Semester Thesis

Using PCG Solvers in Structural Optimization

Uche Mennel
umennel@student.ethz.ch

July 2, 2004

Federal Institute of Technology, Zurich
Centre of Structure Technologies

Ref Nr: 04-143

Contents

Chapter 1

Introduction

Within the subject of Structural Optimization, the computer solution of large linear systems are of increasing importance. Simulating the physical behavior of a complex structure usually requires the solution of partial differential equations, which cannot be solved analytically anymore. By discretization of the problem with the Finite Element method, it can be transformed into a linear equation system. With the growing complexity of the problem, the memory and computational requirements of solving such a linear system may seriously become a challenging issue.
Until recently, direct solution methods were often preferred to iterative methods because of their robustness and predictable behavior. Nowadays, the discovery of efficient iterative methods and the increasing need to solve very large linear system have triggered shift toward the iterative methods. It was found that the combination of preconditioning with gradient based methods emerged as "general purpose" procedures approaching the quality of direct solvers.

FELyX, the Finite Element Library eXperiment[11] has been developed at the Center of Structure Technologies of the Swiss Federal Institute of Technology in Zurich. The library is written entirely in C++ and combines the advantage of modern generic and object oriented programming with effective solver routines. Until now, FELyX endues a successful direct sparse skyline solver. Due to the reasons mentioned above, time has come to extend FELyX with an iterative solver and to gain some experience in this new area. Since the equation systems assembled in FELyX match the prerequisites of the well known Preconditioned Conjugate Gradient (PCG) method, it was the main objective of this semester thesis to implement such a method.

This work describes the first approach of integrating an iterative solver into FELyX. Starting with basic theory, it describes the magic behind the so called Krylov space methods and the preconditioning, with regard to memory and computational efficiency. The results and conclusions summarized in this report may well serve as an entry point for future developers who would like to extend the frontiers of numerical capacity of FELyX.

Chapter 2

Background

2.1 Positive definite matrices

The real matrix A is positive definite, if and only if $x^T Ax > 0$ for all $x \neq 0$. These matrices are sometimes referred to as positive real. Further, if A is symmetric positive definite (SPD), we can define the energy-norm

$$\|x\|_A^2 = x^T Ax$$

The following propositions characterize a SPD matrix

- A SPD matrix is non-singular,
- The eigenvalues of A (which are real numbers) are positive,
- There exists a unique real non-singular matrix L such that $A = LL^T$. Such a matrix L is called *Cholesky factor* of A.

2.2 Sparse matrices

The idea of taking advantage of the sparsity was discovered in the early 1960s and revolutionized solving methods. Sparse matrices allow us to store its data in a very economical way; only a few, usually non-zero elements need to be stored, without the loss of any information. A $n \times m$ matrix A with k non-zero entries is *sparse* if $k << nm$. Consider a set S of matrix positions (i, j). If S coincides with the set of positions (i, j) of matrix A, where $a_{i,j} \neq 0$, the set S is also referred to as the *sparsity structure* or *non-zero pattern* of A.

2.3 Sparse matrix data structures

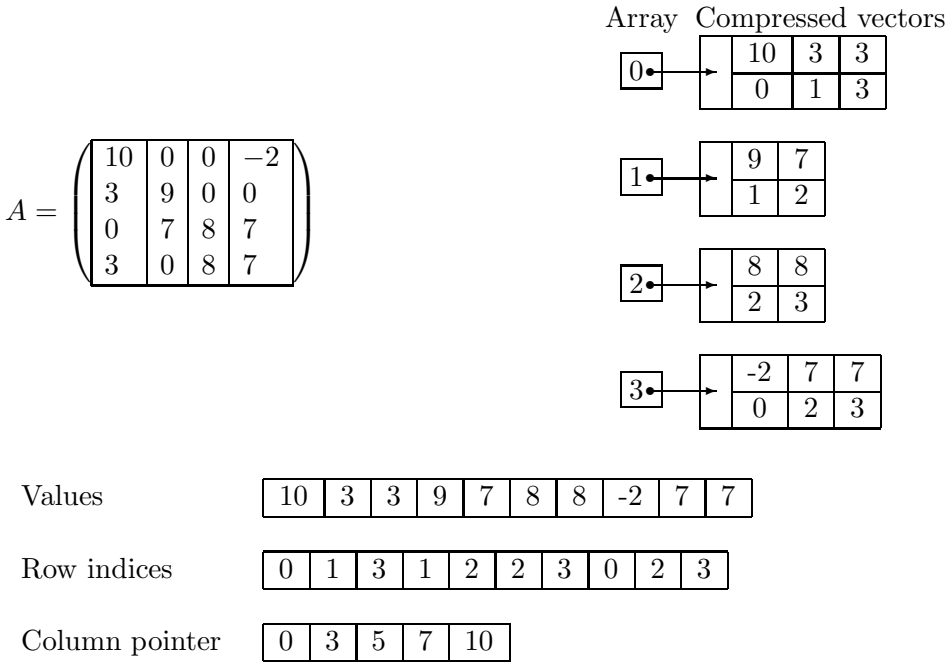
The difficulty of representing sparse matrices is to provide computationally efficient ways of handling the matrices, while omitting unnecessary non-zero elements. The matrix template library (MTL)[8] offers a collection of

generic sparse and dense data structures and algorithms. Although there are more data structures available within the MTL library, only the Compressed Row/Column Storage schemes are documented in this chapter, since they emerged as best choice in combination with iterative solvers.

2.3.1 Compressed Row/Column Storage (CRS)/(CCS)

The Compressed Row and Column Storage formats are the most general; they make absolutely no assumptions about the sparsity structure of the matrix, and they do not store any unnecessary elements. The CCS format is sometimes referred to as the *Harwell-Boeing* sparse matrix format. In the MTL, two different implementations of these formats exist. The first consists of an array of what is defined in the MTL as *compressed1D* vectors. A *compressed1D* vector consists of two arrays of the same length, one containing the values and the other containing the corresponding column/row indices. The second implementation is more popular and slightly differs

Figure 2.1: The Compressed Column Storage



from the previous one. It uses three arrays; one to store the real values, one to store the column/row indices and the third contains pointers to the beginning of each row/column. This structure is called *compressed2D* in the MTL. Both implementations of the CCS format are illustrated in figure 2.1. The CRS format is equivalent to the CCS format, except that the roles of

columns and rows are exchanged. Note that it is a requirement for both formats that the indices are always sorted in ascending order.

2.3.2 Matrix traits

The most important characteristic traits of a matrix A are

- The shape of A,
- The orientation of A,
- The view of A.

The shape is either symmetric, rectangular, banded, triangular or diagonal. The orientation of the matrix describes its access method. A row oriented matrix is usually accessed row-wise while a column oriented matrix is accessed column-wise. For each Matrix, there are two views, namely the lower view and the upper view. This means nothing else than considering the matrix as a lower or upper triangular matrix. These matrix traits are closely related to the data structures in which they are stored. E.g. a matrix represented in the CRS format is row-oriented, a matrix represented in CCS format is obviously column oriented. If a matrix is symmetric, only the lower or the upper view need to be stored, since every off-diagonal element appears twice. Hence, a symmetric matrix can be accessed in four different variants, obtained by combining the two views and two orientations. Figure 2.2 displays the mapping of a symmetric matrix into the CCS format.

