solution $\mathbf{x}^*$

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq \kappa^{(k)} \|\mathbf{x}^{(k)} - \mathbf{x}^*\|^2, \tag{6.21}$$

in such a way that the sequence $\kappa^{(k)} \to 0$. Proof can be found in [3].

Theoretically, the Newton-Raphson method converges quadratically if the initial approximation starts from the D region.

Practically, one cannot guarantee the satisfaction of the above assumptions and might thus experience troubles if the initial approximation is far from the expected solution. Also, more than one solution of $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ might exist in the region.

In each iteration step one has to evaluate the function values as well as the values of its first derivatives, i.e. $\mathbf{g}^{(k)} = \mathbf{g}(\mathbf{x}^{(k)})$ and $\mathbf{J}^{(k)} = \mathbf{J}(\mathbf{x}^{(k)})$, since they are a functions of the current position $\mathbf{x}^{(k)}$, and to solve the system of algebraic equations. On of possible approaches is sketched in the Template 32.

Template 32, Newton-Raphson method

```
Initial approximation
   $\mathbf{x}^{(0)}$
Iterate
   for k = 0, 1, 2, ...
     $\mathbf{g} = \mathbf{g}(\mathbf{x}^{(k)})$                          function values
     $\mathbf{J} = \mathbf{J}(\mathbf{x}^{(k)})$                          values of derivatives
     $\mathbf{J}\ \Delta\mathbf{x} = -\mathbf{g}; \quad \Rightarrow \Delta\mathbf{x}$              solve the system of equations
     $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}$                 next iteration
     test of convergence, continue if not satisfied
   end
```

The vaguely expressed metastatement `test of convergence, continue if not satisfied` requires to satisfy the following conditions.

- The norm of the increment has to be 'smaller' than the required threshold (tolerance) value. It is recommended to work with relative tolerances,
- the root value of the function has to be 'small', i.e. smaller than the required function threshold. Here we have to work with absolute values which are to be compared with 'pure' zeros,
- the total number of iteration should not exceed a 'reasonable' limit.

Due to the vectorial nature of variables their magnitudes, within the iterative process, could only be measured in 'suitably defined' norms and compared to 'reasonable' values of tolerances. The abundance of quote characters in the text above indicate that the choice of the tolerance values is not easy – they are always problem dependent. Besides the generally valid hints there is no universally valid approach which might be advocated.

One of a possible approaches how to finish the iterative process is presented in the Template 33. The value of the logical variable `satisfied` might be either `true` or `false`. The logical operators `and` and `not` are used in their usual meanings, i.e. conjunction and negation.

Template 33, Finishing the iteration process

```
$\varepsilon_x$                 required relative tolerance for the solution (root)
$\varepsilon_g$                 required absolute tolerance for the function value at root
kmax                admissible number of iterations
k = 0               counter
satisfied = false   a priori not satisfied at the beginning
while not(satisfied)                                          iterate, while not satisfied
  k = k + 1
  compute new values of $\mathbf{x}, \Delta\mathbf{x}$ and $\mathbf{g}(\mathbf{x})$
  satisfied = $\frac{\| \Delta\mathbf{x} \|}{\| \mathbf{x} \|} < \varepsilon_x$ and $\| \mathbf{g}(\mathbf{x}) \| < \varepsilon_g$ and k <= kmax        are you satisfied?
end
```

A possible procedure for finding roots of the system of nonlinear equations $\mathbf{g}(\mathbf{x}) = \mathbf{0}$, where $\mathbf{g} = \mathbf{g}(\mathbf{x})$ is given by (6.9), is presented in the Program 27. The function procedure `Newton_Raphson.m` is quite general. Using alternative user created function procedures calling actual functions, i.e. $\mathbf{g}(\mathbf{x})$ and their derivatives $\mathbf{g}'(\mathbf{x})$ – namely `function_g.m` and `jacobi_m` – one can use the program for solving other tasks of this type as well. The rest of the program consists of 'administrative' ballast and the graphical presentation.

Program 27

```
% NR_c2
% Solution of the system of nonlinear equations g(x) = 0.
%
% Required functions
% Newton_Raphson.m, which requires function_F.m, function_g.m, jacobi_g.m,
%
% The functional values of g(x) are generated by function_g.m
% The values of its derivatives J(x) are generated by jacobi_g.m
%
% In this case the function g(x) is a gradient of function F(x).
% At first compute values of F(x) for plotting its contours.
% The values of F(x) are generated by function function_F.m
%
% 1. Preliminaries
clear, clf, format short e, format compact
xmin = -0.2; xmax = 0.9;    % x-boundaries of plot area
ymin = 0.6; ymax = 1.1;     % y-boundaries of plot area
ndivx = 70; ndivy = 70;     % number of plot increments
dx = (xmax - xmin)/ndivx;   % x-increment
dy = (ymax - ymin)/ndivy;   % y-increment
xrange = xmin:dx:xmax;      % plot range for x
yrange = ymin:dy:ymax;      % plot range for y
ixmax = length(xrange);     % x-counter
jymax = length(yrange);     % y-counter
ff = zeros(ixmax,jymax);    % working array, its dimensions
% Evaluate values of F(x1, x2) and store them for future use
for i = 1:ixmax
    xx = xmin + (i - 1)*dx;
    for j = 1:jymax
        yy = ymin + (j - 1)*dy;
        xv = [xx yy];
        ff(i,j) = function_F(xv);
    end
end
% input data
x = [0 1]'; xini = x;       % initial approximation, store it for future
epx = 1e-3;                 % tolerance parameter for solution
epg = 1e-3;                 % tolerance parameter for zero value
itmax = 30;                 % permissible number of iterations

% 2. The solution itself
 [x,ier,n_step,xhist,residuum] = Newton_Raphson(xini,itmax,epx,epg);

% 3. Presentation of results
if ier == 0,                    % print and plot results if OK
    xplot = xhist(1,:); yplot = xhist(2,:);
    res = [ier n_step xplot(n_step) yplot(n_step)];
    disp('ier, no. of steps, x1, x2:'), disp(res)
    disp('residual value of g at x1, x2:'), disp(residuum)
    lab = ...
['Newton-Raphson, no. of iterations = ' int2str(n_step), ', epx =
' num2str(epx), ', epg = ' num2str(epg)];
%
figure(1)
 vv = [-3.1 -3.5 -3.8 -4 -4.2 -4.3 -4.391];
 [c, h] = contour(xrange, yrange, ff', vv); axis('equal'); % axis('square')
 clabel(c,h); title(lab, 'fontsize', 14); colormap([0 0 0])
 xlabel('x_1', 'fontsize', 14); ylabel('x_2', 'fontsize', 14)
 hold on
 plot(0.4, 0.9,'or','markersize', 10, 'linewidth', 2)
 hold on
 plot(xini(1), xini(2),'or','markersize', 10, 'linewidth', 2)
 axis([-0.2 0.9 0.6 1.1])
 hold on
 plot([xini(1) xplot],[xini(2) yplot],'o-k');
 hold off
 print -deps -r300 NR_plot
 format long e
else            % print error message
    disp('Prescribed tolerace not reached ')
end
% end of NR_c2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 function [x,ier,n_step,xhist,residuum] = Newton_Raphson(xini,itmax,epx,epg)
% Newton-Raphson method for the solution of
% the system of nonlinear equations g(x) = 0.
% Two problem dependent functions are required
% 1. function_g.m       generates values of function g(x)
% 2. jacobi_g.m         jacobi matrix for function g(x)
% input values
% xini          initial approximation
% epx;          relative tolerance parameter for solution
% epg           absolute tolerance parameter for zero value of the function
% itmax         permissible number of iterations
% output values
% x             solution
% ier           error parameter, if OK ier = 0; if not OK ier = -1
% n_step        number of iteration steps required
% xhist         history of iterations
% residuum      functional residuum at solution
%
satisfied = false;          % a priori unsatisfied
it = 0;                     % iteration counter
ier = 0;                    % error indicator, 0 if OK, -1 if wrong
x = xini;
% NR itself
while not(satisfied)
    it = it + 1;            % increase iteration counter by one
    if(it > itmax), ier = -1; break, end    % error if too many iterations
    g = function_g(x);      % evaluate function for the current step
    j = jacobi_g(x);        % evaluate Jacobi matrix for the current step
    dx = j\(-g');           % solve for increment dx by Gauss elimination
    x_new = x + dx;         % new iteration
    rdif1(it) = norm(dx)/norm(x_new);   % relative measure for root position
    rdif2(it) = norm(g);    % absolute measure for function value at root
    satisfied = (rdif1(it) <= epx & rdif2(it) <= epg);  % convergence test
    x = x_new;              % update for the next iteration
    xhist(:,it) = x';       % keep history of iteration results
end
residuum = function_g(x);
n_step = it;
% end of Newton_Raphson
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function F = function_F(x)
% the scalar function of a vector variable
 F = x(1)^4 + x(1)^2*x(2)^2 + 2*x(2)^4 - 0.904*x(1) - 6.12*x(2);
% end of function_F
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function g = function_g(x)
% generates values of vector function g(x1,x2) of vector variable
 g(1) = 4*x(1)^3 + 2*x(1)*x(2)^2 - 0.904;
 g(2) = 2*x(1)^2*x(2) + 8*x(2)^3 - 6.12;
% end of function_g
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function j = jacobi_g(x)
% jacobi matrix for function g given by
% g(1) = 4*x(1)^3 + 2*x(1)*x(2)^2 - 0.904;
% g(2) = 2*x(1)^2*x(2) + 8*x(2)^3 - 6.12;
% see function_g.m
 j(1,1) = 12*x(1)^2 + 2*x(2)^2;
 j(1,2) = 4*x(1)*x(2);
 j(2,1) = 4*x(1)*x(2);
 j(2,2) = 2*x(1)^2 + 24*x(2)^2;
% end of jacobi_g
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> NR_c2 output
 ier, no. of steps, x1, x2:
           0   4        4.0000e-001  9.0000e-001
residual value of g at x1, x2:
                        3.5060e-009  1.3158e-009
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

End of Program 27. □

The results for the initial approximation $\{0\ 1\}^{\mathrm{T}}$ and for tolerances $\varepsilon_x = \varepsilon_g = 0.001$ are presented at the end of listing of the Program 27. Parameters `ier, no of steps` denote the error parameter and the number of iteration steps required for obtaining the required accuracy. The values of x1 and x2 point to the root location, i.e. to the location of minimum. The remaining two values indicate the residual values of the **g** function at the minimum when the iteration is stopped. A 'search' trip from the initial approximation to the root is graphically depicted in Fig. 6.3 together with contour lines of the F function.

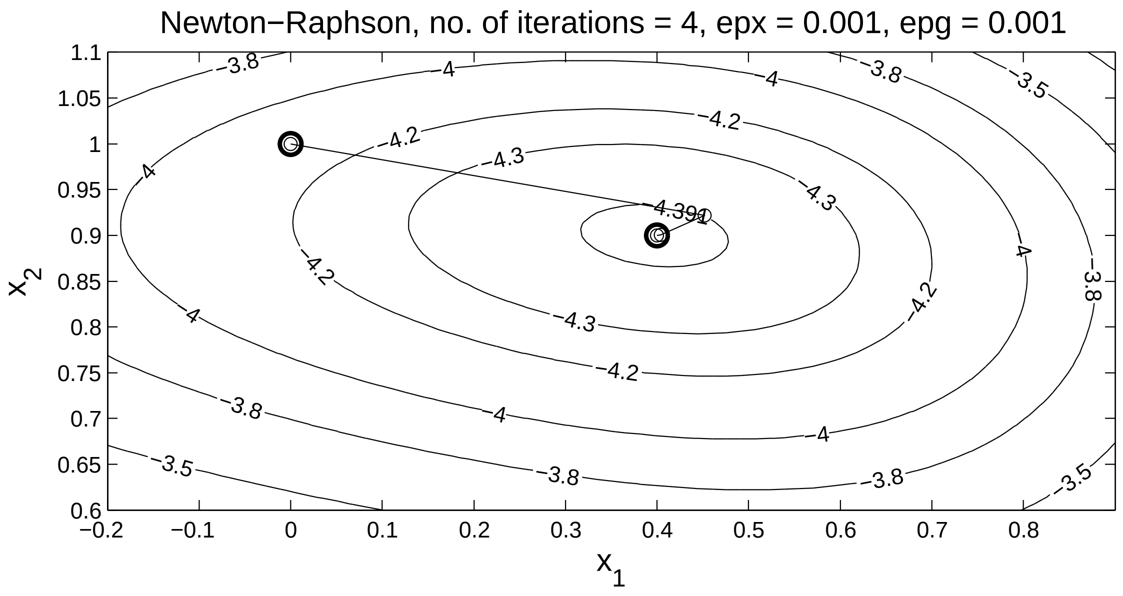


Figure 6.3: Newton-Raphson

6.1.2 Modified Newton-Raphson method

If the evaluation of the Jacobi matrix is too time demanding, then one can compute it only once, before the iteration process starts, using the derivative of the **g** function at the location of the initial approximation. Such an approach is known as the *modified Newton-Raphsom method*. For the simplification of the iterative process we are punished by a slower convergence rate. The iterative process, initially given by relations (6.19), is changed as follows

$$\begin{aligned}&\mathbf{J}(\mathbf{x}^{(0)})\,\Delta\mathbf{x}^{(k)} = -\mathbf{g}(\mathbf{x}^{(k)}), \quad \Rightarrow \Delta\mathbf{x}^{(k)},\\ &\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}.\end{aligned} \tag{6.22}$$

Since the Jacobi matrix $\mathbf{J}^{(0)} = \mathbf{J}(\mathbf{x}^{(0)})$ does not change during the iteration process, the solution of the system of linear algebraic equations can be profitably divided into two parts. The matrix triangulation (factorization) is executed only once, before the iteration process, and within the iteration loop one is providing the reduction of the right-hand side and the back substitution as follows.

$$\begin{aligned}
&\text{Instead of the 'full' elimination of} \quad \mathbf{J}\,\Delta\mathbf{x} = \mathbf{g} \\
&\text{we decompose the matrix} \quad \mathbf{J} \rightarrow \mathbf{L}\,\mathbf{U}, \quad \text{only once, before starting the iteration process} \\
&\Rightarrow \mathbf{L}\,\mathbf{U}\,\Delta\mathbf{x} = \mathbf{g} \\
&\Rightarrow \mathbf{U}\,\Delta\mathbf{x} = \mathbf{L}^{-1}\,\mathbf{g} \\
&\text{and in each iteration step we provide only} \\
&\text{the reduction of the right-hand side} \quad \mathbf{c} = \mathbf{L}^{-1}\,\mathbf{g} \\
&\text{and the back substitution} \quad \Delta\mathbf{x} = \mathbf{U}^{-1}\,\mathbf{c}.
\end{aligned} \tag{6.23}$$

Of course, the inversion of **U** and of **L** matrices need not be explicitly carried out. It should be noted however, that the suggested procedure is not advantageous in Matlab, where the Gauss elimination is simply secured by the backslash operator. Nevertheless, it can bring a substantial time savings in a programming code where the matrix triangularization and the back substitution processes are treated independently. For more details see [2]. A possible way how to algorithmize the modified Newton-Raphson is in the Program 28. Only `modified_Newton_Raphson.m` function is presented.

Program 28

```
 function [x,ier,n_step,xhist,residuum] = modified_Newton_Raphson(xini,itmax,epx,epg)
% Modified Newton-Raphson method
%
% the same comment as in Newton_Raphson
%
satisfied = false;          % a priori unsatisfied
it = 0;                     % iteration counter
ier = 0;                    % error indicator, 0 if OK, -1 if wrong
x = xini;
j = jacobi_g(x);            % use initial approximation and evaluate Jacobi matrix
[L,U] = lu(j);              % as well as its LU decomposition
% modified NR itself
while not(satisfied)
    it = it + 1;            % increase iteration counter by one
    if(it > itmax), ier = -1; break, end    % error if too many iterations
    g = function_g(x);      % evaluate function for the current step
    c = -L\g';              % factorization of the right-hand side
    dx = U\c;               % back substitution
    x_new = x + dx;         % new iteration
    rdif1(it) = norm(dx)/norm(x_new);   % relative measure for root position
    rdif2(it) = norm(g);    % absolute measure for function value at root
    satisfied = (rdif1(it) <= epx & rdif2(it) <= epg);  % convergence test
    x = x_new;              % update for the next iteration
    xhist(:,it) = x';       % keep history of iteration results
end
 residuum = function_g(x);
 n_step = it;
% end of modified Newton_Raphson
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> mNR_c2 output
 ier, no. of steps, x1, x2:
          0  30       3.9984e-001  8.9999e-001
 residual value of g at x1, x2:
                      -5.8603e-004 -4.2605e-004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

End of Program 28. □

The results of the iteration process obtained by the modified Newton-Raphson method are listed at the end of the Program 28. Comparing the results with those previously obtained by the classical Newton-Raphson method we observe that for reaching the same precision we need 30 instead of 4 iteration steps. One should note, however, that the number of iteration steps themselves is not generally directly proportional to total computational demands of the solved task.

The convergence of the modified Newton-Raphson method might be accelerated by the 'correction' of the Jacobi matrix after a few iteration steps.

6.1.3 Method of the steepest gradient

If the vector function $\mathbf{g}(\mathbf{x})$ is a gradient of the scalal function $F(\mathbf{x})$[1], then the solution of the system of nonlinear equations $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ can alternatively be treated as the search for the minimum of the function $F(\mathbf{x})$. In the Example 1 the function $\mathbf{g}(\mathbf{x})$ was created from the function $F(\mathbf{x})$ by means of (6.3). For a solution of the system of equations $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ a suitable scalar function could be generated rather arbitrarily, for example by the relation

$$FF(\mathbf{x}) = (\mathbf{g}(\mathbf{x}))^{\mathrm{T}}\ \mathbf{g}(\mathbf{x}). \tag{6.24}$$

For $\mathbf{g}(\mathbf{x})$, defined by Eq. (6.9), we get the function $FF(\mathbf{x})$ in the form

$$\begin{aligned} FF(\mathbf{x}) = 16\,{x_1}^6 + 20\,{x_1}^4{x_2}^2 - \frac{904}{125}\,{x_1}^3 + 36\,{x_1}^2{x_2}^4 - \\ \frac{452}{125}\,x_1\,{x_2}^2 + \frac{597994}{15625} - \frac{612}{25}\,{x_1}^2x_2 + 64\,{x_2}^6 - \frac{2448}{25}\,{x_2}^3. \end{aligned} \tag{6.25}$$

The symbolical form of the dot product, defined by Eq. (6.24), – including its LaTeX source expression, generating the appearance text of Eq. (6.25) – was 'miraculously' obtained by a few statements listed in the Program 29. It should be noted that the Matlab `.'` operator stands for the transposition.

Program 29

```
% sym_F
clear syms x1 x2 g g1 g2
g1 = 4*x1^3 + 2*x1*x2^2 - 0.904;
g2 = 2*x1^2*x2 + 8*x2^3 - 6.12; g = [g1 ; g2]; FF = g.'*g;
r = expand(FF); latex(r)
```

End of Program 29. □

The functions $F(\mathbf{x})$ and $FF(\mathbf{x})$ have a different minimum value, nevertheless they have it in the same location. The minimum value of the function $FF(\mathbf{x})$ and its location are depicted in Fig. 6.4. Similarly that for the function $F(\mathbf{x})$ in Fig. 6.1. From now on we will proceed with the function $F(\mathbf{x})$, defined by Eq. (6.8).

Generally, the $F(\mathbf{x})$ is a function of n variables. The function $z = F(\mathbf{x})$ is defined in the R^{n+1} space. Differentiating the function $z = F(\mathbf{x})$ we get

[1]Expressed otherwise – if the scalar function has a gradient.

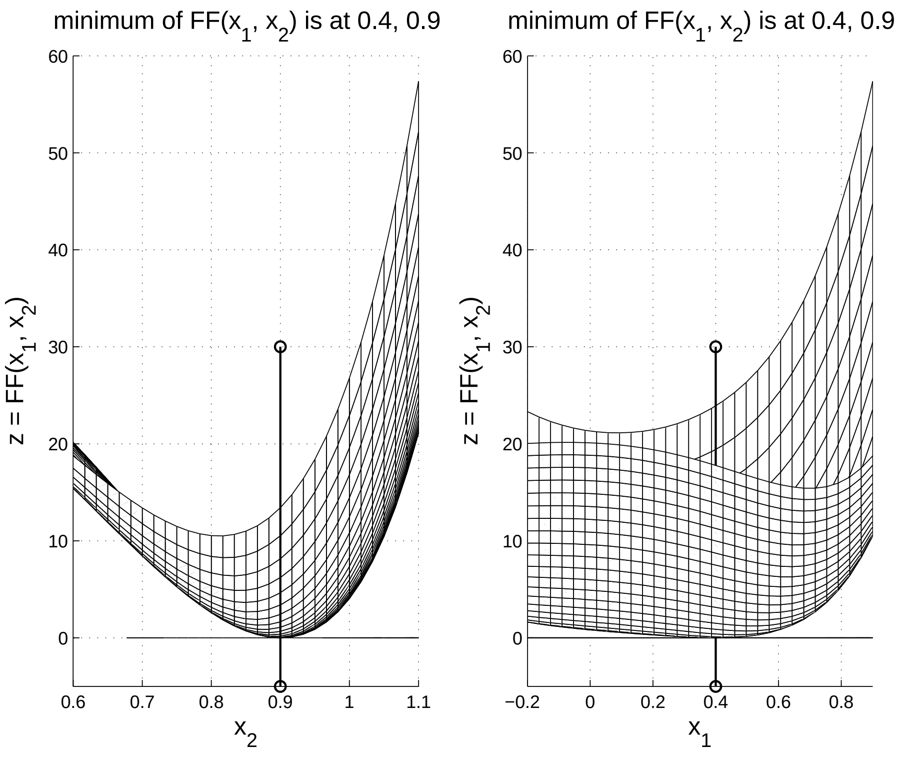


Figure 6.4: Minimum of $z = FF(x_1, x_2)$ and its location

$$\frac{\partial F}{\partial x_1}\mathrm{d}x_1 + \frac{\partial F}{\partial x_2}\mathrm{d}x_2 + \cdots + \frac{\partial F}{\partial x_n}\mathrm{d}x_n - \mathrm{d}z = 0, \tag{6.26}$$

which can be written in the form

$$\mathbf{n}^{\mathrm{T}}\,\mathrm{d}\mathbf{t} = 0, \tag{6.27}$$

where

$$\mathbf{n} = \left\{ \underbrace{\frac{\partial F}{\partial x_1}\;\frac{\partial F}{\partial x_2}\;\cdots\;\frac{\partial F}{\partial x_n}}_{\mathbf{g}^{\mathrm{T}}}\;|\,-1 \right\}^{\mathrm{T}} \tag{6.28}$$

and

$$\mathrm{d}\mathbf{t} = \{\mathrm{d}x_1\;\mathrm{d}x_2 \cdots \mathrm{d}x_n\;\mathrm{d}z\}^{\mathrm{T}}. \tag{6.29}$$

Notice that both vectors have $n+1$ elements. The $\mathbf{n}$ vector represents a surface normal to $F(\mathbf{x})$. Its components, excluding the last one, are identical with components the gradient components $\mathbf{g}(\mathbf{x}) = \boldsymbol{\nabla} F(\mathbf{x})$, representing thus the projection of the normal to the R^n space. And the contour lines – they depict the projection of the function $z = F(\mathbf{x})$, obtained for different values of $z = \text{const}$. The projection of the normal is perpendicular to a particular contour line and represents the direction of the steepest slope of the function F. For the function $z = F(\mathbf{x})$ of two independent variables the situation is depicted in Fig. 6.5.

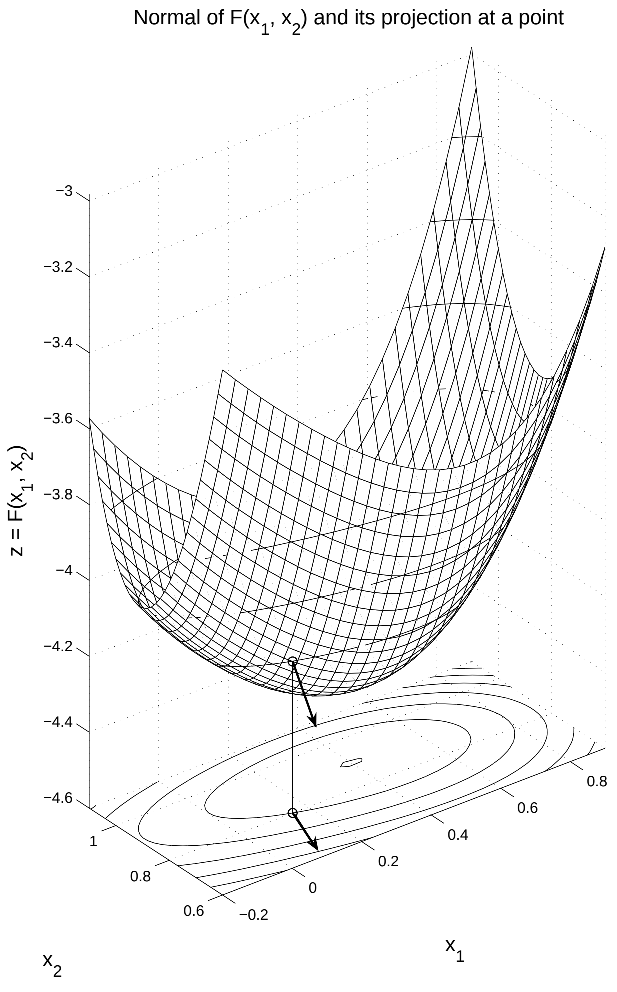


Figure 6.5: Normal of $z = F(x_1, x_2)$ and its projection

The normal, as well as its projection, 'point' out of the surface, thus the the steepest gradient method takes the 'negative' gradient as the approximation of a direction pointing to the location of minimum.

The iterative process can ce conceived as follows

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\,\Delta\mathbf{x}^{(k)} = \mathbf{x}^{(k)} - \alpha\,\boldsymbol{\nabla} F(\mathbf{x}^{(k)}) = \mathbf{x}^{(k)} - \alpha\,\mathbf{g}(\mathbf{x}^{(\mathbf{k})}), \tag{6.30}$$

where the scalar α constant is still to be determined. One possible way if its determination is to look for a local minimum in the gradient direction. This must be done in each iteration step. The function values of F in the gradient direction, for the k-the iteration step, can be written in the form

$$z(\alpha) = F(\mathbf{x}^{(k)} + \alpha\,\Delta\mathbf{x}^{(k)}) \tag{6.31}$$

which, in the vicinity of $\mathbf{x}^{(k)}$, can be approximated by Taylor series, neglecting the third and higher derivatives, by

$$z(\alpha) = F(\mathbf{x}^{(k)}) + \alpha\left(F'(\mathbf{x}^{(k)})\right)^{\mathrm{T}}\Delta\mathbf{x}^{(k)} + \frac{\alpha^2}{2}\left(\Delta\mathbf{x}^{(k)}\right)^{\mathrm{T}}\left(F''(\mathbf{x}^{(k)})\right)\Delta\mathbf{x}^{(k)}, \tag{6.32}$$

where

$$\begin{aligned}
&\Delta\mathbf{x}^{(k)} = -\mathbf{g}(\mathbf{x}^{(k)}),\\
&F'(\mathbf{x}^{(k)}) = \boldsymbol{\nabla} F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}} = \mathbf{g}(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}} = \mathbf{g}(\mathbf{x}^{(k)}),\\
&F''(\mathbf{x}^{(k)}) = \mathbf{g}'(\mathbf{x}^{(k)}) = \frac{\partial \mathbf{g}(\mathbf{x})}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}^{(k)}} = \mathbf{J}(\mathbf{x}^{(k)}) = \mathbf{H}(\mathbf{x}^{(k)}).
\end{aligned} \tag{6.33}$$

Thus

$$z(\alpha) = F(\mathbf{x}^{(k)}) - \alpha\ \left(\mathbf{g}(\mathbf{x}^{(k)})\right)^{\mathrm{T}}\ \mathbf{g}(\mathbf{x}^{(k)}) + \frac{\alpha^2}{2}\ \left(\mathbf{g}(\mathbf{x}^{(k)})\right)^{\mathrm{T}}\ \mathbf{J}(\mathbf{x}^{(k)})\ \mathbf{g}(\mathbf{x}^{(k)}). \tag{6.34}$$

The appropriate α value for a local minimum in the gradient direction can be determined from the condition $\dfrac{\partial z}{\partial x} = 0$. For the k-th iteration we get

$$\alpha^{(k)} = \frac{\left(F'(\mathbf{x}^{(k)})\right)^{\mathrm{T}}\ F'(\mathbf{x}^{(k)})}{\left(F'(\mathbf{x}^{(k)})\right)^{\mathrm{T}}\ F''(\mathbf{x}^{(k)})\ F'(\mathbf{x}^{(k)})} = \frac{\left(\mathbf{g}(\mathbf{x}^{(k)})\right)^{\mathrm{T}}\ \mathbf{g}(\mathbf{x}^{(k)})}{\left(\mathbf{g}(\mathbf{x}^{(k)})\right)^{\mathrm{T}}\ \mathbf{J}(\mathbf{x}^{(k)})\ \mathbf{g}(\mathbf{x}^{(k)})}. \tag{6.35}$$

In this case we exploited the fact that the Hess matrix of the function $F(\mathbf{x})$ is at the same time the Jacobi matrix of the function $\mathbf{g}(\mathbf{x})$. The steepest gradient method could be programmed as indicated in the `steep_grad.m` procedure listed as a part of the Program 30. The results of our benchmark test are attached at the end of the listing.

Program 30

```
 function [x,ier,n_step,xhist,residuum,minimum] = steep_grad(xini,itmax,epx,epg)
% Method of the steepest gradient for finding
% 1. the minimum of a scalar function F({x}) and its location,
% 2. the root of a system of equations g({x}) = {0},
% where the vector {x} = {x(1,) x(2), ... x(n)}.
% It is assumed that function g({x}) is a gradient of F({x}).
% In such a case the system of equations g = 0 is satisfied at the same location
% where the function F has a minimum.

% Three problem dependent functions are required
% 1. function_F.m       values of function F(x)
% 2. function_g.m       values of its gradient, i.e. of function g({x})
% 3. jacobi_g.m         Jacobi matrix for function g({x}),
%                       i.e Hess matrix of function F({x})

% input values
% xini          initial approximation
% epx;          relative tolerance parameter for solution
% epg           absolute tolerance parameter for zero value of the function
% itmax         permissible number of iterations

% output values
% x             solution
% ier           error parameter, if OK ier = 0; if not OK ier = -1
% n_step        number of iteration steps required
% xhist         history of iterations
% residuum      functional residuum of g at solution
% minimum       The value of F at minimum
%
satisfied = false;              % a priori unsatisfied
```

```
it = 0;                         % iteration counter
ier = 0;                        % error indicator, 0 if OK, -1 if wrong
x = xini;
% the method of the steepest gradient itself
while not(satisfied)
    it = it + 1;                % increase iteration counter by one
    if(it > itmax), ier = -1; break, end    % error if too many iterations
    dx = (-function_g(x))';     % direction = -gradient, transposed
    a = jacobi_g(x);            % Hess matrix of F, i.e. Jacobi matrix of g
    alphak = dx'*dx/(dx'*a*dx);
    x_new = x + alphak*dx;       % new iteration
    rdif1(it) = norm(dx)/norm(x_new);   % relative measure for root position
    rdif2(it) = norm(dx);        % absolute measure for function value at root
    satisfied = (rdif1(it) <= epx & rdif2(it) <= epg);  % convergence test
    x = x_new;                  % update for the next iteration
    xhist(:,it) = x';           % keep history of iteration results
end
 residuum = function_g(x);
 minimum = function_F(x);
 n_step = it;
% end of steep_grad
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> steep_grad_plots_c2 output
ier, no. of steps, x1, x2:
           0  19    3.9990e-001  9.0003e-001
residual value of g at x1, x2:
                   -3.2114e-004  4.5122e-004
minimum value of F at x1, x2:
                   -4.4022e+000
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

End of Program 30. □

The actual 'search trip' for the minimum can be observed in Fig. 6.6. It always starts perpendicularly to the actual contour line. After the local minimum, in that direction has been found, the descent is stopped and a new direction has to be found. There is a nice analogy to a skier who cannot turn, who is descending in a narrow mountain valley. He starts going straight down the slope until he stops. Then he finds a new straight-down-the-hill direction and continues with another iteration. Poor fellow.

A few last steps of the above descent are plotted in Fig. 6.7 in detail. The position of successive approximations is marked by circles. The adjacent numbers are the iteration counters. The cross indicates the location of the looked-after minimum. The diameter of the circle around the minimum corresponds to the prescribed location tolerance, i.e. $\varepsilon_x = 0.001$. One can observe that there is a long way to the minimum if it it hidden in a deep and narrow valley.

In this case the iteration process is stopped after the root function tolerance has been reached. This is due to the fact that in the decision statement, defining the end of the iteration process, both tolerances are coupled by the logical `and` operator.

An open-eyed reader surely notices that the number of iteration steps, indicated above Fig. 6.7, is greater by one than that associated with the last iteration step. This is due to the fact that the starting iteration is denoted by the counter one, not zero. So after n iteration steps we reach the counter $n + 1$.

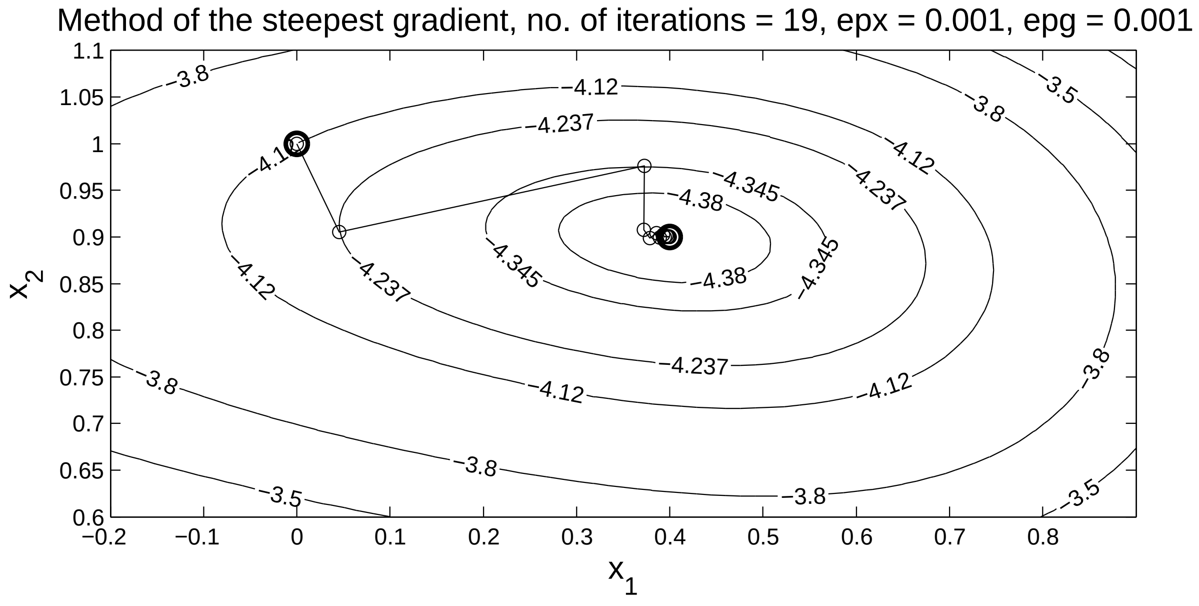


Figure 6.6: Steepest gradient method

The steepest gradient method, used for a minimum search, requires the function evaluation, as well as that of the first two derivatives. No system of algebraic equations has to be solved, however.

6.1.4 Conjugate gradient method

More details about the conjugate gradient method with an extensive list of references can be found for example in [1]. Here we only mention that the method is actually a misnomer, taking its name from the the theory of conic section geometry.

Each line passing through the center of an ellipse is called a diameter. Two diameters are mutually conjugate if each of them divide (into two segments of the same length) the chords being parallel with the other diameter. For more details see the Chapter 7.

Bibliography

[1] C. Höschl and M. Okrouhlík. Solution of systems of non-linear equations. *Strojnícky Časopis*, 54:197–227, 2003.

[2] M. Okrouhlík. *Aplikovaná mechanika kontinua II.* Vydavatelství ČVUT, Praha, 1989.

[3] J.M. Ortega and W.C. Rheinbold. *Iterative solution of nonlinear equations in several variables.* Academia Press. New York, 1970.

[4] G. Strang. *Introduction to Applied Mathematics.* Wellesley-Cambridge Press, USA, 1986.

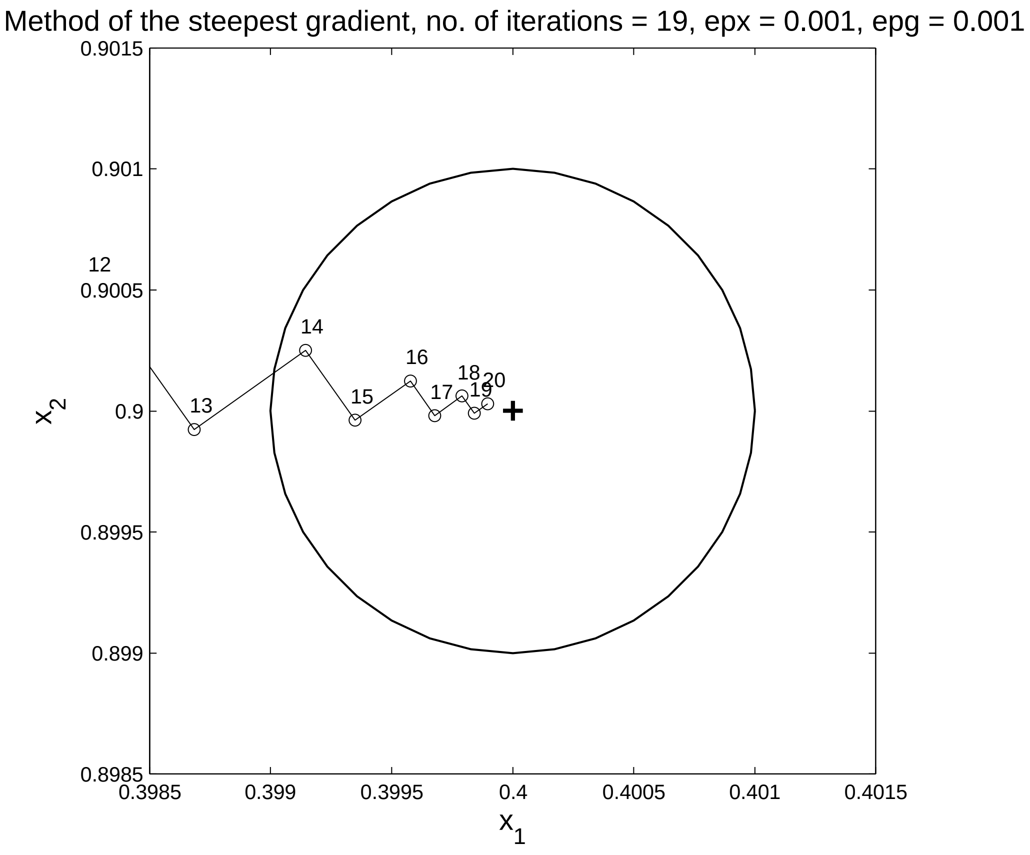


Figure 6.7: A few last steps of the steepest gradient method in detail

Chapter 7

Domain decomposition methods

This part was written and is maintained by Mrs. Marta Čertíková. More details about the author can be found in the Chapter 16.

7.1 List of abbreviations

There is a lot of abbreviations used throughout this chapter.
For a quick reference they are listed in alphabetical order here.

BDD	Balanced Domain Decomposition
BDDC	Balanced Domain Decomposition by Constraints
DD	Domain Decomposition
dof	degree of freedom
FEM	Finite Element Method
FETI	Finite Element Tearing and Interconnecting
FETI-DP	FETI Dual-Primal
GMRES	Generalized Minimal Residual
PCG	Preconditioned Conjugate Gradient

7.2 Introduction

Numerical solution of many problems in linear and nonlinear mechanics requires solving of large, sparse, unstructured linear systems.

Direct methods are often used for solution of these systems, like a frontal algorithm by Irons [11] - a variant of Gauss elimination especially designed for the FEM. Its more recent generalization suitable for parallel computers, a multifrontal algorithm, was proposed by Duff and Reid [7] and is implemented for instance in MUMPS library [1]. The direct solvers usually need a lot of memory and also computational time increases fast with data size.

Iterative solvers like PCG are less expensive in terms of memory and computational time, but they does not guarantee to converge for ill-conditioned systems. The convergence rate of iterative methods deteriorates with growing condition number of the solved linear system. The condition number of linear systems obtained by discretization of many problems in mechanics typically grows as $O(h^{-2})$, where h is the meshsize of the triangulation, so the larger the problem, the better preconditioner is usually needed.

Linear systems derived from huge problems are hard to solve by direct solvers because of their size and their lack of structure. They are also hard to solve by iterative solvers because of their large condition number. Most efficient recent methods use combination of both approaches, often together with some hierarchy in meshing. *Domain Decomposition* methods are powerful tools to handle huge linear systems arising from the discretization of differential equations. They have many common traits with another efficient tool, the multigrid methods, which are, however, out of scope of this section (an introductory article to multigrid methods is for example [28]).

Historically, they emerged from the analysis of partial differential equations, beginning with the work [25] of Schwarz in 1870. A general approach of DD methods is to decompose the underlying domain into subdomains and use this information for splitting the original large linear system into number of smaller and numerically more convenient ones. Most often DD methods are used as very efficient preconditioners for an iterative method like PCG. The intrinsic parallelism of DD algorithms and the straightforward distributability of the associated data makes this approach suitable for parallel computing.

Basic ideas of two main classes of DD methods are described here, namely of overlapping and nonoverlapping Schwarz methods. The *overlapping methods* presented in section 7.4 appeared earlier and are more intuitive. The *nonoverlapping methods*, sometimes also called *substructuring*, are described in section 7.5. It also includes two of the more recent leading DD algorithms, FETI-DP and BDDC. DD methods are generally used as preconditioners; this is why they are formulated as Richardson methods and in the section 7.6 the use of Richardson methods as preconditioners is described.

Brief illustrations of the partitioning of a domain into overlapping or nonoverlapping subdomains are given in both sections 7.4 and 7.5. Usually some existing software tool for partitioning graphs is used to complete this task, like METIS [12] or Chaco [10].

For simplicity we mainly stick to matrix and vector notation here and work only with linearized discretized equations. However, for a better understanding of DD methods it is vital to study their abstract operator and differential properties. We also assume already meshed domains and work only with matching meshes on subdomains. Discretization by means of FEM is assumed throughout this text.

For a better insight, algebraical formulations are illustrated on an example of a 2D Poisson equation, one of the most common partial differential equations encountered in various areas of engineering.

DD methods have developed a lot during past twenty years and the literature of the field is quite extensive. Let us mention just two summary texts here. A good introduction to the field is a monograph by Le Tallec [17]; it introduces the model problem in solid mechanics, presents its mathematical formulation and describes the principles of the DD methods in several forms: differential, operator and matrix form. Monograph by Toselli and Widlund [27] is a modern comprehensive 400-pages book with a list of nearly 500 related references.

An excellent textbook on iterative methods generally is Saad [24]. Aside from detailed description of iterative methods suitable for sparse linear systems, a background on linear algebra and the basic concepts of FEM and other disretization methods are given there; a whole chapter is devoted also to DD. Full text of the first edition of this book is available on url http://www-users.cs.umn.edu/˜saad/books.html. A survey information with templates on iterative methods and preconditioning can also be found in a book by Barrett et al [2] accessible on url http://www.netlib.org/templates/Templates.html.

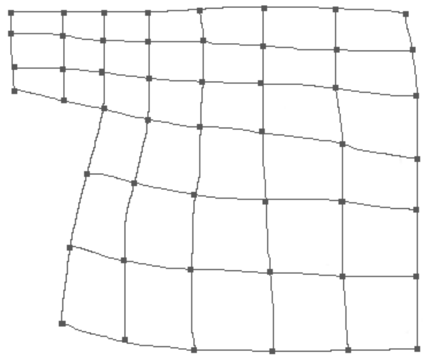
7.3 Problem to be solved

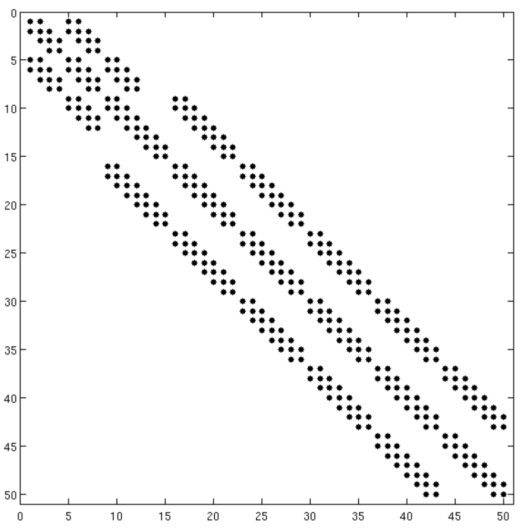
After the discretization of a linearized partial differential equation in a given domain Ω, a linear algebraic system

$$\mathbf{Ku} = \mathbf{f} \tag{7.1}$$

is to be solved with a matrix $\mathbf{K}$ and a right hand side $\mathbf{f}$ for unknown vector $\mathbf{u}$. Components of $\mathbf{u}$ are often called *degrees of freedom (dofs)*.

The matrix depicted in Fig. 7.1 right, with nonzero items marked by black dots, is an example of a matrix $\mathbf{K}$ which arises from a 2D Poisson equation discretized by four-node bilinear finite elements in the domain Ω depicted in the Fig. 7.1 left. Most of the items of the matrix $\mathbf{K}$ are zeros. Distribution of nonzero items is given by numbering of the nodes. In this example nodes in the domain are numbered along columns from top down and from left to right in order to achieve a banded matrix (it is schematically indicated in Fig. 7.1 bellow the domain).





1	5	9	16	23	30	37	44
2	6	10	17	24	31	38	45
3	7	11	18	25	32	39	46
4	8	12	19	26	33	40	47
		13	20	27	34	41	48
		14	21	28	35	42	49
		15	22	29	36	43	50

Figure 7.1: Domain Ω discretized by bilinear finite elements (left) and a structure of the corresponding matrix (right), assuming standard node numbering by columns (the numbering is schematically indicated bellow the domain).

In terms of mechanics the Poisson equation on the domain Ω can be regarded for instance as an equation of a thin membrane displacement. In the discretized Poisson equation of the form (7.1) the operator K represented by the stiffness matrix $\mathbf{K}$ associates vertical displacements at nodes represented by a vector $\mathbf{u}$, with vertical force represented by a node force vector $\mathbf{f}$. Then i-th component of the vector $\mathbf{u}$ represents numerically computed value of a displacement at i-th node.

In order to have a problem well posed, usually the displacements are prescribed in a part of a boundary of Ω, or, by other words, the solution satisfies a *Dirichlet boundary condition* there. In a discretized form it means that there are prescribed values of solution at certain boundary dofs, sometimes called *fixed variables.* Let us renumber nodes so that nodes with fixed variables are the last ones, let $\mathbf{u}_{\mathrm{x}}$ represent prescribed values of the fixed variables and $\mathbf{u}_{\mathrm{f}}$ represent unknown values of the rest, *free* variables. Let us decompose $\mathbf{K}$, $\mathbf{u}$ and $\mathbf{f}$ into blocks accordingly:

$$\mathbf{K} = \begin{bmatrix} \mathbf{K}_{\mathrm{ff}} & \mathbf{K}_{\mathrm{fx}} \\ \mathbf{K}_{\mathrm{xf}} & \mathbf{K}_{\mathrm{xx}} \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \mathbf{u}_{\mathrm{f}} \\ \mathbf{u}_{\mathrm{x}} \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \mathbf{f}_{\mathrm{f}} \\ \mathbf{f}_{\mathrm{x}} \end{bmatrix}.$$

In order to have the problem well posed, the fixed variables need to be chosen so that the matrix $\mathbf{K}_{\mathrm{ff}}$ is regular. There are two different approaches to solving (7.1) for unknowns $\mathbf{u}_{\mathrm{f}}$, either direct (7.2) or via Lagrange multipliers (7.3). Both of these approaches are used in the following text.

The unknowns $\mathbf{u}_{\mathrm{f}}$ in dependence on the prescribed values $\mathbf{u}_{\mathrm{x}}$ can be obtained directly by solving a problem

$$\mathbf{K}_{\mathrm{ff}}\mathbf{u}_{\mathrm{f}} = \mathbf{f}_{\mathrm{f}} - \mathbf{K}_{\mathrm{fx}}\mathbf{u}_{\mathrm{x}}. \tag{7.2}$$

By means of Lagrange multipliers, problem (7.2) for unknowns $\mathbf{u}_{\mathrm{f}}$ can be expressed as

$$\begin{bmatrix} \mathbf{K}_{\mathrm{ff}} & \mathbf{K}_{\mathrm{fx}} & \mathbf{0} \\ \mathbf{K}_{\mathrm{xf}} & \mathbf{K}_{\mathrm{xx}} & \mathbf{I} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{f}} \\ \mathbf{u}_{\mathrm{x}} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_{\mathrm{f}} \\ \mathbf{f}_{\mathrm{x}} \\ \mathbf{u}_{\mathrm{x}} \end{bmatrix}, \tag{7.3}$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{u}_{\mathrm{x}}$ is prescribed.

In terms of the above example of the thin membrane displacements, values $\boldsymbol{\lambda}$ of the Lagrange multipliers can be interpreted as reactions (node forces) at the nodes with fixed variables. In the context of DD methods the notion of the reactions at the nodes with fixed variables proved to be very useful. We will use the formulation (7.3) both for notational purposes and for computation of model examples of problems with fixed variables inside a domain.

In order not to get stuck in technical details, in the rest of this chapter the matrix $\mathbf{K}$ of (7.1) is assumed to be invertible, i.e. with given Dirichlet boundary condition incorporated to the right hand side by technique (7.2). However, in graphical representation of a structure of the different permutations of $\mathbf{K}$ all variables are expected to be free in order not to distort the structure.

7.4 Overlapping methods

Overlapping methods represent whole class of iterative methods for solving (7.1). Just to explain the main ideas, only two basic algorithms are described here, namely *additive* and *multiplicative* overlapping Schwarz methods on matching meshes. Both these methods at every iteration step solve only local, subdomain problems with a Dirichlet boundary condition prescribed on the inner part of subdomain boundary (it is the part that lies inside the whole domain). Solving the local problem is the same as solving the global problem (7.1) for unknown variables inside subdomain only, with all other variables fixed.

The *additive* method solves at each iteration step all subdomain problems simultaneously using only information from previous iteration step and then sums the results up, so there is a straightforward way of its parallelization by subdomains.

The *multiplicative* method at each iteration step solves all subdomain problems one after another, always using information obtained from subdomain problem just solved to improve a formulation of remaining subdomain problems. Its parallelization is more difficult, but its convergence is usually faster.

More on overlapping methods can be found for instance in Toselli and Widlund [27] or Saad [24].

7.4.1 Division of a discretized domain into overlapping subdomains

In this subsection two techniques are shown for generating overlapping subdomains with matching meshes.

Let us start by dividing all the nodes into disjunct clusters as in the Fig 7.2 left: there are three clusters of nodes marked yellow, blue and red, respectively. For every cluster let us construct a subdomain by grouping together all elements that include at least one of the nodes from the cluster as is shown in the Fig. 7.2 right for a yellow cluster – the subdomain is colored yellow. All the nodes which are not involved in elements lying outside the subdomain represent a set of *interior nodes* (marked yellow) of the yellow subdomain. Note that also nodes on the boundary of the original domain are labeled as "interior" in this sense.

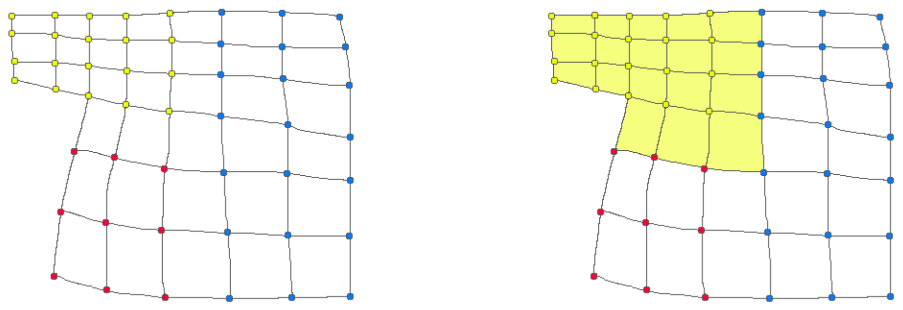


Figure 7.2: Left: distinct clusters of nodes. Right: subdomain specified by yellow nodes.

By this process the domain is divided into subdomains that overlap over one layer of elements, see Fig. 7.3 left (the overlaps are characterized by mixed colors, i.e. overlap of the blue and the yellow subdomains is green, and so on). For larger overlaps, every cluster can be enlarged by adding all nodes which are neighbors of the nodes in the cluster; now the clusters are not disjunct any more and resulting subdomains overlap over three layers of elements. This process can be repeated, always adding another extra layer of neighboring vertices to every cluster, making the overlap larger and larger.

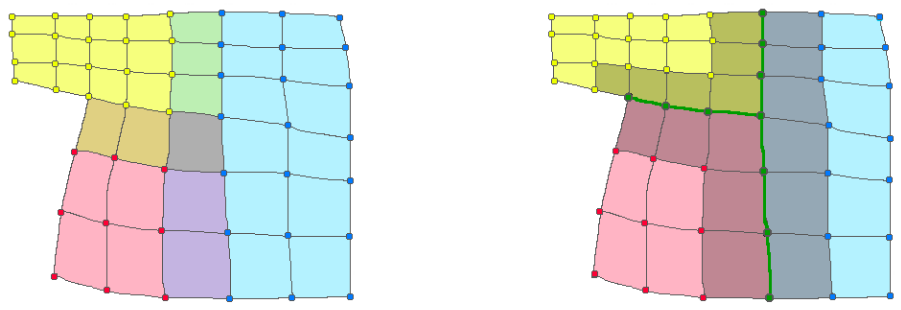


Figure 7.3: Left: three subdomains (red, yellow, blue) overlapping over one layer of elements. Right: three nonoverlapping subdomains (red, yellow, blue), all overlapped by an interface subdomain (grey).

There is another way how to divide a domain into overlapping subdomains that is related to an approach of the nonoverlapping methods. First, the domain is split into *nonoverlapping* subdomains and *interface nodes* are identified as in the subsection 7.5.1 bellow. Then one extra *interface subdomain* is constructed so that it consists of all elements that include at least one interface node, see Fig. 7.3 right. This way a set of nonoverlapping subdomains is obtained that generates set of mutually independent problems, which are "glued together" by an interface problem generated by the interface subdomain. Again, an overlap of the interface subdomain over the others can be expanded by adding neighboring layers of nodes to the cluster of interface nodes.

7.4.2 Additive overlapping Schwarz method

The additive Schwarz methods solve a boundary value problem for a partial differential equation approximately by splitting it into boundary value problems on smaller domains and adding the results.

Let us assume that the given domain is divided into N overlapping subdomains. The first approximation of the solution on the whole domain is somehow chosen, for instance all zeros. Computation of the next approximation from the previous one is described in Template 34.

Template 34, One iteration step of the additive overlapping Schwarz method

1. On each subdomain compute a new *local approximation*: solve local (subdomain) problem using values from the last (global) approximation on the inner part of subdomain boundary.
   This can be done in parallel.
2. On each subdomain compute a *local correction* as a difference between the last global approximation restricted to the subdomain and the new local approximation (computed in the previous step).
3. Compute the next global approximation by adding all the local corrections to the last global approximation.

For algebraic description of the Schwarz additive method let us introduce an operator R^i of a *restriction* from Ω to interior of Ω_i in a standard manner (see Saad [24], Le Tallec [17] or Toselli and Widlund [27]). Operator R^i is represented by a rectangular permutation matrix $\mathbf{R}^i$ of zeros and ones, which extracts only those components of a vector that belong to interior of Ω_i. Its transpose $\mathbf{R}^{i\mathrm{T}}$ corresponds to the operator $\mathrm{R}^{i\mathrm{T}}$ of a *prolongation* from interior of Ω_i to Ω.

Let us denote $\mathbf{u}_\mathrm{o}^i = \mathbf{R}^i\mathbf{u}$ variables corresponding to interior of Ω_i and $\mathbf{u}_\mathrm{r}^i$ the rest of the components of the vector $\mathbf{u}$. Let $\mathbf{u}^{(k)}$ be the last approximation of the solution of (7.1). In $(k+1)$-th iteration step, problem (7.1) is solved on each subdomain Ω_i for its interior unknowns $\mathbf{u}_\mathrm{o}^{i\,(k+1)}$ with the rest of variables $\mathbf{u}_\mathrm{r}^{i\,(k)}$ fixed, see also (7.2):

$$\mathbf{K}_\mathrm{oo}^i \mathbf{u}_\mathrm{o}^{i\,(k+1)} = \mathbf{f}_\mathrm{o}^i - \mathbf{K}_\mathrm{or}^i \mathbf{u}_\mathrm{r}^{i\,(k)}, \tag{7.4}$$

where $\mathbf{f}_\mathrm{o}^i = \mathbf{R}^i\mathbf{f}$, $\quad \mathbf{K}_\mathrm{or}^i\mathbf{u}_\mathrm{r}^{i\,(k)} = \mathbf{R}^i\mathbf{K}\mathbf{u}^{(k)} - \mathbf{K}_\mathrm{oo}^i\mathbf{R}^i\mathbf{u}^{(k)}$ and

$$\mathbf{K}_\mathrm{oo}^i = \mathbf{R}^i\mathbf{K}\mathbf{R}^{i\mathrm{T}} \tag{7.5}$$

is a stiffness matrix of the local (subdomain) problem with zero values prescribed on the inner part of its boundary, which can be constructed by a *subassembling process* by considering only degrees of freedom inside Ω_i.

Local correction $\Delta\mathbf{u}_\mathrm{o}^{i\,(k)}$ is a difference between the new local approximation $\mathbf{u}_\mathrm{o}^{i\,(k+1)}$ and the last approximation $\mathbf{u}^{(k)}$ restricted to i-th subdomain and can be expressed as

$$\begin{aligned}\Delta\mathbf{u}_\mathrm{o}^{i\,(k)} &= \mathbf{u}_\mathrm{o}^{i\,(k+1)} - \mathbf{R}^i\mathbf{u}^{(k)} = (\mathbf{K}_\mathrm{oo}^i)^{-1}(\mathbf{f}_\mathrm{o}^i - \mathbf{K}_\mathrm{or}^i\mathbf{u}_\mathrm{r}^{i\,(k)} - \mathbf{K}_\mathrm{oo}^i\mathbf{R}^i\mathbf{u}^{(k)}) \\ &= (\mathbf{K}_\mathrm{oo}^i)^{-1}\mathbf{R}^i(\mathbf{f} - \mathbf{K}\mathbf{u}^{(k)}) = (\mathbf{K}_\mathrm{oo}^i)^{-1}\mathbf{R}^i\mathbf{r}^{(k)},\end{aligned}$$

where $\mathbf{r}^{(k)}$ is a residual of $\mathbf{u}^{(k)}$. Local components $\Delta\mathbf{u}_\mathrm{o}^{i\,(k)}$ are put on their global places by the prolongation operator $\mathrm{R}^{i\mathrm{T}}$ and all these local corrections are summed up and added to the last approximation $\mathbf{u}^{(k)}$ to obtain a next approximation $\mathbf{u}^{(k+1)}$ as

$$\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} + \sum_{i=1}^{N} \mathbf{R}^{i\mathrm{T}} \Delta\mathbf{u}_\mathrm{o}^{i\,(k)}, \tag{7.6}$$

where N is the number of subdomains.

This can be formulated as a Richardson method for the problem (7.1) as

$$\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} + \sum_{i=1}^{N} \mathbf{R}^{i\mathrm{T}} (\mathbf{K}_{\mathrm{oo}}^{i})^{-1} \mathbf{R}^{i}\, \mathbf{r}^{(k)}. \tag{7.7}$$

Algebraic description of the additive overlapping Schwarz method is summarized in Template 35.

Template 35, Algebraic description of the additive overlapping Schwarz method

Choose $\mathbf{u}^{(0)}$ arbitrarily and for $k := 0, 1, 2, \ldots$ repeat

1. Compute residual $\mathbf{r}^{(k)} := \mathbf{f} - \mathbf{K}\mathbf{u}^{(k)}$. If residual is small enough, stop.
2. Compute local corrections $\Delta\mathbf{u}_{\mathrm{o}}^{i\,(k)} := (\mathbf{K}_{\mathrm{oo}}^{i})^{-1}\mathbf{R}^{i}\mathbf{r}^{(k)}$.
   This can be done on subdomains in parallel.
3. Compute the next approximation by adding the local corrections to the last approximation: $\mathbf{u}^{(k+1)} := \mathbf{u}^{(k)} + \sum_{i=1}^{N} \mathbf{R}^{i\mathrm{T}} \Delta\mathbf{u}_{\mathrm{o}}^{i\,(k)}$.

Implementation remarks:

Matrices $\mathbf{R}^{i}$ and $\mathbf{R}^{i\mathrm{T}}$ are used here only for notational purposes, they need not to be constructed.

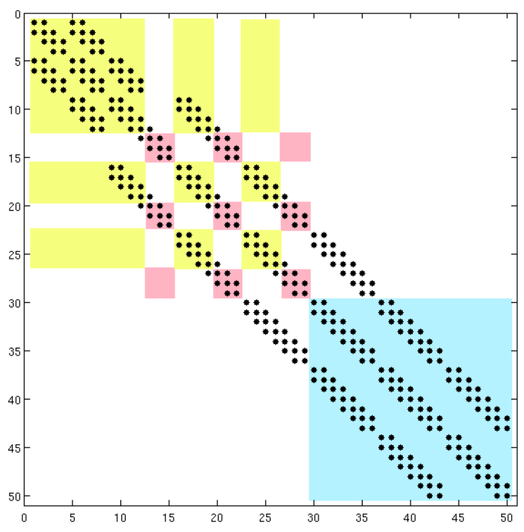
The computation of the local correction $\Delta\mathbf{u}_{\mathrm{o}}^{i\,(k)}$ is usually implemented as solution of a subdomain problem with zero Dirichlet boundary condition on the inner boundary.

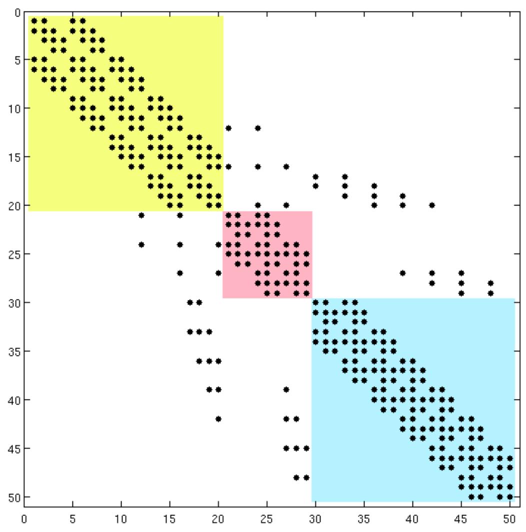
A computation of the residual at the first step of the algorithm can also be implemented by subdomains and so the whole matrix $\mathbf{K}$ need not be assembled.

Let us ilustrate the overlapping Schwarz additive method on the example given above with the division of Ω depicted in Fig. 7.3 left. In the Fig. 7.4 left there is the matrix $\mathbf{K}$ with parts corresponding to interior nodes of subdomains coloured by the same color as the corresponding subdomains, with scheme of node numbering bellow. The algorithm is more clear after permutation of the matrix as in the Fig. 7.4 right: first take all interior nodes of the first subdomain (yellow), then all interior nodes of the second one (red) and then the last one (blue). The resulting matrix has on its diagonal square matrices $\mathbf{K}_{\mathrm{oo}}^{i}$ (yellow, red and blue) corresponding to interior nodes of individual subdomains. In every step linear systems with the diagonal blocks $\mathbf{K}_{\mathrm{oo}}^{i}$ are solved. It can be seen that in this special case the algorithm of the Schwarz additive method is the same as a standard block-Jacobi method.

Such a permutation can be only done in case that all clusters of interior nodes are disjunct. If subdomains overlap over more then one layer of elements or elements of higher degree are used, this simplifying rearrangement of the whole matrix is not possible and the additive overlapping Schwarz method can be regarded as a more general block-Jacobi method with overlapping blocks (for more details see Saad [24]).

The stiffness matrix after the permutation corresponding to the division of Ω as in Fig. 7.3 right is depicted in the Fig. 7.5.





1	5	9	16	23	30	37	44
2	6	10	17	24	31	38	45
3	7	11	18	25	32	39	46
4	8	12	19	26	33	40	47
		13	20	27	34	41	48
		14	21	28	35	42	49
		15	22	29	36	43	50

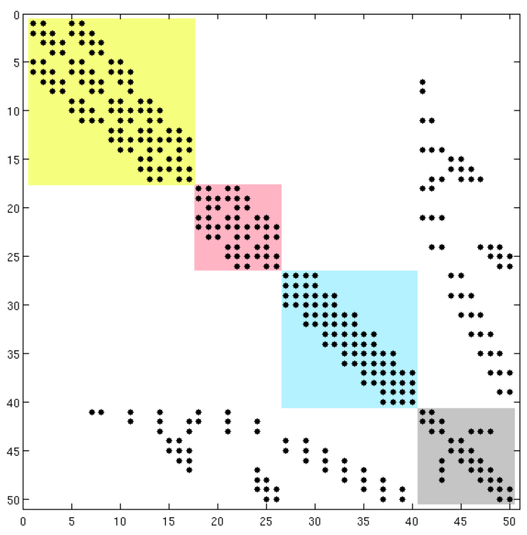
1	5	9	13	17	30	31	32
2	6	10	14	18	33	34	35
3	7	11	15	19	36	37	38
4	8	12	16	20	39	40	41
		21	24	27	42	43	44
		22	25	28	45	46	47
		23	26	29	48	49	50

Figure 7.4: The stiffness matrix for domain in Fig. 7.3 left before (left) and after renumbering (right). The scheme of node numbering is bellow the corresponding matrix.

7.4.3 Multiplicative overlapping Schwarz method

Multiplicative overlapping Schwarz method (or *Schwarz alternating* method) differs from the additive Schwarz method in Template 35 by solving subdomain problems not simultaneously but one after another, immediately updating the current approximation by the local correction and using this updated approximation in the following computation.

One iteration step of the multiplicative overlapping Schwarz method is described in Template 36.



1	5	9	12	15	44	27	28
2	6	10	13	16	45	29	30
3	7	11	14	17	46	31	32
4	8	41	42	43	47	33	34
		18	21	24	48	35	36
		19	22	25	49	37	38
		20	23	26	50	39	40

Figure 7.5: The stiffness matrix for domain in Fig. 7.3 right, after renumbering.

Template 36, One iteration step of the multiplicative overlapping Schwarz method

Repeat for all subdomains, one after another:

1. Compute a new local approximation: solve local (subdomain) problem using values from the current global approximation as a Dirichlet boundary condition prescribed on the inner part of the subdomain boundary.

2. Compute a local correction as a difference between the current global approximation restricted to the subdomain and the new local approximation.

3. Update the current global approximation by adding the local correction to it.

Algebraic description of the multiplicative overlapping Schwarz method is in Template 37.

Template 37, Algebraic description of the multiplicative overlapping Schwarz method

