Contents

Chapter 1

Introduction

Welcome to the Manual of the Hexaheadral Finite Element Solver (hexfem). The program implements a hexahedral element with incompatible displacement modes [1, 2]. The hexahedral element implemented is strictly ressembling the ANSYS *Solid 45* element. This program shall serve both as an educational tool to teach about the concepts of a finite element solver as well as its numerical implemetation. As such the code was designed to be short and efficent. It does not use any external libraries besides the C library. It implements the finite elements, a sparse matrix system, and a sparse matrix conjugate gradient with jabocbi preconditioning. The solver is further available in vectorized forms for the SSSE3 and AVX2 instruction sets found on computers using the X86_64 instruction set.

The following manual has three sections:

1. Basic Usage: All the necessary to use this finite element solver, without thinking to much about it.
2. Theory: The theory that makes such a tool possible. This shall not serve as a finite element course replacement but rather as a summery of what is implemented herin, and provide an overview of what is necessary to get such a tool baked together.
3. Code Algorithms: Some hints for those who want to mess with the code of this tool, expand it, modify it for their own purpose.

The license of the software:

Chapter 2

Basic Usage

Within this chapter you will find all the necessary to install and use this tool for your own problems. This tool implements the very basic of Finite Element Theory, and only one specific element so far. Nevertheless you might find it interesting to use if for your own purposes. The author however encorages you to read at least the theory section in order to know what you are actually doing.

2.1 Installation - Compiling the code

2.1.1 On GNU-Linux/BSD/Unix Systems

The program has been programmed with portablitiy in mind. As this is kind of an educatinal tool incorporating every thing one needs to make a finite element calculation this tool does not require any external library besides a standard C library.

In order to install the program one currently needs, make (GNU make 4.1 has been tested), a suiteable c compilier (gcc 4.9.3 and clang 3.5.0 have been tested), and c library (glibc 2.21 has been tested).

The supplied makefile can be adapted to suite compilier and necessary compilier flags on your system and should work out of the box on common GNU/Linux systems having the listed libraries installed.

Issuing make in the directory containing this program should suffice to build the hexfem binary.

2.1.2 On Windows

Windows binaries can be compiled using the MingW64 toolchain.

2.2 The hexfem input file

Runnig this program, an input file is required. The input file is modeled after the ANSYS input file syntax but has some significant differences that are outlined below. The input file has to be composed of the following statements:

- `N, [node_ID], [X], [Y], [Z]`
  Node definition: Defines a node in the the mesh with a corresponding unique identifier [node_ID], at the coordinates [X], [Y], [Z]. Each node can only to appear once in the input file. Double definitions where the last definition is taken into account as in ANSYS are not implemented and result in corrupt results. A node has to be defined before force or constraint definitions acting on that node.
- `Mat, [material_ID]`
  Material pointer: points to a material with a unique id. Each material pointer can only be defined once. Double usage of the same statements is per se not allowed in the input file. The latest material pointer in front of an element specification is used in order to attribute the corresponding material to that element.
- `MP, EX, [material_ID], [value]`
  Young's modulus: Defines the Youngs modulus of the material indicated by the [material_ID] statement. A young's modulus can be located anywhere in the input file. Materials are so far limited to an istropic Youngs modulus, and hence the EX field is valid for all directions.
- `MP, NUXY, [material_ID], [value]`
  Poisson's ratio: Defines Poisson's ratio of the material. Such statement can just as the Young's modulus be defined anywhere within the input file. Due to isotropicity the NUXY field defines the Poisson's ratio for all directions.
- `E, [N], [N], [N], [N], [N], [N], [N], [N]`
  Element definition: Defines a hexahedral element using the nodes [N], where [N] is the [node_ID] of a node defined in the input file. Each element specification obtains the material indicated by a Material pointer in front of it. Otherwise these statements can be placed anywhere in the input file.
- `D, [node_ID], ALL, 0` Displacement constraint: Defines a constraint on a node. Currently only removing all degrees of freedom and hence, fixing a nodes position is implemented. The statement requires the node with [node_ID] to be already defined.

— `F, [node_ID], [Direction], [value]` Force definition: Defines the external force to be applied on a node. The [Direction] field can be any of FX, FY, FZ in order to specify specfic components. This statement can only occur after the node with [node_ID] has been specified in the input file.
— `ZOU, [specification]` Output specification: Defines the values shall be calculated and printed. The [specification] field can currently be any of: DIS for nodal displacemnts, FOR for forces exercised on nodes, STE for an output of a strain tensor for each element, PST for an output of principal strains for each element, VOB for the volumes of each element before deformation and VOA for the volumes of each element after deformation. VOA, VOB, STE and PST are requiring additional computational time, while DIS and FOR are always calculated, but not printed if the statements are missing.