Figure 2.2: Storing a symmetric matrix in the Compressed Column Storage

$$A = \begin{pmatrix} 10 & 3 & 0 & -2 \\ 3 & 0 & 7 & 0 \\ 0 & 7 & 8 & 9 \\ -2 & 0 & 9 & 7 \end{pmatrix}$$

Values	10	3	-2	7	8	9	7
Row indices	0	1	3	2	2	3	3
Column pointer	0	3	4	6	7		

Chapter 3

The Preconditioned Conjugate Gradient Method

The conjugate gradient method is the oldest and best known Krylov subspace method. It is effective for solving symmetric positive definite (SPD) systems.

3.1 Theory

For a SPD system, minimizing the parabolic function

$$\Phi(x) = \frac{1}{2}x^T Ax - x^T b$$

is equivalent to solving the system

$$\nabla\Phi(x) = Ax - b = 0$$

The method proceeds by generating vector sequences of iterates, residuals of iterates and search directions. Starting with an initial guess $x^{(0)}$, the iterates $x^{(i)}$ are updated in each iteration by a multiple α_i of the search direction vector $p^{(i)}$ (also called *gradient*);

$$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)} \tag{3.1}$$

where α_i is chosen to minimize $\Phi(x^{(i)})$, i.e. to find a local minimum in direction of $p^{(i)}$. The residual $r^{(i)}$ is defined as

$$r^{(i)} = b - Ax^{(i)} \tag{3.2}$$

and is used as a common measure of error.
Obviously, the ith iterate $x^{(i)}$ is always an element of the space

$$x^{(0)} + \text{span}\{r^{(0)}, \ldots, A^{i-1}r^{(0)}\}$$

which is called the Krylov subspace. If the gradients $p^{(i)}$ are chosen to be $A-conjugate$ or $A-orthogonal$, that is

$$p^{(i)^T} A p^{(j)} = 0 \text{ for } i \neq j$$

then the $p^{(i)}$ form an orthogonal basis of this subspace. These are the conjugate gradients that give the method its name. Hence, the minimum can be expressed as

$$x = \sum_{i=0}^{n} \alpha_i p^{(i)} \tag{3.3}$$

where n equals the problem size, i.e. the number of degrees of freedom (DOF). In algorithm 3.2, the conjugate gradients are constructed by two coupled two-term recurrences, derived from the three-term *Lanzcos* recurrence; see [6]. The first recursive step updates the residuals using a search vector, and the second one updates the search direction with a newly computed residual. Updating the residuals can be derived from definition 3.2;

$$r^{(i)} - r^{(i-1)} = b - Ax^{(i)} - (b - Ax^{(i-1)}) = -A(x^{(i)} - x^{(i-1)})$$

and with the iteration 3.1 follows that

$$r^{(i)} = r^{(i-1)} - \alpha_i A p^{(i)} \tag{3.4}$$

To preserve orthogonality of the search directions, the residuals also have to fulfill this property. Hence, it is necessary that

$$r^{(i)^T} r^{(i-1)} = (r^{(i-1)} - A\alpha_i p^{(i)})^T r^{(i-1)} = 0$$

and as a result, we obtain

$$\alpha = \frac{r^{(i-1)^T} r^{(i-1)}}{Ap^{(i)})^T r^{(i-1)}} \tag{3.5}$$

Now, we can update the search directions as linear combination of the old direction and residual;

$$p^{(i)} = r^{(i-1)} + \beta_{i-1} p^{(i-1)}$$

Thus, a consequence of the above relation is that

$$(Ap^{(i)})^T r^{(i-1)} = (Ap^{(i)})^T (p^{(i)} - \beta_{i-1} p^{(i-1)}) = (Ap^{(i)})^T p^{(i)}$$

because $(Ap^{(i)})^T p^{(i-1)} = 0$, i.e. $p^{(i)}$ and $p^{(i-1)}$ are conjugate. Then, 3.5 becomes

$$\alpha = \frac{r^{(i-1)^T} r^{(i-1)}}{Ap^{(i)})^T p^{(i)}}$$

and β_{i-1} evaluates to

$$\beta_{i-1} = \frac{p^{(i)} - r^{(i-1)}}{p^{(i-1)}} = \frac{(Ap^{(i-1)})^T(p^{(i)} - r^{(i-1)})}{(Ap^{(i-1)})^T p^{(i-1)}} = -\frac{(Ap^{(i-1)})^T r^{(i-1)}}{(Ap^{(i-1)})^T p^{(i-1)}} \quad (3.6)$$

Note that from 3.4 follows that

$$Ap^{(i)} = \frac{1}{\alpha} r^{(i)} - r^{(i-1)}$$

inserted into 3.6 finally yields

$$\beta_i = -\frac{(Ap^{(i)})^T r^{(i)}}{(Ap^{(i)})^T r^{(i-1)}} = \frac{(r^{(i)} - r^{(i-1)})^T r^{(i)}}{(r^{(i)} - r^{(i-1)})^T r^{(i-1)}} = \frac{{r^{(i)}}^T r^{(i)}}{{r^{(i-1)}}^T r^{(i-1)}}$$

This recursive procedure makes the Conjugate Gradient Method quite attractive computationally.

Input: A linear system $Ax = b$ of size n
Output: Solution x with corresponding residual r

```
1  for i=1,2,...,n do
2      ρ_{i-1} := r^T r
3      if i=1 then
4          p := z
5      else
6          β := ρ_{i-1} / ρ_{i-2}
7          p := z + βp
8      end
9      q := Ap
10     α := ρ_{i-1} / p^T q
11     x := x + αp
12     r := r − αq
13     check convergence; continue if necessary
14 end
```

Algorithm 3.1: The Conjugate Gradient Method (adapted from [7])

As described in 3.2, the CG algorithm converges after at least n iteration steps in exact arithmetic, where n is the number of DOFs. Using a computer which only has a finite arithmetic at its disposal may lead to even more iteration steps, or even worse, to divergence. Therefore, if solving large systems with a computer, the only way to provide the solution in reasonable time is to reduce the iteration steps as much as possible, i.e. to improve the convergence rate of the iteration phase by preconditioning. To formalize this idea, suppose that we want to solve a linear system $Ax = b$. By applying a

preconditioner M, realized by a non-singular matrix, to our system, we can transform[1] it to

$$M^{-1}Ax = M^{-1}b. \tag{3.7}$$

Then we can use the iterative method to solve the system 3.7, which has the same solution, but favorable properties leading to a faster convergence. Such a property might be a better condition number $\kappa(M^{-1}A)$, since it is well known that the condition of a problem has a principal influence on the convergence rate.
Hence, the preconditioned version of algorithm 3.1 uses a different subspace for constructing the iterates, but it satisfies the same minimization property. In addition, it requires that the preconditioner M is an SPD matrix. Algorithm 3.2 displays the Preconditionend Conjugate Gradient (PCG) method. It is easy to see, that the unpreconditioned algorithm 3.1 can be obtained from algorithm 3.2 by just substituting M with an identity matrix.