```
Choose $\mathbf{u}^{(0)}$ arbitrarily and for $k := 0, 1, 2, \dots$ repeat
  1. Set $\mathbf{u}^{(k,0)} := \mathbf{u}^{(k)}$ and for $i := 1, 2, \dots N$ repeat
     (a) Compute the local correction on subdomain $\Omega_i$:
         $\Delta\mathbf{u}_{\mathrm{o}}^{i\,(k)} := \left(\mathbf{K}_{\mathrm{oo}}^{i}\right)^{-1}\mathbf{R}^{i}\left(\mathbf{f} - \mathbf{K}\mathbf{u}^{(k,i-1)}\right)$ .
     (b) Update the approximation by adding the local correction to it:
         $\mathbf{u}^{(k,i)} := \mathbf{u}^{(k,i-1)} + \mathbf{R}^{i\mathrm{T}}\Delta\mathbf{u}_{\mathrm{o}}^{i\,(k)}$ .
     Set $\mathbf{u}^{(k+1)} := \mathbf{u}^{(k,N)}$
  2. Compute residual $\mathbf{r}^{(k+1)} := \mathbf{f} - \mathbf{K}\mathbf{u}^{(k+1)}$. If residual is small enough, stop.
```

Implementation remarks:

Matrices $\mathbf{R}^i$ and $\mathbf{R}^{i\mathrm{T}}$ are used here only for notational purposes, they need not to be constructed. Moreover, the local residual $\mathbf{R}^i(\mathbf{f} - \mathbf{K}\mathbf{u}^{(k,i-1)})$ at the inner cycle can be updated locally.

Computation of the local correction $\Delta\mathbf{u}_{\mathrm{o}}^{i\,(k)}$ is usually implemented as solution of a subdomain problem with zero Dirichlet boundary condition on the inner boundary.

The global residual at the last step is in fact already computed by successive updating during the computation of the local residuals at the inner cycle and so the whole matrix $\mathbf{K}$ need not be assembled.

On the example of the domain divided as in Fig. 7.3 left with corresponding matrix in the Fig. 7.4, one step of the Schwarz multiplicative method can be described like this: consider only components that correspond to interior nodes of the yellow subdomain (yellow blocks) as unknown and compute their new value; all other variables are fixed with values of the last approximation. Construct an updated approximation that consists of these new values at interior nodes of the yellow subdomain and old values elsewhere. Now repeat the same procedure for red subdomain using the updated approximation as values for fixed variables, then repeat it for the last, blue subdomain.

With help of the Fig. 7.4 right it can be seen that in this example the algorithm of the Schwarz multiplicative method is the same as a standard block-Gauss-Seidel method. The permutation of the Fig. 7.4 can be done only in case that all clusters of interior nodes are disjunct. If subdomains overlap over more then one layer of elements or elements of higher degree are used, this simplifying rearrangement of the whole matrix is not possible and the multiplicative overlapping Schwarz method can be regarded as a more general block-Gauss-Seidel method with overlapping blocks (for more details see Saad [24]).

Multiplicative Schwarz methods generally have better convergence properties, on the other hand there is no straightforward way of their parallelization. However, if the domain is divided into subdomains like in the Fig. 7.3 right, an *interface problem* on the interface subdomain can be alternated with problems on the rest of the subdomains, which are mutually independent and can be solved in parallel, see also the Fig. 7.5.

7.5 Nonoverlapping methods

Nonoverlapping methods (also called substructuring) can be regarded as a generalization of the overlapping algorithms using the partition of the domain like in the Fig. 7.3 right: the domain is split into nonoverlapping subdomains tied-up together by means of some interface communication. There are many other choices of the communication than the interface subdomain as in the case of the overlapping methods.

Nonoverlapping methods are represented here by two basic approaches to interconnection among the subdomains, a *primal* and a *dual* one, with the interface problem formed by a Schur complement problem. Recent leading algorithms of both types, namely the BDDC and FETI-DP methods, are described. More details on nonoverlapping methods can be found for instance in Toselli and Widlund [27], comprehensive overview and comparison of these methods can also be found in [26].

7.5.1 Interface and interior

Let us start again with the discretized domain as in the Fig. 7.1 and split the domain into nonoverlapping subdomains so that every element belongs to exactly one subdomain, as in the Fig. 7.6 left. Now we can distinguish two types of nodes, see Fig. 7.6 right:

- *Interface nodes* - marked green - nodes belonging to more than one subdomain.
- *Interior nodes* - marked the same colour as the corresponding subdomain - nodes belonging only to one subdomain. Note that nodes on the boundary of the original domain are labeled as "interior" in this sense, if they belong to (outer) boundary of one subdomain only.

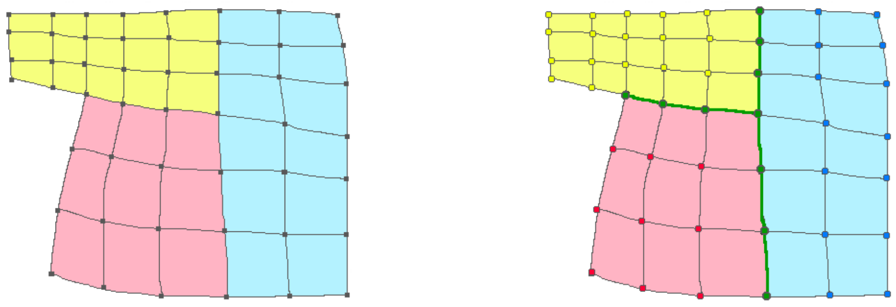
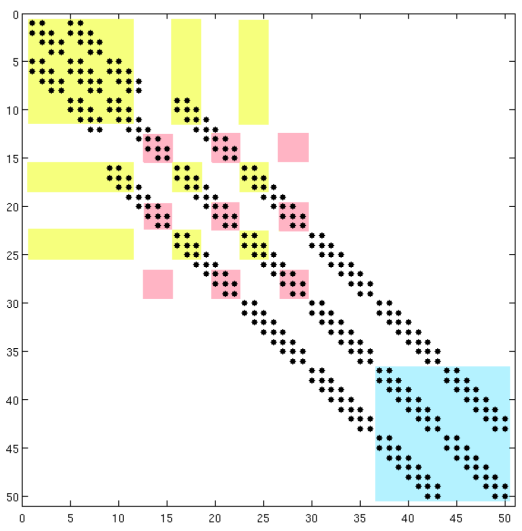


Figure 7.6: Left: Domain divided into nonoverlapping subdomains.
Right: Interior (yellow, red and blue) and interface (green) nodes

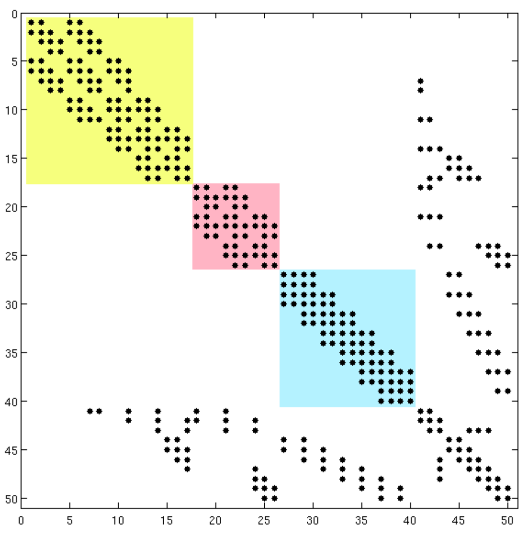
7.5.2 Schur complement system for interface nodes

The Schur complement problem represents a reduction of the original problem to the interface by eliminating all interior unknowns; this reduction is sometimes called a *static condensation*.

The matrix in Fig. 7.7 left is an example of a stiffness matrix which arises from a Poisson equation discretized by bilinear finite elements in the domain in the Fig. 7.6. Items of the matrix corresponding to interior of a particular subdomain are colored by the same color as the subdomain, nonzero items are marked as black dots. Nodes are numbered as usual in order to achieve a banded matrix, this is column by column in this example. Scheme of node numbering is bellow the matrix.



1	5	9	16	23	30	37	44
2	6	10	17	24	31	38	45
3	7	11	18	25	32	39	46
4	8	12	19	26	33	40	47
		13	20	27	34	41	48
		14	21	28	35	42	49
		15	22	29	36	43	50



1	5	9	12	15	44	27	28
2	6	10	13	16	45	29	30
3	7	11	14	17	46	31	32
4	8	41	42	43	47	33	34
		18	21	24	48	35	36
		19	22	25	49	37	38
		20	23	26	50	39	40

Figure 7.7: The stiffness matrix for domain in Fig. 7.6 before (left) and after renumbering (right). The scheme of node numbering is bellow the corresponding matrix.

In order to get a suitable structure for Schur complement system, let us rearrange the system $\mathbf{Ku} = \mathbf{f}$ and rewrite it in a block form, with the first block corresponding to unknowns in subdomain interiors ordered subdomain after subdomain and the second block corresponding to unknowns on the interface (see Fig. 7.7 right):

$$\begin{bmatrix} \mathbf{K}_{\mathrm{oo}} & \mathbf{K}_{\mathrm{or}} \\ \mathbf{K}_{\mathrm{ro}} & \mathbf{K}_{\mathrm{rr}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{o}} \\ \widehat{\mathbf{u}} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_{\mathrm{o}} \\ \widehat{\mathbf{f}} \end{bmatrix}, \tag{7.8}$$

where $\widehat{\mathbf{u}}$ represents all the interface unknowns. The hat ($\widehat{\ }$) symbol is used to denote global interface quantities.

Different subdomains have disjunct sets of interior unknowns with no connections among them, so $\mathbf{K}_{\mathrm{oo}}$, $\mathbf{u}_{\mathrm{o}}$ and $\mathbf{f}_{\mathrm{o}}$ have the structure

$$\mathbf{K}_{\mathrm{oo}} = \begin{bmatrix} \mathbf{K}^1_{\mathrm{oo}} & & & \\ & \mathbf{K}^2_{\mathrm{oo}} & & \\ & & \ddots & \\ & & & \mathbf{K}^N_{\mathrm{oo}} \end{bmatrix}, \quad \mathbf{u}_{\mathrm{o}} = \begin{bmatrix} \mathbf{u}^1_{\mathrm{o}} \\ \mathbf{u}^2_{\mathrm{o}} \\ \vdots \\ \mathbf{u}^N_{\mathrm{o}} \end{bmatrix}, \quad \mathbf{f}_{\mathrm{o}} = \begin{bmatrix} \mathbf{f}^1_{\mathrm{o}} \\ \mathbf{f}^2_{\mathrm{o}} \\ \vdots \\ \mathbf{f}^N_{\mathrm{o}} \end{bmatrix}, \tag{7.9}$$

where $\mathbf{u}^i_{\mathrm{o}}$ and $\mathbf{f}^i_{\mathrm{o}}$ represent $\mathbf{u}_{\mathrm{o}}$ and $\mathbf{f}_{\mathrm{o}}$ restricted to Ω_i and $\mathbf{K}_{\mathrm{oo}}$ is a block diagonal matrix with each block $\mathbf{K}^i_{\mathrm{oo}}$ corresponding to interior unknowns of subdomain Ω_i. Each $\mathbf{K}^i_{\mathrm{oo}}$ can be interpreted as a matrix of a problem on Ω_i with Dirichlet boundary condition on the interface and so it is invertible. Interface unknowns cannot be separated this way, as every of them belongs to two or more subdomains.

After eliminating all the interior unknowns from (7.8) we get

$$\begin{bmatrix} \mathbf{K}_{\mathrm{oo}} & \mathbf{K}_{\mathrm{or}} \\ \mathbf{0} & \widehat{\mathbf{S}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{o}} \\ \widehat{\mathbf{u}} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_{\mathrm{o}} \\ \widehat{\mathbf{g}} \end{bmatrix}, \tag{7.10}$$

where $\widehat{\mathbf{S}} = \mathbf{K}_{\mathrm{rr}} - \mathbf{K}_{\mathrm{ro}}\mathbf{K}^{-1}_{\mathrm{oo}}\mathbf{K}_{\mathrm{or}}$ is a *Schur complement* of (7.8) with respect to interface and $\widehat{\mathbf{g}} = \widehat{\mathbf{f}} - \mathbf{K}_{\mathrm{ro}}\mathbf{K}^{-1}_{\mathrm{oo}}\mathbf{f}_{\mathrm{o}}$ is sometimes called *condensed right hand side*.

Problem (7.10) can be split into subdomain problems

$$\mathbf{K}_{\mathrm{oo}}\mathbf{u}_{\mathrm{o}} = \mathbf{f}_{\mathrm{o}} - \mathbf{K}_{\mathrm{or}}\widehat{\mathbf{u}}\,, \tag{7.11}$$

and a *Schur complement problem*

$$\widehat{\mathbf{S}}\,\widehat{\mathbf{u}} = \widehat{\mathbf{g}}\,. \tag{7.12}$$

Problem (7.11) represents N independent subdomain problems with Dirichlet boundary condition $\mathbf{u}^i_{\mathrm{r}}$ prescribed on the interface

$$\mathbf{K}^i_{\mathrm{oo}}\mathbf{u}^i_{\mathrm{o}} = \mathbf{f}^i_{\mathrm{o}} - \mathbf{K}^i_{\mathrm{or}}\mathbf{u}^i_{\mathrm{r}}\,, \tag{7.13}$$

where $\mathbf{u}^i_{\mathrm{r}}$ represents $\widehat{\mathbf{u}}$ restricted to the interface of Ω_i and $\mathbf{K}^i_{\mathrm{or}}$ is a block of $\mathbf{K}_{\mathrm{or}}$ corresponding to Ω_i (when using FEM for discretization, $\mathbf{K}^i_{\mathrm{oo}}$ and $\mathbf{K}^i_{\mathrm{or}}$ are assembled only from element matrices for elements contained in Ω_i). The Schur complement problem (7.12) represents a problem for interface unknowns $\widehat{\mathbf{u}}$ only. Its decomposition to subdomain problems is handled in the next section, as it is not so straightforward.

Original problem (7.8) now can be solved by formulating and solving the Schur complement problem (7.12) for $\widehat{\mathbf{u}}$ and then obtaining $\mathbf{u}_{\mathrm{o}}$ from (7.11) after substitution of $\widehat{\mathbf{u}}$ to the right hand side. Schur complement algorithm for solving (7.8) is described in Template 38.

Template 38, Schur complement algorithm

1. Compute $\widehat{\mathbf{S}}$ and $\widehat{\mathbf{g}}$ by eliminating all interior unknowns from (7.8), i.e. by factorization of $\mathbf{K}_{\mathrm{oo}}$. The elimination can be done in parallel on subdomains.
2. Solve (7.12) for $\widehat{\mathbf{u}}$.
3. Solve (7.11) for $\mathbf{u}_{\mathrm{o}}$. This represents a backsubstitution using factorization of $\mathbf{K}_{\mathrm{oo}}$ from step (1) and can be done in parallel on subdomains.

Implementation remark:

The Schur complement $\widehat{\mathbf{S}}$ need not be computed and stored explicitly if (7.12) is solved by iterative methods like PCG, that use only multiplication of $\widehat{\mathbf{S}}$ by some vector $\mathbf{v}$. The multiplication can be implemented by using existing block factorization of $\mathbf{K}_{\mathrm{oo}}$ as $\widehat{\mathbf{S}}\mathbf{v} = (\mathbf{K}_{\mathrm{rr}} - \mathbf{K}_{\mathrm{ro}}\mathbf{K}_{\mathrm{oo}}{}^{-1}\mathbf{K}_{\mathrm{or}})\mathbf{v} = \mathbf{K}_{\mathrm{rr}}\mathbf{v} - \mathbf{K}_{\mathrm{ro}}\mathbf{v}_{\mathrm{or}}$, where $\mathbf{v}_{\mathrm{or}}$ is a solution of $\mathbf{K}_{\mathrm{oo}}\mathbf{v}_{\mathrm{or}} = -\mathbf{K}_{\mathrm{or}}\mathbf{v}$. In a matrix form this can be also expressed as

$$\begin{bmatrix} \mathbf{K}_{\mathrm{oo}} & \mathbf{K}_{\mathrm{or}} & \mathbf{0} \\ \mathbf{K}_{\mathrm{ro}} & \mathbf{K}_{\mathrm{rr}} & \mathbf{I} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}_{\mathrm{or}} \\ \mathbf{v} \\ -\widehat{\mathbf{S}}\mathbf{v} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{v} \end{bmatrix}, \tag{7.14}$$

where $\mathbf{I}$ is an identity matrix.

Benefits of the Schur complement algorithm consist in decomposing one large global problem into several independent, smaller local (subdomain) problems (7.11) and the Schur complement problem (7.12), that is global, but smaller and tends to be better conditioned than the original problem. Its condition number grows as $O(H^{-1}h^{-1})$, where h is a characteristic element size and H is a characteristic subdomain size, while a condition number of $\mathbf{K}$ grows as $O(h^{-2})$ (see Brenner [3]). Standard algebraic methods, either direct or iterative, often can be used for its solution. Typically PCG is used for symmetric and preconditioned GMRES for nonsymmetric linear systems, with standard preconditioners. Schur complement technique can be used recursively, which leads to multilevel methods.

For later use, let us analyze the problem (7.10) in more detail. The interior part $\mathbf{u}_{\mathrm{o}}$ of the solution of (7.10) can be further split as $\mathbf{u}_{\mathrm{o}} = \mathbf{u}_{\mathrm{oo}} + \mathbf{u}_{\mathrm{or}}$ with $\mathbf{u}_{\mathrm{oo}}$ reflecting an effect of $\mathbf{f}_{\mathrm{o}}$ and the problem (7.11) can be split accordingly:

$$\mathbf{K}_{\mathrm{oo}}\mathbf{u}_{\mathrm{oo}} = \mathbf{f}_{\mathrm{o}}\,, \tag{7.15}$$

$$\mathbf{K}_{\mathrm{oo}}\mathbf{u}_{\mathrm{or}} = -\mathbf{K}_{\mathrm{or}}\widehat{\mathbf{u}}\,. \tag{7.16}$$

Problem (7.15) represents independent subdomain problems with zero Dirichlet boundary condition on the interface and its solution $\mathbf{u}_{\mathrm{oo}}$ can be used for formulation of the right hand side of (7.10) or (7.12) as $\widehat{\mathbf{g}} = \mathbf{f}_{\mathrm{r}} - \mathbf{K}_{\mathrm{ro}}\mathbf{u}_{\mathrm{oo}}$. Problem (7.15) can be expressed in a way compatible with (7.10) as

$$\begin{bmatrix} \mathbf{K}_{\mathrm{oo}} & \mathbf{K}_{\mathrm{or}} \\ \mathbf{0} & \widehat{\mathbf{S}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{oo}} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_{\mathrm{o}} \\ \mathbf{0} \end{bmatrix}. \tag{7.17}$$

Problem (7.16) represents independent subdomain problems with $\widehat{\mathbf{u}}$ prescribed as Dirichlet boundary condition on the interface. The Schur complement problem can be expressed either as (7.12) in terms of just the interface unknowns, or on the whole domain using (7.12) and (7.16) together as

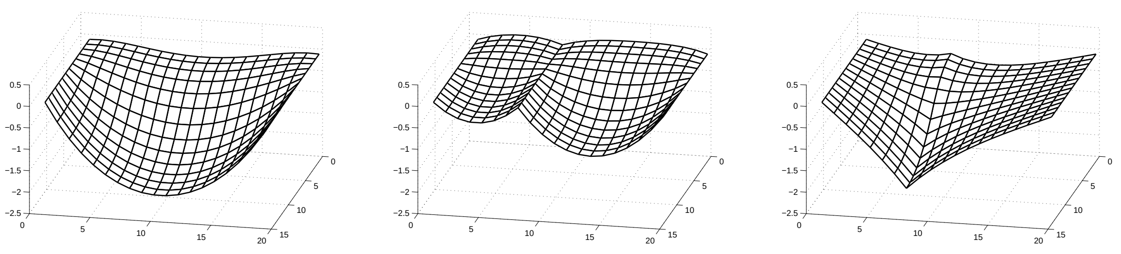
$$\begin{bmatrix} \mathbf{K}_{\mathrm{oo}} & \mathbf{K}_{\mathrm{or}} \\ \mathbf{0} & \widehat{\mathbf{S}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{or}} \\ \widehat{\mathbf{u}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \widehat{\mathbf{g}} \end{bmatrix}. \tag{7.18}$$

Solutions of (7.18) are sometimes called *functions with minimal energy* inside subdomains. They are completely represented by their values $\widehat{\mathbf{u}}$ on the interface; interior values $\mathbf{u}_{\mathrm{or}}$ are determined by (7.16).

For a better insight, this algebraical formulation will be illustrated on an example of a 2D Poisson equation representing displacements of a thin membrane. In the discretized

Poisson equation $\mathbf{Ku} = \mathbf{f}$ the operator K represented by the matrix $\mathbf{K}$ associates vector $\mathbf{u}$ of vertical displacements of a thin membrane at nodes of a given 2D domain with node force vector $\mathbf{f}$. A solution of the Poisson equation on a rectangular domain is depicted in the Fig.7.8 left. The membrane is fixed on the left and right edge by prescribed zero displacements and its front part is loaded by forces oriented down.

Let us assume the domain to be divided into two rectangular subdomains so that the interface is parallel with the fixed edges. Equation (7.15) or (7.17) represents problem with zero displacements on the interface and the given node force $\mathbf{f}_\mathrm{o}$ at interior nodes, see Fig. 7.8 center. Equation (7.16) or (7.18) represents problem with displacements $\widehat{\mathbf{u}}$ on the interface and zero node forces at interior nodes, see Fig. 7.8 right.



$$\begin{bmatrix} \mathbf{u}_\mathrm{o} \\ \widehat{\mathbf{u}} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_\mathrm{oo} \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{u}_\mathrm{or} \\ \widehat{\mathbf{u}} \end{bmatrix}$$

Figure 7.8: An example of a solution of a 2D Poisson equation (left), decomposed to the Schur complement part (right) and the part reflecting an effect of the interior load forces (center).

Problem (7.11) can be solved for any given vector $\widehat{\mathbf{u}}$ of interface displacements, but only if also (7.12) holds, the interior solution $\mathbf{u}_\mathrm{o}$ computed from (7.11) represents the interior part of the solution of the original problem $\mathbf{Ku} = \mathbf{f}$. If $\widehat{\mathbf{u}}$ does not satisfy equation (7.12), solution computed from (7.11) shows inbalanced reactions on the interface corresponding to additional interface nodal forces $\widehat{\mathbf{S}}\widehat{\mathbf{u}} - \widehat{\mathbf{g}}$, which disturb the equilibrium on the interface.

7.5.3 Decomposition of the Schur complement problem

If the original problem is too large, even the Schur complement problem (7.12) might be too large and ill-conditioned to be solved by standard algebraic methods. An advantage of using iterative substructuring DD methods for solving (7.12) is that the Schur complement problem is not assembled and solved as a whole. Instead only *local Schur complement problems* on subdomains are repeatedly solved and in every iteration step just interface information between neighboring subdomains is exchanged. Moreover, even local Schur complement problems need not be assembled; subdomain problems with Dirichlet and Neumann boundary condition on the interface can be solved instead.

The local Schur complement operator S^i operates only on the interface unknowns of a subdomain Ω_i. The local Schur complement problem is obtained as in the previous section, the only difference is that the process is performed on the subdomain Ω_i rather than on the whole domain Ω. Let us consider problem (7.8) restricted to Ω_i:

$$\begin{bmatrix} \mathbf{K}^i_{\mathrm{oo}} & \mathbf{K}^i_{\mathrm{or}} \\ \mathbf{K}^i_{\mathrm{ro}} & \mathbf{K}^i_{\mathrm{rr}} \end{bmatrix} \begin{bmatrix} \mathbf{u}^i_{\mathrm{o}} \\ \mathbf{u}^i_{\mathrm{r}} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^i_{\mathrm{o}} \\ \mathbf{f}^i_{\mathrm{r}} \end{bmatrix}, \tag{7.19}$$

where $\mathbf{u}^i_{\mathrm{r}}$ represents interface unknowns belonging to Ω_i and $\mathbf{K}^i_{\mathrm{rr}}$, $\mathbf{K}^i_{\mathrm{ro}}$ and $\mathbf{K}^i_{\mathrm{or}}$ represent a local contribution of Ω_i to the global blocks $\mathbf{K}_{\mathrm{rr}}$, $\mathbf{K}_{\mathrm{ro}}$ and $\mathbf{K}_{\mathrm{or}}$ (when using FEM for discretization, $\mathbf{K}^i_{\mathrm{rr}}$, $\mathbf{K}^i_{\mathrm{ro}}$ and $\mathbf{K}^i_{\mathrm{or}}$ are assembled from element matrices for elements contained in Ω_i only). However, it is not clear how to determine local interface forces $\mathbf{f}^i_{\mathrm{r}}$; for more detail see end of this section.

After eliminating all the interior unknowns from (7.19) we get

$$\begin{bmatrix} \mathbf{K}^i_{\mathrm{oo}} & \mathbf{K}^i_{\mathrm{or}} \\ \mathbf{0} & \mathbf{S}^i \end{bmatrix} \begin{bmatrix} \mathbf{u}^i_{\mathrm{o}} \\ \mathbf{u}^i_{\mathrm{r}} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^i_{\mathrm{o}} \\ \mathbf{g}^i \end{bmatrix}, \tag{7.20}$$

where $\mathbf{S}^i = \mathbf{K}^i_{\mathrm{rr}} - \mathbf{K}^i_{\mathrm{ro}}(\mathbf{K}^i_{\mathrm{oo}})^{-1}\mathbf{K}^i_{\mathrm{or}}$ is the *local Schur complement* of (7.19) with respect to interface and $\mathbf{g}^i = \mathbf{f}^i_{\mathrm{r}} - \mathbf{K}^i_{\mathrm{ro}}(\mathbf{K}^i_{\mathrm{oo}})^{-1}\mathbf{f}^i_{\mathrm{o}}$. Problem (7.20) can be split into two problems: the local subdomain problem (7.13) and the *local Schur complement problem*

$$\mathbf{S}^i\, \mathbf{u}^i_{\mathrm{r}} = \mathbf{g}^i. \tag{7.21}$$

In terms of the example of a 2D discretized Poisson equation, the local Schur complement $\mathbf{S}^i$ can be viewed as an assignment of the local interface node force vector $\mathbf{g}^i$ to the local interface nodal displacement $\mathbf{u}^i_{\mathrm{r}}$ so that a problem (7.13) with Dirichlet boundary condition $\mathbf{u}^i_{\mathrm{r}}$ on the interface and a problem (7.20) with Neumann boundary condition $\mathbf{g}^i$ on the interface have the same solution $\mathbf{u}^i$ on Ω_i.

The local Schur complement problem (7.21) generally is not well posed, as local Schur complement $\mathbf{S}^i$ is not invertible if Ω_i is so called *floating subdomain*, boundary of which does not contain any part of Dirichlet boundary condition of the original problem.

The local Schur complement problem can be expressed either as (7.21) in terms of just local interface unknowns, or like (7.18) on the whole subdomain as

$$\begin{bmatrix} \mathbf{K}^i_{\mathrm{oo}} & \mathbf{K}^i_{\mathrm{or}} \\ \mathbf{0} & \mathbf{S}^i \end{bmatrix} \begin{bmatrix} \mathbf{u}^i_{\mathrm{or}} \\ \mathbf{u}^i_{\mathrm{r}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{g}^i \end{bmatrix} \tag{7.22}$$

with solution representing some function with minimal energy on Ω_i.

Now we can decompose the Schur complement problem (7.12) to independent local Schur complement problems as

$$\mathbf{S}\mathbf{u}_{\mathrm{r}} = \mathbf{g}\,, \tag{7.23}$$

where

$$\mathbf{S} = \begin{bmatrix} \mathbf{S}^1 & & & \\ & \mathbf{S}^2 & & \\ & & \ddots & \\ & & & \mathbf{S}^N \end{bmatrix}, \qquad \mathbf{u}_{\mathrm{r}} = \begin{bmatrix} \mathbf{u}_{\mathrm{r}}{}^1 \\ \mathbf{u}_{\mathrm{r}}{}^2 \\ \vdots \\ \mathbf{u}_{\mathrm{r}}{}^N \end{bmatrix}, \qquad \mathbf{g} = \begin{bmatrix} \mathbf{g}^1 \\ \mathbf{g}^2 \\ \vdots \\ \mathbf{g}^N \end{bmatrix}. \tag{7.24}$$

In order to establish relations among the Schur complement $\widehat{\mathbf{S}}$ and the local Schur complements $\mathbf{S}^i$, function spaces $\widehat{W}$, W and W^i and operators R and R^i are introduced in a standard manner (see Mandel, Dohrmann and Tezaur [21]):

$\widehat{W}$ ... a space of functions with minimal energy on subdomains, continuous across the interface. Function $\widehat{u} \in \widehat{W}$ is represented by a vector $\widehat{\mathbf{u}}$ of values at global degrees of freedom at interface. Values inside subdomains can be then determined from (7.16). An example of a function from $\widehat{W}$ is in Fig. 7.8 right.
W^i ... a space of functions from $\widehat{W}$ restricted to Ω_i; $u_\mathrm{r}^i \in W^i$ is represented by a vector $\mathbf{u}_\mathrm{r}^i$ of values at local degrees of freedom at interface of Ω_i.
$W = W^1 \times W^2 \times \cdots \times W^N$ ... space of functions with minimal energy on subdomains, possible discontinuous ("teared apart") across the interface. Function $u_\mathrm{r} \in W$ is represented by a vector $\mathbf{u}_\mathrm{r}$ of values at union of independent instances of all local interface dofs from all subdomains (so for every global interface dof belonging to m subdomains there can be m different local values coming from different subdomains). An example of a function from W is in Fig. 7.9 bellow.
$\widehat{W}'$, $W^{i'}$ and W' ... dual spaces to $\widehat{W}$, W^i and W, respectively.

Schur complement operator $\widehat{S}$: $\widehat{W} \to \widehat{W}'$ is represented by Schur complement $\widehat{\mathbf{S}}$, local Schur complement operator S^i: $W^i \to W^{i'}$ is represented by local Schur complement $\mathbf{S}^i$.

In terms of mechanics, primal spaces $\widehat{W}$, W and W^i represent displacements resulting in zero reactions inside subdomains, dual spaces $\widehat{W}'$, W' and $W^{i'}$ represent nodal forces (or reactions) at interface. Spaces $\widehat{W}$ and $\widehat{W}'$ represent quantities continuous across the interface (displacements and nodal forces, respectively). Spaces W and W' represent these quantities discontinuous across the interface, it means that at interface nodes they can have different values coming from different subdomains containing this node. Space W^i represents displacements restricted to interface of Ω_i and space $W^{i'}$ represents reactions (node forces) at interface of Ω_i - as if we cut the subdomain Ω_i away from the rest of the domain and considered it independently.

R^i: $\widehat{W} \to W^i$ ... operator of a restriction from Ω to Ω_i. Operator R^i is represented by a matrix $\mathbf{R}^i$ that keeps only those components of a vector that belong to closure of Ω_i. Note that operator of a restriction R^i defined here keeps also interface components that lie on the boundary of Ω_i and so it differs from the operator of restriction in a context of overlapping subdomains defined in the section (7.4) that keeps only interior components of Ω_i.

$\mathrm{R}^{i\mathrm{T}}$: $W^{i'} \to \widehat{W}'$ ... operator of a prolongation from Ω_i to Ω. It is represented by a transpose $\mathbf{R}^{i\mathrm{T}}$ to the matrix $\mathbf{R}^i$, which takes a variable from Ω_i and represents it as the equivalent variable in Ω.

R: $\widehat{W} \to W$ ... operator of tearing interface unknowns to independent subdomains. It is represented by a matrix

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}^1 \\ \mathbf{R}^2 \\ \vdots \\ \mathbf{R}^N \end{bmatrix} \qquad (7.25)$$

which m-times copies every global unknown belonging to m subdomains.

R^T: $W' \to \widehat{W}'$ ... a transpose of the operator R. It is represented by a matrix $\mathbf{R}^\mathrm{T}$, which sums local interface values from adjacent subdomains.

Relations among global and local displacements can be expressed as $\mathbf{u}_\mathrm{r} = \mathbf{R}\widehat{\mathbf{u}}$ and $\mathbf{u}_\mathrm{r}^i = \mathbf{R}^i\,\widehat{\mathbf{u}}$.

The Schur complement $\widehat{\mathbf{S}}$ and the condensed right hand side $\widehat{\mathbf{g}}$ can be expressed as

$$\widehat{\mathbf{S}} = \mathbf{R}^{\mathrm{T}}\mathbf{S}\mathbf{R} = \sum \mathbf{R}^{i\mathrm{T}}\mathbf{S}^{i}\mathbf{R}^{i}, \tag{7.26}$$

$$\widehat{\mathbf{g}} = \mathbf{R}^{\mathrm{T}}\mathbf{g} = \sum \mathbf{R}^{i\mathrm{T}}\mathbf{g}^{i}, \tag{7.27}$$

where

$$\mathbf{g}^{i} = \mathbf{f}^{i}_{\mathrm{r}} - \mathbf{K}^{i}_{\mathrm{ro}}(\mathbf{K}^{i}_{\mathrm{oo}})^{-1}\mathbf{f}^{i}_{\mathrm{o}}, \qquad \widehat{\mathbf{f}} = \sum \mathbf{R}^{i\mathrm{T}}\mathbf{f}^{i}_{\mathrm{r}}. \tag{7.28}$$

In terms of mechanics, R copies values of displacements from global interface nodes to local interface nodes of all subdomains, R^i copies values of displacements from global interface nodes to local interface nodes of subdomain Ω_i, R^{T} sums local interface reactions from adjacent subdomain interfaces to create global interface node forces representing inbalance in reactions at interface and $\mathrm{R}^{i\mathrm{T}}$ expresses subdomain interface node reactions in Ω_i as global interface forces in Ω.

It would be nice if we could replace problem (7.12) by problem (7.23) consisting of N independent subdomain problems, but there are two difficulties preventing it.

First, local Schur complements $\mathbf{S}^i$ are not invertible on floating subdomains, so if a decomposition of Ω contains floating subdomains, problem (7.23) is not invertible.

Second, the decomposition (7.23) is not unique. Its right hand side $\mathbf{g}$ is determined by subdomain interface node forces $\mathbf{f}^i_{\mathrm{r}}$ via (7.28) left, whereas $\mathbf{f}^i_{\mathrm{r}}$ must represent a decomposition of the prescribed global interface node forces $\widehat{\mathbf{f}}$ given by the relationship (7.28) right. However, this relationship is not enough for specifying $\mathbf{f}^i_{\mathrm{r}}$ uniquely: there are many ways how to decompose global interface node forces $\widehat{\mathbf{f}}$ to subdomain interface node forces $\mathbf{f}^i_{\mathrm{r}}$ so that (7.28) holds. Only one choice is correct and leads to solution $\mathbf{u}_{\mathrm{r}}$ of (7.23) which is continuous across the interface (and so exists $\widehat{u}_{\mathrm{r}} \in \widehat{W}$ such that $\mathbf{R}\widehat{\mathbf{u}} = \mathbf{u}_{\mathrm{r}}$). For other choices the solution $\mathbf{u}_{\mathrm{r}}$ has different values on adjacent subdomains at the same global dofs.

Let us illustrate this uncertainty concerning the choice of the decomposition of $\widehat{\mathbf{f}}$ (and consequently $\widehat{\mathbf{g}}$, see relationships (7.27) and (7.28)) on the example from the Fig. 7.8. The correct decomposition of $\widehat{\mathbf{f}}$ leads to the solution in the Fig. 7.8 right, unfortunately this decomposition is not known in advance. Other decompositions lead to solutions that are not continuous across the interface, as the solution in the Fig. 7.9, which was obtained by distributing the global interface node forces a half to the first subdomain and a half to the second one.

7.5.4 Primal and Dual methods

Let us suppose in this section that every diagonal block of $\mathbf{S}$, formed by local Schur complement $\mathbf{S}^i$, is invertible (floating subdomains will be treated in the next section).

Both *primal* (*Neumann-Neumann*, *BDD* type) and *dual* (*FETI* type) methods are iterative methods for solving the Schur complement problem (7.12) using the decomposed problem (7.23). Only local Schur complement problems are solved, although repetitively.

7.5.4.1 Primal methods

The primal DD methods iterate on the primal space $\widehat{W}$. In order to clarify ideas, description of one iteration step in terms of mechanics is described first in Template 39. (An algebraic description follows in Template 40.)

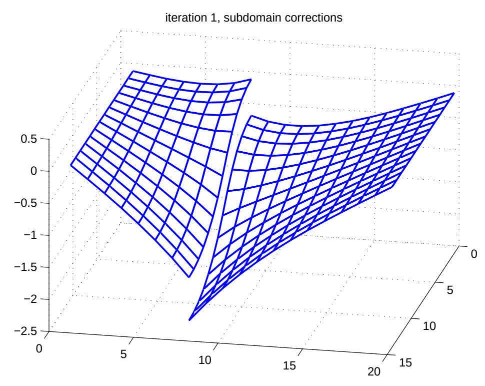


Figure 7.9: An example of a solution of local Schur complement problems (7.23): function with minimal energy on every subdomain, possibly discontinuous across the interface.

Template 39, Primal method in terms of mechanics

Let $\widehat{\mathbf{u}}$ be the last approximation of (global) interface displacements.

1. Compute residual $\widehat{\mathbf{r}} = \widehat{\mathbf{g}} - \widehat{\mathbf{S}}\widehat{\mathbf{u}}$ of (7.12) subdomain by subdomain:

   (a) Decouple all subdomains: for every subdomain, create a vector of local interface displacements ${\mathbf{u}_\mathrm{r}}^i$ as a restriction of $\widehat{\mathbf{u}}$ to Ω_i.

   (b) For every subdomain, compute local interface reactions $\mathbf{g}^i$ that correspond to local interface displacements: $\mathbf{g}^i = \mathbf{S}^i {\mathbf{u}_\mathrm{r}}^i$.

   (c) From local interface reactions, compute residual $\widehat{\mathbf{r}}$ at global interface nodes (as the global interface node forces minus a sum of local interface reactions at all local nodes corresponding to the given global node.

   Residual represents force inbalance at interface.

2. If residual $\widehat{\mathbf{r}}$ in a suitable norm is still large, distribute it back to subdomain interfaces: for every subdomain Ω_i create a vector of local interface reactions $\Delta\mathbf{r}^i$ that allocates to Ω_i its (approximate) share of a global inbalance represented by $\widehat{\mathbf{r}}$.

3. Find subdomain displacements corrections $\Delta{\mathbf{u}_\mathrm{r}}^i$ that would locally compensate for the local interface reactions: $\mathbf{S}^i\Delta{\mathbf{u}_\mathrm{r}}^i = \Delta\mathbf{r}^i$. This is the place where the assumption of invertibility of $\mathbf{S}^i$ is used.

4. Compute interface correction $\Delta\widehat{\mathbf{u}}$ as some average of subdomain corrections $\Delta{\mathbf{u}_\mathrm{r}}^i$. Subdomain corrections generally are not continuous across the interface, i.e. different values of displacement computed at different subdomains can exist for the same global interface node.

5. Take $\widehat{\mathbf{u}} + \Delta\widehat{\mathbf{u}}$ as the next approximation.

Let us consider the example in the Fig. 7.8 with the first approximation on the interface chosen as zero. Then the function in Fig. 7.8 center represents the first approximation of the solution on the whole domain. According to the Template 39, its residual is computed and distributed among the two subdomains and the correction is computed. If the residual was distributed among the subdomains in the right manner, the correction would be the solution of the Schur complement problem (Fig. 7.8 right) and the accurate solution would be obtained in the first iteration. Generally the proper distribution of the residual is not guessed correctly and so the resulting correction is discontinuous as in the Fig. 7.9. The continuity is then achieved by some averaging.

For algebraic description of the primal method algorithm in Template 40 we need to introduce an operator E for averaging displacements discontinuous across the interface:
E: $W \rightarrow \widehat{W}$ ... operator of averaging of interface values from adjacent subdomains; it is represented by a matrix $\mathbf{E}$
E^{T}: $\widehat{W}' \rightarrow W'$ ... operator of distributing of global interface forces to subdomains, represented by a transpose $\mathbf{E}^{\mathrm{T}}$ of the matrix $\mathbf{E}$.
The simple example of E is an arithmetic average: value at interface node is set as an arithmetic average of values at corresponding node from all subdomains containing that node. For more sofisticated choices of E see Mandel, Dohrmann and Tezaur [21] or Klawonn and Widlund [15]. The only limitation on E is that

$$\mathbf{E}\mathbf{R}\widehat{\mathbf{u}} = \widehat{\mathbf{u}} \tag{7.29}$$

holds for any $\widehat{u} \in \widehat{W}$. Consequently, for any $\widehat{g} \in \widehat{W}'$

$$\mathbf{R}^{\mathrm{T}}\mathbf{E}^{\mathrm{T}}\widehat{\mathbf{g}} = \widehat{\mathbf{g}}. \tag{7.30}$$

In terms of mechanics, relation (7.29) demands that, after copying global interface displacements $\widehat{\mathbf{u}}$ to local interface displacements by operator R and then averaging them back by operator E, we get the original displacements $\widehat{\mathbf{u}}$. Relation (7.30) states that as a consequence of (7.29), after distributing interface node forces $\widehat{\mathbf{g}}$ to subdomain interfaces using operator E^{T} and then summing them back by operator R^{T}, we get the original interface node forces $\widehat{\mathbf{g}}$.

Template 40, Algebraic description of the primal method algorithm

```
Choose $\widehat{\mathbf{u}}^{(0)}$ arbitrarily and repeat for $k := 0, 1, 2, \ldots$:

  1. (a) $\mathbf{u_r}^{(k)} := \mathbf{R}\,\widehat{\mathbf{u}}^{(k)}$
     (b) $\mathbf{g}^{(k)} := \mathbf{S}\mathbf{u_r}^{(k)}$
     (c) $\widehat{\mathbf{r}}^{(k)} := \widehat{\mathbf{g}} - \mathbf{R}^{\mathrm{T}}\mathbf{g}^{(k)}$
     if a suitable norm of $\widehat{\mathbf{r}}^{(k)}$ is small enough, stop

  2. $\Delta\mathbf{r}^{(k)} := \mathbf{E}^{\mathrm{T}}\widehat{\mathbf{r}}^{(k)}$

  3. $\Delta\mathbf{u_r}^{(k)} := \mathbf{S}^{-1}\,\Delta\mathbf{r}^{(k)}$

  4. $\Delta\widehat{\mathbf{u}}^{(k)} := \mathbf{E}\Delta\mathbf{u_r}^{(k)}$

  5. $\widehat{\mathbf{u}}^{(k+1)} := \widehat{\mathbf{u}}^{(k)} + \Delta\widehat{\mathbf{u}}^{(k)}$
```

Putting the last four steps of the algorithm in Template 40 together, we get a mathematical formulation of the primal (Neumann-Neumann) method as a Richardson method for the Schur complement problem (7.12):

$$\widehat{\mathbf{u}}^{(k+1)} = \widehat{\mathbf{u}}^{(k)} + \mathbf{E}\mathbf{S}^{-1}\mathbf{E}^{\mathrm{T}}\widehat{\mathbf{r}}^{(k)}. \tag{7.31}$$

Implementation remarks:

Matrices $\mathbf{E}$, $\mathbf{R}$ and their transposes are used just for description of the algorithm, they are not expected to be explicitly constructed in programs.

Computations at the steps (1b) and (3) of the algorithm in the Template 40 can be realized locally subdomain by subdomain due to block-diagonal structure of $\mathbf{S}$ as $\mathbf{g}^i := \mathbf{S}^i{\mathbf{u}_{\mathrm{r}}}^i$ and $\Delta{\mathbf{u}_{\mathrm{r}}}^i := (\mathbf{S}^i)^{-1}\,\Delta\mathbf{r}^i$, respectively (superscript $^{(k)}$ was omitted for simplicity).

Moreover, local Schur complements $\mathbf{S}^i$ need not be stored nor factorized explicitly. Multiplication of a vector $\mathbf{v}$ by $\mathbf{S}^i$, needed at the step of (1b), can be implemented by solving the Dirichlet problem (7.16) on subdomain using existing block factorization of $\mathbf{K}_{\mathrm{oo}}$: $\mathbf{S}^i\mathbf{v} = (\mathbf{K}^i_{\mathrm{rr}} - \mathbf{K}^i_{\mathrm{ro}}(\mathbf{K}^i_{\mathrm{oo}})^{-1}\mathbf{K}^i_{\mathrm{or}})\mathbf{v} = \mathbf{K}^i_{\mathrm{rr}}\mathbf{v} - \mathbf{K}^i_{\mathrm{ro}}\mathbf{v}_{\mathrm{or}}$, where $\mathbf{v}_{\mathrm{or}}$ is a solution of $\mathbf{K}^i_{\mathrm{oo}}\mathbf{v}_{\mathrm{or}} = -\mathbf{K}^i_{\mathrm{or}}\mathbf{v}$, see also implementation remark bellow the Schur complement algorithm in Template 38.

A solution $\mathbf{v}$ of a problem $\mathbf{S}^i\mathbf{v} = \Delta\mathbf{r}^i$ needed at the step (3) can be implemented by solving the Neumann problem (7.19) on subdomain with $\mathbf{f}^i_{\mathrm{r}} = \Delta\mathbf{r}^i$ and $\mathbf{f}^i_{\mathrm{o}} = \mathbf{0}$ and using just an interface part $\mathbf{u}^i_{\mathrm{r}}$ of the solution. For $\mathbf{f}^i_{\mathrm{o}} = \mathbf{0}$, problem (7.19) is equivalent to the problem (7.22) with $\mathbf{g}^i = \mathbf{f}^i_{\mathrm{r}}$.

7.5.4.2 Dual methods

The dual DD methods iterate on the dual space W' or, strictly speaking, on a space of Lagrange multipliers, as will be shown later. The correct decomposition of $\widehat{\mathbf{g}}$ is to be found such that simultaneously a solution $\mathbf{u}_{\mathrm{r}}$ of (7.23) is continuous across the interface and (7.27) holds. Relation (7.27) can be expressed as

$$\widehat{\mathbf{g}} = \mathbf{R}^{\mathrm{T}}(\mathbf{g} - \mathbf{r}), \tag{7.32}$$

for any $\mathbf{r}$ satisfying

$$\mathbf{R}^{\mathrm{T}}\mathbf{r} = \mathbf{0}, \tag{7.33}$$

whereas $\mathbf{g}$ is an arbitrarily chosen decomposition of $\widehat{\mathbf{g}}$ satisfying (7.27). For instance we can set $\mathbf{g} = \mathbf{E}^{\mathrm{T}}\widehat{\mathbf{g}}$, then (7.27) follows from (7.30). Vector $\mathbf{r}$ can be interpreted as local interface reactions and equation (7.33) states that interface reactions from adjacent subdomains must annulate each other. Problem (7.12) can be formulated using (7.23) and (7.32) as to find subdomain interface reactions $\mathbf{r}$ such that (7.33) holds and a problem

$$\mathbf{S}\mathbf{u}_{\mathrm{r}} = \mathbf{g} - \mathbf{r} \tag{7.34}$$

has solution $\mathbf{u}_{\mathrm{r}}$ with no jumps across the interface.

In order to clarify ideas, description of one iteration step of a dual method in terms of mechanics is described first in Template 41.

Template 41, Dual method in terms of mechanics

Let $\mathbf{g}$ be a decomposition of $\widehat{\mathbf{g}}$ satisfying (7.27). Let $\mathbf{r}$ be the last approximation of local interface reactions satisfying (7.33) i.e. they cancel each other.

1. Compute local interface displacements ${\mathbf{u}_\mathrm{r}}^i$ corresponding to the local interface reactions: $\mathbf{S}^i {\mathbf{u}_\mathrm{r}}^i = \mathbf{g}^i - \mathbf{r}^i$. This is the place where the assumption of invertibility of $\mathbf{S}^i$ is used.

2. Compute the jump across the interface in local interface displacements of adjacent subdomains. If there is no jump, ${\mathbf{u}_\mathrm{r}}^i$ represent solution of (7.12).

3. Distribute the jump among subdomains: for every subdomain Ω_i create a vector of local interface displacements $\Delta{\mathbf{u}_\mathrm{r}}^i$ that allocates to Ω_i its (approximate) share of the global interface jump.

4. Find subdomain corrections of reactions $\Delta\mathbf{r}^i$ that would locally compensate for the local interface jump: $\mathbf{S}^i \Delta{\mathbf{u}_\mathrm{r}}^i = \Delta\mathbf{r}^i$.

5. Take $\mathbf{r}^i + \Delta\mathbf{r}^i$ as the next approximation.

For algebraic description of the dual algorithm we need to introduce a vector space Λ of Lagrange multipliers that represents jumps in values of functions from W, its dual space Λ' representing jumps in W', and two operators B and $\mathrm{B_D}$ for computing jumps. In terms of mechanics, Λ represents jumps in displacements across the interface between adjacent subdomains, characterized by red strokes in the Fig. 7.10, Λ' represents jumps (inbalance) in reactions across the interface.

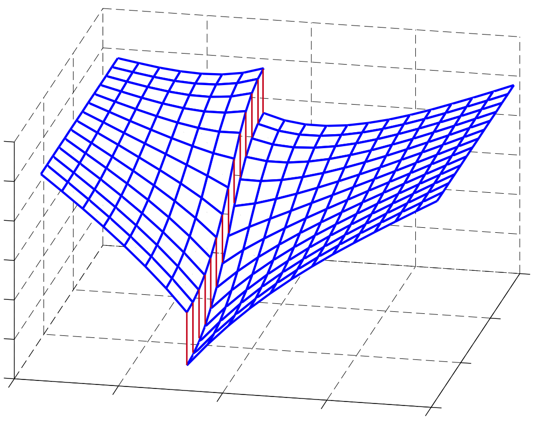


Figure 7.10: Jumps in displacements across the interface between adjacent subdomains, represented by the red strokes.

B: $W \to \Lambda$ ... operator which computes interface jumps in displacements in W; it is represented by a matrix $\mathbf{B}$
$\mathrm{B_D}$: $W' \to \Lambda'$ ... operator which computes interface jumps in reactions in W' as a weighted sum of local interface reactions; it is represented by a matrix $\mathbf{B_D}$
B^T: $\Lambda' \to W'$ ... expresses an interface jumps in reactions as some function in W' with the same jumps across the interface; it is represented by a transpose $\mathbf{B}^\mathrm{T}$ to the matrix

B
$\mathrm{B}_\mathrm{D}^\mathrm{T}$: $\Lambda \rightarrow W$ ... expresses an interface jumps in displacements as some function in W with the same jumps across the interface; it is represented by a transpose $\mathbf{B}_\mathrm{D}^\mathrm{T}$ to the matrix $\mathbf{B}_\mathrm{D}$.

In terms of mechanics, in case an interface node is common to just two adjacent subdomains, a jump in displacements is computed as a difference of local displacements at this node, a jump in reactions is computed as a weighted sum of local interface reactions at this node.

The only limitation on a choice of B_D is that

$$\mathbf{B}_\mathrm{D}\mathbf{B}^\mathrm{T}\boldsymbol{\lambda}' = \boldsymbol{\lambda}' \tag{7.35}$$

holds for any $\lambda' \in \Lambda'$. Consequently, for any $\lambda \in \Lambda$

$$\mathbf{B}\mathbf{B}_\mathrm{D}^\mathrm{T}\boldsymbol{\lambda} = \boldsymbol{\lambda}. \tag{7.36}$$

In terms of mechanics, relation (7.35) just demands that if we represent given jump λ' by B^T as interface node reactions and then determine jump in these reactions using B_D, we obtain the original jump λ'. Relation (7.36) states that as a consequence of (7.35), if we use B for computing jump in displacements determined by $\mathrm{B}_\mathrm{D}^\mathrm{T}$ as a representation of a given jump λ in displacements, we get the same jump λ.

Functions from the space $\widehat{W}$ (displacements continuous across the interface) can be represented in the space W (displacements with possible jumps across the interface) as functions with zero jumps across the interface or, by other words, as a null space of operator B: $\widehat{u} \in \widehat{W}$ can be identified with $u_\mathrm{r} \in W$ such that

$$\mathbf{B}\mathbf{u}_\mathrm{r} = \mathbf{0}. \tag{7.37}$$

Functions from the space $\widehat{W}'$ ("balanced" reactions across the interface) can be represented in space W' (reactions with possible jumps across the interface) as functions with zero jumps across the interface or as a null space of operator B_D: $\widehat{g} \in \widehat{W}'$ can be identified with $g \in W'$ such that $\mathbf{B}_\mathrm{D}\mathbf{g} = \mathbf{0}$.

Algebraic description of the dual method algorithm in reactions is given by Template 42. We are seeking $\mathbf{r}$ satisfying (7.33) such that the solution $\mathbf{u}_\mathrm{r}$ of the problem (7.34) has zero jump across the interface, i.e. it satisfies (7.37).

Template 42, Algebraic description of the dual method algorithm in reactions

```
Choose $\mathbf{g} := \mathbf{E}^\mathrm{T}\widehat{\mathbf{g}}$ and $\mathbf{r}^{(0)} := \mathbf{0}$ and repeat for $k := 0, 1, 2, \ldots$:
    1. ${\mathbf{u}_\mathrm{r}}^{(k)} := \mathbf{S}^{-1}(\mathbf{g} - \mathbf{r}^{(k)})$
    2. $\boldsymbol{\lambda}^{(k)} := \mathbf{B}{\mathbf{u}_\mathrm{r}}^{(k)}$ , if $\boldsymbol{\lambda}^{(k)}$ is small enough, stop
    3. $\Delta{\mathbf{u}_\mathrm{r}}^{(k)} := \mathbf{B}_\mathrm{D}^\mathrm{T}\boldsymbol{\lambda}^{(k)}$
    4. $\Delta\mathbf{r}^{(k)} := \mathbf{S}\Delta{\mathbf{u}_\mathrm{r}}^{(k)}$
    5. $\mathbf{r}^{(k+1)} := \mathbf{r}^{(k)} + \Delta\mathbf{r}^{(k)}$
```

Putting the last three steps of algorithm described by Template 42 together, we get

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} + \mathbf{S}\mathbf{B}_\mathrm{D}^\mathrm{T}\boldsymbol{\lambda}^{(k)}. \tag{7.38}$$

More suitable mathematical formulation of the dual method is in terms of jumps in reactions, rather than in terms of the reactions themselves. A condition that the solution $\mathbf{u}_\mathrm{r}$ of (7.34) has no jumps can be expressed as:

$$\mathbf{0} = \mathbf{B}\mathbf{u}_\mathrm{r} = \mathbf{B}\mathbf{S}^{-1}(\mathbf{g} - \mathbf{r}). \tag{7.39}$$

Using operator B^T, reactions $\mathbf{r}$ can be expressed as representing some jump λ' in reactions across the interface:

$$\mathbf{r} = \mathbf{B}^\mathrm{T}\boldsymbol{\lambda}'. \tag{7.40}$$

Substituting (7.40) into (7.39), we obtain a problem (7.34) reformulated for unknown jumps in reactions across the interface:

$$\mathbf{B}\mathbf{S}^{-1}\mathbf{B}^\mathrm{T}\boldsymbol{\lambda}' = \mathbf{B}\mathbf{S}^{-1}\mathbf{g}. \tag{7.41}$$

Residual of the problem (7.41) for a given $\boldsymbol{\lambda}'$ is a vector $\boldsymbol{\lambda} = \mathbf{B}\mathbf{S}^{-1}(\mathbf{g} - \mathbf{B}^\mathrm{T}\boldsymbol{\lambda}')$ which represents jumps in displacements $\mathbf{u}$ obtained from (7.34) after substitution of (7.40) to the right hand side.

Substitution of (7.40) to the algorithm formulated in reactions $\mathbf{r}$ in Template 42 leads to algorithm formulated in Lagrange multipliers $\boldsymbol{\lambda}'$ in Template 43.

Template 43, Algebraic description of the dual algorithm in Lagrange multipliers

Choose $\mathbf{g} := \mathbf{E}^\mathrm{T}\widehat{\mathbf{g}}$ and $\boldsymbol{\lambda}'^{(0)} := \mathbf{0}$ and repeat for $k := 0, 1, 2, \ldots$:

```
1. $\mathbf{u}_\mathrm{r}^{(k)} := \mathbf{S}^{-1}(\mathbf{g} - \mathbf{B}^\mathrm{T}\boldsymbol{\lambda}'^{(k)})$
2. $\boldsymbol{\lambda}^{(k)} := \mathbf{B}\mathbf{u}_\mathrm{r}^{(k)}$ , if $\boldsymbol{\lambda}^{(k)}$ is small, stop
3. $\Delta\mathbf{u}_\mathrm{r}^{(k)} := \mathbf{B}_\mathrm{D}^\mathrm{T}\boldsymbol{\lambda}^{(k)}$
4. $\Delta\boldsymbol{\lambda}'^{(k)} := \mathbf{B}_\mathrm{D}\mathbf{S}\Delta\mathbf{u}_\mathrm{r}^{(k)}$
5. $\boldsymbol{\lambda}'^{(k+1)} := \boldsymbol{\lambda}'^{(k)} + \Delta\boldsymbol{\lambda}'^{(k)}$
```

Putting the last three steps of algorithm described by Template 43 together, we get a mathematical formulation of the dual method as a Richardson method for the problem (7.41):

$$\boldsymbol{\lambda}'^{(k+1)} = \boldsymbol{\lambda}'^{(k)} + \mathbf{B}_\mathrm{D}\mathbf{S}\mathbf{B}_\mathrm{D}^\mathrm{T}\boldsymbol{\lambda}^{(k)}. \tag{7.42}$$

Implementation remarks:

Matrices $\mathbf{E}$, $\mathbf{B}$, $\mathbf{B}_\mathrm{D}$ and their transposes are used just for description of the algorithm, they are not expected to be explicitly constructed in programs.

Computations with $\mathbf{S}$ can be realized locally subdomain by subdomain due to block-diagonal structure of $\mathbf{S}$; moreover, local Schur complements $\mathbf{S}^i$ need not be stored nor factorized explicitly. See also *Implementation remarks* bellow the Primal method algorithm in Template 40.

Remark:

If K is a symmetric positive definite operator, then S is a symmetric positive semidefinite operator and problem (7.23) constrained by (7.37) is equivalent to minimization

$$\frac{1}{2}\mathbf{u}_\mathrm{r}{}^\mathrm{T}\mathbf{S}\mathbf{u}_\mathrm{r} - \mathbf{u}_\mathrm{r}{}^\mathrm{T}\mathbf{g} \rightarrow \mathit{min} \quad \text{subject to} \quad \mathbf{B}\mathbf{u}_\mathrm{r} = \mathbf{0}. \tag{7.43}$$

Then problem (7.41) can be derived from (7.43) by using Lagrange multipliers for formulating a saddle point problem equivalent to (7.43)

$$\begin{bmatrix} \mathbf{S} & \mathbf{B}^{\mathrm{T}} \\ \mathbf{B} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{r}} \\ \lambda' \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ \mathbf{0} \end{bmatrix} \tag{7.44}$$

and after eliminating primal unknowns $\mathbf{u}_{\mathrm{r}}$:

$$\begin{bmatrix} \mathbf{S} & \mathbf{B}^{\mathrm{T}} \\ \mathbf{0} & -\mathbf{B}\mathbf{S}^{-1}\mathbf{B}^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{r}} \\ \lambda' \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ -\mathbf{B}\mathbf{S}^{-1}\mathbf{g} \end{bmatrix}, \tag{7.45}$$

problem (7.41) for unknown λ' immediately follows.

Relation between primal and dual methods
Primal and dual methods are very closely related. Until now no relation between the choice of an operator $\mathrm{B_D}$ for computing a weighted sum of reactions and an averaging operator E was demanded. Let us assume now that

$$\mathbf{B}_{\mathrm{D}}^{\mathrm{T}}\mathbf{B}\mathbf{u}_{\mathrm{r}} + \mathbf{R}\mathbf{E}\mathbf{u}_{\mathrm{r}} = \mathbf{u}_{\mathrm{r}} \tag{7.46}$$

holds for every $u_{\mathrm{r}} \in W$. This means that a function $u_{\mathrm{r}} \in W$ with jumps across the interface is decomposed to a sum of some averaged function $\mathrm{RE}u_{\mathrm{r}}$ continuous across the interface and some function $\mathrm{B}_{\mathrm{D}}^{\mathrm{T}}\mathrm{B}u_{\mathrm{r}}$ representing a jump of the u_{r} across the interface. If u_{r} is continuous across the interface, then $u_{\mathrm{r}} = \mathrm{RE}u_{\mathrm{r}}$ and $\mathrm{B}u_{\mathrm{r}} = 0$ (and consequently $\mathrm{B}_{\mathrm{D}}^{\mathrm{T}}\mathrm{B}u_{\mathrm{r}} = 0$). Under assumption of (7.46), RE and $\mathrm{B}_{\mathrm{D}}^{\mathrm{T}}\mathrm{B}$ are complementary projections on W and corresponding primal and dual method are in a sense complementary, too. Primal method seeks solution $\mathbf{u}_{\mathrm{r}}$ of the problem (7.23) such that $\mathbf{u}_{\mathrm{r}} = \mathbf{R}\mathbf{E}\mathbf{u}_{\mathrm{r}}$ and consequently $\mathbf{B}_{\mathrm{D}}^{\mathrm{T}}\mathbf{B}\mathbf{u}_{\mathrm{r}} = \mathbf{0}$. Dual method seeks solution $\mathbf{u}_{\mathrm{r}}$ of the problem (7.23) such that $\mathbf{B}\mathbf{u}_{\mathrm{r}} = \mathbf{0}$ and consequently $\mathbf{u}_{\mathrm{r}} = \mathbf{R}\mathbf{E}\mathbf{u}_{\mathrm{r}}$. For more details on relationship between primal and dual methods and their similar convergence properties see Mandel, Dohrmann and Tezaur [21] or Klawonn and Widlund [15].

7.5.5 BDDC and FETI-DP

Primal (Neumann-Neumann) and dual (FETI type) methods, as described in the previous section, have two main drawbacks. First, no floating subdomains are allowed in order to have local Schur complements invertible. Second, there is no global communication as in each iteration step information is exchanged between neighbouring subdomains only. This leads to deteriorating of the convergence rate with growing number of subdomains.

There have been many different attempts to tackle the first drawback. Let us mention just two successful methods from early 1990s, the FETI method by Farhat and Roux [9] and the BDD method by Mandel [19].

Most advanced recent methods seem to be the BDDC (Balancing Domain Decomposition by Constraints) developed by Dohrmann [5] and the FETI-DP (FETI Dual-Primal) introduced by Farhat et al. [8]. Both methods are described and compared in an abstract algebraic setting in Mandel, Dohrmann and Tezaur [21] and in Mandel and Sousedík [22], or in a functional analytic framework in Brenner and Sung [4].

Both BDDC and FETI-DP methods construct a new space $\widetilde{W} \subset W$ by imposing some continuity constraints across the interface in *coarse degrees of freedom (coarse unknowns)* so that $\widehat{W} \subset \widetilde{W}$ and $\mathbf{S}$ from (7.23) restricted to $\widetilde{W}$ is invertible. Then methods from previous section are used on $\widetilde{W}$ instead of W: primal method in the case of BDDC and dual method in the case of FETI-DP.

It was shown that a smart choice of the coarse degrees of freedom resolves even the second drawback. It can not only improve convergence properties, but also make convergence independent of the number of subdomains (see Toselli and Widlund [27] or Mandel and Dohrmann [20]).

The coarse degrees of freedom

A choice of the coarse dofs usually starts by a selection of some nodes on the interface as *coarse nodes*, typically corners of subdomains are chosen first and then other nodes are added as needed. Values at coarse nodes are used as coarse dofs. $\widetilde{W} \subset W$ consists of functions continuous across the interface at coarse nodes, represented by functions in W for which values at coarse dofs coincide. There are two examples of functions from $\widetilde{W}$ in Fig. 7.11. Coarse dofs need not be teared apart to subdomains, because values at coarse dofs do not differ in adjacent subdomains.

For better convergence properties not only values at coarse nodes are used as coarse dofs, but also weighted averages of values over edges and faces of adjacent subdomains. More details can be found for instance in Mandel and Dohrmann [20], Klawonn, Widlund and Dryja [16], Klawonn and Rheinbach [14] or in Li and Widlund [18], where a change of variables is used for treating averages so that each average corresponds to an explicit degree of freedom, like a coarse node.

For simplicity let us assume now that coarse dofs are values at coarse nodes only, so we can divide all the interface unknowns to *coarse unknowns* and the rest.

Let us denote the system (7.23) restricted to $\widetilde{W}$ as

$$\widetilde{\mathbf{S}}\widetilde{\mathbf{u}} = \widetilde{\mathbf{g}} \tag{7.47}$$

and rewrite it in a block form with the last block corresponding to all the coarse unknowns:

$$\begin{bmatrix} \mathbf{S}_{\mathrm{rr}} & \mathbf{S}_{\mathrm{rc}} \\ \mathbf{S}_{\mathrm{cr}} & \widehat{\mathbf{S}}_{\mathrm{c}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{r}} \\ \widehat{\mathbf{u}}_{\mathrm{c}} \end{bmatrix} = \begin{bmatrix} \mathbf{g}_{\mathrm{r}} \\ \widehat{\mathbf{g}}_{c} \end{bmatrix}, \tag{7.48}$$

where a symbol $\widehat{\ }$ denotes coarse (continuous) quantities of the problem, which are the same as in the original Schur complement problem (7.12). $\mathbf{S}_{\mathrm{rr}}$, $\mathbf{u}_{\mathrm{r}}$ and $\mathbf{g}_{\mathrm{r}}$ represent "teared" interface quantities and can be written in a block form analogous to (7.24):

$$\mathbf{S}_{\mathrm{rr}} = \begin{bmatrix} \widetilde{\mathbf{S}}^1 & & & \\ & \widetilde{\mathbf{S}}^2 & & \\ & & \ddots & \\ & & & \widetilde{\mathbf{S}}^N \end{bmatrix}, \qquad \mathbf{u}_{\mathrm{r}} = \begin{bmatrix} \widetilde{\mathbf{u}}_{\mathrm{r}}^1 \\ \widetilde{\mathbf{u}}_{\mathrm{r}}^2 \\ \vdots \\ \widetilde{\mathbf{u}}_{\mathrm{r}}^N \end{bmatrix}, \qquad \mathbf{g}_{\mathrm{r}} = \begin{bmatrix} \widetilde{\mathbf{g}}^1 \\ \widetilde{\mathbf{g}}^2 \\ \vdots \\ \widetilde{\mathbf{g}}^N \end{bmatrix}, \tag{7.49}$$

where $\widetilde{\mathbf{u}}_{\mathrm{r}}^i$ consists of interface unknowns of Ω_i that are not coarse. Coarse unknowns $\widehat{\mathbf{u}}_{\mathrm{c}}$ cannot be separated this way, as every of them belongs to two or more subdomains.

Coarse dofs must be chosen so that $\widetilde{\mathbf{S}}$ and $\mathbf{S}_{\mathrm{rr}}$ are invertible (and consequently every subdomain matrix $\widetilde{\mathbf{S}}^i$ is invertible). This means that no subdomain nor group of subdomains can be left floating; all subdomains must be sufficiently interconnected by coarse dofs.

If all the interface unknowns are selected as coarse, then $\widehat{\mathbf{S}}_{\mathrm{c}} = \widehat{\mathbf{S}}$, $\widehat{\mathbf{u}}_{\mathrm{c}} = \widehat{\mathbf{u}}$ and $\widehat{\mathbf{g}}_{\mathrm{c}} = \widehat{\mathbf{g}}$ and the problem (7.48) become the original problem (7.12). On the other hand, if no coarse unknowns are selected, then $\widetilde{\mathbf{S}} = \mathbf{S}$, $\widetilde{\mathbf{u}} = \mathbf{u}_\mathrm{r}$ and $\widetilde{\mathbf{g}} = \mathbf{g}_\mathrm{r}$ and the problem (7.48) become the totally decomposed problem (7.23).

Let us denote
$\widetilde{\mathrm{R}}^\mathrm{T}$: $\widetilde{W}' \to \widehat{W}'$ ... a restriction of operator R^T to $\widetilde{W}'$,
$\widetilde{\mathrm{R}}$: $\widehat{W} \to \widetilde{W}$ ... transpose of the operator $\widetilde{\mathrm{R}}^\mathrm{T}$,
$\widetilde{\mathrm{E}}$: $\widetilde{W} \to \widehat{W}$ ... a restriction of an average operator E to $\widetilde{W}$.
In the case the coarse dofs are values at coarse nodes only, operators $\widetilde{\mathrm{R}}$ and $\widetilde{\mathrm{E}}$ written in a block form have a block diagonal structure:

$$\widetilde{\mathbf{R}} = \begin{bmatrix} \mathbf{R}_\mathrm{r} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \widetilde{\mathbf{R}}_\mathrm{r} \\ \mathbf{R}_\mathrm{c} \end{bmatrix}, \qquad \widetilde{\mathbf{E}} = \begin{bmatrix} \mathbf{E}_\mathrm{r} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \widetilde{\mathbf{E}}_\mathrm{r} & \mathbf{R}_\mathrm{c}^\mathrm{T} \end{bmatrix}, \tag{7.50}$$

where matrix $\mathbf{R}_\mathrm{c}$ keeps only coarse unknowns of a vector, $\widetilde{\mathbf{R}}_\mathrm{r}$ tears apart only unknowns that are not coarse and similarly $\widetilde{\mathbf{E}}_\mathrm{r}$ averages only unknowns that are not coarse. If all the interface unknowns are selected as coarse, then $\widetilde{W} = \widehat{W}$ and $\widetilde{\mathrm{R}} = \widetilde{\mathrm{E}} = \mathrm{I}$ (identity operator), if no coarse unknowns are selected, then $\widetilde{W} = W$, $\widetilde{\mathrm{R}} = \mathrm{R}$ and $\widetilde{\mathrm{E}} = \mathrm{E}$.

The Schur complement $\widehat{\mathbf{S}}$ can be expressed as

$$\widehat{\mathbf{S}} = \widetilde{\mathbf{R}}^\mathrm{T}\widetilde{\mathbf{S}}\widetilde{\mathbf{R}} = \begin{bmatrix} \mathbf{R}_\mathrm{r}^\mathrm{T}\mathbf{S}_\mathrm{rr}\mathbf{R}_\mathrm{r} & \mathbf{R}_\mathrm{r}^\mathrm{T}\mathbf{S}_\mathrm{rc} \\ \mathbf{S}_\mathrm{cr}\mathbf{R}_\mathrm{r} & \widehat{\mathbf{S}}_\mathrm{c} \end{bmatrix} \tag{7.51}$$

and for relations among $\widehat{\mathbf{g}}$, $\widehat{\mathbf{u}}$ and $\widetilde{\mathbf{g}}$, $\widetilde{\mathbf{u}}$ it holds

$$\widehat{\mathbf{g}} = \widetilde{\mathbf{R}}^\mathrm{T}\widetilde{\mathbf{g}} = \begin{bmatrix} \mathbf{R}_\mathrm{r}^\mathrm{T}\mathbf{g}_\mathrm{r} \\ \widehat{\mathbf{g}}_\mathrm{c} \end{bmatrix}, \qquad \widetilde{\mathbf{u}} = \widetilde{\mathbf{R}}\widehat{\mathbf{u}} = \begin{bmatrix} \mathbf{R}_\mathrm{r}\widehat{\mathbf{u}}_\mathrm{r} \\ \widehat{\mathbf{u}}_\mathrm{c} \end{bmatrix}. \tag{7.52}$$

7.5.5.1 BDDC method

The BDDC algorithm (Template 44) is the primal algorithm (Template 40) rewritten for partially decomposed problem (7.47) instead of totally decomposed problem (7.23).
Template 44, The BDDC method algorithm

