ITERATIVE METHODS FOR SOLVING SYSTEMS OF LINEAR EQUATIONS: HYPERPOWER, CONJUGATE GRADIENT AND MONTE CARLO METHODS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2014

Lukas Steiblys

School of Mathematics

Contents

List of Figures **5**

Abstract **6**

Declaration **7**

Copyright Statement **8**

Acknowledgements **10**

1 Introduction **11**

1.1 Systems of Linear Equations 12

1.2 Direct vs. Iterative Methods 14

1.3 Target Hardware 15

1.4 Algorithm-Based Fault Tolerance 16

2 Newton-Schultz Iteration **20**

2.1 The Algorithm 20

2.2 Expansion of the Series 21

2.3 Convergence of Newton-Schultz Iteration 22

2.4 Bounding the Number of Terms 24

2.5 Actual results 25

2.6 Problems With This Method 27

3 Conjugate Gradient Method 28

3.1 The Algorithm 28

3.1.1 Convergence 30

3.2 Synchronization Reduction 32

3.2.1 Theory 32

3.2.2 Results 34

4 Soft Errors and Conjugate Gradient Method 40

4.1 Errors as Perturbations to the Matrix 40

4.2 A Sequence of Sums 42

4.3 A Sequence of Products 43

4.4 Key Problems 44

5 A Monte Carlo Method 46

5.1 Particle Filters 46

5.2 The Algorithm 47

5.3 Results 48

5.4 Further Work 49

6 Summary **50**

6.1 Suggestions for Further Work 51

Bibliography **52**

Appendices **55**

A Code for the Number of Terms **56**

B Modified Conjugate Gradient Method **58**

C Code for the Monte Carlo Method **60**

List of Figures

2.1 General plot of the 2-norm of T_α as a function of α in black, $|1 - \alpha M^2|$ in green and $|1 - \alpha m^2|$ in blue. 22

2.2 Plot of the number of terms that have magnitude greater than 2^{-16} in terms of n and α. 26

3.1 Residual plot for ”af_3_k101” matrix. 35

3.2 Residual plot for ”nasa4704” matrix. 35

3.3 Residual plot for ”olafu” matrix. 36

3.4 Residual plot for ”parabolic_fem” matrix. 36

3.5 Residual plot for ”smt” matrix. 37

3.6 Residual plot for ”thermal1” matrix. 37

5.1 Logarithmic plot of the norm of the smallest residuals at every iteration . . . 48

Abstract

Today's fastest machines used in High-Performance Computing can have hundreds of thousands of cores. To take advantage of all the available processing power, algorithms that do computations in parallel must be used. Some of the limiting factors of these algorithms are the need for synchronization between different computational units and the amount of communication that has to be done between them. In addition, as the number of cores grows, useful results may not be achievable because of the failure of these cores.

In this thesis, named "Iterative Methods for Solving Systems of Linear Equations: Hyperpower, Conjugate Gradient and Monte Carlo Methods" and submitted to the University of Manchester by Lukas Steiblys for the degree of Master of Philosophy in May 2014, modifications to Newton-Schultz iteration and Conjugate Gradient algorithms are investigated for applicability to solving large systems of linear equations that may also have better performance on multi-core machines than then standard versions of the algorithms. The notions of hard-error and soft-error are also explained, a solution for mitigation of soft-errors in the Conjugate Gradient Method is proposed and its efficacy is examined. In the end, a different method for solving systems of linear equations that incorporates random sampling is introduced.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright Statement

Acknowledgements

I would like to thank Professor David J. Silvester for the support he provided during the first year of study and in the compilation of my thesis. His comments and guidelines helped form the first version of this thesis.

Secondly, I could not have done this without Professor Jack Dongarra who provided the initial ideas for my research and gave pointers when I was stuck.

Professor Francoise Tisseur and Dr Craig Lucas have also greatly helped me improve the thesis and reach its final shape.

Finally, I am grateful to my parents for their support and interest in all the things I do.

Chapter 1

Introduction

Throughout this thesis we will be concerned with solving large systems of linear equations usually arising from the discretization of models for solving physical problems. Three distinct iterative algorithms will be presented that take advantage of current highly-parallel architectures of supercomputers and rely heavily on matrix-vector multiplications.