A sample input file is shown in listing 2.1.

```
N, 1, 0, 0, 0
N, 2, 1, 0, 0
N, 3, 1, 1, 0
N, 4, 0, 1, 0
N, 5, 0, 0, 1
N, 6, 1, 0, 1
N, 7, 1, 1, 1
N, 8, 0, 1, 1
N, 9, 0, 0, 2
N, 10, 1, 0, 2
N, 11, 1, 1, 2
N, 12, 0, 1, 2

D, 1, ALL, 0
D, 2, ALL, 0
D, 3, ALL, 0
D, 4, ALL, 0

F, 9, FZ, -5.
F, 10, FZ, -5.
F, 11, FZ, -5.
F, 12, FZ, -5.

MAT, 1
MP, EX, 1, 300.
MP, NUXY, 1, 0.4

E, 1, 2, 3, 4, 5, 6, 7, 8
E, 5, 6, 7, 8, 9, 10, 11, 12

ZOU, DIS
ZOU, FOR

ZOU, STE
ZOU, PST

ZOU, VOB
```

```
ZOU, VOA
```

Listing 2.1 – Input file containing a brick made up of two elements that is locked at its base and under load from the top

2.3 Running the Program/Program Ouput

Successful compilation of the program results in a binary file named hexfem. This file can then be run using a correctly constructed input file. It will print the requested outputs to the standard output on the terminal.

A command like the following may be invoked to execute the program:

```
./hexfem [Inputfile]
```

Listing 2.2 – Running the finite element code

Here [Inputfile] is the input file in the syntax specified.

As the output is directly printed to *standard out*, it is suggested to redirect the standard output to a file using the capabilities that the different operating system shells provide.

Chapter 3

Theory

3.1 Static Finite Element Analyses

This hexaheardral finite element solver implements the so called *direct stiffness appoach* [3]. At the center of this approach lies the solution of the linear sparse system:

$$K_{ij}u_j = F_i \tag{3.1}$$

where K is the stiffness matrix, u the displacement vector and F the vector of applied forces. For n nodes i and j ranges from $1 \leq i, j \leq 3n$ as each node has three degrees of freedom. The problem can thus be solved in finding the appropriate stiffness matrix K for the mesh in question. Solve the linear system (3.1) one obtains the displacements of each node. These displacements allow us then calculate strains, stresses as well as other linked results with relative ease.

3.1.1 Determination of the Stiffness Matrix

As we sketch the algorithm, we start with the description of how we derive the stiffness matrix for a system. The basic idea here is that one can determine a local stiffness matrix for each finite element in the system. Once such a local stiffness matrix is obtained, it can, due to linearty, be summed into the global stiffness matrix. The derterminination of the local stiffness matrix requires several fine tricks. The first such trick is to transpose the element in question into an easy to integratable local coordiante system ξ, η, ζ where each element represents a perfect cubic shape with an edgelength of 2, ranging from $-1 \leq \xi, \eta, \zeta \leq 1$. The coordinate transformations for the element are then described by so called shape functions. As we only treat

hexaheadral elements with this solver the shape functions are given by the following equations:

$$\begin{aligned} h_1 &= \frac{1}{8}(1-\xi)(1-\eta)(1-\zeta), \\ h_2 &= \frac{1}{8}(1+\xi)(1-\eta)(1-\zeta), \\ h_3 &= \frac{1}{8}(1+\xi)(1+\eta)(1-\zeta), \\ h_4 &= \frac{1}{8}(1-\xi)(1+\eta)(1-\zeta), \\ h_5 &= \frac{1}{8}(1-\xi)(1-\eta)(1+\zeta), \\ h_6 &= \frac{1}{8}(1+\xi)(1-\eta)(1+\zeta), \\ h_7 &= \frac{1}{8}(1+\xi)(1+\eta)(1+\zeta), \\ h_8 &= \frac{1}{8}(1-\xi)(1+\eta)(1+\zeta), \end{aligned} \tag{3.2}$$

together with the coordinate transformation:

$$x = \sum_{i=1}^{8} x_i h_i, \qquad y = \sum_{i=1}^{8} y_i h_i, \qquad z = \sum_{i=1}^{8} z_i h_i, \tag{3.3}$$

where x_i,y_i,z_i are the coordinates of the hexheadral element's nodes in the global reference frame.

So called non conforming displacement nodes, introduce three further functions which are of special interest:

$$p_1 = (1-\xi^2), \qquad p_2 = (1-\eta^2), \qquad p_3 = (1-\zeta^2). \tag{3.4}$$

These functions are not coordinate transformations in the sens that they are not reltated to any node. They can be interpreted as a deformation of the element in a parabobla-like way along the ξ,η or ζ axes. Therefore, they are further known by the name *bubble functions* in literature. These functions are necessary to avoid the so called *shear locking* phenomena in the mesh [1].