```
Choose $\widehat{\mathbf{u}}^{(0)}$ arbitrarily and repeat for $k := 0, 1, 2, \ldots$:
  1. (a) $\widetilde{\mathbf{u}}^{(k)} := \widetilde{\mathbf{R}}\,\widehat{\mathbf{u}}^{(k)}$
     (b) $\widetilde{\mathbf{g}}^{(k)} := \widetilde{\mathbf{S}}\,\widetilde{\mathbf{u}}^{(k)}$
     (c) $\widehat{\mathbf{r}}^{(k)} := \widehat{\mathbf{g}} - \widetilde{\mathbf{R}}^\mathrm{T}\,\widetilde{\mathbf{g}}^{(k)}$
     if $\widehat{\mathbf{r}}^{(k)}$ is small enough, stop
  2. $\Delta\widetilde{\mathbf{r}}^{(k)} := \widetilde{\mathbf{E}}^\mathrm{T}\,\widehat{\mathbf{r}}^{(k)}$
  3. $\Delta\widetilde{\mathbf{u}}^{(k)} := (\widetilde{\mathbf{S}})^{-1}\,\Delta\widetilde{\mathbf{r}}^{(k)}$
  4. $\Delta\widehat{\mathbf{u}}^{(k)} := \widetilde{\mathbf{E}}\,\Delta\widetilde{\mathbf{u}}^{(k)}$
  5. $\widehat{\mathbf{u}}^{(k+1)} := \widehat{\mathbf{u}}^{(k)} + \Delta\widehat{\mathbf{u}}^{(k)}$
```

Putting the last four steps of this algorithm together, we get a mathematical formulation of the BDDC method as a Richardson method for the Schur complement problem

(7.12):

$$\widehat{\mathbf{u}}^{(k+1)} = \widehat{\mathbf{u}}^{(k)} + \widetilde{\mathbf{E}}(\widetilde{\mathbf{S}})^{-1}\widetilde{\mathbf{E}}^{\mathrm{T}}\,\widehat{\mathbf{r}}^{(k)}. \tag{7.53}$$

Remark:

Computation of the residual at the step 1 of the Template 44 could also be performed with fully decomposed Schur complement as in the step 1 of the basic primal algorithm in the Template 40.

Let us consider the example in the Fig. 7.8 with the first approximation on the interface chosen as zero. Then the function in Fig. 7.8 center represents the first approximation of the solution on the whole domain. Its residual is computed and distributed among the two subdomains and the correction is computed, see Fig. 7.11, where 2 (left) or 3 (right) coarse nodes were selected. It differs by the continuity at the coarse nodes from the correction depicted in the Fig. 7.9 of the basic primal method algorithm described in Template 39.

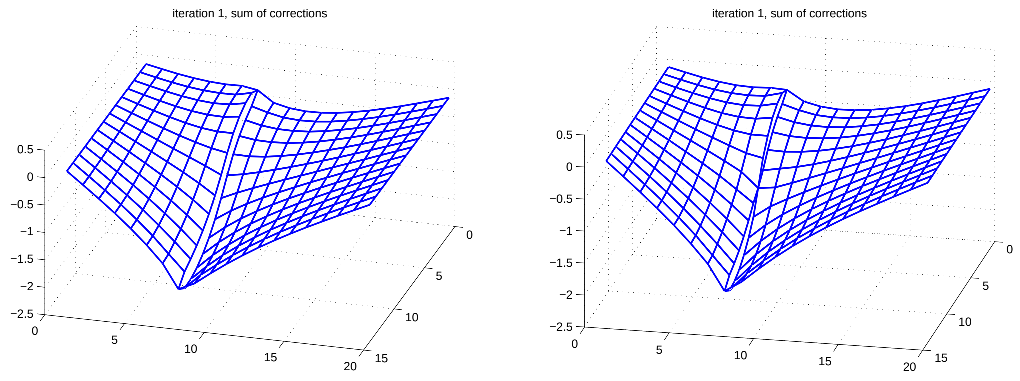


Figure 7.11: An example of subdomain displacements corrections for two subdomains: function with minimal energy on every subdomain, continuous at the coarse nodes and possibly discontinuous on the rest of the interface. Left: 2 coarse nodes (the corners). Right: 3 coarse nodes (the corners and the middle-point).

7.5.5.2 FETI-DP method

The FETI-DP algorithm (Template 45) is the dual algorithm (Template 43) rewritten for partially decomposed problem (7.47) instead of totally decomposed problem (7.23). Consequently a problem

$$\widetilde{\mathbf{B}}\widetilde{\mathbf{S}}^{-1}\widetilde{\mathbf{B}}^{\mathrm{T}}\lambda' = \widetilde{\mathbf{B}}\widetilde{\mathbf{S}}^{-1}\widetilde{\mathbf{g}} \tag{7.54}$$

is solved instead of (7.41), whereas $\widetilde{\mathrm{B}}$: $\widetilde{W} \to \Lambda$ is a restriction of the operator B to $\widetilde{W}$. Let $\widetilde{\mathrm{B}}_{\mathrm{D}}$: $\widetilde{W'} \to \Lambda'$ be a restriction of the operator B_{D} to $\widetilde{W'}$.

Residual of the problem (7.54) for a given $\boldsymbol{\lambda}'$ is a vector $\boldsymbol{\lambda} = \widetilde{\mathbf{B}}\widetilde{\mathbf{S}}^{-1}(\widetilde{\mathbf{g}} - \widetilde{\mathbf{B}}^{\mathrm{T}}\boldsymbol{\lambda}')$ which represents jumps across the interface at displacements $\mathbf{u}_{\mathrm{r}}$ (there are no jumps at the coarse displacements $\widehat{\mathbf{u}}_{\mathrm{c}}$).

Template 45, The FETI-DP algorithm

Choose $\widetilde{\mathbf{g}} := \widetilde{\mathbf{E}}^{\mathrm{T}}\,\widehat{\mathbf{g}}$ and $\boldsymbol{\lambda}'^{(0)} := \mathbf{0}$ and repeat for $k := 0, 1, 2, \ldots$:

1. $\widetilde{\mathbf{u}}^{(k)} := (\widetilde{\mathbf{S}})^{-1}(\widetilde{\mathbf{g}} - \widetilde{\mathbf{B}}^{\mathrm{T}}\boldsymbol{\lambda}'^{(k)})$
2. $\boldsymbol{\lambda}^{(k)} := \widetilde{\mathbf{B}}\,\widetilde{\mathbf{u}}^{(k)}$, if $\boldsymbol{\lambda}^{(k)}$ is small enough, stop
3. $\Delta\mathbf{u}^{(k)} := \widetilde{\mathbf{B}}_{\mathbf{D}}^{\mathrm{T}}\,\boldsymbol{\lambda}^{(k)}$
4. $\Delta\boldsymbol{\lambda}'^{(k)} := \widetilde{\mathbf{B}}_{\mathbf{D}}\widetilde{\mathbf{S}}\,\Delta\mathbf{u}^{(k)}$
5. $\boldsymbol{\lambda}'^{(k+1)} := \boldsymbol{\lambda}'^{(k)} + \Delta\boldsymbol{\lambda}'^{(k)}$

Putting the last three steps of this algorithm together, we get a mathematical formulation of the FETI-DP method as a Richardson method for the problem (7.41):

$$\boldsymbol{\lambda}'^{(k+1)} = \boldsymbol{\lambda}'^{(k)} + \widetilde{\mathbf{B}}_{\mathbf{D}}\widetilde{\mathbf{S}}\,\widetilde{\mathbf{B}}_{\mathbf{D}}^{\mathrm{T}}\,\boldsymbol{\lambda}^{(k)}. \tag{7.55}$$

7.5.5.3 Coarse space and coarse problem

The BDDC algorithm (Template 44) and the basic primal algorithm (Template 40) seems to be nearly the same, but there is one great difference between them: $\mathbf{S}$ is a block diagonal matrix but $\widetilde{\mathbf{S}}$ is not. In consequence, operations $\mathbf{Sv}$ and $\mathbf{S}^{-1}\mathbf{r}$ in the basic primal algorithm can be done independently on subdomains, in contrast to operations $\widetilde{\mathbf{S}}\mathbf{v}$ and $(\widetilde{\mathbf{S}})^{-1}\mathbf{r}$ in the BDDC algorithm. The same difference exists between the basic dual algorithm (Template 43) and the FETI-DP algorithm (Template 45).

In order to decompose global interface problem (7.47), both BDDC and FETI-DP introduce a *coarse space* and decompose the problem (7.47) to local problems and a global *coarse problem*. To this end Schur complement technique can be used in a similar way as in section 7.5.2, as the problems (7.8) and (7.47) have the same structure.

Recall that only values at coarse nodes are assumed as coarse dofs, so all the interface unknowns can be divided into coarse ones and the rest, not involved in coarse dofs. (If averages are used as coarse dofs, similar ideas as described here are employed, but everything is technically much more complicated; more about this can be found for instance in Toselli and Widlund [27] or Mandel and Dohrmann [20].)

Every subdomain matrix $\widetilde{\mathbf{S}}^{i}$ is assumed to be invertible, so all interface unknowns that are not coarse can be eliminated from (7.48):

$$\begin{bmatrix} \mathbf{S}_{\mathrm{rr}} & \mathbf{S}_{\mathrm{rc}} \\ \mathbf{0} & \mathbf{S}_{\mathrm{c}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{r}} \\ \widehat{\mathbf{u}}_{\mathrm{c}} \end{bmatrix} = \begin{bmatrix} \mathbf{g}_{\mathrm{r}} \\ \mathbf{g}_{\mathrm{c}} \end{bmatrix}, \tag{7.56}$$

where $\mathbf{S}_{\mathrm{c}} = \widehat{\mathbf{S}}_{\mathrm{c}} - \mathbf{S}_{\mathrm{cr}}\mathbf{S}_{\mathrm{rr}}^{-1}\mathbf{S}_{\mathrm{rc}}$ is a Schur complement of (7.47) with respect to the coarse unknowns and $\mathbf{g}_{\mathrm{c}} = \widehat{\mathbf{g}}_{\mathrm{c}} - \mathbf{S}_{\mathrm{cr}}\mathbf{S}_{\mathrm{rr}}^{-1}\mathbf{g}_{\mathrm{r}}$.

Problem (7.56) can be split into independent subdomain problems

$$\mathbf{S}_{\mathrm{rr}}\mathbf{u}_{\mathrm{r}} = \mathbf{g}_{\mathrm{r}} - \mathbf{S}_{\mathrm{rc}}\widehat{\mathbf{u}}_{\mathrm{c}} \tag{7.57}$$

and a global *coarse problem*

$$\mathbf{S}_{\mathrm{c}}\widehat{\mathbf{u}}_{\mathrm{c}} = \mathbf{g}_{\mathrm{c}}\,. \tag{7.58}$$

In order to disjoin the problems (7.57) and (7.58), solution $\mathbf{u}_{\mathrm{r}}$ of (7.57) is split as $\mathbf{u}_{\mathrm{r}} = \mathbf{u}_{\mathrm{rr}} + \mathbf{u}_{\mathrm{rc}}$ with $\mathbf{u}_{\mathrm{rr}}$ reflecting an effect of $\mathbf{g}_{\mathrm{r}}$, and the problem (7.57) is split accordingly:

$$\mathbf{S}_{\mathrm{rr}}\mathbf{u}_{\mathrm{rr}} = \mathbf{g}_{\mathrm{r}}\,, \tag{7.59}$$

$$\mathbf{S}_{\mathrm{rr}}\mathbf{u}_{\mathrm{rc}} = -\mathbf{S}_{\mathrm{rc}}\widehat{\mathbf{u}}_{\mathrm{c}}\,. \tag{7.60}$$

Problem (7.59) represents subdomain interface problems with zero Dirichlet boundary condition at coarse nodes and can be solved as N independent problems $\widetilde{\mathbf{S}}^i\widetilde{\mathbf{u}}^i_{\mathrm{rr}} = \widetilde{\mathbf{g}}^i$. Solution $\mathbf{u}_{\mathrm{rr}}$ of (7.59) can be used for formulation of the right hand side of (7.58) as $\mathbf{g}_{\mathrm{c}} = \widehat{\mathbf{g}}_c - \mathbf{S}_{\mathrm{cr}}\mathbf{u}_{\mathrm{rr}}$. Problem (7.60) represents subdomain interface problems with $\widehat{\mathbf{u}}_{\mathrm{c}}$ prescribed as Dirichlet boundary condition at coarse nodes and can be solved as N independent problems, too.

The coarse problem can be expressed either as (7.58) in terms of just the coarse unknowns, or on the whole interface using (7.58) and (7.60) together as

$$\begin{bmatrix} \mathbf{S}_{\mathrm{rr}} & \mathbf{S}_{\mathrm{rc}} \\ \mathbf{0} & \mathbf{S}_{\mathrm{c}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{rc}} \\ \widehat{\mathbf{u}}_{\mathrm{c}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{g}_{\mathrm{c}} \end{bmatrix}, \tag{7.61}$$

or on the whole domain using (7.18) and (7.61) together as

$$\begin{bmatrix} \mathbf{K}_{\mathrm{oo}} & \mathbf{K}_{\mathrm{or}} & \mathbf{K}_{\mathrm{oc}} \\ \mathbf{0} & \mathbf{S}_{\mathrm{rr}} & \mathbf{S}_{\mathrm{rc}} \\ \mathbf{0} & \mathbf{0} & \mathbf{S}_{\mathrm{c}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{orc}} \\ \mathbf{u}_{\mathrm{rc}} \\ \widehat{\mathbf{u}}_{\mathrm{c}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{g}_c \end{bmatrix} \tag{7.62}$$

with solution representing some function with values $\widehat{\mathbf{u}}_{\mathrm{c}}$ at the coarse nodes and minimal energy elsewhere on subdomains.

The problem (7.59) can be expressed on the whole interface as

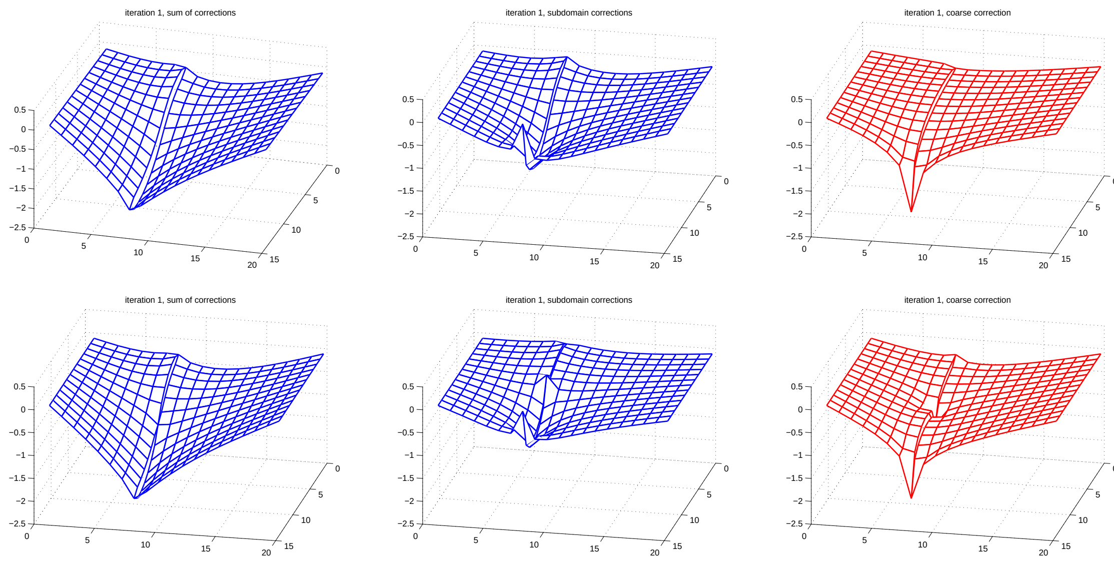
$$\begin{bmatrix} \mathbf{S}_{\mathrm{rr}} & \mathbf{S}_{\mathrm{rc}} \\ \mathbf{0} & \mathbf{S}_{\mathrm{c}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{rr}} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{g}_{\mathrm{r}} \\ \mathbf{0} \end{bmatrix}, \tag{7.63}$$

or on the whole domain using (7.18) and (7.63) together as

$$\begin{bmatrix} \mathbf{K}_{\mathrm{oo}} & \mathbf{K}_{\mathrm{or}} & \mathbf{K}_{\mathrm{oc}} \\ \mathbf{0} & \mathbf{S}_{\mathrm{rr}} & \mathbf{S}_{\mathrm{rc}} \\ \mathbf{0} & \mathbf{0} & \mathbf{S}_{\mathrm{c}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{orr}} \\ \mathbf{u}_{\mathrm{rr}} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{g}_{\mathrm{r}} \\ \mathbf{0} \end{bmatrix} \tag{7.64}$$

with solution representing some function with zero values at the coarse nodes, values $\mathbf{u}_{\mathrm{rr}}$ on the interface out of the coarse nodes and minimal energy inside subdomains.

Let us illustrate the coarse space on the example of the Poisson equation with 2 or 3 coarse nodes. Functions from corresponding $\widetilde{W}$ spaces are depicted in Fig. 7.11, reproduced again on the left column of the Fig. 7.12. They are decomposed to the coarse part - the solution of 7.62 (right) and the rest - the solution of 7.64, with zero values at the coarse nodes (center).



$$\begin{bmatrix}\mathbf{u}_{\mathrm{r}}\\ \widehat{\mathbf{u}}_{\mathrm{c}}\end{bmatrix} = \begin{bmatrix}\mathbf{u}_{\mathrm{rr}}\\ \mathbf{0}\end{bmatrix} + \begin{bmatrix}\mathbf{u}_{\mathrm{rc}}\\ \widehat{\mathbf{u}}_{\mathrm{c}}\end{bmatrix}$$

Figure 7.12: An example of two functions from space $\widetilde{W}$ (left column), for 2 and 3 coarse nodes (upper and lower row, respectively) decomposed to the coarse part (right) and the rest (center).

Remark:

In the section 7.5.2, a splitting of the global problem (7.8) to *local problems* (7.15), (7.16) and a smaller *global interface* problem (7.12) was described. In this section, a splitting of this global interface problem (7.12) to *local interface problems* (7.59), (7.60) and even smaller *global coarse problem* (7.58) was described. Both processes use technique of the Schur complement and are essentially the same. Following this way the global coarse problem (7.58) can also be split to get the remaining global problem even smaller and "coarser" and so on, resulting in *multilevel methods*, see Mandel, Sousedík and Dohrmann [23].

For description of the BDDC and the FETI-DP algorithms with the coarse problem, let us start with a decomposition of the space $\widetilde{W}$ as

$$\widetilde{W} = \widetilde{W}_{\mathrm{r}} \oplus \widetilde{W}_{\mathrm{c}}, \tag{7.65}$$

where $\widetilde{W}_{\mathrm{r}}$ ... a space of functions that have zero values at coarse degrees of freedom and minimal energy inside subdomains, corresponding to solutions of the problem (7.64). It can be further decomposed as $\widetilde{W}_{\mathrm{r}} = \widetilde{W}^1 \times \widetilde{W}^2 \times \cdots \times \widetilde{W}^N$, where $\widetilde{W}^i$ is a space of functions from $\widetilde{W}_{\mathrm{r}}$ restricted to Ω_i,
$\widetilde{W}_{\mathrm{c}}$ ... a *coarse space* of functions that have minimal energy out of the coarse nodes, corresponding to solutions of the problem (7.62).

Since spaces $\widetilde{W}_{\mathrm{c}}$ and $\widetilde{W}_{\mathrm{r}}$ are $\widetilde{\mathrm{S}}$-orthogonal (i.e. $\mathbf{w}_{\mathrm{c}}^{\mathrm{T}}\,\widetilde{\mathbf{S}}\,\mathbf{w} = \mathbf{0}\ \forall w_{\mathrm{c}} \in \widetilde{W}_{\mathrm{c}},\ w \in \widetilde{W}_{\mathrm{r}}$), problem (7.47) can be split to independent problems on $\widetilde{W}_{\mathrm{c}}$ and $\widetilde{W}_{\mathrm{r}}$ that can be solved in parallel and then summed up. In the space $\widetilde{W}_{\mathrm{r}}$ the basic primal or dual algorithm from section 7.5.4 is used. In the coarse space $\widetilde{W}_{\mathrm{c}}$ the coarse problem is formulated and solved by direct or iterative method.

Functions from $\widetilde{W}_{\mathrm{c}}$ are represented by their values $\widehat{\mathbf{u}}_{\mathrm{c}}$ at the coarse nodes, that can be obtained from $\widehat{\mathbf{u}}$ by an operator R_{c} represented by the matrix $\mathbf{R}_{\mathrm{c}}$ from (7.50) that keeps only coarse unknowns of a vector:

$$\widehat{\mathbf{u}}_{\mathrm{c}} = \mathbf{R}_{\mathrm{c}}\widehat{\mathbf{u}}. \tag{7.66}$$

Values $\widetilde{\mathbf{u}}_{\mathrm{c}}$ at the whole interface are then given by r-block of (7.61) or (7.62) as

$$\widetilde{\mathbf{u}}_{\mathrm{c}} = \begin{bmatrix} \mathbf{u}_{\mathrm{rc}} \\ \widehat{\mathbf{u}}_{\mathrm{c}} \end{bmatrix} = \boldsymbol{\Psi}\,\widehat{\mathbf{u}}_{\mathrm{c}}\,, \quad \text{where} \quad \boldsymbol{\Psi} = \begin{bmatrix} \boldsymbol{\Psi}_{\mathrm{r}} \\ \mathbf{I} \end{bmatrix} = \begin{bmatrix} -\mathbf{S}_{\mathrm{rr}}^{-1}\mathbf{S}_{\mathrm{rc}} \\ \mathbf{I} \end{bmatrix}. \tag{7.67}$$

Columns of the matrix $\boldsymbol{\Psi}$ consist of interface values of *coarse base functions* (see Fig. 7.13, where coarse base functions for 3 coarse nodes from example in the Fig. 7.12 bellow are depicted) that have value of 1 at one of the coarse dofs, zeros at all other coarse dofs and minimal energy on the rest of the subdomains. Coarse base functions are continuous only at coarse dofs. The coarse space with the coarse basis can be regarded as a finite element space formed by ”superelements” represented by subdomains.

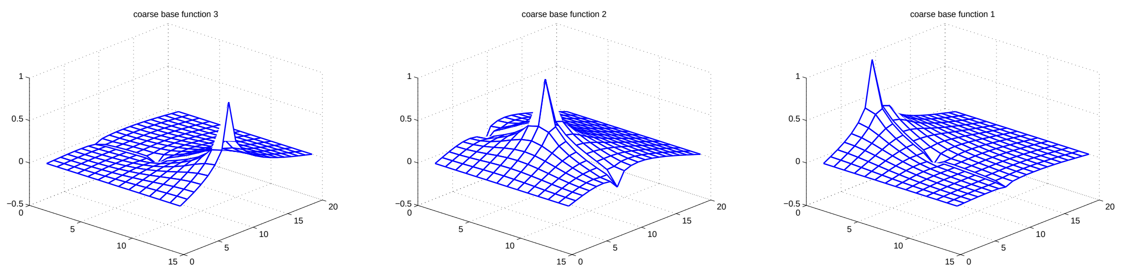


Figure 7.13: An example of the three coarse base functions for 3 coarse nodes (note that on the interface they are continuous only at the coarse nodes).

Transpose $\boldsymbol{\Psi}^{\mathrm{T}}$ of the matrix $\boldsymbol{\Psi}$ allows to project $\widetilde{r} \in \widetilde{W}'$ to $r_{\mathrm{c}} \in \widetilde{W}_{\mathrm{c}}'$ as

$$\mathbf{r}_{\mathrm{c}} = \boldsymbol{\Psi}^{\mathrm{T}}\widetilde{\mathbf{r}}\,. \tag{7.68}$$

In mechanics it represents a replacement of node forces (or reactions) given at all the interface nodes by coarse node forces only.

The coarse basis $\boldsymbol{\Psi}$ and the coarse matrix $\mathbf{S}_{\mathrm{c}}$ can be also expressed in a way similar to (7.14) as a problem with multiple right hand side

$$\begin{bmatrix} \mathbf{S}_{\mathrm{rr}} & \mathbf{S}_{\mathrm{rc}} & \mathbf{0} \\ \mathbf{S}_{\mathrm{cr}} & \widehat{\mathbf{S}}_{\mathrm{c}} & \mathbf{I} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Psi}_{\mathrm{r}} \\ \mathbf{I} \\ -\mathbf{S}_{\mathrm{c}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{I} \end{bmatrix}. \tag{7.69}$$

The BDDC algorithm with explicitly formulated coarse problem is described in Template 46. It differs from the BDDC algorithm in Template 44 only by organization of the computation: in Template 46 a parallelization is made explicit by means of the coarse problem (parallel operations at single steps are separated by a ”|” symbol). Results of both algortihms should be the same in every iteration (except rounding errors).

Template 46, The BDDC method with the coarse problem

```
Choose $\widehat{\mathbf{u}}^{(0)}$ arbitrarily and repeat for $k := 0, 1, 2, \ldots$:
  1. (a) ${\mathbf{u}_\mathrm{r}}^{(k)} := \mathbf{R}\,\widehat{\mathbf{u}}^{(k)}$
     (b) $\mathbf{g}^{(k)} := \mathbf{S}{\mathbf{u}_\mathrm{r}}^{(k)}$
     (c) $\widehat{\mathbf{r}}^{(k)} := \widehat{\mathbf{g}} - \mathbf{R}^\mathrm{T}\mathbf{g}^{(k)}$
     if $\widehat{\mathbf{r}}^{(k)}$ is small enough, stop
  2. $\Delta\mathbf{r}_\mathrm{c}^{(k)} := \mathbf{\Psi}^\mathrm{T}\,\widetilde{\mathbf{E}}^\mathrm{T}\,\widehat{\mathbf{r}}^{(k)}$   |   $\Delta\mathbf{r}_\mathrm{r}^{(k)} := \widetilde{\mathbf{E}}_\mathrm{r}^\mathrm{T}\,\widehat{\mathbf{r}}^{(k)}$
  3. $\Delta\widehat{\mathbf{u}}_\mathrm{c}^{(k)} := \mathbf{S}_\mathrm{c}^{-1}\,\Delta\mathbf{r}_\mathrm{c}^{(k)}$   |   $\Delta{\mathbf{u}_\mathrm{r}}^{(k)} := \mathbf{S}_\mathrm{rr}^{-1}\,\Delta\mathbf{r}_\mathrm{r}^{(k)}$
  4. $\Delta\widehat{\mathbf{u}}^{(k)} := \widetilde{\mathbf{E}}\,\mathbf{\Psi}\,\Delta\widehat{\mathbf{u}}_\mathrm{c}^{(k)} + \widetilde{\mathbf{E}}_\mathrm{r}\,\Delta{\mathbf{u}_\mathrm{r}}^{(k)}$
  5. $\widehat{\mathbf{u}}^{(k+1)} := \widehat{\mathbf{u}}^{(k)} + \Delta\widehat{\mathbf{u}}^{(k)}$
```

Putting the last four steps of this algorithm together, we get a mathematical formulation of the BDDC method with explicit coarse problem as a Richardson method for the Schur complement problem (7.12):

$$\widehat{\mathbf{u}}^{(k+1)} = \widehat{\mathbf{u}}^{(k)} + \widetilde{\mathbf{E}}\,\mathbf{\Psi}\,\mathbf{S}_\mathrm{c}^{-1}\,\mathbf{\Psi}^\mathrm{T}\,\widetilde{\mathbf{E}}^\mathrm{T}\,\widehat{\mathbf{r}}^{(k)} + \widetilde{\mathbf{E}}_\mathrm{r}\,\mathbf{S}_\mathrm{rr}^{-1}\,\widetilde{\mathbf{E}}_\mathrm{r}^\mathrm{T}\,\widehat{\mathbf{r}}^{(k)}. \tag{7.70}$$

The last two terms are sometimes called *coarse* and *subdomain correction*, respectively. Their examples are depicted in the Fig. 7.13 right and center columns, respectively. Both corrections are independent and can be computed in parallel.
Remark: Computation of the residual at the step 1 of the Template 46 is the same as in the basic primal algorithm in the Template 40. It could also be done with the coarse problem as

1. (a) $\widehat{\mathbf{u}}_\mathrm{c}^{(k)} := \mathbf{R}_\mathrm{c}\,\widehat{\mathbf{u}}^{(k)} \quad | \quad \mathbf{u}_\mathrm{r}^{(k)} := \widetilde{\mathbf{R}}_\mathrm{r}\,\widehat{\mathbf{u}}^{(k)}$
   (b) $\widehat{\mathbf{g}}_\mathrm{c}^{(k)} := \mathbf{S}_\mathrm{c}\,\widehat{\mathbf{u}}_\mathrm{c}^{(k)} \quad | \quad {\mathbf{g}_\mathrm{rr}}^{(k)} := \mathbf{S}_\mathrm{rr}\,(\mathbf{u}_\mathrm{r}^{(k)} + \mathbf{\Psi}_\mathrm{r}^\mathrm{T}\widehat{\mathbf{u}}_\mathrm{c}^{(k)})$
   (c) $\widehat{\mathbf{r}}^{(k)} := \widehat{\mathbf{g}} - \mathbf{R}_\mathrm{c}^\mathrm{T}\,\widehat{\mathbf{g}}_\mathrm{c}^{(k)} - \widetilde{\mathbf{R}}_\mathrm{r}^\mathrm{T}\,{\mathbf{g}_\mathrm{rr}}^{(k)}$

If the coarse dofs are values at coarse nodes only as assumed above, jump operators $\widetilde{\mathrm{B}}$ and $\widetilde{\mathrm{B}}_\mathrm{D}$ written in a block form have zeros at the coarse blocks

$$\widetilde{\mathbf{B}} = \begin{bmatrix} \mathbf{B}_\mathrm{r} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} = \begin{bmatrix} \widetilde{\mathbf{B}}_\mathrm{r} \\ \mathbf{0} \end{bmatrix}, \qquad \widetilde{\mathbf{B}}_\mathbf{D} = \begin{bmatrix} \mathbf{B}_{\mathrm{Dr}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} = \begin{bmatrix} \widetilde{\mathbf{B}}_{\mathbf{D}\mathrm{r}} & \mathbf{0} \end{bmatrix}. \tag{7.71}$$

Coarse components of both $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda}'$ are zeros, as there are no jumps in the coarse unknowns. The problem (7.54) then shrinks to

$$\mathbf{B}_\mathrm{r}{\mathbf{S}_\mathrm{rr}}^{-1}\mathbf{B}_\mathrm{r}^\mathrm{T}\boldsymbol{\lambda}_\mathrm{r}' = \widetilde{\mathbf{B}}_\mathrm{r}\widetilde{\mathbf{S}}^{-1}\widetilde{\mathbf{g}}. \tag{7.72}$$

The FETI-DP algorithm with explicitly formulated coarse problem is described in Template 47. It differs from the FETI-DP algorithm in Template 45 by restriction to coarse nodes only (no averages) and by an organization of the computation at the first step: in Template 47 a parallelization is made explicit by means of the coarse problem (parallel operations are separated by a "|" symbol). Results of both algortihms should be the same in every iteration (except rounding errors).

Template 47, The FETI-DP algorithm with coarse problem

```
Choose $\mathbf{g}_c := \boldsymbol{\Psi}^T \widetilde{\mathbf{E}}^T \widehat{\mathbf{g}}$ , $\mathbf{g}_r := \widetilde{\mathbf{E}}_r^T \widehat{\mathbf{g}}$ , $\boldsymbol{\lambda}'^{(0)} := \mathbf{0}$ and repeat for $k := 0, 1, 2, \ldots$ :
    1. $\widehat{\mathbf{u}}_c^{(k)} := \mathbf{S}_c^{-1} (\mathbf{g}_c - \boldsymbol{\Psi}_r^T \boldsymbol{\lambda}'^{(k)}_r) \quad | \quad \mathbf{u}_{rr}^{(k)} := \mathbf{S}_{rr}^{-1} (\mathbf{g}_r^{(k)} - \mathbf{B}_r^T \boldsymbol{\lambda}'^{(k)}_r)$
    2. $\boldsymbol{\lambda}_r^{(k)} := \mathbf{B}_r (\mathbf{u}_{rr}^{(k)} + \boldsymbol{\Psi}_r \widehat{\mathbf{u}}_c^{(k)})$ , if $\boldsymbol{\lambda}_r^{(k)}$ is small enough, stop
    3. $\Delta \mathbf{u}_r^{(k)} := \mathbf{B}_{\mathbf{D}r}^T \boldsymbol{\lambda}_r^{(k)}$
    4. $\Delta \boldsymbol{\lambda}'^{(k)}_r := \mathbf{B}_{\mathbf{D}r} \mathbf{S}_{rr} \Delta \mathbf{u}_r^{(k)}$
    5. $\boldsymbol{\lambda}'^{(k+1)}_r := \boldsymbol{\lambda}'^{(k)}_r + \Delta \boldsymbol{\lambda}'^{(k)}_r$
```

7.6 DD methods as preconditioners

DD methods usually are not used on their own. They are used as outstanding preconditioners, specifically tailored to the given problem. The original problem (7.1), or the Schur complement problem (7.12), or problems (7.41) or (7.54) for Lagrange multipliers, are actually solved using some other iterative method, typically PCG for symmetric problems and GMRES for nonsymetric ones.

A preconditioner $\mathbf{M}$ for a given problem $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ is sought so that it has two concurrent properties:

- the problem $\mathbf{M}\mathbf{A}\,\mathbf{x} = \mathbf{M}\mathbf{b}$ has good spectral properties (in this sense $\mathbf{M}$ can be regarded as some approximation of $\mathbf{A}^{-1}$), and
- a preconditioned residual $\mathbf{p} = \mathbf{M}\mathbf{r}$ is cheap to obtain for any given $\mathbf{r}$.

A good preconditioner improves the convergence of the iterative method; without a preconditioner the iterative method may even fail to converge. More about preconditioners and iterative methods can be found in Saad [24] or Barret et al [2].

Next idea is adopted from Le Tallec [17]: DD methods in preceeding sections are formulated as Richardson iterative methods with a preconditioner $\mathbf{M}$ for a problem $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \rho\,\mathbf{M}\mathbf{r}^{(k)}, \tag{7.73}$$

where $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\,\mathbf{x}^{(k)}$ is a residual at k-th iterative step and $\rho = 1$. Any such method can be understood as a recipe for computing a preconditioned residual $\mathbf{p}$ by using only second term of (7.73) as $\mathbf{p} = \mathbf{M}\mathbf{r}$. This is the way how DD methods are used in practice.

Remark:

When DD methods are used as preconditioners, computer time and memory can be saved by solving subdomain problems (i.e. problems solved in $\widetilde{W}$) only approximately, see [17], [6] or [13].

Bibliography

[1] P. R. Amestoy and I. S. Duff. MUMPS – a multifrontal massively parallel sparse direct solver. http://mumps.enseeiht.fr.

[2] R. Barrett, M. Berry, T. F. 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, 1994. http://www.netlib.org/templates/Templates.html.

[3] Susanne C. Brenner. The condition number of the Schur complement in domain decomposition. *Numer. Math.*, 83(2):187–203, 1999.

[4] Susanne C. Brenner and Li-Yeng Sung. BDDC and FETI-DP without matrices or vectors. *Comput. Methods Appl. Mech. Engrg.*, 196(8):1429–1435, 2007.

[5] Clark R. Dohrmann. A preconditioner for substructuring based on constrained energy minimization. *SIAM J. Sci. Comput.*, 25(1):246–258, 2003.

[6] Clark R. Dohrmann. An approximate BDDC preconditioner. *Numerical Linear Algebra with Applications*, 14(2):149–168, 2007.

[7] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 9:302–325, 1983.

[8] C. Farhat, M. Lesoinne, P. Le Tallec, K. Pierson, and Rixen, D. FETI-DP: a dual-primal unified FETI method. I. A faster alternative to the two-level FETI method. *Internat. J. Numer. Methods Engrg.*, 50(7):1523–1544, 2001.

[9] Charbel Farhat and Francois-Xavier Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *Internat. J. Numer. Methods Engrg.*, 32:1205–1227, 1991.

[10] B. Hendrickson and R. Leland. CHACO: Software for partitioning graphs. http://www.sandia.gov/˜ bahendr/chaco.html.

[11] Bruce M. Irons. A frontal solution scheme for finite element analysis. *Internat. J. Numer. Methods Engrg.*, 2:5–32, 1970.

[12] G. Karypis and V. Kumar. METIS – family of multilevel partitioning algorithms. http://glaros.dtc.umn.edu/gkhome/views/metis/index.html.

[13] Axel Klawonn and Oliver Rheinbach. Inexact FETI-DP methods. *International journal for numerical methods in engineering*, 69(2):284–307, 2007.

[14] Axel Klawonn and Oliver Rheinbach. Robust FETI-DP methods for heterogeneous three dimensional elasticity problems. *Comput. Methods Appl. Mech. Engrg.*, 196(8):1400–1414, 2007.

[15] Axel Klawonn and Olof B. Widlund. FETI and Neumann-Neumann iterative substructuring methods: connections and new results. *Comm. Pure Appl. Math.*, 54(1):57–90, 2001.

[16] Axel Klawonn, Olof B. Widlund, and Maksymilian Dryja. Dual-primal FETI methods for three-dimensional elliptic problems with heterogeneous coefficients. *SIAM J. Numer. Anal.*, 40(1):159–179, 2002.

[17] Patrick Le Tallec. Domain decomposition methods in computational mechanics. *Computational Mechanics Advances*, 1(2):121–220, 1994.

[18] Jing Li and Olof B. Widlund. FETI-DP, BDDC, and block Cholesky methods. *Internat. J. Numer. Methods Engrg.*, 66(2):250–271, 2006.

[19] Jan Mandel. Balancing domain decomposition. *Comm. Numer. Methods Engrg.*, 9(3):233–241, 1993.

[20] Jan Mandel and Clark R. Dohrmann. Convergence of a balancing domain decomposition by constraints and energy minimization. *Numer. Linear Algebra Appl.*, 10(7):639–659, 2003.

[21] Jan Mandel, Clark R. Dohrmann, and Radek Tezaur. An algebraic theory for primal and dual substructuring methods by constraints. *Appl. Numer. Math.*, 54(2):167–193, 2005.

[22] Jan Mandel and Bedřich Sousedík. BDDC and FETI-DP under minimalist assumptions. *Computing*, 81:269–280, 2007.

[23] Jan Mandel, Bedřich Sousedík, and Clark R. Dohrmann. On multilevel BDDC. *Lecture Notes in Computational Science and Engineering*, 60:287–294, 2007. Domain Decomposition Methods in Science and Engineering XVII.

[24] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, 2003. http://www-users.cs.umn.edu/˜ saad/books.html.

[25] H. A. Schwarz. Über einen Grenzübergang durch alternierendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich*, 15:272–286, May 1870.

[26] Bedřich Sousedík. *Comparison of some domain decomposition methods.* PhD thesis, Czech Technical University in Prague, Faculty of Civil Engineering, Department of Mathematics, 2008. http://www-math.cudenver.edu/˜ sousedik/papers/BSthesisCZ.pdf.

[27] Andrea Toselli and Olof Widlund. *Domain decomposition methods—algorithms and theory*, volume 34 of *Springer Series in Computational Mathematics.* Springer-Verlag, Berlin, 2005.

[28] Irad Yavneh. Why multigrid methods are so efficient. *Computing in Science and Engineering*, 8(6):12–22, November/December 2006.

Chapter 8

FETI Based Domain Decompositions

This part was written and is maintained by Jiří Dobiáš. More details about the author can be found in the Chapter 16.

8.1 Introduction

In general, any domain decomposition method is based on the idea that an original domain can be divided into several sub-domains that may or may not overlap. This is depicted in Fig. 8.1 for a two-dimensional problem.

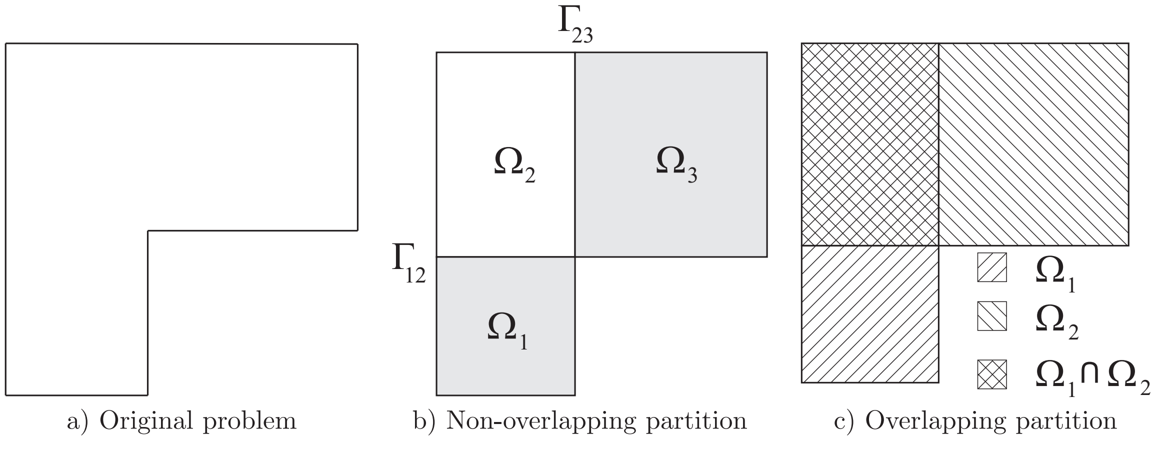


Figure 8.1: Principle of domain decomposition methods

There are three sub-domains in the non-overlapping case and two in the overlapping one. Then the original problem is considered on every sub-domain, which generates sub-problems of reduced sizes. They are coupled with each other in terms of variables along their interfaces or throughout the overlapping regions. The overlapping region is shown as the cross-hatched area in Fig. 8.1c). The techniques based on overlapping sub-domains are referred to as Schwarz methods, while those making use of non-overlapping sub-domains are usually referred to as sub-structuring techniques.

There exist three fundamental monographs on the subject of the domain docomposition methods,[11, 10, 9], and a great number of other references. The list of references appended to the first book is quite extensive.

In this chapter we are concerned with one of the domain decomposition methods called FETI and its variant TFETI. In the following sections we explain principles of the method, introduce briefly the underlying mathematics, and show results of some numerical experiments.

8.2 The FETI method

In 1991 Ch. Farhat and F.-X. Roux came up with a novel domain decomposition method called FETI (Finite Element Tearing and Interconnecting method) [7]. This method belongs to the class of non-overlapping totally disconnected spatial decompositions. Its key concept stems from the idea that satisfaction of the compatibility between spatial sub-domains, into which a domain is partitioned, is ensured by the Lagrange multipliers, or forces in this context. After eliminating the primal variables, which are displacements in the displacement based analysis, the original problem is reduced to a small, relatively well conditioned, typically equality constrained quadratic programming problem that is solved iteratively. The CPU time that is necessary for both the elimination and iterations can be reduced nearly proportionally to the number of processors, so that the algorithm exhibits the parallel scalability. This method has proved to be one of the most successful algorithms for parallel solution of problems governed by elliptic partial differential equations. Observing that the equality constraints may be used to define so called ‘natural coarse grid’, Farhat, Mandel and Roux modified the basic FETI algorithm so that they were able to prove its numerical scalability, i.e. asymptotically linear complexity [6].

In addition, the fact that sub-domains act on each other in terms of forces suggests that the FETI approach can also be naturally applied to solution to the contact problems with great benefit. To this effect the FETI methodology is used to prescribe conditions of non-penetration between bodies.

8.2.1 FETI method principle

Consider two bodies in contact as in Fig. 8.2. Decompose them into sub-domains. The interfaces between sub-domains can pass either along the contact interface or through the bodies, which creates fictitious borders between the sub-domains, while we should abide by all the recommendations regarding the aspect ratios, and so on. This process may produce either reasonably constrained sub-domains, which are unhatched here, or partly constrained, which are cross-hatched, or without any constrained, which are hatched. All the sub-domains with not enough constraints can move like a rigid body, or can float, or can undergo the rigid body modes. This class of sub-domains requires special treatment, because their underlying stiffness matrices are singular so that they exhibit defects ranging from one to the maximum, which is 6 for 3D mechanic problems or 3 for 2D problems.

Let us consider the static case of a system of two solid deformable bodies that are in contact. This is basically the boundary value problem known from the continuum solid mechanics. The problem is depicted in Fig. 8.3 a).

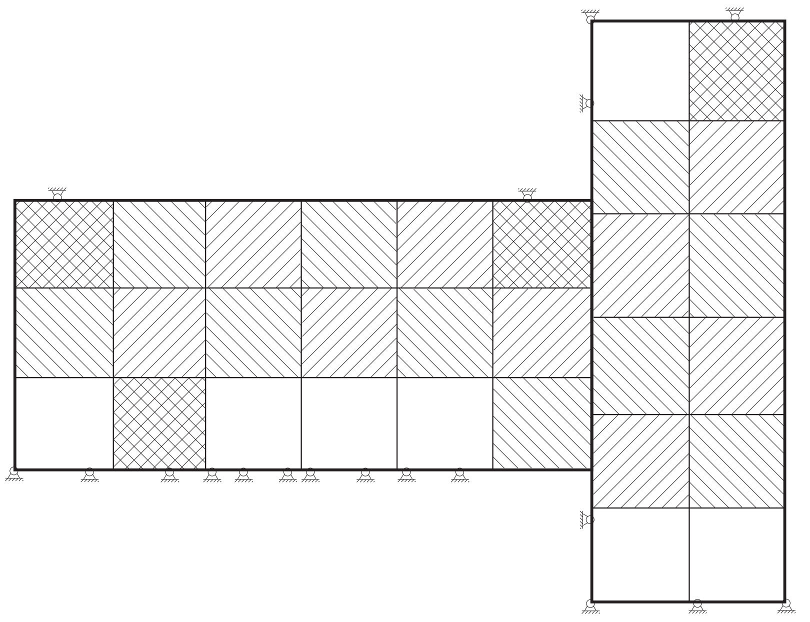


Figure 8.2: Constrained and floating sub-domains

Two bodies are denoted by $(\Omega_1, \Omega_2) \subset \mathbf{R}^n$, $n = 2$ or $n = 3$, where n stands for number of the Euclidean space dimensions. Γ denotes their boundary. We assume that the boundary is subdivided into three disjoint parts. The Dirichlet and Neumann boundary conditions are prescribed on the parts Γ^u and Γ^f, respectively. The third kind of the boundary conditions, Γ^c, is defined along the regions where contact occurs and can in general be treated as both the Dirichlet or Neumann conditions. The governing equations are given by the equilibrium conditions of the system of bodies. In addition to these equations, the problem also is subject to the boundary conditions; see, e.g., [8, Chapter 2] for comprehensive survey of formulations.

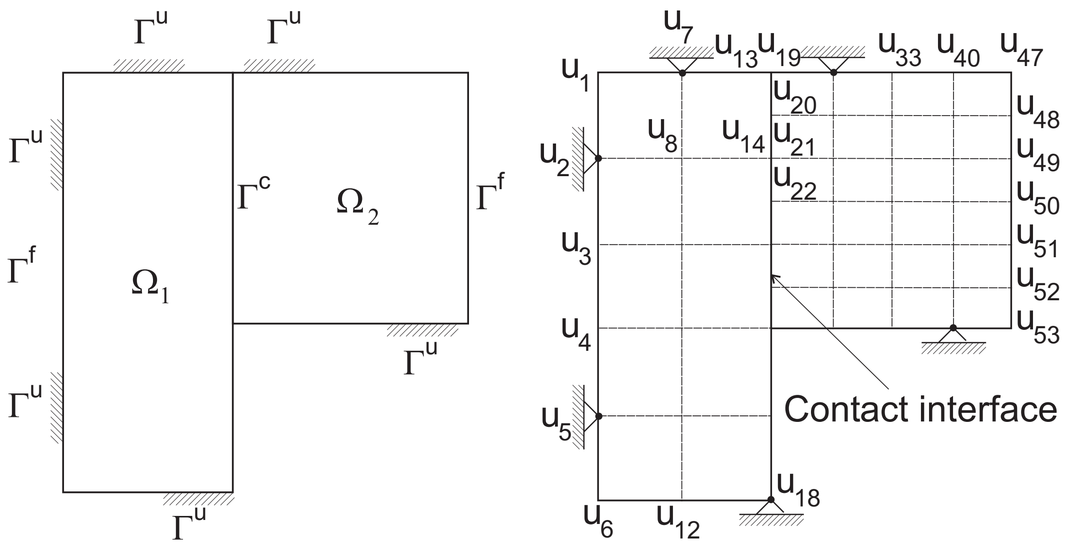
Fig. 8.3 b) shows the discretised version of the problem from Fig. 8.3 a). Both sub-domains are discretised in terms of the finite elements method. This figure also shows applied Dirichlet boundary conditions, some displacements, denoted as $\mathbf{u}$, associated with the nodal points, and the contact interface. The displacements are the primal variables in the context of the displacement based finite element analysis.

The result of application of the FETI method to the computational model from Fig. 8.3 a) is depicted in Fig. 8.4 a).

The sub-domain Ω_1 is decomposed into two sub-domains in this case with fictitious interface between them. The contact interface remains the same. The fundamental idea of the FETI method is that the compatibility between sub-domains is ensured by means of the Lagrange multipliers or forces. $\boldsymbol{\lambda}^{\mathrm{E}}$ denotes the forces along the fictitious interface and $\boldsymbol{\lambda}^{\mathrm{I}}$ stands for the forces generated by contact.

8.2.2 FETI method basic mathematics

Let N be a number of sub-domains and let us denote for $i = 1, \ldots, N$ by $\mathbf{K}^{(i)}$, $\mathbf{f}^{(i)}$, $\mathbf{u}^{(i)}$ and $\mathbf{B}^{(i)}$ the stiffness matrix, the vector of externally applied forces, the vector of displacements and the signed matrix with entries $-1, 0, 1$ defining the sub-domain interconnectivity for the (i)-th sub-domain, respectively. The matrix $\mathbf{B}$ is composed of



a) Original problem b) Primal variables

Figure 8.3: Basic notation

matrices $\mathbf{B}^{\mathrm{I}}$ and $\mathbf{B}^{\mathrm{E}}$, $\mathbf{B} = \left[\mathbf{B}^{\mathrm{I}}\,\mathbf{B}^{\mathrm{E}}\right]$. $\mathbf{B}^{\mathrm{E}}$ introduces connectivity conditions along the fictitious interfaces and $\mathbf{B}^{\mathrm{I}}$ along the contact ones.

The discretised version of the problem is governed by the equation

$$\min \frac{1}{2}\mathbf{u}^\top \mathbf{K}\,\mathbf{u} - \mathbf{f}^\top \mathbf{u} \quad \text{subject to} \quad \mathbf{B}^{\mathrm{I}}\mathbf{u} \le 0 \quad \text{and} \quad \mathbf{B}^{\mathrm{E}}\mathbf{u} = 0 \tag{8.1}$$

where

$$\mathbf{K} = \begin{bmatrix} \mathbf{K}^{(1)} & & \\ & \ddots & \\ & & \mathbf{K}^{(N)} \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \vdots \\ \mathbf{f}^{(N)} \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \mathbf{u}^{(1)} \\ \vdots \\ \mathbf{u}^{(N)} \end{bmatrix}. \tag{8.2}$$

The original FETI method assumes that Dirichlet boundary conditions are inherited from the original problem, which is shown in Fig. 8.4 a). This fact implies that defects of the stiffness matrices, $\mathbf{K}^{(i)}$, may vary from zero, for the sub-domains with enough Dirichlet conditions, to the maximum (6 for 3D solid mechanics problems and 3 for 2D ones) in the case of the sub-domains exhibiting some rigid body modes. General solution to such systems requires computation of generalised inverses and bases of the null spaces, or kernels, of the underlying singular matrices. The problem is that the magnitudes of the defects are difficult to evaluate because this computation is extremely disposed to the round off errors [5].

To circumvent the problem of computing bases of the kernels of singular matrices, Dostál came up with a novel solution [2]. His idea was to remove all the prescribed Dirichlet boundary conditions and to enforce them by the Lagrange multipliers denoted as λ^{B} in Fig. 8.4 b). The effect of the procedure on the stiffness matrices of the sub-domains is that their defects are the same and their magnitude is known beforehand. From the computational point of view such approach is advantageous, see [5] for discussion of this topic. This variant of the FETI method is called the total FETI because there are no primal variables whatsoever considered after elimination of the displacements. The fundamental mathematics behind both FETI and TFETI is presented next. It stems from [6] and others, e.g. [3].

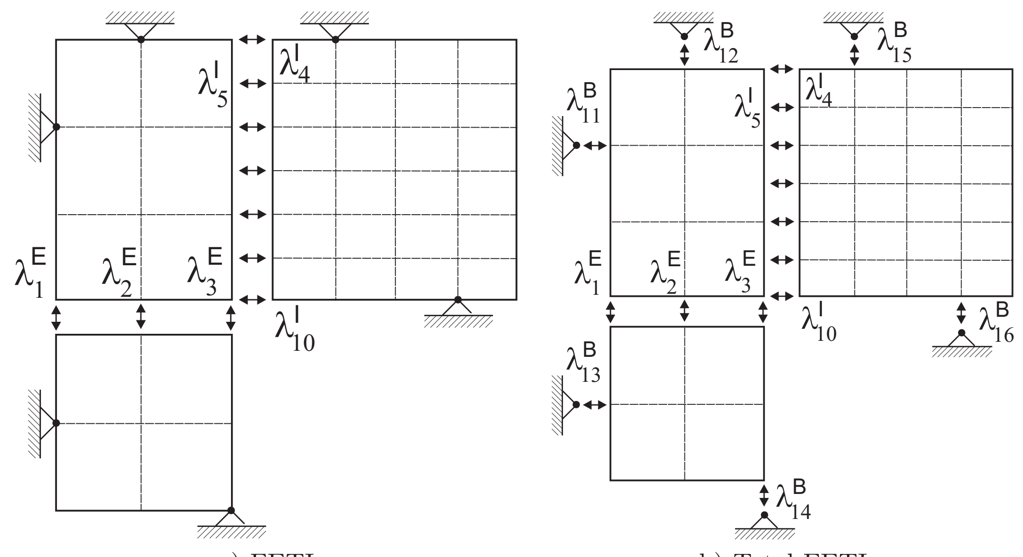


Figure 8.4: Principles of FETI and TFETI

The Lagrangian associated with the problem governed by Equation (8.1) is as reads

$$\mathrm{L}(\mathbf{u},\lambda) = \frac{1}{2}\mathbf{u}^\top\mathbf{K}\mathbf{u} - \mathbf{f}^\top\mathbf{u} + \lambda^\top\mathbf{B}\mathbf{u}. \tag{8.3}$$

This is equivalent to the saddle point problem

$$\text{Find} \quad (\bar{\mathbf{u}},\bar{\lambda}) \quad \text{so that} \quad \mathrm{L}(\bar{\mathbf{u}},\bar{\lambda}) = \sup_{\lambda}\inf_{\mathbf{u}} \mathrm{L}(\mathbf{u},\lambda). \tag{8.4}$$

For λ fixed, the Lagrangian $\mathrm{L}(.,\lambda)$ is convex in the first variable and a minimiser $\mathbf{u}$ of $\mathrm{L}(.,\lambda)$ satisfies the following equation

$$\mathbf{K}\mathbf{u} - \mathbf{f} + \mathbf{B}^\top\lambda = 0. \tag{8.5}$$

Equation (8.5) has a solution if and only if

$$\mathbf{f} - \mathbf{B}^\top\lambda \in \operatorname{Im}\mathbf{K}, \tag{8.6}$$

which can be expressed in a more convenient way in terms of the matrix $\mathbf{R}$ whose columns span the kernel of $\mathbf{K}$ as follows

$$\mathbf{R}^\top(\mathbf{f} - \mathbf{B}^\top\lambda) = 0. \tag{8.7}$$

The kernels of the sub-domains are known and can be assembled directly.

It is necessary to eliminate the primal variable $\mathbf{u}$ from (8.5). Assume that λ satisfies (8.6) and denote by $\mathbf{K}^\dagger$ any symmetric positive definite matrix satisfying at least one of the Penrose axioms

$$\mathbf{K}\mathbf{K}^\dagger\mathbf{K} = \mathbf{K}. \tag{8.8}$$

It may be easily verified that if $\mathbf{u}$ is a solution to (8.5), then there exists a vector α such that

$$\mathbf{u} = \mathbf{K}^\dagger(\mathbf{f} - \mathbf{B}^\top\lambda) + \mathbf{R}\alpha. \tag{8.9}$$

Substituting (8.9) into (8.4), we get the following minimisation problem

$$\min \frac{1}{2}\lambda^\top \mathbf{B}\,\mathbf{K}^\dagger\,\mathbf{B}^\top\lambda - \lambda^\top\mathbf{B}\,\mathbf{K}^\dagger\mathbf{f}, \quad \text{s.t.} \quad \mathbf{R}^\top(\mathbf{f} - \mathbf{B}^\top\lambda) = 0. \tag{8.10}$$

Let us introduce notations

$$\mathbf{F} = \mathbf{B}\mathbf{K}^\dagger\mathbf{B}^\top, \quad \mathbf{G} = \mathbf{R}^\top\mathbf{B}^\top, \quad \mathbf{e} = \mathbf{R}^\top\mathbf{f}, \quad \mathbf{d} = \mathbf{B}\mathbf{K}^\dagger\mathbf{f}, \tag{8.11}$$

so that the problem (8.10) reads

$$\min \frac{1}{2}\lambda^\top\mathbf{F}\lambda - \lambda^\top\mathbf{d} \quad \text{s.t.} \quad \mathbf{G}\lambda = 0. \tag{8.12}$$

The final step stems from observation that the problem (8.12) is equivalent to

$$\min \frac{1}{2}\lambda^\top\mathbf{P}\mathbf{F}\mathbf{P}\lambda - \lambda^\top\mathbf{P}\mathbf{d} \quad \text{s.t.} \quad \mathbf{G}\lambda = 0, \tag{8.13}$$

where

$$\mathbf{P} = \mathbf{I} - \mathbf{Q} \quad \text{and} \quad \mathbf{Q} = \mathbf{G}^\top(\mathbf{G}\mathbf{G}^\top)^{-1}\mathbf{G} \tag{8.14}$$

stand for the orthogonal projectors on the kernel of $\mathbf{G}$ and the image space of $\mathbf{G}^\top$, respectively.

The problem (8.13) may be solved efficiently by the conjugate gradient method because the estimate of the spectral condition number κ for the FETI method

$$\kappa(\mathbf{P}\mathbf{F}\mathbf{P}|\mathrm{Im}\mathbf{P}) \leq \mathrm{const}\frac{H}{h} \tag{8.15}$$

holds also for TFETI [2]. Here H denotes the decomposition parameter a h stands for the discretisation parameter. Minute analysis of the proof of (8.15) reveals that the constants in the bound may be in many cases more favourable for TFETI than for the original FETI.

It was shown that application of TFETI methodology to the contact problems converts the original problem to the quadratic programming one with simple bounds and equality constraints. This problem can be further transformed by Semi-Monotonic Augmented Lagrangians with Bound and Equality constraints (SMALBE) method to the sequence of simply bounded quadratic programming problems. These auxiliary problems may be solved efficiently by the Modified Proportioning with Reduced Gradient Projection (MPRGP) method. The detail descriptions of SMALBE and MPRGP are beyond the scope of this text and can be found in [4]. It was proved in [1] that application of combination of both these methods to solution to contact problems benefits the numerical and parallel scalabilities.

8.3 Numerical Experiments

To demonstrate the ability of our algorithms to solve contact problems, we show results of two sets of numerical experiments we carried out. The first case is concerned with contact problem of two cylinders, and the second one with contact problem of the pin in hole with small clearance.

8.3.1 The Contact Problem of Two Cylinders

Consider contact of two cylinders with parallel axes. Mesh of the computational model is shown in Fig. 8.5 and its detail in vicinity of the contact region in Fig. 8.6. We can consider only one half of the problem due to its symmetry. The diameter of the upper cylinder $R_u = 1\ m$ and of the lower one $R_l = \infty$. In spite of the fact that it is the 2D problem, it is modelled with 3D continuum tri-linear elements with two layers of them along the axis of symmetry of the upper cylinder. Nevertheless, the number of layers is irrelevant. The model consists of 8904 elements and 12765 nodes. The boundary conditions are imposed in such a way that they generate, from the physical point of view, the plane strain problem. The material properties are as follows: Young's modulus $E = 2.0 \times 10^{11}\ Pa$ and Poisson's ratio $\nu = 0.3$.

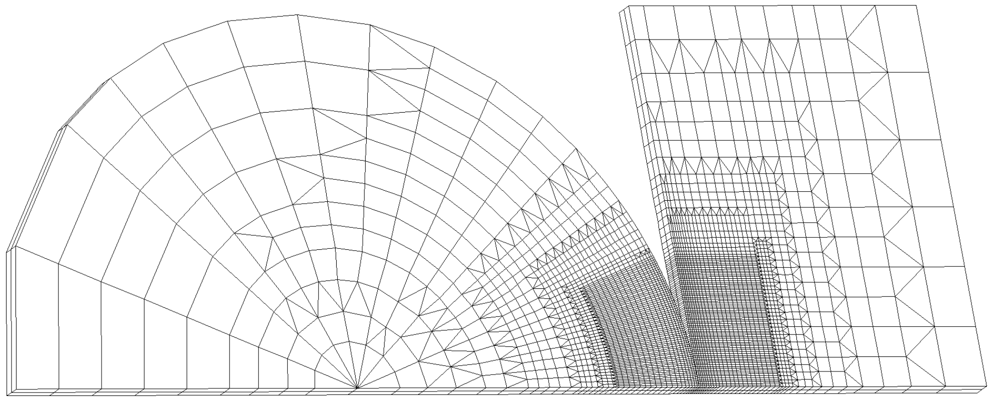


Figure 8.5: Problem of two cylinders: mesh

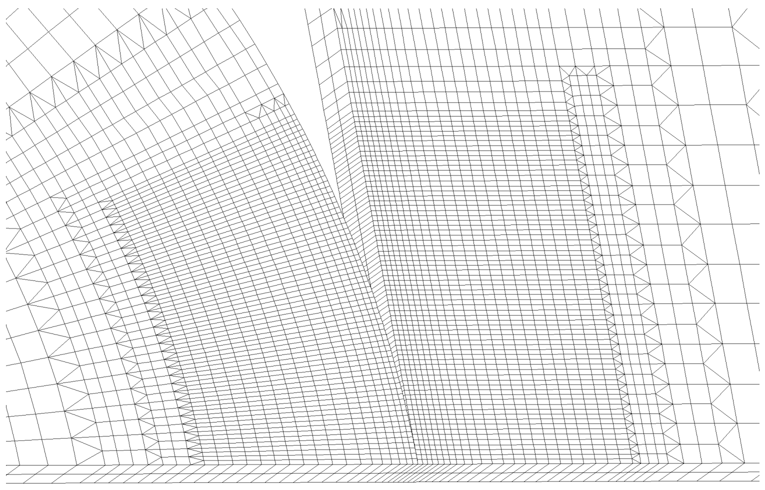


Figure 8.6: Detail of mesh

First, the upper cylinder is loaded by 40 MN/m along its upper line and the problem is considered as linearly elastic and linearly geometric. Fig. 8.7 shows solution in terms of the deformed mesh.

Next, the problem was computed on the same mesh with the same loading, but we considered the linearly–elastic–perfectly–plastic material model with the yield stress

$\sigma_Y = 800\ MPa$. We also considered the geometric non-linearity. The deformed mesh is depicted in Fig. 8.8.

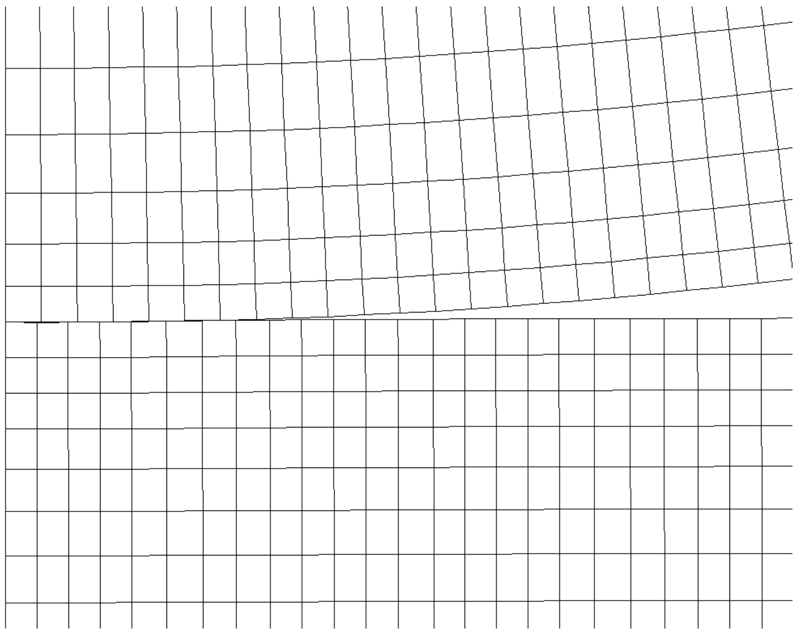


Figure 8.7: Deformed mesh, linear problem.

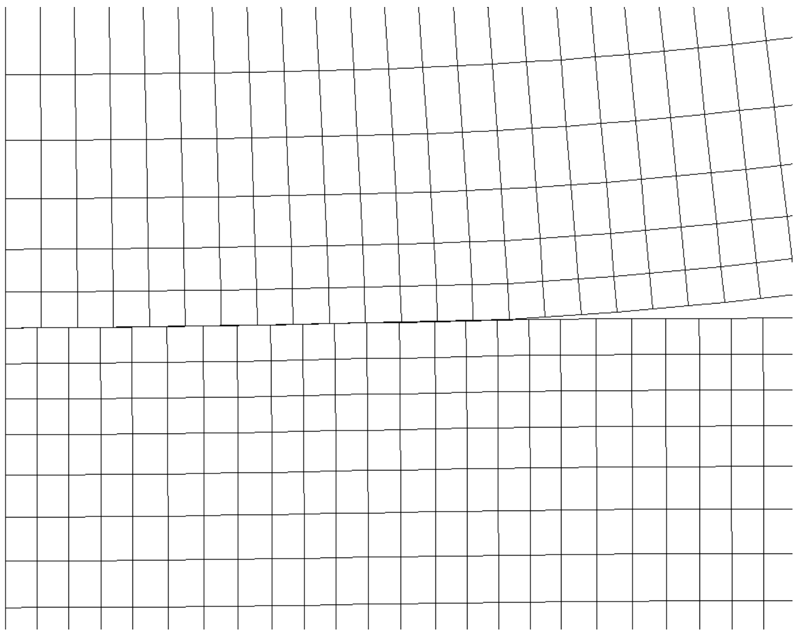


Figure 8.8: Deformed mesh, non-linear problem.

8.3.2 The Pin-in-Hole Contact Problem

Consider a problem of the circular pin in circular hole with small clearance. The radius of the hole is 1 m and the pin has its radius by 1% smaller. Again, the 2D problem is modelled with 3D elements. The model consists of 15844 tri-linear elements and 28828 nodes. The pin is loaded along its centre line by 133 MN/m. The geometric non-linearity was considered. The material properties are the same as in the previous case.

Fig. 8.9 shows von Mises stress distribution on the deformed mesh.

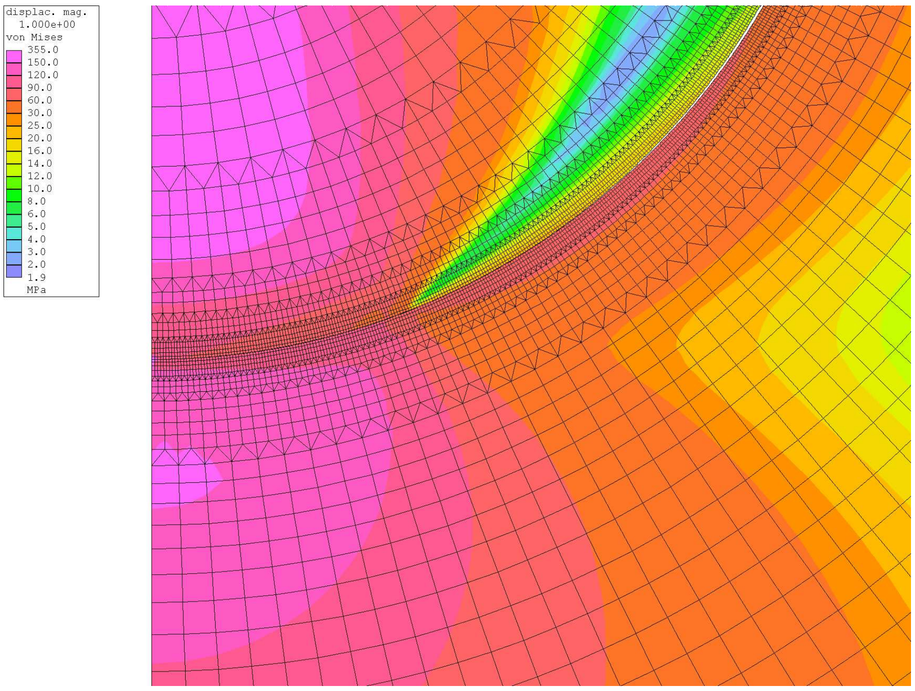


Figure 8.9: The pin-in-hole problem: deformed mesh, von Mises stress, geometrically non-linear problem

Fig. 8.10 depicts results in terms of distributions of the normal contact stress along surfaces of both the pin and hole from the plane of symmetry upwards. Both curves are practically identical, but they are not quite smooth. It is caused by the fact that we used the linear finite elements with different sizes for modelling the hole and pin, so that the mesh was general with faceted surfaces.

Bibliography

[1] Z. Dostál. Inexact Semi-monotonic Augmented Lagrangians with Optimal Feasibility Convergence for Convex Bound and Equality Constrained Quadratic Programming. *SIAM Journal on Numerical Analysis*, 43(2):96–115, 2005.

[2] Z. Dostál, D. Horák, and R. Kučera. Total FETI - an Easier Implementable Variant of the FETI Method for Numerical Solution of Elliptic PDE. *Communications in Numerical Methods in Engineering, to be published.*

[3] Z. Dostál, D. Horák, R. Kučera, V. Vondrák, J. Haslinger, J. Dobiáš, and S. Pták. FETI Based Algorithms for Contact Problems: Scalability, Large Displacements and 3D Coulomb Friction. *Computer Methods in Applied Mechanics and Engineering*, 194(2–5):395–409, 2005.

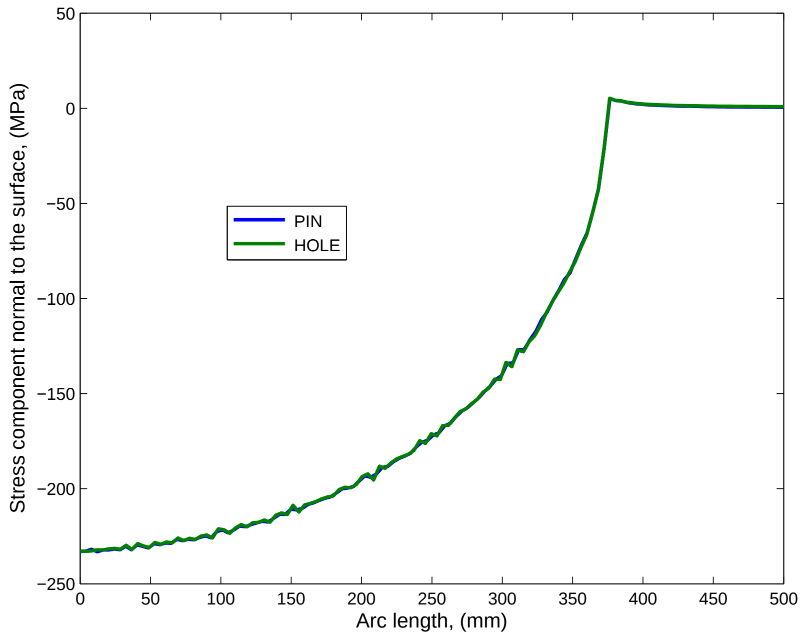


Figure 8.10: Normal stress distribution

[4] Z. Dostál and J. Schöberl. Minimizing Quadratic Functions over Non-negative Cone with the Rate of Convergence and Finite Termination. *Computational Optimization and Application*, 30(1):23–43, 2005.

[5] Ch. Farhat and M.G Géradin. On the General Solution by a Direct Method of a Large-scale Singular System of Linear Equations: Application to the Analysis of Floating Structures. *International Journal for Numerical Methods in Engineering*, 41(7):675–696, 1998.

[6] Ch. Farhat, J. Mandel, and F.-X. Roux. Optimal Convergence Properties of the FETI Domain Decomposition Method. *Computer Methods in Applied Mechanics and Engineering*, 115(5):365–385, 1994.

[7] Ch. Farhat and F.-X. Roux. A Method of Finite Element Tearing and Interconnecting and its Parallel Solution Algorithm. *International Journal for Numerical Methods in Engineering*, 32(12):1205–1227, 1991.

[8] T.A. Laursen. *Computational Contact and Impact Mechanics*. Springer-Verlag, 2002.

[9] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford Science Publication, 1999.

[10] B Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition*. Cambridge University Press, 1996.

[11] A. Toselli and O. Widlund. *Domain Decomposition Methods – Algorithms and Theory*. Springer, 2004.

Chapter 9

Solution of nonlinear equilibrium equations – BFGS method

This part was written and is maintained by Dušan Gabriel. More details about the author can be found in the Chapter 16.

Consider a discrete system of governing equilibrium equations of the form of residual vector $\mathbf{g}(\mathbf{u})$

$$\mathbf{g}(\mathbf{u}) = \mathbf{F}(\mathbf{u}) - \mathbf{R}(\mathbf{u}) = \mathbf{0}, \tag{9.1}$$

where $\mathbf{u}$ represents the solution vector, $\mathbf{F}$ the internal force vector depending (nonlinearly) on $\mathbf{u}$, and $\mathbf{R}$ the prescribed vector of the external forces acting on the nodal points. The vectors $\mathbf{u}, \mathbf{F}, \mathbf{R}$ (all of length `LSOL` denoting the total number of degrees of freedom) have been generated by a finite element discretization. The most frequently used iteration scheme for the solution of nonlinear finite element equations is the Newton-Raphson method (NR). Assume that we have evaluated approximation of solution $\mathbf{u}_i$

$$\mathbf{g}(\mathbf{u}_i) \neq \mathbf{0}, \tag{9.2}$$

where i is the iteration counter. Denote $\mathbf{g}_i = \mathbf{g}(\mathbf{u}_i)$ and perform the Taylor expansion of (9.1) about $\mathbf{u}_i$

$$\mathbf{g}(\mathbf{u}) = \mathbf{g}_i + \mathbf{K}_i(\mathbf{u} - \mathbf{u}_i) + \mathbf{O}(\mathbf{u} - \mathbf{u}_i)^2 = \mathbf{0}, \tag{9.3}$$

where

$$\mathbf{K}_i = \left.\frac{\partial \mathbf{g}}{\partial \mathbf{u}}\right|_{\mathbf{u}_i} \tag{9.4}$$

is the Jacobian (tangential) matrix and $\mathbf{O}(\mathbf{u} - \mathbf{u}_i)^2$ represents an approximation error. Neglecting the higher-order terms in (9.3), we get the relation for a new approximation $\mathbf{u}_{i+1}$

$$\mathbf{K}_i(\mathbf{u}_{i+1} - \mathbf{u}_i) = -\mathbf{g}_i. \tag{9.5}$$

The major disadvantage of the Newton-Raphson method is the need for a fresh refactorization every time the tangential matrix $\mathbf{K}_i$ is formed. Such frequent factorizations require a lot of CPU time, which might sometimes be prohibitive. Another drawback of the method is its potential divergence. Replacing $\mathbf{K}_i$ with $\mathbf{K}_0$, where $\mathbf{K}_0$ is the tangential matrix computed at the beginning of each loading step and then kept constant, we arrive at the modified Newton-Raphson method (MNR), which is relatively robust, but its convergence is slow [6].

In this section an alternative approach to the Newton-Raphson methods, known as quasi-Newton methods, is discussed. We consider the most popular quasi-Newton solver for finite element applications, the BFGS (Broyden-Fletcher-Goldfarb-Shanno) method [4]. Several BFGS implementations are possible for the solution of (9.1). Based on the general framework given in [4] we outline the implementation utilized in the finite element code PMD (Package for Machine Design [7]), which is fully documented in [6].

9.1 Line search

The line search is crucial for a general success of quasi-Newton algorithms. Instead of taking $\mathbf{u}_{i+1}$ as an approximation, we define a searching direction $\mathbf{d}_{i+1}$ as

$$\mathbf{d}_{i+1} = \mathbf{u}_{i+1} - \mathbf{u}_i, \tag{9.6}$$

which we try to improve by a suitable multiplier β. It cannot be expected that the new guess will find exact solution $\mathbf{g}(\mathbf{u}_i + \beta\mathbf{d}_{i+1}) = \mathbf{0}$. However, it is possible to choose β so that the component $\mathbf{g}$ in the search direction $\mathbf{d}_{i+1}$ is zero

$$G(\beta) = \mathbf{d}_{i+1}^{\mathrm{T}}\mathbf{g}(\mathbf{u}_i + \beta\mathbf{d}_{i+1}) = 0. \tag{9.7}$$

To solve the non-linear algebraic equation (9.7), we first bracket the solution by guesses $\beta \in \{\beta_k\} = \{1, 2, 4, 8, 16\}$ and stop the search once $G(\beta)$ has changed its sign. The finer adjustement of β is done by the Illinois algorithm [4], which is an accelerated secant method. The criteria of convergence is

$$G(\beta) \leq \mathtt{STOL} * G(0) = \mathtt{STOL} * \mathbf{d}_{i+1}^{\mathrm{T}}\mathbf{g}(\mathbf{u}_i). \tag{9.8}$$

Since too accurate the line search can unnecessarily increase the total cost of computation, it is important to choose the search tolerance `STOL` efficiently. In most references the tolerance `STOL` is typically set to `STOL` $= 0.5$ ([4], [3]). In case that function (9.7) has not change its sign, we take as the last value $\beta = 16$.

9.2 BFGS method

The idea of the quasi-Newton methods is the replacement of the tangential matrix $\mathbf{K}_i$ with a secant matrix $\tilde{\mathbf{K}}_i$, which can be constructed as

$$\tilde{\mathbf{K}}_i\Delta\mathbf{u}_i = \Delta\mathbf{g}_i \tag{9.9}$$

where increments

$$\begin{aligned} \Delta\mathbf{u}_i &= \mathbf{u}_i - \mathbf{u}_{i-1}, \\ \Delta\mathbf{g}_i &= \mathbf{g}(\mathbf{u}_i) - \mathbf{g}(\mathbf{u}_{i-1}) \end{aligned} \tag{9.10}$$

are determined from the current and previous iteration steps. Having established $\tilde{\mathbf{K}}_i$, we can solve

$$\tilde{\mathbf{K}}_i\mathbf{d}_{i+1} = -\mathbf{g}_i \tag{9.11}$$

and the line search described in Section 9.1 can be used to find the optimum parameter β according to the equation (9.8). Finally, the new approximation of solution $\mathbf{u}_{i+1}$ is updated

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \beta \mathbf{d}_{i+1}. \tag{9.12}$$

Note that the secant matrix $\tilde{\mathbf{K}}_i$ is not uniquely defined by (9.9) (with the exception of one-dimensional case). The idea of Davidon is to update $\tilde{\mathbf{K}}_i$ or $\tilde{\mathbf{K}}_i^{-1}$ in a simple way after each iteration, rather than to recompute it entirely (NR) or leave it unchanged (MNR). In the BFGS method we apply symmetric transformation to the inverse of matrix

$$\tilde{\mathbf{K}}_i^{-1} = \mathbf{A}_i^{\mathrm{T}} \tilde{\mathbf{K}}_{i-1}^{-1} \mathbf{A}_i, \tag{9.13}$$

where

$$\mathbf{A}_i = \mathbf{I} + \mathbf{v}_i \mathbf{w}_i^{\mathrm{T}}. \tag{9.14}$$

The auxiliary vectors $\mathbf{v}_i, \mathbf{w}_i$ are defined by (9.17). Substituting into (9.11) we get

$$\mathbf{d}_{i+1} = -\mathbf{A}_i^{\mathrm{T}} \tilde{\mathbf{K}}_{i-1}^{-1} \mathbf{A}_i \mathbf{g}_i \tag{9.15}$$

or, by recursion

$$\mathbf{d}_{i+1} = -\left[\prod_{m=1}^{i} (\mathbf{I} + \mathbf{w}_m \mathbf{v}_m^{\mathrm{T}}) \right] \mathbf{K}_0^{-1} \left[\prod_{n=1}^{i} (\mathbf{I} + \mathbf{v}_n \mathbf{w}_n^{\mathrm{T}}) \right] \mathbf{g}_i \ . \tag{9.16}$$

Thus, in contrast to Newton-Raphson the updated inverses $\tilde{\mathbf{K}}_i^{-1}$ are not explicitly computed and stored. Instead, the updated search direction is more economically calculated by means of vector dot products, addition operations and a backsolve using the factorized matrix $\tilde{\mathbf{K}}_0$ done in the initialization of iteration process. Another important advantage of the algorithm is that the number of auxiliary vectors $\mathbf{v}_i, \mathbf{w}_i$ can be kept reasonably small. They are set to satisfy (9.9) as

$$\begin{aligned} \mathbf{v}_i &= -\sqrt{\frac{\Delta \mathbf{u}_i^{\mathrm{T}} \Delta \mathbf{g}_i}{\Delta \mathbf{u}_i^{\mathrm{T}} \tilde{\mathbf{K}}_{i-1} \Delta \mathbf{u}_i}} \tilde{\mathbf{K}}_{i-1} \Delta \mathbf{u}_i - \Delta \mathbf{g}_i \ , \\ \mathbf{w}_i &= \frac{\Delta \mathbf{u}_i}{\Delta \mathbf{u}_i^{\mathrm{T}} \mathbf{g}_i} \end{aligned} \tag{9.17}$$

or substituting from (9.7) and (9.11)

$$\begin{aligned} \mathbf{v}_i &= \left(1 + \frac{\beta}{c}\right) \mathbf{g}_{i-1} - \mathbf{g}_i \ , \\ \mathbf{w}_i &= \frac{\Delta \mathbf{u}_i}{G(\beta) - G(0)} \ , \end{aligned} \tag{9.18}$$

where

$$c = \sqrt{\frac{\beta G(0)}{G(0) - G(\beta)}} \ . \tag{9.19}$$

We show that c is also the condition number. The eigenvalues of matrix $\mathbf{A}$ are

$$\lambda_j = \begin{cases} 1 \ , & j = 1, 2, \ldots, N-1 \\ 1 + \mathbf{w}_i^{\mathrm{T}} \mathbf{v}_i \ , & j = N \ . \end{cases} \tag{9.20}$$

Realizing that the condition number as $c = |\lambda_N / \lambda_1|$, i.e.

Template 48, BFGS algorithm