The first algorithm is a modification of the Newton-Schultz iteration for matrix inversion to solve linear systems of equations. The next two chapters describe a modified Conjugate Gradient method for linear systems with symmetric definite matrices that tries to reduce the number of synchronization points to shorten the running time. The last chapter introduces a Monte Carlo algorithm that does not rely on the matrix having any special structure and makes use of building a large set of approximate solutions and independently sampling it with replacement based on their accuracy.

Next I will present the background material necessary to understand the rest of the thesis.

1.1 Systems of Linear Equations

A linear equation in the n variables $x_1, x_2, ..., x_n$ is an equation that can be written in the form

$$a_1x_1 + a_2x_2 + ... + a_nx_n = b, \tag{1.1}$$

where the coefficients $a_1, a_2, ..., a_n$ and the constant term b are constants. A system of linear equations is a finite set of linear equations, each with the same variables. A solution of a system of linear equations is a set of values for each variable x_i that are simultaneously a solution of each equation in the system. This set of values is often written in a so-called vector format as $v = [x_1, x_2, ..., x_n]$ where the variable name is replaced with the actual value. An example of a system of m linear equations in n variables is

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n &= b_2 \\
\vdots &= \vdots \\
a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n &= b_m.
\end{aligned}$$

This is usually written in matrix form as

$$Ax = b, \tag{1.2}$$

where $x = [x_1, x_2, ..., x_n]^T, b = [b_1, b_2, ..., b_m]^T$, A is a matrix with the entry in row i and column j equal to a_{ij} and Ax is the standard matrix-vector product between matrix A and vector x.

We are trying to find the solution vector x that satisfies these m equations. A matrix $A \in \mathbb{R}^{n \times n}$ is called non-singular when it has an inverse matrix $A^{-1} \in \mathbb{R}^{n \times n}$ such that $AA^{-1} = A^{-1}A = I$, where I is the identity matrix, or equivalently, for every vector $b \in \mathbb{R}^{n \times 1}$ there is a corresponding vector $x \in \mathbb{R}^{n \times 1}$ that satisfies the equation $Ax = b$. Only square matrices can be non-singular and the rank of such a matrix is equal to the number of rows or columns of a matrix. The rank of a matrix A is the size of the largest collection of linearly independent columns of A. A set of vectors $\{v_1, v_2, ..., v_n\}$ are linearly independent when the only solution to the equation $a_1v_1 + a_2v_2 + ... + a_nv_n = 0$ is $a_1 = a_2 = ... = a_n = 0$. For a more detailed explanation of "rank", see [8] p.49. The span of a set of vectors $\{v_1, v_2, \ldots, v_n\}$ is the set of all linear combinations of those vectors:

$$span(\{v_1, v_2, \ldots, v_n\}) = \{a_1v_1 + a_2v_2 + \cdots + a_nv_n | \forall a_1, a_2, \ldots, a_n \in \mathbb{R}\}. \tag{1.3}$$

A matrix is called sparse if most of the entries a_{ij} are zero. In many algorithms, sparseness can be exploited to reduce the number of operations performed to calculate the answer.

An eigenvalue λ of a square matrix A is a real or complex scalar such that $Ae = \lambda e$ for some vector e, called an eigenvector of A. A matrix can have up to n distinct eigenvalue and eigenvector pairs, where n is the dimension of the matrix. A real symmetric square matrix A is called positive-definite if $x^T Ax > 0$ for all vectors $x \neq 0$ and in that case all the eigenvalues of A are real and positive.

1.2 Direct vs. Iterative Methods

Direct methods for solving systems of linear equations try to find the exact solution and do a fixed amount of work. Unfortunately, the exact solution may not be found using conventional computers because of the way real numbers are approximated and the arithmetic is performed. The errors introduced during computation can compound and render the method unusable for very large systems of equations. In addition, the number of arithmetic operations to be done to solve a large system using direct methods may become infeasible and in the case where many of the coefficients a_{ij} are zero, a lot of this work is redundant. Iterative methods try to find the solution by generating a sequence of vectors that are approximate solutions to the system of equations. These methods try to approach the actual solution as fast and as accurately as possible. This means the algorithm can be stopped once the sequence of vectors approaches a solution that is close enough to the actual solution. In addition, iterative methods can sometimes take better advantage of the structure found in the matrix. A lot of matrices in practice, especially when considering physical models, are sparse, meaning that most of the coefficients a_{ij} are equal to zero. In that case iterative methods arrive at a solution in a shorter period of time than direct methods and perform computations only when necessary.