Th shape functions allow us to evaluate the Jacobian J and its inverse J^{-1}:

$$J = \frac{\partial(x,y,z)}{\partial(\xi,\eta,\zeta)}, \qquad J^{-1} = \frac{\partial(\xi,\eta,\zeta)}{\partial(x,y,z)}, \tag{3.5}$$

which, looking at the edges of the elements (defined by the nodes) where both x,y,z and ξ,η,ζ are known, is found to be:

$$J = \begin{bmatrix} \sum_{i=1}^{8} x_i \frac{\partial h_i}{\partial \xi} & \sum_{i=1}^{8} y_i \frac{\partial h_i}{\partial \xi} & \sum_{i=1}^{8} z_i \frac{\partial h_i}{\partial \xi} \\ \sum_{i=1}^{8} x_i \frac{\partial h_i}{\partial \eta} & \sum_{i=1}^{8} y_i \frac{\partial h_i}{\partial \eta} & \sum_{i=1}^{8} z_i \frac{\partial h_i}{\partial \eta} \\ \sum_{i=1}^{8} x_i \frac{\partial h_i}{\partial \zeta} & \sum_{i=1}^{8} y_i \frac{\partial h_i}{\partial \zeta} & \sum_{i=1}^{8} z_i \frac{\partial h_i}{\partial \zeta} \end{bmatrix}. \tag{3.6}$$

Using d'Alemberts variational calculus one has the possibility to show from physical principals that [3]:

$$K = \int_V B^T D B dV, \tag{3.7}$$

where V is the volume occupied, B is the strain displacement relation which we will be describe in the following. D is the stress strain relation, which can be derived from Hooke's law [4] in three dimensions and is given by:

$$\begin{aligned} \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{13} \end{bmatrix} &= \Gamma \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) \end{bmatrix} \begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ 2\epsilon_{13} \\ 2\epsilon_{23} \\ 2\epsilon_{13} \end{bmatrix} \\ &= D \begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ 2\epsilon_{12} \\ 2\epsilon_{23} \\ 2\epsilon_{13} \end{bmatrix}, \end{aligned} \tag{3.8}$$

with the prefactor $\Gamma = \frac{E}{(1+\nu)(1-2\nu)}$. Here E is the Young's modulus and ν the Poisson material constant. The strain displacement relation can be obtained from the partial derivatives of the shape function:

$$\begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ 2\epsilon_{23} \\ 2\epsilon_{13} \\ 2\epsilon_{12} \end{bmatrix} = \begin{bmatrix} \frac{\partial h_i}{\partial x} & 0 & 0 \\ 0 & \frac{\partial h_i}{\partial y} & 0 \\ 0 & 0 & \frac{\partial h_i}{\partial z} \\ \frac{\partial h_i}{\partial y} & \frac{\partial h_i}{\partial x} & 0 \\ \frac{\partial h_i}{\partial z} & 0 & \frac{\partial h_i}{\partial y} \\ \frac{\partial h_i}{\partial z} & 0 & \frac{\partial h_i}{\partial x} \end{bmatrix} \begin{bmatrix} u_{i,x} \\ u_{i,y} \\ u_{i,z} \end{bmatrix} = B \begin{bmatrix} u_{i,x} \\ u_{i,y} \\ u_{i,z} \end{bmatrix}. \tag{3.9}$$

In (3.9) i includes all the shape functions going from 1 to 8, which creates a repeated pattern in the matrix. Hence, for a single hexahedral element B

is a 24 by 6 matrix. In the same way $u_{i,xyz}$ is a 24 component vector. The operators needed in order to evaluate the partial derivatives in equation (3.9) can be obtained by analytically deriving the shape functions shown in (3.2) and in calculating the inverse of the Jacobian matrix (3.6):

$$\begin{bmatrix} \frac{\partial p_i}{\partial x} \\ \frac{\partial p_i}{\partial y} \\ \frac{\partial p_i}{\partial z} \end{bmatrix} = J^{-1}(\xi, \eta, \zeta) \begin{bmatrix} \frac{\partial p_i}{\partial \xi} \\ \frac{\partial p_i}{\partial \eta} \\ \frac{\partial p_i}{\partial \zeta} \end{bmatrix} \tag{3.10}$$

Writing down these matricies we did not account for the supplimentary bubble functions in equation (3.4).

As the bubble functions are not connected to any nodes, a trick called static condensation is used in order for them to be affective. This allows us to take the virtual displacement coordinates a and the virtual displacement matrix P into account. Where a is the virtual analogue to displacements u and the matrix G to the strain-displacement relation B. As there are three bubble functions (3.4) three further virtual displacement coordinates a are introduced, resulting in 9 virtual degrees of freedom. For a single element G is hence, a 9 by 6 matrix, taking for three components the same form as B. To clarify this let us rewrite (3.7) taking these virtual displacents into account:

$$K = \begin{bmatrix} \int_V B^T D B dV & \int_V B^T D G dV \\ \int_V (B^T D G)^T dV & \int_V G^T D G dV \end{bmatrix}, \tag{3.11}$$

with,

$$G = \begin{bmatrix} \frac{\partial p_i}{\partial x} & 0 & 0 \\ 0 & \frac{\partial p_i}{\partial y} & 0 \\ 0 & 0 & \frac{\partial p_i}{\partial z} \\ \frac{\partial p_i}{\partial y} & \frac{\partial p_i}{\partial x} & 0 \\ \frac{\partial p_i}{\partial z} & 0 & \frac{\partial p_i}{\partial y} \\ \frac{\partial p_i}{\partial z} & 0 & \frac{\partial h_i}{\partial x} \end{bmatrix}. \tag{3.12}$$