```
1. Initialize the iteration process
    (a) Set iteration counter: $i = 0$
    (b) Initialize values: $\mathbf{R}_0, \tilde{\mathbf{K}}_0 = \mathbf{K}_{\text{elastic}}, \mathbf{u}_0 = \mathbf{u}_{\text{elastic}}$ (initial guess)
    (c) Evaluate: $\tilde{\mathbf{K}}_0^{-1}, \mathbf{g}(0), \mathbf{g}(\mathbf{u}_0)$
2. Loop on $i$ (iteration counter) for equilibrium
    (a) Compute search direction: $\tilde{\mathbf{K}}_i \mathbf{d}_{i+1} = \mathbf{g}(\mathbf{u}_i)$
    (b) Line search and solution vector update:
        i. Evaluate: $G(0) = \mathbf{d}_{i+1}^{\mathrm{T}} \mathbf{g}(\mathbf{u}_i)$
        ii. Loop over $\beta$; $\beta \in \{\beta_k\} = \{1, 2, 4, 8, 16\}$
            • Evaluate: $G(\beta) = \mathbf{d}_{i+1}^{\mathrm{T}} \mathbf{g}(\mathbf{u}_i + \beta \mathbf{d}_{i+1})$
            • IF $G(\beta) \leq \texttt{STOL} * G(0)$ THEN go to step iii
            • IF $G(\beta)$ changed its sign THEN
                – finer adjustment of $\beta \in \langle \beta_{k-1}, \beta_k \rangle$ by means of accelerated secant
                  method (Illinois algorithm [4])
                ENDIF
        iii. Update: $\mathbf{u}_{i+1} = \mathbf{u}_i + \beta \mathbf{d}_{i+1}$
    (c) Equilibrium check
        IF ($\|\mathbf{g}(\mathbf{u}_{i+1})\| < \texttt{RTOL} \|\mathbf{g}(0)\|$) THEN equilibrium achieved $\rightarrow$ EXIT
    (d) Increment iteration counter: $i = i + 1$
    (e) Stability check
        i. Evaluate the condition number: $c = \sqrt{\dfrac{\beta G_0}{G_0 - G(\beta)}}$
        ii. IF $c > c_{\text{crit}} \approx 5$ THEN
            • Take previous quasi-secant matrix: $\tilde{\mathbf{K}}_i = \tilde{\mathbf{K}}_{i-1}$
            • go to step 2a
    (f) Perform BFGS update
        i. Evaluate: $\Delta \mathbf{u}_i = \mathbf{u}_i - \mathbf{u}_{i-1}$ , $\Delta \mathbf{g}_i = \mathbf{g}(\mathbf{u}_i) - \mathbf{g}(\mathbf{u}_{i-1})$
        ii. Compute BFGS auxiliary vectors:
            $\mathbf{v}_i = -\sqrt{\dfrac{\Delta \mathbf{u}_i^{\mathrm{T}} \Delta \mathbf{g}_i}{\Delta \mathbf{u}_i^{\mathrm{T}} \tilde{\mathbf{K}}_{i-1} \Delta \mathbf{u}_i}} \tilde{\mathbf{K}}_{i-1} \Delta \mathbf{u}_i - \Delta \mathbf{g}_i$
            $\mathbf{w}_i = \dfrac{\Delta \mathbf{u}_i}{\Delta \mathbf{u}_i^{\mathrm{T}} \mathbf{g}_i}$
        iii. Compute the inverse quasi-secant matrix: $\tilde{\mathbf{K}}_i^{-1} = (\mathbf{I} + \mathbf{w}_i \mathbf{v}_i^{\mathrm{T}}) \tilde{\mathbf{K}}_{i-1}^{-1} (\mathbf{I} + \mathbf{v}_i \mathbf{w}_i^{\mathrm{T}})$
        iv. go to step 2a
```

$$c = \left|1 + \mathbf{w}_i^{\mathrm{T}} \mathbf{v}_i\right| \ , \tag{9.21}$$

which is equal to (9.19). In most references the critical value of c is taken as 10^5. However, realizing that $\beta \in \langle 0, 16 \rangle$ and $G(\beta)/G(0) \leq 1$ at most but probable $G(\beta)/G(0) \leq$ `STOL`, the critical value is set to a more safe value $c \approx 5$, which follows from the Fig. 9.1. Otherwise, the transformation (9.13) becomes nearly singular.

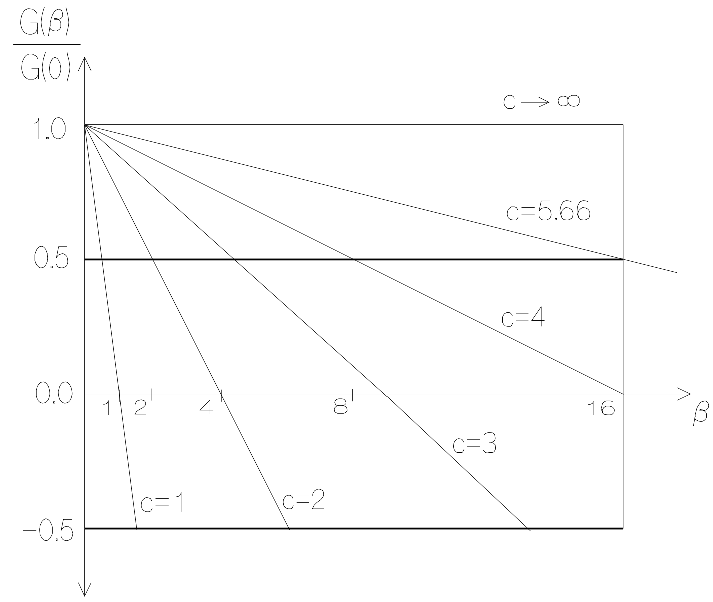


Figure 9.1: Contours of the condition number c (adopted from [6]).

The implementation of the BFGS algorithm utilized in the finite element code PMD is outlined in Template 48. The algorithm starts with an initialization of iteration process. The elastic solution is usually taken as the starting approximation. In the iteration loop, we first compute the search direction $\mathbf{d}_{i+1}$ (step 2a), which we improve by the line search.

When the optimum multiplier β is found and the solution vector $\mathbf{u}_{i+1}$ is updated, the equilibrium check follows in step 2c. The user-prescribed tolerance `RTOL` related to the residual norm $\|\mathbf{g}(0)\|$ is usually set to 10^{-3}. In the case that equilibrium is achieved, the calculation is finished. Otherwise, the algorithm continues with a stability check procedure, which consists in the evaluation of condition number c (9.19). Finally, BFGS update is performed in step 2f, where the inverse of the quasi-Newton secant matrix

$$\tilde{\mathbf{K}}_i^{-1} = (\mathbf{I} + \mathbf{w}_i \mathbf{v}_i^{\mathrm{T}}) \tilde{\mathbf{K}}_{i-1}^{-1} (\mathbf{I} + \mathbf{v}_i \mathbf{w}_i^{\mathrm{T}}) \tag{9.22}$$

is iteratively updated by means of auxiliary vectors $\mathbf{v}_i, \mathbf{w}_i$ (9.17).

9.3 The BFGS method for a constrained system

In this section we outline the modification of the BFGS method for constrained nonlinear system that results, for example, from the finite element discretization of contact problem where the contact conditions are enforced by the penalty method [1, 2]. The penalty parameter must be chosen as large as possible, in most cases several orders of magnitude greater than the stiffness of support, which makes considerable demands on the solution technique.

9.3.1 A simple model problem

In order to demonstrate the behaviour of the proposed solution scheme, we now present a solution of a simple model problem. We consider a one-dimensional, one degree of freedom mechanical system, subject to a single constraint: a simple linear spring with stiffness k, whose right end contacts against an obstacle. The spring is loaded by external force F_{ext} (see Fig. 9.2). This contact imposes a unilateral constraint on unknown displacement u, allowing a gap to open but preventing from penetration. The solution to this system is apparent: if $F_{\text{ext}} < 0$, then $u = F_{\text{ext}}/k$; if $F_{\text{ext}} \geq 0$, then $u = 0$ to avoid penetration.

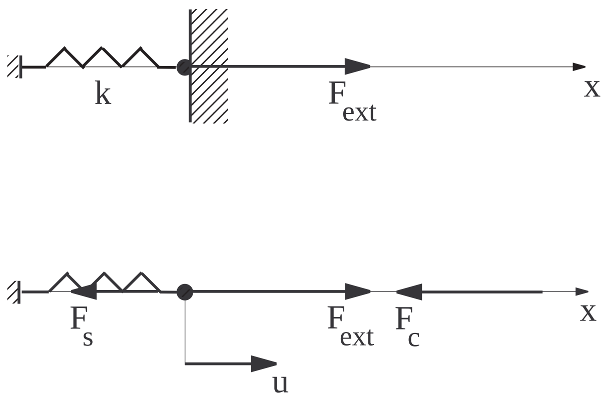


Figure 9.2: Simple one dimensional spring system subject to a single constraint.

Denoting by F_s the force produced by the spring and $F_c = \xi d$ the penalty force expressed as a linear function of the penetration depth d multiplied by penalty parameter ξ, equilibrium is enforced through the definition of the residual force G

$$G = F_{\text{s}} + F_{\text{c}} - F_{\text{ext}}. \tag{9.23}$$

When $G = 0$, the equilibrium condition is satisfied. In addition to the equilibrium condition (9.23), u and F_c are subject to the contact conditions

$$u \leq 0 \quad F_{\text{c}} \geq 0 \quad F_{\text{c}}\, u = 0. \tag{9.24}$$

A graphical illustration of solution of this example with penalty constraint, plotted as the residual force G versus u, is shown in Fig. 9.3, which represents polygonal characteristic (solid line) with slopes given by the penalty parameter ξ (for $u > 0$) and the spring stiffness k (for $u < 0$), denoted by $\texttt{KCS} = 1$. We should emphasize that the final numerical solution ($G = 0$) is always entailed by undesirable penetration d_{final}, which violates conditions (9.24).

Now, we demonstrate the BFGS and MNR iteration process. Both methods start at point '0' directed by initial stiffness k toward point '1' corresponding to elastic solution u_{elastic} for unconstrained spring. There contact force F_c induces an increase of the residual G. As a result, the spring rebounds to point '2', from where the MNR method keeping

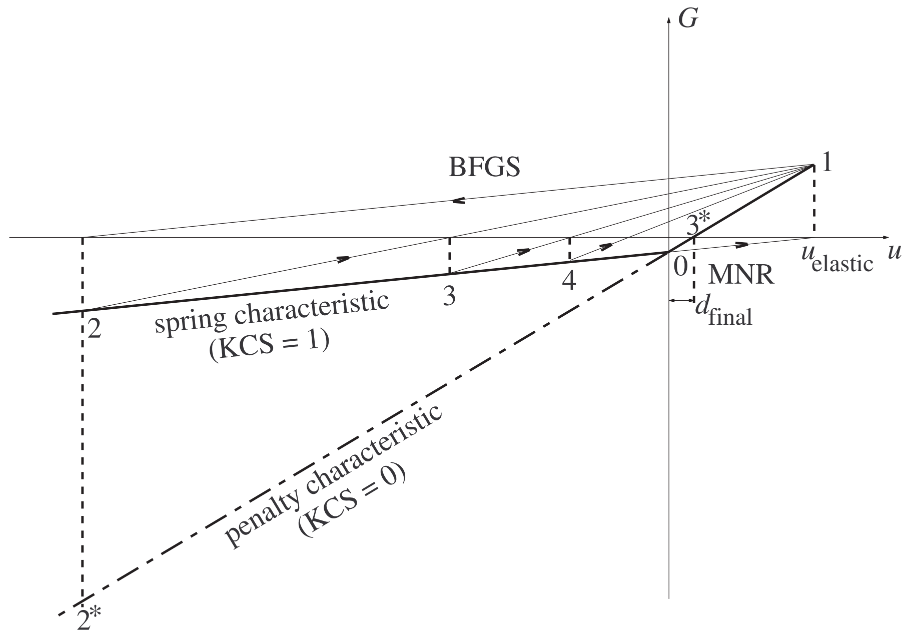


Figure 9.3: The BFGS and MNR iterations for a single degree of freedom system with penalty constraint.

initial direction regresses to point '1' and the cycle is repeated. The BFGS method using algorithmic secant based on the curent and previous iteration step gradually drifts to the solution but the convergence is very slow (points '3', '4',...).

In order to improve the convergence properties we fix spring to obstacle by penalty force after initial bounce has been encountered. In graphical interpretation it represents the replacement of the spring part of characteristic with penalty one for $u < 0$ (dash-and-dot line), denoted by `KCS` $= 0$ in Fig. 9.3. Then, the solution of the spring model is obtained in three steps for the BFGS method (points '2*', '3*'), whereas the MNR method still diverges.

9.3.2 The modified BFGS algorithm

Now, we generalize the discussion of this simple model by considering the multiple degrees of freedom discrete system of governing equilibrium equations (see equation (18) in [2]), which is useful to rewrite in the form of the residual vector $\mathbf{g}(\mathbf{u}_i)$

$$\mathbf{g}(\mathbf{u}_i) = \mathbf{F}(\mathbf{u}_i) - \mathbf{R}(\mathbf{u}_i) - \mathbf{R}_{c1}(\mathbf{u}_i) - \mathbf{R}_{c2}(\mathbf{u}_i) \tag{9.25}$$

or briefly

$$\mathbf{g}_i = \mathbf{F}_i - \mathbf{R}_i - \mathbf{R}_{c1i} - \mathbf{R}_{c2i}. \tag{9.26}$$

The vectors $\mathbf{u}, \mathbf{F}, \mathbf{R}$ and $\mathbf{R}_{c1}, \mathbf{R}_{c2}$ are of length `LSOL`. The vectors $\mathbf{R}_{c1}, \mathbf{R}_{c2}$ contain penalty forces resulting from finite element discretization. Similarly, as in one dimensional example the iteration process is controlled by setting of key `KCS`: `KCS` $= 1$ means that contact search is performed while `KCS` $= 0$ corresponds to the case when existing contact surfaces are only re-established regardless of sign the penetration d_{IG} [2]. The implementation of the BFGS algorithm for constrained nonlinear system is outlined in Template 49.

The crucial point of the algorithm is the following: after the contact search has been performed in the evaluation of $\mathbf{g}(\mathbf{u}_0)$ the assigned contact surfaces are sticked together by penalty tractions ($\texttt{KCS} = 0$) regardless of their signs. In a general 3D case it cannot be expected that the next approximation will find solution $\mathbf{g}(\mathbf{u})$ as in the model example, thus the calculation must be performed until the equilibrium is reached (see Example 9.3.3). Then, the verification of the correct contact state (step 3) is necessary since tension tractions can occur on contact boundaries. All the penalty constraints are removed and the convergence condition is tested again with $\texttt{KCS} = 1$. If it is not satisfied, the algorithm continues with the iteration process with re-initialization $\tilde{\mathbf{K}}_0 = \mathbf{K}_{\text{elastic}}, \mathbf{u}_0 = \mathbf{u}_\text{c}$ and $\texttt{KCS} = 1$ by returning to step 2.

Template 49, BFGS algorithm for constrained nonlinear system.