Input: A linear system $Ax = b$ of size n
Output: Solution x with corresponding residual r

for *i=1,2,...,n* **do**
  solve $Mz := r$
  $\rho_{i-1} := r^T z$
  if *i=1* **then**
    $p := z$
  else
    $\beta := \frac{\rho_{i-1}}{\rho_{i-2}}$
    $p := z + \beta p$
  end
  $q := Ap$
  $\alpha := \frac{\rho_{i-1}}{p^T q}$
  $x := x + \alpha p$
  $r := r - \alpha q$
  check convergence; continue if necessary
end

Algorithm 3.2: The Preconditioned Conjugate Gradient Method (adapted from [7])

3.2 Convergence

As follows from expression 3.3, the CG algorithm converges after at most n iterations. However, if solving large systems, it is important that a certain

[1] This is usually called left preconditioning, as we multiply matrix A from the left by M^{-1}. There also exist right and symmetric preconditioning, which are described in [5].

convergence criteria is reached after less than n iterations to provide a solution in reasonable time. Such a convergence criteria is usually a relative error

$$\epsilon \geq \frac{\|r^{(i)}\|_2}{\|b\|_2}$$

where the two-norm $\|x\|_2 \equiv \sqrt{x^T x}$ is used. Sometimes an absolute error

$$\epsilon \geq \|r\|_2$$

is desirable. With the energy-norm defined as $\|x\|_A^2 = x^T A x$ and κ denoted as the spectral condition number of the preconditioned system $M^{-1}A$ (The spectral condition number of a matrix is the ratio of the largest to the smallest eigenvalue of that matrix), the following statement gives some error bound of the CG algorithm

$$\|x^{(i)} - \hat{x}\|_A \leq 2\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\|x^{(0)} - \hat{x}\|_A$$

where $\hat{x}$ is the exact solution of the linear system $Ax = b$. The proof of this theorem can be consulted in [6]. From this relation we see that the number of iterations to reach a certain relative reduction in the error is proportional to $\sqrt{\kappa}$.

3.3 Implementation details

The Conjugate Gradient iteration described in algorithm 3.2 involves

- One matrix-vector product (line 10),
- Three vector updates (*axpy*[2] operations, lines 8,12 and 13),
- Two inner products (dot products, lines 3 and 11) and
- A preconditioner solve (line 2).

The method has modest memory requirements: besides the overhead to store the matrix and a few scalars, we only need four storage vectors. This allows us to keep these vectors dense and handle vector operations by a tuned dense linear algebra package like ATLAS[10]. A generic preconditioned iteration concept is included in the iterative template library (ITL)[9]. However, for efficiency reasons, it is recommended to restrict the iteration to dense vectors, whenever possible.

[2] *axpy* is an abbreviation for $\alpha x + y$ where α is a scalar and x and y are vectors.

Chapter 4

Preconditioning Techniques

Thus, when building a PCG solver, expending a lot of work to enhance the quality of the preconditioner is one of the most crucial factors. But finding a good preconditioner is often viewed as a combination of art and science. Theoretical results are rare and it may happen, that performance of a certain method differs from one's expectations. This chapter describes the most successful techniques, all of them having their strengths and weaknesses.

4.1 Incomplete factorization preconditioners

A very broad and popular class of preconditioners is based on incomplete factorizations. Such a preconditioner is given in the form $M = LU$ with L lower and U upper triangular matrix. Sometimes the form $M = LDU$ with D a diagonal matrix, L lower and U upper triangular, each containing a unit diagonal, is preferable. A factorization is called incomplete, if, during the factorization process, elements matching certain criteria are dropped, i.e. set to zero. Such a factorization can be obtained by a Gaussian elimination and is unique as proved in many linear algebra books, such as [5]. Denoted S as a set of matrix positions (i, j) and $l_{i,j}$ as the element of the factor L at position (i, j), it holds that $l_{ij} = 0$ if $(i, j) \notin S$. The same holds, of course, for the upper factor U. S can be determined by structural, e.g. a nonzero pattern, or numerical, e.g. a threshold value, criteria. A position that is zero in A but not in the resulting factors of the exact factorization is termed *fill*. So if S is equal to the nonzero pattern of A, we talk about a *zero-fill* factorization, because all fill is discarded. The efficacy of the preconditioner depends on how well M^{-1} approximates A. Allowing more fill leads in most cases to better approximation, but on the other side, more memory is needed to store the factors and at the same time the computational costs are rising.

Incomplete factorizations may brake down[1] or result in indefinite ma-

[1] Attempted division by zero pivots

trices[2] even if the full factorization is guaranteed to exist and yield a positive definite matrix. The existence of an incomplete factorization has been proved for so called *M-matrices*, *H-matrices* or *diagonally dominant* matrices, which fulfill stronger conditions than an arbitrary SPD matrix. This was originally done for *M-matrices* by Meijerink and van der Vorst [3] and extended to *H-matrices* in reference [2]. However, the structure of matrices arising from finite element problems reflects the irregularity and complexity of this domain. These matrices are not *H-matrices*, nor are they diagonally dominant. Many strategies have been proposed to avoid zero pivots or indefinite matrices such as pointed to in 4.7.2 or 4.7.1.

4.2 Incomplete Cholesky decomposition

From now on we assume the input matrix A to be an arbitrary SPD matrix, as this is the case in FELyX; see [11]. If we transform such an SPD matrix by Gaussian elimination, we obtain the Cholesky decomposition $M = LL^T$ or, if preferred LDL^T. For symmetry reasons, the upper factor U equals the transposed lower factor L^T. Its implementation requires a combination of three nested loops, indexed with $\{i, k, j\}$. Depending on the orientation and view of A, the computer architecture and fill-in strategy, the decomposition can be computed by different algorithms. With focus on the lower view of A, according to [5], there are three basic types of algorithms for Cholesky factorization, depending on which of the three indices is placed in the outer loop:

1. Row-based: Taking i in the outer loop, successive rows of L are computed one by one, with the inner loops solving a triangular system for each new row in terms of the previously computed rows. This type is also denoted as bordering ijk algorithm in [5]. It is rarely used for a number of reasons, including the difficulty in providing a (sparse) row-oriented[3] data structure that can be accessed efficiently during the factorization, and the difficulty in vectorizing or parallelizing the triangular solutions required.

2. Column-based: Taking j in the outer loop, successive columns of L are computed one by one. This type is also denoted as inner-product jki algorithm in [5]. Although it requires row-wise accessing of elements in the k loop, it can be implemented quite efficient by using a suitable data structure to avoid index searching along the (sparse) rows. The main benefit of this algorithm is the feature that once a column is

[2]Negative pivots. This is especially a problem during a Cholesky factorization which contains square roots. The factorization process must then be aborted.

[3]In many finite element applications, e.g. FELyX, the matrix A is assembled in a column-oriented structure (the lower view of A).

computed, it will never be modified again, permitting an inclusion of numerical aspects during the factorization process. Threshold strategies (4.5) form a prime example of taking advantage of this fact.

3. Sub-matrix: Taking k in the outer loop, successive columns of L are computed one by one, while modifying the remaining sub-matrix. This type is also denoted as outer-product kij algorithm in [5]. Row-wise access of elements occurs only in the most inner loop, resulting into a simple and efficient algorithm. Therefore, it can be successfully applied to a zero-fill 4.3 factorization.