Further as pointed out by R. L. Taylor et al. in [2] in order for the elements to pass the patch test G should not influence the volume of the element. The calculation of G requires just as the calculation of B terms that result from the inverse Jacobian 3.6. The trick here is to take the Jacobien at the center of the element, at $\xi = \eta = \zeta = 0$ calculating $\frac{\partial p_i}{\partial x}$, $\frac{\partial p_i}{\partial y}$ and $\frac{\partial p_i}{\partial z}$. Symbolically

written down to be:

$$\begin{bmatrix} \frac{\partial p_i}{\partial x} \\ \frac{\partial p_i}{\partial y} \\ \frac{\partial p_i}{\partial z} \end{bmatrix} = J^{-1}|_{\xi=\eta=\zeta=0} \begin{bmatrix} \frac{\partial p_i}{\partial \xi} \\ \frac{\partial p_i}{\partial \eta} \\ \frac{\partial p_i}{\partial \zeta} \end{bmatrix} \frac{|J(\xi=\eta=\zeta=0)|}{|J(\xi,\eta,\zeta)|}, \tag{3.13}$$

with the correction factor $\frac{|J(\xi=\eta=\zeta=0)|}{|J(\xi,\eta,\zeta)|}$.

The bubble functions p shall in no further way influence the coordinate transformation and hence, the jacobian is always calculated by use of the classical shape functions (3.2). The jacobian is always varied when used in evaluating B or as a determinant in the volume element dV of equation. It is only kept constant $\xi = \eta = \zeta = 0$ at calculating the partial derivatives: $\frac{\partial p_i}{\partial x}$, $\frac{\partial p_i}{\partial y}$ and $\frac{\partial p_i}{\partial z}$ for equation (3.11) using (3.13).

Static condensation now consists of matrix coefficient modification. Looking at equation (3.11) the trick here is to turn the lower part into the identity matrix. Hence, the system takes the form:

$$K^* = \begin{bmatrix} \left[\int_V (B^T D B) dV\right]_{G-inc} & 0 \\ 0 & I \end{bmatrix}, \tag{3.14}$$

where using partial guassian elemination the lower part is transfomed into the identity matrix I modifying the coefficients in the $\int_V B^T D B dV$ part of the matrix, incorporating the virtual stress-strain terms in the matrix [5]. Looking at equation $K_{ij}u_j = F_i$ (3.1) this is possible as we are not interested in virtual displacements a that as the force acting on this points can only be 0 as we only let the force act on real nodes. $K^* = \left[\int_V (B^T D B) dV\right]_{G-inc}$ for local elementary stiffness matrices can then due to linearity directly summed into the global stiffness matrix at positions effecting the correct nodal displacemnts u.

The static condensation algorithm further also impacts the strain-displacement relation, B. E. L. Wilson noted this [5] and provided an algorithm in order to manipluate the stress-displacement relation denoted in the paper as A. In our tests the implementation shown however did not result in the same results as ANSYS would print out. We hence, based ourselfs on a different method noting that the static condensation can be seen expressed as the application of a matrix M in such that:

$$\begin{aligned} K^* = MK &= M \begin{bmatrix} \int_V B^T D B dV & \int_V B^T D G dV \\ \int_V (B^T D G)^T dV & \int_V G^T D G dV \end{bmatrix} \\ &= \begin{bmatrix} \left[\int_V (B^T D B) dV\right]_{G-inc} & 0 \\ 0 & I \end{bmatrix}, \end{aligned} \tag{3.15}$$

as we can see from equations (3.11,3.14). M is thus a partial inverse of the matrix K which can be obtained by the traditional gauss jordan elemination method with an extended identity matrix. The only difference is here that the elemination is stopped once the lower left section of K given by $\int_V G^T DGdV$ is transformed into the identity matrix.

Knowing M, the condensated B used for further stress ϵ or strain σ. These evalutions can be obtained using the following formula:

$$B^T_{G-inc} = M[BG]^T, \tag{3.16}$$

where $[BG]$ discribes the a 33 by six matrix which contains B extended by G components. Matrix multiplaction can be stopped so that B_{G-inc} results in a 24 by six matrix. The static condensation [5] paper evokes that this step could be done without the matrix multiplication, and by deducing the values directly during the Gauss Jordan elemination process. As $[BG]$ changes for every point in an element M stays static. Performing the reduction during the elemination process would have meant to change the eight B matrices for one element at the same time. We believe that it is more advantageous and intuitive to store this information once in matrix M.

3.1.2 Integration, Volume and Mean

Integration in all our equations is effectuated using a gaussian quadrature approximation, in the ξ, η, ζ coordinate system. Integration over an element is thus numerically performed in the follow way:

$$\begin{aligned}\int_V A(x,y,z)dV &= \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} A(\xi,\eta,\zeta)|J|d\xi d\eta d\zeta \\ &\approx \sum_{i=1}^{8} A(r_i)|J(r_i)|,\end{aligned} \tag{3.17}$$

where $|J|$ is the determinant of the Jacobian. A is scalar, vector or tensor field to be integrated and r_i are the gaussian quadrature points found to be:

$$\begin{aligned} r_1 &= \frac{1}{\sqrt{3}}[-1,-1,-1]^T \\ r_2 &= \frac{1}{\sqrt{3}}[1,-1,-1]^T \\ r_3 &= \frac{1}{\sqrt{3}}[1,1,-1]^T \\ r_4 &= \frac{1}{\sqrt{3}}[-1,1,-1]^T \end{aligned}$$