1.3 Target Hardware

The machines used to perform calculations in high-performance scientific computing industry are massively parallel, meaning that they can run millions of tasks simultaneously. The machines usually have multiple racks of interconnected nodes and each node has a bunch of processors running independently and all these processors come with multiple cores that can also work independently. For a list of the fastest supercomputers in the world, see [10].

The tasks that are performed need to coordinate with one another and usually algorithms have many synchronization points, where the algorithm has to stop until all the data that is necessary to move on to the next step of the process has to be computed and collected.

Numerical computation happens in floating point arithmetic [7]. The most common standard is the IEEE 754 standard [1] and it is used in almost all modern processors for representing real numbers in finite number of bits (32 or 64 bits, called single and double precision respectively). Not every real number can be represented in that standard as there is an infinite number of real numbers, but only a finite number of bits to store them, and the real numbers have to be approximated. Great care must be taken in designing the algorithms so that these approximations can be used to compute meaningful answers.

Performance can be measured in two main ways: the amount of time an algorithm takes to produce an answer or the number of FLOPs (Floating-point operations) performed to arrive at the answer. The first method is fairly self explanatory and the second one is the total number of all the elementary mathematical operations performed. An elementary operation is an addition, subtraction, multiplication and division of two floating point numbers, although variations on what can be considered and elementary operation exists and sometimes include the computation of other elementary functions such as exponentials and trigonometric functions and the multiply-add operation. Because of the parallel nature of the execution of algorithms, the relation between the running time and the number of operations performed is not always direct.

The algorithms presented here rely heavily on doing a lot of matrix-vector multiplications. These operations can be performed in parallel using a multi-threaded BLAS (Basic Linear Algebra Subprograms [9]) software library. We do not delve into the details of BLAS in this thesis, but rather build on top of it by attempting to perform multiple matrix-vector multiplications simultaneously.

1.4 Algorithm-Based Fault Tolerance

Transistors, used as electronic switches, are the building blocks of all processors operating in practice today. As the transistors used in modern processors and memory cells shrink and the number of cores that can perform tasks independently in a processor grows, failure to perform their duty becomes a real possibility. We need to be aware of the errors that this increased failure rate might introduce during the execution of the algorithms.

Failure types can be split into two broad categories: hard errors and soft errors. Soft errors are one time events that are mostly caused by cosmic radiation. They invalidate the data stored in a memory cell or affect the flipping of a gate in a microprocessor, but do not damage the device permanently. This type of error does not halt the entire computation or affect it in any noticeable way, but the small error that is introduced might propagate during subsequent steps of the algorithm and render most or all of the results useless. Special care needs to be taken in detecting and rectifying soft errors, especially with long running computations prevalent in high-performance scientific computing that might take days to come up with an answer. On the other hand, hard errors completely halt the algorithm because of a computer failure or network connectivity problems.

Algorithm-based fault tolerance moves the responsibility of detecting and correcting errors from the hardware side to the software side. One example of that is matrix-matrix and matrix-vector multiplications that can use checksums to detect and correct errors introduced during computation. Suppose we have a n-by-m matrix A

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}. \tag{1.4}$$

The column checksum matrix A_c of matrix A is defined as an (n+1)-by-m matrix with the matrix A in the first n rows and the last row is generated as $a_{n+1,j} = \sum_{i=1}^{n} a_{i,j}$. Thus

$$A_c = \begin{bmatrix} A \\ e_c A \end{bmatrix}, e_c = [1, 1, ..., 1] \in \mathbb{R}^{1\times n}. \tag{1.5}$$

Similarly, the row checksum matrix A_r of matrix A is defined as an n-by-(m+1) matrix with the matrix A in the first m columns and the last column is generated as $a_{i,m+1} = \sum_{j=1}^{m} a_{i,j}$.

$$A_r = \begin{bmatrix} A & Ae_r \end{bmatrix}, e_r = [1, 1, ..., 1]^T \in \mathbb{R}^{m\times 1}. \tag{1.6}$$