```
1. Initialize the iteration process
    (a) Set initial parameters: $i = 0, \texttt{KCS} = 1$
    (b) Initialize values: $\mathbf{R}_0, \tilde{\mathbf{K}}_0 = \mathbf{K}_{\text{elastic}}, \mathbf{u}_0 = \mathbf{u}_{\text{elastic}}$ (initial guess)
    (c) Evaluate: $\tilde{\mathbf{K}}_0^{-1}, \mathbf{g}(0), \mathbf{g}(\mathbf{u}_0)$
    (d) Set key: $\texttt{KCS} = 0$
2. Loop on $i$ (iteration counter) for equilibrium
    (a) Compute search direction: $\tilde{\mathbf{K}}_i \mathbf{d}_{i+1} = \mathbf{g}(\mathbf{u}_i)$
    (b) Line search and solution vector update:
        i. Evaluate $G(0) = \mathbf{d}_{i+1}^{\mathrm{T}} \mathbf{g}(\mathbf{u}_i)$
        ii. Loop over $\beta$; $\beta \in \{\beta_k\} = \{1, 2, 4, 8, 16\}$
            • Evaluate: $G(\beta) = \mathbf{d}_{i+1}^{\mathrm{T}} \mathbf{g}(\mathbf{u}_i + \beta \mathbf{d}_{i+1})$
            • IF $G(\beta) \le \texttt{STOL} * G(0)$ THEN go to step iii
            • IF $G(\beta)$ changed its sign THEN
                – finer adjustment of $\beta \in \langle \beta_{k-1}, \beta_k \rangle$ by means of accelerated secant
                  method (Illinois algorithm [4])
              ENDIF
        iii. Update: $\mathbf{u}_{i+1} = \mathbf{u}_i + \beta \mathbf{d}_{i+1}$
    (c) Equilibrium check
        IF ($\|\mathbf{g}(\mathbf{u}_{i+1})\| < \texttt{RTOL}\|\mathbf{g}(0)\|$) THEN
            • IF ($\texttt{KCS} = 1$) THEN equilibrium achieved $\rightarrow$ EXIT
            • go to step 3
        ENDIF
    (d) Increment iteration counter: $i = i + 1$
    (e) Stability check
        i. Evaluate the condition number: $c = \sqrt{\dfrac{\beta G_0}{G_0 - G(\beta)}}$
        ii. IF $c > c_{\text{crit}} \approx 5$ THEN
            • Take previous quasi-secant matrix: $\tilde{\mathbf{K}}_i = \tilde{\mathbf{K}}_{i-1}$
            • go to step 2a
    (f) Perform BFGS update
        i. Evaluate: $\Delta \mathbf{u}_i = \mathbf{u}_i - \mathbf{u}_{i-1}$ , $\Delta \mathbf{g}_i = \mathbf{g}(\mathbf{u}_i) - \mathbf{g}(\mathbf{u}_{i-1})$
        ii. Compute BFGS auxiliary vectors:
            $\mathbf{v}_i = -\sqrt{\dfrac{\Delta \mathbf{u}_i^{\mathrm{T}} \Delta \mathbf{g}_i}{\Delta \mathbf{u}_i^{\mathrm{T}} \tilde{\mathbf{K}}_{i-1} \Delta \mathbf{u}_i}} \tilde{\mathbf{K}}_{i-1} \Delta \mathbf{u}_i - \Delta \mathbf{g}_i, \quad \mathbf{w}_i = \dfrac{\Delta \mathbf{u}_i}{\Delta \mathbf{u}_i^{\mathrm{T}} \mathbf{g}_i}$
        iii. Compute the inverse quasi-secant matrix: $\tilde{\mathbf{K}}_i^{-1} = (\mathbf{I} + \mathbf{w}_i \mathbf{v}_i^{\mathrm{T}}) \tilde{\mathbf{K}}_{i-1}^{-1} (\mathbf{I} + \mathbf{v}_i \mathbf{w}_i^{\mathrm{T}})$
        iv. go to step 2a
3. Check on correct contact state on contact boundaries
    (a) Remove all penalty constraint at Gauss points
    (b) Set parameters: $i = 0$, $\texttt{KCS} = 1$
    (c) Equilibrium check with previous converged solution $\mathbf{u}_{\mathrm{c}}$:
        IF ($\|\mathbf{g}(\mathbf{u}_{\mathrm{c}})\| < \texttt{RTOL}\|\mathbf{g}(0)\|$) THEN equilibrium achieved $\rightarrow$ EXIT
    (d) Re-initialize values: $\tilde{\mathbf{K}}_0 = \mathbf{K}_{\text{elastic}}, \mathbf{u}_0 = \mathbf{u}_{\mathrm{c}}$
    (e) go to step 2
```

In fact, the box is virtually identical to the algorithm outlined in Template 48, which is executed twice, firstly with KCS $= 0$ and then again with KCS $= 1$. The converged solution of the first run $\mathbf{u}_c$ serves as an approximation for the second one. The capability of this procedure was confirmed in a number of problems [1, 2].

9.3.3 Example: Two cubes contact

Let us consider the two cubes shown in Fig. 9.4 subjected to uniformly distributed pressure p, interacting over their common contact interface [5]. The pressure acting on bottom face is replaced with the equivalent reactions introduced by appropriate boundary conditions. The top cube is suspended on soft regularization springs of total stiffness k.

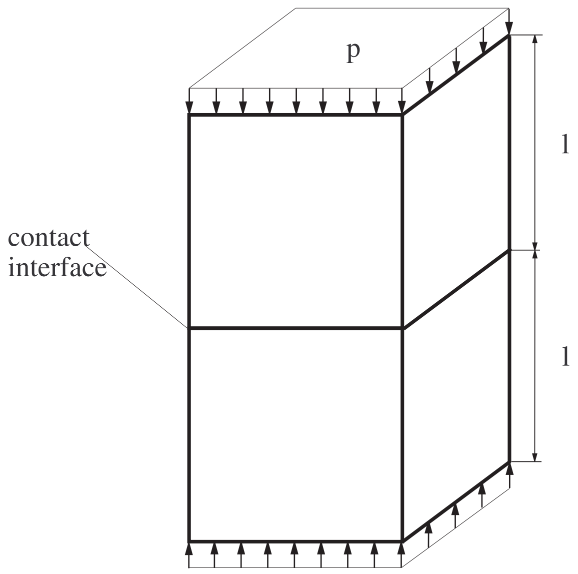


Figure 9.4: Two cubes subjected to uniformly distributed pressure p, interacting over their common contact interface. Geometry: $l = 1$ [m]. Material properties: Young modulus $E = 2.1\times\ 10^5$ [MPa], Poisson's ratio $\nu = 0.3$. Loading: $p = 10$ [MPa].

This example demonstrates a generalization of a simple one-dimensional spring model described in Section 9.3.1. We investigated an accuracy of the numerical solution for various values of the penalty parameter ξ and for different stiffnesses of the regularization springs k. The stiffness k was by several orders of magnitude smaller than the stiffnesses of the cubes so that the results were not significantly affected. The performance of both linear and quadratic isoparametric elements was tested.

The results are summarized in Tab. 9.1, where relative displacement errors ϵ calculated as the magnitudes of penetrations related to the displacements of cubes and the numbers of iterations NITER (linear elements/quadratic elements) are shown. The divergence of calculation is denoted by symbol '***', the results were obtained within one load step.

Similarly, as in one dimensional example the initial rebound of the top cube arose. Its amount, which was directly proportional to ξ and inversely proportional to k, was enormous (e.g. for $\xi = 10^{14}$ [N/m^3] and $k = 10^7$ [N/m] the initial rebound was 10^7 [m]).

k	ξ[N/m^3]									
	10^{12}		10^{13}		10^{14}		10^{15}		10^{16}	
[N/m]	NITER	ϵ[%]	NITER	ϵ[%]	NITER	ϵ[%]	NITER	ϵ[%]	NITER	ϵ[%]
10^6	5/11	20	8/16	2	11/***	0.2	***/***	***	***/***	***
10^7	5/9	20	5/15	2	8/***	0.2	11/***	0.02	***/***	***
10^8	5/7	20	5/12	2	8/14	0.2	9/***	0.02	9/***	0.002
10^9	4/5	20	4/9	2	5/13	0.2	9/17	0.02	8/***	0.002

Table 9.1: Relative displacement errors ϵ and numbers of iterations `NITER`.

Although it was reduced by the line search the nodal displacements were affected by round-off errors leading to the distortion of the contact plane. This explains the increase of number of iterations `NITER` or even the divergence of the calculation of normal vector with increasing ξ or with decreasing k in Tab. 9.1. This effect is particularly apparent for quadratic mesh since a larger distortion of contact plane occurs. Note that a number of iterations is always greater than three in contrast to one dimensional spring model. It is caused by the 3D effect when besides huge rebound large compression of the top cube occurs due to presence of regularization springs.

Bibliography

[1] D. Gabriel. Numerical solution of large displacement contact problems by the finite element method. *CTU Reports*, 7(3):1–75, 2003.

[2] Plešek J. Gabriel, D. and M. Ulbin. Symmetry preserving algorithm for large displacement frictionless contact by the pre-discretization penalty method. *Int. J. Num. Met. Engng.*, 61(15):2615–2638, 2004.

[3] S.H. Lee. Rudimentary considerations for effective line search method in nonlinear finite element analysis. *Comput. Struct.*, 32(6):1287–1301, 1989.

[4] H. Matthies and G. Strang. The solution of nonlinear finite element equations. *Int. J. Num. Met. Engng.*, 14:1613–1626, 1979.

[5] J. Plešek and D. Gabriel. *PMD example manual.* Institute of Thermomechanics, Academy of Sciences of the Czech Republic, 2000.

[6] J. Plešek. Solution of nonlinear equilibrium problems: The quasi-newton methods. In M. Okrouhlík, editor, *Implementation of Nonlinear Continuum Mechanics in Finite Element Codes*, pages 216–226, 1995.

[7] VAMET/Institute of Thermomechanics. *PMD version f77.9*, 2003.

Chapter 10

The Frontal Solution Technique

This part was written and is maintained by Svatopluk Pták. More details about the author can be found in the Chapter 16.

10.1 Introduction to the solution of algebraic systems

An algebraic system – regardless of being linear or not, and of producing one, many or none solutions – can be generally written as

$$F\,(x,\,d) = 0\;, \tag{10.1}$$

where x and d are data sets and F is the functional relation between x and d. According to the kind of problem, the variables x and d may be real numbers, vectors or functions.

The equation (10.1) can represent

- a *direct problem* if F and d are given and x is the unknown,
- an *inverse problem* if F and x are known and d is the unknown and
- an *identification problem* if x and d are given while the functional relation F is to be sought (identification problems will not be dealt in the next text).

Problem (10.1) is *well posed* or *stable* if it admits to obtain a unique unknown data set x (or d) which depends with continuity on given F and d (or F and x).
The problem which does not enjoy the property above is called *ill posed* or *unstable* and before undertaking its numerical solution it has to be regularized, that is, it must be suitably transformed into a well posed one [25]. Indeed, it is not appropriate to pretend the numerical method can cure the the pathologies of an intrinsically ill-posed problem [26].

Continuous dependence on the data means that small perturbations δd on the data d yield small changes δx in the solution data x, i.e.

$$\begin{array}{c} \text{if} \qquad F\,(x+\delta x,\, d+\delta d) = 0\;, \qquad \text{than} \\ \forall \eta > 0,\, \exists\, K(\eta,\, d) : \|\delta d\| < \eta \Rightarrow \|\delta x\| \le K(\eta, d)\, \|\delta d\|\;. \end{array} \tag{10.2}$$

The norms used for x, d, etc may not coincide, whenever such variables represent quantities of different physical kinds.

The system property to be well- or ill posed is measured by a *condition number.* This tool quantifies this property either by means of the relative condition number $c(d)$

$$c(d) \;=\; \sup_{\delta d \,\in\, D} \frac{\|\delta d\| \; \|x\|}{\|d\| \; \|\delta x\|} \;, \tag{10.3}$$

or - if the norms of sets x or/and d are zeros - by the absolute condition number $c_a(d)$

$$c_a(d) \;=\; \sup_{\delta d \,\in\, D} \frac{\|\delta d\|}{\|\delta x\|} \;,$$

where D is a neighborhood of the origin and denotes the set of admissible perturbations on the data for which the perturbed problem (10.2) still makes sense.

Problem (10.1) is called ill-conditioned if $c(d)$ is 'big' for *any* admissible data d. The precise meaning of 'small' and 'big' will be added.

The property of a problem of being well-conditioned is independent of the numerical method that is being used to solve it. In fact, it is possible to generate stable as well as unstable numerical schemes for solving well-conditioned problems. For more interesting details see [26].

10.1.1 Matrix forms of algebraic systems

In usual cases, the algebraic system can be written in *matrix* form

$$\mathbf{K}\,\mathbf{x} = \mathbf{f} \qquad \text{or} \qquad \mathbf{K}\,\mathbf{X} \;=\; \mathbf{F}\;, \quad \mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k,]\;, \quad \mathbf{F} = [\mathbf{f}_1, \mathbf{f}_2, ..., \mathbf{f}_k,]\;, \tag{10.4}$$

where
$\mathbf{K}$ stands for given *regular* $(n{\times}n)$ matrix, ie. $\mathrm{rank}(\mathbf{K}) = n$,
$\mathbf{x}, \mathbf{f}$ for the $(n{\times}1)$ vectors collecting unknowns and given items, respectively, and
$\mathbf{X}, \mathbf{F}$ for $(n{\times}k)$ matrices, if solutions for more than one given vector are needed.
Let us call (10.4) as the *model system* and note that this system has due to regularity of $\mathbf{K}$ just the only solution.

Among different formulations of (10.4) let us mention *the nullified form*

$$\mathbf{r}(\mathbf{x}) = \mathbf{0} \qquad \text{or} \qquad \mathbf{f} - \mathbf{K}\mathbf{x} = \mathbf{0}\;, \tag{10.5}$$

i.e. the true solution makes the residuum null, or many other modifications of (10.4) as e.g.

$$\hat{\mathbf{P}}\,\mathbf{K}\,\mathbf{x} \;=\; \hat{\mathbf{P}}\,\mathbf{f}\;, \qquad [\hat{\mathbf{P}}\,\mathbf{K}\,\hat{\mathbf{Q}}]\;\{\hat{\mathbf{Q}}\mathbf{x}\} \;=\; \{\hat{\mathbf{P}}\,\mathbf{f}\}\;, \tag{10.6}$$

which could represent the suitably *permuted* (by the left- or both left and right permutation matrices $\hat{\mathbf{P}}$ and $\hat{\mathbf{Q}}$, see Note 1) and *symmetrized* system

$$\mathbf{K}^{\mathrm{T}}\,\mathbf{K}\,\mathbf{x} \;=\; \mathbf{K}^{\mathrm{T}}\,\mathbf{f}\;, \tag{10.7}$$

or as

$$\mathbf{P}\,\mathbf{K}\,\mathbf{x} \;=\; \mathbf{P}\,\mathbf{f}\;; \qquad\qquad \mathbf{P} = \mathbf{L}\,\mathbf{L}^{\mathrm{T}}\;, \quad \mathbf{L}^{\mathrm{T}}\,\mathbf{K}\,\mathbf{L}\,\mathbf{y} \;=\; \mathbf{L}^{\mathrm{T}}\,\mathbf{f}\;, \;\; \mathbf{x} \;=\; \mathbf{L}\,\mathbf{y}\;, \tag{10.8}$$

which could stand for the *pre-conditioned* and *symmetrically pre-conditioned* equation system, respectively. These all systems are *equivalent* to (10.4) and therefore have the

same solution $\mathbf{x}$ – if only round-off errors would not exist. The mentioned modifications (10.6), (10.7) and (10.8) enable us apply algorithms using of which would be unefficient or impossible for the system in its original form (10.4).
While equations (10.6) represent some reordering of the original system leading especially to reduction of computational and storage demands, to reduction of round-off error propagation, the equations (10.7) and (10.8) transform the original system to suitable reformulated, better conditioned algebraic systems.

Notes:
1) Permutation matrices are very special sparse matrices – identity matrices with reordered rows or columns. They are frequently used in matrix computations to represent the ordering of components in a vector or the rearrangement of rows and columns in a matrix. An *elementary permutation matrix* $\hat{\mathbf{P}}^{(i,j)}$ results from identity matrix $\mathbf{I}$ by a mutual replacement of two its rows i and j; the same holds for $\hat{\mathbf{Q}}^{(i,j)}$, of course. Any *permutation matrix* $\hat{\mathbf{P}}$ – as a product of elementary permutation matrices – results from identity matrix by more replacements of row- or column-pairs in $\mathbf{I}$.
Hence all permutation matrices are regular, symmetric, their determinants are either +1 or -1 and any even power of given permutation matrix equals to identity matrix $\mathbf{I}$ while any odd power of given permutation matrix only reproduces its copy, i.e. $\hat{\mathbf{P}}^{\text{even}} = \mathbf{I}$ and $\hat{\mathbf{P}}^{\text{odd}} = \hat{\mathbf{P}}$. Moreover, as $\hat{\mathbf{P}}\hat{\mathbf{P}}^{\mathrm{T}} = \hat{\mathbf{P}}^{\mathrm{T}}\hat{\mathbf{P}} = \mathbf{I}$, i.e. $\hat{\mathbf{P}}^{\mathrm{T}} = \hat{\mathbf{P}}^{-1}$, is any permutation matrix orthogonal one and as $\mathbf{K}$ and $\hat{\mathbf{P}}^{-1}\mathbf{K}\hat{\mathbf{P}}$ are similar matrices, share the same spectrum and the same characteristic polynomial.
Multiply (10.4) by the permutation matrix $\hat{\mathbf{P}}$ from the left then ordering of equations is changed (destroying the possible symmetry of the system matrix); multiply $\mathbf{x}$ of (10.4) by the permutation matrix $\hat{\mathbf{Q}}$ from the right then ordering of unknowns is changed.
The possible symmetry of original system matrix $\mathbf{K}$ will be retained if same permutation matrices are applied from the left as well as from the right:

$$\hat{\mathbf{P}}\,\mathbf{K}\,\mathbf{x} = \hat{\mathbf{P}}\,\mathbf{f}\,, \qquad [\hat{\mathbf{P}}\,\mathbf{K}\,\hat{\mathbf{Q}}]\,\{\hat{\mathbf{Q}}\,\mathbf{x}\} = \hat{\mathbf{P}}\,\mathbf{f}\,,\;\; \hat{\mathbf{Q}}\,\hat{\mathbf{Q}} = \mathbf{I}\,. \tag{10.9}$$

Performance of *optimizer*- and/or *pivoting* algorithms can be described using of permutation matrices. Optimizers play the key role for systems with sparse matrices especially if direct solvers are used, pivoting is necessary by solving of non-symmetric and/or ill-conditioned systems. By means of permutation matrices are defined important terms of reducibility and irreducibility.

2) A triangular- or, more general, a block-triangular matrix form of system $\mathbf{A}\mathbf{x} = \mathbf{f}$ allows the corresponding set of linear equations to be solved as a sequence of subproblems. A matrix $\mathbf{A}$ which can be permuted to the block lower form (10.10)

$$\hat{\mathbf{P}}\mathbf{A}\hat{\mathbf{Q}} = \begin{bmatrix} \mathbf{B}_{11} & \cdot & \cdot & \cdot & \cdot \\ \mathbf{B}_{21} & \mathbf{B}_{22} & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \mathbf{B}_{N1} & \mathbf{B}_{N2} & \cdot & \cdot & \mathbf{B}_{NN} \end{bmatrix}, \tag{10.10}$$

(though with only minor modification the block upper triangular form could be obtained) with $N > 1$ is said to be *reducible*. If no block triangular form other than the trivial one ($N = 1$) can be found, the matrix is called *irreducible*. We expect each $\mathbf{B}_{ii}$ to be irreducible, for otherwise a finer decomposition is possible. The advantage of using block triangular forms is that the original set of equations may be solved by a simple forward substitution process [6]

$$\mathbf{B}_{ii}\,\mathbf{y}_i \;=\; \{\hat{\mathbf{P}}\,\mathbf{f}\}_i \;-\; \sum_{j=1}^{i-1} \mathbf{B}_{ij}\,\mathbf{y}_j \;,\;\; i = 1, 2, ..., N \tag{10.11}$$

(where the sum is zero for $i = 1$) and the permutation $\mathbf{x} = \hat{\mathbf{Q}}\,\mathbf{y}$. We have to factorize only the diagonal blocks $\mathbf{B}_{ii}\,,\; i = 1, 2, ..., N$. Notice in particular that all fill-in is confined to the diagonal blocks. If row and column interchanges are performed within each diagonal block for sake the stability and sparsity, this will not affect the block triangular structure.

3) As all the operations that have been introduced here can be extended to the case of *block-partitioned* matrices and vectors, provided that size of each single block is such that any single matrix operation is well-defined, we can present (10.4) in the form, e.g.

$$\begin{bmatrix} \mathbf{K}_{11} & \mathbf{K}_{12} \\ \mathbf{K}_{21} & \mathbf{K}_{22} \end{bmatrix} \left\{ \begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \end{array} \right\} = \left\{ \begin{array}{c} \mathbf{f}_1 \\ \mathbf{f}_2 \end{array} \right\} . \tag{10.12}$$

Partial Gaussian elimination of unknowns collected in $\mathbf{x}_1$ leads then to system

$$\begin{bmatrix} \mathbf{K}_{11} & \mathbf{K}_{12} \\ \mathbf{0}_{21} & \mathbf{K}_{22} - \mathbf{K}_{21}\,{\mathbf{K}_{11}}^{-1}\,\mathbf{K}_{12} \end{bmatrix} \left\{ \begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \end{array} \right\} = \left\{ \begin{array}{c} \mathbf{f}_1 \\ \mathbf{f}_2 - \mathbf{K}_{21}\,{\mathbf{K}_{11}}^{-1}\,\mathbf{f}_1 \end{array} \right\} \tag{10.13}$$

where the diagonal block $\mathbf{K}_{22}$ is transformed into the *Schur-complement* of $\mathbf{K}$, i.e. $\mathbf{S}_{\mathrm{K}}$

$$\mathbf{S}_{\mathrm{K}} \;=\; \left[\mathbf{K}_{22} - \mathbf{K}_{21}\,{\mathbf{K}_{11}}^{-1}\,\mathbf{K}_{12}\right] \;. \tag{10.14}$$

If $\mathbf{K}$ is symmetric, is also $\mathbf{S}_{\mathrm{K}}$ symmetric because ${\mathbf{K}_{21}}^{\mathrm{T}} = \mathbf{K}_{12}$. Between the condition numbers (see [2] for the proof; also [3]) holds the relation

$$c(\mathbf{S}_{\mathrm{K}}) \;\le\; c(\mathbf{K}) \;,$$

By separation of n items of $\mathbf{x}$ into some groups $\mathbf{x}_1, \mathbf{x}_2, ...$ as e.g.

- separation of the unknowns bounded by a priori linear relations, say $\mathbf{x}_1$, from the others, say $\mathbf{x}_2$, among these non-linear relationships can occur,
- separation of the (sub)domain boundary unknowns, say $\mathbf{x}_2$, from the unknowns, say $\mathbf{x}_1$, which are subsistent/corresponding of the (sub)domain interior,
- separation of so called master unknowns, say $\mathbf{x}_2$, from the slave ones, say $\mathbf{x}_1$, through any nearer non-specified reason,

one can represent complex problem in a suitably structured form and many efficient algorithms can be applied by using one or more of the following techniques, as e.g.

- substructure and superelement method (one or more levels of substructuring),
- Schwarz alternating methods (with overlapping sudomains), [27],
- multigrid methods (at least two grids of different densities model usually the whole domain, i.e. grids overlap completely), see [17], [18]
- domain decomposition methods (usually non-overlapping sudomains), see e.g. [10], [29], [30] and the Chapters 7 and 8.

10.1.2 Scalar forms of algebraic systems

Scalar formulations of an algebraic system use one of the three *Schwarz error-functions* $F_{\rm er}, F_{\rm rr}, F_{\rm ee}$, [20] and the text below. It is not important if these functions are or are not considered as pre-multiplied by some none-zero real factor. However, by using of factor 1/2 we can easily explain physical meaning of Schwarz error-functions

$$\begin{aligned} F_{\rm er} &= \frac{1}{2}\mathbf{e}^{\rm T}\mathbf{r} = \frac{1}{2}\{\mathbf{x}^* - \mathbf{x}\}^{\rm T}\{\mathbf{f} - \mathbf{K}\mathbf{x}\} \\ &= \frac{1}{2}\mathbf{x}^{\rm T}\mathbf{K}\mathbf{x} - \mathbf{x}^{\rm T}\frac{1}{2}[\mathbf{K}^{\rm T} + \mathbf{K}]\mathbf{x}^* + \frac{1}{2}\mathbf{f}^{\rm T}\mathbf{x}^* \\ &= \frac{1}{2}\mathbf{x}^{\rm T}\mathbf{K}\mathbf{x} - \mathbf{x}^{\rm T}\mathbf{f} + \frac{1}{2}\mathbf{f}^{\rm T}\mathbf{x}^* , \end{aligned} \tag{10.15}$$

$$F_{\rm rr} = \frac{1}{2}\mathbf{r}^{\rm T}\mathbf{r} = \frac{1}{2}\{\mathbf{f} - \mathbf{K}\mathbf{x}\}^{\rm T}\{\mathbf{f} - \mathbf{K}\mathbf{x}\} = \frac{1}{2}\mathbf{x}^{\rm T}[\mathbf{K}^{\rm T}\mathbf{K}]\mathbf{x} - \mathbf{x}^{\rm T}\mathbf{K}^{\rm T}\mathbf{f} + \frac{1}{2}\mathbf{f}^{\rm T}\mathbf{f} , \tag{10.16}$$

$$F_{\rm ee} = \frac{1}{2}\mathbf{e}^{\rm T}\mathbf{e} = \frac{1}{2}\{\mathbf{x}^* - \mathbf{x}\}^{\rm T}\{\mathbf{x}^* - \mathbf{x}\} = \frac{1}{2}\mathbf{x}^{\rm T}\mathbf{x} - \mathbf{x}^{\rm T}\mathbf{K}^{-1}\mathbf{f} + \frac{1}{2}\mathbf{x}^{*\rm T}\mathbf{x}^* . \tag{10.17}$$

Here $\mathbf{x}^*$ stands for the exact solution, $\mathbf{x}$ for its approximation, $\mathbf{e}$ and $\mathbf{r}$ for its absolute error and residuum. We prefer definitions (compare (10.35), (10.36) and (10.18))

$$\mathbf{e} = \mathbf{x}^* - \mathbf{x} , \qquad \mathbf{r} = \mathbf{f} - \mathbf{K}\mathbf{x} = \mathbf{K}\mathbf{x}^* - \mathbf{K}\mathbf{x} = \mathbf{K}\mathbf{e} \tag{10.18}$$

for another legitimate possibilities (convergence is judged in norms $\|\mathbf{e}\|$ and $\|\mathbf{r}\|$)

$$\bar{\mathbf{e}} = -\mathbf{e} , \ \bar{\mathbf{r}} = -\mathbf{r} ; \qquad \mathbf{e} , \ \bar{\mathbf{r}} ; \qquad \bar{\mathbf{e}} , \ \mathbf{r} ,$$

because we intend

- take for the residuum as some right-hand-side vector or its increment and
- put $\mathbf{e}$ and $\mathbf{r}$ together by the relation of the same type as (10.4) and not as $\mathbf{K}\,\mathbf{e} = -\mathbf{r}$

By definitions (10.18) one can express the Schwarz functions (10.15) – (10.17) compactly

$$F_{\rm er} = \frac{1}{2}\mathbf{e}^{\rm T}\mathbf{K}\mathbf{e} , \qquad F_{\rm rr} = \frac{1}{2}\mathbf{e}^{\rm T}\mathbf{K}^{\rm T}\mathbf{K}\mathbf{e} , \qquad F_{\rm ee} = \frac{1}{2}\mathbf{e}^{\rm T}\mathbf{I}\mathbf{e} . \tag{10.19}$$

Hence, for $\mathbf{e} = \mathbf{0}$ are all error-functions equal to zero while for $\mathbf{e} \neq \mathbf{0}$ these functions can equal to any value in range $(-\infty, \infty)$, in dependence on their matrices $\mathbf{K}$ and $\mathbf{K}^{\rm T}\mathbf{K}$. If the matrices $\mathbf{K}$ and $\mathbf{K}^{\rm T}\mathbf{K}$ are *positive definite* then only positive values for error-functions take into consideration

$$\|\mathbf{K}\| > 0 \text{ for } F_{\rm er} \qquad \text{and} \qquad \|\mathbf{K}^{\rm T}\mathbf{K}\| > 0 \text{ for } F_{\rm rr} \tag{10.20}$$

and for Schwarz error-functions hold

$$\begin{aligned} F_{\rm er} &= \frac{1}{2}\|\mathbf{e}\|^2_{\rm K} , & F_{\rm er} &\geq 0 \ \forall\, \mathbf{e} , & F_{\rm er} &= 0 \Leftrightarrow \mathbf{e} = \mathbf{0} , \\ F_{\rm rr} &= \frac{1}{2}\|\mathbf{e}\|^2_{\rm KK} , & F_{\rm rr} &\geq 0 \ \forall\, \mathbf{e} , & F_{\rm rr} &= 0 \Leftrightarrow \mathbf{e} = \mathbf{0} , \\ F_{\rm ee} &= \frac{1}{2}\|\mathbf{e}\|^2_{\rm L2} , & F_{\rm ee} &\geq 0 \ \forall\, \mathbf{e} , & F_{\rm ee} &= 0 \Leftrightarrow \mathbf{e} = \mathbf{0} . \end{aligned} \tag{10.21}$$

The symbols $\|.\|_{\mathrm{L2}}$, $\|.\|_{\mathrm{K}}$ and $\|.\|_{\mathrm{KK}}$ stand successively for the Euclidean- and both energetic norms (induced by energetic products $\mathbf{e}^{\mathrm{T}}\,\mathbf{K}\,\mathbf{e}$ and $\mathbf{e}^{\mathrm{T}}\,\mathbf{K}^{\mathrm{T}}\mathbf{K}\,\mathbf{e}$).
The zero gradient of each error-functions determines location of the stationary point $\mathbf{x}^*$ i.e. solution of the equivalent system with (10.4) and (10.7)

$$\nabla F_{\mathrm{e\,r}} \;=\; \mathbf{K}\mathbf{x} - \mathbf{f} \;=\; \mathbf{0} \qquad \rightarrow \qquad \mathbf{K}\mathbf{x}^* = \mathbf{f}\,, \tag{10.22}$$

$$\nabla F_{\mathrm{r\,r}} \;=\; \mathbf{K}^{\mathrm{T}}\mathbf{K}\mathbf{x} - \mathbf{K}^{\mathrm{T}}\mathbf{f} \;=\; \mathbf{0} \quad \rightarrow \quad \mathbf{K}^{\mathrm{T}}\mathbf{K}\mathbf{x}^* = \mathbf{K}^{\mathrm{T}}\mathbf{f}\,, \tag{10.23}$$

$$\nabla F_{\mathrm{e\,e}} \;=\; \mathbf{x} - \mathbf{K}^{-1}\mathbf{f} \;=\; \mathbf{0} \qquad \rightarrow \qquad \mathbf{x}^* = \mathbf{K}^{-1}\mathbf{f}\,. \tag{10.24}$$

The matrices of second derivatives of error-functions (i.e. their Hessians) are positive definite

$$\frac{\partial^2 F_{\mathrm{e\,r}}}{\partial \mathbf{x}^2} = \mathbf{K} = \frac{\partial^2 F_{\mathrm{e\,r}}}{\partial \mathbf{e}^2}\,, \qquad \frac{\partial^2 F_{\mathrm{r\,r}}}{\partial \mathbf{x}^2} = \mathbf{K}^{\mathrm{T}}\mathbf{K} = \frac{\partial^2 F_{\mathrm{r\,r}}}{\partial \mathbf{e}^2}\,, \qquad \frac{\partial^2 F_{\mathrm{e\,e}}}{\partial \mathbf{x}^2} = \mathbf{I} = \frac{\partial^2 F_{\mathrm{e\,e}}}{\partial \mathbf{e}^2} \tag{10.25}$$

and does not depend on fact whether the independent variables of Schwarz error-functions are $\mathbf{x}$ or $\mathbf{e}$. The Schwarz error-functions are convex, take on in $\mathbf{x}^*$ their minima – and if items of $\mathbf{K}$ and $\mathbf{K}^{\mathrm{T}}\mathbf{K}$ does not depend on $\mathbf{x}$ – quadratic functions of their n variables $\mathbf{x}$. Even though the most important properties of the Schwarz error-functions does not depend on their absolute terms $a(F_{\mathrm{e\,r}})$, $a(F_{\mathrm{r\,r}})$ and $a(F_{\mathrm{e\,e}})$, even these manifest significant physical meaning:

$$\begin{gathered} a(F_{\mathrm{e\,r}}) \;=\; \frac{1}{2}\,\mathbf{f}^{\mathrm{T}}\mathbf{x}^* \;=\; \frac{1}{2}\,\mathbf{x}^{*\mathrm{T}}\mathbf{K}^{\mathrm{T}}\mathbf{x}^* \;=\; \frac{1}{2}\,\|\mathbf{x}^*\|_{\mathrm{K}}^2 \;=\; -\mathrm{min}L_{\mathrm{var}}\,, \\ a(F_{\mathrm{r\,r}}) \;=\; \frac{1}{2}\,\mathbf{f}^{\mathrm{T}}\mathbf{f} \;=\; \frac{1}{2}\,\|\mathbf{f}^*\|_{\mathrm{L2}}^2 \;=\; \frac{1}{2}\,\|\mathbf{x}^*\|_{\mathrm{KK}}^2\,, \qquad a(F_{\mathrm{e\,e}}) \;=\; \frac{1}{2}\,\mathbf{x}^{*\mathrm{T}}\mathbf{x}^* \;=\; \frac{1}{2}\,\|\mathbf{x}^*\|^2{}_{\mathrm{L2}}\,, \end{gathered} \tag{10.26}$$

where L_{var} stands for a discretized variational functional of the total potential energy, so called Lagrangian, and $\mathrm{min}L_{\mathrm{var}}$ for its minimum value

$$L_{\mathrm{var}} \;=\; \frac{1}{2}\mathbf{x}^{\mathrm{T}}\,\mathbf{K}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\,\mathbf{f} \;=\; F_{\mathrm{e\,r}} \;-\; 2\,a(F_{\mathrm{e\,r}}) \;=\; F_{\mathrm{e\,r}} \;-\; \|\mathbf{x}^*\|_{\mathrm{K}}^2\,. \tag{10.27}$$

The following Fig (10.1) shows the relations among Schwarz error-functions formulated in according to (10.19), while Fig (10.2) shows relations their context with Lagrangian L_{var}, if these functions depend in according to on (10.22) – (10.24) and (10.27).

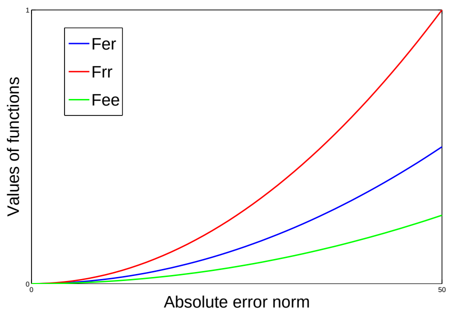


Figure 10.1: Schwarz error-functions versus $\|\mathbf{e}\|_{\mathrm{L2}}$

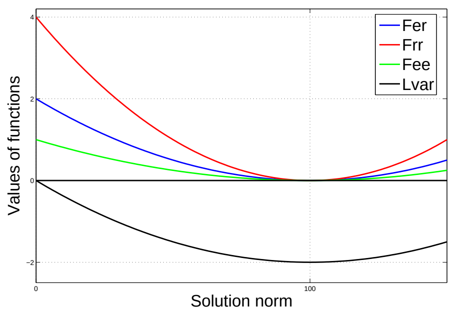


Figure 10.2: Schwarz error-functions and Lagrangian versus $\|\mathbf{x}\|_{\mathrm{L2}}$

10.1.3 Overdetermined and underdetermined linear systems

We know that the solution of the linear systems (10.4) – (10.8) exists and is unique if the number of equations m (it is the number of right-hand-side vector items too) equals to number of unknowns n and the system matrix $\mathbf{K}$ is regular. The algebraic system is called *overdetermined* if $m > n$ and *underdetermined* if $m < n$. For sake simplicity and prevent possible confusion we will use for the rectangular, *not square* system matrix symbol $\mathbf{A}$ instead of $\mathbf{K}$.

The **overdetermined** system (10.28) does not have generally the exact solution $\mathbf{x}$ for any vector $\mathbf{f}$

$$\mathbf{A}\,\mathbf{x} \;=\; \mathbf{f}\,, \quad \mathbf{A}\,(m\times n)\,,\;\; \mathbf{x}\,(n\times 1)\,,\;\; \mathbf{f}\,(m\times 1)\,, \tag{10.28}$$

because $\mathbf{x}$ and $\mathbf{f}$ have different dimensions n and m, i.e. are elements of different vector spaces, say E_n and E_m. In spite of it we anticipate that try for some approximative solution $\tilde{\mathbf{x}}$ is meaningful. According to (10.16), (10.23), and (10.25) one can see that the system arising from nullified gradient of Schwarz function F_{rr}

$$\begin{aligned} \nabla F_{\mathrm{rr}} \;=\; \mathbf{A}^{\mathrm{T}}\mathbf{A}\,\tilde{\mathbf{x}} - \mathbf{A}^{\mathrm{T}}\mathbf{f} \;=\; \mathbf{A}^{\mathrm{T}}\mathbf{A}\,\tilde{\mathbf{x}} - \tilde{\mathbf{f}} \;=\; \mathbf{0} \quad \rightarrow \\ \tilde{\mathbf{x}} \;=\; [\mathbf{A}^{\mathrm{T}}\mathbf{A}]^{-1}\,\mathbf{A}^{\mathrm{T}}\mathbf{f}\,, \qquad \tilde{\mathbf{f}} \;=\; \mathbf{A}^{\mathrm{T}}\mathbf{f}\,,\;\; \tilde{\mathbf{x}}\,(n\times 1)\,,\;\; \tilde{\mathbf{f}}\,(n\times 1) \end{aligned} \tag{10.29}$$

gives $\tilde{\mathbf{x}}$ the exact solution of system (10.29) only if its system matrix (Hessian of F_{rr})

$$\mathbf{G} \;=\; \mathbf{A}^{\mathrm{T}}\mathbf{A}\,,\;\; \mathbf{G}\,(n\times n) \tag{10.30}$$

is regular. Matrix $\mathbf{G}$ is *the metric matrix* of the n-dimensional vector space E_n spanned by columns of $\mathbf{A}$ as basis vectors, so we refer E_n to column space of $\mathbf{A}$. Such base vectors are generally neither orthogonal nor normalized and so span non-cartesian linear vector space.

The best approximation $\mathbf{x}^* \in E_n$ of $\mathbf{x}$ is this one, for which exists the smallest distance $\mathbf{r} \in E_m$ between the vectors $\hat{\mathbf{f}} = \mathbf{A}\,\mathbf{x}^*$ and $\mathbf{f}$, both elements of E_m space, i.e. for which holds the smallest residuum

$$\mathbf{r} \;=\; \mathbf{f} \;-\; \hat{\mathbf{f}} \;=\; \mathbf{f} \;-\; \mathbf{A}\,\mathbf{x}^*\,,\;\; \mathbf{r}\,(m\times 1)\,. \tag{10.31}$$

This property has certainly the orthogonal projection of $\mathbf{f} \in E_m$ into the column space of $\mathbf{A}$, i.e. $\hat{\mathbf{f}} \in E_n$. Since solution $\tilde{\mathbf{x}}$ of problem (10.29) nullifies the following inner product

$$\begin{aligned}\hat{\mathbf{f}}^{\mathrm{T}}\,\mathbf{r} \;&=\; \tilde{\mathbf{x}}^{\mathrm{T}}\,\mathbf{A}^{\mathrm{T}}\,\{\mathbf{f} \;-\; \mathbf{A}\tilde{\mathbf{x}}\} \;=\; \tilde{\mathbf{x}}^{\mathrm{T}}\,\mathbf{A}^{\mathrm{T}}\,\{\mathbf{f} \;-\; \mathbf{A}\,\{[\mathbf{A}^{\mathrm{T}}\mathbf{A}]^{-1}\,\mathbf{A}^{\mathrm{T}}\mathbf{f}\}\} \\ &=\; \tilde{\mathbf{x}}^{\mathrm{T}}\,\mathbf{A}^{\mathrm{T}}\,\mathbf{f} \;-\; \tilde{\mathbf{x}}^{\mathrm{T}}\,\mathbf{A}^{\mathrm{T}}\,\mathbf{f} \;=\; 0\,,\end{aligned} \tag{10.32}$$

it is simultaneously proven that $\tilde{\mathbf{x}} = \mathbf{x}^*$, i.e. $\tilde{\mathbf{x}}$ represents the best approximation, giving for any $\mathbf{f}$ residuum $\mathbf{r}$ orthogonal to the column space of $\mathbf{A}$. Indeed, it is objective expressed by m equations

$$\mathbf{A}^{\mathrm{T}}\,\mathbf{r} \;=\; \mathbf{0}\,. \tag{10.33}$$

Moreover, if the actual right-hand-side $\mathbf{f}$ falls accidently into E_n, the null residuum is obtained according to (10.31) and so $\tilde{\mathbf{x}}$ presents the exact solution of original problem (10.28).

Note, that the projection of right-hand-side vector $\mathbf{f}$ into the column space of $\mathbf{A}$ can be – see (10.29) and (10.30) – written as

$$\hat{\mathbf{f}} \;=\; \mathbf{A}\,\tilde{\mathbf{x}} \;=\; \mathbf{A}\,\{[\mathbf{A}^{\mathrm{T}}\mathbf{A}]^{-1}\,\mathbf{A}^{\mathrm{T}}\mathbf{f}\} \;=\; [\mathbf{A}\,\mathbf{G}^{-1}\,\mathbf{A}^{\mathrm{T}}]\,\mathbf{f} \;=\; \mathbf{P}_{\mathrm{G}}\,\mathbf{f}\,, \tag{10.34}$$

where $\mathbf{P}_{\mathrm{G}}$ represents the *projector* of elements of m-dimensional space E_{m} into n-dimensional space E_{n}. Residuum $\mathbf{r}$ lies in the orthogonal complement of E_{n} to E_{m}, say in $E_{\mathrm{m-n}}$ space, and holds

$$\mathbf{r} \;=\; [\mathbf{I} - \mathbf{P}_{\mathrm{G}}]\,\mathbf{f} \;=\; \mathbf{P}_{\mathrm{R}}\,\mathbf{f}\,, \tag{10.35}$$

where $\mathbf{P}_{\mathrm{R}}$ stands for the complementary projector, with $\mathbf{P}_{\mathrm{G}} + \mathbf{P}_{\mathrm{R}} = \mathbf{I}$.

Exploitation of sparsity in the solution of (10.29) will often yield significant gains. A related approach is to solve the $(m+n)$ order system

$$\begin{bmatrix} \mathbf{I} & \mathbf{A} \\ \mathbf{A}^{\mathrm{T}} & \mathbf{0} \end{bmatrix} \begin{Bmatrix} \mathbf{r} \\ \mathbf{x} \end{Bmatrix} = \begin{Bmatrix} \mathbf{f} \\ \mathbf{0} \end{Bmatrix}. \tag{10.36}$$

The block rows of the last equation are

$$\mathbf{r} \;=\; \mathbf{f} - \mathbf{A}\,\mathbf{x}\,, \qquad \mathbf{A}^{\mathrm{T}}\,\mathbf{r} \;=\; \mathbf{0}\,, \tag{10.37}$$

i.e. the block rows equal to (10.31) and (10.33). By substituting for $\mathbf{r}$ from the first block row into the second one we obtain (10.29). Thus, solving (10.36) is an alternative way of obtaining the solution to $\mathbf{A}^{\mathrm{T}}\mathbf{A}\;\tilde{\mathbf{x}} \;=\; \tilde{\mathbf{f}}$. If dense matrix methods are used then the solution of (10.29) is less costly than the solution of (10.36). But if $\mathbf{A}$ is sparse and sparse methods are used, the solution of equation (10.36) may be less costly than the formation and solution of (10.29). Additionally, formulation (10.36) is better behaved numerically since its solution is less sensitive to small changes in its data than the solution of (10.29) is to small changes in $\mathbf{A}^{\mathrm{T}}\,\mathbf{f}$.

The **underdetermined** system will be added later.

10.1.4 Condition number

From practical point of view, system (10.4) and/or (10.6) could be considered as a disturbed one

$$[\mathbf{K}+\delta\mathbf{K}]\,\{\mathbf{x}+\delta\mathbf{x}\} \;=\; \{\mathbf{f}+\delta\mathbf{f}\} \qquad \text{or} \qquad \tilde{\mathbf{K}}\,\tilde{\mathbf{x}} \;=\; \tilde{\mathbf{f}}\,, \tag{10.38}$$

because certain doubts relative to properties, geometry, boundary conditions, loads and computational aspects always exist.
In this way, the validity of system (10.4), particularly its solution $\mathbf{x}$, depends on only of vague set differences between values of actual- and model- elements of involved matrices. Thus, the assessment of system validity and selection of suitable solution method is a great and ongoing challenge for specialists in the field.

Let us suppose the detailed knowledge of $\delta\mathbf{K}$, $\delta\mathbf{f}$ and at the same time assume that $\operatorname{rank}(\mathbf{K}+\delta\mathbf{K}) = n$, i.e. the disturbed system matrix keeps regularity. Quality of the computed solution $\tilde{\mathbf{x}}$ of the disturbed system (10.38) is measured in practice via residual $\tilde{\mathbf{r}}$ indirectly (imbalance or defect of equilibrium)

$$\mathbf{r} \;=\; \mathbf{f}-\mathbf{K}\,\mathbf{x} \qquad \text{or} \qquad \tilde{\mathbf{r}} \;=\; \tilde{\mathbf{f}}-\tilde{\mathbf{K}}\,\tilde{\mathbf{x}}\,. \tag{10.39}$$

An alternative quality measure of the computed solution – an absolute error $\mathbf{e}$ – is of theoretical nature and it is usually unknown

$$\tilde{\mathbf{e}} \;=\; \tilde{\mathbf{x}}-\mathbf{x} \;=\; \delta\mathbf{x}\,, \tag{10.40}$$

although $\mathbf{e}$ is for us of a primary interest.

A good insight in problem of (10.38) offers a computation using norms. An important fact is that for the present none round-off errors are considered and we will suppose the exactness of computations.

Right-hand-side disturbed

Let us begin with a simple case when only the right-hand-side is disturbed. Then it holds consistently with (10.38)

$$\mathbf{K}\,\{\mathbf{x}+\delta\mathbf{x}\} \;=\; \{\mathbf{f}+\delta\mathbf{f}\} \quad \rightarrow \quad \mathbf{K}\,\{\delta\mathbf{x}\} \;=\; \{\delta\mathbf{f}\} \quad \text{and} \quad \mathbf{K}\,\mathbf{x} \;=\; \mathbf{f}\,. \tag{10.41}$$

By normalizing and using the Cauchy-Schwarz unequality

$$\|\delta\mathbf{x}\| = \|\mathbf{K}^{-1}\,\delta\mathbf{f}\| \;\leq\; \|\mathbf{K}^{-1}\|\;\|\delta\mathbf{f}\| \quad \text{and} \quad \|\mathbf{f}\| \leq \|\mathbf{K}\|\;\|\mathbf{x}\|$$

and multiplying of both last relations we obtain the following unequality

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \;\leq\; \|\mathbf{K}^{-1}\|\;\|\mathbf{K}\|\frac{\|\delta\mathbf{f}\|}{\|\mathbf{f}\|} \;=\; c(\mathbf{K})\,\frac{\|\delta\mathbf{f}\|}{\|\mathbf{f}\|} \tag{10.42}$$

in which the relative disturbance norm of $\mathbf{x}$ is estimated *from above* as a product of the *condition number* $c(\mathbf{K})$ of the system matrix and the relative disturbance norm of $\mathbf{f}$.

Now some questions come up: Is (10.42) only a very rough estimate or can it hold with the sign equality? Has a big condition number of $\mathbf{K}$ so desolating influence on the relative disturbance norm of $\mathbf{x}$ for any on the relative disturbance norm of $\mathbf{f}$? Answers on these questions offers the following paragraph.

In a simple case, if the matrix $\mathbf{K}$ of system (10.38) is positive definite, we can use positive eigenvalues $\lambda_i, i = 1, ..., n$ and corresponding normalized eigenvectors $\mathbf{v}_i, i = 1, ..., n$ coming out from the standard eigenvalue problem solution

$$\mathbf{K}\,\mathbf{v}_i = \lambda_i\,\mathbf{v}_i \quad \rightarrow \quad \mathbf{K}\left\{\frac{1}{\lambda_i}\mathbf{v}_i\right\} = \mathbf{v}_i\;, \qquad \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n\,. \tag{10.43}$$

Suppose firstly that the actual right-hand-side vector of (10.41) equals to the eigenvector $\mathbf{v}_1$ corresponding to the smallest eigenvalue λ_1 of problem (10.43) and denote this actual vector as $\mathbf{f}_1$, so that holds $\mathbf{f}_1 = \mathbf{v}_1$. Suppose more that disturbance of $\mathbf{f}_1$ is given by a simple combination of other remaining eigenvectors $\delta\mathbf{f}_1 = \varepsilon(\mathbf{v}_2 + \mathbf{v}_3 + ... + \mathbf{v}_n)$ where $\varepsilon \in \Re$, so that for disturbed right-hand-side vector of (10.41) holds $\tilde{\mathbf{f}}_1 = \{\mathbf{f}_1 + \delta\mathbf{f}_1\}$. Because of linearity, we have in accordance with (10.43) the disturbed solution

$$\begin{aligned}\tilde{\mathbf{x}}_1 &= \frac{1}{\lambda_1}\mathbf{v}_1 + \varepsilon\left(\frac{1}{\lambda_2}\mathbf{v}_2 + \frac{1}{\lambda_3}\mathbf{v}_3 + ... + \frac{1}{\lambda_n}\mathbf{v}_n\right) \\ &= \mathbf{x}_1 + \frac{\varepsilon}{\lambda_1}\left(\frac{\lambda_1}{\lambda_2}\mathbf{v}_2 + ...\frac{\lambda_1}{\lambda_3}\mathbf{v}_3 + ... + \frac{\lambda_1}{\lambda_n}\mathbf{v}_n\right) \;=\; \mathbf{x}_1 + \delta\mathbf{x}_1.\end{aligned} \tag{10.44}$$

As eigenvectors are normalized, the norms ratio for disturbed and actual right-hand-side vectors comes to

$$\frac{\|\delta\mathbf{f}_1\|}{\|\mathbf{f}_1\|} \;=\; \varepsilon\frac{\|\mathbf{v}_2\| + \|\mathbf{v}_3\| + ... + \|\mathbf{v}_n\|}{\|\mathbf{v}_1\|} \;=\; (n-1)\,\varepsilon \tag{10.45}$$

while the norms ratio for disturbed and non-disturbed solution vectors equals

$$\frac{\|\delta\mathbf{x}_1\|}{\|\mathbf{x}_1\|} \;=\; \frac{\varepsilon}{\lambda_1}\frac{\left(\frac{\lambda_1}{\lambda_2}\|\mathbf{v}_2\| + \frac{\lambda_1}{\lambda_3}\|\mathbf{v}_3\| + ... + \frac{\lambda_1}{\lambda_n}\|\mathbf{v}_n\|\right)}{\frac{1}{\lambda_1}\|\mathbf{v}_1\|} \;=\; \left(\frac{\lambda_1}{\lambda_2} + ... + \frac{\lambda_1}{\lambda_n}\right)\varepsilon\,. \tag{10.46}$$

Suppose vice versa that the actual right-hand-side vector equals $\mathbf{f}_n = \mathbf{v}_n$ and that for its disturbance holds $\delta\mathbf{f}_n = \varepsilon(\mathbf{v}_1 + \mathbf{v}_2 + ... + \mathbf{v}_{n-1})$. Then we obtain, likewise as previously, the disturbed right-hand-side vector of system (10.41) in the form $\tilde{\mathbf{f}}_n = \{\delta\mathbf{f}_n + \mathbf{f}_n\}$. The disturbed solution in this case is, analogously to (10.43),

$$\tilde{\mathbf{x}}_n \;=\; \frac{\varepsilon}{\lambda_n}\left(\frac{\lambda_n}{\lambda_1}\mathbf{v}_1 + ...\frac{\lambda_n}{\lambda_2}\mathbf{v}_2 + ... + \frac{\lambda_n}{\lambda_{n-1}}\mathbf{v}_{n-1}\right) \;+\; \frac{1}{\lambda_n}\mathbf{v}_n \;=\; \delta\mathbf{x}_n + \mathbf{x}_n \tag{10.47}$$

and the ratio of norms of disturbed and actual right-hand-side vectors and the ratio of norms of disturbed and non-disturbed solution vectors are

$$\frac{\|\delta\mathbf{f}_n\|}{\|\mathbf{f}_n\|} \;=\; (n-1)\,\varepsilon\;, \qquad \frac{\|\delta\mathbf{x}_n\|}{\|\mathbf{x}_n\|} \;=\; \left(\frac{\lambda_n}{\lambda_1} + ... + \frac{\lambda_n}{\lambda_{n-1}}\right)\varepsilon\,. \tag{10.48}$$

Answers on two questions come out from comparing of equations (10.45), (10.46) and (10.48) with (10.42). We will comment only both limiting cases here.

If $\mathbf{f}_1 = \mathbf{v}_1$ is affected only by a multiple of eigenvector $\mathbf{v}_n$ responding to the greatest eigenvalue λ_n, i.e. $\delta\mathbf{f}_1 = \varepsilon\mathbf{v}_n$, it will be its normalized relative response $(\|\delta\mathbf{x}_1\|/\|\mathbf{x}_1\|)$ amplified minimally, because in accordance with (10.45) a (10.46) holds

$$\frac{\|\delta\mathbf{x}_1\|}{\|\mathbf{x}_1\|} = \varepsilon\,\frac{\lambda_1}{\lambda_n}\,, \quad \frac{\|\delta\mathbf{f}_1\|}{\|\mathbf{f}_1\|} = \varepsilon \quad \rightarrow \quad \frac{\|\delta\mathbf{x}_1\|}{\|\mathbf{x}_1\|} = \frac{\lambda_1}{\lambda_n}\,\frac{\|\delta\mathbf{f}_1\|}{\|\mathbf{f}_1\|} \quad \rightarrow \quad {}^{f_1}c(\mathbf{K}) = \frac{\lambda_1}{\lambda_n} << 1\,.$$

And vice versa, if $\mathbf{f}_n = \mathbf{v}_n$ is affected only by an multiple of eigenvector $\mathbf{v}_1$ responding to the smallest eigenvalue λ_1, i.e. $\delta\mathbf{f}_n = \varepsilon\mathbf{v}_1$, it will be its normalized relative response $(\|\delta\mathbf{x}_n\|/\|\mathbf{x}_n\|)$ amplified maximally, because in accordance with (10.48) we have

$$\frac{\|\delta\mathbf{x}_n\|}{\|\mathbf{x}_n\|} = \varepsilon\,\frac{\lambda_n}{\lambda_1}\,, \quad \frac{\|\delta\mathbf{f}_n\|}{\|\mathbf{f}_n\|} = \varepsilon \quad \rightarrow \quad \frac{\|\delta\mathbf{x}_n\|}{\|\mathbf{x}_n\|} = \frac{\lambda_n}{\lambda_1}\,\frac{\|\delta\mathbf{f}_n\|}{\|\mathbf{f}_n\|} \quad \rightarrow \quad {}^{f_n}c(\mathbf{K}) = \frac{\lambda_n}{\lambda_1} >> 1\,.$$

Thus the unequality (10.42) represents at most pessimistic assessment only and the real bounds of $(\|\delta\mathbf{x}\|/\|\mathbf{x}\|)$ – the *both attainable* – are

$$\frac{1}{c(\mathbf{K})}\frac{\|\delta\mathbf{f}\|}{\|\mathbf{f}\|} \le \frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le c(\mathbf{K})\frac{\|\delta\mathbf{f}\|}{\|\mathbf{f}\|}\,. \tag{10.49}$$

End of paragraph Right-hand-side disturbed □

Matrix and right-hand-side disturbed

Considerations complicate if also items of $\mathbf{K}$ matrix can be disturbed. Instead of (10.42) for the condition number holds, see [26], [24], [13].

$$c^*(\mathbf{K}) = \frac{c(\mathbf{K})}{1 - c(\mathbf{K})\dfrac{\|\delta\mathbf{K}\|}{\|\mathbf{K}\|}} = \frac{c(\mathbf{K})}{1 - \|\mathbf{K}\|^{-1}\,\|\delta\mathbf{K}\|} \tag{10.50}$$

and the real bounds of $\dfrac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|}$ – the *both attainable* – are

$$\frac{1}{c^*(\mathbf{K})}\left(\frac{\|\delta\mathbf{K}\|}{\|\mathbf{K}\|} + \frac{\|\delta\mathbf{f}\|}{\|\mathbf{f}\|}\right) \le \frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le c^*(\mathbf{K})\left(\frac{\|\delta\mathbf{K}\|}{\|\mathbf{K}\|} + \frac{\|\delta\mathbf{f}\|}{\|\mathbf{f}\|}\right)\,. \tag{10.51}$$

An explanation (needs 2–3 pages) will be added later.

End of paragraph Matrix and right-hand-side disturbed □

10.1.5 Solution of linear algebraic systems

There are many methods for solving of algebraic systems (10.4). Usually, we refer to direct-, iterate-, semiiterate-, combined- and statistical methods.

Direct methods

An algorithm of direct method gives solution $\mathbf{x}$ of a *linear* equation system in a number of operations that is determined (or estimated from above) *a priori* by the size (or by the size and the pattern of system matrix, if it is sparse) of the system.
In exact arithmetic, a direct method yields the true solution to the system. See [6].
Merits as well as disadvantages of direct methods are as follows:

Merits of direct methods

- Solution $\mathbf{x}$ of a system $\mathbf{Kx} = \mathbf{f}$ is obtained in an *a priori* given or in an *a priori* from above estimated number of arithmetic operations.
- Such as this estimate is known then it holds for any right-hand-side vector $\mathbf{f}$ of $\mathbf{F}$.
- Further, if any solution, say $\mathbf{x}_i$ – corresponding to given right-hand-side vector $\mathbf{f}_i$ – is found, then each other solution corresponding to any $k - 1$ remaining right-hand-side vectors can be reached considerably more effectively then it was done for the first time.
- Realizing the fore-mentioned property in a great extent, one can construct the inverse matrix $\mathbf{K}^{-1}$ when for the set of n right-hand-side vectors, collected in $\mathbf{F} = \mathbf{I}$ matrix, see (10.4, finds n corresponding solutions and sets those in columns of matrix $\mathbf{X} = \mathbf{K}^{-1}$.
- Moreover, if the algorithm, e.g. elimination or factorization, of the direct method is carried out only partially (i.e., for example only for some set of unknowns or for some "vectors"), it can produce so-called Schur complement $\mathbf{S}_{\mathrm{K}}$ of $\mathbf{K}$ matrix (ie. super-element or sub-structure matrix) and/or a certain type of preconditioning-matrix $\mathbf{P}_{\mathrm{K}}$ (exploitable e.g. in the group of iterative methods).
- Moreover, parallelization techniques can be also implemented, [7], [8].
- None starting approximations of $\mathbf{x}^0$ as well as none terminating criteria are needed.

Disadvantages of direct methods

- If $\mathbf{K}$ is indefinite then the **pivoting** is needed. It means that *in the course of solution* may be either order of equations or order of unknowns or both changed. Hence pivoting modifies either $\mathbf{K}$, $\mathbf{f}$ or $\mathbf{K}$, $\mathbf{x}$ or all $\mathbf{K}$, $\mathbf{f}$, $\mathbf{x}$. These transformations disturb step by step the originally existing arrangement of system, ie. the pattern of non-zero items of involved matrices.
- If $\mathbf{K}$ is sparse then during solution of an algebraic system so called **fill-in effect** sets in and spreads out with growing intensity as the solution proceeds. Fill-in consists in fact that originally zero items of $\mathbf{K}$ are replaced by non-zeros. In particular, distinctly sparse systems – and it is just a case of FE analysis – incline to this disease.
- For this reason, even relative small systems, say of order($n > 10^3$), strongly need an suitable **optimizer** for some restriction of storage demands, arithmetic operations and fill-in manifestations.
- If pivoting is needed the possible positive effect of an optimizer – which is always applied before start of an equation solver – can be wasted.

Individual pivoting steps can be performed as a multiplication of actual system by elementary permutation matrices $\hat{\mathbf{P}}^{(i,j)}$ (multiplication from the left) or/and $\hat{\mathbf{Q}}^{(k,l)}$ (multiplication from the right).

Effects of individual optimizers on an original sparse equation system can be presented as multiplication of this system by a permutation matrix $\hat{\mathbf{P}}$ from the left and as

multiplication of vector $\mathbf{x}$ by a permutation matrix $\hat{\mathbf{Q}}$ from the left. Of cause, problems of optimizers, data structures, data management and overheads are sophisticated and intricate, but we have only mentioned their resulting "trace" in a system to be solved.
There are various groups of **optimizers** today (the more apposite term for an optimizer should be a reducer, but we will use next only the first one). We note in advance that this art of optimization in essence means 'only' improvement of topological properties of the computational model (or its mesh). This optimization makes solution of (10.4) more effective or sometimes at all accessible but it should not influence the quality of computational models as well as their responses (with the exception of rounding errors).
Optimizers can be briefly divided according to target of intention into the band-, skyline, front- and fill-in optimizer groups. There are some differences between the methods handling with special systems (e.g. symmetric, antisymmetric) and the methods handling with general non-symmetric systems. In the next text we will focus our attention to symmetric systems.

The band optimizer aims at attainment of such a pattern of $\mathbf{K}$ matrix which would place its non-zero items as nearly as possible about/along the main diagonal of $\mathbf{K}$. The maximum of distances, found successively between non-zero- and its diagonal items, is called the bandwidth (for non-symmetric systems is the such defined value called as the half bandwidth). The $(n \times n)$ matrix $\mathbf{K}$ is banded with the bandwidth b, if holds

$$b = \max(b_i) \,, \quad b_i = j - i + 1 \,, \quad \text{and } \forall i \;\; K(i, b_i) \neq 0 \;\text{ and }\; K(i, j) = 0 \;\; \forall j > i + b_i \,.$$

In context of the finite element method this process depends only on the numbering of nodes, the ordering of elements is irrelevant. [4], [14], [23].

The skyline optimizer searches for such a pattern of $\mathbf{K}$ whose *outline* bounded by non-zero matrix items – and by main diagonal matrix items if one deals with symmetric problems – demarcates the smallest possible area. This area is called a *profile* and therefore the skyline optimizer is sometimes known as the *profile optimizer*. Hence, when renumbering process to the matrix $\mathbf{K}$ is completed, it is noted for a *profile-pattern*.
The outline is a borderline demarcated by items, positions of which manifests the maximum distance from the main diagonal of $\mathbf{K}$, if it is 'column-wise measured'. In context of the finite element method this process depends also only on the numbering of nodes, the ordering of elements is irrelevant, [19], [11], [9]. Of course, the optimizer target function here is different as it was for the bandwidth optimizer.

The front optimizer assumes connection to a special solver of algebraic systems so called frontal solver. The frontal solver (see more in the next sections) does not use any assembled equation system of an explicit type (10.4) but it assembles - successively and step by step - only quasi-global system which represents - in a certain sense - a relative small part of the global system (10.4). After each assemble step follows an elimination step in which these unknowns, the rows and columns of which have been yet completed/(assembled) in the preceding assembling step, are eliminated *as soon as possible.*
The front optimizer aims at attainment of such ordering of finite elements so that for the quasi-global system (i.e. for the front) as small as possible extent/dimension of a triangular- or square matrix array (according to system manifest symmetry or not). In context of the finite element method depends this process - contrary to the band optimizer - only on the numbering of elements, the ordering of nodes is irrelevant, [28]. In practice, especially if nodes as well as elements are ordered in accordance, one can also use band- and/or profile optimizer successfully, [14], [15], [16], [19].

The fill-in optimizers present the newest and also the most powerful tool of optimization of meshes, see the Pařík's Chapter 11 and [31].

Iterative methods

An algorithm produces a sequence of approximations to the solution of a (linear) system of equations; the length of the sequence is not given a priori. If is the algorithm convergent, the longer one iterates, the closer one is able to get to the true solution. There are reasons for it. For one thing, at that in exact arithmetic, successive iterations of any convergent algorithm approaches generally the exact solution in a limit sense - i.e. when number of iterations approaches infinity - and for another, their algorithms require stoping criteria suitably set for 'reasonable' termination of computation.

Merits as well as disadvantages are as follows:

Merits of iterative methods

- Simplicity of algorithms.
- Pattern of non-zero items of $\mathbf{K}$ matrix holds during the whole process, so the fill-in phenomenon does not occur.
- Storage requirements remain constant during the solution process.
- Relative easy implementation of parallelization technology is possible
- For sparse matrices, common storage schemes avoid storing zero elements can be used; as a result they involve indices, stored as integer data, that indicate where the stored elements fit into the global matrix. Hence the simple band-optimizer is sufficient so that no very sophisticated renumbering of nodes or elements with the regard to fill-in effect reduction is needed, see the chapter of colleague Pařík.
- Assembling of resulting algebraic system is not always needed; an iterative solver can work with individual element matrices and vectors or with individual equations; again, an easy implementation of parallelization technology is possible.

Disadvantages of iterative methods

- A suitable preconditioning is practical necessary (which can bee a hard task).
- None estimate of number of arithmetic operations holds for computation of $\mathbf{x}_i$ corresponding to $\mathbf{f}_i$.
- Even if such a number were at hand for $\mathbf{f}_i$, another right-hand-side vector $\mathbf{f}_j$ would require widely different number of operations; departure from the rule present right-hand-sides vectors which are each other sufficiently proximal so that solution corresponding to one right-hand-side vector serve as a good approximation for the other (this feature can be attractive in non-linear analyzes).
- Some initial guess $\mathbf{x}_1$, i.e. starting approximation of solution, is needed. Unfortunately, the computational costs can hard depend on the quality of this initial guess.

- Stopping criterion or criteria are needed; since an iterative method computes successive approximations to the solution, a practical test is needed to determine when to stop the iteration. Ideally this test would measure the distance of the last iterate to the true solution, but this is not possible. Instead, various other metrics are used, typically involving the residual.
- Neither matrix inversion $\mathbf{K}^{-1}$ or Schur complement $\mathbf{S}_\mathrm{K}$ of $\mathbf{K}$ matrix or any multiplicative splits of $\mathbf{K}$ can be constructed.

Among the iterative methods belongs, e.g. Richardson-, Jacobi-, Relaxation- and Gauss-Seidel method. **Semi-iterative methods**
Semi-iterative methods are of two-faces. They have a nature of direct ones if only none rounding and cut off errors arise and propagate during the numerical process. But, because it is not true, their iterative nature allow us via next iterations a successive reduction of errors. The exact solution is reached - similarly to the iterative methods - in a limit sense for infinite number of iterations and thus their algorithms need also stoping criteria and initial guesses, i.e. starting iterations. A simple and robust estimate of number of arithmetic operations is generally not possible. An explanation (needs 2–3 pages) will be added later. Among the semi-iterative methods belongs, e.g. the conjugated gradient- or the preconditioned gradient method.
Combined methods Combined methods are intricate and sophisticated. Name especially the hierarchical substructure method, domain decomposition methods and multigrid methods. Combined methods have a nature of direct or iterative procedures or – and it is more usual case – they combine the advantageous properties both of fundamental approaches. An explanation (needs 3–4 pages) will be added later.

10.2 The Frontal Solution Technique

The frontal solution technique, in its best known form, has originally been devised by Bruce M. Irons and developed at the University College of Swansea by his coworkers who pioneered the isoparametric approach to finite element method [22], [1]. Although more general applications are possible, the frontal method can be considered as a particular technique to assemble of FE-matrices and FE-vectors into global equation system and to solve this system by means of Gaussian elimination and backsubstitution; it is designed to minimize the core storage requirements, the arithmetic operations and the use of peripheral equipment. Its efficiency and advantages over other methods such as the band solution will be presented in next paragraphs.

The main idea of the frontal solution is to perform assembly, implementation of boundary conditions and elimination of unknowns at the same time: as soon as the coefficients of an equation are completely summed, the corresponding unknown parameter can be eliminated. The global matrices and global right-hand-side vectors are never formed as such, they are immediately sent to the back-up storage under reduced form.

10.3 The basic, simplest version of frontal technique

This version is stemming from following assumptions about the algebraic system $\mathbf{Kx} = \mathbf{f}$ of n equations

- the local, element matrices $\mathbf{K}_e, e = 1, 2, ...,$ are symmetric and full
- the global, sparse structure matrix $\mathbf{K}$ is positive semi-definite (i.e. none boundary conditions have to be respected)
- Gaussian elimination (factorization) without pivoting followed by backsubstitution seems to be the most efficient solution process: $\mathbf{GKx} = \mathbf{Ux} = \mathbf{Gp} \quad \rightarrow \quad \mathbf{K} = \mathbf{G}^{-1}\mathbf{U} = \mathbf{LU}$. Here $\mathbf{U}$ stands for the upper triangular matrix while $\mathbf{G} = \mathbf{G}_{n-1}\mathbf{G}_{n-2}...\mathbf{G}_2\mathbf{G}_1$ represents the lower triangular matrix of all needed elimination steps. Inverse of $\mathbf{G}$, named $\mathbf{L}$, has also lower triangular form. Matrix product $\mathbf{LU}$ express one of infinitely numerous multiplicative splits of matrix $\mathbf{K}$, often called simple its factorization.

The significant features of the system matrix, as its 'outline' (e.g. banded or skyline structure) and symmetry are retained during elimination process. Indeed, for example, it holds after the first elimination step (if $n = 4$ and $\mathbf{K}$ is generally full)

$$\mathbf{G}_1\,\mathbf{K} \;=\; \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{K_{12}}{K_{11}} & 1 & 0 & 0 \\ -\frac{K_{13}}{K_{11}} & 0 & 1 & 0 \\ -\frac{K_{14}}{K_{11}} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} \\ K_{12} & K_{22} & K_{23} & K_{24} \\ K_{13} & K_{23} & K_{33} & K_{34} \\ K_{14} & K_{24} & K_{34} & K_{44} \end{bmatrix} = \begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} \\ 0 & \tilde{K}_{22} & \tilde{K}_{23} & \tilde{K}_{24} \\ 0 & \tilde{K}_{23} & \tilde{K}_{33} & \tilde{K}_{34} \\ 0 & \tilde{K}_{24} & \tilde{K}_{34} & \tilde{K}_{44} \end{bmatrix},$$

where the (3×3) submatrix $\tilde{\mathbf{K}}$ formed by items signed by tilda

$$\tilde{\mathbf{K}} \;=\; \begin{bmatrix} \tilde{K}_{22} & \tilde{K}_{23} & \tilde{K}_{24} \\ \tilde{K}_{32} & \tilde{K}_{33} & \tilde{K}_{34} \\ \tilde{K}_{42} & \tilde{K}_{43} & \tilde{K}_{44} \end{bmatrix} = \begin{bmatrix} K_{22} - \dfrac{K_{12}}{K_{11}}K_{12} & K_{23} - \dfrac{K_{12}}{K_{11}}K_{13} & K_{24} - \dfrac{K_{12}}{K_{11}}K_{14} \\ K_{32} - \dfrac{K_{13}}{K_{11}}K_{12} & K_{33} - \dfrac{K_{13}}{K_{11}}K_{13} & K_{34} - \dfrac{K_{13}}{K_{11}}K_{14} \\ K_{42} - \dfrac{K_{14}}{K_{11}}K_{12} & K_{43} - \dfrac{K_{14}}{K_{11}}K_{13} & K_{44} - \dfrac{K_{14}}{K_{11}}K_{14} \end{bmatrix}$$

remains *symmetric*, if only $\mathbf{K}$ matrix had this property. The last equation also shows that $\tilde{\mathbf{K}}$ is the sum of two matrices: the former one is a restriction of $\mathbf{K}$ (or $\mathbf{GK}$) while the latter matrix is always of *rank only one*, because its every row is only some multiple of an another one.
Further, it is well known that the addition of any row-multiple of $\mathbf{K}$ to any its row does not change the det($\mathbf{K}$). Hence the product of all n eigenvalues of $\mathbf{K}$ – one of its invariants, say $\prod(\lambda_i) = (\lambda_1 \cdot \lambda_2 \cdot \ldots \cdot \lambda_n)$, preserves its original value in the course of elimination process.
The sum of all diagonal items or the sum of all n eigenvalues, say $(\lambda_1 + \lambda_2 + \ldots + \lambda_n)$ represents an another of matrix invariants – trace($\mathbf{K}$). The last equation shows that the trace($\mathbf{K}$) undergoes in the course of elimination certain changes. For this case holds

$$\text{trace}(\mathbf{K}) - \text{trace}(\mathbf{G_1K}) \;=\; ((K_{12})^2 + (K_{13})^2 + (K_{14})^2)/K_{11}$$

Hence, in the course of elimination process are condition numbers of matrices $\mathbf{K}$, $\mathbf{G_1K}$, $\mathbf{G_2G_1K}$, ... , $\mathbf{GK}$ generally not equal.

What is present in core at any given time during the application of the frontal solution is the upper triangular part of a square matrix and its relevant right-hand-side vector containing the equations whose summation is in progress at that particular time. These equations, corresponding nodes and their degrees of freedom are called the **front**. The number of these equations – i.e. dimension of the upper triangular part of a square matrix – is the **frontwidth**; it can be changing all the time; the program capacity is generally conditioned by the **maximum frontwidth**. The variables, nodes and degrees of freedom belonging to the front are called **active**; those which have not been considered yet are **inactive**; those which have gone through the front and have been reduced or eliminated are **deactivated**.

In the frontal technique the FE elements are considered each in turn. Whenever a new FE element is called in, its stiffness coefficients are computed and summed either into existing equations if the nodes were active already or in new equations which have to find place in the front if the nodes are activated for the first time. If some nodes are appearing for the last time, the corresponding equations can be reduced and stored away on back-up files; so doing they free space in the front; these nodes are deactivated. The next element can then be considered.

The sequence of figures 10.3a to 10.3g shows that the procedure name can be justified on military grounds [12]. During the assemblage and elimination phase, the front moves forward like an advancing army. The front is the battleground separating conquered from unconquered territory; it comprises all the border nodes before attacking a new element and the border nodes plus the element nodes while the element is being conquered, i.e. added to the global stiffness; after new territory has been conquered, the captured nodes are sent to the rear, on the back-up storage.

At the end of elimination phase is all territory – i.e. the domain structured by its computational mesh – conquered. But during the following back substitution phase will be step by step all this territory lost.

10.3.1 The Prefront part of the program

Purpose of Prefront

To keep track of the multiple appearances of degrees of freedom, to predict their last appearance and to decide where in the front the corresponding equations will be assembled, an elaborate house keeping procedure is required. Partly because some of it is necessary in advance and partly to avoid the uneconomic consequences of a failure occuring after much time has already been spent in the frontal assemlage and reduction, the housekeeping is prepared in the "Prefront" part of the program. Prefront can be considered as a separate module and will indeed be managed as a subroutine. It essentially runs through a mock elimination process paving the way for the real one which is going to take place slightly later. The example of Fig. (10.3) will be used to explain Prefront. Note that the elements are numbered in a stupid order and that every node is supposed to possess only one degree of freedom (one may consider the elements for analysis of scalar fields, e.g. temperature field or potential problems). The purpose of these artificial characteristics is to demonstrate different features of the program while keeping the explanations as short as possible. The Figs (10.3a) to (10.3g) are accompanied by arrays whose meaning and content will be described hereafter.

- number of nodes for each element is the same (here `NNODE=3` was accepted),

- number of degrees of freedom per node `NDOFN` is the same for the whole FE-model,
- moreover, in the following paragraph we used `NDOFN=1` (as corresponds to scalar field analysis),
- only the simplest case of boundary condition was considered,
- only one right-hand-side vector was considered for the simultaneous processing,
- none strategy concerning of the external storage handling was studied.

Arrays used and created in Prefront

Most values, variables and arrays used in the program are defined in [12] but, for the reader's facility, the most important ones used in Prefront are already introduced below. Their value in case of the example is given immediately after the definition whenever this value should be known before entering Prefront.

Prefront input parameters	Text	
`MXEFW`	Maximum 'allowed' frontwidth	10^3
`MAXLI`	Maximum of the in core-space reserved by program; this space can be expressed by the length of an equivalent integer array.	10^6
`MXDEQ`	Maximum 'expected' requirement for needed external storage capacity	10^9
Prefront outputs	Text	
`MAXFW`	Maximum frontwidth; found for processed model it must hold `MAXFW` $\leq$ `MXEFW`	5
`LI`	In-core storage capacity requirement it must hold `LI` $\leq$ `MAXLI`	100
`LIDEQ`	External storage capacity requirement it must hold `LIDEQ` $\leq$ `MXDEQ`	5×10^2

Value	Text	Taken in the example
`NELEM`	Number of elements	3
`NNOD`	Number of nodes	5
`LSOL`	Length of solution vector i.e. the total number of all unknown parameters	5
`NDOFN`	Number degrees of freedom per a node if only the same value of `DOF` holds for all nodes; if it is not true, see the `DOF`-array in the table 3.	1

Variable	Text	Taken in the example
`MFRON`	The actual frontwidth	`MFRON` $\leq$ `MAXFW`
`NNODE`	Number of nodes per element generally, each element have a different `NNODE`;	3
`NDOFE`	Number of `DOF`s per element generally, each element have a different `NDOFE`;	1

Array	Text	Taken in the example

`INET(IELEM, INODE),`
`IELEM = 1,.., NELEM`
`INODE = 1,.., NNODE`
	List of node-numbers
	for all finite-elements
	and all nodes of each finite-element
		4, 1, 2
		3, 2, 5
		2, 3, 4

`IELAS(INOD),`
`INOD = 1,.., NNOD`
	Number of element containing the last
	appearance of `INOD`-th node
		1, 3, 3, 3, 2

	The last appearance can be recorded with the minus sign into `INET`-array too:	

`INET(IELEM, INODE),`
`IELEM = 1,.., NELEM`
`INODE = 1,.., NNODE`
	List of node-numbers
	for all finite-elements
	and all nodes of each finite-element
		4, -1, 2
		3, 2, -5
		-2, -3, -4

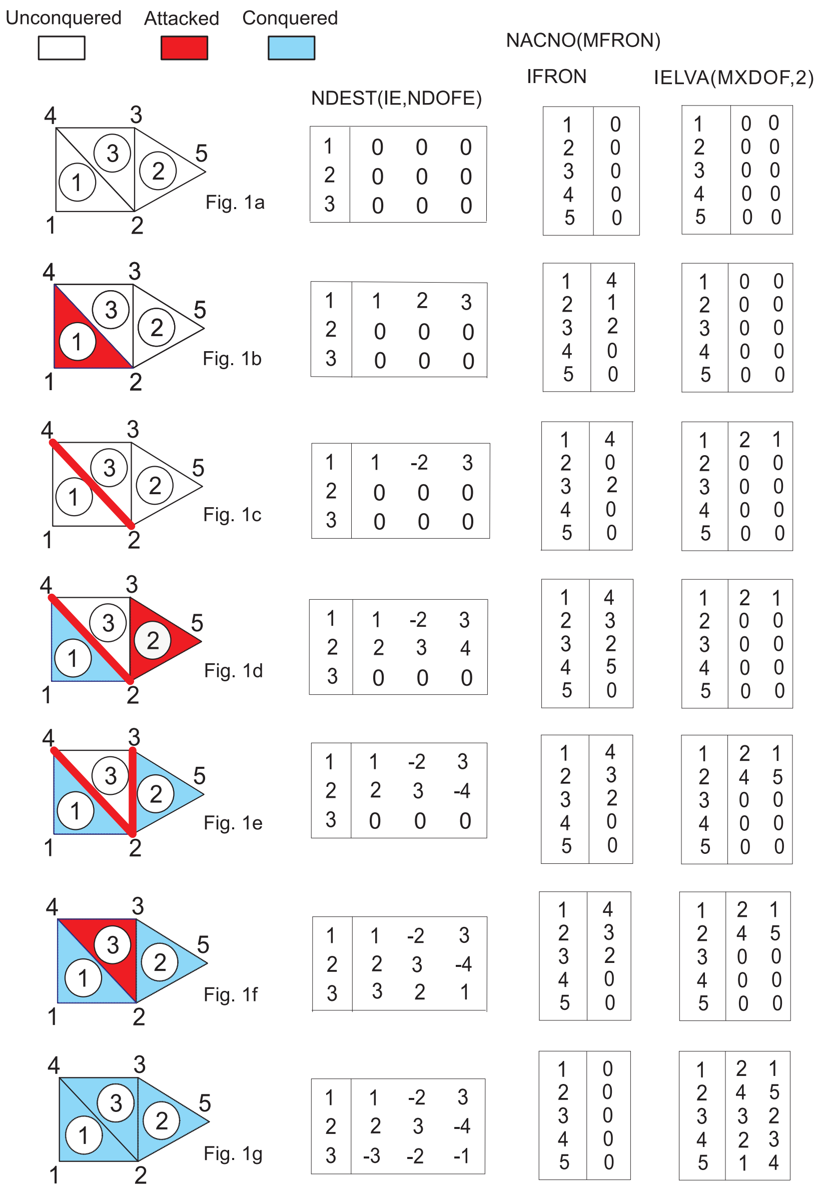
`NDEST(IDOFE, IELEM),`
`IDOFE = 1,.., NDOFE`
`IELEM = 1,.., NELEM`
	List of destination-numbers in the front
	for each of element-degrees-of-freedom
	and for each of finite elements;
	a last appearance can be recorded
	with the minus sign.
		see
		`NDEST` array
		evolution
		in Fig. (10.3)

`NACNO(MFRON)`	Number of active nodes in each of the `MFRON` equations of the front; a zero indicates an available space; the content of the frontwidth and consequently the `NACNO`-vector is changing as the front is moving across the structure.	see `NACNO` array evolution in Fig. (10.3)

`IELVA(IXFW, IC),`
`IXFW = 1,.., MAXFW`
`IC = 1,.., NC`
`IC = 1` contains
 	
`IC = 2` contains
 	
... eventually
`IC = 3` contains
`IC = 4` contains
	Numbers relative to the eliminated variables
	for each of element-degrees-of-freedom
	column numbers of `IELVA` matrix.
	the front number of the equation
	used for elimination
	the global number of the node
	to which the variable belongs
	
	the `DOF` number within the node
	the imposed boundary condition number.
		see
		`IELVA` array
		evolution
		in Fig. (10.3)

Tabulation of the last appearance for each node

This is realized by means of a node-DO-loop nested within an element-DO-loop. The considered element number is stored in the components of `IELAS` corresponding to the nodes which belong to this element. This has the effect of overwriting any previous content of these locations in `IELAS`; since the elements are considered in increasing order, the last recorded value will represent the last appearance of each node. The vector `IELAS` has an ephemeral existence. It will be used shortly afterwards in Prefront to record a node last appearance by a change of sign in the `NDEST` array; after that, it will not be needed anymore. For the considered example, the final result is `IELAS(NNODE) = 1, 3, 3, 3, 2 .`



Unconquered ⬜ Attacked 🟥 Conquered 🟦

Fig. 1a

NDEST(IE,NDOFE)			
1	0	0	0
2	0	0	0
3	0	0	0

IFRON	NACNO(MFRON)
1	0
2	0
3	0
4	0
5	0

IELVA(MXDOF,2)		
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0

Fig. 1b

NDEST(IE,NDOFE)			
1	1	2	3
2	0	0	0
3	0	0	0

IFRON	NACNO(MFRON)
1	4
2	1
3	2
4	0
5	0

IELVA(MXDOF,2)		
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0

Fig. 1c

NDEST(IE,NDOFE)			
1	1	-2	3
2	0	0	0
3	0	0	0

IFRON	NACNO(MFRON)
1	4
2	0
3	2
4	0
5	0

IELVA(MXDOF,2)		
1	2	1
2	0	0
3	0	0
4	0	0
5	0	0

Fig. 1d

NDEST(IE,NDOFE)			
1	1	-2	3
2	2	3	4
3	0	0	0

IFRON	NACNO(MFRON)
1	4
2	3
3	2
4	5
5	0

IELVA(MXDOF,2)		
1	2	1
2	0	0
3	0	0
4	0	0
5	0	0

Fig. 1e

NDEST(IE,NDOFE)			
1	1	-2	3
2	2	3	-4
3	0	0	0

IFRON	NACNO(MFRON)
1	4
2	3
3	2
4	0
5	0

IELVA(MXDOF,2)		
1	2	1
2	4	5
3	0	0
4	0	0
5	0	0

Fig. 1f

NDEST(IE,NDOFE)			
1	1	-2	3
2	2	3	-4
3	3	2	1

IFRON	NACNO(MFRON)
1	4
2	3
3	2
4	0
5	0

IELVA(MXDOF,2)		
1	2	1
2	4	5
3	0	0
4	0	0
5	0	0

Fig. 1g

NDEST(IE,NDOFE)			
1	1	-2	3
2	2	3	-4
3	-3	-2	-1

IFRON	NACNO(MFRON)
1	0
2	0
3	0
4	0
5	0

IELVA(MXDOF,2)		
1	2	1
2	4	5
3	3	2
4	2	3
5	1	4

Figure 10.3: Frontal factorization process

Formation of the destination, active nodes and eliminated variables arrays

1. Before starting with the first finite element, `NDEST`, `NACNO` and `IELVA` are initialized to zero, see Fig. (10.3)a.

2. Element 1 is attacked. Nodes 4, 1 and 2 become active. The vector `NACNO` is searched for available space; since nothing has been stored yet, the three `NDOFE` will be able to occupy the first three equations of the front when the system will actually be formed. The destinations 1, 2, 3 of the first element are recorded in `NDEST`; the active nodes 4, 1, 2 from which the front-equations originate are recorded in `NACNO`, see Fig. (10.3b).

3. Before going to element 2, a scanning of the components of `IELAS` corresponding to the active nodes reveals that node 1 will never receive any contribution from the

further elements. Node 1 can therefore be deactivated. The destination number for one degree of freedom (`NDOFN` = 1) at node 1 receives a minus sign to indicate that the corresponding equation will be reduced and frozen in back-up storage when the actual process takes place. The space occupied by this equation in the front will be made available again; this is shown by a zero replacing number 1 in the array of active nodes (`NACNO(2) = 0`). For back-substitution it must also be recorded that the first variable eliminated comes from equation 2 of the front and belongs to node 1; this is done in `IELVA`, see Fig. (10.3c).

4. Element 2 is attacked. Its nodes 3, 2, 5 are checked against the list of active nodes: 4, 1, 2. This reveals that node 2 is active already and the corresponding equation will be third of the front; the contribution from node 2 will be added to it. Nodes 3 and 5 being active for the first time will be sent to rows 2 and 4 of the frontwidth where zeros in `NACNO` indicated available space. The front will then contain the upper triangular part of a 4×4 submatrix of the global stiffness, see Fig. (10.3d).

5. The array `IELAS` shows that node 5 can be deactivated; i.e. the fourth equation of the front will by then have been summed up, it can be reduced and frozen. Appropriate modifications are made to the destination and active nodes arrays: `NDEST(2,3)` changes sign, `NACNO(4)` becomes zero. In `IELVA` the second row becomes `IELVA(2,1)` = `IELVA(2,2)` to indicate that the second eliminated variable comes from equation 4 of the front and belongs to node 5, see Fig. (10.3e).

6. Element 3 is attacked, its nodes 2, 3, 4 are already active, `NACNO` is not modified. All contributions from element 3 will add up to existing equations 3, 2, 1 respectively, see Fig. (10.3f).

7. No more contributions will ever be added. The remaining equations in the front can be frozen away in reduced form while the corresponding remaining nodes are deactivated in the order of their last appearance in element 3. The chosen order of elimination, i.e. 3, 2, 1 for the equations in front and 2, 3, 4 for the nodes to which they belong is recorded in rows 3, 4, 5 of `IELVA`. This completes the mock elimination phase, see Fig. (10.3g).

Case of multiple DOF per nodes

When the elements possess more than one `DOF` per node (i.e. `NDOFN` > 1), a few more inner DO loops have to be added in the formation of the destination, active nodes and eliminated variables arrays.

1. `NDEST` must be enlarged because the number of columns is now superior to the number of element nodes (i.e. `NDOFE` > `NNODE`); the extra columns are filled with destination numbers of equations in front found during an inner DO loop over the `DOF` at each node.

2. `NACNO` becomes longer because node numbers are repeated for each `DOF` which belongs to them; in the original `NACNO`, no difference is made between first, second, ... nodal `DOF`.

3. `IELVA` possesses a third column which, in regard to the node number, indicates the `DOF` number from which the eliminated variable originates; this column is also

filled by means of an inner DO loop over nodal `DOF`. For some types of element the number of nodal `DOF` is varying from node to node. In this case an integer array might be specified with the element type (coded, say, by `ITE`-number) giving the numbers of degrees of freedom `DOF` at the element nodes (i.e. numbers `NDOFN(INE), INE = 1,2,..., NNE`) considered in a special order. This special order must be used to inspect the element-node-list as they appear in one column of `INET` for each element of this type.

10.3.2 The actual assemblage and elimination phase

Partial assembling

For each individual element, the assemblage of a system either with the symmetric matrix or not proceeds much in the same way as the direct stiffness method.

- The rows of the global stiffness, `GSTIF`, and global load vectors, `GLOAD`, have been initialized to zero or partially summed up depending on whether or not the nodes are active for the first time.
- The element stiffness, `ESTIF`, and element load vector, `ELOAD`, are formed in the element subroutine.
- A pair of nested DO loops is performed over the element degrees of freedom (i.e. DO-loop-counters run over `IDOFE` = 1, 2, ... , `NDOFE`). So, the loop over rows `IDOFE` and the loop over columns `JDOFE` run, to consider all components in `ESTIF` (or in its half, if `ESTIF` is symmetric).
- By means of the destination array `NDEST`, one can find `IG`, and `JG`, which represent the global row `I` and global column `J` corresponding to the local degree of freedom `IDOFE` and `JDOFE`.
- The coefficients `ESTIF(IDOFE, JDOFE)` are added to `GSTIF(IG, JG)`.
- The loads `ELOAD(IDOFE)` are added to `GLOAD(IG)`.

It is well known that problems inducing symmetrical system matrices occur very often. In those cases only one (upper or lower) triangle part of matrices `ESTIF` and `GSTIF` can be used. To increase speed of computation, these matrices were usually considered as one dimensional arrays in the seventieths. So, the only program peculiarity – today rather of a historical nature – are transformations between indexes of two- and one dimensional arrays, as show us formulas (10.52)

The conversion of a symmetric matrix $\mathbf{K} \equiv K_{ij},\ i,j = 1,2,...,\mathrm{n}$ from its "natural" 2D-array into the 1D-array, so called vector form of a matrix, $\mathbf{K} \equiv K_k,\ k = 1,2,...,\mathrm{N}$, where $\mathrm{N} = \mathrm{n}(\mathrm{n}+1)/2$, is performed by means of two variants of function `INDC`:

$$\begin{aligned} k &= \texttt{INDC}\,(i\,(i-1)/2+j) \quad \text{if} \quad i \geq j\,, \quad i,j \leq \mathrm{n} \quad \text{(store lower triangle)} \\ k &= \texttt{INDC}\,(j\,(j-1)/2+i) \quad \text{if} \quad i \leq j\,, \quad i,j \leq \mathrm{n} \quad \text{(store upper triangle)} \end{aligned} \tag{10.52}$$

Both functions give the same final vector starting from a symmetric matrix.

Elimination

In the ordinary Gaussian process, variables are eliminated in the order in which they are met going down the matrix. In the frontal technique, the order of elimination is different from the order of formation of the equations; and the order of formation is not straightforward but is governed by the available space in front.

Which means that, quite often, one finds oneself in the position illustrated on the 4×4 example here below where one has to eliminate first the variable x_3.

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} \\ K_{21} & K_{22} & K_{23} & K_{24} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ K_{41} & K_{42} & K_{43} & K_{44} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{Bmatrix} . \tag{10.53}$$

The result is

$$\begin{bmatrix} K_{11} - \frac{K_{13}}{K_{33}} K_{31} & K_{12} - \frac{K_{13}}{K_{33}} K_{32} & 0 & K_{14} - \frac{K_{13}}{K_{33}} K_{34} \\ K_{21} - \frac{K_{23}}{K_{33}} K_{31} & K_{22} - \frac{K_{23}}{K_{33}} K_{32} & 0 & K_{24} - \frac{K_{23}}{K_{33}} K_{34} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ K_{41} - \frac{K_{43}}{K_{33}} K_{31} & K_{42} - \frac{K_{43}}{K_{33}} K_{32} & 0 & K_{44} - \frac{K_{43}}{K_{33}} K_{34} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} f_1 - \frac{K_{13}}{K_{33}} f_3 \\ f_2 - \frac{K_{23}}{K_{33}} f_3 \\ f_3 \\ f_4 - \frac{K_{43}}{K_{33}} f_3 \end{Bmatrix}$$

or using of substitutions $\tilde{K}_{11} = K_{11} - \frac{K_{13}}{K_{33}} K_{31}$, $\tilde{f}_1 = f_1 - \frac{K_{13}}{K_{33}} f_3$, etc in a lucid form

$$\begin{bmatrix} \tilde{K}_{11} & \tilde{K}_{12} & K_{13} & \tilde{K}_{14} \\ \tilde{K}_{21} & \tilde{K}_{22} & K_{23} & \tilde{K}_{24} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ \tilde{K}_{41} & \tilde{K}_{42} & K_{43} & \tilde{K}_{44} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ f_3 \\ \tilde{f}_4 \end{Bmatrix} . \tag{10.54}$$

Outside of row and column 3, the symmetry is maintained: one can keep working on the half upper triangle of front. But for back-substitution purposes, *the whole equation* 3 is necessary; left of diagonal terms K_{31} and K_{32} are fetched in the 'symmetric' column under the name K_{13} and K_{23} and the whole row is frozen away; immediately afterwards, row 3 is reset to zero to prepare a free available space for summation of a new equation which is going to come and occupy the available space. In the program column 3 is also set to zero rather than computed and found equal to zero anyway.

To save time on peripheral operations row 3 is not immediately sent to back-up storage; it is temporarily sent to a buffer array (matrix). When several variables have been eliminated and the buffer matrix is full, its content is sent to back-up storage. Full or not the last buffer needs not to be stored away; it is going to be used very soon for back-substitution.

The actual frontwidth, monitored by `NFRON`, is updated during assemblage of each element and again after all permissible eliminations accompanying this element have been performed. It starts from zero before attacking the first element and comes back to zero when the last elimination has been performed. Thanks to Prefront execution we are now sure of validity of inequality `MFRON` $\leq$ `MAXFW`.

Case of prescribed displacements

When the elimination reaches a *prescribed variable*, the process becomes trivial. The right-hand-sides of the system of equations are modified and the corresponding column of the front matrix (outside the diagonal term) is set to zero. If, for instance, x_3 is fixed in the 4×4 example given above, the system becomes

$$\begin{bmatrix} K_{11} & K_{12} & 0 & K_{14} \\ K_{21} & K_{22} & 0 & K_{24} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ K_{41} & K_{42} & 0 & K_{44} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} f_1 - K_{13}x_3 \\ f_2 - K_{23}x_3 \\ f_3 \\ f_4 - K_{13}x_3 \end{Bmatrix} + \begin{Bmatrix} 0 \\ 0 \\ r_3 \\ 0 \end{Bmatrix} . \tag{10.55}$$

Again K_{43} has to be fetched under the name K_{34} since only the upper triangular part of the matrix is retained in front. But the whole row 3 has to be sent into buffer matrix and later to bach-up storage.

The nodal force f_3 is the sum of contribution coming from all loaded elements to which node 3 belongs. It is not the definitive nodal force; in addition to f_3 there must be a concentrated, but still unknown force r_3 applied at this node to produce the prescribed displacement; the value of r_3 can only be found in the back-substitution phase.

The substantially more complex and more sophisticated approach takes into consideration when so called multipoint constraints (MPC) have to be implemented. In such cases no *single* degree of freedom (i.e. single component of the vector $\mathbf{x}$ of 'unknowns') but a number of, say m, *linear combinations* among several – say 2, 9 or all n – components of $\mathbf{x}$, are prescribed. Problem of MPC-implementation into system of algebraic equations can be expressed analogously as the algebraic system (10.4) alone:

$$\mathbf{K}\,\mathbf{x} = \mathbf{f} \qquad \mathbf{C}^{\mathrm{T}}\,\mathbf{x} = \mathbf{c}\,, \tag{10.56}$$

where
$\mathbf{C}^{\mathrm{T}}$ stands for given $(m \times n)$ matrix, ie. $\mathrm{rank}(\mathbf{K}) = n$ and
$\mathbf{c}$ for $(m \times 1)$ vector collecting the prescribed absolute values of linear combinations of unknowns.

is need not where the conditions some of methods as least square (penalty) method, Lagrangian multipliers method, Orthogonal projection method and Row and column reduction

and for Schwarz error-functions hold

$$\begin{aligned} F_{\mathbf{er}} &= \frac{1}{2}\,\|\mathbf{e}\|^2_{\mathrm{K}}\,, & F_{\mathbf{er}} &\geq 0 \;\forall\, \mathbf{e}\,, & F_{\mathbf{er}} &= 0 \Leftrightarrow \mathbf{e} = \mathbf{0}\,, \\ F_{\mathbf{rr}} &= \frac{1}{2}\,\|\mathbf{e}\|^2_{\mathrm{KK}}\,, & F_{\mathbf{rr}} &\geq 0 \;\forall\, \mathbf{e}\,, & F_{\mathbf{rr}} &= 0 \Leftrightarrow \mathbf{e} = \mathbf{0}\,, \\ F_{\mathbf{ee}} &= \frac{1}{2}\,\|\mathbf{e}\|^2_{\mathrm{L2}}\,, & F_{\mathbf{ee}} &\geq 0 \;\forall\, \mathbf{e}\,, & F_{\mathbf{ee}} &= 0 \Leftrightarrow \mathbf{e} = \mathbf{0}\,. \end{aligned} \tag{10.57}$$

10.3.3 Back-substitution phase

Back-substitution and output of results

The back-substitution can be understood as a frontal process applied backwards. All lost territory is reconquered as the front moves back through all elements taken in reverse order. It is better explained again by means of the example shown on Fig. (10.3a) to Fig. (10.3g).

1. Element 3 is reconquered. By counting the negative signs for element 3 in the destination array `NDEST`, one knows that the three variables which were formed in rows 1, 2 and 3 of the front can be recovered. The eliminated variables array `IELVA` shows that these rows were reduced in the order 3, 2, 1 and that these variables correspond to nodes 2, 3, 4 which are precisely the node of element 3, see Fig. (10.3g). Therefore, one keeps recalling buffer matrices and back-substituting from the last equation upwards until these three displacements have been found.

2. The displacements for element 3 are gathered in the vector of element displacements `DISPE` and printed. A call can then be made to the element stress subroutine, see Fig. (10.3f).

3. Element 2 is reconquered. The arrays `NDEST` and `IELVA` show that one variable can be recovered: the one which was originally in equation 2 of front, which was second to be eliminated and which originated from node 5, see Fig. (10.3e). Back-substitution eventually requires recalling another buffer matrix.

4. Other nodal displacements for element 2 have already been calculated and still in the front vector `GLOAD`; all element 2 displacements are gathered in `DISPE` and printed; they can be immediately used for computing the stress, see Fig. (10.3d).

5. Element 1 is reconquered, the variable originally stored in equation 2 of front, the first to be eliminated and originating from node 1 can be recovered Fig. (10.3c).

6. Element 1 displacements are gathered, printed and used to compute stresses Fig. (10.3b).

Although the elimination has been performed in a nontraditional way, the reduced matrix which is sent on backup storage via buffer matrices finally can be treated in the usual way for back-substitution. Going upwards from the last one coefficient - one unknown equation, every new line offers only one new unknown quantity.
When an element has been reconquered, all its nodal displacements are known. For, either they have appeared for the last time with this element in the assemblage process, have received a minus sign on their destination and are recovered while the element is being reconquered, or they have appeared later in the assemblage process and have been recovered already when the elements are taken in reverse order for back-substitution. Not only are all element displacements known but, because they belong to nodes which are necessarily active while the element is being considered, all their values are presently in the front vector `GLOAD` ready to be picked up for gathering of the element-displacement-vector `DISPE`.

Case of prescribed displacements

Where the nodal displacement is imposed, one would like to know the magnitude of the concentrated load which must be applied at this node to resist the forces coming from the neighbouring loaded elements and to produce the imposed displacement.

One takes again the 4×4 example used in (10.53). Its reduced form is

$$\begin{bmatrix} K_{31} & K_{32} & K_{33} & K_{34} \\ K_{11} & K_{12} & 0 & K_{14} \\ 0 & \tilde{K}_{22} & 0 & \tilde{K}_{24} \\ 0 & 0 & 0 & \tilde{K}_{44} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} f_3 + r_3 \\ f_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \end{Bmatrix} . \tag{10.58}$$

where:

- The equations appear in a different order because of the elimination process;
- the primes denote coefficients modified for reduction purposes either in (10.55) or immediately afterwards;
- the displacement x_3 is imposed.

Because of the back-substitution organization all other relevant variables are known already and are still available in the frontal part of `GLOAD`. When it comes to determining the unknown associated with equation 3, one can therefore write

$$-r_3 \;=\; f_3 - K_{31}\,x_1 - K_{32}\,x_2 - K_{33}\,x_3 - K_{34}\,x_4 \;. \tag{10.59}$$

This looks very much like an ordinary back-substitution except that the diagonal term, which is usually skipped, is also included in the summation on the right. r_3 is the required reaction force complementing equilibrium in (10.59).

10.3.4 Discussion of the frontal technique

The front solution is a very efficient direct solution. Its main attraction is that variables are seized up later and eliminated earlier than in most methods. The active life of a node spans from the element in which it first appears to the element in which it last appears. This has the following consequences.
1) In the frontal technique, the ordering of elements is crucial and the ordering of nodal numbers is irrelevant: the opposite requirement to which governs a banded solution and one easier to satisfy because, usually, there are fewer elements than nodes, especially in three-dimensional problems. Further, if a mesh is found too coarse in some region, it does not necessitate extensive renumbering of the nodes.
2) The storage requirements are at most the same as with a banded Gaussian solution, generally less. This is especially true for structures analyzed by means of elements with midside nodes, which give "re-entrant band" in classical schemes. Several justificative examples can be found in [1] a [5]. One of the most striking is striking is shown on Fig. 11. An ordinary banded solver leads to the total number of equations as the half-band width; this is reduced to two by the frontal technique.

$$\begin{bmatrix} \circ & & & & & & & & \circ \\ & \circ & & & & & & & \circ \\ & & \circ & & & & & & \circ \\ & & & \circ & & & & & \circ \\ & & & & \circ & & & & \circ \\ & & & & & \circ & & & \circ \\ & & & & & & \circ & & \circ \\ & & & & & & & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \end{bmatrix} \tag{10.60}$$

3) Because of the compactness of the front and because variables are eliminated as soon as possible, the zero coefficients are minimized and the total arithmetical operations are fewer then with other methods.
4) Because the new equations occupy the first available lines in the front, there is no need for a bodily shift of the in-core band as in many other large capacity equation solvers.
5) When a front program is used there is no need to send the element stiffness on back-up storage and recover them for assembly; they are assembled as soon as formed; the displacements are also used at the element level as soon as they are obtained by back-substitution. One can therefore save on peripheral operations. But this advantage will disappear when the frontal technique is incorporated as a solution routine in an existing program.
6) An inconvenient is the cumbersome bookkeeping which is required. Since it is entirely performed with integer variables however, it uses little space and computer time.

10.4 Concluding remarks

The simplest version of the frontal algorithm was presented in detail. This version can handle only very simple computational models for which are true the following facts: - number of nodes for each element is the same (here `NNODE`=3 was accepted), - number of degrees of freedom per node `NDOFN` is the same for the whole FE-model, - moreover, we used `NDOFN`=1 (as corresponds to scalar field analysis), - only the simplest case of boundary condition was considered, - only one right-hand-side vector was considered for the simultaneous processing, - none strategy concerning of the external storage handling was studied.

Absence of any just mentioned simplifications complicates the described algorithm version. However, any solver for professional use needs to works with different element types, unequal `DOF`-numbers for nodes, with boundary and/or sophisticated additional conditions expressed in the forms of linear combinations among unknown parameters, etc. See e.g. non-symmetr. frontal [21], [5]. See e.g. multifrontal [7], [8].

Bibliography

[1] D.G. Ashwell and R.H. Gallagher. *Finite Elements for Thin Shells and Curved Members*, chapter The Semiloof Shell Element. John Wiley and Sons, New York, 1976.

[2] O. Axelsson. *Iterative Solution Methods.* Cambridge University Press, New York, 1994.

[3] T. Chan and T. Mathew. Domain decomposition algorithms. *Acta Numerica*, pages 61–143, 1944.

[4] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings 24th National Conference of the Association for Computing Machinery*, pages 157–172. Brandon Press, New Jersey, 1969.

[5] I.S. Duff. Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. *SIAM J. Sci. Stat. Comput.*, 5:270–280, 1984.

[6] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[7] I.S. Duff and J.K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 9:302–325, 1983.

[8] I.S. Duff and J.K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM J. Sci. Stat. Comput.*, 5:633–641, 1984.

[9] A.M. Erisman, R.G. Grimes, J.G. Lewis, and W.G.Jr. Poole. A structuraly stable modification of hellerman-rarick's p^4 algorithm for reordering unsymmetric sparse matrices. *SIAM J. Numer. Anal.*, 22:369–385, 1985.

[10] C. Farhat and F.-X. Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *Int. J. Num. Meth. in Engng.*, 32(12):1205–1227, 1991.

[11] C.A. Felippa. Solution of linear equations with skyline-stored symmetric matrix. *Computers and Structures*, 5:13–29, 1975.

[12] G.A. Fonder. The frontal solution technique: Principles, examples and programs. Technical report, Department of Civil Engineering, University of Liège, Belgium, 1974.

[13] L. Gastinel. *Linear Numerical Analysis*. Kershaw Publishing, London, 1983.

[14] A. George. *Computer implementation of the finite-element method*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, 1971. Report STAN CS-71-208, Ph.D Thesis.

[15] A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[16] A. George. An automatic one-way dissection algorithm for irregular finite-element problems. *SIAM J. Numer. Anal.*, 17:740–751, 1980.

[17] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer, Berlin, New-York, 1985.

[18] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*. Springer, New-York, 1994.

[19] E. Hellerman and D.C. Rarick. Reinversion with the preassigned pivot procedure. *Math. Programming*, 1:195–216, 1971.

[20] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49(12):409–436, 1952.

[21] P. Hood. Frontal solution program for unsymmetric matrices. *Int. J. Num. Meth. in Engng.*, 10:379–400, 1976.

[22] B.M. Irons. A frontal solution program for finite element analysis. *Int. J. Num. Meth. in Engng.*, 2(1):5–32, 1970.

[23] A. Jennings. *Matrix computation for engineers and scientists.* John Wiley and Sons, Chichester, New-York, Brisbane and Toronto, 1977.

[24] W. Kahan. Numerical linear algebra. *Canadian Math. Bull.*, 9:757–801, 1966.

[25] V. Morozov. *Methods for Solving Incorrectly Posed Problems.* Springer-Verlag, New York, 1984.

[26] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics.* Texts in Applied Mathematics, 37. Springer-Verlag, New York, 2000.

[27] H.A. Schwarz. Über einen grenzübergang durch alternierendes verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich*, 15:272–286, 1870.

[28] S.W. Sloan and M.F. Randolph. Automatic element reordering for finite-element analysis with frontal schemes. *Int. J. Num. Meth. in Engng.*, 19:1153–1181, 1983.

[29] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition.* Cambridge University Press, 1996.

[30] A. Toselli and O. Widlund. *Domain Decomposition Methods – Algorithms and Theory.* Springer-Verlag, Berlin, Heidelberg, New York, 2005.

[31] C.W. Ueberhuber. *Numerical Computation.* Springer, Berlin, 1994.

Chapter 11

Sparse Storage Schemes

This part was written and is maintained by Petr Pařík. More details about the author can be found in the Chapter 16.

In mechanics of solids, mathematical models are usually described by a system of ordinary or partial differential equations. Discretization of these equations results in linearized matrix-vector equations, which are suitable for a numerical solution on a computer. Although there can be a large number of equations (variables) to describe the problem, each equation usually contains only a few variables due to the *local* dependencies between the variable and its derivative at a certain discretization point. This results in a coefficient matrix with a majority of elements zero; matrices of this structure are said to be *sparse*. Typically, the discretized system consists of $n = 10^3$ to 10^5 equations. A full (dense) coefficient matrix requires n^2 elements to be stored, which in the case of higher dimensions quickly overflows the capacity of any computer. However, if the sparse structure of the coefficient matrix is exploited and a suitable storage scheme is used, even a very large problems can be solved.

The various storage schemes for both dense and sparse matrices are described in sections 11.1 and 11.2.

The choice of a proper storage scheme for a particular algorithm is not a trivial task, as it has direct impact on the total time and space required by the solution method. Both direct and iterative methods have different demands the matrix storage scheme, and are discussed in sections 11.3 and 11.4.

11.1 Storage Schemes for Dense Matrices

Dense matrices have all or most of the elements non-zero, therefore no reduction of the storage space is possible as all elements need to be stored. For practical purposes, only very small matrices ($n \leq 10$) are stored as dense matrices, because dense matrices of higher dimensions have excessive storage requirements. Moreover, solution algorithms on large dense matrices are generally very slow, because mathematic operations have to be carried out for each and every single matrix element, even if it is zero. Dense storage schemes are however very useful for storing submatrices of *block sparse matrices*, which are discussed in the next section.

11.1.1 Full Storage

The full storage format is the simplest approach to store dense matrices: all matrix elements are stored in an array of floating-point values, either in a row-wise or column-wise fashion. There is no algorithmical overhead as all matrix elements can be accessed directly. Storage requirements are n^2 floating-point values. Example: Matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

can be stored in an array `val` in the full storage format by rows

i	1	2	3	4	5	6	7	8	9
`val(i)`	a_{11}	a_{12}	a_{13}	a_{21}	a_{22}	a_{23}	a_{31}	a_{32}	a_{33}

or by columns

i	1	2	3	4	5	6	7	8	9
`val(i)`	a_{11}	a_{21}	a_{31}	a_{12}	a_{22}	a_{32}	a_{13}	a_{23}	a_{33}

In FORTRAN, the latter (column) version is used internally to store multi-dimensional arrays.

11.1.2 Triangular Storage

The triangular storage format is useful for storing *symmetric* or *triangular matrices*. It is similar to the full storage format, however only the upper—or lower—triangular part of the matrix is stored in an array of floating-point values. Matrix elements are accessed directly with an additional simple logical test to determine in which part of the matrix the element is. For matrix elements which are not stored, the value of the symmetric element (i.e. element with swapped row and column indices) is returned in the case of a symmetric matrix, or zero in the case of a triangular matrix. Storage requirements are $n(n+1)/2$ floating-point values. Example: Matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{pmatrix}$$

can be stored in an array `val` in the triangular storage format by columns (upper triangle) or by rows (lower triangle)

i	1	2	3	4	5	6
`val(i)`	a_{11}	a_{12}	a_{22}	a_{13}	a_{23}	a_{33}

or by rows (upper triangle) or by columns (lower triangle):

i	1	2	3	4	5	6
`val(i)`	a_{11}	a_{12}	a_{13}	a_{22}	a_{23}	a_{33}

11.2 Storage Schemes for Sparse Matrices

Sparse matrices have a large percentage of zero elements, i.e. the number of non-zero elements m on any row is much lower that the number of elements on a row ($m \ll n$). Reduction of storage space is possible by storing only the non-zero elements, but because the non-zero elements can generally reside at arbitrary locations in the matrix, these locations, or coordinates (i.e. row and column indices), generally need to be stored along with the element values. Although this increases the storage requirements, it is not an issue except for very small matrices or matrices with a very small percentage of non-zero elements. In those cases, a dense storage scheme is usually more suitable. If a sparse matrix is symmetric, it is possible to reduce the storage space to approximately one half by storing only one triangular part of the matrix.
Band matrices and *skyline matrices* can be considered a special case of sparse matrices. Their structure can be exploited in storage schemes that have less space requirements and are less complex than storage schemes for sparse matrices with a general structure.

In the description of the different storage schemes, $\mathbf{A} \in \mathrm{R}^{n\times n}$ denotes the sparse matrix, n denotes the dimension of the matrix (the number of rows and columns of the matrix), and nnz denotes the number of non-zero elements in the matrix.

More information about sparse matrix storage schemes can be found for example in [3] and [1].

11.2.1 Coordinate Storage

The coordinate storage format is the simplest approach to store sparse matrices: the values of the non-zero elements are stored along with their coordinates (i.e. row and column indices) within three separate arrays `val`, `row_idx` and `col_idx`, all of the length nnz. For any element value `val(k)`, the row and column indices can be retrieved from `row_idx(k)` and `col_idx(k)`, and vice versa. The order of elements in the array `val` is irrelevant, which makes this scheme not very efficient for traversing the matrix by rows or columns. Also, storage requirements of nnz floating point values and $2 \times nnz$ integer values are higher than other sparse storage schemes.

Example: Matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 \\ 0 & a_{22} & a_{23} & a_{24} & 0 \\ 0 & 0 & a_{33} & 0 & a_{35} \\ a_{41} & 0 & 0 & a_{44} & a_{45} \\ 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

can be stored using the coordinate format (elements are intentionally in an unsorted order):

i	1	2	3	4	5	6	7	8	9	10	11	12	13
`val(i)`	a_{41}	a_{35}	a_{33}	a_{52}	a_{11}	a_{23}	a_{55}	a_{24}	a_{12}	a_{44}	a_{14}	a_{45}	a_{22}
`row_idx(i)`	4	3	3	5	1	2	5	2	1	4	1	4	2
`col_idx(i)`	1	5	3	2	1	3	5	4	2	4	4	5	2

11.2.2 Compressed Row Storage

The compressed row storage format puts subsequent non-zero elements of the matrix in contiguous array locations, allowing for a more efficient storage of element coordinates. The values of the non-zero elements are stored in the floating-point array `val` as they are traversed in a row-wise fashion. The integer array `col_idx` contains the column indices of elements as in the coordinate storage format. However, instead of storing nnz row indices, only pointers to the beginning of each row are stored in the integer array `row_ptr` of length $n+1$. By convention, `row_ptr(n+1)=nnz+1`. For any element value `val(k)`, the column index j can be retrieved from `j=col_idx(k)` and row index i from `row_ptr(i)<=k<row_ptr(i+1)`, and vice versa. Storage requirements are nnz floating-point values and $nnz+n+1$ integer values.

Example: Matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 \\ 0 & a_{22} & a_{23} & a_{24} & 0 \\ 0 & 0 & a_{33} & 0 & a_{35} \\ a_{41} & 0 & 0 & a_{44} & a_{45} \\ 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

can be stored using the compressed row storage format

i	1	2	3	4	5	6	7	8	9	10	11	12	13
`val(i)`	a_{11}	a_{12}	a_{14}	a_{22}	a_{23}	a_{24}	a_{33}	a_{35}	a_{41}	a_{44}	a_{45}	a_{52}	a_{55}
`col_idx(i)`	1	2	4	2	3	4	3	5	1	4	5	2	5

j	1	2	3	4	5	6
`row_ptr(j)`	1	4	7	9	12	14

11.2.3 Compressed Column Storage

The compressed column storage format (CRS) is identical to the compressed row storage format (CRS), except that columns are traversed instead of rows. In other words, the CCS format for $\mathbf{A}$ is the CRS format for $\mathbf{A}^{\mathrm{T}}$.

11.2.4 Block Compressed Row Storage

If a sparse matrix is composed of square dense blocks of non-zero elements in a regular pattern, the compressed row storage (or compressed column storage) format can be modified to exploit such block structure. In this storage format, the matrix is partitioned into small blocks with an equal size, and each block is treated as a dense matrix, even though it may have some zero elements. If n_{b} is the size of each block and nnz_{b} is the number of non-zero blocks in the $n \times n$ matrix $\mathbf{A}$, then the storage needed for the non-zero blocks is $nnz_{\mathrm{b}} \times n_{\mathrm{b}}^2$ floating-point values. The block dimension N of $\mathbf{A}$ is defined by $N = n/n_{\mathrm{b}}$.

Similar to the compressed row storage format, three arrays are needed to store the matrix: a three-dimensional floating-point array `val(1:nnz`$_{\mathrm{b}}$`,1:n`$_{\mathrm{b}}$`,1:n`$_{\mathrm{b}}$`)`, which stores the non-zero blocks in a row-wise fashion, an integer array `col_idx(1:nnz`$_{\mathrm{b}}$`)`, which stores the original matrix column indices of the (1,1) element of the non-zero blocks, and an integer array `row_ptr(1:N+1)`, which stores pointers to the beginning of each block row in `val` and `col_idx`. Storage requirements for indexing arrays are $nnz_{\mathrm{b}}+N+1$ integer values. The savings in storage space and reduction in time spent doing indirect addressing for block compressed row storage format over plain compressed row storage format can be significant for matrices with a large n_{b}.

11.2.5 Block Compressed Column Storage

The block compressed column storage format (BCCS) is identical to the block compressed row storage format (BCRS), except that columns are traversed instead of rows. In other words, the BCCS format for $\mathbf{A}$ is the BCRS format for $\mathbf{A}^{\mathrm{T}}$.

11.2.6 Compressed Diagonal Storage

The compressed diagonal storage format is useful for storing *band matrices*.

In this storage format, only the main diagonal and subdiagonals containing non-zero elements are stored. The $n \times n$ band matrix with left bandwidth p and right bandwidth q is stored in a two-dimensional floating-point array `val(1:n,-p:q)`. It involves storing some zero elements, because some array elements do not correspond to actual matrix elements. Because the element coordinates are given implicitly by p and q, no additional indexing arrays are needed. Storage requirements are $(p+q+1)\times n$ floating-point values.

Example: Matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & 0 & a_{35} \\ 0 & 0 & 0 & a_{44} & a_{45} \\ 0 & 0 & 0 & 0 & a_{55} \end{pmatrix}$$

can be stored using the compressed diagonal storage format (array entries which do not correspond to matrix elements are denoted by —)

i	1	2	3	4	5
`val(i,-1)`	—	0	a_{32}	0	0
`val(i,0)`	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}
`val(i,1)`	a_{12}	a_{23}	0	a_{45}	—
`val(i,2)`	0	a_{24}	a_{35}	—	—

11.2.7 Jagged Diagonal Storage

The jagged diagonal storage format is useful for storing *band matrices*, especially in iterative solution methods on parallel and vector processors. It is more complex than the compressed diagonal storage format, but has less space requirements.

It the simplified version, all non-zero elements are shifted left and the rows are padded with zeros to give them equal length. The elements of the 'shifted' matrix are then stored in a column-wise fashion in the floating-point array `val` similarly to the compressed diagonal storage format. Column indices of the elements are stored in the integer array `col_idx`. Row indices are not stored as they can be determined implicitly.

The simplified jagged diagonal storage format is also called *Purdue storage*. Storage requirements are $n \times nnz_{\mathrm{rm}}$ floating-point values and $n \times nnz_{\mathrm{rm}}$ integer values, where nnz_{rm} is the maximum number of non-zeros per row.

Example: Matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & 0 & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{pmatrix} \to \begin{pmatrix} a_{11} & a_{12} & a_{14} & 0 \\ a_{22} & a_{23} & 0 & 0 \\ a_{31} & a_{33} & a_{34} & 0 \\ a_{42} & a_{44} & a_{45} & a_{46} \\ a_{55} & a_{56} & 0 & 0 \\ a_{65} & a_{66} & 0 & 0 \end{pmatrix}$$

can be stored using the simplified jagged diagonal storage format

i	1	2	3	4	5	6
`val(i,1)`	a_{11}	a_{22}	a_{31}	a_{42}	a_{55}	a_{65}
`val(i,2)`	a_{12}	a_{23}	a_{33}	a_{44}	a_{56}	a_{66}
`val(i,3)`	a_{14}	0	a_{34}	a_{45}	0	0
`val(i,4)`	0	0	0	a_{46}	0	0

i	1	2	3	4	5	6
`col_idx(i)`	1	2	1	2	5	5
`col_idx(i)`	2	3	3	4	6	6
`col_idx(i)`	4	0	4	5	0	0
`col_idx(i)`	0	0	0	6	0	0

The jagged diagonal storage format uses row compression to remove the disadvantage of storing padded zeros, which can be quite inefficient if the bandwidth varies greatly. In this storage format, rows are reordered descendingly according to the number of non-zero elements per row, and stored in the compressed row storage format in the floating-point array `val`. Column indices are stored in the integer array `col_idx`, row (jagged diagonal) pointers in the integer array `jd_ptr` and row permutation (reordering) in the integer array `perm`.

11.2.8 Skyline Storage

The skyline storage format is useful for storing *skyline matrices* (also called *variable band matrices* or *profile matrices*), especially in direct solution methods.

In this storage format, matrix elements are stored in the floating-point array `val` in a column-wise fashion, each column beginning with the first non-zero element and ending with the diagonal element. Corresponding column pointers are stored in the integer array `col_ptr`. Row indices are not stored as they can be determined implicitly.

In the case of a non-symmetric matrix, row-oriented version of skyline storage format is used to store the lower triangular part of the matrix; there is a number of ways to link the two parts together.

Because any zeros after the first non-zero element in a column have to be stored, the skyline storage format is most effective when non-zero elements are tightly packed around the main diagonal. Storage requirements are nnz floating-point values and $n+1$ integer values.

Example: Matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & 0 & a_{13} & 0 & 0 \\ 0 & a_{22} & 0 & a_{24} & 0 \\ a_{13} & 0 & a_{33} & a_{34} & 0 \\ 0 & a_{24} & a_{34} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{45} & a_{55} \end{pmatrix}$$

can be stored using the column-oriented skyline storage format

i	1	2	3	4	5	6	7	8	9	10
`val(i)`	a_{11}	a_{22}	a_{13}	0	a_{33}	a_{24}	a_{34}	a_{44}	a_{45}	a_{55}

j	1	2	3	4	5	6
`col_ptr(j)`	1	2	3	6	9	11

11.3 Storage Schemes for Iterative Methods

For iterative solvers, any sparse matrix storage scheme presented in the previous section can be used as described. Iterative methods are based on the matrix-vector multiplication and do not suffer from *fill-in* (explained in the next section), i.e. the storage requirements remain constant during the solution process.

The choice of a suitable storage scheme for an iterative method depends on the type of the method as well as the type of the problem. Often a certain storage scheme can be chosen in a natural way according to the distinctive features of the problem.

11.4 Storage Schemes for Direct Methods

The major drawback of direct solution methods is the *fill-in*—a process where an initially zero element of the matrix becomes non-zero during the elimination. This increases the storage requirements, and, in some cases, might even make the solution impossible because of insufficient storage space due to excessive fill-in. Direct solvers should therefore incorporate some method of minimizing the fill-in to avoid running out of memory. There are several approaches to adjust matrix storage schemes to make the fill-in possible, some of which are discussed below.

11.4.1 Fill-in analysis

A relatively simple way to implement the fill-in is to include corresponding (initially zero) entries in the matrix structure. The exact entries to be included must be identified by analyzing the fill-in, usually by some kind of a 'symbolic' elimination. Any presented matrix storage scheme can be used this way without modification, however it must be pointed out that the fill-in analysis requires an additional time and space to proceed.

11.4.2 Linked Lists

The compressed row (or column) storage format (or its block version) can be modified to implement the fill-in using *linked lists*. Newly added elements can only be appended at the end of the `val` array, so an additional information about the order of elements on matrix rows and/or columns need to be stored. For each element value `val(i)`, the array `row_lst(i)` contains the pointer to the next element in the same row, or zero pointer if there are no more elements on the row. A second array `col_lst` can be used analogically for storing the sequences of elements on matrix columns, to speed up the access to the elements in the same row. When the fill-in is added, the arrays `row_lst` and/or `col_lst` are modified to include the new element at the correct location of the corresponding matrix row and column. This modification of the compressed row storage format requires a lot of additional storage space as redundant information about the element indices are stored.

Example: Matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 \\ 0 & a_{22} & a_{23} & a_{24} & 0 \\ 0 & 0 & a_{33} & 0 & a_{35} \\ a_{41} & 0 & 0 & a_{44} & a_{45} \\ 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

is stored using the compressed row storage format. When a single fill-in (say element a_{34}) is added, the storage structure changes as follows

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`val(i)`	a_{11}	a_{12}	a_{14}	a_{22}	a_{23}	a_{24}	a_{33}	a_{35}	a_{41}	a_{44}	a_{45}	a_{52}	a_{55}	a_{34}
`col_idx(i)`	1	2	4	2	3	4	3	5	1	4	5	2	5	**4**
`row_lst(i)`	2	3	0	5	6	0	**14**	0	10	11	0	13	0	**8**
`col_lst(i)`	9	4	6	12	7	**14**	0	11	0	0	13	0	0	**10**

j	1	2	3	4	5	6
`row_ptr(j)`	1	4	7	9	12	**15**

11.4.3 Cyclically Linked Lists

Cyclically linked lists greatly reduce the overhead information needed to insert the fill-in when using the compressed row storage format (see previous subsection). The non-zero elements of the matrix are stored in the floating-point array `val` skipping the first n positions, so the array has the length of $n + nnz$. An integer array `row` is used to store either a pointer to the next non-zero element in the row, or if it was the last element, the index of the row itself. Another integer array `col` can be used analogically to store column indices to speed up the access to the elements by columns. New elements can be appended to the end of the `val` array and the `row` and `col` arrays can be modified without much effort.

Example: Matrix

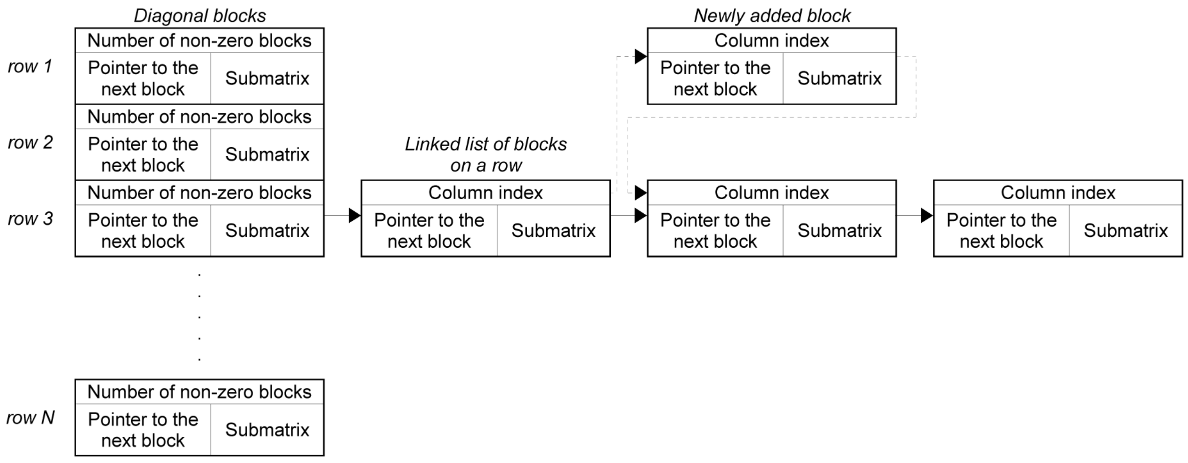

Diagonal blocks
Newly added block
row 1
row 2
row 3
row N
Number of non-zero blocks
Pointer to the next block
Submatrix
Linked list of blocks on a row
Column index

Figure 11.1: Block structure of the symmetric dynamic block storage format

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 \\ 0 & a_{22} & a_{23} & a_{24} & 0 \\ 0 & 0 & a_{33} & 0 & a_{35} \\ a_{41} & 0 & 0 & a_{44} & a_{45} \\ 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

can be stored using cyclically linked lists (the elements are deliberately disordered)

i	1	2	3	4	5	6	7	8	9	10
`val(i)`	0	0	0	0	0	a_{41}	a_{35}	a_{33}	a_{52}	a_{11}
`row(i)`	10	18	8	6	9	15	0	7	12	14
`col(i)`	10	14	11	16	7	0	17	0	0	6

i	11	12	13	14	15	16	17	18
`val(i)`	a_{23}	a_{55}	a_{24}	a_{12}	a_{44}	a_{14}	a_{45}	a_{22}
`row(i)`	13	0	0	16	17	0	0	11
`col(i)`	8	0	15	18	0	13	12	9

11.4.4 Dynamically Allocated Blocks

Modern programming languages support structured data types and dynamic memory allocation, concepts which were not available in the time of FORTRAN 77 or concurrent languages. These two features can be used to construct an advanced matrix storage scheme ([4], [2]) that is somewhat similar to the block compressed row (column) storage format.

The idea of this storage scheme is based on a simple structure called a ‘block’, which contains both the submatrix data and its indices. The blocks are stored by rows using linked lists, but can be also stored by columns, or both. Both symmetric and unsymmetric matrices can be stored; in the case of a symmetric matrix, the blocks are usually stored beginning with a diagonal block.

The storage space is not allocated all at once, but gradually, using memory allocation functions, one block at a time. That means it is not necessary to know or to guess the exact storage requirements beforehand like in all the previous storage schemes. In the most conservative version, the dynamic block storage format has the same storage requirements as the block compressed row storage format.

Bibliography

[1] M. Okrouhlík and et al. *Mechanika poddajných těles, numerická matematika a superpočítače*. Institute of Thermomechanics AS CR, Prague, 1997.

[2] P. Pařík. Implementation of sparse matrix storage schemes for direct solution methods. In *Book of Extended Abstracts, Computational Mechanics 2008, Nectiny*, 2008.

[3] C. W. Ueberhuber. *Numerical Computation*. Springer, Berlin, 1994.

[4] V. Vondrák. Description of the k3 sparse matrix storage system. In *unpublished, Technical University, Ostrava*, 2000.

Chapter 12

Object oriented approach to FEA

This part was written and is maintained by Vítězslav Štembera. More details about the author can be found in the Chapter 16.

Abstract

We describe an object oriented programming approach to the finite element method for solving a pressure load problem for an elastic isotropic body. Large deformations and large displacements are considered as well as a full three-dimensional case. We are going to solve the steady-state problem – the response of a body to surface pressure forces. The forthcoming text contains the theoretical background including numerical methods, two test examples and fully commented code in C# language.

12.1 Basics of continuum mechanics

The continuum mechanics theory in this section is taken from Holzapfel [1].

12.1.1 Basic tensor definition

Consider a continuum body which is embedded in the three-dimensional Euclidian space at time t. Let us introduce an undeformed configuration of the body at time $t = 0$, denoted by V_0 (reference configuration), and a deformed configuration V at time $t > 0$ (also called spatial configuration). Each point $\mathbf{X} \in V_0$ is mapped to the new position $\mathbf{x} \in V$ by the so-called deformation mapping

$$\mathbf{x}(t) = \mathbf{x}(\mathbf{X}, t). \tag{12.1}$$

Let us introduce a deformation tensor

$$\mathbf{F} = \frac{\partial \mathbf{x}(\mathbf{X}, t)}{\partial \mathbf{X}}. \tag{12.2}$$

The left and the right Cauchy-Green tensors (left or right means the position of tensor $\mathbf{F}$) are defined as

$$\mathbf{b} = \mathbf{F}\mathbf{F}^{\mathrm{T}}, \mathbf{C} = \mathbf{F}^{\mathrm{T}}\mathbf{F}. \tag{12.3}$$

$$\tag{12.4}$$

Note that tensor $\mathbf{b}$ is defined in the spatial configuration whereas tensor $\mathbf{C}$ is defined in the reference configuration. Moreover let us define the displacement field in the reference configuration

$$\mathbf{U}(\mathbf{X},t) = \mathbf{x}(\mathbf{X},t) - \mathbf{X}, \tag{12.5}$$

and the same in the spatial configuration

$$\mathbf{u}(\mathbf{x},t) = \mathbf{x} - \mathbf{X}(\mathbf{x},t). \tag{12.6}$$

The Euler-Lagrange strain tensor in the reference configuration is defined

$$\mathbf{G} = \frac{1}{2}\left(\mathbf{F}^{\mathrm{T}}\mathbf{F} - \mathbf{I}\right), \tag{12.7}$$

and the same in the spatial configuration

$$\mathbf{g} = \frac{1}{2}\left(\mathbf{I} - \mathbf{F}^{-\mathrm{T}}\mathbf{F}^{-1}\right), \tag{12.8}$$

between which the following relation holds [1]

$$\mathbf{g} = \mathbf{F}^{-\mathrm{T}}\mathbf{G}\mathbf{F}^{-1}. \tag{12.9}$$

The Cauchy (true) stress tensor, defined in the spatial configuration, is denoted by $\boldsymbol{\sigma}$. The so-called second Piola-Kirchhoff stress tensor $\mathbf{S}$, defined in the reference configuration, takes the form

$$\mathbf{S} = J\mathbf{F}^{-1}\boldsymbol{\sigma}\mathbf{F}^{-\mathrm{T}}, \quad J = \det\mathbf{F}. \tag{12.10}$$

12.1.2 Deformation problem definition

Suppose that the surface of the deformed body has two distinct parts

$$\partial V = \partial V_u \cup \partial V_t, \qquad \partial V_u \cap \partial V_t = \emptyset. \tag{12.11}$$

The initial-boundary value problem (in the strong form) for unknown displacement $\mathbf{u}$, where no volume forces are assumed, takes the form

$$\begin{aligned} \operatorname{div}\boldsymbol{\sigma} &= \mathbf{0} \quad \text{in } V, \\ \mathbf{u} &= \bar{\mathbf{u}} \quad \text{on } \partial V_u, \\ \boldsymbol{\sigma}\mathbf{n} &= \bar{\mathbf{t}} \quad \text{on } \partial V_t. \end{aligned} \tag{12.12}$$

where $\bar{\mathbf{u}}$ is the given surface displacement, $\mathbf{n}$ is the outward unit normal to ∂V_t and $\bar{\mathbf{t}} = -p\mathbf{n}$ and p is the outside pressure. In the weak form the problem transformes to [2]

$$\int_V \boldsymbol{\sigma} : \operatorname{grad} \boldsymbol{\varphi} \ \ \mathrm{d}V = \int_{\partial V_t} \bar{\mathbf{t}} \cdot \boldsymbol{\varphi} \ \ \mathrm{d}S \quad \forall \boldsymbol{\varphi} \in W(V), \tag{12.13}$$

where $W(V)$ is the given set of test functions defined in V for which $\boldsymbol{\varphi} = \mathbf{0}$ on ∂V_u. We suppose that all function in equation (12.13) are sufficiently smooth.

[1] Tensors $\mathbf{G}, \mathbf{g}$ are also denoted by $\mathbf{E}, \mathbf{e}$ in some books.

[2] $\mathbf{A} : \mathbf{B} = \sum_{i,j} A_{ij}B_{ij}$, $(\operatorname{div}\mathbf{A})_i = \frac{\partial A_{ij}}{\partial x_j}$, $(\operatorname{grad}\mathbf{v})_{ij} = \frac{\partial v_i}{\partial x_j}$.

12.1.3 Virtual displacement, first variation

Virtual displacement field $\delta\mathbf{u}$ is the difference between two neighboring displacement fields

$$\delta\mathbf{u} = \bar{\mathbf{u}} - \mathbf{u}. \tag{12.14}$$

The virtual displacement contains no derivatives, therefore it can be expressed in both coordinates as $\delta\mathbf{u} = \delta\mathbf{U}$. Therefore we do not distinguish between them and use only $\delta\mathbf{u}$ in the forthcoming text. The first variation of smooth (scalar, vector, tensor) function $\mathcal{F}(\mathbf{U})$ in the material coordinates is defined as

$$\delta\mathcal{F}(\mathbf{U}) = \mathcal{D}_{\delta\mathbf{u}}\mathcal{F}(\mathbf{U}) = \frac{\mathrm{d}}{\mathrm{d}\varepsilon}\mathcal{F}(\mathbf{U}+\varepsilon\delta\mathbf{u})\Big|_{\varepsilon=0}. \tag{12.15}$$

that is nothing else then the directional derivative in the direction of $\delta\mathbf{u}$. Let us compute two useful variations [3]

$$\begin{aligned}
&\delta\mathbf{F} = \delta(\mathbf{I}+\mathrm{Grad}\mathbf{U}) = \delta\mathrm{Grad}\mathbf{U} \overset{(12.15)}{=} \mathrm{Grad}\delta\mathbf{u},\\
&\delta\mathbf{G} \overset{(12.7)}{=} \frac{1}{2}\delta\left(\mathbf{F}^{\mathrm{T}}\mathbf{F}\right) = \frac{1}{2}\left[\left(\delta\mathbf{F}^{\mathrm{T}}\right)\mathbf{F} + \mathbf{F}^{\mathrm{T}}\delta\mathbf{F}\right] = \frac{1}{2}\left[\left(\mathbf{F}^{\mathrm{T}}\mathrm{Grad}\delta\mathbf{u}\right)^{\mathrm{T}} + \mathbf{F}^{\mathrm{T}}\mathrm{Grad}\delta\mathbf{u}\right].
\end{aligned} \tag{12.16}$$

In the spatial coordinates the first variation is defined as the push-forward of the variation in the material coordiantes (push-forward of the Green-Lagrange tensor was computed already in (12.9)) [4]

$$\begin{aligned}
\delta\mathbf{g} = \chi_*(\delta\mathbf{G}) &\overset{(12.9)}{=} \mathbf{F}^{-\mathrm{T}}\delta\mathbf{G}\mathbf{F}^{-1} \overset{(12.16)}{=} \frac{1}{2}\mathbf{F}^{-\mathrm{T}}\left[\left(\mathbf{F}^{\mathrm{T}}\mathrm{Grad}\delta\mathbf{u}\right)^{\mathrm{T}} + \mathbf{F}^{\mathrm{T}}\mathrm{Grad}\delta\mathbf{u}\right]\mathbf{F}^{-1} =\\
&= \frac{1}{2}\left[\left(\mathrm{Grad}\delta\mathbf{u}\mathbf{F}^{-1}\right)^{\mathrm{T}} + \mathrm{Grad}\delta\mathbf{u}\mathbf{F}^{-1}\right] = \frac{1}{2}\left[\mathrm{grad}^{\mathrm{T}}\delta\mathbf{u} + \mathrm{grad}\delta\mathbf{u}\right].
\end{aligned} \tag{12.17}$$

12.1.4 Principle of virtual work

In the following section we describe the stationary principal in which the displacement vector $\mathbf{u}$ is the only unknown field. As the test function in (12.13) let us take the virtual (unreal) infitesimal displacement $\delta\mathbf{u}$

$$\int_V \boldsymbol{\sigma} : \mathrm{grad}\ \delta\mathbf{u}\ \mathrm{d}V = \int_{\partial V_t} \bar{\mathbf{t}}\cdot\delta\mathbf{u}\ \ \mathrm{d}S, \tag{12.18}$$

We recompute the integral on the left-hand side

$$\begin{aligned}
&\int_V \boldsymbol{\sigma} : \mathrm{grad}\ \delta\mathbf{u}\ \ \mathrm{d}V \overset{\text{definition of :}}{=} \int_V \boldsymbol{\sigma} : \underbrace{\frac{1}{2}\left(\mathrm{grad}\delta\mathbf{u} + \mathrm{grad}^{\mathrm{T}}\delta\mathbf{u}\right)}_{\delta\mathbf{g}}\ \ \mathrm{d}V \overset{(12.17)}{=}\\
&= \int_V \boldsymbol{\sigma} : \mathbf{F}^{-\mathrm{T}}\delta\mathbf{G}\mathbf{F}^{-1}\ \ \mathrm{d}V = \int_{V_0} J\boldsymbol{\sigma} : \mathbf{F}^{-\mathrm{T}}\delta\mathbf{G}\mathbf{F}^{-1}\ \ \mathrm{d}^0V \overset{(12.10)}{=} \int_{V_0} \mathbf{S} : \delta\mathbf{G}\ \ \mathrm{d}^0V.
\end{aligned} \tag{12.19}$$

[3]Note definitions $\mathrm{Grad} := \sum_i \frac{\partial}{\partial X_i}$, $\mathrm{grad} := \sum_i \frac{\partial}{\partial x_i}$.

[4]Using definiton $\mathrm{sym}\mathbf{A} = \frac{1}{2}(\mathbf{A}^{\mathrm{T}} + \mathbf{A})$ we can also write $\delta\mathbf{G} = \mathrm{sym}(\mathbf{F}^{\mathrm{T}}\mathrm{Grad}\delta\mathbf{u}), \delta\mathbf{g} = \mathrm{sym}(\mathrm{grad}\delta\mathbf{u})$.

In the last step we have used indentity $\mathbf{A} : \mathbf{BC} = \mathbf{B}^{\mathrm{T}}\mathbf{A} : \mathbf{C} = \mathbf{AC}^{\mathrm{T}} : \mathbf{B}$. The balance law (12.18) transforms to

$$\int_{V_0} \mathbf{S} : \delta\mathbf{G} \quad \mathrm{d}^0V = \int_{\partial V_t} \bar{\mathbf{t}} \cdot \delta\mathbf{u} \;\; \mathrm{d}S, \tag{12.20}$$

which is called the principal of virtual work and it is the starting point for the finite element formulation.

12.2 Finite element method formulation

The text in this section is based on Okrouhlík [2]. The method presented here is the total lagrangian formulation. Let us suppose that we know body configuration at time t, denoted ${}^tC$, and we look for the new configuration at time $t + \triangle t$, denoted ${}^{t+\triangle t}C$. This one step procedure will allow us to solve any stationary problem which is our aim in this text. By derivation of formula (12.5) with respect to $\mathbf{X}$ we get new tensor $\mathbf{Z}$

$$\mathbf{Z} = \frac{\partial \mathbf{U}}{\partial \mathbf{X}} = \mathbf{F} - \mathbf{I}. \tag{12.21}$$

The Green-Lagrange deformation tensor can be rewritten as

$$\mathbf{G} = \frac{1}{2}\left(\mathbf{Z}^{\mathrm{T}}\mathbf{Z} + \mathbf{Z}^{\mathrm{T}} + \mathbf{Z}\right). \tag{12.22}$$

We define the increments

$$\triangle\mathbf{G} = \mathbf{G}^{t+\triangle t} - \mathbf{G}^t, \;\; \triangle\mathbf{S} = \mathbf{S}^{t+\triangle t} - \mathbf{S}^t, \tag{12.23}$$

where $\mathbf{S}$ is the second Piola-Kirchhoff stress tensor. To keep the notation simple, we omit in the forthcoming text the upper index t denoting the old time level (for example $\mathbf{G}$ instead of $\mathbf{G}^t$). Let us continue in the index form. The deformation tensor in the new configuration is

$$\triangle G_{ij} = G_{ij}^{t+\triangle t} - G_{ij} = \frac{1}{2}\left[\frac{\partial(U_i + \triangle U_i)}{\partial X_j} + \frac{\partial(U_j + \triangle U_j)}{\partial X_i} + \frac{\partial(U_k + \triangle U_k)}{\partial X_i}\frac{\partial(U_k + \triangle U_k)}{\partial X_j}\right] -$$

$$-\frac{1}{2}\left[\frac{\partial U_i}{\partial X_j} + \frac{\partial U_j}{\partial X_i} + \frac{\partial U_k}{\partial X_i}\frac{\partial U_k}{\partial X_j}\right] = \underbrace{\frac{1}{2}\left[\frac{\partial \triangle U_i}{\partial X_j} + \frac{\partial \triangle U_j}{\partial X_i} + \frac{\partial \triangle U_k}{\partial X_i}Z_{kj} + Z_{ki}\frac{\partial \triangle U_k}{\partial X_j}\right]}_{\triangle G_{ij}^{\mathrm{L}}} + \underbrace{\frac{1}{2}\frac{\partial \triangle U_k}{\partial X_i}\frac{\partial \triangle U_k}{\partial X_j}}_{\triangle G_{ij}^{\mathrm{N}}}$$

$i, j = 1, 2, 3.$

Summing over two same indices is considered. The second Piola-Kirchhoff stress tensor is computed via incremets with the use of the Hooke law

$$\triangle S_{ij} = \mathbb{C}_{ijkl}\triangle G_{kl}^{\mathrm{L}} \quad i, j = 1, 2, 3. \tag{12.24}$$

The integral (12.19) can be rewritten as

$$\begin{aligned}\int_{V_0} S_{ij}^{t+\triangle t}\, \delta G_{ij}^{t+\triangle t}\, \mathrm{d}^0V &= \int_{V_0} (S_{ij}+\triangle S_{ij})\, \delta(G_{ij}+\triangle G_{ij}^{\mathrm{L}}+\triangle G_{ij}^{\mathrm{N}})\, \mathrm{d}^0V \\ &= \int_{V_0} (S_{ij}+\triangle S_{ij})(\delta\triangle G_{ij}^{\mathrm{L}}+\delta\triangle G_{ij}^{\mathrm{N}})\, \mathrm{d}^0V \\ &\approx \underbrace{\int_{V_0} S_{ij}\delta\triangle G_{ij}^{\mathrm{L}}\;\mathrm{d}^0V}_{\delta U_1} + \underbrace{\int_{V_0} S_{ij}\delta\triangle G_{ij}^{\mathrm{N}}\;\mathrm{d}^0V}_{\delta U_2} + \underbrace{\int_{V_0} \triangle S_{ij}\delta\triangle G_{ij}^{\mathrm{L}}\;\mathrm{d}^0V}_{\delta U_3}.\end{aligned}$$

In the third row one of the terms is neglected because it is by one order smaller then other terms. In the forthcoming sections we transform terms δU_1, δU_2, δU_3 into matrix forms which are more suitable for computer implementation.

12.2.1 Transformation of term δU_1

Let us define vectors

$$\begin{aligned}\triangle\mathbf{g}^{\mathrm{L}} &= \left(\triangle G_{11}^{\mathrm{L}}, \triangle G_{22}^{\mathrm{L}}, \triangle G_{33}^{\mathrm{L}}, 2\triangle G_{12}^{\mathrm{L}}, 2\triangle G_{23}^{\mathrm{L}}, 2\triangle G_{31}^{\mathrm{L}}\right)^{\mathrm{T}}, \\ \mathbf{s} &= \left(S_{11}, S_{22}, S_{33}, S_{12}, S_{23}, S_{31}\right)^{\mathrm{T}},\end{aligned} \tag{12.25}$$

which immediately yields to

$$\int_{V_0} S_{ij}\; \delta\triangle G_{ij}^{\mathrm{L}}\; \mathrm{d}^0V = \int_{V_0} \left(\delta\triangle\mathbf{g}^{\mathrm{L}}\right)^{\mathrm{T}} \mathbf{s}\; \mathrm{d}^0V. \tag{12.26}$$

12.2.2 Transformation of term δU_2

Nonlinear part of the Green-Lagrange deformation tensor equals to

$$\triangle G_{ij}^{\mathrm{N}} = \frac{1}{2}\frac{\partial\triangle U_k}{\partial X_i}\frac{\partial\triangle U_k}{\partial X_j} \tag{12.27}$$

We rewrite the last equation (note that tensor $\mathbf{S}$ is symmetric)

$$\begin{aligned}S_{ii}\delta\triangle G_{ii}^{\mathrm{N}} &= \frac{S_{ii}}{2}\delta\left(\frac{\partial\triangle U_k}{\partial X_i}\right)^2 = S_{ii}\frac{\partial\triangle U_k}{\partial X_i}\delta\frac{\partial\triangle U_k}{\partial X_i},\; i=1,2,3, \\ S_{ij}\delta\triangle G_{ij}^{\mathrm{N}} + S_{ji}\delta\triangle G_{ji}^{\mathrm{N}} &= S_{ij}\left(\frac{\partial\triangle U_k}{\partial X_j}\delta\frac{\partial\triangle U_k}{\partial X_i} + \frac{\partial\triangle U_k}{\partial X_i}\delta\frac{\partial\triangle U_k}{\partial X_j}\right),\; i,j=1,2,3,\; i\neq j.\end{aligned}$$

For simplicity reason we define

$$\triangle\mathbf{g}^{\mathrm{N}} = \left(\frac{\partial\triangle U_1}{\partial X_1}, \frac{\partial\triangle U_1}{\partial X_2}, \frac{\partial\triangle U_1}{\partial X_3}, \frac{\partial\triangle U_2}{\partial X_1}, \frac{\partial\triangle U_2}{\partial X_2}, \frac{\partial\triangle U_2}{\partial X_3}, \frac{\partial\triangle U_3}{\partial X_1}, \frac{\partial\triangle U_3}{\partial X_2}, \frac{\partial\triangle U_3}{\partial X_3}\right)^{\mathrm{T}}, \tag{12.28}$$

and the tensor

$$\widetilde{\mathbf{S}} = \begin{pmatrix} S_{11} & S_{12} & S_{31} & & & & & & \\ S_{12} & S_{22} & S_{23} & & 0 & & & 0 & \\ S_{31} & S_{23} & S_{33} & & & & & & \\ & & & S_{11} & S_{12} & S_{31} & & & \\ & 0 & & S_{12} & S_{22} & S_{23} & & 0 & \\ & & & S_{31} & S_{23} & S_{33} & & & \\ & & & & & & S_{11} & S_{12} & S_{31} \\ & 0 & & & 0 & & S_{12} & S_{22} & S_{23} \\ & & & & & & S_{31} & S_{23} & S_{33} \end{pmatrix}_{9\times 9}. \tag{12.29}$$

In this notations the matrix form is as follows

$$\delta U_2 = \int_{V_0} S_{ij}\delta\triangle G^{\mathrm{N}}_{ij}\ \mathrm{d}^0V = \int_{V_0} \left(\delta\triangle\mathbf{g}^{\mathrm{N}}\right)^{\mathrm{T}} \widetilde{\mathbf{S}}\ \triangle\mathbf{g}^{\mathrm{N}}\ \mathrm{d}^0V. \tag{12.30}$$

12.2.3 Transformation of term δU_3

The matrix form of equation (12.24) is as follows

$$\triangle\mathbf{s} = \mathbf{D}\triangle\mathbf{g}^{\mathrm{L}},$$

$$\mathbf{D} = \frac{E}{(1+\sigma)(1-2\sigma)} \begin{pmatrix} 1-\sigma & \sigma & \sigma & & & \\ \sigma & 1-\sigma & \sigma & & 0 & \\ \sigma & \sigma & 1-\sigma & & & \\ & & & \frac{1-2\sigma}{2} & 0 & 0 \\ & 0 & & 0 & \frac{1-2\sigma}{2} & 0 \\ & & & 0 & 0 & \frac{1-2\sigma}{2} \end{pmatrix}_{6\times 6}. \tag{12.31}$$

where E is the Young modulus and σ is the Poisson ratio. Finally

$$\int_{V_0} \triangle S_{ij}\delta\triangle G^{\mathrm{L}}_{ij}\ \mathrm{d}^0V = \int_{V_0} \left(\delta\triangle\mathbf{g}^{\mathrm{L}}\right)^{\mathrm{T}} \triangle\mathbf{s}\ \mathrm{d}^0V = \int_{V_0} \left(\delta\triangle\mathbf{g}^{\mathrm{L}}\right)^{\mathrm{T}} \mathbf{D}\ \triangle\mathbf{g}^{\mathrm{L}}\ \mathrm{d}^0V. \tag{12.32}$$

12.2.4 Transformation to particular element

The principal of virtual work has now the form

$$\int_{V_0} \left(\delta\triangle\mathbf{g}^{\mathrm{L}}\right)^{\mathrm{T}} \mathbf{s}\ \mathrm{d}^0V + \int_{V_0} \left(\delta\triangle\mathbf{g}^{\mathrm{N}}\right)^{\mathrm{T}} \widetilde{\mathbf{S}}\ \triangle\mathbf{g}^{\mathrm{N}}\ \mathrm{d}^0V + \int_{V_0} \left(\delta\triangle\mathbf{g}^{\mathrm{L}}\right)^{\mathrm{T}} \mathbf{D}\ \triangle\mathbf{g}^{\mathrm{L}}\ \mathrm{d}^0V$$
$$= \int_{\partial V_t} \bar{\mathbf{t}}\cdot\delta\mathbf{u}\ \ \mathrm{d}S. \tag{12.33}$$

Now the particular finite element comes into the play. We consider a three-dimensional element with eight vertices and twenty-four unknown displacements (located in vertices)

$$\mathbf{q} = \left(q_1,\dots,q_{24}\right)^{\mathrm{T}}, \tag{12.34}$$

where the first eight components are displacemets along X_1 axis, the second eight components are displacemets along X_2 axis and the last eight components are displacemets

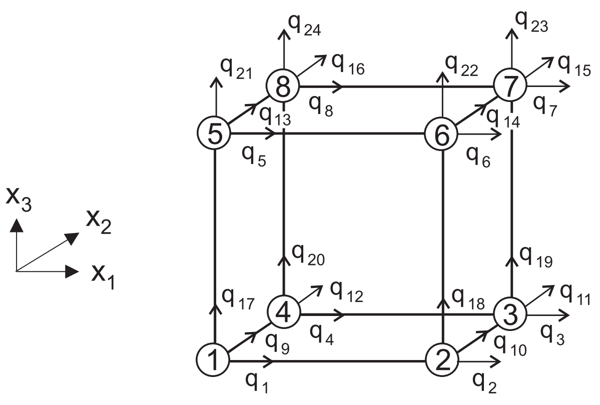


Figure 12.1: Finite element

along X_3 axis. The analogous notation is used for displacement increments between old configuration ${}^{t}C$ and new configuration ${}^{t+\triangle t}C$

$$\triangle \mathbf{q} = \left(\triangle q_1, \ldots, \triangle q_{24}\right)^{\mathrm{T}}. \tag{12.35}$$

Displacement increments $\triangle \mathbf{U}(\mathbf{X}) = (\triangle U_1(\mathbf{X}), \triangle U_2(\mathbf{X}), \triangle U_3(\mathbf{X}))$, $\mathbf{X} = (X_1, X_2, X_3)$, considered as a continuous space functions, are connected to vertex displacement increments $\triangle \mathbf{q}$ by

$$\triangle U_i(\mathbf{X}) = \sum_{s=1}^{8} a_s(\mathbf{X}) \triangle q_{s+8(i-1)}, \quad i = 1, 2, 3, \tag{12.36}$$

where $a_i(\mathbf{X}),\ i = 1, .., 8$ are the element base functions. If we denote element vertices as $\mathrm{P}_i,\ i = 1, .., 8$ then these functions satisfy $a_i\left(\mathrm{P}_j\right) = \delta_{ij};\ i, j = 1, .., 8$. For more information about the element base functions and the connected algebra see section 12.4. By linearization we can get a relation between vectors $\mathbf{g}^{\mathrm{L}}, \mathbf{g}^{\mathrm{N}}$ and vertex increments $\triangle \mathbf{q}$

$$\triangle \mathbf{g}^{\mathrm{L}}(\mathbf{X}) = \mathbf{B}^{\mathrm{L}}(\mathbf{X}) \triangle \mathbf{q}, \tag{12.37}$$

$$\triangle \mathbf{g}^{\mathrm{N}}(\mathbf{X}) = \mathbf{B}^{\mathrm{N}}(\mathbf{X}) \triangle \mathbf{q}. \tag{12.38}$$

Matrix $\mathbf{B}^{\mathrm{L}}$ is of type 6×24, matrix $\mathbf{B}^{\mathrm{N}}$ is of type 9×24. For the exact form of matrices $\mathbf{B}^{\mathrm{L}}$ and $\mathbf{B}^{\mathrm{N}}$ see sections 12.2.5 and 12.2.6. Using formulae (12.37), (12.38) we rewrite integrals as

$$\delta U_1 = (\delta \triangle \mathbf{q})^{\mathrm{T}} \left[\int_{V_0} \left(\mathbf{B}^{\mathrm{L}}\right)^{\mathrm{T}} \mathbf{s} \ \mathrm{d}^0 V \right], \tag{12.39}$$

$$\delta U_2 = (\delta \triangle \mathbf{q})^{\mathrm{T}} \left[\int_{V_0} \left(\mathbf{B}^{\mathrm{N}}\right)^{\mathrm{T}} \widetilde{\mathbf{S}}\ \mathbf{B}^{\mathrm{N}} \ \mathrm{d}^0 V \right] \triangle \mathbf{q}, \tag{12.40}$$

$$\delta U_3 = (\delta \triangle \mathbf{q})^{\mathrm{T}} \left[\int_{V_0} \left(\mathbf{B}^{\mathrm{L}}\right)^{\mathrm{T}} \mathbf{D}\ \mathbf{B}^{\mathrm{L}} \ \mathrm{d}^0 V \right] \triangle \mathbf{q}. \tag{12.41}$$

The integral in formula (12.39) yields to 24×1 type of vector, the integrals in formulae (12.40) and (12.41) yield to 24×24 type of matrix. The right hand side of equation (12.33), representing the virtual work of external pressure forces, can be approximated by the sum of work in all nodes

$$\int_{\partial V_t} \bar{\mathbf{t}} \cdot \delta \mathbf{u} \ \ \mathrm{d}S \approx (\delta \triangle \mathbf{q})^{\mathrm{T}} \cdot \mathbf{r}(\triangle \mathbf{q}). \tag{12.42}$$

For example in our test example 1 (see section 12.3.3) the approximative force $\mathbf{r}(\triangle\mathbf{q})$ is supposed to be non-zero only on the upper surface of the loaded beam (in the direction of x_3 axis) [5]

$$\mathbf{r} = \left(0,0,0,0,\frac{F_1}{4},\frac{F_1}{4},\frac{F_1}{4},\frac{F_1}{4},0,0,0,0,\frac{F_2}{4},\frac{F_2}{4},\frac{F_2}{4},\frac{F_2}{4},0,0,0,0,\frac{F_3}{4},\frac{F_3}{4},\frac{F_3}{4},\frac{F_3}{4}\right),$$
$$\mathbf{F} = -p \cdot (\mathbf{l}_1 \times \mathbf{l}_2)\,.$$

Finally using (12.39)-(12.42), after canceling out term $(\delta\triangle\mathbf{q})^{\mathrm{T}}$, we get the system of twenty-four equations (for each element)

$$\left[\int_{V_0} \left(\mathbf{B}^{\mathrm{L}}\right)^{\mathrm{T}} \mathbf{D}\,\mathbf{B}^{\mathrm{L}}\,\mathrm{d}^0V + \int_{V_0} \left(\mathbf{B}^{\mathrm{N}}\right)^{\mathrm{T}} \widetilde{\mathbf{S}}\,\mathbf{B}^{\mathrm{N}}\,\mathrm{d}^0V\right] \triangle\mathbf{q} = -\int_{V_0} \left(\mathbf{B}^{\mathrm{L}}\right)^{\mathrm{T}} \mathbf{s}\,\mathrm{d}^0V + \mathbf{r}(\triangle\mathbf{q}), \tag{12.43}$$

which can be written in the simpler form

$$\mathbf{K}^{local}(\triangle\mathbf{q})\triangle\mathbf{q} = \mathbf{f}^{local}(\triangle\mathbf{q}). \tag{12.44}$$

Creating such a local system for each element, we create a global system of equations

$$\mathbf{K}(\triangle\mathbf{q})\triangle\mathbf{q} = \mathbf{f}(\triangle\mathbf{q}). \tag{12.45}$$

System of equations (12.45) is computed iteratively as a sequence of linear problems until the right hand side comes to zero in an appropriate norm. Then solution additions $\triangle\mathbf{q}$ also comes to zero.

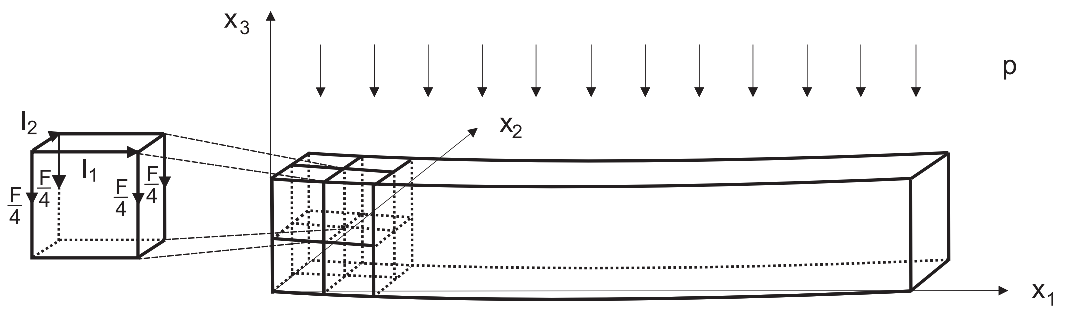


Figure 12.2: Deformed grid

12.2.5 Components of matrix $\mathbf{B}^{\mathrm{L}}$

We have

$$\triangle G^{\mathrm{L}}_{ij} = \frac{1}{2}\left[\frac{\partial\triangle U_i}{\partial X_j} + \frac{\partial\triangle U_j}{\partial X_i} + \frac{\partial\triangle U_k}{\partial X_i} Z_{kj} + Z_{ki}\frac{\partial\triangle U_k}{\partial X_j}\right], \quad i,j = 1,2,3,$$

[5] $\mathbf{l}_1 = (q_6 + \triangle q_6,\ q_{14} + \triangle q_{14},\ q_{22} + \triangle q_{22}) - (q_5 - \triangle q_5,\ q_{13} + \triangle q_{13},\ q_{21} + \triangle q_{21})\,,$
$\mathbf{l}_2 = (q_8 + \triangle q_8,\ q_{16} + \triangle q_{16},\ q_{24} + \triangle q_{24}) - (q_5 - \triangle q_5,\ q_{13} + \triangle q_{13},\ q_{21} + \triangle q_{21})\,.$

where we denoted $Z_{ij} = \frac{\partial U_i}{\partial X_j}; \quad i,j = 1,2,3$, which is known from the previous iteration. Let us compute some terms with help of equation (12.36) (other terms are computed analogously)

$$\triangle G^{\mathrm{L}}_{11} = \frac{\partial \triangle U_1}{\partial X_1} + Z_{k1}\frac{\partial \triangle U_k}{\partial X_1} = (1+Z_{11})\frac{\partial a_s}{\partial X_1}\triangle q_s + Z_{21}\frac{\partial a_s}{\partial X_1}\triangle q_{s+8} + Z_{31}\frac{\partial a_s}{\partial X_1}\triangle q_{s+16}$$

$$2\triangle G^{\mathrm{L}}_{12} = \frac{\partial \triangle U_1}{\partial X_2} + \frac{\partial \triangle U_2}{\partial X_1} + Z_{k2}\frac{\partial \triangle U_k}{\partial X_1} + Z_{k1}\frac{\partial \triangle U_k}{\partial X_2} = \left[Z_{12}\frac{\partial a_s}{\partial X_1} + (1+Z_{11})\frac{\partial a_s}{\partial X_2}\right]\triangle q_s + \left[(1+Z_{22})\frac{\partial a_s}{\partial X_1} + Z_{21}\frac{\partial a_s}{\partial X_2}\right]\triangle q_{s+8} + \left[Z_{32}\frac{\partial a_s}{\partial X_1} + Z_{31}\frac{\partial a_s}{\partial X_2}\right]\triangle q_{s+16}$$

Summing over s and k is supposed. The matrix $\mathbf{B}^{\mathrm{L}}$ then has the form

$$\mathbf{B}^{\mathrm{L}} = \begin{pmatrix}
\underset{s=1,\dots,8}{(1+Z_{11})\frac{\partial a_s}{\partial X_1}} & \underset{s=1,\dots,8}{Z_{21}\frac{\partial a_s}{\partial X_1}} & \underset{s=1,\dots,8}{Z_{31}\frac{\partial a_s}{\partial X_1}} \\
\underset{s=1,\dots,8}{Z_{12}\frac{\partial a_s}{\partial X_2}} & \underset{s=1,\dots,8}{(1+Z_{22})\frac{\partial a_s}{\partial X_2}} & \underset{s=1,\dots,8}{Z_{32}\frac{\partial a_s}{\partial X_2}} \\
\underset{s=1,\dots,8}{Z_{13}\frac{\partial a_s}{\partial X_3}} & \underset{s=1,\dots,8}{Z_{23}\frac{\partial a_s}{\partial X_3}} & \underset{s=1,\dots,8}{(1+Z_{33})\frac{\partial a_s}{\partial X_3}} \\
\underset{s=1,\dots,8}{Z_{12}\frac{\partial a_s}{\partial X_1} + (1+Z_{11})\frac{\partial a_s}{\partial X_2}} & \underset{s=1,\dots,8}{(1+Z_{22})\frac{\partial a_s}{\partial X_1} + Z_{21}\frac{\partial a_s}{\partial X_2}} & \underset{s=1,\dots,8}{Z_{32}\frac{\partial a_s}{\partial X_1} + Z_{31}\frac{\partial a_s}{\partial X_2}} \\
\underset{s=1,\dots,8}{Z_{13}\frac{\partial a_s}{\partial X_2} + Z_{12}\frac{\partial a_s}{\partial X_3}} & \underset{s=1,\dots,8}{Z_{23}\frac{\partial a_s}{\partial X_2} + (1+Z_{22})\frac{\partial a_s}{\partial X_3}} & \underset{s=1,\dots,8}{(1+Z_{33})\frac{\partial a_s}{\partial X_2} + Z_{32}\frac{\partial a_s}{\partial X_3}} \\
\underset{s=1,\dots,8}{Z_{13}\frac{\partial a_s}{\partial X_1} + (1+Z_{11})\frac{\partial a_s}{\partial X_3}} & \underset{s=1,\dots,8}{Z_{23}\frac{\partial a_s}{\partial X_1} + Z_{21}\frac{\partial a_s}{\partial X_3}} & \underset{s=1,\dots,8}{(1+Z_{33})\frac{\partial a_s}{\partial X_1} + Z_{31}\frac{\partial a_s}{\partial X_3}}
\end{pmatrix}_{6\times 24}. \tag{12.46}$$

12.2.6 Components of matrix $\mathbf{B}^{\mathrm{N}}$

From (12.28), (12.37) we immediately get

$$\mathbf{B}^{\mathrm{N}} = \begin{pmatrix}
\frac{\partial a_1}{\partial X_1} \cdots \frac{\partial a_8}{\partial X_1} & 0 & 0 \\
\frac{\partial a_1}{\partial X_2} \cdots \frac{\partial a_8}{\partial X_2} & 0 & 0 \\
\frac{\partial a_1}{\partial X_3} \cdots \frac{\partial a_8}{\partial X_3} & 0 & 0 \\
0 & \frac{\partial a_1}{\partial X_1} \cdots \frac{\partial a_8}{\partial X_1} & 0 \\
0 & \frac{\partial a_1}{\partial X_2} \cdots \frac{\partial a_8}{\partial X_2} & 0 \\
0 & \frac{\partial a_1}{\partial X_3} \cdots \frac{\partial a_8}{\partial X_3} & 0 \\
0 & 0 & \frac{\partial a_1}{\partial X_1} \cdots \frac{\partial a_8}{\partial X_1} \\
0 & 0 & \frac{\partial a_1}{\partial X_2} \cdots \frac{\partial a_8}{\partial X_2} \\
0 & 0 & \frac{\partial a_1}{\partial X_3} \cdots \frac{\partial a_8}{\partial X_3}
\end{pmatrix}_{9\times 24}. \tag{12.47}$$

Term $\left(\mathbf{B}^{\mathrm{N}}\right)^{\mathrm{T}} \widetilde{\mathbf{S}}\, \mathbf{B}^{\mathrm{N}}$ has the form

$$\left(\mathbf{B}^{\mathrm{N}}\right)^{\mathrm{T}} \widetilde{\mathbf{S}}\, \mathbf{B}^{\mathrm{N}} = \begin{pmatrix} \mathbf{A} & & \\ & \mathbf{A} & \\ & & \mathbf{A} \end{pmatrix}_{24\times 24}, \quad \mathbf{A}_{ij} = S_{kl} \frac{\partial a_i}{\partial X_k} \frac{\partial a_j}{\partial X_l}, \quad i,j = 1,..,8. \tag{12.48}$$

12.2.7 Small deformations

Small deformations means that

$$\begin{aligned} \triangle G^{\mathrm{L}}_{ij} &= \frac{1}{2}\left[\frac{\partial \triangle U_i}{\partial X_j} + \frac{\partial \triangle U_j}{\partial X_i}\right], \\ \triangle G^{\mathrm{N}}_{ij} &= 0. \end{aligned} \tag{12.49}$$

Integrand $\int_{V_0} \left(\mathbf{B}^{\mathrm{N}}\right)^{\mathrm{T}} \widetilde{\mathbf{S}}\, \mathbf{B}^{\mathrm{N}}\, \mathrm{d}^0 V$ diminishes. The matrix $\mathbf{B}^{\mathrm{L}}$ has the form

$$\mathbf{B}^{\mathrm{L}} = \begin{pmatrix} \frac{\partial a_1}{\partial X_1} \cdots \frac{\partial a_8}{\partial X_1} & 0 & 0 \\ 0 & \frac{\partial a_1}{\partial X_2} \cdots \frac{\partial a_8}{\partial X_2} & 0 \\ 0 & 0 & \frac{\partial a_1}{\partial X_3} \cdots \frac{\partial a_8}{\partial X_3} \\ \frac{\partial a_1}{\partial X_2} \cdots \frac{\partial a_8}{\partial X_2} & \frac{\partial a_1}{\partial X_1} \cdots \frac{\partial a_8}{\partial X_1} & 0 \\ 0 & \frac{\partial a_1}{\partial X_3} \cdots \frac{\partial a_8}{\partial X_3} & \frac{\partial a_1}{\partial X_2} \cdots \frac{\partial a_8}{\partial X_2} \\ \frac{\partial a_1}{\partial X_3} \cdots \frac{\partial a_8}{\partial X_3} & 0 & \frac{\partial a_1}{\partial X_1} \cdots \frac{\partial a_8}{\partial X_1} \end{pmatrix}_{6\times 24}.$$

and because it is a constant matrix it can be precomputed. The resulting problem is of course linear.

12.3 Numerics

12.3.1 Numerical integration

The integrals in equation (12.43) need to be evaluated numerically. First of all we map the undeformed element to the reference element $< -1, 1 > \times < -1, 1 > \times < -1, 1 >$ by

$$\Psi : \quad \mathbb{R}_3 \to \mathbb{R}_3, \quad (r,s,t) \to (X_1(r,s,t), X_2(r,s,t), X_3(r,s,t)). \tag{12.50}$$

Let us denote the Jacobian matrix of mapping Ψ as $\mathbf{J}$. For derivation of this mapping see section 12.4. The integration of all integrals is made by means of the n-point Gauss quadrature ($n = 2, 3$). The Gauss quadrature is often used in one dimension according to formula $\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{i=1}^{n} w_i f(x_i)$, where weights w_i and points x_i are given in the following table

Number of points	Points	Weights
2	$x_1 = -\frac{1}{\sqrt{3}},\ x_2 = +\frac{1}{\sqrt{3}}$	$w_1 = 1, w_2 = 1$
3	$x_1 = -\sqrt{\frac{3}{5}},\ x_2 = 0,\ x_3 = +\sqrt{\frac{3}{5}}$	$w_1 = \frac{5}{9},\ w_2 = \frac{8}{9},\ w_3 = \frac{5}{9}$

The n-point Gauss quadrature gives an exact result for polynomials of degree $2n - 1$ or less. In three dimensions we extend results from one dimension—we get n^3 points given by $\mathrm{P}_{ijk} = [x_i,\ x_j,\ x_k]$ with weights $w_{ijk} = w_i w_j w_k;\ i,j,k = 1,\ldots,n$. Integration then equals to

$$\int_{\mathrm{V}_0} f(\mathbf{X})\ \mathrm{d}^0 V = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f(r,s,t)\ \det\mathbf{J}(r,s,t)\ \mathrm{d}r\ \mathrm{d}s\ \mathrm{d}t \doteq \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} w_{ijk} f(\mathrm{P}_{ijk})\ \det\mathbf{J}(\mathrm{P}_{ijk}).$$

In practical computations we used the two point Gauss quadrature because the difference to the three point Gauss quadrature was negligible.

12.3.2 Iterative procedure

The main computational algorithm consists of these steps:

1. Fill matrix $\mathbf{D}$ according to (12.31).
2. For all Gauss points within each element compute $\frac{\partial a_i}{\partial X_j},\ i = 1,\ldots,8,\ j = 1,2,3,$ and $\det\mathbf{J}$.
3. Set $n = 0$ (iteration number).
4. Null the global matrix and the global right hand side (RHS). Increment n by one.
5. For all elements do the following:
   (a) Null the local matrix and the local RHS.
   (b) If $n = 1$ then set $\mathbf{q}^n = 0$ else set $\mathbf{q}^n = \mathbf{q}^{n-1} + \triangle\mathbf{q}^{n-1}$.
   (c) Update boundary pressure forces $\mathbf{r}^n(\triangle\mathbf{q}^n)$.
   (d) At each Gauss point within the element evaluate the following expressions:
      i. If $n = 1$ then set $\mathbf{s}^n = 0$ else compute $\mathbf{s}^n = \mathbf{s}^{n-1} + \triangle\mathbf{s}$, $\triangle\mathbf{s} = \mathbf{D}\triangle\mathbf{g}^{\mathrm{L}} = \mathbf{D}\mathbf{B}^{\mathrm{L}}\triangle\mathbf{q}^n$.
      ii. Compute tensor $\mathbf{Z}$: $\mathbf{Z}_{ij} = \frac{\partial U_i}{\partial X_j} = \frac{\partial a_s}{\partial x_j} q^n_{s+8(i-1)},\quad i,j = 1,2,3.$
      iii. Compute tensor $\mathbf{B}^{\mathrm{L}}(\mathbf{Z})$ using (12.46).
      iv. Update the local matrix by products $\left(\mathbf{B}^{\mathrm{L}}\right)^{\mathrm{T}}\mathbf{D}\mathbf{B}^{\mathrm{L}}$.
      v. Update the local matrix by products $\left(\mathbf{B}^{\mathrm{N}}\right)^{\mathrm{T}}\widetilde{\mathbf{S}}\mathbf{B}^{\mathrm{N}}$ according to (12.48).
      vi. Update the local RHS by product $\left(\mathbf{B}^{\mathrm{L}}\right)^{\mathrm{T}}\mathbf{s}^n$.
   (e) Copy the local matrix and the local RHS to the global matrix and the global RHS.
6. Invert the global matrix in order to obtain new solution increment $\triangle\mathbf{q}^n$.
7. If $n = 1$ go to step 4.
8. If $||\triangle\mathbf{q}^n||_{\max} < \varepsilon$ then stop, otherwise go to step 4. (Ussually $\varepsilon = 10^{-9}$.)

In the programming code step 2. is realized by method `Element.Initialize()` and step 5. by method `Element.Do()`. For more details see source code on page 278.

12.3.3 Test example 1 - loaded beam

As a test example we consider a pressure loaded beam which is fully fixed at $x_1 = 0$ and $x_1 = l$, see Figure 12.3. The pressure above the beam is $p = 50$ Pa. Other parameters are

$$l = 0.11 \text{ m},\ z_0 = 0.00275\ m,\ E = 10\ MPa,\ \sigma = 0.25,$$

where l is the beam length, z_0 is the beam width and depth, E is the Young modulus and σ is the Poisson ratio. The example is solvable from linear theory using the Euler-Bernoulli beam equation which gives the result

$$u_{middle} = \frac{1}{384}\frac{ql^4}{EJ}, \tag{12.51}$$

where q is the load density given by $q = pz_0$ and J is the second moment of area given by $J = \frac{z_o^4}{12}$. Finally we have

$$u_{middle} = \frac{1}{32}\frac{pz_0}{E}\left(\frac{l}{z_0}\right)^4 = 0.0011\ m. \tag{12.52}$$

The results were also compared with the results computed by ANSYS v.11 software package. The own code is called FELIB (see Section 12.4). The following table shows the numerical results

Program	Maximal displacement u_{middle}	Relative error comparing to (12.52)	Number of elements	Note
ANSYS	0.001104 m	0.36 %	$200 \times 5 \times 5$	Elements: SOLID45
ANSYS	0.001104 m	0.36 %	$320 \times 8 \times 8$	Elements: SOLID45
FELIB	0.001079 m	1.9 %	$200 \times 5 \times 5$	-
FELIB	0.001093 m	0.64 %	$320 \times 8 \times 8$	-
FELIB	0.001096 m	0.36 %	$800 \times 5 \times 5$	-

We verified that for the increasing number of elements the results given by FELIB converge to the value (12.52).

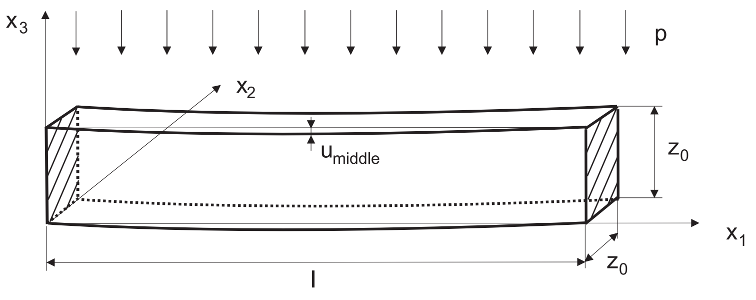


Figure 12.3: Pressure loaded beam, fully fixed at $x_1 = 0$ and $x_1 = l$.

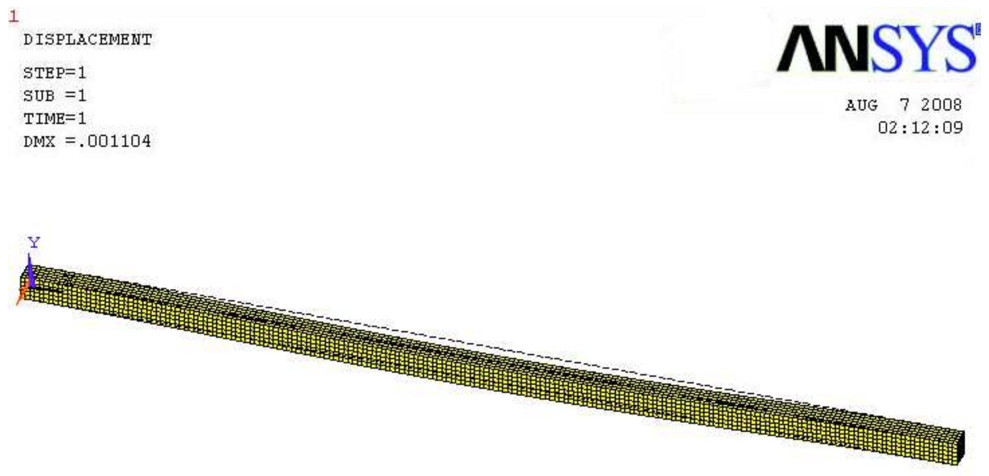


Figure 12.4: Test example 1 in ANSYS v.11

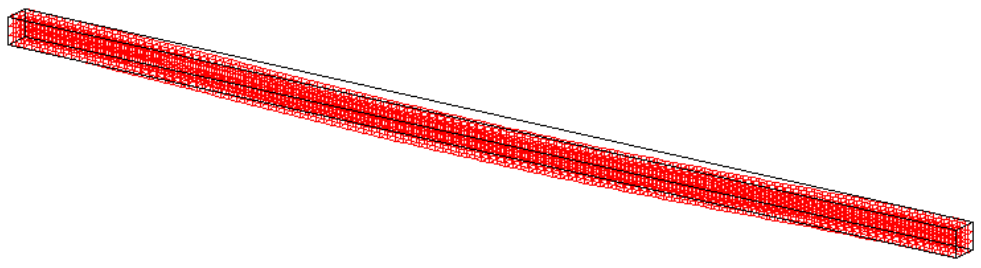


Figure 12.5: Test example 1 in FELIB.

12.3.4 Test example 2 - pressed beam

As another test example we consider a beam pressed by pressure p against the wall. We consider two cases: small deformations (loading pressure $p = 10^4$ Pa) and finite deformations (loading pressure $p = 10^6$ Pa). The beam has length $l_0 = 0.11\ m$, width and depth $z_0 = 0.00275\ m$ and material constants $E = 10\ MPa,\ \sigma = 0.25$.
The loading pressure acts in the direction opposite to axis x_1. The next table shows the

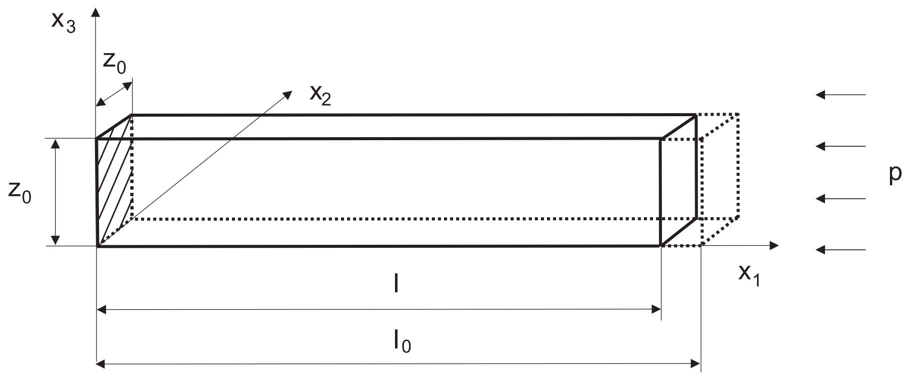


Figure 12.6: Beam pressed in direction opsoite to axis x_1.

small deformation case where pressure was $p = 10^4$ Pa. In this case the Hooke law gives relative displacement $\varepsilon = \frac{\triangle l}{l_0} = \frac{l_0 - l}{l_0} = \frac{p}{E} = 0.001$. The table shows not only convergence but also a perfect coincidence with the linear theory.

Relative displacement ε $[-]$	Difference to the previous line multiplied by $\times\ 10^9$	Number of elements	Number of unknowns
0.00100 0028	-	$40 \times 1 \times 1$	480
0.00100 0605	577	$80 \times 2 \times 2$	2160
0.00100 0768	163	$120 \times 3 \times 3$	5760
0.00100 0842	74	$160 \times 4 \times 4$	12000
0.00100 0883	41	$200 \times 5 \times 5$	21600
0.00100 0909	26	$240 \times 6 \times 6$	35280

The next table shows the finite deformation case where pressure was $p = 10^6$ Pa. In this case the Hooke law gives relative displacement $\varepsilon = \frac{\triangle l}{l_0} = \frac{p}{E} = 0.1$. We observe difference based on the fact that the Hooke law is valid only for small deformations.

Relative displacement ε $[-]$	Difference to the previous line multiplied by $\times\ 10^7$	Number of elements	Number of unknowns
0.123 2484	-	$40 \times 1 \times 1$	480
0.123 3394	910	$80 \times 2 \times 2$	2160
0.123 3662	268	$120 \times 3 \times 3$	5760
0.123 3785	123	$160 \times 4 \times 4$	12000
0.123 3855	70	$200 \times 5 \times 5$	21600
0.123 3899	4	$240 \times 6 \times 6$	35280

12.4 Integration on reference element

We consider a reference element $< -1, 1 > \times < -1, 1 > \times < -1, 1 >$ where we integrate all integrands. We define mapping

$$\Psi : \quad \mathbb{R}_3 \rightarrow \mathbb{R}_3, \quad (r, s, t) \rightarrow (X_1(r, s, t), X_2(r, s, t), X_3(r, s, t)), \tag{12.53}$$

which mapps the reference element to the particular undeformed element. We force the mapping to be a linear combination of eight polynomials (1, r, s, t, rs, st, tr, rst)

$$\begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} \underbrace{1\ r\ s\ t\ rs\ st\ tr\ rst}_{\Phi} & & \\ & 1\ r\ s\ t\ rs\ st\ tr\ rst & \\ & & 1\ r\ s\ t\ rs\ st\ tr\ rst \end{pmatrix} \begin{pmatrix} c_1 \\ \vdots \\ c_8 \\ c_9 \\ \vdots \\ c_{16} \\ c_{17} \\ \vdots \\ c_{24} \end{pmatrix}. \tag{12.54}$$

Let us denote vertexes of the undeformed element as $\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_8$. Let us define the vector of vertex-coordinates as follows

$$\mathbf{p} = \Big(\underbrace{p_1,\ \ldots,\ p_8}_{X_1-components},\ \underbrace{p_9,\ \ldots,\ p_{16}}_{X_2-components},\ \underbrace{p_{17},\ \ldots,\ p_{24}}_{X_3-components} \Big)^{\mathrm{T}}. \tag{12.55}$$

i.e. $P_1 = [p_1, p_9, p_{17}]$, $P_2 = [p_2, p_{10}, p_{18}]$ etc. We relate, using equation (12.54), all eight vertexes to the vertexes of the reference element which allows us to compute constants $c_1, \ldots, c_{24}$

$$\begin{pmatrix} p_1 \\ \vdots \\ p_8 \\ p_9 \\ \vdots \\ p_{16} \\ p_{17} \\ \vdots \\ p_{24} \end{pmatrix} = \begin{pmatrix} \mathbf{S} & & \\ & \mathbf{S} & \\ & & \mathbf{S} \end{pmatrix} \begin{pmatrix} c_1 \\ \vdots \\ c_8 \\ c_9 \\ \vdots \\ c_{16} \\ c_{17} \\ \vdots \\ c_{24} \end{pmatrix}, \ \mathbf{S} = \begin{pmatrix} 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 \end{pmatrix}.$$

By inversion we get

$$\begin{pmatrix} c_1 \\ \vdots \\ c_8 \\ c_9 \\ \vdots \\ c_{16} \\ c_{17} \\ \vdots \\ c_{24} \end{pmatrix} = \begin{pmatrix} \mathbf{S}^{-1} & & \\ & \mathbf{S}^{-1} & \\ & & \mathbf{S}^{-1} \end{pmatrix} \begin{pmatrix} p_1 \\ \vdots \\ p_8 \\ p_9 \\ \vdots \\ p_{16} \\ p_{17} \\ \vdots \\ p_{24} \end{pmatrix}, \mathbf{S}^{-1} = \frac{1}{8} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \\ -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 \end{pmatrix}.$$

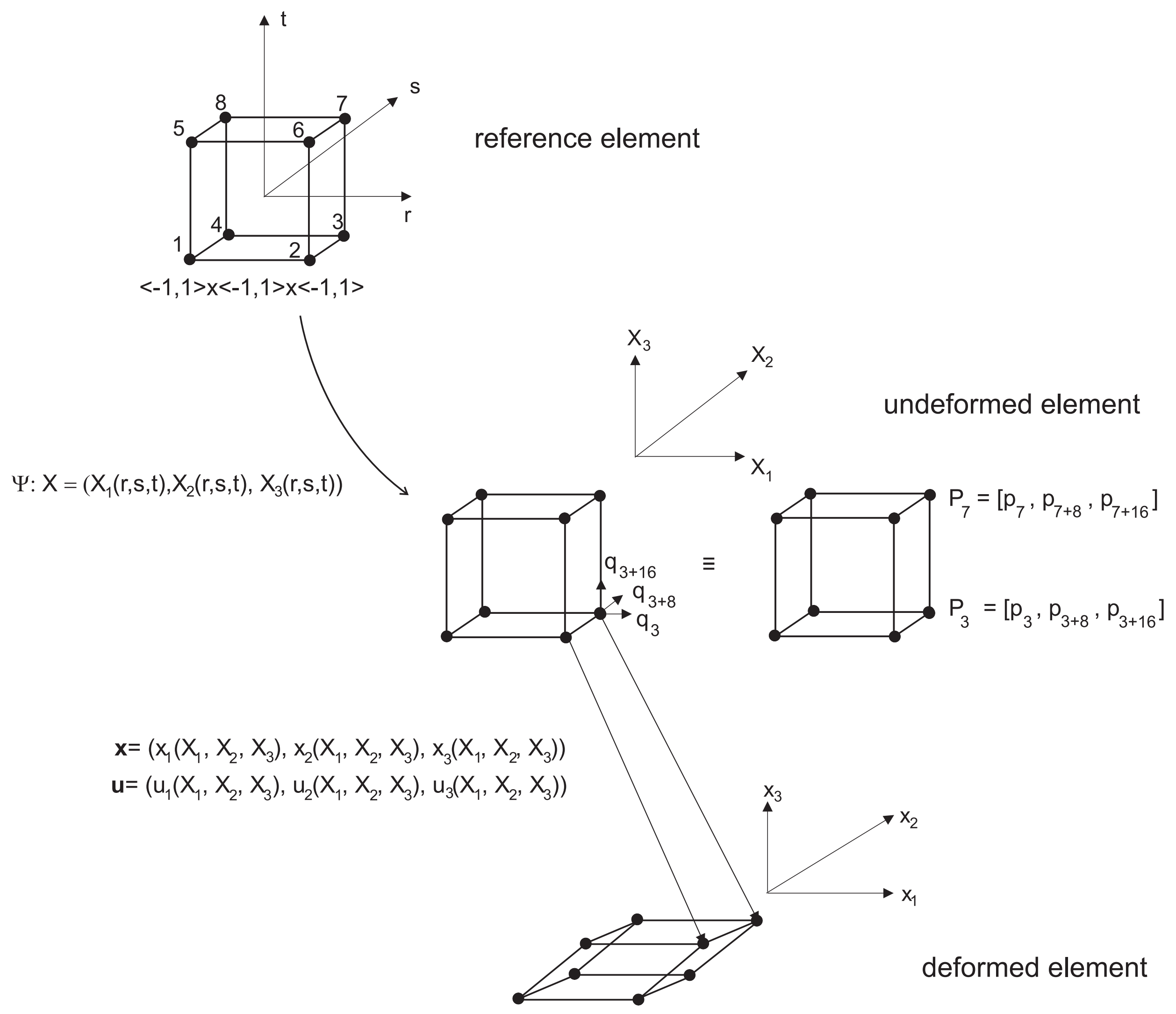


Figure 12.7: The reference, undeformed and deformed element.

Equations (12.54) can be now rewritten as

$$\begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} \Phi & & \\ & \Phi & \\ & & \Phi \end{pmatrix} \begin{pmatrix} \mathbf{S}^{-1} & & \\ & \mathbf{S}^{-1} & \\ & & \mathbf{S}^{-1} \end{pmatrix} \begin{pmatrix} p_1 \\ \vdots \\ p_8 \\ p_9 \\ \vdots \\ p_{16} \\ p_{17} \\ \vdots \\ p_{24} \end{pmatrix} =$$

$$= \begin{pmatrix} a_1(r,s,t),\ldots,a_8(r,s,t) & & \\ & a_1(r,s,t),\ldots,a_8(r,s,t) & \\ & & a_1(r,s,t),\ldots,a_8(r,s,t) \end{pmatrix} \begin{pmatrix} p_1 \\ \vdots \\ p_8 \\ p_9 \\ \vdots \\ p_{16} \\ p_{17} \\ \vdots \\ p_{24} \end{pmatrix},$$

$$\begin{aligned} a_1 &= \frac{1}{8}(1-r)(1-s)(1-t), & a_5 &= \frac{1}{8}(1-r)(1-s)(1+t), \\ a_2 &= \frac{1}{8}(1+r)(1-s)(1-t), & a_6 &= \frac{1}{8}(1+r)(1-s)(1+t), \\ a_3 &= \frac{1}{8}(1+r)(1+s)(1-t), & a_7 &= \frac{1}{8}(1+r)(1+s)(1+t), \\ a_4 &= \frac{1}{8}(1-r)(1+s)(1-t), & a_8 &= \frac{1}{8}(1-r)(1+s)(1+t), \end{aligned}$$

or in the non-vector form

$$X_1 = \sum_{i=1}^{8} a_i(r,s,t)p_i,\ X_2 = \sum_{i=1}^{8} a_i(r,s,t)p_{i+8},\ X_3 = \sum_{i=1}^{8} a_i(r,s,t)p_{i+16}. \tag{12.56}$$

The displacements are defined in vertexes of the undeformed element. Let us define vector of all displacement components

$$\mathbf{q} = \Big(\underbrace{q_1,\ \ldots,\ q_8}_{X_1-components},\ \underbrace{q_9,\ \ldots,\ q_{16}}_{X_2-components},\ \underbrace{q_{17},\ \ldots,\ q_{24}}_{X_3-components} \Big)^{\mathrm{T}}. \tag{12.57}$$

The displacement $\mathbf{u} = (u_1, u_2, u_3)(r,s,t)$, considered as a continuous function, is approximated analogously to formula (12.56) (the element is isoparametric)

$$u_1 = \sum_{i=1}^{8} a_i(r,s,t)q_i,\ u_2 = \sum_{i=1}^{8} a_i(r,s,t)q_{i+8},\ u_3 = \sum_{i=1}^{8} a_i(r,s,t)q_{i+16}. \tag{12.58}$$

Note that in the finite element method the equation (12.58) is often used for displacement increments instead of displacements, i.e.

$$\triangle u_1 = \sum_{i=1}^{8} a_i(r,s,t)\triangle q_i,\ \triangle u_2 = \sum_{i=1}^{8} a_i(r,s,t)\triangle q_{i+8},\ \triangle u_3 = \sum_{i=1}^{8} a_i(r,s,t)\triangle q_{i+16}. \tag{12.59}$$

We can easily compute the Jacobian determinant of mapping Ψ

$$\det\mathbf{J} = \begin{vmatrix} \frac{\partial X_1}{\partial r} & \frac{\partial X_2}{\partial r} & \frac{\partial X_3}{\partial r} \\ \frac{\partial X_1}{\partial s} & \frac{\partial X_2}{\partial s} & \frac{\partial X_3}{\partial s} \\ \frac{\partial X_1}{\partial t} & \frac{\partial X_2}{\partial t} & \frac{\partial X_3}{\partial t} \end{vmatrix} = \begin{vmatrix} \sum_{i=1}^{8} \frac{\partial a_i}{\partial r} p_i & \sum_{i=1}^{8} \frac{\partial a_i}{\partial r} p_{i+8} & \sum_{i=1}^{8} \frac{\partial a_i}{\partial r} p_{i+16} \\ \sum_{i=1}^{8} \frac{\partial a_i}{\partial s} p_i & \sum_{i=1}^{8} \frac{\partial a_i}{\partial s} p_{i+8} & \sum_{i=1}^{8} \frac{\partial a_i}{\partial s} p_{i+16} \\ \sum_{i=1}^{8} \frac{\partial a_i}{\partial t} p_i & \sum_{i=1}^{8} \frac{\partial a_i}{\partial t} p_{i+8} & \sum_{i=1}^{8} \frac{\partial a_i}{\partial t} p_{i+16} \end{vmatrix}. \tag{12.60}$$

In order to compute values of $\frac{\partial a_i}{\partial X_j}$ $i = 1, \ldots, 8,\ j = 1, \ldots, 3$ we obtain relation

$$\begin{pmatrix} \frac{\partial a_i}{\partial r} \\ \frac{\partial a_i}{\partial s} \\ \frac{\partial a_i}{\partial t} \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{\partial X_1}{\partial r} & \frac{\partial X_2}{\partial r} & \frac{\partial X_3}{\partial r} \\ \frac{\partial X_1}{\partial s} & \frac{\partial X_2}{\partial s} & \frac{\partial X_3}{\partial s} \\ \frac{\partial X_1}{\partial t} & \frac{\partial X_2}{\partial t} & \frac{\partial X_3}{\partial t} \end{pmatrix}}_{\mathbf{J}} \begin{pmatrix} \frac{\partial a_i}{\partial X_1} \\ \frac{\partial a_i}{\partial X_2} \\ \frac{\partial a_i}{\partial X_3} \end{pmatrix} \quad \Longrightarrow \quad \begin{pmatrix} \frac{\partial a_i}{\partial X_1} \\ \frac{\partial a_i}{\partial X_2} \\ \frac{\partial a_i}{\partial X_3} \end{pmatrix} = \mathbf{J}^{-1} \begin{pmatrix} \frac{\partial a_i}{\partial r} \\ \frac{\partial a_i}{\partial s} \\ \frac{\partial a_i}{\partial t} \end{pmatrix}. \tag{12.61}$$

where derivatives $\frac{\partial a_i}{\partial r}, \frac{\partial a_i}{\partial s}, \frac{\partial a_i}{\partial t}$ $i = 1, \ldots, 8,\ j = 1, \ldots, 3$ are given by

	$\frac{\partial a_i}{\partial r}$	$\frac{\partial a_i}{\partial s}$	$\frac{\partial a_i}{\partial t}$
$i = 1$	$-\frac{1}{8}(1-s)(1-t)$	$-\frac{1}{8}(1-r)(1-t)$	$-\frac{1}{8}(1-r)(1-s)$
$i = 2$	$\frac{1}{8}(1-s)(1-t)$	$-\frac{1}{8}(1+r)(1-t)$	$-\frac{1}{8}(1+r)(1-s)$
$i = 3$	$\frac{1}{8}(1+s)(1-t)$	$\frac{1}{8}(1+r)(1-t)$	$-\frac{1}{8}(1+r)(1+s)$
$i = 4$	$-\frac{1}{8}(1+s)(1-t)$	$\frac{1}{8}(1-r)(1-t)$	$-\frac{1}{8}(1-r)(1+s)$
$i = 5$	$-\frac{1}{8}(1-s)(1+t)$	$-\frac{1}{8}(1-r)(1+t)$	$\frac{1}{8}(1-r)(1-s)$
$i = 6$	$\frac{1}{8}(1-s)(1+t)$	$-\frac{1}{8}(1+r)(1+t)$	$\frac{1}{8}(1+r)(1-s)$
$i = 7$	$\frac{1}{8}(1+s)(1+t)$	$\frac{1}{8}(1+r)(1+t)$	$\frac{1}{8}(1+r)(1+s)$
$i = 8$	$-\frac{1}{8}(1+s)(1+t)$	$\frac{1}{8}(1-r)(1+t)$	$\frac{1}{8}(1-r)(1+s)$

Bibliography

[1] H.A. Gerhard. *Nonlinear Solid Mechanics – A continuum approach for engineering.* John Wiley & Sons Ltd, Chichester England, 2001.

[2] M. Okrouhlík and S. Pták. *Počítačová Mechanika I – Základy nelineární mechaniky kontinua.* ČVUT, Prague, 2006.

Appendix - Source code

The program is written in C# language and has two parts: the numerical library providing a basic linear algebra (MATHLIB) and the finite element library (FELIB).

12.4.1 MATHLIB

First of all we represent the MATHLIB library. It contains the following structures and classes:

- `Vector` (class representing a real vector)
- `ThreeVectorStruct` (structure representing vector with three real components; more efficient then Vector.)
- `ThreeLongVectorStruct` (structure representing vector with three integer components)
- `FullMatrix` (class representing a non-sparse matrix)
- `LinearSystemSolver` (class representing a linear system which includes the sparse matrix, the right hand side vector and the solution vector)