$$
\begin{aligned}
r_5 &= \frac{1}{\sqrt{3}}[-1,-1,1]^T \\
r_6 &= \frac{1}{\sqrt{3}}[1,-1,1]^T \\
r_7 &= \frac{1}{\sqrt{3}}[1,1,1]^T \\
r_8 &= \frac{1}{\sqrt{3}}[-1,1,1]^T.
\end{aligned} \tag{3.18}
$$

in ξ, η, ζ coordinates. Using this numerical approximation the volume of a hexahedral cell casts into:

$$V = \sum_{i=1}^{8} |J(r_i)| \tag{3.19}$$

In order to obtain numerical compatibility stress and strain are values are currently exempt from this procedure and are simply meaned without taking the volume into account and hence, are given by:

$$\bar{\epsilon} = \frac{1}{8} \sum_{i=1}^{8} \epsilon(r_i). \tag{3.20}$$

3.1.3 Solution of a Linear System

The system to be solved is in general a linear sparse symmetric system, as highlighted by equation (3.1).

As a linear, operator in our case the n by n stiffness matrix K, forms an inner product. Hence, one can define a base of n dimensional space associated with this inner product. Let us denote:

$$p_1, p_2, \ldots, p_n, \tag{3.21}$$

a set of orthonormal vectors in respect to the inner product:

$$\langle u|v\rangle_K = u^T K v \tag{3.22}$$

to be a base of our n dimensional space. We can then develop a solution for our problem $Ku = F$, in this base:

$$u = \sum_{i=1}^{n} \alpha_i p_i \tag{3.23}$$

and further:

$$Ku = \sum_{i=1}^{n} \alpha_i K p_i \tag{3.24}$$

$$p^T K u = \sum_{i=1}^{N} \alpha_i p_k^T K p_i \tag{3.25}$$

$$p_k^T F = \sum_{i=1}^{N} \alpha_i \langle p_k | p_i \rangle_K \tag{3.26}$$

$$\tag{3.27}$$

from where we obtain, the coefficients:

$$\alpha_k = \frac{p_k^T F}{p_k^T K p_k} \tag{3.28}$$

which means that a solution for u can be found in finding a set of orthonormal (conjugate) vectors according to our n dimensional space with the inner product defind by the stiffness matrix, and further calculating the coefficients α_k.

Using this as an iterative method in our case we define a residual at step k to be:

$$r_k = F - K u_k. \tag{3.29}$$

From where, like at Gram Schmitt Orthonormalisation we want to find a vector normal on the previous vectors:

$$p_k = r_k - \sum_{i<k} \frac{p_i^T K r_k}{p_i^T K p_i} p_i. \tag{3.30}$$

The improved solution is then given by:

$$u_{k+1} = u_k + \alpha_k p_k. \tag{3.31}$$

This works as due to the normalisation step p_k is orthogonal to u_{k-1}, according to the inner product defined by K.

As r_{k+1} is in fact orthogonal to all p_i the sum is not needed and the algorithm can be very efficiently implemented by only knowing previous r_k,p_k, and u_k. We initialize with:

$$r_0 = F - K u_0, \tag{3.32}$$

$$p_0 = r_0, \tag{3.33}$$

$$\tag{3.34}$$

use the recurrence relations:

$$\begin{aligned} \alpha_k &= \frac{r_k^T r_k}{p_k^T K p_k}, && (3.35)\\ u_{k+1} &= u_k + \alpha_k p_k, && (3.36)\\ r_{k+1} &= r_k - \alpha_k K p_k, && (3.37)\\ \beta_k &= \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}, && (3.38)\\ p_{k+1} &= r_{k+1} + \beta_k p_k, && (3.39) \end{aligned}$$

and repeat until r_k is sufficiently small. The convergence of the algorithm outlined here largely depends on the properties of the system to be solved. In order to speed up convergence a so called preconditionning matrix M can be used which results in the solution of the equivalent linear system:

$$\begin{aligned} E^{-1}K(E^{-1})^T\hat{u} &= E^{-1}F, && (3.40)\\ EE^T &= M, && (3.41)\\ \hat{u} &= E^T u. && (3.42) \end{aligned}$$

This changes the algorithm outlined above to an initialisation using:

$$\begin{aligned} r_0 &= F - Ku_0, && (3.43)\\ z_0 &= M^{-1}r_0 && (3.44)\\ p_0 &= z_0, && (3.45)\\ & && (3.46) \end{aligned}$$

and the recurrence relations to:

$$\begin{aligned} \alpha_k &= \frac{r_k^T z_k}{p_k^T K p_k}, && (3.47)\\ u_{k+1} &= u_k + \alpha_k p_k, && (3.48)\\ r_{k+1} &= r_k - \alpha_k K p_k, && (3.49)\\ z_{k+1} &= M^{-1} r_{k+1}, && (3.50)\\ \beta_k &= \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}, && (3.51)\\ p_{k+1} &= z_{k+1} + \beta_k p_k, && (3.52) \end{aligned}$$

The preconditioning matrix M can be more or less complicated. In the optimal case M^{-1} would be the inverse of K. In our implemented algorithm we used the so called Jacobi precontionning method setting M to the diagonal of K, and 0 elsewhere. Even though this method is almoast trivial and straightforward to implement it is suggested to be one of the fasted methods on SIMD (vector) systems [6].