Now when we want to multiply matrices $A \in \mathbb{R}^{n\times m}$ and $B \in \mathbb{R}^{m\times k}$ to get $C \in \mathbb{R}^{n\times k}$, we just multiply their corresponding column and row checksum matrices A_c and B_r. The result is a full checksum matrix C_f

$$A_c \times B_r = \begin{bmatrix} A \\ e_c A \end{bmatrix} \times \begin{bmatrix} B & Be_r \end{bmatrix} = \begin{bmatrix} C & Ce_r \\ e_c C & e_c C e_r \end{bmatrix} = C_f \in \mathbb{R}^{(n+1)\times(k+1)}. \tag{1.7}$$

The checksums can then be used to detect, locate and correct a single erroneous element using the following procedure:

1) Find the column and row sums of C

$$SC = [sc_1, sc_2, ..., sc_k], sc_i = \sum_{j=1}^{n} c_{i,j} \tag{1.8}$$

$$SR = [sr_1, sr_2, ..., sr_n]^T, sr_j = \sum_{i=1}^{k} c_{i,j} \tag{1.9}$$

2) Compare each sum in SC and SR with the corresponding checksum elements e_cC and Ce_r. Suppose we detect inconsistent elements $sc_i \neq (e_cC)_i$ and $sr_j \neq (Ce_r)_j$. That means an error occurred when computing the element of C located at row i and column j.

3) The error can be corrected by finding the true value of

$$c_{i,j} = \hat{c_{i,j}} - sc_j + (e_cC)_j = \hat{c_{i,j}} - sr_i + (Ce_r)_i, \tag{1.10}$$

where $\hat{c_{i,j}}$ is the erroneous element that was computed.

We will be using this technique to identify the errors and compute their magnitude in later chapters on fault-tolerant algorithms.

Chapter 2

Newton-Schultz Iteration

Newton-Schultz iteration, also known as the hyperpower method, is an iterative procedure for finding the inverse of a matrix A if A is non-singular and its pseudo-inverse otherwise [2]. Here I will describe the method and try to apply it to solving a system of linear equations instead of computing the inverse of A.

2.1 The Algorithm

Given a non-singular matrix A its inverse can be estimated using an iterative process $X_{i+1} = 2X_i - X_iAX_i$ such that $X_1 = \alpha A^T$ for $0 < \alpha < 2/\lambda_L^2$, where λ_L^2 is the largest eigenvalue of A^TA and A^T is the transpose of A (reflected over the main diagonal). As $i \to \infty, X_i \to A^{-1}$. Thus, the solution to $Ax = b$ is estimated by $x = X_ib$ and the estimate approaches the actual solution as $i \to \infty$. For more details about convergence, see Section 2.3 of this thesis or [2].

2.2 Expansion of the Series

Usually we do not need to find the inverse of A, but simply a solution x to $Ax = b, x = A^{-1}b$. In case of a sparse matrix A, A^{-1} may not be sparse and for large dimensions we might not even be able to store the inverse or the intermediate results. If we replace A^{-1} with X_i for some $i > 1$ and expand X_i in terms of X_1, it might be possible to find an approximate solution to x just by doing matrix-vector multiplications and summing them up. We hope that finding just the solution x instead of the whole matrix A^{-1} will require less computational and memory resources. Let us first expand X_i for $i = 2, 3, 4$ in terms of X_1:

$X_2 = 2X_1 - X_1AX_1$

$X_3 = 4X_1 - 6(X_1A)X_1 + 4(X_1A)^2X_1 - (X_1A)^3X_1$

$X_4 = 8X_1 - 28(X_1A)X_1 + 56(X_1A)^2X_1 - 70(X_1A)^3X_1 + 56(X_1A)^4X_1 - 28(X_1A)^5X_1 + 8(X_1A)^6X_1 - (X_1A)^7X_1$

The general form of the sum is

$$X_{n+1} = \sum_{k=1}^{2^n} \binom{2^n}{k} (-1)^{k-1} (X_1A)^{k-1} X_1. \tag{2.1}$$

The number of terms to be added grows exponentially as 2^n, but the good news is that for a reasonably small α parameter, used for computing $X_1 = \alpha A^T$, a lot of the terms at the end