```
//***************************************************************
// Vector - represents vector
//***************************************************************
public class Vector
{
    long n;
    double[] p;
    public long N { get { return n; } }

    public Vector(long n)
    {
        p = new double[this.n = n];
    }
    public Vector(double x1, double x2, double x3)
    {
        p = new double[this.n = 3];
        p[0] = x1;
        p[1] = x2;
        p[2] = x3;
    }
    public Vector(Vector x)
    {
        p = new double[n = x.N];
        for (int i = 0; i <= n - 1; i++) p[i] = x.p[i];
    }
    public double this[long i]
    {
        get
        {
            if ((i < 1) || (i > n))
            {
                throw new Exception("Vector.operator[i]: (i < 1) || (i > n)");
            }
            return p[i - 1];
        }
        set
        {
            if ((i < 1) || (i > n))
            {
                throw new Exception("Vector.operator[i]: (i < 1) || (i > n)");
            }
            p[i - 1] = value;
        }
    }
    public static Vector operator +(Vector l, Vector r)
    {
        if (l.n != r.n) throw new Exception("Vector operator + : l.n != r.n");
        Vector x = new Vector(l);
        for (int i = 1; i <= x.n; i++) x.p[i - 1] += r.p[i - 1];
        return x;
    }
    public static Vector operator -(Vector l, Vector r)
    {
        if (l.n != r.n) throw new Exception("Vector operator - : l.n != r.n");
        Vector x = new Vector(l);
        for (int i = 1; i <= x.n; i++) x.p[i - 1] -= r.p[i - 1];
        return x;
    }
    public static Vector operator *(Vector l, double r)
    {
        Vector x = new Vector(l);
        for (int i = 1; i <= x.n; i++) x.p[i - 1] *= r;
        return x;
    }
    public static Vector operator *(double l, Vector r)
    {
        Vector x = new Vector(r);
        for (int i = 1; i <= x.n; i++) x.p[i - 1] *= l;
        return x;
    }
    public double GetL1() //Get L1 norm
    {
        double x = 0.0;
        for (int i = 1; i <= n; i++) x += Math.Abs(p[i - 1]);
        return x;
    }
    public double GetMaxNorm() //Get max norm
    {
        double max = Math.Abs(p[0]);
        for (int i = 2; i <= n; i++)
        {
            if (max < Math.Abs(p[i - 1]))
            {
                max = Math.Abs(p[i - 1]);
            }
        }
        return max;
    }
    public void Add(Vector x)
    {
        for (int i = 1; i <= n; i++)
        {
            p[i - 1] += x[i];
        }
    }
}
//******************************************************************
// ThreeVectorStruct - represents vector with 3 components, more
//                     efficient then class Vector
//******************************************************************
public struct ThreeVectorStruct
{
    double x, y, z;
    public ThreeVectorStruct(double x, double y, double z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    public ThreeVectorStruct(ThreeVectorStruct p)
    {
        x = p[1];
        y = p[2];
```

```
            z = p[3];
        }
        public double this[long i]
        {
            get
            {
                switch (i)
                {
                    case (1): return x;
                    case (2): return y;
                    case (3): return z;
                    default: throw new Exception("ThreeVectorStruct.operator[i]. i<1 || i>3 !");
                }
            }
            set
            {
                switch (i)
                {
                    case (1): x = value; break;
                    case (2): y = value; break;
                    case (3): z = value; break;
                    default: throw new Exception("ThreeVectorStruct.operator[i]. i<1 || i>3 !");
                }
            }
        }

        public static ThreeVectorStruct operator *(double l, ThreeVectorStruct r)
        {
            ThreeVectorStruct xv = new ThreeVectorStruct(l * r[1], l * r[2], l * r[3]);
            return xv;
        }
        public ThreeVectorStruct VectorMultiplication(ThreeVectorStruct x)
        {
            return new ThreeVectorStruct(this[2] * x[3] - this[3] * x[2],
                this[3] * x[1] - this[1] * x[3], this[1] * x[2] - this[2] * x[1]);
        }
        public ThreeVectorStruct NormalizedVectorMultiplication(ThreeVectorStruct x)
        {
            ThreeVectorStruct data = new ThreeVectorStruct(this[2] * x[3] - this[3] * x[2],
                this[3] * x[1] - this[1] * x[3], this[1] * x[2] - this[2] * x[1]);
            double norm = Math.Sqrt(data[1] * data[1] + data[2] * data[2] + data[3] * data[3]);
            return (1.0 / norm) * data;
        }
        public static ThreeVectorStruct operator +(ThreeVectorStruct l, ThreeVectorStruct r)
        {
            return new ThreeVectorStruct(l[1] + r[1], l[2] + r[2], l[3] + r[3]);
        }
        public static ThreeVectorStruct operator -(ThreeVectorStruct l, ThreeVectorStruct r)
        {
            return new ThreeVectorStruct(l[1] - r[1], l[2] - r[2], l[3] - r[3]);
        }
    }
    //****************************************************************
    // ThreeLongVectorStruct - represents integer vector with 3 components
    //****************************************************************
    public struct ThreeLongVectorStruct
    {
        private long x, y, z;
        public ThreeLongVectorStruct(long x, long y, long z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }
        public ThreeLongVectorStruct(ThreeLongVectorStruct p)
        {
            x = p[1];
            y = p[2];
            z = p[3];
        }
        public long this[long i]
        {
            get
            {
                switch (i)
                {
                    case (1): return x;
                    case (2): return y;
                    case (3): return z;
                    default: throw new Exception("ThreeLongVectorStruct.operator[i]. i<1 || i>3 !");
                }
            }
            set
            {
                switch (i)
                {
                    case (1): x = value; break;
                    case (2): y = value; break;
                    case (3): z = value; break;
                    default: throw new Exception("ThreeLongVectorStruct.operator[i]. i<1 || i>3 !");
                }
            }
        }
    }
    //****************************************************************
    // FullMatrix - represents a full rectangle matrix
    //              (all components are stored in memory)
    //****************************************************************
    public class FullMatrix
    {
        private int n, m;
        private double[,] p;

        private void ConstructorFullMatrix(int n, int m)
        {
            if (n < 1)
                throw new Exception("FullMatrix.Constructor:  n < 1! ");

            if (m < 1)
                throw new Exception("FullMatrix.Constructor:  m < 1! ");

            this.n = n;
            this.m = m;
            try
            {
                p = new double[n, m];

            }
            catch (Exception)
            {
                throw new Exception("FullMatrix.Constructor: Out of memory exception.");
            }
            Null();
        }
        //Initialization to 0.0
        public void Null()
        {
            for (int i = 1; i <= n; i++)
            {
                for (int j = 1; j <= m; j++)
                {
                    p[i - 1, j - 1] = 0.0;
                }
            }
        }
        public FullMatrix(int n)
        {
            ConstructorFullMatrix(n, n);
        }
        public FullMatrix(int n, int m)
        {
            ConstructorFullMatrix(n, m);
        }
        public int N { get { return n; } }
```

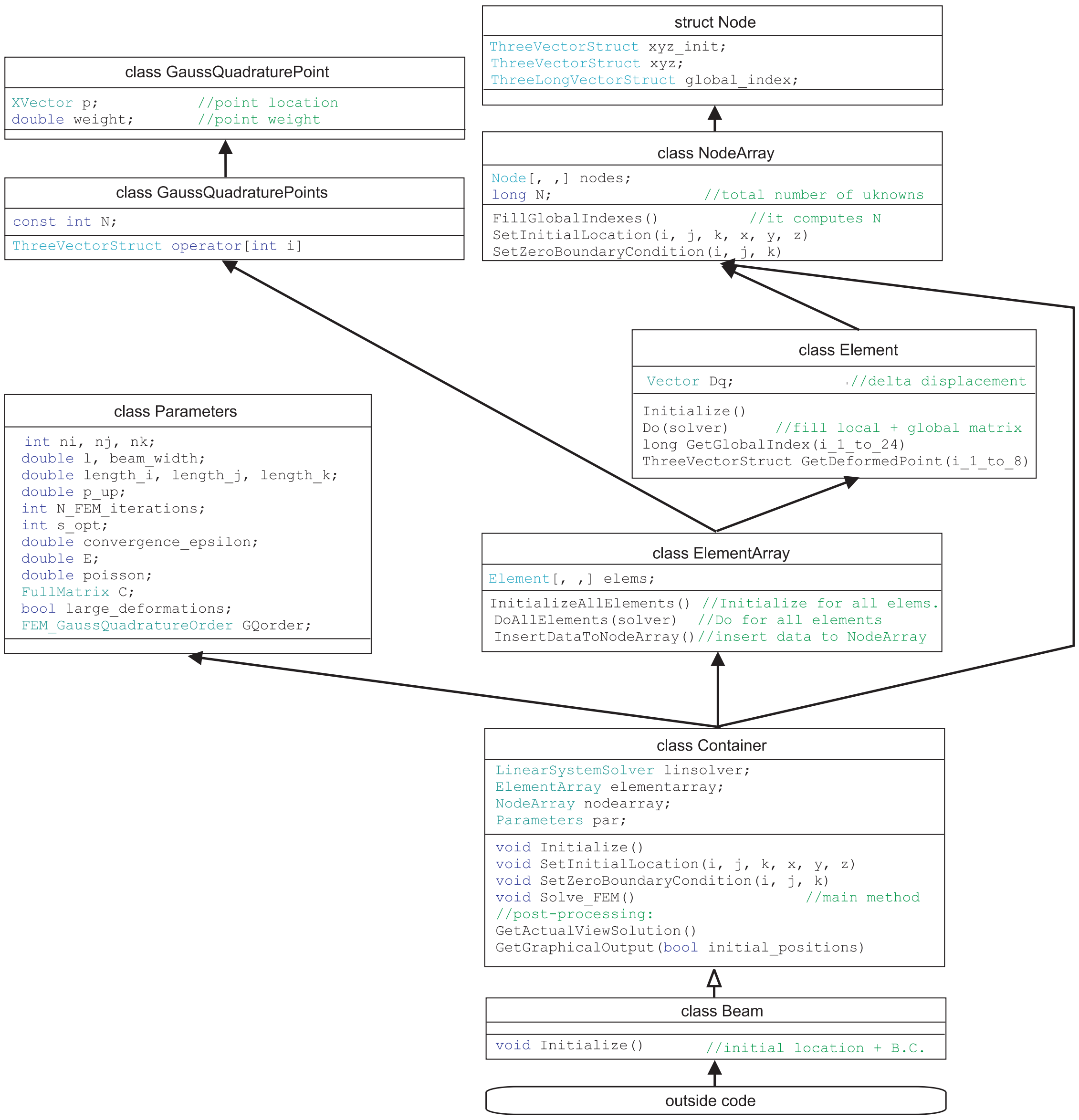
```
public int Getm() { return m; }
public static FullMatrix operator *(FullMatrix l, double r)
{
    FullMatrix x = new FullMatrix(l.N);
    for (int i = 0; i <= x.N - 1; i++)
    {
        for (int j = 0; j <= x.Getm() - 1; j++)
        {
            x.p[i, j] = l.Get(i + 1, j + 1) * r;
        }
    }
    return x;
}

public static Vector operator *(FullMatrix l, Vector r)
{

    if (l.Getm() != r.N)
    {
        throw new Exception("Vector operator * (l, r) : l and r have different dimensions!");
    }

    Vector y = new Vector(l.N);
    for (int i = 1; i <= y.N; i++)
    {
        y[i] = 0.0;
        for (int j = 1; j <= l.Getm(); j++)
        {
            y[i] += l.Get(i, j) * r[j];
        }
    }

    return y;
}
//Multiplication L^T * R
public static FullMatrix operator %(FullMatrix l, FullMatrix r)
{
    int n = l.N;
    if (n != r.N)
    {
        throw new Exception("Vector operator % (l, r) :  l and r have different dimensions!");
    }

    int m1 = l.Getm();
    int m2 = r.Getm();
    //n x m1   n x m2  =>  m1 x m2
    FullMatrix y = new FullMatrix(m1, m2);

    double x;

    for (int i = 1; i <= m1; i++)
    {
        for (int j = 1; j <= m2; j++)
        {
            x = 0.0;
            for (int k = 1; k <= n; k++)
            {
                x += l.Get(k, i) * r.Get(k, j);
            }
            y.Set(i, j, x);
        }
    }

    return y;
}
//Multiplication L^T * R
public static Vector operator %(FullMatrix l, Vector r)
{
    int n = l.N;
    if (n != r.N)
    {
        throw new Exception("Vector.operator(l, r) %: l and r have different dimensions!");
    }

    int m = l.Getm();
    //n x m   n  =>  m
    Vector y = new Vector(m);
    double x;
    for (int i = 1; i <= m; i++)
    {
        x = 0.0;
        for (int k = 1; k <= n; k++)
        {
            x += l.Get(k, i) * r[k];
        }
        y[i] = x;
    }
    return y;
}
public static FullMatrix operator *(FullMatrix l, FullMatrix r)
{
    if (l.Getm() != r.N)
    {
        throw new Exception("Vector.operator(l, r) *: l and r have different dimensions!");
    }

    FullMatrix y = new FullMatrix(l.N, r.Getm());
    double x;
    for (int i = 1; i <= y.N; i++)
    {
        for (int j = 1; j <= y.Getm(); j++)
        {

            x = 0.0;
            for (int k = 1; k <= l.Getm(); k++)
            {
                x += l.Get(i, k) * r.Get(k, j);
            }
            y.Set(i, j, x);
        }
    }
    return y;
}
public double Get(int i, int j)
{
    VALIDATE(i, j);
    return p[i - 1, j - 1];
}
public void Set(int i, int j, double val)
{
    VALIDATE(i, j);
    p[i - 1, j - 1] = val;
}
public void SetSymmetric(int i, int j, double val)
{
    if (n != m)
    {
        throw new Exception("FullMatrix.SetSymmetric(i, j, val):  n != m !");
    }
    VALIDATE(i, j);
    p[i - 1, j - 1] = p[j - 1, i - 1] = val;
}
public Vector GAUSS(Vector b)  //note that it destroys matrix p!
{
    double multiplier, exchanger;
    Vector b_ = new Vector(b);
    // 0. Control - if matrix has same type as right_side
    if (b_.N != n)
    {
        throw new Exception("FullMatrix.GAUSS(): b_.N != n !");
    }

    double bL1 = b.GetL1();
    if (bL1 == 0.0)
    {
        throw new Exception("FullMatrix.GAUSS(): b.GetL1 == 0.0");
    }

    //1. Make upper triangle matrix
    //1.1   From the first to the (last - 1)th column
    for (int j = 1; j <= n - 1; j++)
    {
        //1.2 If the diagonal element is zero
        if (Get(j, j) == 0.0)
        {
            for (int h = j + 1; h <= n; h++)
            {
                if (Get(h, j) != 0.0)
                {

                    //1.1.1 Change of rows j and h
                    for (int t = j; t <= n; t++)
                    {
                        exchanger = Get(j, t);
                        Set(j, t, Get(h, t));
                        Set(h, t, exchanger);
                    }
                    exchanger = b_[j];
                    b_[j] = b_[h];
                    b_[h] = exchanger;
                    // end of change rows j and h
                    break;
                }
                if (h == n) goto End_of_cycle;
                //all numbers under diagonal are zero, so there’s nothing to eliminate
            }

        }
        //1.2 Only if there is some rows to null do this
        for (int i = j + 1; i <= n; i++)
        {
            //If this line has zero don‘t do it
            if (Get(i, j) != 0.0)
            {
                multiplier = Get(i, j) / Get(j, j);
                for (int r = j + 1; r <= n; r++)
                {
                    Add(i, r, -multiplier * Get(j, r));
                }
                b_[i] -= multiplier * b_[j];
            }
        }
        End_of_cycle: ;
    }

    //2. Look if matrix is regular - look for zero’s on the diagonal
    for (int k = 1; k <= n; k++)
    {
        if (Get(k, k) == 0.0) { throw new Exception("FullMatrix.GAUSS: matrix is singular!"); }
    }

    //3.Back process of the Gauss elimination
    Vector Solution = new Vector(n);

    for (int i = n; i >= 1; i--)
    {
        for (int j = i + 1; j <= n; j++)
        {
            b_[i] -= Get(i, j) * Solution[j];
        }
        Solution[i] = b_[i] / Get(i, i);
    }

    return Solution;
}
public FullMatrix Inverse()  //note that it destroys matrix p!
```

```
{
    if (n != m)
    {
        throw new Exception("FullMatrix.Inverse(): n != m!");
    }

    //Solution method is based on solving the system A * AINV = I
    FullMatrix X = new FullMatrix(n * n);
    Vector b = new Vector(n * n);

    //1.0 Fill b
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (i == j)
            {
                b[(j - 1) * n + i] = 1.0;
            }
            else
            {
                b[(j - 1) * n + i] = 0.0;
            }
        }
    }
    //2.0 Fill X
    for (int k = 1; k <= n; k++)   //k times A
    {
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
            {
                X.Set((k - 1) * n + i, (k - 1) * n + j, Get(i, j));
            }
    }

    //3.0 Solve
    b = X.GAUSS(b);

    //4.0 Fill AINV
    FullMatrix AINV = new FullMatrix(n);
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            AINV.Set(i, j, b[(j - 1) * n + i]);
        }
    }
    return AINV;
}
public static double GetDet(FullMatrix A) //only for 3x3 matrix
{
    if ((A.N == 3) && (A.Getm() == 3))
    {
        return
             A.Get(1, 1) * (A.Get(2, 2) * A.Get(3, 3) - A.Get(2, 3) * A.Get(3, 2))
            - A.Get(1, 2) * (A.Get(2, 1) * A.Get(3, 3) - A.Get(2, 3) * A.Get(3, 1))
            + A.Get(1, 3) * (A.Get(2, 1) * A.Get(3, 2) - A.Get(2, 2) * A.Get(3, 1));
    }
    else
    {
        throw new Exception("FullMatrix.GetDet(): it requires matrix of type 3x3!");
    }
}
public void Add(int i, int j, double val)
{
    VALIDATE(i, j);
    p[i - 1, j - 1] += val;
}
//Add Matrix
public void Add(FullMatrix a)
{
    if ((n != a.N) || (m != a.Getm()))
```

```
        {
            throw new Exception("FullMatrix.Add(a): different n or m!");
        }

        for (int i = 1; i <= n; i++)
        {
            for (int j = 1; j <= m; j++)
            {
                p[i - 1, j - 1] += a.Get(i, j);
            }
        }
    }
    private void VALIDATE(int i, int j)
    {
        if ((i < 1) || (i > n))
            throw new Exception("FullMatrix.VALIDATE(i, j): (i < 1) || (i > n)");
        if ((j < 1) || (j > m))
            throw new Exception("FullMatrix.VALIDATE(i, j): (j < 1) || (j > m)");
    }
}
//*****************************************************************
// LinearSystemSolver - represents the direct Gauss solver
// Note that:
// 1.No pivoting (less memory allocated)
// 2.Solution vector included.
// 3.Right hand side (b vector) is in 0-column index of p array
// 4.Rows indexed from 0!
//*****************************************************************
public class LinearSystemSolver
{
    private long n, s;
    private double[,] p;
    private Vector solution;
    public Vector Solution { get { return solution; } }
    private void Constructor(long n, long s)
    {
        if ((n < 1) || (s < 0))
            throw new Exception("LinearSystemSolver:  n < 1  ||  s < 0 ! ");

        if (s > n - 1) s = n - 1;
        this.n = n;
        this.s = s;

        try
        {
            p = new double[n, 2 * s + 2];
            solution = new Vector(n);
        }
        catch (Exception)
        {
            throw new Exception("LinearSystemSolver: Operator new failed: Out of memory.");
        }
        Null();
    }
    public void Null() //Initialization to 0.0
    {
        for (long i = 1; i <= n; i++)
        {
            for (long k = 0; k <= 2 * s + 1; k++)
            {
                p[-1 + i, k] = 0.0;
            }
        }
    }
    public LinearSystemSolver(long n, long s)
    {
        Constructor(n, s);
    }
    //Matrix is stored in array p[ i, j ], where i = 1,..,n are rows and j columns:
    // 1 <= j <= s      - left band
    //   j = s+1       - diagonal
    // s+2 <= j <= 2s+1 - right band
    public void AddValue(long i, long j, double val)
    {
        p[-1 + i, j - i + s + 1] += val;
    }
    public void AddValue(long i, double val)
    {
        p[-1 + i, 0] += val;
    }
    public void Solve()
    {
        double multiplier;
        //1. Make upper triangle matrix
        //1.1   From the first to the last - 1 column
        for (long k = 1; k <= n - 1; k++)
        {
            // I will null rows, but at the end of matrix I have less and less rows to null!!
            for (long h = 1; h <= Math.Min(s, n - k); h++)
            {
                //If this line has zero don‘t do it
                if (p[-1 + k + h, s + 1 - h] != 0.0)
                {

                    multiplier = p[-1 + k + h, s + 1 - h] / p[-1 + k, s + 1];
                    // It now subtracts 2 * s elements
                    for (long j = 2; j <= s + 1; j++)
                    {
                        p[-1 + k + h, s + j - h] -= multiplier * p[-1 + k, s + j];
                    }

                    p[-1 + k + h, 0] -= multiplier * p[-1 + k, 0];
                }
            }
        }

        //3.Backward process of the Gauss elimination
        // k means rows (k = 1 is the last one), j means distance from the diagonal
        for (long k = 1; k <= n; k++)
        {
            for (long j = 1; j <= Math.Min(k - 1, s); j++)
            {
                p[-1 + n - k + 1, 0] -= p[-1 + n - k + 1, s + 1 + j] * solution[n - k + 1 + j];
            }
            solution[n - k + 1] = p[-1 + n - k + 1, 0] / p[-1 + n - k + 1, s + 1];
        }
    }
}
```

12.4.2 FELIB

The second part consists of the library (FELIB) designed to solve the finite element problem (FEP). The class diagram of FELIB is depicted on Figure 12.8.