4.3 Zero-fill IC(0) factorization

If the set S of nonzero matrix positions in the factor L equals the nonzero pattern of A, all fill is discarded. Thus, we get the zero-fill or IC(0) factorization. Because of the simplicity of this method, an outer-product Cholesky factorization satisfies its conditions best.

4.3.1 Discussion

Solving small to moderate problems and having considered the possibility of negative pivot elements, this method exhibits a quite good performance. When solving larger problems, the poor accuracy of this preconditioner causes too many iterations. Since it is best applicable to the problem domain, where usually direct solvers are preferable, its most advantageous feature is its quite precisely predictable and low memory usage. Indeed, this preconditioner takes at worst exactly as much space as the original matrix.

4.4 Level k-fill IC(k) factorization

Another quite simple factorization technique is allowing fill-in until to a certain level. Each off-diagonal, non-zero position (i, j) in matrix A is assigned a fill level value. In [7], this value (say in step k) is proposed as

$$lev_{i,j} = 1 + max\{lev_{i,k}, lev_{k,i}\}$$

starting with $lev_{i,j} = 0$ while in [6] this value is initially set as $lev_{i,j} = \infty$ and updated with

$$lev_{i,j} = 1 + min\{lev_{i,j}, lev_{i,k} + lev_{k,j} + 1\}$$

Note that IC(0), pointed to above, is a special case of this factorization, where $k = 0$. This method allows a very well controllable memory consumption, but its blindness to numerical aspects may not always give an improvement.

4.5 Threshold strategies and ICT

A generic IC algorithm with threshold can be derived from the inner-product version of the Cholesky decomposition, by including a set of dropping rules. Applying a dropping rule to an element in a column means discarding an element, if it satisfies certain numerical criteria. Although this approach is mathematically sensible, it is harder to implement in practice, since the amount of storage needed for the factorization is not easy to predict. Saad [6] describes a very effective strategy to reduce the uncertainty of memory consumption:

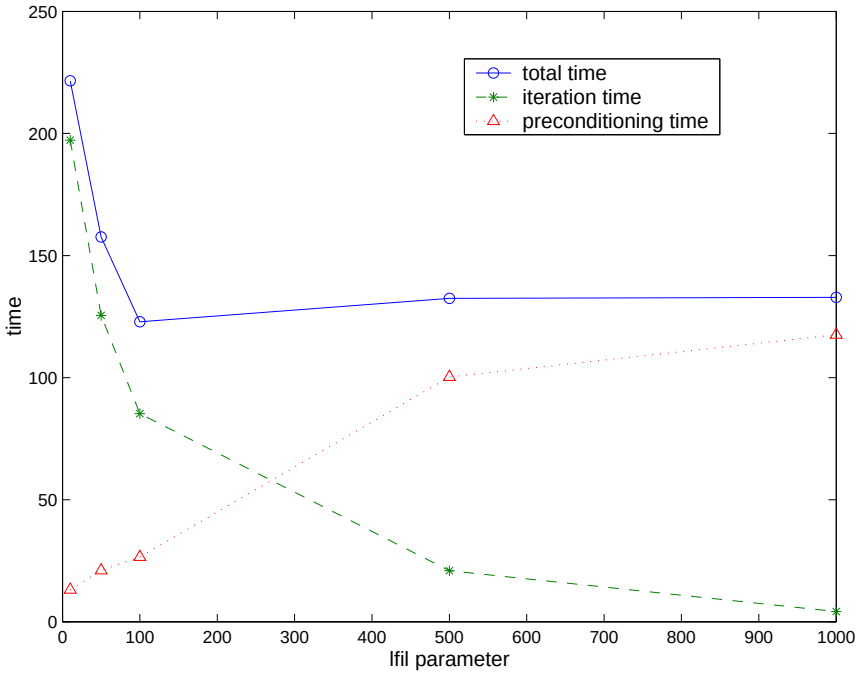
1. In line 12 of algorithm 4.2, an element w_k is dropped (i.e. set to zero), if it is less than then the tolerance τ_i, obtained by multiplying the parameter τ with the norm of the ith column (e.g. the one-norm).

2. A dropping rule of a different kind is applied, namely keeping only the p largest elements of the column in addition to the diagonal element, which is only kept. Line 19 of algorithm 4.2 shows a little modification of this rule, by keeping the largest $nnz(i) + p$ elements, where $nnz(i)$ denotes the number of non-zeros in the i-th column. With this modification, we make sure that our preconditioner has at least the quality of IC(0). Because of its role of determining the sparsity of the proconditioner, p is often termed *level of fill* or *lfil* parameter.

The goal of the second dropping step is to reduce the elements per column. Roughly speaking, p can be viewed as a parameter that helps controlling memory usage, while τ helps to reduce computational cost. Obviously, the upper memory usage bound can be derived with parameter p. In the case where we keep only the largest p elements per column, the upper bound is $O(n * p + n)$ and with modification mentioned above it changes to $O(n * p + nnz(A))$. Hence, if we know the upper memory bound, we can abandon some complex dynamic memory allocation strategy.

Figure 4.1 illustrates how the the parameters p and τ influence the whole solving process. A too sparse preconditioner leads to poor convergence, while the creation of a too dense preconditioner may take too much time (and space). On the other hand, depending on the application, one can take advantage of this flexibility; A very sparse preconditioner may be used in memory critical situations, while a quite dense preconditioner may be a benefit, if solving many similar problems by reusing the same preconditioner. Latter scenario may arise e.g. within evolutionary optimization procedures.

Figure 4.1: Solving time in terms of the ICT preconditioner size (while solving a mesh of 11798 elements)



4.6 Implementation details

4.6.1 The factorization algorithms

Algorithm 4.1 shows the outer-product IC(0) factorization $A \approx LDL^T$.

Since the amount of memory needed by the execution of algorithm 4.1 is exactly known, allocation of memory reduces to a simple matrix copy operation. Depending on the input matrix A, negative pivot elements may occur, but for efficiency reasons, they are ignored. If that is the case, convergence of CG algorithm is not guaranteed anymore. Nevertheless, in many of such cases the algorithm converges at acceptable rates.

On the way to an efficient implementation of a dual threshold preconditioner, the following obstacles have to be overcome:

1. Since we need the inner-product algorithm for this purpose, index searching has to be optimized by dynamically updating a suitable data structure.
   This is solved by using two (dense) vectors, one containing the next row index of the kth row, the other stores a pointer to the according matrix position.
   After processing the jth column, finding the $(j+1)$th row elements

Input: SPD matrix A with nonzero pattern set S
Output: A lower triangular matrix L with unit diagonal (which is not saved) and a diagonal matrix D

```
1  Copy lower part of A into L
2  for k = 1...n do
3      d_k := 1/(a_{k,k} - d_k)
4      for i = k+1...n do
5          temp_i := l_{i,k}
6          l_{i,k} := l_{i,k} d_k
7      end
8      for i = k+1...n do
9          d_i := d_i + l_{i,k} temp_i
10         for j = i+1...n do
11             l_(j,i) := l_(j,i) - l_(j,k) temp_i, {(j,i),(j,k)} ∈ S
12         end
13     end
14 end
```

Algorithm 4.1: Incomplete Cholesky IC(0) with zero-fill

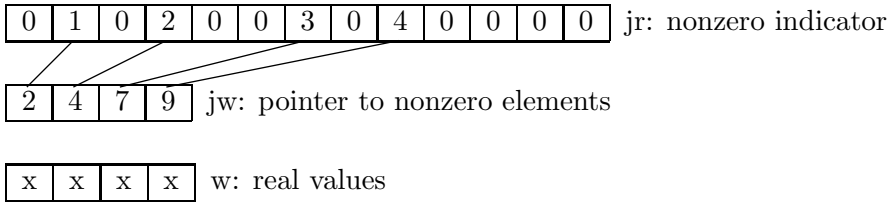
in the first j columns is reduced to comparison of the first j entries of the row index vector with index $j+1$. If an entry matches, we have immediately found the element in the matrix by the associated pointer, and the data structure can be already updated for the next loop. Otherwise we know that such an index doesn't exist in that column (i.e. there is a zero) and can proceed to the next column.
Line 8 of algorithm 4.2 involves the use of this procedure.