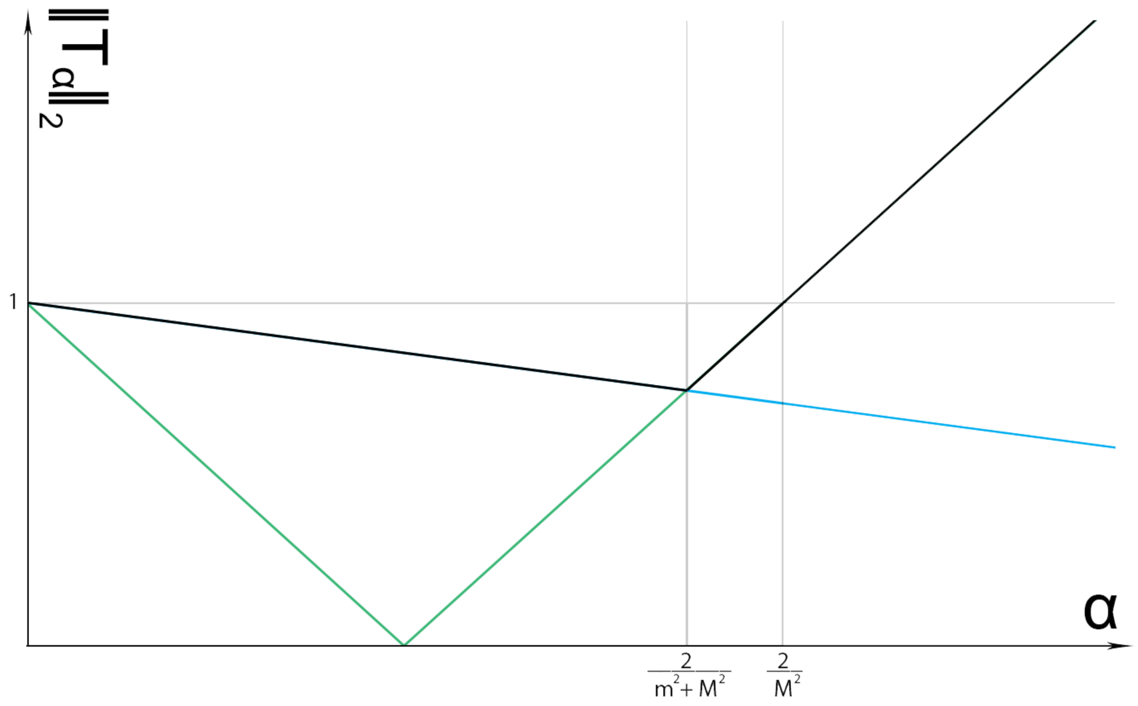

Figure 2.1: General plot of the 2-norm of T_α as a function of α in black, $|1 - \alpha M^2|$ in green and $|1 - \alpha m^2|$ in blue.

of the expanded series have very small magnitude and can be discarded without sacrificing the accuracy of the result. We can compute an upper bound of the norm of the sum of terms at the end of the series. If the norm is sufficiently small, these last terms have very little effect on the solution.

2.3 Convergence of Newton-Schultz Iteration

The method can be constructed by taking a sequence of X_i with the property

$$I - X_{i+1}A = (I - X_iA)^2. \tag{2.2}$$

and from that if follows that

$$I - X_i A = (I - X_1 A)^{2^i} \tag{2.3}$$

$$||I - X_i A||_2 = ||I - X_1 A||_2^{2^i}. \tag{2.4}$$

If the magnitude of $||I - X_i A|| < 1$, then the magnitude of $||I - X_{i+1} A|| = (||I - X_i A||)^2$ will be even smaller and X_i will get closer and closer to A^{-1}. So the convergence of the Newton-Schultz iteration depends on the magnitude of

$$T_\alpha = I - X_1 A = I - \alpha A^T A. \tag{2.5}$$

The magnitude of $||T_\alpha||_2 = \max \frac{||T_\alpha x||_2}{||x||_2}, x \neq 0$, where $||x||_2 = \sqrt{\sum_i^n x_i^2}$, and is also known as the 2-norm of T_α. If $||T_\alpha|| < 1$ then the method will converge as can be seen in Equation (2.4) and this can be achieved by choosing α in the range $(0, 2/\lambda_L^2)$ where $\lambda_L^2 = ||A||_2^2$ and is equal to the largest eigenvalue of $A^T A$. Because $A^T A$ is symmetric positive definite, all the eigenvalues of $A^T A$ are real and positive. From Equation (2.4) we can infer that the speed of convergence of the hyperpower method can be chosen by using the appropriate expansion of the series. In the Newton-Schultz case it is quadratic.