```
struct Node
ThreeVectorStruct xyz_init;
ThreeVectorStruct xyz;
ThreeLongVectorStruct global_index;

class GaussQuadraturePoint
XVector p;              //point location
double weight;          //point weight

class GaussQuadraturePoints
const int N;
ThreeVectorStruct operator[int i]

class NodeArray
Node[, ,] nodes;
long N;                      //total number of uknowns
FillGlobalIndexes()           //it computes N
SetInitialLocation(i, j, k, x, y, z)
SetZeroBoundaryCondition(i, j, k)

class Element
Vector Dq;                  //delta displacement
Initialize()
Do(solver)       //fill local + global matrix
long GetGlobalIndex(i_1_to_24)
ThreeVectorStruct GetDeformedPoint(i_1_to_8)

class Parameters
int ni, nj, nk;
double l, beam_width;
double length_i, length_j, length_k;
double p_up;
int N_FEM_iterations;
int s_opt;
double convergence_epsilon;
double E;
double poisson;
FullMatrix C;
bool large_deformations;
FEM_GaussQuadratureOrder GQorder;

class ElementArray
Element[, ,] elems;
InitializeAllElements() //Initialize for all elems.
DoAllElements(solver)  //Do for all elements
InsertDataToNodeArray()//insert data to NodeArray

class Container
LinearSystemSolver linsolver;
ElementArray elementarray;
NodeArray nodearray;
Parameters par;
void Initialize()
void SetInitialLocation(i, j, k, x, y, z)
void SetZeroBoundaryCondition(i, j, k)
void Solve_FEM()                             //main method
//post-processing:
GetActualViewSolution()
GetGraphicalOutput(bool initial_positions)

class Beam
void Initialize()          //initial location + B.C.

outside code
```

Figure 12.8: UML diagram showing the classes structure of FELIB. The all public members (the middle part) and public methods (the lower part) are shown.

Let us briefly describe the code structure. Nodes are represented by structure `Node` and elements are represented by class `Element`. There are two more classes defined: `NodeArray` and `ElementArray` representing array of nodes and elements. It is obvious that some information describing the FEP belong to a node (for example initial position or boundary conditions) and some belongs to an element (for example local matrixes). Therefore all information about the problem are stored either in `NodeArray` or `ElementArray` classes depending on a particular convenience. The class `Container` manages all classes solving the FEP in general. This class does not contain the definition of our test problem – a loaded beam. This definition is in `Beam` class, which is inherited from `Container` class and it is the top most class in the hierarchy of FELIB which is run from outside (for example from `main()`). The way how it is used is as follows:

1. Create the instance of `Beam` class (let us denote it by `beam`).

2. Call `beam.Initialize`. This method internally calls `beam.SetInitialLocation` and
   `beam.SetZeroBoundaryCondition` methods to insert information about the initial shape of the body and appropriate boundary conditions. These two methods internally call methods

   `NodeArray.SetInitialLocation` and

   `NodeArray.SetZeroBoundaryCondition`

   that make the job. Finally the method calls `Element.Initialize` method to all elements which allocate all local matrixes and vectors and also precompute some data (for example det$\mathbf{J}$).

3. Call `beam.Solve_FEM` that is the main method solving the FEP. The method contains the main convergence loop. Internally the method calls `Element.Do` method for each element which calculates the local vectors and matrixes and fill the global matrix. Then the global matrix is inverted (method `LinearSystemSolver.Solve`) and solution increment is inserted again to all elements to update their local vectors and matrices (method `Container.SolverResultToElementArray`). Then the story is repeated – method `Element.Do` is called for each element, the matrix is inverted and the result is added to local vectors and matrixes in each elements. This is made couple of times until the the convergence criteria is satisfied.

4. Call `beam.GetGraphicalOutput` which returns the computed results in an appropriate form to be displayed.

5. Call `beam.GetActualViewSolution` which returns the computed results in an appropriate form to be saved to file.

There are two more classes to be mentioned:

- Class `Parameters` that contains all problem parameters.

- Classes `GaussQuadraturePoint` and `GaussQuadraturePoints` that wrapp the table of Gauss points in a convenient way.

In the FELIB code there are also some classes that are not connected with the FE problem. Let us briefly summarize them:

- Class X3D_3D represents the graphical 3D object which can be displayed on the screen.

- Class Roller represents message log window.

- Class Exception is the standard .NET class representing an exception error.

```
//*****************************************************************
//Gauss integration points
//Example:
//Parameter.GQorder = FEM_GaussQuadratureOrder.TwoPoints;
//GaussQuadraturePoints x = new GaussQuadraturePoints();
//for (int i = 1; i <= x.N; i++)
//{
//      x[i][1];     //... X1 component of the quadrature point
//      x[i][2];     //... X2 component of the quadrature point
//      x[i][3];     //... X3 component of the quadrature point
//      x[i].weight; //... weight of the point
//}
//*****************************************************************
public class GaussQuadraturePoints
{
    public class GaussQuadraturePoint
    {
        public Vector p;                    //point location
        public double weight;               //point weight
        public GaussQuadraturePoint()
        {
            p = new Vector(3);
        }
    }
    private GaussQuadraturePoint[] p;       //array
    private int n;
    public GaussQuadraturePoints(FEM_GaussQuadratureOrder GQorder)
    {
        if (GQorder == FEM_GaussQuadratureOrder.TwoPoints)
        {
            n =  2 * 2 * 2;
        }
        else
        {
            n = 3 * 3 * 3;
        }
        p = new GaussQuadraturePoint[n];                  //0..7/0..26
        for (int i = 0; i <= n - 1; i++)
        {
            p[i] = new GaussQuadraturePoint();
        }

        int I = 0;
        if (GQorder == FEM_GaussQuadratureOrder.TwoPoints)
        {
            Vector point = new Vector(2);
            point[1] = -1.0 / Math.Sqrt(3.0);
            point[2] = 1.0 / Math.Sqrt(3.0);

            for (int i = 1; i <= 2; i++)
            {
                for (int j = 1; j <= 2; j++)
                {
                    for (int k = 1; k <= 2; k++)
                    {
                        I = 4 * (i - 1) + 2 * (j - 1) + k - 1;   //range 0..7

                        p[I].p[1] = point[i];
                        p[I].p[2] = point[j];
                        p[I].p[3] = point[k];
                        p[I].weight = 1.0;
                    }
                }
            }
        }
        else
        {
            Vector point = new Vector(-Math.Sqrt(0.6), 0.0, Math.Sqrt(0.6));
            Vector weight = new Vector(5.0 / 9.0, 8.0 / 9.0, 5.0 / 9.0);

            for (int i = 1; i <= 3; i++)
            {
                for (int j = 1; j <= 3; j++)
                {
                    for (int k = 1; k <= 3; k++)
                    {
                        I = 9 * (i - 1) + 3 * (j - 1) + k - 1;  //range 0..26
                        p[I].p[1] = point[i];
                        p[I].p[2] = point[j];
                        p[I].p[3] = point[k];
                        p[I].weight = weight[i] * weight[j] * weight[k];
                    }
                }
            }
        }
    }
    public int N  { get{ return n;} }
    public GaussQuadraturePoint this[int i]             //0...GQP.N
    {
        get
        {
            return p[i - 1];
        }
    }
}
//*****************************************************************
// GaussQuadratureOrder
//*****************************************************************
public enum FEM_GaussQuadratureOrder
{
    TwoPoints, ThreePoints
}
//*****************************************************************
// Parameters
//*****************************************************************
public class Parameters
{
    public int ni = 50;                       //number of elements in X1 direction
    public int nj = 2;                        //number of elements in X2 direction
    public int nk = 2;                        //number of elements in X3 direction
    public double l = 0.11;                   //beam length
    public double beam_width = 0.00275;       //beam_width = beam depth
    public double length_i,length_j, length_k;//element length in X1/X2/X3 direction
    public double p_up = 50.0;                //pressure on the upper plane
    public int N_FEM_iterations = 10;         //Newton-Ralphston iterations
    public int s_opt;                     //width of global matrix band
    public double convergence_epsilon = Math.Pow(10.0, -9.0);

    public double E = Math.Pow(10.0, 7.0);   //Young modulus
    public double poisson = 0.25;         //Poisson ratio
    public FullMatrix C = new FullMatrix(6);
    public bool large_deformations = true;    //switch large/small deformations
    public FEM_GaussQuadratureOrder GQorder = FEM_GaussQuadratureOrder.TwoPoints;

    public Parameters()
    {
        length_i = l / (double)ni;
        length_j = beam_width / (double)nj;
        length_k = beam_width / (double)nk;

        double C0 = E / (1.0 + poisson) / (1.0 - 2.0 * poisson);
         for (int i = 1; i <= 3; i++)
         {
             C.Set(i, i, C0 * (1.0 - poisson));
             C.Set(i + 3, i + 3, C0 * (0.5 - poisson));
         }
         C.SetSymmetric(1, 2, C0 * poisson);
         C.SetSymmetric(1, 3, C0 * poisson);
         C.SetSymmetric(2, 3, C0 * poisson);
    }
}
    //*****************************************************************
// Node  - represents a node (vertex)
//*****************************************************************
```

```
public struct Node
{
    public ThreeVectorStruct xyz_init;         //initial position in x/y/z direction
    public ThreeVectorStruct xyz;              //deformed position in x/y/z direction
    public ThreeLongVectorStruct global_index; //matrix global index (value -1 means B.C.)
}
//*****************************************************************
// NodeArray - represents a structural array of nodes
//*****************************************************************
public class NodeArray
{
    public Node[, ,] nodes;
    public long N = 0;
    private Parameters par;
    public NodeArray(Parameters par)
    {
        this.par = par;
        nodes = new Node[par.ni + 1, par.nj + 1, par.nk + 1];   //indexed from 0 !!
    }
    public void FillGlobalIndexes()
    {
        N = 0;
        for (int i = 0; i <= par.ni; i++)
        {
            for (int j = 0; j <= par.nj; j++)
            {
                for (int k = 0; k <= par.nk; k++)
                {
                    for (int I = 1; I <= 3; I++)
                    {
                        if (nodes[i, j, k].global_index[I] != -1)
                        {
                            nodes[i, j, k].global_index[I] = ++N;
                        }
                    }
                }
            }
        }
    }
    public void SetInitialLocation(int i, int j, int k, double x, double y, double z)
    {
        VALIDATE(i, j, k);
        nodes[i, j, k].xyz_init[1] = nodes[i, j, k].xyz[1] = x;
        nodes[i, j, k].xyz_init[2] = nodes[i, j, k].xyz[2] = y;
        nodes[i, j, k].xyz_init[3] = nodes[i, j, k].xyz[3] = z;
    }
    public void SetZeroBoundaryCondition(int i, int j, int k)
    {
        VALIDATE(i, j, k);
        nodes[i, j, k].global_index[1] = -1;
        nodes[i, j, k].global_index[2] = -1;
        nodes[i, j, k].global_index[3] = -1;
    }
    private void VALIDATE(int i, int j, int k)
    {
        if ((i < 0) || (j < 0) || (k < 0))
        {
            throw new Exception("NodeArray.VALIDATE: (i < 0) || (j < 0) || (k < 0)");
        }
        if ((i > par.ni) || (j > par.nj) || (k > par.nk))
        {
            throw new Exception("NodeArray.VALIDATE: i > par.ni || j > par.nj || k > par.nk");
        }
    }
}
//*****************************************************************
// Element
//*****************************************************************
public class Element
{
    private GaussQuadraturePoints GQ;           //Gauss Quadrature reference
    private NodeArray nodearray;                //NodeArray reference
    private Parameters par;                   //Parameters reference
    private int local_i, local_j, local_k;    //location in the Container
    private bool first_method_call;

    //for every Gauss point - initialized once
    private double[] detJ;
    private Vector[,] DaDX123;                //[point 1-8 , a_i (1-8)][derivated by 1-3]
                                                //this array plays the role of BN also
    //for every Gauss point - updated at every iteration step
    private Vector q;                         //displacement 24x1
    private Vector dq;                         //delta displacement 24x1 - one iteration
    private FullMatrix[] BL;
    private Vector[] s;                       //2nd Piola Kirchoff 6x1

    //public properties
    public Vector Dq { get{ return dq; } }
    //Legend:
    //*  0  *  1  *  ni-1  *    elements
    //0     1     2       ni    nodes
    public Element(Parameters par, NodeArray nodearray, GaussQuadraturePoints GQ, int local_i, int local_j, int local_k)
    {
        this.par = par;
        this.nodearray = nodearray;
        this.GQ = GQ;
        q = new Vector(24);                  //displacement 24x1 -- "input"
        dq = new Vector(24);                 //delta displacement 24x1 - one iteration
        detJ = new double[GQ.N];
        DaDX123 = new Vector[GQ.N, 8];
        s = new Vector[GQ.N];                //2nd Piola Kirchoff 6x1
        BL = new FullMatrix[GQ.N];

        for (int k = 0; k <= GQ.N - 1; k++)
        {
            BL[k] = new FullMatrix(6, 24);
            for (int i = 1; i <= 8; i++)
            {
                DaDX123[k, i - 1] = new Vector(3);
            }
            s[k] = new Vector(6);
        }
        first_method_call = true;

        this.local_i = local_i;
        this.local_j = local_j;
        this.local_k = local_k;
    }
    public void Initialize()
    {
        FullMatrix J_inv = new FullMatrix(3);
        Vector DaDrst = new Vector(3);
        for (int k = 1; k <= GQ.N; k++)               //Gauss points
        {
            J_inv.Null();
            //1.0 Jinv a detJ
            for (int i = 1; i <= 8; i++)
            {
                J_inv.Add(1, 1, GetDaDr(i, GQ[k].p) * GetInitialPoisition(i, 1));
                J_inv.Add(2, 1, GetDaDs(i, GQ[k].p) * GetInitialPoisition(i, 1));
                J_inv.Add(3, 1, GetDaDt(i, GQ[k].p) * GetInitialPoisition(i, 1));
                J_inv.Add(1, 2, GetDaDr(i, GQ[k].p) * GetInitialPoisition(i, 2));
                J_inv.Add(2, 2, GetDaDs(i, GQ[k].p) * GetInitialPoisition(i, 2));
                J_inv.Add(3, 2, GetDaDt(i, GQ[k].p) * GetInitialPoisition(i, 2));
                J_inv.Add(1, 3, GetDaDr(i, GQ[k].p) * GetInitialPoisition(i, 3));
                J_inv.Add(2, 3, GetDaDs(i, GQ[k].p) * GetInitialPoisition(i, 3));
                J_inv.Add(3, 3, GetDaDt(i, GQ[k].p) * GetInitialPoisition(i, 3));
            }
            detJ[k - 1] = FullMatrix.GetDet(J_inv);
            J_inv = J_inv.Inverse();

            //2.0 DaDxyz[i] (lokalni)
            for (int i = 1; i <= 8; i++)
            {
                DaDrst[1] = GetDaDr(i, GQ[k].p);
                DaDrst[2] = GetDaDs(i, GQ[k].p);
                DaDrst[3] = GetDaDt(i, GQ[k].p);
                DaDX123[k - 1, i - 1] = J_inv * DaDrst;
            }
        }
        first_method_call = true;
    }
    public void Do(LinearSystemSolver linsolver)
    {
        FullMatrix Klocal = new FullMatrix(24);         //local matrix 24x24
        Vector flocal = new Vector(24);     //local RHS 24x1
        FullMatrix Z = new FullMatrix(3, 3);
        double val, integration_coefficient;

        //1.0 Update q
        if (!first_method_call) { q += dq; }

        // 2.0 Update r
        if (local_k == par.nk - 1)
        {
            ThreeVectorStruct u1 = GetDeformedPoint(6) - GetDeformedPoint(5);
            ThreeVectorStruct u2 = GetDeformedPoint(8) - GetDeformedPoint(5);
            ThreeVectorStruct n = -0.25 * par.p_up * u1.VectorMultiplication(u2);
            for (int K = 1; K <= 3; K++)
            {
                //left points with respect to i
                flocal[5 + 8 * (K - 1)] = n[K];
                flocal[8 + 8 * (K - 1)] = n[K];
                //right points with respect to i
                flocal[6 + 8 * (K - 1)] = n[K];
                flocal[7 + 8 * (K - 1)] = n[K];
            }
        }

        for (int k = 1; k <= GQ.N; k++)         //Gauss points (GQ.N = 8 or 27)
        {
                // 3.0 Update s
                if (!first_method_call)
                {
                    s[k - 1] = s[k - 1] + par.C * BL[k - 1] * dq;
                }

                // 4.0 Compute Z, update BL
                if (par.large_deformations)
                {
                    for (int i = 1; i <= 3; i++)
                    {
                        for (int j = 1; j <= 3; j++)
                        {
                            val = 0.0;
                            for (int s = 1; s <= 8; s++)
                            {
                                val += DaDX123[k - 1, s - 1][j] * q[s + 8 * (i - 1)];
                            }
                            Z.Set(i, j, val);
                        }
                    }
                }
                BL[k - 1].Null();
                Vector Da;
                for (int s = 1; s <= 8; s++)
                {
                Da = DaDX123[k - 1, s - 1];
                BL[k - 1].Add(1, s, (1.0 + Z.Get(1, 1)) * Da[1]);
                BL[k - 1].Add(1, 8 + s, Z.Get(2, 1) * Da[1]);
                BL[k - 1].Add(1, 16 + s, Z.Get(3, 1) * Da[1]);
                BL[k - 1].Add(2, s, Z.Get(1, 2) * Da[2]);
                BL[k - 1].Add(2, 8 + s, (1.0 + Z.Get(2, 2)) * Da[2]);
                BL[k - 1].Add(2, 16 + s, Z.Get(3, 2) * Da[2]);
                BL[k - 1].Add(3, s, Z.Get(1, 3) * Da[3]);
                BL[k - 1].Add(3, 8 + s, Z.Get(2, 3) * Da[3]);
                BL[k - 1].Add(3, 16 + s, (1.0 + Z.Get(3, 3)) * Da[3]);
                BL[k - 1].Add(4, s, Z.Get(1, 2) * Da[1] + (1.0 + Z.Get(1, 1)) * Da[2]);
                BL[k - 1].Add(4, 8 + s, (1.0 + Z.Get(2, 2)) * Da[1] + Z.Get(2, 1) * Da[2]);
                BL[k - 1].Add(4, 16 + s, Z.Get(3, 2) * Da[1] + Z.Get(3, 1) * Da[2]);
                BL[k - 1].Add(5, s, Z.Get(1, 3) * Da[2] + Z.Get(1, 2) * Da[3]);
                BL[k - 1].Add(5, 8 + s, Z.Get(2, 3) * Da[2] + (1.0 + Z.Get(2, 2)) * Da[3]);
                BL[k - 1].Add(5, 16 + s, (1.0 + Z.Get(3, 3)) * Da[2] + Z.Get(3, 2) * Da[3]);
                BL[k - 1].Add(6, s, Z.Get(1, 3) * Da[1] + (1.0 + Z.Get(1, 1)) * Da[3]);
                BL[k - 1].Add(6, 8 + s, Z.Get(2, 3) * Da[1] + Z.Get(2, 1) * Da[3]);
                BL[k - 1].Add(6, 16 + s, (1.0 + Z.Get(3, 3)) * Da[1] + Z.Get(3, 1) * Da[3]);
                }

                // 5.0 Add all to Klocal, flocal
                //5.1 (BN^T)S(BN)
                integration_coefficient = GQ[k].weight * detJ[k - 1];
                if (!first_method_call)
                {
                    if (par.large_deformations)
                    {
                        for (int i = 1; i <= 8; i++)
                        {
                            for (int j = 1; j <= 8; j++)
                            {
                val = s[k - 1][1] * DaDX123[k - 1, i - 1][1] * DaDX123[k - 1, j - 1][1] +
                      s[k - 1][2] * DaDX123[k - 1, i - 1][2] * DaDX123[k - 1, j - 1][2] +
                      s[k - 1][3] * DaDX123[k - 1, i - 1][3] * DaDX123[k - 1, j - 1][3] +
                      s[k - 1][4] * (DaDX123[k - 1, i - 1][1] * DaDX123[k - 1, j - 1][2] +
                                     DaDX123[k - 1, i - 1][2] * DaDX123[k - 1, j - 1][1]) +
                      s[k - 1][5] * (DaDX123[k - 1, i - 1][2] * DaDX123[k - 1, j - 1][3] +
                                     DaDX123[k - 1, i - 1][3] * DaDX123[k - 1, j - 1][2]) +
                      s[k - 1][6] * (DaDX123[k - 1, i - 1][3] * DaDX123[k - 1, j - 1][1] +
                                     DaDX123[k - 1, i - 1][1] * DaDX123[k - 1, j - 1][3]);
                                Klocal.Add(i, j, val);
                                Klocal.Add(i + 8, j + 8, val);
                                Klocal.Add(i + 16, j + 16, val);
                            }
                        }
                    }
                    flocal.Add((BL[k - 1] % s[k - 1]) * -integration_coefficient);
                }
                Klocal.Add(((BL[k - 1] % par.C) * BL[k - 1]) * integration_coefficient);
        }
        //6.0 Add to solver
        int index1, index2;
        for (int r = 1; r <= 24; r++)
        {
            index1 = (int)GetGlobalIndex(r);
            if (index1 > -1)
            {
                val = flocal[r];
                if (val != 0.0)
                {
                    linsolver.AddValue(index1, val);
               }
                //Klocal
                for (int s = 1; s <= 24; s++)
                {
                    index2 = (int)GetGlobalIndex(s);
                    if (index2 > -1)
                    {
                        val = Klocal.Get(r, s);
                        if (val != 0.0)
                        {
                            linsolver.AddValue(index1, index2, val);
                        }
                    }
                }
            }
        }

        first_method_call = false;
    }
```

```
private Node GetNode(int i_from_1_to_8)
{
    switch (i_from_1_to_8)
    {
        case 1: return nodearray.nodes[local_i, local_j, local_k];
        case 2: return nodearray.nodes[local_i + 1, local_j, local_k];
        case 3: return nodearray.nodes[local_i + 1, local_j + 1, local_k];
        case 4: return nodearray.nodes[local_i, local_j + 1, local_k];
        case 5: return nodearray.nodes[local_i, local_j, local_k + 1];
        case 6: return nodearray.nodes[local_i + 1, local_j, local_k + 1];
        case 7: return nodearray.nodes[local_i + 1, local_j + 1, local_k + 1];
        default: return nodearray.nodes[local_i, local_j + 1, local_k + 1];
    }
}
private double GetInitialPoisition(int i_from_1_to_8, int xyz_123)
{
    return GetNode(i_from_1_to_8).xyz_init[xyz_123];
}
public double GetGlobalIndex(int i_from_1_to_24)
{
    int xyz_123 = (i_from_1_to_24 - 1) / 8 + 1;      //1..3
    int i_from_1_to_8 = i_from_1_to_24 - (xyz_123 - 1) * 8;  //1..8
    return GetNode(i_from_1_to_8).global_index[xyz_123];
}
public ThreeVectorStruct GetDeformedPoint(int i_from_1_to_8)
{
    Node node = GetNode(i_from_1_to_8);
    return new ThreeVectorStruct(node.xyz_init[1] + q[i_from_1_to_8],
     node.xyz_init[2] + q[i_from_1_to_8 + 8], node.xyz_init[3] + q[i_from_1_to_8 + 16]);
}
private static double GetDaDr(int i_from_1_to_8, Vector rst)
{
    switch (i_from_1_to_8)
    {
        case 1: return -(1.0 - rst[2]) * (1.0 - rst[3]) / 8.0;
        case 2: return +(1.0 - rst[2]) * (1.0 - rst[3]) / 8.0;
        case 3: return +(1.0 + rst[2]) * (1.0 - rst[3]) / 8.0;
        case 4: return -(1.0 + rst[2]) * (1.0 - rst[3]) / 8.0;
        case 5: return -(1.0 - rst[2]) * (1.0 + rst[3]) / 8.0;
        case 6: return +(1.0 - rst[2]) * (1.0 + rst[3]) / 8.0;
        case 7: return +(1.0 + rst[2]) * (1.0 + rst[3]) / 8.0;
        case 8: return -(1.0 + rst[2]) * (1.0 + rst[3]) / 8.0;
    }
    throw new Exception("GetDaDr(int i_from_1_to_8, Vector rst): wrong value of i_from_1_to_8");
}
private static double GetDaDs(int i_from_1_to_8, Vector rst)
{
    switch (i_from_1_to_8)
    {
        case 1: return -(1.0 - rst[1]) * (1.0 - rst[3]) / 8.0;
        case 2: return -(1.0 + rst[1]) * (1.0 - rst[3]) / 8.0;
        case 3: return +(1.0 + rst[1]) * (1.0 - rst[3]) / 8.0;
        case 4: return +(1.0 - rst[1]) * (1.0 - rst[3]) / 8.0;
        case 5: return -(1.0 - rst[1]) * (1.0 + rst[3]) / 8.0;
        case 6: return -(1.0 + rst[1]) * (1.0 + rst[3]) / 8.0;
        case 7: return +(1.0 + rst[1]) * (1.0 + rst[3]) / 8.0;
        case 8: return +(1.0 - rst[1]) * (1.0 + rst[3]) / 8.0;
    }
    throw new Exception("GetDaDs(int i_from_1_to_8, Vector rst): wrong value of i_from_1_to_8");
}
private static double GetDaDt(int i_from_1_to_8, Vector rst)
{
    switch (i_from_1_to_8)
    {
        case 1: return -(1.0 - rst[1]) * (1.0 - rst[2]) / 8.0;
        case 2: return -(1.0 + rst[1]) * (1.0 - rst[2]) / 8.0;
        case 3: return -(1.0 + rst[1]) * (1.0 + rst[2]) / 8.0;
        case 4: return -(1.0 - rst[1]) * (1.0 + rst[2]) / 8.0;
        case 5: return +(1.0 - rst[1]) * (1.0 - rst[2]) / 8.0;
        case 6: return +(1.0 + rst[1]) * (1.0 - rst[2]) / 8.0;
        case 7: return +(1.0 + rst[1]) * (1.0 + rst[2]) / 8.0;
        case 8: return +(1.0 - rst[1]) * (1.0 + rst[2]) / 8.0;
```

```
            }
            throw new Exception("GetDaDt(int i_from_1_to_8, Vector rst): wrong value of i_from_1_to_8");
        }
    }
    //*******************************************************************
    // ElementArray  - represents a structural array of elements
    //*******************************************************************
    public class ElementArray
    {
        private Parameters par;
        private GaussQuadraturePoints GQ;
        private Element[, ,] elems;
        public Element[, ,] Elems
        {
            get
            {
                return elems;
            }
        }
        public ElementArray(Parameters par, NodeArray nodearray)
        {
            this.par = par;
            GQ = new GaussQuadraturePoints(par.GQorder);
            elems = new Element[par.ni, par.nj, par.nk];   //indexed from 0 !!
            for (int i = 0; i <= par.ni - 1; i++)
            {
                for (int j = 0; j <= par.nj - 1; j++)
                {
                    for (int k = 0; k <= par.nk - 1; k++)
                    {
                        elems[i, j, k] = new Element(par, nodearray, GQ, i, j, k);
                    }
                }
            }
        }
        public void InitializeAllElements()
        {
            for (int i = 0; i <= par.ni - 1; i++)
            {
                for (int j = 0; j <= par.nj - 1; j++)
                {
                    for (int k = 0; k <= par.nk - 1; k++)
                    {
                        elems[i, j, k].Initialize();
                    }
                }
            }
        }
        public void DoAllElements(LinearSystemSolver linsolver)
        {
            for (int i = 0; i <= par.ni - 1; i++)
            {
                for (int j = 0; j <= par.nj - 1; j++)
                {
                    for (int k = 0; k <= par.nk - 1; k++)
                    {
                        elems[i, j, k].Do(linsolver);
                    }
                }
            }
        }
        public void InsertDataToNodeArray(NodeArray nodearray)
        {
            for (int i = 0; i <= par.ni; i++)   //for each node...
            {
                for (int j = 0; j <= par.nj; j++)
                {
                    for (int k = 0; k <= par.nk; k++)
                    {
                        nodearray.nodes[i, j, k].xyz = GetDeformedPoint(i, j, k);
                    }
                }
            }
        }
        //Get local node number 1..8 from 3 boolean information. See numbering vertexes
        private int GetLocalNodeNumber(bool i_left_point, bool j_left_point, bool k_left_point)
        {
            if (i_left_point)
            {
                if (j_left_point)
                {
                    if (k_left_point) { return 1; } else { return 5; }
                }
                else
                {
                    if (k_left_point) { return 4; } else { return 8; }
                }
            }
            else
            {
                if (j_left_point)
                {
                    if (k_left_point) { return 2; } else { return 6; }
                }
                else
                {
                    if (k_left_point) { return 3; } else { return 7; }
                }
            }
        }
        //Get deformed position for any vertex
        private ThreeVectorStruct GetDeformedPoint(int i_0_ni, int j_0_nj, int k_0_nk)
        {
            bool i_left_point, j_left_point, k_left_point;
            int i_elem, j_elem, k_elem;
            int local_index;  //1,..,8

            //1.0 For each node find an appropriate element (there are more possibilities).
            if (i_0_ni == 0)
            {
                i_left_point = true;
                i_elem = i_0_ni;
            }
            else
            {
                i_left_point = false;
                i_elem = i_0_ni - 1;
            }
            if (j_0_nj == 0)
            {
                j_left_point = true;
                j_elem = j_0_nj;
            }
            else
            {
                j_left_point = false;
                j_elem = j_0_nj - 1;
            }
            if (k_0_nk == 0)
            {
                k_left_point = true;
                k_elem = k_0_nk;
            }
            else
            {
                k_left_point = false;
                k_elem = k_0_nk - 1;
            }
            //2.0 local node number
            local_index = GetLocalNodeNumber(i_left_point, j_left_point, k_left_point);
            return Elems[i_elem, j_elem, k_elem].GetDeformedPoint(local_index);
        }
    }
    //*****************************************************************
```

```
// Container - This is the main class. Note, it does not implement
//             the concrete geometry (look to Beam class)!!
//*************************************************************
public class Container
{
    private NodeArray nodearray;
    private ElementArray elementarray;
    public Parameters par;
    private LinearSystemSolver linsolver;
    public Container()
    {
        par = new Parameters();
        nodearray = new NodeArray(par);
        elementarray = new ElementArray(par, nodearray);
    }
    public void Initialize()
    {
        nodearray.FillGlobalIndexes();
        linsolver = new LinearSystemSolver((int)nodearray.N, par.s_opt);
        elementarray.InitializeAllElements();
    }
    public void SetInitialLocation(int i, int j, int k, double x, double y, double z)
    {
        nodearray.SetInitialLocation(i, j, k, x, y, z);
    }
    public void SetZeroBoundaryCondition(int i, int j, int k)
    {
        nodearray.SetZeroBoundaryCondition(i, j, k);
    }
    private void SolverResultToElementArray()
    {
        Element local_element;
        int index;
        double val;
        for (int i = 0; i <= par.ni - 1; i++)  //for all elems
        {
            for (int j = 0; j <= par.nj - 1; j++)
            {
                for (int k = 0; k <= par.nk - 1; k++)
                {
                    local_element = elementarray.Elems[i, j, k];
                    for (int r = 1; r <= 24; r++)
                    {
                        index = (int)local_element.GetGlobalIndex(r);
                        if (index > -1)
                        {
                            val = linsolver.Solution[index];
                            local_element.Dq[r] = val;
                        }
                    }
                }
            }
        }
    }
    //get |u_middle|
    private double GetConvergenceCriterion()
    {
        Element element = elementarray.Elems[par.ni / 2, 0, 0];
        ThreeVectorStruct TP = element.GetDeformedPoint(3);
        return Math.Abs(TP[3]);
    }
    //The main method - solve method depends also on computed shape
    //which is defined in Beam class
    public virtual void Solve_FEM()
    {
        //Run Initialize ahead!
        double max_decay, deltaqmax;
        for (int k = 1; k <= par.N_FEM_iterations; k++)
        {
            linsolver.Null();
            elementarray.DoAllElements(linsolver);
            linsolver.Solve();
            SolverResultToElementArray();

            deltaqmax = linsolver.Solution.GetMaxNorm();
            max_decay = GetConvergenceCriterion();
            Roller.Stringroller.PutAndPrint(" " + k.ToString() + ". dqmax = " + deltaqmax.ToString()
                + ", max_decay = " + max_decay.ToString());
            if ((max_decay < 0.0) || (max_decay > 2.0))
            {
                throw new Exception("Solution diverged, i.e. max_decay  = " + max_decay.ToString());
            }
            if ((Math.Abs(deltaqmax) < par.convergence_epsilon) && (k > 1))
            {
                Roller.Stringroller.PutAndPrint("Solve stopped due to: max.decay < epsilon.");
                break;
            }
        }
        //The following command returns solved displacements from ElementArray
        //where they are stored during computation to NodeArray.
        //The reason is that in fact all displacements are connected to nodes not elemets.
        elementarray.InsertDataToNodeArray(nodearray);
    }
    //post-processor output
    public ViewSolution GetActualViewSolution()
    {
        ViewSolution vs = new ViewSolution(par);
        for (int i = 0; i <= par.ni; i++)
        {
            for (int j = 0; j <= par.nj; j++)
            {
                for (int k = 0; k <= par.nk; k++)
                {
                    vs.r_q[i, j, k] = nodearray.nodes[i, j, k].xyz;
                }
            }
        }
        return vs;
    }
    public XMath.X3D_3D GetGraphicalOutput(bool initial_positions)
    {
        XMath.X3D_3D p = new XMath.X3D_3D();
        p.Create(par.ni + 1, par.nj + 1, par.nk + 1);
        Node local_node;

        for (int i = 0; i <= par.ni; i++)
        {
            for (int j = 0; j <= par.nj; j++)
            {
                for (int k = 0; k <= par.nk; k++)
                {
                    local_node = nodearray.nodes[i, j, k];
                    if (initial_positions)
                    {
                        p.Put(local_node.xyz_init[1], local_node.xyz_init[2],
                            local_node.xyz_init[3], i + 1, j + 1, k + 1);
                    }
                    else
                    {
                        p.Put(local_node.xyz[1], local_node.xyz[2], local_node.xyz[3], i + 1, j + 1, k + 1);
                    }
                }
            }
        }
        return p;
    }
}
```

The typical example how FELIB library can be used is as follows