2. A data structure has to be introduced, which allows sparse vectors to be sorted by values and indices at the same time and nevertheless, random access at constant time must be sill possible. Such a data structures is illustrated in [6] consisting of the three vectors shown in figure 4.2. A dense vector $jr(1:n)$ is used, the values of which are zero except when there is a non-zero at a certain index. With this vector, a dense real vector $w(1:wmax)$ has to be maintained which contains the real vector of the column, as well as a corresponding dense integer array $jw(1:wmax)$, which points to the column positions of the real values in the column.

3. A fast splitting technique must be available, which extracts the largest p values out of a vector of length m. A *heap* data structure would do it in $O(m + plog(m))$ time, i.e. $O(m)$ for the *heap* construction and $O(log(m))$ for each extraction, whereas the *quicksplit* algorithm

with $O(m)$ time effort in the average case give a slightly enhanced performance. This algorithm consists of choosing an element, e.g. $x := w_1$, in the array $w(1 : wmax)$, then permuting the data so that $|w_k| \leq |x|$ if $k < mid$ and $|w_k| \geq mid$ if $k \geq mid$, where mid is some split point. If $mid = p$, then exit. Otherwise, split one of the left or right sub-arrays recursively[4], depending on whether mid is smaller or larger than p. The *quicksplit* is invoked in line 18 of algorithm 4.2

Figure 4.2: Data structure used for the working row in ICT



Note that the LDL^T variant is used in both algorithms 4.1 and 4.2 to avoid any square root therms. Since the output lower triangular L has a unit diagonal in that case, there is no need to save this diagonal elements. Therefore, L is stored as a $(n-1) \times (n-1)$ matrix. In addition to that, the diagonal matrix D is represented by storing its reciprocal values. The reason for this particular storage scheme is on the one hand to keep the memory usage as low as possible, and on the other hand, to simplify and accelerate the preconditioner solve step during the cg iteration described in the next subsection.

4.6.2 Preconditioner solve

If using incomplete factorization preconditioners, the most time consuming operation is the preconditioner solve in line 2 of algorithm 3.2, because it consists of one front and one back substitution procedure. Formally, if the preconditioner M is factorized as LDL^T, by solving the following triangular and diagonal systems

$$\begin{aligned} Mz = LDL^T z &= r \\ Ly &= r \\ Dx &= y \\ L^T z &= x \end{aligned}$$

[4]Despite the recursion, this method is well suited for an iterative implementation

Input: SPD matrix A with nonzero pattern set S
Output: A lower triangular matrix L with unit diagonal (which is not saved) and a diagonal matrix D

```
1  for j = 1, ..., n do
2      d_j := a_{j,j} − d_j
3      for i = j + 1, ..., n do
4          w_i := a_{j,i}
5      end
6      calculate columnnorm
7      for k = 1, ..., j − 1 do
8          if (j, k) ∈ S then
9              temp := l_{j,k} d_k
10             for i = j + 1, ..., n do
11                 w_i = w_i − l_{i,k} × temp, (i, k) ∈ S
12                 if w_i < τ × columnnorm then
13                     discard w_i
14                 end
15             end
16         end
17     end
18     extract largest nnz(j) + p elements out of w
19     sort indices of w
20     forall i ∈ indices of w do
21         l_{i,j} := w_i / d_j
22         d_i := d_i + l_{i,j} w_i
23     end
24     set w to zero
25     end
26 end
27 update d with the reciprocal values of d
```

Algorithm 4.2: Incomplete Cholesky ICT(τ, p) with dual threshold strategy

we get

$$
\begin{aligned}
y &= L^{-1}r \\
x &= D^{-1}y \\
z &= L^{T^{-1}}x \\
\text{Hence, } z &= L^{T^{-1}}D^{-1}L^{-1}r = M^{-1}r
\end{aligned}
$$

Recall the matrix traits in section 2.3.2. If inverting a triangular matrix L, we have to distinguish four different access schemes, namely row oriented upper view, row oriented lower view, column oriented upper view and column oriented lower view. Algorithms 4.3 and 4.4 show two cases of solving a unit diagonal triangular system. The remaining two cases can be easily derived from the previous ones. Note that line 3 in algorithm 4.3 describes a sparse axpy (vector update) operation, whereas line 3 of algorithm 4.4 describes a sparse dot product. Handling these common sparse vector operations with a fast sparse library, e.g. the MTL, may give this procedures a significant boost.

Input: A lower, column oriented unit diagonal triangular system $Ly = r$. Note that the unit diagonal is not stored. L_i refers to the ith column of L.
Output: Solution $y = L^{-1}r$

```
1 y := r
2 for i = 1, ..., n − 1 do
3     y = y − y_i L_i
4 end
```

Algorithm 4.3: Column oriented lower unit diagonal triangular solve

Input: An upper, row oriented unit diagonal triangular system $L^T z = x$. Note that the unit diagonal is not stored. L_i refers to the ith column of L.
Output: Solution $z = L^{T^{-1}}x$

```
1 z := x
2 for i = n − 1, ..., 1 do
3     z = z − z L̇_i
4 end
```

Algorithm 4.4: Row oriented upper unit diagonal triangular solve

4.7 Stabilizing methods

4.7.1 Shifted IC

Due to the irregularity of finite element matrices, incomplete factorizations of such matrices may yield indefinite preconditioners. In that case, the convergence of the PCG iteration process is no longer guaranteed. Especially during a batch process, it is indispensable that every single step terminates to provide a successful run. One method to stabilize the incomplete factorization is the *shifted* incomplete Cholesky factorization by Manteuffel [2]. He proposed to force the positivity of the preconditioner by adding a *shift value* α to the diagonal of matrix A:

$$A = (1 + \alpha)D + B$$

where the D contains the diagonal elements and B contains the off-diagonal elements of A. Obviously, a very large difference of the shifted matrix from the original matrix degrades the accuracy of the preconditioner and leads to poor convergence rates. However, there is always an optimal α resulting in an optimal convergence. But the catch is, that there is no direct way to find such an optimal value of α. Selecting α so that A becomes diagonally dominant is a bad choice, since the optimal value of α is often much smaller than the value that makes A diagonally dominant. Finding α by trial and error may enormously extend the preconditioning time, because every try consists at least of the whole incomplete factorization process. For all that, once an optimal value α for a problem has been found, solving many similar problems with improved convergence by reusing the same preconditioner may amortize the extended preconditioning costs.