Chapter 4

Code Algorithms

In this chapter we make some remakes about how things are organised in the current program, which data structures we use, and which choices we had to make during the implementation in order to save memory and computational time.

Further the code shall be well anotated using doxygen processable annotations. This should help you adapting the code to your needs and to understand in great detail each function that is implemented in this code.

Currently the program consists of the following different code files.

— mesh_from_file.c: This file contains the implementation of the input file parser, and the code to create the internal data structures from the input file.

— shape_func.c: The definitions of the shape as well as the bubble functions is found within this file. Further the partial derivatives in both ξ,η,ζ and x,y,z space are defined here.

— matrix_math.c: This file contains small helper functions in order to calculate the determinants, the inverses and eigenvalues of three by three matrices, which comes handy as we calculate the jacobian.

— node_shuffle.c: This file generates hints on structural information of the global stiffness matricies, that are needed later on for correct memory allocation etc. This is done using so called index matricies.

— sparse_matrix.c: Herein one finds all functions that implement sparse matricies in this algorithm.

— stiffness_matrix_for_element.c: All the code in order to calculate the stiffness matrix for a single element is found here. This includes the Jacobian matrix, the strain displacement as well as the stress strain relation. Further the virtual strain displacement relation that is caused by the bubble functions can be found in this file.

— global_stiffness_matrix.c: This code works with the global stiffness

matrix, constructed from summing the elementary stiffness matrix into this global matrix.

— post_processing.c: In this file all the post processing code is found. Currently this includes the elementary volume, strain calculation and principal strains.

— cg_* These files implement the conjugate gradient solver on different SIMD architectures and in scalar form.

4.1 Data Structures

When the program starts the input file is read and the mesh and boundary conditions are translated into the programs internal data structures that we outline in the following.

We start listing the node structure:

```
typedef struct {
  int id;      /*!< The node id as stated in the input file */
  int fixed;   /*!< 0 this node has three degrees of freedom 1 has none */
  double x[3]; /*!< The initial positions as read from the input file */
  double u[3]; /*!< The displacements of each node */
  double f[3]; /*!< The forces applied to these nodes (fixed ones) */
} node;
```

Listing 4.1 – Node structure

where displacements and forces at fixed nodes are not defined at input, but calculated during the FEM process. These values are then further used for post-processing.

The next structure to be outlined is the element structure which has the following form:

```
typedef struct {
  int* node_ids;              // node ids as in input file in element
  int* node_indicies;         // node array indicies
  double E_modulus;           // E modulus ( if available )
  double G_modulus;           // G modulus ( if available )
  double poisson_number;      // poisson_number if available
} element;
```

Listing 4.2 – Element structure

As each element is built out of eight nodes we store their ids in the node_ids array. A difference does exist between node_ids and node_indices in that the indices are the actual array indices in the structure structure which follows later on. This allows for rapid memory access to these nodes. E modulus and G modulus are the elasticity and shear modulus. The poisson_number is read just as the E_modulus directly from the input file.

Finally we have the structure super structure which points to the node and element structure arrays and hence, allows us to pass the whole object to

be investigated during our finite element calculation in a very simple fashion. This structure is defined in the following way:

```
typedef struct {
  node* nodes;       /*!< array of nodes */
  element* elements; /*!< array of elements */
  int n_nodes;       /*!< number of nodes */
  int n_elements;    /*!< number of elements */
  postprocessed_outputs outputs; /*!< structure telling what to output*/
} structure;
```

Listing 4.3 – Structure structure

4.2 Sparse Matrix Implementation

In order for our program to be efficient we needed to have an efficient sparse matrix implementation. In general sparse matrices are implemented with three arrays of row indices, colomn indices and associated values where zeros are by definition not stored. A big problem is hence finding the value in these arrays that is associated to a given row and column index. In our implementation we use a binary matrix which stores in a very efficient manner only 0 and 1 to avoid the lookup of undefined values. Further we use sorting to order our sparse matrices and a bisection algorithm for efficient value lookup. This method improved the speed of the algorithm by several magnitudes compared to initially implemented linear search.

4.2.1 Sparse matrix strucutre

Sparse matrices are defined by the following structure:

```
typedef struct {
  int range;         /*!< range of the square sparse matrix*/
  int non_zeros;     /*!< number of non zero entries in the square sparse m*/
  int triangle_type; /*!< 0 = both triangles, 1 = upper -, 2 lower trianlge */
  int* row_indicies; /*!< row indicies array*/
  int* col_indicies; /*!< column indicies array*/
  double* values;    /*!< values array*/
  int has_eye;       /*!< does the matrix have a binary index */
  int has_node_eye;  /*!< does the matrix have a binary index per node (3x3) */
  char* eye;         /*!< the pointer to the binary eye*/
  char* node_eye;    /*!< the pointer to the node eye*/
} sparse_matrix;
```

Listing 4.4 – Sparse matrix structure

where range is the range of the matrix (all our sparse matrices are square matrices). non_zeros is the internally computed number of non zero elements in the matrix. Triangle_type is currently not used in our program, but could further improve the speed and memory footprint in storing only half sparse

matrices. Then we have an eye and a node eye. The difference here is that the node eye only stores the binary information taking nodes into account the binary matrix hence has a range of only one third of the range of the complete matrix, which further can save memory in certain conditions. Eye is the eye where a binary index exists for the whole matrix.