```
            Beam beam = new Beam();
            beam.Initialize();
            beam.Solve_FEM();
            X3D_3D result = beam.GetGraphicalOutput(true);  //further post-processing
```

Chapter 13

ALE approach to fluid-solid interaction

This part was written and is maintained by Alexandr Damašek. More details about the author can be found in the Chapter 16.

13.1 Introduction

First, we study the general problem of fluid-solid interaction. We consider viscous incompressible fluid described by Navier-Stokes equations and large displacements of an elastic body. For the description of fluid on moving domain, Arbitrary Lagrangian-Eulerian (ALE) method is used [1], [4] and [5].

After deriving general formulation, we deal with numerical solution, when both elastic structure and fluid are discretized using finite elements. For numerical solution of the coupled problem, domain decomposition algorithm in each time step is applied.

13.2 Transport equation

Let domain $\Omega \subset I\!R^3$ contain moving continuum, described by general equations, independently, if it is a structure (solid) or fluid. Let us describe its motion.

Lagrangian approach: Let us consider a particle at time t_0 and located at point X [2]. Its location $x =$ at time t will depend on the point X and time. Let us use the following notation $x = \varphi(X,t)$. Velocity of the particle is then defined as

$$\mathcal{U}(X,t) = \frac{\partial\varphi}{\partial t}(X,t).$$

Euler approach: Let x be an arbitrary, but fixed point in the domain Ω. We consider fluid particles passing through point x at time t. Denote by $\mathbf{u}(x,t)$ velocity of the fluid particle, passing through point x at time t, i.e.

$$\mathbf{u}(x,t) = \mathcal{U}(X,t) = \frac{\partial\varphi}{\partial t}(X,t), \text{ where } x = \varphi(X,t).$$

For each time t, $\mathbf{u}$ represents vector field defined on Ω.

We consider fluid particles, contained in domain $\Omega(t_0)$ at time t_0 and these particles move into domain $\Omega(t)$ at time t. We assume, that the mapping

$$\begin{aligned} \varphi : \Omega(t_0) &\rightarrow \Omega(t) \\ X &\rightarrow x = \varphi(X,t) \end{aligned}$$

is a dipheomorphism of $\Omega(t_0)$ onto $\Omega(t)$ and $J_\varphi(X,t) > 0$ in $\Omega(t_0)$ (see Fig. 13.1), where the Jacobian J_φ is

$$J_\varphi = \begin{vmatrix} \frac{\partial \varphi_1}{\partial X_1} & \frac{\partial \varphi_2}{\partial X_1} & \frac{\partial \varphi_3}{\partial X_1} \\ \frac{\partial \varphi_1}{\partial X_2} & \frac{\partial \varphi_2}{\partial X_2} & \frac{\partial \varphi_3}{\partial X_2} \\ \frac{\partial \varphi_1}{\partial X_3} & \frac{\partial \varphi_2}{\partial X_3} & \frac{\partial \varphi_3}{\partial X_3} \end{vmatrix}.$$

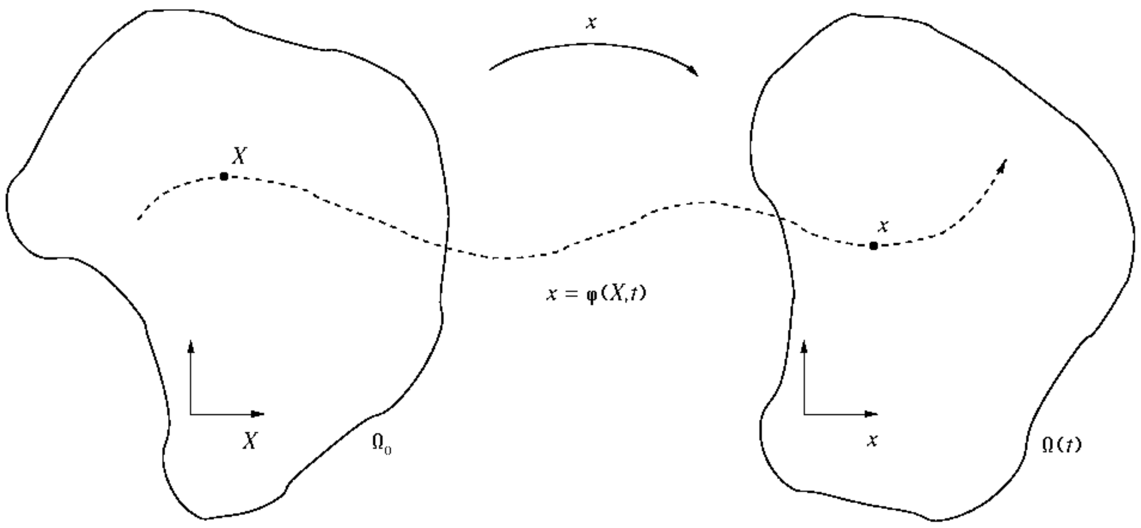


Figure 13.1: Referential configuration $\Omega_0 = \Omega(t_0)$ and actual configuration $\Omega(t)$

We show, that following lemma holds:
Lemma:

$$\frac{\partial J_\varphi(X,t)}{\partial t} = J_\varphi(X,t)\ \mathrm{div}_x\, \mathbf{u}(x,t),\ \text{ where } x = \varphi(X,t).$$

Proof: Let us write for components $\varphi = (\varphi_1(X,t), \varphi_2(X,t), \varphi_3(X,t))$. According to the definition of fluid velocity field

$$\frac{\partial}{\partial t}\varphi(X,t) = \mathbf{u}(\varphi(X,t),t).$$

The matrix determinant is multilinear in columns (and also in rows). If X is hold fixed, then

$$\frac{\partial J_\varphi}{\partial t} = \begin{vmatrix} \frac{\partial}{\partial t}\frac{\partial \varphi_1}{\partial X_1} & \frac{\partial \varphi_2}{\partial X_1} & \frac{\partial \varphi_3}{\partial X_1} \\ \frac{\partial}{\partial t}\frac{\partial \varphi_1}{\partial X_2} & \frac{\partial \varphi_2}{\partial X_2} & \frac{\partial \varphi_3}{\partial X_2} \\ \frac{\partial}{\partial t}\frac{\partial \varphi_1}{\partial X_3} & \frac{\partial \varphi_2}{\partial X_3} & \frac{\partial \varphi_3}{\partial X_3} \end{vmatrix} + \begin{vmatrix} \frac{\partial \varphi_1}{\partial X_1} & \frac{\partial}{\partial t}\frac{\partial \varphi_2}{\partial X_1} & \frac{\partial \varphi_3}{\partial X_1} \\ \frac{\partial \varphi_1}{\partial X_2} & \frac{\partial}{\partial t}\frac{\partial \varphi_2}{\partial X_2} & \frac{\partial \varphi_3}{\partial X_2} \\ \frac{\partial \varphi_1}{\partial X_3} & \frac{\partial}{\partial t}\frac{\partial \varphi_2}{\partial X_3} & \frac{\partial \varphi_3}{\partial X_3} \end{vmatrix} + \begin{vmatrix} \frac{\partial \varphi_1}{\partial X_1} & \frac{\partial \varphi_2}{\partial X_1} & \frac{\partial}{\partial t}\frac{\partial \varphi_3}{\partial X_1} \\ \frac{\partial \varphi_1}{\partial X_2} & \frac{\partial \varphi_2}{\partial X_2} & \frac{\partial}{\partial t}\frac{\partial \varphi_3}{\partial X_2} \\ \frac{\partial \varphi_1}{\partial X_3} & \frac{\partial \varphi_2}{\partial X_3} & \frac{\partial}{\partial t}\frac{\partial \varphi_3}{\partial X_3} \end{vmatrix}$$

Because of the continuity of second derivatives of the mapping φ, we can write

$$\begin{array}{ccccc}
\frac{\partial}{\partial t}\frac{\partial \varphi_1}{\partial X_1} & = & \frac{\partial}{\partial X_1}\frac{\partial \varphi_1}{\partial t} & = & \frac{\partial \mathbf{u}_1}{\partial X_1}, \\
\frac{\partial}{\partial t}\frac{\partial \varphi_1}{\partial X_2} & = & \frac{\partial}{\partial X_2}\frac{\partial \varphi_1}{\partial t} & = & \frac{\partial \mathbf{u}_1}{\partial X_2}, \\
\vdots & & & & \vdots \\
\frac{\partial}{\partial t}\frac{\partial \varphi_3}{\partial X_3} & = & \frac{\partial}{\partial X_3}\frac{\partial \varphi_3}{\partial t} & = & \frac{\partial \mathbf{u}_3}{\partial X_3}.
\end{array}$$

Components $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$ of velocity $\mathbf{u}$ in this expression are functions of X_1, X_2, X_3 by means of $\varphi(X, t)$, thus

$$\begin{array}{ccccccc}
\frac{\partial \mathbf{u}_1}{\partial X_1} & = & \frac{\partial \mathbf{u}_1}{\partial x_1}\frac{\partial \varphi_1}{\partial X_1} & + & \frac{\partial \mathbf{u}_1}{\partial x_2}\frac{\partial \varphi_2}{\partial X_1} & + & \frac{\partial \mathbf{u}_1}{\partial x_3}\frac{\partial \varphi_3}{\partial X_1}, \\
\frac{\partial \mathbf{u}_1}{\partial X_2} & = & \frac{\partial \mathbf{u}_1}{\partial x_1}\frac{\partial \varphi_1}{\partial X_2} & + & \frac{\partial \mathbf{u}_1}{\partial x_2}\frac{\partial \varphi_2}{\partial X_2} & + & \frac{\partial \mathbf{u}_1}{\partial x_3}\frac{\partial \varphi_3}{\partial X_2}, \\
\vdots & & & & \vdots & & \\
\frac{\partial \mathbf{u}_3}{\partial X_3} & = & \frac{\partial \mathbf{u}_3}{\partial x_1}\frac{\partial \varphi_1}{\partial X_3} & + & \frac{\partial \mathbf{u}_3}{\partial x_2}\frac{\partial \varphi_2}{\partial X_3} & + & \frac{\partial \mathbf{u}_3}{\partial x_3}\frac{\partial \varphi_3}{\partial X_3}.
\end{array}$$

After substitution of these formulas into an expression for $\frac{\partial J}{\partial t}$ we get

$$\frac{\partial \mathbf{u}_1}{\partial x_1} J_{\varphi}(X,t) + \frac{\partial \mathbf{u}_2}{\partial x_2} J_{\varphi}(X,t) + \frac{\partial \mathbf{u}_3}{\partial x_3} J_{\varphi}(X,t) = \mathrm{div}_x \mathbf{u} \cdot J_{\varphi}(X,t),$$

because remaining determinants are zero due to linearly dependent columns.

In the next steps, the following *transport equation* is important. Let F be a scalar function. The amount of a quantity in volume $\Omega(t)$ is represented by $G = \int_{\Omega(t)} F(x,t)\mathrm{d}x$. Let us deal with rate of change of the quantity G, i.e. with the expression

$$\frac{\mathrm{d}G}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega(t)} F(x,t)\mathrm{d}x.$$

Transport equation

Let the mapping

$$\varphi : X \quad \rightarrow \quad x = \varphi(X,t)$$

of the domain $\Omega(t_0)$ on $\Omega(t)$ be a dipheomorphism and $J_{\varphi}(X,t) > 0$ on $\Omega(t_0)$ and let

$$\mathbf{u}(\varphi(X,t),t) = \frac{\partial \varphi}{\partial t}(X,t) \text{ pro } X \in \Omega(t_0), t \in [t_0, T].$$

Let F be a real function with continuous partial derivatives of the first order for $x \in \Omega(t)$, $t \in [t_0, T]$. Then

$$\boxed{\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega(t)} F(x,t)\mathrm{d}x = \int_{\Omega(t)} \left[\frac{\partial F}{\partial t}(x,t) + \mathbf{u}(x,t)\cdot\nabla_x F(x,t) + F(x,t)\mathrm{div}_x\mathbf{u}(x,t)\right]\mathrm{d}x. \quad (13.1)}$$

Proof: According to the substitution theorem (A3) in Appendix we transform the integral to an integration over a fixed domain

$$\int_{\Omega(t)} F(x,t)\ \mathrm{d}x = \int_{\Omega(t_0)} F(\varphi(X,t),t) J_{\varphi}(X,t)\ \mathrm{d}X.$$

Because the domain of integration is fixed, theorem about the derivative of a parameter-dependent integral can be used (Theorem (A5) in Appendix)

$$\begin{aligned} \tfrac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega(t)} F(x,t)\ \mathrm{d}x &= \int_{\Omega_0}\Big[\left(\tfrac{\partial F(\varphi(X,t),t)}{\partial t} + \tfrac{\partial F(\varphi(X,t),t)}{\partial x_i}\tfrac{\partial \varphi_i(X,t)}{\partial t}\right)\cdot J_{\varphi}(X,t) \\ &+ F(\varphi(X,t),t)\tfrac{\partial J_{\varphi}}{\partial t}(X,t)\Big]\ \mathrm{d}X. \end{aligned}$$

If we substitute for $\frac{\partial J_{\varphi}}{\partial t}(X,t)$ from the previous lemma and for $\frac{\partial \varphi}{\partial t}(X,t)$, we get

$$\begin{aligned} \tfrac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega(t)} F(x,t)\ \mathrm{d}x &= \int_{\Omega_0}\Big[\tfrac{\partial F(\varphi(X,t),t)}{\partial t} + \tfrac{\partial F(\varphi(X,t),t)}{\partial x_i}\mathbf{u}_i(\varphi(X,t),t) \\ &+ F(\varphi(X,t),t)\ \mathrm{div}_x\mathbf{u}(\varphi(X,t),t)\Big]J_{\varphi}(X,t)\ \mathrm{d}X. \end{aligned}$$

Now we use the substitution theorem again and transform the integration to the domain $\Omega(t)$, from which the required relation (13.1) is obtained.

13.3 Derivation of the ALE-method

Let $\Omega(t) = \Omega^s(t) \cup \Omega^f(t)$ be a connected domain, where symbol s stands for solids, symbol f for fluids. For general continuum, which is characterized by the velocity of particles $\mathbf{u}(x,t)$, density $\rho(x,t)$, stress $\boldsymbol{\sigma}(x,t)$ and volumetric load $f(x,t)$ acts on it, we formulate according to [5] in integral form

1. mass conservation
2. conservation of momentum.

After that, we introduce special expression for stress tensor on each domain $\Omega^f(t)$, $\Omega^s(t)$ respectively.

13.3.1 Mass conservation

Mass conservation Let at time t_0 particles of the fluid occupy a domain $\Omega(t_0)$, at time t occupy $\Omega(t)$. *Mass of the part of the fluid contained in domain $\Omega(t)$ does not depend on time, i.e.*

$$\frac{\mathrm{d}m(\Omega(t))}{\mathrm{d}t} = 0.$$

Assume, that there exists *density*, i.e. a function $\rho(x,t)$ such, as for an arbitrary $\Omega(t)$

$$m(\Omega(t)) = \int_{\Omega(t)} \rho(x,t)\mathrm{d}x.$$

Applying transport equation to the function $F = \rho$ we obtain

$$\int_{\Omega(t)} \left[\frac{\partial \rho(x,t)}{\partial t} + \mathbf{u}_i(x,t) \cdot \frac{\partial \rho(x,t)}{\partial x_i} + \rho(x,t)\mathrm{div}_x \mathbf{u}(x,t) \right] \mathrm{d}x = 0,\ t \in [t_0, T].$$

Because the equation is valid for an arbitrary volume $W \subset \Omega(t)$, it is possible to consider an arbitrary fixed point x_0, $x_0 \in W$ for different W. Passing to the limit for $\mu(W) \to 0$, $x_0 \in W$ under the assumption of continuity of integrated function we get the equation

$$\frac{\partial \rho}{\partial t} + \mathrm{div}(\rho \mathbf{u}) = 0$$

satisfied in an arbitrary point (x_0, t). Multiplying this equation by a test function $\hat{q}$ and integrating over an arbitrary volume $\Omega(t)$ we get the formulation of mass conservation in continuum $\Omega(t)$

$$\boxed{\int_{\Omega(t)} \left(\frac{\partial \rho}{\partial t}|_x + \mathrm{div}_x(\rho \mathbf{u}) \right) \hat{q} = 0,}$$

where $\hat{q}$ is arbitrary, $\hat{q}:\ \Omega(t) \longrightarrow I\!R$.

13.3.2 Conservation of momentum

Conservation of momentum There act surface and volumetric loads on continuum. Let $T(x,t,n(x))$ represent stress vector at point x in direction of outward unit normal $n(x)$ to the boundary $\partial\Omega(t)$. Volumetric loads are then given by the relation $F_V = \int_{\Omega(t)} \rho(x,t) f(x,t)\mathrm{d}x$, surface loads by $F_S = \int_{\partial\Omega(t)} T(x,t,n(x))\mathrm{d}S$. Momentum is given

$$H_{\Omega(t)} = \int_{\Omega(t)} \rho(x,t)\mathbf{u}(x,t)\mathrm{d}x,$$

for total load we have

$$F_{\Omega(t)} = \int_{\Omega(t)} \rho(x,t) f(x,t)\mathrm{d}x + \int_{\partial\Omega(t)} T(x,t,n(x))\mathrm{d}S.$$

Change of momentum in a given fixed volume of continuum occupied by identical particles, which are contained in domain $\Omega(t)$ at time t, equals to the load, acting on $\Omega(t)$, i.e.

$$\frac{\mathrm{d}H_{\Omega(t)}}{\mathrm{d}t} = F_{\Omega(t)}$$

After substituting into this relation, using transport equation for $\rho\mathbf{u}_i$, we get subsequently

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega(t)}\rho(x,t)\mathbf{u}_i(x,t)\mathrm{d}x=\int_{\Omega(t)}\left[\frac{\partial}{\partial t}\left(\rho(x,t)\mathbf{u}_i(x,t)\right)+\mathrm{div}_x\left(\rho(x,t)\mathbf{u}_i(x,t)\mathbf{u}(x,t)\right)\right]\mathrm{d}x.$$

Thus

$$\int_{\Omega(t)}\left[\frac{\partial}{\partial t}\left(\rho(x,t)\mathbf{u}_i(x,t)\right)+\mathrm{div}_x\left(\rho\mathbf{u}_i\mathbf{u}\right)\right]\mathrm{d}x=\int_{\Omega(t)}\rho(x,t)f_i(x,t)\mathrm{d}x+\int_{\partial\Omega(t)}T_i(x,t,n(x))\mathrm{d}S.$$

After substituting the relation $T_i(x,t,n(x))=\sigma_{ij}(x,t)n_j(x)$ according to Green's formula (theorem (A1) in Appendix) we get the following equation

$$\int_{\Omega(t)}\left[\frac{\partial}{\partial t}\left(\rho(x,t)\mathbf{u}_i(x,t)\right)+\mathrm{div}_x\left(\rho\mathbf{u}_i\mathbf{u}\right)\right]\mathrm{d}x=\int_{\Omega(t)}\rho(x,t)f_i(x,t)\mathrm{d}x+\int_{\Omega(t)}\frac{\partial\sigma_{ij}}{\partial x_j}\mathrm{d}x,$$

which leads to the classical equation

$$\frac{\partial}{\partial t}\left(\rho(x,t)\mathbf{u}_i(x,t)\right)+\mathrm{div}_x\left(\rho\mathbf{u}_i\mathbf{u}\right)=\rho(x,t)f_i(x,t)+\frac{\partial\sigma_{ij}}{\partial x_j},$$

which can be written in the weak sense as

$$\int_{\Omega(t)}\left[\frac{\partial}{\partial t}\left(\rho(x,t)\mathbf{u}_i(x,t)\right)\hat{U}_i+\mathrm{div}_x\left(\rho\mathbf{u}_i\mathbf{u}\right)\hat{U}_i\right]\mathrm{d}x=\int_{\Omega(t)}\rho(x,t)f_i(x,t)\hat{U}_i\mathrm{d}x+\int_{\Omega(t)}\frac{\partial\sigma_{ij}}{\partial x_j}\hat{U}_i\mathrm{dx}.$$

Reusing Green's formula we have

$$\boxed{\begin{aligned}&\textstyle\int_{\Omega(t)}\left[\frac{\partial}{\partial t}\left(\rho(x,t)\mathbf{u}_i(x,t)\right)\hat{U}_i+\mathrm{div}_x\left(\rho\mathbf{u}_i\mathbf{u}\right)\hat{U}_i\right]\mathrm{d}x\\&=\textstyle\int_{\Omega(t)}\rho(x,t)f_i(x,t)\hat{U}_i\mathrm{d}x-\int_{\Omega(t)}\sigma_{ij}\frac{\partial\hat{U}_i}{\partial x_j}\mathrm{d}x+\int_{\partial\Omega(t)}\sigma_{ij}n_j\hat{U}_i\mathrm{dS},\end{aligned}}$$

where $\hat{U}$ is arbitrary, $\hat{U}:\ \Omega(t)\longrightarrow I\!R^3$.

13.4 Geometry of the deformation

The problem of nonlinear elasticity is to find an equilibrium position of an elastic body occupying *referential configuration* Ω, when external forces do not act on it, where Ω is a bounded open connected subset $I\!R^3$ with Lipschitzian boundary. If there act external forces on the body, body occupies deformed configuration $\varphi(\Omega)$, characterized by the mapping $\varphi:\Omega\to I\!R^3$ which does not change orientation and must be one-to-one on Ω. Such mappings φ are called deformations. Concerning their geometrical properties, it can be shown, that change of volumes, surfaces and lengths of corresponding deformations φ are given subsequently by scalar $\det\nabla\varphi$, matrix $\mathrm{Cof}\nabla\varphi$, and right Cauchy-Green deformation tensor $\mathbf{C}=\nabla\varphi^T\nabla\varphi$. Here $\nabla\varphi$ stands for Jacobi matrix

$$\nabla\varphi=\begin{pmatrix}\frac{\partial\varphi_1}{\partial X_1}&\frac{\partial\varphi_1}{\partial X_2}&\frac{\partial\varphi_1}{\partial X_3}\\\frac{\partial\varphi_2}{\partial X_1}&\frac{\partial\varphi_2}{\partial X_2}&\frac{\partial\varphi_2}{\partial X_3}\\\frac{\partial\varphi_3}{\partial X_1}&\frac{\partial\varphi_3}{\partial X_2}&\frac{\partial\varphi_3}{\partial X_3}\end{pmatrix}.$$

Further, it can be shown, that Green-St Venant deformation tensor $\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I})$ asociated with the deformation φ measures discrepancy between φ and deformation of solid (which corresponds $\mathbf{C} = \mathbf{I}$), see [3].

Let $\mathbf{A}$ be a matrix of order n. For each pair (i, j) of indices let $\mathbf{A}'_{\mathrm{ij}}$ be a matrix of order $(n-1)$ obtained by deleting of $i-$th row and $j-$th column of matrix $\mathbf{A}$. Scalar

$$d_{ij} := (-1)^{i+j}\det\mathbf{A}'_{\mathrm{ij}}$$

is called (i, j)*-th coffactor* of matrix $\mathbf{A}$ and matrix

$$\mathrm{Cof}\mathbf{A} := (d_{ij})$$

coffactor matrix of matrix $\mathbf{A}$.

Well-known formulas for determinant expansion are equivalent to the relations

$$\mathbf{A}(\mathrm{Cof}\mathbf{A})^{\mathrm{T}} = (\mathrm{Cof}\mathbf{A})^{\mathrm{T}}\mathbf{A} = (\det\mathbf{A})\mathbf{I}.$$

If matrix $\mathbf{A}$ is invertible,

$$\mathrm{Cof}\mathbf{A} = (\det\mathbf{A})\mathbf{A}^{-\mathrm{T}},$$

where $\mathbf{A}^{-\mathrm{T}} = (\mathbf{A}^{-1})^{\mathrm{T}}$. Alternatively, in case $n = 3$

$$\mathrm{Cof}\mathbf{A} = \begin{pmatrix} a_{22}a_{33} - a_{23}a_{32} & a_{23}a_{31} - a_{21}a_{33} & a_{21}a_{32} - a_{22}a_{31} \\ a_{32}a_{13} - a_{33}a_{12} & a_{33}a_{11} - a_{31}a_{13} & a_{31}a_{12} - a_{32}a_{11} \\ a_{12}a_{23} - a_{13}a_{22} & a_{13}a_{21} - a_{11}a_{23} & a_{11}a_{22} - a_{12}a_{21} \end{pmatrix}$$

or equally

$$(\mathrm{Cof}\mathbf{A})_{\mathrm{ij}} = \frac{1}{2}\varepsilon_{\mathrm{mni}}\varepsilon_{\mathrm{pqj}}\mathrm{a}_{\mathrm{mp}}\mathrm{a}_{\mathrm{nq}},$$

where the summation is over indices appearing twice.

13.5 Piola transform

Assume φ is a deformation, which is one-to-one on Ω, i.e. the matrix $\nabla\varphi$ is invertible at each point of its referential configuration (see Fig. 13.2). If $\mathbf{T}^{\varphi}$ is a tensor defined at point $x = X^{\varphi} = \varphi(X)$ of the deformed configuration, we then assign to the tensor $\mathbf{T}^{\varphi}(X^{\varphi})$ a tensor $\mathbf{T}(X)$ defined at point X of the referential configuration according to the relation

$$\mathbf{T}(X) := (\det\nabla\varphi(X))\mathbf{T}^{\varphi}(X^{\varphi})\nabla\varphi(X)^{-T} = \mathbf{T}^{\varphi}(X^{\varphi})\mathrm{Cof}\varphi(X), \quad x = X^{\varphi} = \varphi(X).$$

Theorem: (properties of the Piola transform)

Let $\mathbf{T} : \Omega \to \mathbf{M}^3$ *be Piola transform* $\mathbf{T}^{\varphi} : \Omega^{\varphi} \to \mathbf{M}^3$*, where* $\mathbf{M}^3$ *denotes regular matrices of order 3. Then*

$$\mathrm{div}\mathbf{T}(X) = (\det\nabla\varphi(X))\mathrm{div}\varphi\mathbf{T}^{\varphi}(X^{\varphi}) \text{ for all } X^{\varphi} = \varphi(X), X \in \Omega.$$

Proof: Key step is to prove the *Piola identity*

$$\mathrm{div}\left((\det\nabla\varphi)\nabla\varphi^{-T}\right) = \mathrm{div}\mathrm{Cof}\nabla\varphi = 0,$$

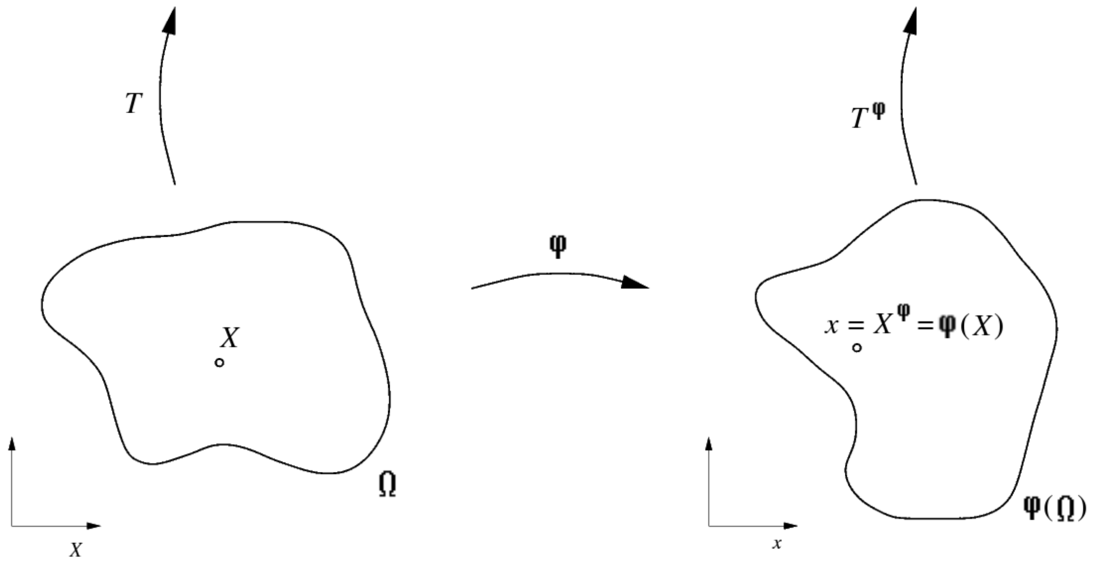


Figure 13.2: Piola transform of a tensor from the deformed configuration to the referential

which will be shown first. Considering indices over modulo 3, the matrix $\mathrm{Cof}\nabla\varphi$ is given

$$(\mathrm{Cof}\nabla\varphi)_{ij} = \partial_{j+1}\varphi_{i+1}\partial_{j+2}\varphi_{i+2} - \partial_{j+2}\varphi_{i+1}\partial_{j+1}\varphi_{i+2} \text{ (no summation)}.$$

Direct calculation gives

$$\begin{aligned}\partial_j\left((\det\nabla\varphi)\nabla\varphi^{-T}\right)_{ij} &= \partial_j(\mathrm{Cof}\nabla\varphi)_{ij}\\ &= \sum_{j=1}^{3}\left[\left(\frac{\partial^2\varphi_{i+1}}{\partial X_j\partial X_{j+1}}\partial_{j+2}\varphi_{i+2} - \frac{\partial^2\varphi_{i+1}}{\partial X_j\partial X_{j+2}}\partial_{j+1}\varphi_{i+2}\right)\right.\\ &+ \left.\left(\partial_{j+1}\varphi_{i+1}\frac{\partial^2\varphi_{i+2}}{\partial X_j\partial X_{j+2}} - \partial_{j+2}\varphi_{i+1}\frac{\partial^2\varphi_{i+2}}{\partial X_j\partial X_{j+1}}\right)\right] = 0.\end{aligned}$$

Then from the formulas

$$\mathbf{T}_{ij}(X) = (\det\nabla\varphi(X))\mathbf{T}^{\varphi}_{ik}(X^{\varphi})(\nabla\varphi(X)^{-T})_{kj}$$

follows

$$\begin{aligned}\partial_j\mathbf{T}_{ij}(X) &= (\det\nabla\varphi(X))\partial_j T^{\varphi}_{ik}(X^{\varphi})(\nabla\varphi(X)^{-T})_{kj} + T^{\varphi}_{ik}(X^{\varphi})\partial_j\left((\det\nabla\varphi(X))\nabla\varphi(X)^{-T}\right)_{kj}\\ &= (\det\nabla\varphi(X))\partial_j T^{\varphi}_{ik}(X^{\varphi})(\nabla\varphi(X)^{-T})_{kj}\end{aligned}$$

because second term equals zero as a consequence of the Piola identity. Further, according to the chain rule (Theorem (A2) in the Appendix)

$$\partial_j\mathbf{T}^{\varphi}_{ik}(X) = \partial^{\varphi}_l\mathbf{T}^{\varphi}_{ik}(\varphi(X))\partial_j\varphi_l(X) = \partial^{\varphi}_l\mathbf{T}^{\varphi}_{ik}(X^{\varphi})(\nabla\varphi(X))_{lj},$$

and relation between $\mathrm{div}\mathbf{T}(X)$ and $\mathrm{div}\varphi\mathbf{T}^{\varphi}(X^{\varphi})$ follows from the fact, that

$$(\nabla\varphi(X))_{li}(\nabla\varphi(X)^{-T})_{ki} = \delta_{lk},$$

i.e.

$$\partial_j \mathbf{T}_{ij}(X) = (\det\nabla\varphi(X))\partial_l^{\varphi}\mathbf{T}_{ik}^{\varphi}(X^{\varphi})(\nabla\varphi(X))_{lj}(\nabla\varphi(X)^{-T})_{kj} = (\det\nabla\varphi(X))\partial_k^{\varphi}\mathbf{T}_{ik}^{\varphi}(X^{\varphi}).$$

Similar formula can be derived, if $T = A_i$ is a vector

$$\partial_j A_j(X) = (\det\nabla\varphi(X))\partial_k^{\varphi} A_k^{\varphi}(X^{\varphi}), \tag{13.2}$$

where

$$A_j(X) = (\det\nabla\varphi(X))A_k^{\varphi}(X^{\varphi})\nabla\varphi(X)_{kj}^{-T}.$$

13.6 Application to the equations of conservation

First, we derive formulas for transporting of equations of conservation laws to the referential configuration Ω_0.

We introduce a few new notations. Function $\hat{q}, \hat{U}$ defined in $\Omega(t)$ can be transported to Ω_0 using the following formula. Define the functions

$$\begin{aligned} \hat{q}^{\varphi}: \ \Omega_0 \longrightarrow I\!R, \ \hat{q}^{\varphi}(X): \ &= \ \hat{q}\circ\varphi(X) \\ \hat{U}^{\varphi}: \ \Omega_0 \longrightarrow I\!R^3, \ \hat{U}^{\varphi}(X): \ &= \ \hat{U}\circ\varphi(X). \end{aligned}$$

For simplicity we write $\hat{q}$ and $\hat{U}$ instead of $\hat{q}^{\varphi}(X)$ and $\hat{U}^{\varphi}(X)$. Again, for simplicity we write $X \longrightarrow x(X,t)$ instead of $X \longrightarrow \varphi(X,t)$.

Choice of the configuration Ω_0 and a mapping φ can be arbitrary, from what comes the name *Arbitrary Lagrangian-Eulerian method* for the resulting equations. In order to simplify following computations, we choose for the structural part material configuration as referential. In other words, the point $x(X,t)$ in structure Ω^s corresponds to the current position of a material point, which was located at point X at time t_0. From that then follows, that *velocity of the configuration* (or *velocity of the grid*)

$$\mathbf{u}_G(x) := \frac{\partial x}{\partial t}(X,t)$$

is always equal to the *velocity of the structure* at an arbitrary point $x \in \Omega^s$.

Next, we denote

$$\begin{aligned} J_{\varphi} \ &= \ \det\left(\nabla\varphi(X)\right), \\ (\nabla\varphi)^{-T} \ &= \ (\nabla\varphi(X))^{-T}, \\ \rho_0 \ &= \ \rho\cdot J_{\varphi}. \end{aligned}$$

According to the chain rule and substitution theorem

$$\int_{\Omega(t)} \sigma_{ij}(x)\frac{\partial \hat{U}_i(x,t)}{\partial x_j}\mathrm{d}x = \int_{\Omega_0} \sigma_{ij}(X)\frac{\partial \hat{U}_i(x(X,t),t)}{\partial X_k}\cdot\frac{\partial X_k}{\partial x_j}(\det\nabla\varphi(X))\mathrm{d}X.$$

According to the theorem on local dipheomorphism (Theorem (A4) in the Appendix) $\nabla\varphi(X)\cdot(\nabla\varphi(X))^{-T} = \mathrm{I}$, we can write this relation

$$\boxed{\int_{\Omega(t)} \sigma_{ij}(x)\frac{\partial \hat{U}_i(x,t)}{\partial x_j}\mathrm{d}x = \int_{\Omega_0} \sigma_{ij}(X)\frac{\partial \hat{U}_i(x(X,t),t)}{\partial X_k}\cdot(\nabla\varphi)^{-T}J_{\varphi}(X)\mathrm{d}X.} \tag{13.3}$$

Next, using substitution theorem and Piola relation according to (13.2) (when $\Omega(t) = \varphi(\Omega_0)$, $x = \varphi(X,t)$, t is now a fixed parameter)

$$\int_{\Omega(t)} \frac{\partial A_i^{\varphi}}{\partial X_i^{\varphi}}(X^{\varphi})\hat{q}(X^{\varphi})\mathrm{d}X^{\varphi} = \int_{\Omega_0} \frac{\partial A_i^{\varphi}}{\partial X_i^{\varphi}}(\varphi(X))(\det\nabla\varphi(X))\hat{q}(X)\mathrm{d}X =$$
$$\int_{\Omega_0} \mathrm{div}_{\mathrm{X}} \left((\det\nabla\varphi(X))A^{\varphi}(X^{\varphi})(\nabla\varphi(X))^{-T}\right)\hat{q}(X)\mathrm{d}X,$$

which, in a more simple way written for $A_i := A_i\mathbf{u}$, gives

$$\boxed{\int_{\Omega(t)} \mathrm{div}_x(A_i\mathbf{u}(x,t)) \cdot \hat{q}(x)\mathrm{d}x = \int_{\Omega_0} \mathrm{div}_{\mathrm{X}}\left(J_{\varphi}A_i\mathbf{u}(X,t)(\nabla\varphi)^{-T}\right) \cdot \hat{q}(X)\mathrm{d}X.} \quad (13.4)$$

Because

$$\begin{aligned}
\mathrm{div}_x\left(F\cdot\mathbf{u}\right)(x,t) &= \frac{\partial}{\partial x_i}\left(F\cdot\mathbf{u}_i(x,t)\right)\\
&= \mathbf{u}_i(x,t)\frac{\partial F}{\partial x_i}(x,t) + F(x,t)\frac{\partial \mathbf{u}_i}{\partial x_i}(x,t)\\
&= \mathbf{u}(x,t)\cdot\nabla_x F(x,t) + F(x,t)\mathrm{div}_x\mathbf{u}(x,t),
\end{aligned}$$

classical transport equation can be also written in the form

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega(t)} F(x,t)\mathrm{d}x = \int_{\Omega(t)}\left[\frac{\partial F}{\partial t}(x,t) + \mathrm{div}_x\left(F\cdot\mathbf{u}\right)(x,t)\right]\mathrm{d}x.$$

For the case of a vector function, by putting $F_i(x,t) = A_i(x,t)\hat{q}(x)$, we have

$$\int_{\Omega(t)}\left[\frac{\partial A_i(x,t)}{\partial t} + \mathrm{div}_x\left(A_i(x,t)\mathbf{u}(x,t)\right)\right]\hat{q}(x)\mathrm{d}x = \frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega(t)} A_i(x,t)\hat{q}(x)\mathrm{d}x.$$

After modification

$$\int_{\Omega(t)}\frac{\partial A_i(x,t)}{\partial t}\hat{q}(x)\mathrm{d}x + \int_{\Omega(t)}\mathrm{div}_x\left(A_i\mathbf{u}(x,t)\right)\hat{q}(x)\mathrm{d}x = \frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega_0} A_iJ_{\varphi}(X,t)\hat{q}(X)\mathrm{d}X.$$

According to (13.4) the second term equals

$$\int_{\Omega(t)}\mathrm{div}_x\left(A_i\mathbf{u}(x,t)\right)\hat{q}(x)\mathrm{d}x = \int_{\Omega_0}\mathrm{div}_X\left(J_{\varphi}A_i\mathbf{u}(X,t)(\nabla\varphi)^{-T}\right)\hat{q}(X)\mathrm{d}\Omega_0,$$

from where

$$\int_{\Omega(t)}\frac{\partial A_i(x,t)}{\partial t}\hat{q}(x)\mathrm{d}\Omega = \int_{\Omega_0}\left[\frac{\partial\left(A_iJ_{\varphi}(X,t)\right)}{\partial t}\,|_X - \mathrm{div}_X\left(J_{\varphi}A_i\mathbf{u}(X,t)(\nabla\varphi)^{-T}\right)\right]\hat{q}(X)\mathrm{d}\Omega_0,$$

where $\mathbf{u}(X,t) = \mathbf{u}_G$ (the grid velocity), then

$$\boxed{\int_{\Omega(t)}\frac{\partial A_i(x,t)}{\partial t}\hat{q}(x)\mathrm{d}\Omega = \int_{\Omega_0}\left[\frac{\partial\left(A_iJ_{\varphi}(X,t)\right)}{\partial t}\,|_X - \mathrm{div}_X\left(J_{\varphi}A_i\mathbf{u}_G(\nabla\varphi)^{-T}\right)\right]\hat{q}(X)\mathrm{d}\Omega_0.} \quad (13.5)$$

Applying of equations (13.3 - 13.5) to conservation of mass and momentum enables us to transport these relations to the fixed configuration Ω_0. **u**sing equations (13.4) and (13.5) for $A_i = \rho$ we get

$$\boxed{\int_{\Omega_0}\left(\frac{\partial\rho_0}{\partial t}|_X + \mathrm{div}_X(\rho_0(\mathbf{u}-\mathbf{u}_G)\cdot(\nabla\varphi)^{-T}\right)\hat{q} = 0,\ \forall\hat{q}\in Q,\quad (\mathrm{mass}).}$$

From equations (13.3), (13.4) and (13.5) follows for $A_i = \rho\mathbf{u}_i$

$$\boxed{\begin{aligned}&\int_{\Omega_0}\left(\frac{\partial\rho_0\mathbf{u}_i}{\partial t}|_X + \mathrm{div}_X(\rho_0\mathbf{u}_i(\mathbf{u}-\mathbf{u}_G)\cdot(\nabla\varphi)^{-T})\right)\hat{U}_i\mathrm{d}X + \int_{\Omega_0}(J\varphi\cdot\sigma\cdot(\nabla\varphi)^{-T})_{ij}\frac{\partial\hat{U}_i}{\partial X_j}\mathrm{d}X\\ &= \int_{\Omega(t)} f\cdot\hat{U}\mathrm{d}x + \int_{\partial\Omega(t)} g\cdot\hat{U}\mathrm{d}x,\ \forall\hat{U}\in V,\quad (\mathrm{momentum}).\end{aligned}}$$

Up to now we considered all functions sufficiently smooth; in this weak formulation we define now these spaces of test functions

$$\begin{aligned}Q &= \{\hat{q}:\Omega_0 \longrightarrow I\!R, \hat{q}\in L^2(\Omega_0)\},\\ V &= \{\hat{U}:\Omega_0 \longrightarrow I\!R^3, \hat{U}\in H^1(\Omega_0, I\!R^3)\}.\end{aligned}$$

13.7 Decomposition of fluid and structure

We have to add constitutive equations for the dependence of *stresses on deformations* to the equations of conservation. Now, it is necessary to distinguish between fluid and structure. For doing this, we decompose spaces of test functions into 'fluid' test functions, acting in the fluid domain and 'structural' – acting on the complement – structure

$$V = V^s \oplus V^f,\ Q = Q^s \oplus Q^f.$$

Spaces of functions for fluid and structure are defined as follows

$$\begin{aligned}V^f &= \{\hat{U}\in H^1(\Omega_0, I\!R^3), \hat{U}_{|\Omega_0^s} = 0\},\\ Q^f &= \{\hat{q}\in L^2(\Omega_0), \hat{q}_{|\Omega_0^s} = 0\},\\ V^s &= \{\hat{U}\in H^1(\Omega_0, I\!R^3)\},\\ Q^s &= \{\hat{q}\in L^2(\Omega_0), \hat{q}_{|\Omega_0^f} = 0\}.\end{aligned}$$

In this construction, 'fluid' test functions are zero on the interface. In the opposite, 'structural' test functions are nonzero on the interface and prolongate continuously to the fluid.

13.7.1 Fluid problem

We obtain the fluid problem by restricting conservation laws to the test functions from Q^f and V^f only. As this functions are zero in Ω_0^s, we get

$$\int_{\Omega_0^f}\left(\frac{\partial\rho_0}{\partial t}|_X + \mathrm{div}_X(\rho_0(\mathbf{u}-\mathbf{u}_G^f)\cdot\nabla\varphi^{-T})\right)\hat{q}\mathrm{d}X = 0,\ \forall\hat{q}:\Omega_0^f \longrightarrow I\!R,$$

$$\int_{\Omega_0^f}\left(\frac{\partial(\rho_0\mathbf{u}_i)}{\partial t}|_X + \mathrm{div}_X\rho_0\mathbf{u}_i(\mathbf{u}-\mathbf{u}_G^f)\cdot\nabla\varphi^{-T})\right)\hat{U}_i\mathrm{d}X + \int_{\Omega_0^f}(J_\varphi\,\sigma\cdot\nabla\varphi^{-T})_{ij}\frac{\partial\hat{U}_i}{\partial X_j}\mathrm{d}X$$
$$= \int_{\Omega_0^f} f\cdot\hat{U}\mathrm{d}X + \int_{\partial\Omega_0^f-\Gamma_{I_0}} g\cdot\hat{U}\mathrm{d}X,\ \forall\hat{U}\in V^f$$

It is easier to compute time derivatives in these integrals on a fixed configuration Ω_0, and convective terms in actual configuration $\Omega(t)$. It is therefore necessary to transport convective terms, and the terms containing stress, back to the actual configuration and use the relations for transport from the previous chapter. Considering viscous incompressible fluid, where the dependence of fluid stress on deformation velocity tensor is given by $\sigma_{ij}^f = -p\mathrm{I} + \mu(\partial_j\mathbf{u}_i + \partial_i\mathbf{u}_j)$, and relation $\nabla\varphi(\nabla\varphi)^{-T} = \mathrm{I}$, we get

$$\int_{\Omega^f(t)}\left(\frac{1}{J_\varphi}\frac{\partial(J_\varphi\rho)}{\partial t}|_X + \mathrm{div}_x(\rho(\mathbf{u}-\mathbf{u}_G^f))\right)\hat{q}\mathrm{d}x = 0,\ \forall\hat{q}:\Omega_0^f\longrightarrow I\!R, \tag{13.6}$$

$$\int_{\Omega^f(t)}\left(\frac{1}{J_\varphi}\frac{\partial(J_\varphi\rho\mathbf{u}_i)}{\partial t}|_X + \mathrm{div}_x(\rho\mathbf{u}_i(\mathbf{u}-\mathbf{u}_G^f))\right)\hat{U}_i\mathrm{d}x \tag{13.7}$$
$$+\int_{\Omega^f(t)}(-p\mathrm{I}+\mu(\partial_j\mathbf{u}_i+\partial_i\mathbf{u}_j))\frac{\partial\hat{U}_i}{\partial x_j}\mathrm{d}x = \int_{\Omega^f(t)} f\cdot\hat{U}\mathrm{d}x + \int_{\partial\Omega^f(t)} g\cdot\hat{U}\mathrm{d}x,\ \forall\hat{U}\in V^f$$

In this formulation we recognize description of the ALE *(Arbitrary Lagrangian-Eulerian)* method for Navier-Stokes equations. These equations are completely determining completely the state of fluid, when appropriate boundary conditions, on the exterior part $\partial\Omega^f(t)-\Gamma_I(t)$ and on the interface $\Gamma_I(t)$, are given. As test functions for the fluid vanish on the interface, velocity of the fluid can be prescribed independently on conservation laws by prescribing the kinematic continuity condition

$$\mathbf{u}_{\Gamma_I}^f = \mathbf{u}_{\Gamma_I}^s, \tag{13.8}$$

where the equation is considered in the sense of traces.

We remark, that in ALE formulation of fluid problem (13.6) and (13.7) the grid velocity appears

$$\mathbf{u}_G^f = \frac{\partial x}{\partial t}_{|X},$$

and so computation of the mapping $x(X,t):\Omega_0^f\to\Omega^f(t)$ is required. Moreover, $x(X,t)$ must be identical to the position x_s of a material point X of the structure.

13.7.2 Structural problem

Structural problem is obtained by restricting the conservation laws to test functions in Q^s and V^s. As functions in Q^s are zero in Ω^f and configuration Ω_0^s is a material configuration – $x(X,t)$ on the structure corresponds to the position of a material point, which was located at point X at referential time t_0, so $\mathbf{u}=\mathbf{u}_G^s$, remains

$$\frac{\partial\rho_0}{\partial t} = 0 \text{ na } \Omega_0^s,$$

$$\int_{\Omega_0^s}\left(\rho_0\dot{\mathbf{u}}_i\cdot\hat{U}_i+(J_\varphi\,\sigma\cdot(\nabla\varphi)^{-T})_{ij}\frac{\partial\hat{U}_i}{\partial X_j}\right)\mathrm{d}X$$
$$=\int_{\Omega_0^s}f_0\cdot\hat{U}\mathrm{d}X+\int_{\partial\Omega_0^s-\Gamma_{I_0}}g_0\cdot\hat{U}\mathrm{d}X+L_{\text{interface}}(\hat{U}),\ \forall\hat{U}\in V^f$$

Here $L_{\text{interface}}$ represents an impact of fluid forces on the structure and is obtained by choosing structural test functions, which are nonzero on the interface

$$\begin{aligned}L_{\text{interface}}(\hat{U}) &= \int_{\Omega^f(t)}f\cdot\hat{U}\mathrm{d}x+\int_{\partial\Omega^f(t)-\Gamma_I(t)}g\cdot\hat{U}\mathrm{d}x \\ &- \int_{\Omega^f(t)}\left(\frac{1}{J_\varphi}\frac{\partial(J_\varphi\rho\mathbf{u}_i)}{\partial t}|_X+\operatorname{div}_x(\rho\mathbf{u}_i(\mathbf{u}-\mathbf{u}_G^f))\right)\hat{U}_i\mathrm{d}x-\int_{\Omega^f(t)}\sigma_{ij}\frac{\partial\hat{U}_i}{\partial x_j}\mathrm{d}x.\end{aligned}\tag{13.9}$$

This formulation represents abstract Lagrangian formulation of structural problem. It is necessary to be completed with the constitutive equation of depence of stresses and with boundary conditions, given on the structural surface. For hyperelastic structure, when first Piola-Kirchhoff stress tensor equals to the derivative of the density of elastic energy, we have

$$T(X):=J_\varphi\cdot\sigma\cdot(\nabla\varphi)^{-T}(X)=\frac{\partial}{\partial(\nabla\varphi)}\Psi(X,\nabla\varphi).$$

Then the structural problem converts to the classical form [3]

$$m^s(\ddot{x}^s,\hat{U})+a^s(x^s,\hat{U})=L_{\text{interface}}(\hat{U}),\ \forall\hat{U}\in V^s,\tag{13.10}$$

where we denoted

$$\begin{aligned}\ddot{x}^s &= \dot{U}^s=\frac{\partial^2x^s}{\partial t^2_{|X}}, \\ m^s(\ddot{x}^s,\hat{U}) &= \int_{\Omega_0^s}\rho_0\ddot{x}^s\cdot\hat{U}, \\ a^s(x^s,\hat{U}) &= \int_{\Omega_0^s}\frac{\partial}{\partial(\nabla\varphi)}\Psi(X,\nabla\varphi)\cdot\frac{\partial\hat{U}}{\partial X}.\end{aligned}$$

13.7.3 Global coupled problem

Because each test function decomposes by construction into a sum of a structural test function and a fluid test function, global conservation laws are equivalent to the system, obtained by satisfying its fluid component (fluid problem without condition on the boundary) and its structural component (structural problem) together. Global coupled problem, which is obtained by prescribing the mapping $x(X,t)$, kinematic conditions on the boundary and global equilibrium equations is finally transformed to this three-field system:

1. calculate velocity field $\mathbf{u}^f$ in the fluid domain by solving fluid problem (13.6), (13.7) with Dirichlet conditions (13.8) on the interface [5];
2. calculate structural position x^s by solving the structural problem (13.10);
3. construct a mapping x from Ω_0 to $\Omega(t)$ such, that

$$x_{|\Omega_0^s}=x^s.$$

13.8 Numerical solution

Possible scheme, applicable to computation of the new configuration is the *trapezoidal formula* , i.e.

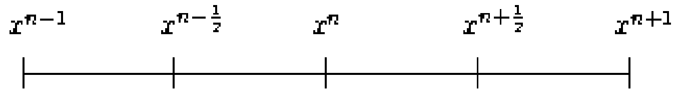
$$x^{n+1} = x^n + \frac{\Delta t}{2}(\mathbf{u}^{n+1} + \mathbf{u}^n).$$

Schemes of Euler type are not used too frequently in structural dynamics, because they are too dissipative. It is recommended to use linear conservative schemes similar to the class of Newmark schemes or central differences, which guarantee conservation of mechanical energy in linear elasticity. We use subsequent formula:

$$x^{n+1} = x^n + \frac{3\Delta t}{2}\mathbf{u}^n - \frac{\Delta t}{2}\mathbf{u}^{n-1}.$$

Proof:

We consider points located this way:



using Lagrange interpolation theorem applied to the function $\mathbf{u}(x)$ and points x^n and x^{n+1} we can write

$$\mathbf{u}(x) = \mathbf{u}^{n-1}\frac{x - x^n}{x^{n-1} - x^n} + \mathbf{u}^n\frac{x - x^{n-1}}{x^n - x^{n-1}}.$$

We set here now $x = x^{n+\frac{1}{2}}$. Then

$$\mathbf{u}^{n+\frac{1}{2}} = -\frac{1}{2}\mathbf{u}^{n-1} + \frac{3}{2}\mathbf{u}^n$$

Substituting in this formula from the relation for central difference

$$\mathbf{u}^{n+\frac{1}{2}} = \frac{x^{n+1} - x^n}{\Delta t}$$

we have

$$\frac{x^{n+1} - x^n}{\Delta t} = -\frac{1}{2}\mathbf{u}^{n-1} + \frac{3}{2}\mathbf{u}^n,$$

from which the required relation follows

$$x^{n+1} = x^n + \frac{3\Delta t}{2}\mathbf{u}^n - \frac{\Delta t}{2}\mathbf{u}^{n-1}.$$

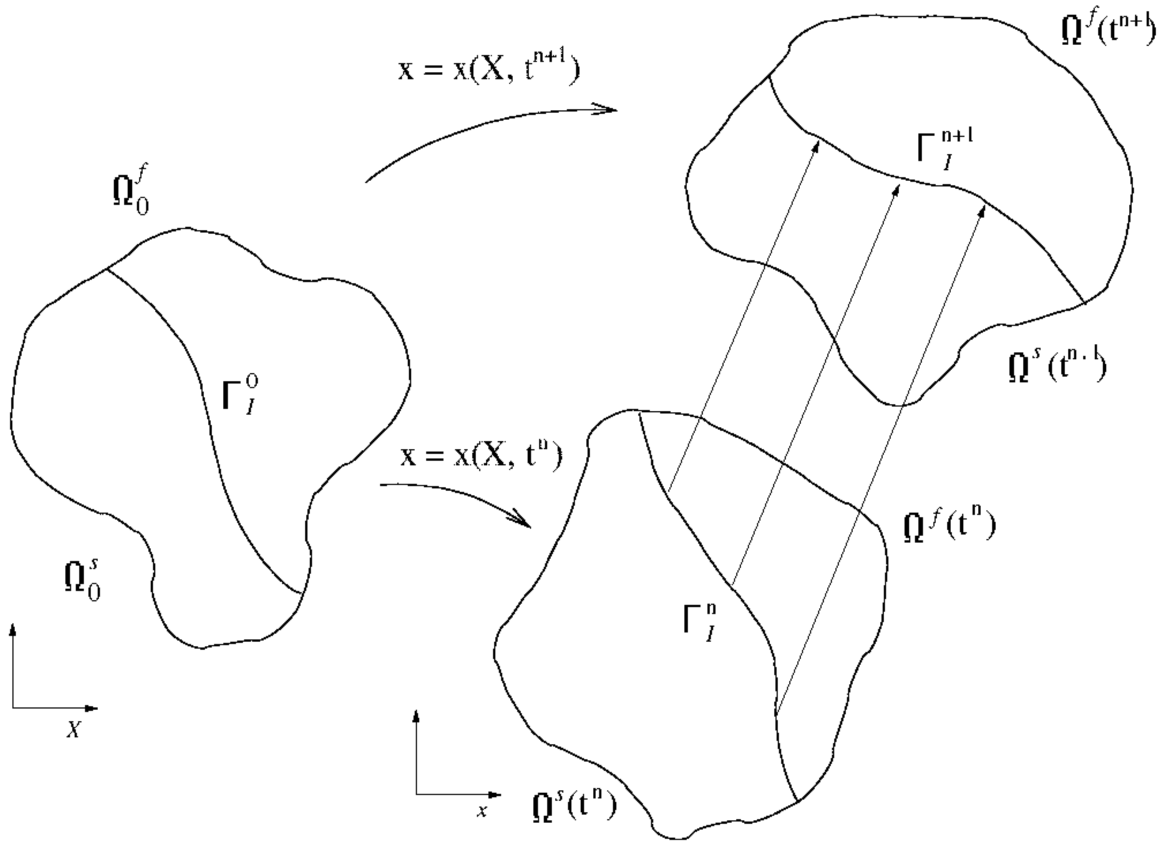


Figure 13.3: Migration from interface position $\Gamma_I^n = x(X, t^n) \mid_{\Gamma_I}$ at time t^n to the new position $\Gamma_I^{n+1} = x(X, t^{n+1}) \mid_{\Gamma_I}$ at time t^{n+1}

13.9 Basic algorithm

Disadvantage of the implicit scheme introduced in previous chapter is algebraic coupling of fluid and structural problem. By introducing relaxation algorithms of the fixed point type it is possible to separate the solution of these problems. Suppose, that the problem is solved at time $t = t^n$ (see Fig. 13.3).
Procedure for migration of the interface position at time step t^n to the position of the interface at time step t^{n+1} is as follows:
Template 50, interface migration algorithm I

- aproximate $\mathcal{X} = \left(x^s_{|\Gamma_I}\right)^{n+1}$ the interface position by an explicit choice

$$\mathcal{X} = \left(x^s_{|\Gamma_I}\right)^n + \frac{3\Delta t}{2}(\mathbf{u}^s_{|\Gamma_I})^n - \frac{\Delta t}{2}(\mathbf{u}^s_{|\Gamma_I})^{n-1};$$

- Now follow spatial iterations 1 - 6 until the required accuracy is reached

  1. calculation of fluid velocities on the interface

  $$\mathbf{u} = \frac{\mathcal{X} - \left(x^s_{|\Gamma_I}\right)^n}{\Delta t} = \mathbf{u}_G^{n+1}{}_{|\Gamma_{I_0}};$$

  2. actualization of fluid configuration (fluid grid) $x(X, t^{n+1})$ in Ω_0^f when respecting the material condition

  $$x(X, t^{n+1})_{|\Gamma_{I_0}} = \mathcal{X};$$

  3. solution of the fluid problem on the actualized configuration

  $$\int_{\Omega_0^f} \frac{J_{\boldsymbol{\varphi}}^{n+1}\rho^{n+1} - J_{\boldsymbol{\varphi}}^n\rho^n}{\Delta t}\hat{q}\mathrm{dX} + \int_{\Omega^f(t)} \mathrm{div}_x(\rho^{n+1}(\mathbf{u}^{n+1} - \mathbf{u}_G^{n+1}))\hat{q}\mathrm{d}x = 0,$$

  $$\begin{aligned} \int_{\Omega_0^f} \frac{J_{\boldsymbol{\varphi}}^{n+1}\rho^{n+1}\mathbf{u}^{n+1} - J_{\boldsymbol{\varphi}}^n\rho^n\mathbf{u}^n}{\Delta t} \cdot \hat{U}\mathrm{d}X \quad &+ \\ + \int_{\Omega^f(t)} \mathrm{div}_x(\rho^{n+1}\mathbf{u}_i^{n+1}(\mathbf{u}^{n+1} - \mathbf{u}_G^{n+1})) \cdot \hat{U}\mathrm{d}x \quad &+ \\ + \int_{\Omega^f(t)} (-p^{n+1}I + \mu(\partial_i\mathbf{u}_j^{n+1} + \partial_j\mathbf{u}_i^{n+1}))\frac{\partial\hat{U}_i}{\partial x_j}\mathrm{d}x \quad &= \\ = \int_{\Omega^f(t)} f \cdot \hat{U}\mathrm{d}x + \int_{\partial\Omega^f(t)-\Gamma_I(t)} g \cdot \hat{U}\mathrm{d}S&, \end{aligned}$$

  $$\forall\ \hat{q} : \Omega_0^f \longrightarrow I\!R, \quad \forall\ \hat{U} \in V^f, (\mathbf{u}^{n+1} - \mathbf{u}) \in V^f;$$

Template 51, interface migration algorithm II

4 computation of interface loads

$$\begin{aligned} L^{n+1}_{interface}(\hat{U}) &= \int_{\Omega^f(t)} f \cdot \hat{U} \mathrm{d}x + \int_{\partial\Omega^f(t)-\Gamma_I(t)} g \cdot \hat{U} \mathrm{d}S - \\ &- \int_{\Omega^f_0} \frac{J^{n+1}_{\varphi} \rho^{n+1} \mathbf{u}^{n+1} - J^n_{\varphi} \rho^n \mathbf{u}^n}{\Delta t} \cdot \hat{U} \mathrm{d}X - \\ &- \int_{\Omega^f(t)} \mathrm{div}_x (\rho^{n+1} \mathbf{u}^{n+1}_i (\mathbf{u}^{n+1} - \mathbf{u}^{n+1}_G)) \cdot \hat{U} \mathrm{d}x - \\ &- \int_{\Omega^f(t)} (-p^{n+1} I + \mu(\partial_i \mathbf{u}^{n+1}_j + \partial_j \mathbf{u}^{n+1}_i)) \frac{\partial \hat{U}_i}{\partial x_j} \mathrm{d}x; \end{aligned}$$

5 solution of the structural problem for x_s

$$m^s((\ddot{x}^s)^{n+1}, \hat{U}) + a^s((x^s)^{n+1}, \hat{U}) = L^{n+1}_{interface}(\hat{U}),\ \forall \hat{U} \in V^s,\ (x^s)^{n+1} \in V^s;$$

6 actualize new position by relaxation ($\omega_n \in (0, 1]$)

$$\mathcal{X} = (1 - \omega_n)\mathcal{X} + \omega_n (x^s_{|\Gamma_I})^{n+1};$$

and goto 1.

Appendix

A1. Green theorem

Let $\Omega \subset I\!R^n$ be a bounded domain with Lipschitzian boundary. Let $\mathbf{n}(x)$ be a unit vector of outward normal to the boundary $\partial\Omega$. Let u, v be Lipschitzian functions from $\bar{\Omega}$ to $I\!R$. Then

$$\int_\Omega \frac{\partial u}{\partial x_i} v \ \mathrm{d}x = \int_{\partial\Omega} u \ v \ n_i \ \mathrm{d}S - \int_\Omega u \frac{\partial v}{\partial x_i} \ \mathrm{d}x.$$

Especially, setting $u = u_i$, $v_i = 1$ and summing up the equations over i, we get the theorem on the divergence of a vector field

$$\int_\Omega \frac{\partial u_i}{\partial x_i} \ \mathrm{d}x = \int_{\partial\Omega} \mathbf{u} \cdot \mathbf{n} \ \mathrm{d}S.$$

A2. Chain rule theorem

Let $\mathbf{G} : I\!R^n \to I\!R^k$, $t_0 \in I\!R^n$, and let exists total differential $\mathbf{G}'(t_0)$. Denote $x_0 = \mathbf{G}(t_0)$, let $\mathbf{F} : I\!R^k \to I\!R^s$ and exists $\mathbf{F}'(x_0)$. Then there exists total differential $(\mathbf{F} \circ \mathbf{G})'(t_0)$ and the following holds

$$(\mathbf{F} \circ \mathbf{G})'(t_0) = \mathbf{F}'(x_0) \cdot \mathbf{G}'(t_0).$$

If

$$\mathbf{G}'(t_0) = \begin{pmatrix} \frac{\partial g_1}{\partial t_1}(t_0), & \dots, & \frac{\partial g_1}{\partial t_n}(t_0) \\ \vdots & & \vdots \\ \frac{\partial g_k}{\partial t_1}(t_0), & \dots, & \frac{\partial g_k}{\partial t_n}(t_0) \end{pmatrix} \text{ a } \mathbf{F}'(x_0) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_0), & \dots, & \frac{\partial f_1}{\partial x_k}(x_0) \\ \vdots & & \vdots \\ \frac{\partial f_s}{\partial x_1}(x_0), & \dots, & \frac{\partial f_s}{\partial x_k}(x_0) \end{pmatrix}$$

then the derivative of a compose mapping represents composition of linear forms

$$(\mathbf{F} \circ \mathbf{G})'(t_0) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_0), & \dots, & \frac{\partial f_1}{\partial x_k}(x_0) \\ \vdots & & \vdots \\ \frac{\partial f_s}{\partial x_1}(x_0), & \dots, & \frac{\partial f_s}{\partial x_k}(x_0) \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial g_1}{\partial t_1}(t_0), & \dots, & \frac{\partial g_1}{\partial t_n}(t_0) \\ \vdots & & \vdots \\ \frac{\partial g_k}{\partial t_1}(t_0), & \dots, & \frac{\partial g_k}{\partial t_n}(t_0) \end{pmatrix}.$$

Regular mapping Let $\mathbf{F}$ be a mapping of an open set $G \subset I\!R^n$ on $I\!R^n$. Mapping $\mathbf{F}$ is called regular on G, if

1. function $\mathbf{F}$ is continuously differentiable in G
2. Jacobian $J_{\mathbf{F}}(x) \neq 0$ at each point $x \in G$.

A3. Substitution theorem

Let Φ be a mapping of an open set $P \subset I\!R^n$ on $Q \subset I\!R^n$. Let Φ be regular and injective in P, with its determinant J_Φ. Let $M \subset Q$ and let F be an arbitrary real function. Then (we consider Lebesgue integrals)

$$\int_M F(x)\mathrm{d}x = \int_{\Phi^{-1}(M)} F(\Phi(t))|J_\Phi(t)|\mathrm{d}t,$$

as soon as one of the integrals exists.

A4. Local dipheomorphism theorem

Let mapping $\mathbf{F}$ from $G \subset I\!R^n$ on $I\!R^n$ be regular. Then

1. For each point $x_0 \in G$ there exists neighborhood $\mathbf{u}(x_0)$ such, that mapping $\mathbf{F}_{|\mathbf{u}(x_0)}$ is injection
2. Image $\mathbf{F}(\mathbf{u}(x_0))$ is an open set
3. If $\mathbf{F}$ is injective, then mapping $\mathbf{F}^{-1}$ has continuous derivatives of first order in $\mathbf{F}(\mathbf{u}(x_0))$.

A5. Derivative of parameter-dependent integral

Let $G \subset I\!R^n$, $f(x,t)$ be a real function of $n+1$ variables, $x \in G$, $t \in (a,b) \subset I\!R$. Suppose it is satisfied

1. Integral $F(t) = \int_G f(x,t)\,\mathrm{d}x$ converges at least for one $t \in (a,b)$.
2. For each $t \in [a,b]$ is a function $t \to f(x,t)$ is measurable in G.
3. There exists a set $N \subset G$ of zero measure such, that for each $x \in G \dot{-} N$ and $t \in (a,b)$ finite $\frac{\partial f(x,t)}{\partial t}$ exists.

4. There exists a function $g(x)$, with finite $\int_G g(x)\mathrm{d}x$ such, that for each $x \in G \dot{-} N$ and $t \in (a,b)$ is $|\frac{\partial f(x,t)}{\partial t}| \leq g(x)$.

Then it holds: For each $t \in (a,b)$ is

$$F'(t) = \int_G \frac{\partial f(x,t)}{\partial t}\,\mathrm{d}x.$$

Bibliography

[1] J. Boujot. Mathematical formulation of fluid-structure interaction problems. *Model. Math. Anal. Numer.*, 21:239–260, 1987.

[2] J. Chorin and E. Marsden. *A mathematical introduction to fluid mechanics*. Springer, 1993.

[3] P.G. Ciarlet. *Mathematical elasticity, Vol. 1 - Three-dimensional elasticity*. North-Holland, 1988.

[4] T.J.R. Hughes, W.K. Liu, and T.K. Zimmerman. Lagrangian-eulerian finite element formulation for incompressible viscous flows. *Comput. Meth. Appl. Mech. Eng.*, 29:329–349, 1981.

[5] P. Le Tallec and J. Mouro. Fluid structure interaction with large structural displacements. In *Proceedings of the 4-th European CFD conference, Athens*, 1998.

Chapter 14

FE analysis in a nutshell

This part was written and is maintained by M. Okrouhlík. More details about the author can be found in the Chapter 16.

14.1 Introduction

To elucidate the programming steps required when an efficient matrix storage mode is employed a few simple programs allowing to do the FE analysis almost by hand are presented in this chapter. The finite element method could be applied to various physical problems. Probably the most frequent use in mechanical engineering is that based on equilibrium conditions and on the Newton's second law.

The solution of linear finite element tasks in solid mechanics (by the deformation[1] variant of the finite element method) could be broadly classified as static, transient and steady state. In the deformation variant the displacements are considered to be primary unknowns – the strains, stresses and forces are calculated from displacements later.

- Statics. Solving statics problems leads to the solution of the system of linear algebraic equations, having the form

$$\mathbf{K}\,\mathbf{q} = \mathbf{P}. \tag{14.1}$$

  For a given loading $\mathbf{P}$, one has to find the the displacements $\mathbf{q}$ of a body (structure), whose stiffness and boundary conditions are defined by the $\mathbf{K}$ matrix.

- Transient. Solving the transient response of a body (structure) to an impact loading, one has to solve the system of ordinary differential equations of the second order.

$$\mathbf{M}\,\ddot{\mathbf{q}} + \mathbf{C}\,\dot{\mathbf{q}} + \mathbf{K}\,\mathbf{q} = \mathbf{P}(t). \tag{14.2}$$

  For a given loading, prescribed as a function of time $\mathbf{P}(t)$, one is to find the motion of a body (structure) in time and space.

- Steady state. Solving an undamped steady state vibration problems requires to solve the generalized eigenvalue problem

$$(\mathbf{K} - \lambda\,\mathbf{M})\,\bar{\mathbf{q}} = \mathbf{0}. \tag{14.3}$$

[1]By displacements we usually understand the displacements themselves or the rotations (as in the beam theory) using sometimes a generic term generalized displacements. Similarly for forces and generalized forces.

Knowing the mass and stiffness distribution of a body (structure) one is able to find its eigenvalues (eigenfrequencies) and eigenmodes (natural modes of vibration).

In mechanical engineering the physical meanings of variables appearing in the above equations are usually as follows

$\mathbf{q}$	displacements,
$\dot{\mathbf{q}}$	velocities,
$\ddot{\mathbf{q}}$	accelerations,
$\bar{\mathbf{q}}$	amplitudes,
$\mathbf{M}$	mass matrix,
$\mathbf{C}$	damping matrix,
$\mathbf{K}$	stiffness matrix,
$\mathbf{P}$	static loading,
$\mathbf{P}(t)$	loading as a function of time.

Efficient implementation of the finite element method requires to consistently employ the special matrix properties as their symmetry, bandedness, positive definiteness, etc. A possible approach is sketched in the following paragraph.

14.2 Rectangular band storage

One can save a considerable amount of memory space by storing a symmetric banded matrix, say **R**, in a rectangular array, say **A**, as indicated in the following figure.

```
        * * * * . . . . . .           * * * *
        . * * * *         .           * * * *
        .   * * * *       .           * * * *
        .     * * * *     .           * * * *
        .       * * * *   .           * * * *
  [R] = .         * * * * .     [A] = * * * *
        .           * * * *           * * * *
        .             * * *           * * * .
        .               * *           * * . .
        . . . . . . . . . *           * . . .
                           (N * N)           (N * NBAND)
```

The variable `NBAND` is a measure of the half-band width. The *index function*, relating the indices `I,J` of a standardly stored element of `R` array to indices `K,L` of the equivalent element in rectangular array `A`, is given by

$$\begin{aligned} \mathtt{K} &= \mathtt{I} \\ \mathtt{L} &= \mathtt{J - I + 1}. \end{aligned} \tag{14.4}$$

So the `A` array could be created from the `R` array by a few lines

```
DO 40 I = 1,N
   JM = I + NBAND - 1
   IF (JM .GT. N) JM = N
   DO 40 J = I,JM
      L = J - I + 1
      A(I,L) = R(I,J)
40 CONTINUE
```

Of course, nobody would proceed this way. The big matrix, not fitting the computer RAM memory, is usually stored in our minds only. We usually address the element of a standardly stored matrix by indices (obtained from (14.4)) pointing to its location in the rectangular array.

14.3 Solution of the system of equations

Classical Gauss elimination method is used. It is assumed that the matrix is symmetric and banded and that is stored in the rectangular array as shown above. The matrix is also assumed to be positive definite, so no pivoting is carried out. In statics the stiffness matrix is positive definite, if the studied body (structure) has no rigid-body degrees of freedom. This could be used a suitable check.

Solution of $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ is carried out in two steps.

- **Triangular decomposition** is formally described as follows.

  $$\mathbf{A} \rightarrow \mathbf{L}\,\mathbf{D}\,\mathbf{L}^{\mathrm{T}} \rightarrow \mathbf{L}\,\mathbf{U},$$

  where

$\mathbf{A}$	matrix of the system,
$\mathbf{L}$	lower triangular matrix,
$\mathbf{D}$	diagonal matrix,
$\mathbf{L}^{\mathrm{T}}$	lower triangular matrix transposed,
$\mathbf{U}$	upper triangular matrix containing the triangular decomposition of $\mathbf{A}$.

- **Reduction of the right hand side and the backsubstitution** are formally described by following relations.

$$\begin{aligned} \mathbf{L}\,\mathbf{U}\,\mathbf{x} &= \mathbf{b}, \\ \mathbf{U}\,\mathbf{x} &= \mathbf{L}^{-1}\,\mathbf{b}, \\ \mathbf{x} &= \mathbf{U}^{-1}\,\mathbf{L}^{-1}\,\mathbf{b}. \end{aligned} \tag{14.5}$$

The above matrix formulation of the Gauss elimination could be used for the implementation of actual equation solution. It would, however, be very inefficient, in practice the row modifications – similar to those carried out by hand computations – are used, as it will be shown in the text to follow.

The indicated decomposition of the Gauss process allows for the efficient treatment of a system of equations having more right hand sides. This way the elimination process is carried out only once, while the second part of the process – reduction of the right hand side and the backsubstitution, which is substantially cheaper in terms of required floating point operations – could be repeated as often as needed.

The actual solution of the system of algebraic equations is carried out by the `DGRE` subroutine which has to be called twice. It is assumed that the matrix is efficiently stored in a rectangular array. At first, with `KEY = 1`, for the matrix triangularization, then, with `KEY = 2`, for the reduction of the right hand side and for the back substitution.

```
KEY = 1
CALL DGRE(A,B,X, ... KEY, ...)
KEY = 2
CALL DGRE(A,B,X, ... KEY, ...)
```

The elimination process is carried out 'in place'. From it follows that the original input matrix is destroyed – overwritten by elements of the upper triangular part of the triangularized matrix. The `DGRE` subroutine listing is in the Program 31.

Program 31

```
      SUBROUTINE DGRE(A,B,Q,N,NDIM,NBAND,DET,EPS,IER,KEY,KERPIV)
      DIMENSION A(NDIM,NBAND),B(N),Q(N)
      DOUBLE PRECISION SUM,A,B,Q,AKK,AKI,AIJ,AKJ,ANN,T,AL1,AL2,AII
     1                 ,DET,EPS
C
C     ***   Solution of [R]{Q} = {B}
C     ***   by Gauss elimination (no pivoting)
C     ***   for a symmetric, banded, positive definite matrix [R]
C
C     It is assumed that the upper triangular part
C     of the band [R] is stored in a rectangular array A(NBAND,N)

C     Parameters
C      A   on input         contains the upper triangular band of [R] matrix
C          on output        triangularized part of [R]
C
C      B                    RHS vector
C      Q                    result vector
C      N                    number of unknowns
C      NDIM                 row dimension of A array declared in main
C      NBAND                half bandwidth (including diagonal)
C      DET                  matrix determinant
C      EPS                  smallest acceptable pivit value
C      IER                  error parameter
C                              = 0 ... O.K.
C                              = -1    matrix is singular or not positive definite
C                                      computation is interupted
C
C     KEY                   key of the solution
C                              = 1 ... triangularization of the input matrix
C                              = 2 ... RHS reduction and back substitution
C     KERPIV                the row number where the solution failed
C
      NM1=N-1
      IER=0
      II=KEY
      GO TO (1000,2000), II
C
C     Triangularization
C
1000  DET=1.
```

```
      IRED=0
      DO 9 K=1,NM1
      AKK=A(K,1)
      KERPIV=K
      IMAX=K+NBAND-1
      IF(IMAX .GT. N) IMAX=N
      JMAX=IMAX
      IF(AKK .GT. EPS) GO TO 5
      IER=-1
      RETURN
C     DET=DET*AKK
5     CONTINUE
      KP1=K+1
      DO 9 I=KP1,IMAX
      AKI=A(K,I-K+1)
      IF(ABS(AKI) .LT. EPS) GO TO 9
      T=AKI/AKK
      DO 8 J=I,JMAX
      AIJ=A(I,J-I+1)
      AKJ=A(K,J-K+1)
8     A(I,J-I+1)=AIJ-AKJ*T
9     CONTINUE
      ANN=A(N,1)
      DET=DET*ANN
C
C     Triangularization successfully finished
C
      IRED=1
      RETURN
C
C     Reduction of the RHS vector
C
2000  DO 90 K=1,NM1
      KP1=K+1
      AKK=A(K,1)
      IMAX=K+NBAND-1
      IF(IMAX .GT. N) IMAX=N
      DO 90 I=KP1,IMAX
      AKI=A(K,I-K+1)
      T=AKI/AKK
90    B(I)=B(I)-T*B(K)
C
C     Back substitution
C
      Q(N)=B(N)/A(N,1)
      AL1=A(N-1,2)
      AL2=A(N-1,1)
      Q(N-1)=(B(N-1)-AL1*Q(N))/AL2
      DO 10 IL=3,N
      I=N-IL+1
      AII=A(I,1)
      SUM=0.D0
      J1=I+1
      JMAX=MIN0(I+NBAND-1,N)
      DO 20 J=J1,JMAX
      AIJ=A(I,J-I+1)
20    SUM=SUM+AIJ*Q(J)
```

```
 10    Q(I)=(B(I)-SUM)/AII
       RETURN
       END
```

End of Program 31. □

The disk oriented version of the above procedure, suitable for large matrices not fitting the internal memory, is called DBANGZ and is listed as the Program 25.

14.4 Solution of the system of ordinary differential equations

The system of ordinary differential equations of the second order, Eq. (14.2), describes the behavior of a mechanical system (body, structure) characterized by its mass, damping and stiffness properties, i.e. by mass $\mathbf{M}$, $\mathbf{C}$ and $\mathbf{K}$ matrices respectively. Solving the equations, for given initial conditions and for prescribed loading forces in time, we get the spatial and temporal response of the structure in terms of displacements, velocities and accelerations.

The solution could be obtained by plethora of approaches – here we show the treatment of ordinary differential equations by the *Newmark method* and by the *method of central differences*. In more detail the methods are described in the Chapter 3.

14.4.1 The Newmark method

The Newmark method might be implemented as a two part process, as indicated in Templates 52 and 53.

Template 52, Get ready for the Newmark method

1. Assemble mass, damping and stiffness matrices, i.e. $\mathbf{M}$, $\mathbf{C}$ and $\mathbf{K}$ and store their upper bands (including diagonals) in rectangular arrays as shown above.
2. Make a copy of the mass matrix $\mathbf{M}^{\text{copy}} \leftarrow \mathbf{M}$.
3. Prescribe the time distribution of the loading $\mathbf{P}(t)$.
4. Set the initial conditions, i.e. the displacements and velocities $\mathbf{q}_0$ and $\dot{\mathbf{q}}_0$ at time $t = 0$.
5. Compute initial accelerations from $\mathbf{M}\,\ddot{\mathbf{q}} = \mathbf{Q}$, where $\mathbf{Q} = \mathbf{P}_0 - \mathbf{K}\,\mathbf{q}_0$ using `DGRE` subroutine twice. Since the input matrix is destroyed during the triangularization process we should work with a copy $\mathbf{M}$. The original mass matrix will be needed again.

   `CALL DGRE` $(\mathbf{M}^{\text{copy}}, \mathbf{Q}, \ddot{\mathbf{q}}_0, \ldots$ `1`$)$
   `CALL DGRE` $(\mathbf{M}^{\text{copy}}, \mathbf{Q}, \ddot{\mathbf{q}}_0, \ldots$ `2`$)$
6. Set the step of integration `H`.
7. Set the Newmark parameter `GAMMA`. If `GAMMA = 0.5` there is no algorithmic damping. If `GAMMA > 0.5` the frequencies of the upper part of the spectrum are filtered-out.
8. Compute the second Newmark parameter from `BETA = 0.25*(0.5 + GAMMA)**2`.
9. Compute the constants `A1 = 1./(BETA*H*H)` and `A1D = GAMMA/(BETA*H)`.
10. Compute the effective stiffness matrix $\mathbf{K}^{\text{eff}} = \mathbf{K} + \texttt{A1} * \mathbf{M} + \texttt{A1D} * \mathbf{C}$,
11. Triangularize the effective stiffness matrix by

    `CALL DGRE` $(\mathbf{K}^{\text{eff}}, \ldots$ `1`$)$ ... the $\mathbf{K}^{\text{eff}}$ matrix is triangularized in place.

Template 53, Newmark itself

The integration process itself is secured by the repeated invocation of the `NEWMD` subroutine as follows

```
      T = 0.D0
      TMAX = ...
 10    CALL NEWMD(BETA, GAMMA, q, q̇, q̈, P, M, K^eff, ··· )
      T = T + H
      IF(T .LE. TMAX) GOTO 10
```

Notice that the input parameter $\mathbf{K}^{\text{eff}}$ is the triangularized effective stiffness matrix, not the effective stiffnes matrix. The `NEWMD` subroutine is listed in the Program 5. The Matlab implementation of the Newmark algorithm is in the Program 4. The vectors $\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}$ are being constatly rewritten – on input they contain the old values, on output the new ones at time increased by the time step.

Notes to the Newmark method

- On input the arrays corresponding to $\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}$ contain the values at time `T`, on output those at time `T + H`.
- The triangular decomposition of the effective stiffness matrix is carried out only once, before the time integration process has started.
- The `NEWMD` subroutine calls two other procedures, namely `DGRE` and `DMAVB`.
- The Newmark procedure is unconditionally stable. Using an unreasonably high integration steps leads to wrong results, but the user is not warned by 'explosion' of results as it is when conditionally stable methods are used. How to set a 'correct' step of integration is sketched in the Template 56.

14.4.2 The central difference method

The method of central differences might be implemented as a two part process, see Templates 54 and 55.

Template 54, Get ready for the central difference method

1 - 4. The first four items are identical with those of the Template 52.

5. Make a copy of the mass matrix $\mathbf{M}^{\text{copy}} \leftarrow \mathbf{M}$.

6. Set the step of integration `H`.

7. Calculate constants `A0=1./(H*H), A1=1./(2.*H), A2=2.*A0, A3=1./A2.`

8. Compute the effective mass matrix $\mathbf{M}^{\text{eff}} = \texttt{A0} * \mathbf{M}$.

9. Triangularize the effective mass matrix by
   `CALL DGRE` $(\mathbf{M}^{\text{eff}}, \ldots$ `1`$)$ ... the $\mathbf{M}^{\text{eff}}$ matrix is triangularized in place.

10. Evaluate displacements at time `T - H`, i.e. $\mathbf{q}_{\texttt{-H}} = \mathbf{q} - \texttt{H} * \dot{\mathbf{q}}_0 + \texttt{A3} * \ddot{\mathbf{q}}_0$.

Template 55, The central difference method itself

```
      T = 0.D0
      TMAX = ...
 10   CALL CEDIF ( M, Meff, K, qT-H, qT, qT+H, q̇, q̈, P, ··· )
      T = T + H
      IF(T .LE. TMAX) GOTO 10
```

The subroutine `CEDIF` is listed in the Program 32. The Matlab implementation of the central difference method – not observing any saving considerations – is in the Program 3.

Notes to the central difference method

- The subroutine, as it is conceived, is suitable for the consistent mass matrix. It could be used for the diagonal one as it stands, but the computation process – in terms of the required floating point operations – would be highly inefficient. For the diagonal mass matrix the procedure should be completely rewritten.
- The method is conditionally stable. It explodes if the time step is greater than the critical stel.
- The global mass matrix has to positive definite.
- The `CEDIDF` subroutine calls two other procedures, namely `DGRE` and `MAVBA`.

Program listing for the central difference method is in the Program 32.

Program 32

```
      SUBROUTINE CEDIF(XM,XM1,XK,DISS,DIS,DISN,VEL,ACC,P,R,
     1           IMAX,NBAND,NDIM,H,EPS,IER,A0,A1,A2,A3,IMORE)
C        :        :         :         :         :         :         :
      DIMENSION XM(NDIM,NBAND),XM1(NDIM,NBAND),XK(NDIM,NBAND),
     1          DISS(IMAX),DIS(IMAX),DISN(IMAX),VEL(IMAX),ACC(IMAX),
     2          P(IMAX),R(IMAX)
C
C     *** Time integration by the central difference method ***
C
C     Equations of motion are
C            [M]{ACC}+[K]{DIS}={P}  (no damping)
C     The mass matrix (assumed to be symmetric, banded, positive definite) and
C     the stiffness matrix (assumed to be symmetric, banded but need not be positive definite)
C     are efficiently stored in
C     rectangular arrays XM, XK with dimensions NDIM*NBAND.
C     The same storage mode is required for the effective mass matrix XM1
C     Only the upper part of the band (including diagonal) is stored.
C
C     Required subroutines
C       MAVBA    ...    matrix vector multiplication for the rectangular storage mode
C       GRE      ...    solution of algebraic equations by Gauss elimination
C                       for the rectangular storage mode
C
C     Parameters
C
C     XM(NDIM,NBAND) ...... upper band part of the mass matrix
C     XM1(NDIM,NBAND) ..... upper band part of the triangularized effective mass matrix
C     XK(NDIM,NBAND) ...... upper band part of the stiffness matrix
C     DISS(IMAX) .......... displacements at time T-H
C     DIS (IMAX) .......... displacements at time T
C     DISN(IMAX) .......... displacements at time T+H
C     VEL(IMAX) ........... velocities at time T
C     ACC(IMAX) ........... accelerations at time T
C     P(IMAX) ............. vector of loading forces at time T
C     R(IMAX) ............. vector of effective loading forces at time T
C     IMAX ................ number of unknowns
C     NBAND ............... half-band width (including diagonal)
C     NDIM ................ Row dimension of all matrices in main
C     H ................... Step of integration
C     EPS ................. Tolerance for the GRE subroutine
C     IER ................. Error parameter - see GRE
C     A0,A1,A2,A3 ......... Constants
C     IMORE ............... = 0 only new displacements, at time T+H, are computed
C                           = any other value means that
C                             velocities and accelerations at time T+H are evaluated as well
C
C      **************************************************************
C
C     The vector of effective loading forces at time  T
C     {R}={P}-[K]{DIS}+A2*[M]{DIS}-A0*[M]{DISS}
C
      CALL MAVBA(XK,DIS,VEL,IMAX,NDIM,NBAND)
      CALL MAVBA(XM,DIS,ACC,IMAX,NDIM,NBAND)
      CALL MAVBA(XM,DISS, R,IMAX,NDIM,NBAND)
```

```
C
      DO 10 I=1,IMAX
10    R(I)=P(I)-VEL(I)+A2*ACC(I)-A0*R(I)
C
C     displacements at time T+H
C
      CALL GRE(XM1,R,DISN,IMAX,NDIM,NBAND,DET,EPS,IER,2)
C
      IF(IMORE .EQ. 0) GO TO 30
C
C     Evaluate velocities and acceleration as well
C
      DO 20 I=1,IMAX
      VEL(I)=A1*(-DISS(I)+DISN(I))
20    ACC(I)=A0*(DISS(I)-2.*DIS(I)+DISN(I))
C
30    DO 40 I=1,IMAX
      DISS(I)=DIS(I)
40    DIS(I)=DISN(I)
      RETURN
      END
```

□ End of Program 32.

Instructions how to set the 'correct' step of time integration are in the Template 56.

Template 56, How to set the integration step value

```
 lmin            length of the smallest element
 c               wave speed
 tmin = lmin/c   time needed for the wave passage through
                   the smallest element
 hmts            how many time steps are needed to go
                   through the smallest element

****************************************
 H = tmin/hmts    a suitable time step *
****************************************

 hmts < 1    high frequency components are filtered out --
             applicable only for implicit methods
 hmts = 1    stability limit for explicit methods
             approximately equals to 2/omegamax,
             where omegamax = max(eig(M,K))
 hmts = 2    my choice
 hmts > 2    the high frequency components, which
             -- due to the time and space dispersion effects --
             are physically wrong, are integrated 'correctly'
```

14.5 Assembling global mass and stiffness matrices

Let's have a mechanical structure with `imax` global displacements and with `kmax` elements – each having `lmax` local displacements (local degrees of freedom).

To simplify the explication of the assembly process, we assume that all the elements are of the same type and have the same number of degrees of freedom, i.e. `lmax`.

Assume that we have already created the procedure `CODE(k,ic)`, that – when called – fills the array `ic(lmax)` by proper code numbers of the `k`-th element. Also, we already have procedures `RIG(k,xke)` and `MAS(k,xme)`, that provide the local stiffness and mass matrices of the `k`-th element, i.e. `xke(lmax, lmax)` and `xme(lmax, lmax)`.

If the standard storage scheme is assumed then the global stiffness and mass matrices could be assembled as shown in the Program 33. Notice that each matrix element of each local element matrix has to be 'touched'. This is what we call a low level programming.

Program 33
```
C     Loop on elements
      DO 10 k = 1,kmax
C       Code numbers of the k-th element
        CALL CODE(k, ic)
C       Local matrices of the k-th element
        CALL RIG(k, xke)
        CALL MAS(k, xme)
C       Loop on elements of local matrices
        DO 20 k1 = 1,lmax
          DO 20  k2 = 1,lmax
C         Pointers to locations in global matrices (standard storage)
          i1 = ic(k1)
          j1 = ic(k2)
          xk(i1,j1) = xke(k1,k2) + xk(i1,j1)
          xm(i1,j1) = xke(k1,k2) + xm(i1,j1)
20      CONTINUE
10    CONTINUE
```

End of Program 33. □

The resulting global matrices `xk` and `xm` assembled this way bear no information about prescribed boundary conditions. But this is another story to be treated elsewhere.

In Matlab, where we could use the array of pointers in the pointer position, the assembly process could be formally simplified as shown in the Template 34.

Program 34
```
%   Loop on elements
for k = 1:kmax
    %   code numbers of the k-th element
    ic = code(k);
    %   Local matrices for the k-th element
    xke = rig(k); xme = mas(k);
    %   assembling itself
    xm(ic,ic) = xm(ic,ic) + xme;
    xk(ic,ic) = xk(ic,ic) + xke;
end
```

End of Program 34. □

This is nice, elegant and simple. The advantages of high level programming has to be regretfully abandoned if an efficient matrix storage scheme is employed. We could proceed as indicated in the Template 35.

Program 35

```
      SUBROUTINE GLOB1(NDIM,IMAX,KMAX,LMAX,XK,XM,XKE,XME,IC,NBAND)
C
C     This procedure create global stiffness (XK) and mass (XM) matrices.
C     Both matrices are assumed to be symmetric and banded and
C     efficiently stored in a rectangular array.
C     Procedure GLOB1 calls CODE, RIG, MAS and CHANGE procedures.
C
C     NDIM ............ Row dimension of XK and XM in main
C     IMAX ............ Number of (generalized) displacements
C     KMAX ............ Number of elements
C     LMAX ............ Number of dof's of an element (all of the same type)
C     XK(IMAX,NBAND) .. Efficiently stored stiffness matrix
C     XM(IMAX,NBAND) .. Efficiently stored mass matrix
C     XKE(LMAX,LMAX) .. Local stiffness matrix
C     XME(LMAX,LMAX) .. Local mass matrix
C     IC(LMAX) ........ Code numbers of an element
C     NBAND ........... Half-band width (including diagonal)
C
      DIMENSION XK(NDIM,NBAND), XM(NDIM,NBAND), XKE(LMAX,LMAX),
     1          XME(LMAX,LMAX), IC(LMAX)
C
C     Insert zeros into the arrays
C
      DO 5 I = 1,IMAX
        DO 5 J = 1,NBAND
          XK(I,J) = 0.
          XM(I,J) = 0.
5     CONTINUE
C
C     Loop on elements
C
      DO 10 K = 1,KMAX
C
C       Compute the stiffness and mass matrices of the k-th element
C
        CALL RIG(K,XKE)
        CALL MAS(K,XME)
C
C       Compute the code numbers of the k-th element
C
        CALL CODE(K,IC)
C
C       Loop on the upper triangular parts of local matrices
C
        DO 20 K1 = 1,LMAX
          DO 20 K2 = K1,LMAX
C           In a standard storage mode we have
            I1 = IC(K1)
            J1 = IC(K2)
C           In case we try to address under-diagonal elements exchange indices
            IF(I1 .GT. J1) CALL CHANGE(I1,J1)
```

```
C           Calculate the equivalent pointers in the rectangular storage mode
C           (I1 is not changed)
            JL = J1 - I1 + 1
            XK(I1,JL) = XKE(K1,K2) + XK(I1,JL)
            XM(I1,JL) = XME(K1,K2) + XM(I1,JL)
20        CONTINUE
10      CONTINUE
        RETURN
        END
```

End of Program 35. □

Chapter 15

About authors

Mrs. Marta Čertíková

is the author of the Chapter 7.

- **First name, Family name:** Marta Čertíková
- **Titles:** RNDr
- **Affiliation:** Lecturer at Department of Technical Mathematics Faculty of Mechanical Engineering Czech Technical University Karlovo nmst 13 Karlovo náměstí 13 121 35 Praha 2 Czech Republic
- **Field of expertise:** Numerical Analysis and Algorithms, Finite Element Method, Domain Decomposition, Databases, Programming
- **Selected publications:**

```
Certikova, M.: Parallel Implementation and Optimization of
Balancing Domain Decomposition in Elasticity, Science and
Supercomputing in Europe, report 2006, CINECA, Bologna, Italy,
2006.

Burda, P., Certikova, M., Novotny, J. and Sistek, J.: BDDC method
with simplified coarse problem and its parallel implementation,
Proceedings of MIS 2007, Josefuv Dul, Czech Republic, Matfyzpress,
Praha, 2007.

Sistek, J., Novotny, J., Mandel, J., Certikova, M. and Burda, P.:
BDDC by a frontal solver and stress computation in a hip joint
replacement. Math. and Comp. Simulation (Elsevier), spec. issue
devoted to Computational Biomechanics and Biology, to appear in
2009.
```

- Email address: marta.certikova@fs.cvut.cz

Alexandr Damašek

is the author of Chapter 13.

- **First name, Family name:** Alexandr Damašek
- **Titles:** Ing., Ph.D.
- **Affiliation:** Institute of Thermomechanics, Academy of Sciences of the Czech Republic, 182 00 Prague, Dolejškova 5, the Czech Republic
- **Field of expertise:** Fluid dynamics, Elasticity of deformable bodies, Fluid-structure interaction
- **Selected publications:**

```
Damasek, A., Burda, P.: Numerical Solution of the Navier-Stokes
Equations for Large Reynolds Numbers using SUPG Method.
In: Proceedings Interactions and Feedbacks '01, Institute of
Thermomechanics, Academy of Sciences of the Czech
Republic, Prague, (in czech), 2001

Damasek, A., Burda, P.: SUPG Method for Solution of Unsteady
Fluid Flow for Large Reynolds Numbers, Seminar Topical
Problems of Fluid Mechanics 2002, Prague, (in czech), 2002

Damasek, A., Burda, P.: Solution of Higher Reynolds Flow along
Profile using Finite Element Method, Interaction and
Feedbacks 2002, Prague, (in czech), 2002

Damasek, A., Burda, P.: Finite Element Modelling of Viscous
Incompressible Fluid Flow in Glottis, Engineering Mechanics
2003, Svratka, (in czech), 2003

Damasek, A., Burda, P., Novotny, J.: Interaction of Viscous
Incompressible Fluid and Elastically Mounted Body when
Considering Finite Deformations}, Computational Mechanics 2004,
Nectiny, (in czech), 2004

Damasek, A., Burda, P.: Solution of Interaction of Viscous
Incompressible Fluid and Elastically Mounted Body using
ALE-method}, DTDT 2006, Usti nad Labem, (in czech)
20.09.2006-21.09.2006
```

- Email address: damasek@it.cas.cz

Jiří Dobiáš

is the author of Chapter 8.

- **First name, Family name:** Jiří Dobiáš
- **Titles:** Ing., Ph.D.
- **Affiliation:** Institute of Thermomechanics, Academy of Sciences of the Czech Republic, 182 00 Prague, Dolejškova 5, the Czech Republic
- **Field of expertise:** Non-linear problems of continuum mechanics with special emphasis to contact/impact, finite element method, domain decomposition methods and parallel algorithms. The chief investigator of three grant projects supported by the Grant Agency of the Czech Republic.
- **Selected publications:**

```
Dostal Z., Horak D., Kucera R., Vondrak V., Haslinger J., Dobias
J., Ptak S.: FETI based algorithms for contact problems:
scalability, large displacements and 3D Coulomb friction, Computer
Meth. in Appl. Mech. and Engineering, 194 (2005), pp. 395-409.

Dobias J., Ptak S., Dostal Z., Vondrak V.: Total FETI based
algorithm for contact problems with additional non-linearities,
Advances in Eng. Software, to be published.

Dobias J., Ptak S., Dostal Z., Vondrak V.: Scalable Algorithms for
Contact Problems with Additional Nonlinearities, Proceedings of
the Fifth International Conference on Engineering Computational
Technology, Las Palmas, Spain, Sept. 2006, Editors: Topping,
Montero, Montenegro, Civil-Comp Press, Scotland, paper No. 105,
ISBN 1-905088-10-8

Vondrak V., Dostal Z., Dobias J., Ptak S.: A FETI domain
decomposition method to solution of contact problems with large
displacements. Domain Decomposition Methods in Science and
Engineering XVI, Editors Olof Widlund and David Keyes, Volume 55
of Lecture Notes in Computational Science and Engineering.
Springer, 2007, pp. 771-778, ISSN 1439-7358

Vondrak V., Dostal Z., Dobias J., Ptak S.: Primal and Dual Penalty
Methods for Contact Problems with Geometrical Non-linearities,
Proc. Appl. Math. Mech. 5(2005), pp. 449-450

Dobias J.: Comparative Analysis of Penalty Method and Kinematic
Constraint Method in Impact Treatment, Proceedings of the Fifth
World Congress on Computational Mechanics, July 7-12, 2002,
Vienna, Austria, Editors: Mang H.A., Rammerstorfer F.G.,
Eberhardsteiner J., Publisher: Vienna University of Technology,
```

`Austria, ISBN 3-9501554-0-6.`

- Email address: jdobias@it.cas.cz

Dušan Gabriel

is the author of Chapter 9.

- **First name, Family name:** Dušan Gabriel
- **Titles:** Ing., Ph.D.
- **Affiliation:** Institute of Thermomechanics, Academy of Sciences of the Czech Republic, 182 00 Prague, Dolejškova 5, the Czech Republic
- **Field of expertise:** Research in the fields of computational mechanics, numerical methods, continuum mechanics, development of the finite element code PMD, stress analyses. Teaching course 'Theory of Elasticity' at CTU.
- **Selected publications:**

```
D. Gabriel, J. Plesek, R. Kolman, F. Vales. Dispersion of elastic
waves in the contact-impact problem of a long
cylinder. In: Proceedings of the 8th International Conference on
Mathematical and Numerical Aspects of Waves, N. Biggs
et al (eds.), 334--336, University of Reading, 2007.

D. Gabriel, J. Plesek, F. Vales, F. Okrouhlik. Symmetry preserving
algorithm for a dynamic contact-impact problem. In:
Book of Abstract of the III European Conference on Computational
Mechanics, C.A. Mota Soarez et al (eds.), Springer,
pp.318, CD-ROM 1-8, 2006.

I. Hlavacek, J. Plesek, D. Gabriel. Validation and sensitivity study
of an elastoplastic problem using the Worst
Scenario Method. Comp. Meths. Appl. Mech. Engng., 195, 736-774, 2006.

D. Gabriel, J. Plesek. Implementation of the pre-discretization
penalty method in contact problems. In: Computational
Plasticity VIII, D.R.J. Owen, E. Onate, B. Suarez (eds.), Barcelona,
Spain, 839-842, 2005.

J. Plesek, I. Hlavacek, D. Gabriel. Using the Worst Scenario Method
for error and scatter estimation in elasto-plastic
analysis. In: Computational Plasticity VIII, D.R.J. Owen, E. Onate,
B. Suarez (eds.), Barcelona, Spain, 1134-1137, 2005

D. Gabriel, J. Plesek, M. Ulbin. Symmetry preserving algorithm for
large displacement frictionless contact by the
pre-discretization penalty method. Int. J. for Num. Meth. in Engng.,
61, 2615-2638, 2004.

D. Gabriel. Numerical solution of large displacement contact problems
by the finite element method. CTU Reports, 7(3),
2003.

E. Hirsch, J. Plesek, D. Gabriel. How the liner material metallurgical
texture affects the shaped charge jet break-up
time. J. Phys.IV, 110, 723-727, 2003.
```

- Email address: gabriel@it.cas.cz

Miloslav Okrouhlík

is the editor of this e-book and the author of chapters 1, 2, 4, 5, 6 and 15. His personal whereabout are as follows:

- **First name, Family name:** Miloslav Okrouhlík
- **Titles:** Prof., Ing., CSc.
- **Affiliation:** Institute of Thermomechanics, Academy of Sciences of the Czech Republic, 182 00 Prague, Dolejškova 5, the Czech Republic
- **Field of expertise:** Computational mechanics. Stress wave propagation. Properties of finite element method in nonstationary dynamics. Finite element technology.
- **Selected publications:**

```
LANSKY, M.  OKROUHLIK, M.  NOVOTNY, J.:  Supercomputer NEC SX-4
Employed for Scientific and Technical Computations. Engineering
Mechanics, Vol. 7, No. 5, pp. 341-352, 2000.

LUNDBERG, B.  OKROUHLIK, M.: Influence of 3D effects on the
efficiency of percussive rock drilling. International Journal of
Impact Eng. Vol. 25, pp.345  360, 2001.

OKROUHLIK M.: Computational Aspects of Stress Waves Problems in
Solids  2nd European Conference on Computational Mechanics,
Abstracts Vol. 1, pp. 1  30, ISBN 83-85688-68-4, Fundacja Zdrovia
publicznego Vesalius, Cracow, Poland 2001.

STEJSKAL, V.  OKROUHLIK M.: Vibration with Matlab (In Czech:
Kmitn s Matlabem), ISBN 80-01-02435-0, Vydavatelstv CVUT, pp.
376, Praha 2002.

OKROUHLIK, M.  LUNDBERG, B.: Assessment of Rock Drilling
Efficiency Based on Theoretical and Numerical Stress Wave
Considerations, Proceedings of the Second International Conference
on Advances in Structural Engineering and Mechanics, Session W4A,
p. 34, ISBN 89-89693-05-5 93530, Busan, Korea, August 21  23,
2002

LUNDBERG, B.  OKROUHLIK, M.: Approximate Transmission Equivalence
of Elastic Bar Transitions Under 3-D Conditions, Journal of Sound
and Vibration, Vol. 256, No. 5, pp. 941-954, 2002

OKROUHLIK, M.  PTAK S.: Pollution-Free Energy Production by a
Proper Misuse of Finite Element Analysis, Engineering
Computations, Vol. 20, No. 5/6, pp.601  610, 2003
```

OKROUHLIK, M. PTAK S.: Numerical Modeling of Axially Impacted Rod with a Spiral Groove, Engineering Mechanics, Vol. 10, No. 5, pp. 359 374, 2003.

HOSCHL, C. OKROUHLIK, M.: Solution of Systems of Nonlinear Equations, Strojncky casopis, Vol. 54, No. 4, pp. 197 227, 2003.

OKROUHLIK, M. PTAK, S.: Assessment of experiment by finite element analysis: Comparison, self-check and remedy, Strojncky casopis, Vol. 56, 2005, No. 1.

LUNDBERG, B. OKROUHLIK, M.: Efficiency of percussive rock drilling with consideration of wave energy radiation into the rock. Int. Journal of Impact Engineering. 32 (2006) 1573 - 1583.

OKROUHLIK, M., PTAK, S., LUNDBERG, B.: Wave Energy Transfer in Percussive Rock Drilling. 6th European Solid Mechanics Conference, Budapest, 28 August to 1 September 2006.

OKROUHLIK, M.: Computational limits of FE transient analysis, The Seventh International Conference on Vibration Problems ICOVP 2005, Springer Proceedings in Physics, Turkey, Istanbul, 05 -09 September 2005, pp. 357 369.

OKROUHLIK, M.: When a 'good agreement' is not enough. In: Sbornk prednek z V. mezinrodn konference Dynamika tuhch a deformovatelnch teles. Univerzita Jana Evangelisty Purkyne v st nad Labem. 3. 4. rjna 2007. ISBN 80-7044-914-1.

GABRIEL, D. - PLESEK, J. - VALES, F. - OKROUHLIK, M.: Symmetry preserving algorithm for a dynamic contact-impact problem., III European Conference on Computational Mechanics. Dordrecht : Springer, 2006, Lisabon, s. 1-7. ISBN 1-4020-4994-3.

OKROUHLIK, M. - PTAK, S.: Torsional and bending waves in an axially impacted rod with a spiral groove. Euromech solid mechanics /5./ : Book of abstracts. Thessaloniki: Aristotle university of Thessaloniki, 2003 - s. 145-145.

PTAK, S. - TRNKA, J.- OKROUHLIK, M. - VESELY, E. - DVORAKOV, P.: Comparison of numerically modelled and measured responses of a hollow cylinder under impact loads. Euromech solid mechanics conference: book of abstracts. Thessaloniki : Aristotle University of Thessaloniki, 2003 - s. 144-144

OKROUHLIK, M.: From the Gauss elimination, via GRID, to kvantum computers. Seminar of Parallel programming 2: Abstracts. Praha,

```
Ustav termomechaniky AV CR, 2002. s. 1-2.

OKROUHLIK, M.: Limits of computability within the scope of
computational mechanics. Vydavatelstvi CVUT, ISBN
978-80-01-03898-7, 2007.

OKROUHLIK, M., PTAK, S., VALDEK,U.: Self-assessment of Finite
Element Solutions Applied to Transient Phenomena in Solid
Continuum Mechanics, To be published in Engineering Mechanics,
```

- **E-mail address:** ok@it.cas.cz

Petr Pařík

is the author of Chapter 11.

- **First name, Family name:** Petr Pařík
- **Titles:** Ing. (M.Sc.)
- **Affiliation:** Institute of Thermomechanics, Academy of Sciences of the Czech Republic, 182 00 Prague, Dolejškova 5, the Czech Republic
- **Field of expertise:** Finite Element Method, Continuum Mechanics.
- **E-mail address:** petr.parik@seznam.cz

Svatopluk Pták

is the author of Chapter 10.

- **First name, Family name:** Svatopluk Pták
- **Titles:** Ing. CSc.
- **Affiliation:** Institute of Thermomechanics, Academy of Sciences of the Czech Republic, 182 00 Prague, Dolejškova 5, the Czech Republic
- **Field of expertise:** Finite element method, Computational mechanics, Mechanics of solids
- **Selected publications:**

```
Dobias, J., Ptak, S., Dostal, Z., Vondrak, V.:  Total FETI Based
Algorithm for Contact Problems with Additional Nonlinearities,
Advances in Engineering Software

Dobias, J., Ptak, S., Dostal, Z., Vondrak V.: Domain Decomposition
Based Contact Solver Proceedings of the Eight International
Conference on Computer Methods and Experimental Measurements for
Surface Effects and Contact Mechanics, (Editors: Hosson, Brebbia,
Nishida), New Forest, UK, 16-18 May, 2007, pp 207-216 WIT Press,
UK.

Dobias, J., Ptak, S., Dostal, Z., Vondrak, V.: Dynamic Nonlinear
TFETI Domain Decomposition Based Solver Proceedings of the 11th
International Conference on Civil, Structural and Environmental
Engineering Computing, (Editor: Topping) St. Julians, Malta,
18-21. Sept. 2007, Paper No. 20, pp 1-12 Civil-Comp Press, UK.
ISBN 978-1-905088-16-4

Vondrak, V., Dostal, Z., Dobias, J., Ptak, S.: A FETI Domain
Decomposition Method Applied to Contact Problems with
Large Displacements, in: Widlund, Keyes (eds.): Lecture
Notes in Computational Science and Engineering, Vol. 55, Domain
Decomposition Methods in Science and Engineering XVI, pp 771-778,
Springer-Verlag, Heidelberg, 2007. ISBN
987-3540-34468-1

Okrouhlik, M., Ptak, S., Lundberg, B.: Wave Energy Transfer
in Percussive Rock Drilling  6th European Solid Mechanics
Conference, Budapest, 28 August - 1 September 2006.

Okrouhlik, M., Ptak, S.: Assessment of experiment by finite
element analysis: Comparison, self-check and remedy Journal of
Mechanical Engineering (2005), Vol. 56, No. 1, pp 18-39 Strojnicky
casopis
```

```
Okrouhlik, M., Ptak, S.: Numerical modelling of axially impacted
rod with a spiral groove Engineering Mechanics (2003), Vol. 10,
No. 5, pp 359-374 Inzenyrska mechanika

Okrouhlik, M., Ptak, S.: Pollution-free energy production by means
of a proper misuse of finite element analysis  Engineering
Computations (2003), Vol. 20, No. 5, pp 601-610
```

- Email address: svata@it.cas.cz

Vítězslav Štembera

is the author of Chapter 12.

- **First name, Family name:** Vítězslav Štembera
- **Titles:** Mgr.
- **Affiliation:** Mathematical Institute of the Charles University, Prague, Czech Republic
- **Field of expertise:** Computer Modeling, Fluid Mechanics, Numerical Mathematics
- **Selected publications:**

  ```
  Stembera V., Marsik F., Chlup H.: One-Dimensional Mathematical
  Model of the Flow Through a Collapsible Tube with
  Applications to Blood Flow Through Human Vessels,
  Conference Engineering Mechanics 2005, ISBN 80-85918-93-5
  ```

- Email address: vitastembera@hotmail.com

Index