4.7.2 Robust IC

Instead of simply discarding elements dropped during factorization, some techniques attempt to compensate for the discarded elements. One possibility is to add up all the elements in a column and subtract the sum from the diagonal. This diagonal compensation gives rise to modified IC factorizations. A description of these methods can be found in [6] or [7]. However, according to [7], this kind of modified factorizations may brake down and give not the desired stability.

What we are interested in is some technique which compensates the loss of positive definiteness while discarding an element. A factorization using this technique always remains stable and never encounters negative pivot elements. It was also the motivation of Azij and Jennings to develop such a technique; see reference [1]. Whenever an element $l_{i,j}$ of the incomplete

Cholesky factor L is dropped, $|l_{i,j}|$ is added to the corresponding diagonal entries $a_{i,i}$ and $a_{j,j}$ of A. In this method, the incomplete Cholesky factor L is the exact Cholesky factor of a perturbed matrix $A \hat{=} A + C$ where C, *the cancellation matrix*, is positive semidefinite. Hence, no breakdown can occur.

4.8 Approximate inverse preconditioners

All the incomplete factorizations surveyed in the previous sections were based on the well known Cholesky algorithm or, possibly, on its root-free variant (LDL^T factorization). However, this is not the only way to compute a triangular factorization of A. These methods attempt to compute the approximate inverse $M^{-1} \approx A$. If we are able to do this, applying the preconditioner (i.e. executing line 2 in algorithm 3.2) simply amounts to a matrix multiply.

4.8.1 Stabilized AINV

A factorized sparse approximate inverse preconditioner is the AINV method. This preconditioner arises from the remark that if $Z = [z_1, z_2, \ldots, z_n]$ is a set of conjugate directions for A, we have $Z^T AZ = D$ a diagonal matrix with diagonal elements $d_i = (z_i, Az_i)$. A set of conjugate directions can be constructed by a Gram-Schmidt orthogonalization like algorithm. Sparsity is maintained by applying a dropping rule during the construction of the preconditioner, e.g. in line 7 of algorithm 4.5. Since a sufficient condition for AINV to be breakdown-free is that A be an H-matrix, it is not applicable to general SPD matrices. The price that is paid for adding robustness is a slightly higher cost of computing and results into the stabilized AINV, or SAINV, preconditioner [4].

Compared with the incomplete factorization preconditioners, the computational cost of this method seems to be much higher, while memory usage of both methods are quite the same. Another quite difficult implementation subject is the prediction of memory usage. Maybe a dual threshold strategy as introduced in the incomplete factorization sections could solve this problem.

4.9 Effects of Reordering

The non-zero structure of a sparse matrix A has a significant influence on the appropriate solving method. There are many reordering strategies to modify the structure to satisfy different (usually structural) criteria. Information about the most popular ones can be found in [5].
A bandwidth reducing reordering, e.g. Sloan's algorithm, is certainly a asset,

Input: SPD matrix A
Output: A lower triangular matrix Z and a diagonal matrix D

```
1  z_i := e_i (1 ≤ i ≤ n)
2  for i=1,...,n do
3      v_i := A z_i
4      for j=i,..., n do
5          p_j := v_i^T z_j
6      end
7      for j=i+1,...,n do
8          z_j := z_j − (p_j / p_i) z_i
9      end
10 end
11 Z = [z_1, ..., z_n] and D = diag(p_1, ..., p_n)
```

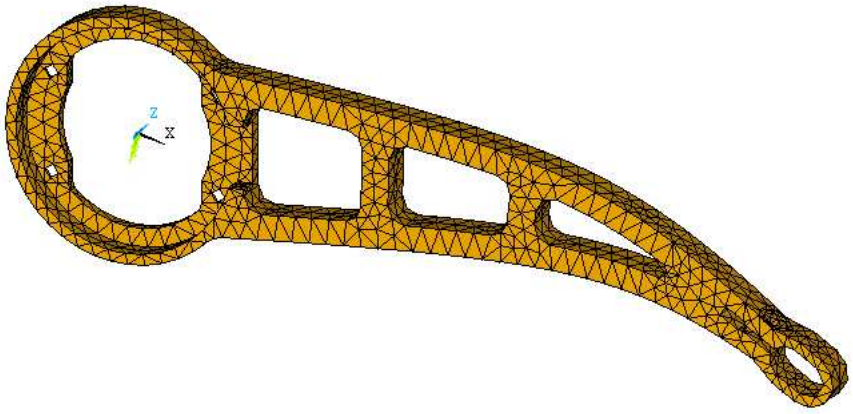
Algorithm 4.5: The SAINV algorithm

if using a direct skyline solver, while a fill reducing reordering, e.g. a Minimum Degree Ordering, might improve an incomplete Cholesky factorization preconditioner. Fill reduction is reached by adapting the non-zero structure of the original matrix to the structure of the Cholesky factor. However, the choice of the most suitable reordering algorithm for a certain method is not always that obvious. Especially in combination with methods, which operate on non-structural driven criteria, such as threshold strategies, a reordering algorithm may yield surprising results. The ICT preconditioner forms an example for this, since it performs best by preparation with Sloan's algorithm. How can a reordering, whose principal purpose is to reduce the bandwidth of matrix, be the best choice for a preconditioner based on Cholesky factorization? A possible explanation could be that a reduction of the bandwidth obviously requires an approximately uniform distribution of the number of non-zeros over the columns of matrix. Since the number of non-zeros per column is limited by the application of a dual threshold strategy, the resulting Cholesky factor may show a similar distribution of the matrix elements. Hence, the fill is minimized, giving a maximum accuracy of the preconditioner.

Chapter 5

Numerical Experiments

Figure 5.1: The model cantilever arm



In order to investigate the efficiency of the PCG implementation, a series of test examples have been performed on a model cantilever arm; see figure 5.1. The corresponding mesh has been refined stepwise from 2896 up to 74476 elements to allow observation of the scaling behavior of the PCG solver. Table 5.1 lists the test matrices arose from the cantilever arm model. All tests were done on a IBM Bladecenter HS 20 with two 2.8GHz Intel Xeon CPUs and 2GB RAM. All test examples were calculated by using double precision arithmetic, until the relative error fell below a tolerance of 10^{-8}.

Tables 5.2, 5.3 and 5.4 display the results of various solvers. IC(0) denotes a zero-fill IC factorization preconditioned CG, while ICT with chosen fill and threshold parameter denotes a stabilized IC dual threshold PCG solver. The stabilizing method described in section 4.7.2 yields quite satisfying results, while the method described in section 4.7.1 emerged as very unpractical. The comparison also includes the FELyX direct skyline solver and the sophisti-

cated ANSYS[1] PCG solver. The abbreviations in the columns of these tables are defined as [solver type, memory usage in floating point units (i.e. double precision units), preconditioning time, iteration time, total time, number of iterations]. The results show that the ICT preconditioner is significantly more effective than IC(0), while on the other hand, the ANSYS PCG shows a superior scaling. Figure 5.2 also clarifies, that the overhead of stabilizing the ICT preconditioner does not influence the performance, because the more robust convergence of this method compensates this slight overhead. Furthermore, these results show the rapidly growing memory consumption of direct solvers. In fact, the FELyX direct skyline solver fails during memory allocation, while solving the matrix of the large 74476 element model. The completely differing scaling of iterations of ICT PCG and ANSYS PCG, as displayed in 5.4, give evidence that the number of iterations needed by the ANSYS PCG is problem independent. Some hierarchical bases or Multigrid method may explain this behavior.