4.2.2 Binary Sparse Eye

The smallest defined data type by C standards up to C11 is the character. Which on most platforms is a 8 bit integer value. Using the character value setting it to 1 or 0 is hence very memory inefficient, as we would store eight bits for a single binary value. We hence use C's binary operators, currently on the character data type to access all binary values. In the future we might switch to an other data type, staying with the same schemes as the retrieval and manipulation of special integer data types shall be way faster then the character data type. Let us sketch how this actually works highlighting the retrieval function for a binary value.

```
int get_value_in_binary_index_at_index(char* bin_index_matrix, long index) {

  char small_index;
  int module = index%(sizeof(char)*8);
  int location = index/(sizeof(char)*8);

  small_index = (char)1 << module;

  return (small_index == (bin_index_matrix[location] & small_index));
}
```

Listing 4.5 – retrieval of a binary value

This function has two arguments the matrix where to retrieve the binary value in, and the index where to lookup if we have a value at this point or whether the matrix element equals to zero. The index is composed in the typical index = row $*$ range + column linearization formula that is used everywhere throughout our code. As we use the character data type here we can only retrieve eight binary values at the same time. We divide the index by eight in order to find the correct array offset from where to recover these eight binary values. Then we take the modulus of the index which highlights the actual element of the eight retreaved values to be investigated. We use the binary shift left operator in combination with the modulus in order to generate eight bits with a 1 at the position that we want to investigate. In the code sample highlighted in listing 4.5 this value is stored in small_index. Once this is done we use the binary *and* operator and compare its result to the number generated. If the numbers are the same, which means that 1 occurs at the position to be investigated this comparison yields true (integer

1 in C terminology) and false (integer 0) if 0 occurs at this postion. One has to note that the usage of binary operators is in general very fast on the processor and thus this method of storing where a non zero value does occur or not does not have a significant overhead, but is necessary as it allows us to save memory at large systems and speed up value lookup inside the sparse matrix by eliminating zero lookups beforehand.

4.2.3 Sparse Matrix Value Lookup

If the binary eye yields that a non zero value is stored at an indexes position the value has to be fond in the arrays as fast as possible. In order to do this we order the matrix by the indices. This is done using the qsort() function which is part of the C standard library since the C89 standard, and shall sort our values at reasonable speed. For the detailed implementation the reader shall be refered to the programs source code. Once the matrix is sorted a bisection algorithm searches in the arrays of values for the correct index.

4.3 Adaptation of the static condensation algorithm

The static condensation algorithm has been adapted to our problem. In our code the static condensation algorithm is applied as the stiffness matrix is calculated for each element before it is added to the global stiffness matrix K. Edward L. Wilson [5] provides some fortran code with his algorithm, implementing this partial gauss elimination efficiently. In our case, as there are no forces acting where the virtual displacements do happen (they are not even attributed to a node), the system is even less complex as we do not have to take care of the right hand side of our equation system. Further Edward L. Wilson highlights an algorithm that inverts the upper left corner of the matrix acting onto the lower right portion of the matrix. As we have to add the elementary matrix to the global stiffness matrix afterwards this would be inefficient as memory offsets in order to locate the lower right portion would be more complicated to calculate then transfering values from the start of the arrays. Hence, we modified the algorithm to invert the lower right portion of the matrix and have it act on the upper left portion of the matrix, as shown in equation 3.14. Our implementation is listed in the following code snippet:

```
void condensate(double* k) {

  int i,j,key;
```

```
  double key_val;

  for(key=32;key>23;key--) {
    key_val = k[key*33+key];
    for(i=0;i<key;i++) {
      k[key*33+i] /= key_val;
      for(j=0;j<key;j++) {
        k[j*33+i] -= k[j*33+key]*k[key*33+i];
      }
    }
  }
}
```

Listing 4.6 – Static condensation implementation

where key_val is the value in the lower left diagonal of the row key currently used to modify coefficients in the upper left diagonal of the matrix. We also notice that each elementary stiffness matrix is reduced from 33 (24 nodal degrees of freedom and nine from the bubble functions) to 24 degrees of freedom.

In order to calculate strains and stresses accordingly from a modified strain-displacement relation we had to implement a static condensation algorithm that allows us to obtain the partial inverse of K denoted M (3.15). This results in a more generalized Gauss elimination procedure that is only used if stresses σ or strains ϵ are calculated. The current implementation is shown below and can probably be optimized as large parts of the inverse are not modified.

```
double* condensation_inverse(double* m) {

  int i,j,k;
  double key_value;
  double row_divisor;

  int range = 33;

  double* in = (double*)malloc(sizeof(double)*1089);

  memset(in,0,1089*sizeof(double));

  for(i=0;i<33;i++) {
    in[i*33+i] = 1.;
  }

  for(i=32;i>23;i--) {
    key_value = m[i*range+i];
    for (j=0;j<range;j++) {
      m[i*range+j] /= key_value;
      in[i*range+j] /= key_value;
    }
    for (j=0;j<i;j++) {
      row_divisor = m[j*range+i];
      for (k=0;k<range;k++) {
        m[j*range+k] -= row_divisor*m[i*range+k];
        in[j*range+k] -= row_divisor*in[i*range+k];
      }
```

```
    }
    for (j=i+1;j<range;j++) {
      row_divisor = m[j*range+i];
      for (k=0;k<range;k++) {
        m[j*range+k] -= row_divisor*m[i*range+k];
        in[j*range+k] -= row_divisor*in[i*range+k];
      }
    }
  }
  return(in);
}
```

Listing 4.7 – Static condensation with partial invers implementation

4.4 Conjungate Gradient Structure

The conjugate gradient needed to calculate the matrix vector product Kp_k as shown in equations (3.35) and (3.36). To calculate this matrix vector product efficiently a standard lookup for the values in the sparse matrix even with the algorithms discribed above would be to slow. Hence we transform the matrix into N = (rangeofthematrix) integer and flaoting point arrays for each row of the matrix. In each flaoting point array we then store the values of the row, while we store its index in the integer. This will come to our advantage computing the dot product as it can be evaluated very quickly looping accross both arrays, using the index array to gather the corresponding value of the vector the matrix is multiplied onto. We further create an array containing the number of values for each row (in order to know when the loop for each row should stop). The structure is most straightforward understood by looking at the code that implements it:

```
double** value = (double**)malloc(r_size*3*sizeof(double*)); // values in rows
int** index = (int**)malloc(r_size*3*sizeof(int*)); // row indicies
int* index_size = (int*)malloc(r_size*3*sizeof(int)); // n values in rows

double* current_values; // pointer to get offsets out of the loops
int* current_indicies;

double* force = create_reduced_force_vector(object);
double * reduced_displacements = (double*)malloc(sizeof(double)*r_size*3);

memset(reduced_displacements,0,sizeof(double)*r_size*3);

double * z = (double*)malloc(sizeof(double)*r_size*3);

// A = x b notation is used in the following

A = reduced_stiffness_matrix;
b = force;
x = reduced_displacements;

A.eye = create_sparse_eye(A);
A.has_eye = 1;
```

```
// To speed up computation create index accessors;

for(i=0;i<r_size*3;i++) {
  value[i] = (double*)malloc(r_size*3*sizeof(double));
  index[i] = (int*)malloc(r_size*3*sizeof(int));
  k=0;
  for(j=0;j<r_size*3;j++) {
    if(get_value_in_binary_index_at_index(A.eye,(long)i*(long)r_size*3+j)
       == 1) {
      value[i][k] = get_value_from_sorted_sparse_matrix(A,j,i);
      index[i][k] = j;
      k++;
    }
  }
  index_size[i] = k;
  value[i] = (double*)realloc(value[i],k*sizeof(double));
  index[i] = (int*)realloc(index[i],k*sizeof(int));
}
```

Listing 4.8 – Conjugate gradient acceleration structure

In order to speed up the conjugated gradient iterations almoast all instructions within the loop have been coded using intrisics instructions. Two versions, one using AVX2 instructions and one using SSE3 instructions, has been programmed. On an Intel Core i7 4771 processor it was found that the SSE3 based version proved to be the fasted one. This might be due to the slow gather instruction (load data into a 256 bit AVX register from arbitrary memory locations), that might improve in successive processor generations. A scalar reference implementation is nevertheless provided that and as such allows for portability to platforms/compiliers missing these instructions/intrinsics.

The default compilier flags in the make file should automatically select the right SIMD architecture apropriate for the processor the program is build on.

4.5 Eigenvalues and Principal Strains

As we wanted to have a library and operating system indipendent solution, we had to write a eigenvalue routine in order to obtain the principal stresses given for a specific element. The characteristic polynome yielding the eigenvalues of the strain tensor is computed using the well known inverse tangent method, that is for instance outlined in the following paper [7].

Chapter 5

Bibliography

[1] Wilson EL, Taylor RL, Doherty WP, Ghaboussi J. In: Incompatible Displacement Models. Academic Press; 1971. p. 43–57.

[2] Taylor RL, Beresford PJ, Wilson EL. A non-conforming element for stress analysis. International Journal for Numerical Methods in Engineering. 1976;10:1211–1219.

[3] Bathe KJ. Finite Element Procedures in Engineering Analysis. Prentice Hall; 1982.

[4] Troger H, Steindl A. Vorlesungsunterlagen Mechanik für Technische Physiker. Technische Universität Wien; 2004.

[5] Wilson EL. The static condensation algorithm. International Journal of Numerical Methods in Engineering, Short Communications. 1974;8(1):198–203.

[6] Bruaset AM. A survey of preconditioned iterative mathods. John Wiley and Sons Inc.; 1995.

[7] Hartmann S. Computational Aspects of the Symmetric Eigenvalue Problem of Second Order Tensors. Technische Mechanik. 2003;23(2-4):283–294.