Table 5.1: Test matrices

elements	nodes	DOFs
2896	6256	18630
2976	6536	19245
3342	7113	21201
3671	7733	23061
4366	8886	26520
6723	12844	38394
7076	13809	40779
9237	17584	51777
11798	21559	64503
16824	29731	88959
22078	38458	115140
39733	66255	198423
74476	119084	356850

Table 5.2: Comparison of different solvers on a small 2896 element mesh

solver	memory	p. time	it. time	tot. time	num. its.
IC(0)	600246	0.243312	212.52	214.833	3874
ICT($1e^{-10}$,250)	3426072	6.2782	1.47124	10.1918	10
Direct skyline	3685608	-	-	5.28263	-
ANSYS PCG	-	-	-	4.21	210

[1]ANSYS is a multi-purpose simulation tool; see http://www.ansys.com/

Table 5.3: Comparison of different solvers on a medium 11798 element mesh

solver	memory	p. time	it. time	tot. time	num. its.
IC(0)	2284038	1.04831	376.78	386.939	1620
ICT($1e^{-8}$,100)	8296486	26.5351	85.268	122.89	213
Direct skyline	29174895	-	-	59.8727	-
ANSYS PCG	-	-	-	16.42	197

Table 5.4: Comparison of different solvers on a large 74476 element mesh

solver	memory	p. time	it. time	tot. time	num. its.
IC(0)	13656213	6.992	4886.98	4995.34	3081
ICT($1e^{-8}$,250)	48793859	758.321	1390.163	2289.18	584
Direct skyline	510305310	-	-	-	-
ANSYS PCG	-	-	-	92.030	144

Figure 5.2: Total time scaling of different solvers

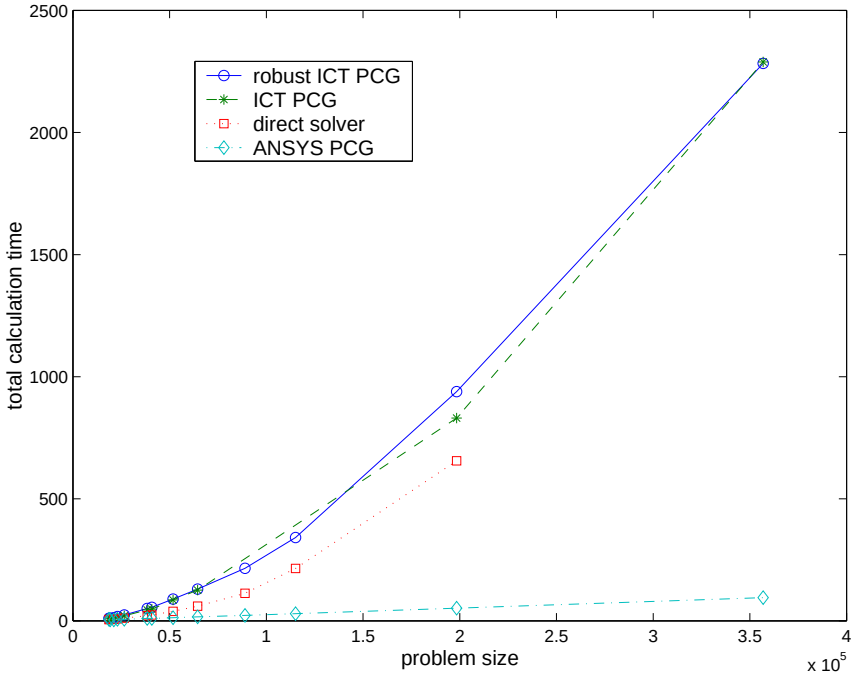


Figure 5.3: Memory scaling of different solvers

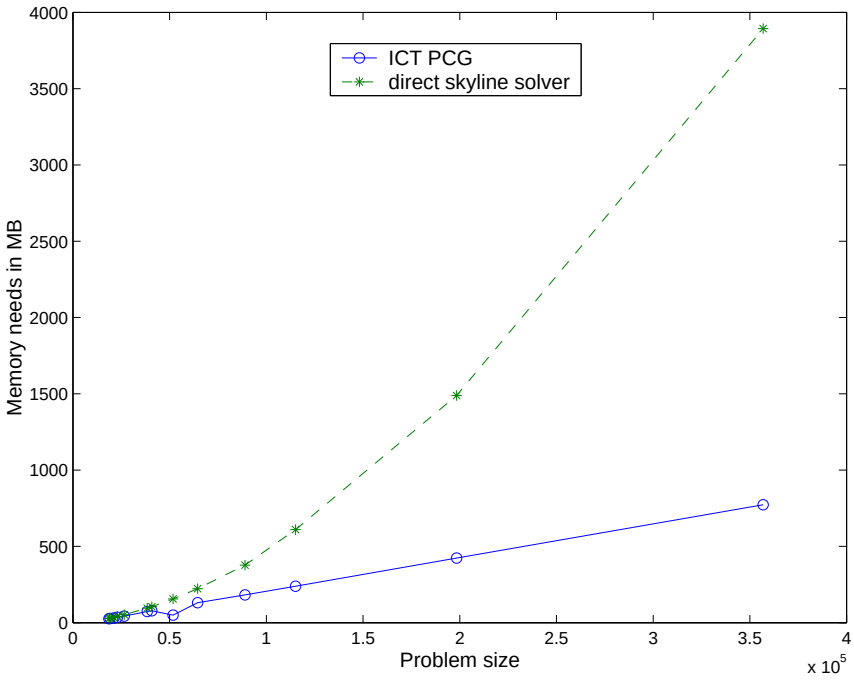
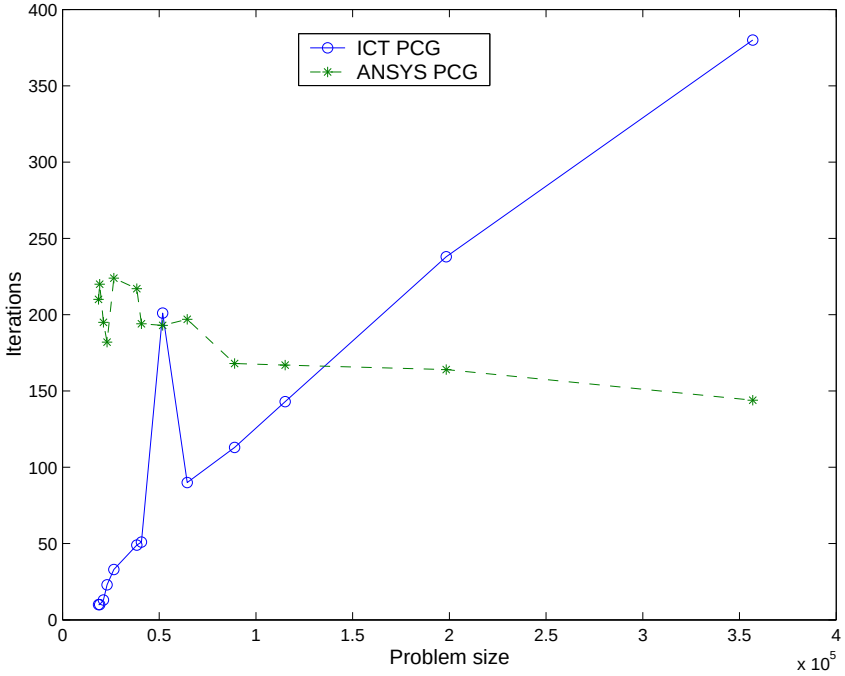


Figure 5.4: Iteration scaling of different solvers



Chapter 6

Conclusion and future work

The flexibility and reliability of the provided solutions in this report enabled the capability of solving a large variety of problems with FELyX. Even on scales where the direct solver quits its service, the capacity of the iterative solver does not tend to bail out. However, comparison with the sophisticated ANSYS PCG solver indicates that much more potential is hidden behind the PCG method.
The lack of theoretical results and published experience made it difficult to find satisfying solutions. Even with the knowledge, that incomplete factorization preconditioned approaches may need hardly less computational work than a direct solver, they were probably the best choice within a short timed semester thesis.
Nevertheless, there are some valuable conclusions, which can be drawn from this report:

- The numerical experiments confirmed the fact that computational effort of incomplete factorization preconditioned CG methods is close to the effort of direct solvers, while memory consumption is much lower.
- The choice of preconditioner is always a trade-off. A too accurate preconditioner consumes a lot of resources and may even slow down the whole solution process, while a too inaccurate preconditioner, leads to a poor convergence rate.
- Since very efficient dense linear algebra packages exist, such as ATLAS[10], it is important to abandon sparse operations, whenever possible.
- The following collection of hints may simplify the application of the newly implemented iterative solvers:
  - The IC(0) preconditioner give best results with Minimum Degree Ordering. Best reordering choice for ICT is Sloan's algorithm.

- The incomplete Cholesky preconditioner with dual threshold strategy showed the best performance for every problem scale, if the parameters are chosen sensibly; select large fill parameters (e.g. 100 or greater) while keeping the drop tolerance in the domain of the iteration convergence criterion.
- It is recommended to use a stabilized version of the IC preconditioner, although, in some special cases, it may perform worse that the unstabilized. However, it certainly avoids bad surprises. The stabilization method of Azij and Jennings [1] proved of value in contrast to the method of Manteuffel [2]

In the future, examining some alternatives might be of interest, rather than trying to enhance incomplete factorization methods. Approximate inverse preconditioners, described in section 4.8 certainly form one major issue, while another promising domain are Multigrid Methods; see [12] for an introduction. Multigrid methods are a prime source of important advances in algorithmic efficiency, finding a rapidly increasing number of users. Unlike other known methods, Multigrid offers the possibility of solving problems with n unknowns with $O(n)$ work and storage, not just for special cases, but for large classes of problems.

Another subject is to integrate the iterative methods into a stochastic optimization procedure, since the Center of Struture Technologies hosts ongoing research projects exploring such optimization methods. The fact, that thousands of quite similar desings have to be evaluated to gain a successful optimization, opens several possibilites of applying an iterative solver:

- If a preconditioner approximates a certain matrix, it also may approximate a second matrix, which is similar to the prevoius one. Thus, the preconditioner can be reused to solve the second matrix and further similar matrices. The solving time then includes only the iteration time, while the preconditioning step is simply omitted. Especially approximate inverse preconditioners, whose application reduces to a simple matrix multiplication during the iteration phase, may perform quite well within this approach.
- The solution of a problem can be reused as an inital guess for further iteration procedures. Like above, one may argue that similar problems have similar solutions. Hence, the solution of a ceratin problem may just be a few iteration steps "away" from the solution of a similar problem.

Of course, these two propositions also can be combined, if desirable.

Appendix A

Using the PCG in FELyX

The PCG solver implemented in FELyX is ready to use with the following preconditioners:

- IC(0) - A zero-fill incomplete cholesky preconditioner
- ICT - A cholesky preconditioner with dual threshold strategy
- Robust ICT - ICT stabilized with Azij and Jennings [1] method
- Shifted ICT - ICT stabilized with Manteuffels [2] method
- SAINV - The preconditioner of Benzi, Cullum and Tuma [4] (not optimized, experimental)
- The identity precontioner

The source code can be found in the following directory: src/itl-4.0.0-1/itl/preconditioner

To invoke the PCG solver, the following lines of code are necessary:

```
//Build a FELyX Object
StructObject FEM( fname, datadir, noiselevel );

FEM.SaveAnsysModel();

//Select appropriate reordering strategy
FEM.NodesReordering( bandwidthAlgorithm );

//Launch the pcg solver
FEM.IterativeSolver(tolerance, precond, lfil, droptol, shift);
```

Where the parameters passed to the methods mean:

- string fname - The filename of the model file
- string datadir - The directory, where the model file is saved
- int noiselevel - A variable to control the amount of text output
- string bandwithAlgorithm - Selects a reordering algorithm. Should be "mmd" when using IC(0), and "sloan" when using ICT
- <float_type> tolerance - This is the relative tolerance specifing the convergence criterion of the pcg solver
- string precond - Selects a preconditioner. This can be one of "ic0", "ict", "rict", "sict", "sainv" or "none".
- int lfil - Specifies the level of fill for ICT preconditioners. This parameter has no effect,if using other preconditioners.
- <float_type> droptol - Determines the drop tolerance. This parameter is only effective in combination with ICT preconditioners or SAINV
- <float_type> shift - Shift parameter α, for the shifted ICT preconditioner only.

Bibliography

[1] M.A. Azij, J. Jennings, A robust incomplete Choleski conjugate gradient algorithm, *International Journal of Numerical Methods in Engineering*,20:949-966 (1984)

[2] T.A. Manteuffel, An Incomplete Factorization Technique for Positive Definite Linear Systems. *Mathematics of Computation*, 34:473-497 (1980)

[3] J.A. Meijerink and H.A. van der Vorst, An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix, *Mathematics of Computation*, 31:148-162 (1977)

[4] M. Benzi, J.K. Cullum M. Tuma, Robust Approximate Inverse Preconditioning for the Conjugate Gradient Method, *SIAM Journal on Scientific Computing*, 22:1318-1332

[5] G. Meurant, Computer Solution of Large Linear Systems, *North Holland, Amsterdam*, (1998)

[6] Y. Saad. Iterative Methods for Sparse Linear Systems. *PWS Publishing Company, Boston*, (1996)

[7] R. Barrett et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, *SIAM Journal on Scientific Computing, Philadelphia*, (1994)

[8] A.Lumsdaine, J. Siek, L. Lee, The Matrix Template Library (MTL), URL: http://www.osl.iu.edu/research/mtl

[9] A.Lumsdaine, J. Siek, L. Lee, The Iterative Template Library (ITL), URL: http://www.osl.iu.edu/research/itl

[10] Automatically Tuned Linear Algebra Software (ATLAS), URL: http://math-atlas.sourceforge.net

[11] O. Koenig, M. Wintermantel, N. Zehnder, The Finite Element Library Experiment, URL: http://felyx.sourceforge.net

[12] Pieter Wesseling, An Introduction to Multigrid Methods, *John Wiley & Sons*, (1992)