The range of values of α that satisfy $||T_\alpha|| < 1$ can be computed by taking an eigenvector e of $A^T A$ and its corresponding eigenvalue λ^2 and multiplying it by T_a. Then $T_\alpha e = (I - \alpha A^T A)e = (1 - \alpha\lambda^2)e$. Suppose the largest and smallest eigenvalues of $A^T A$ are λ_L^2 and λ_S^2. For $\alpha = 0, ||T_0||_2 = ||I||_2 = 1$. Then for some $\alpha_c, 0 \leq \alpha \leq \alpha_c, ||T_\alpha||_2 = 1 - \alpha\lambda_S^2$ until the largest eigenvector e_L of $A^T A$ starts dominating the 2-norm of T_α again. The minimum of

T_α is achieved at $\alpha = \alpha_c = \frac{2}{\lambda_L^2 + \lambda_S^2}$, at the intersection of $|1-\alpha\lambda_L^2|$ and $|1-\alpha\lambda_S|$ where $\alpha \neq 0$. A plot of the 2-norm of T_a can be seen in figure 2.1.

We are interested in values of α that are greater than and close to zero that make the 2-norm of the term $(X_1A)^k = (\alpha A^T A)^k$ in the sum for X_i in equation (2.1) decrease rapidly. The decrease has to be fast enough so that after multiplication by $\binom{2^n}{k}$ the term stays within reasonable bounds.

2.4 Bounding the Number of Terms

An upper bound on the error of X_n when approximating A^{-1} is

$$||A^{-1} - X_n||_2 \leq \frac{1}{\lambda_S} ||T_\alpha||_2^{2^n}. \tag{2.6}$$

It can be derived from equation (2.2) by applying it recursively to get $I - AX_i = (T_1^2)^n$, multiplying both sides by A^{-1} and taking the 2-norm. If X_nb is an approximate solution for x in $Ax = b$, then $||x - X_nb||_2 \leq \frac{||T_\alpha||_2^{2^n}||b||_2}{\lambda_S}$. This allows us to calculate the minimum n for which X_nb is as close to the real solution as we want. Assuming $0 < ||T_\alpha||_2 < 1, \lambda_S > 0$ and we want the error to be less than $2^{-p}, p > 0$

$$||x - X_nb||_2 \leq \frac{||T_\alpha||_2^{2^n}||b||_2}{\lambda_S} \leq 2^{-p} \implies n \geq \log_2\left(\frac{\log_2(\frac{\lambda_S}{||b||_2}) - p}{\log_2 ||T_\alpha||_2}\right). \tag{2.7}$$

Once we decide on the α and n, we bound the number of terms to be added. The series in equation (2.1) is an alternating series. This means that once we find a term of the vector sequence that has a ∞-norm, which is equal to the largest absolute value of all the elements of a vector, that is smaller than 2^{-p}, we can discard the rest of the terms. That happens when

$$||\binom{2^n}{k}(-1)^{k-1}(X_1A)^{k-1}X_1b||_\infty \leq \binom{2^n}{k}||X_1A||_\infty^{k-1}||X_1b||_\infty \leq 2^{-p},$$

where $\|X_1A\|_\infty = \max \frac{\|X_1Ax\|_\infty}{\|x\|_\infty}, x \neq 0$.

Suppose $||X_1A||_\infty \leq 2^{-m}$ and $||X_1b||_\infty \leq 2^{-q}$. Then we have to find k such that

$$\binom{2^n}{k}||(X_1A)||_\infty^{k-1}||X_1b||_\infty \leq \frac{2^{nk}}{k!}2^{-m(k-1)}2^{-q} \leq 2^{-p}$$

or

$$-\sum_{i=1}^{k}\log_2(k) + (n-m)k + m - q \leq -p.$$

2.5 Actual results

The method works best when the condition number $\kappa(A^TA)$ (the ratio of the largest and smallest eigenvalues of A^TA) is close to 1, because in that case $||T_\alpha||_2$ decreases much faster with increasing α.

For the experiment, a real square matrix A of size 200x200, which was a sum of the identity matrix and a matrix with elements drawn from a uniformly random distribution, was used. The test was performed using the GNU Octave software package working with double precision arithmetic. The smallest eigenvalue of A^TA was equal to 0.94405, the largest one was 2.2506 (condition number of 2.38398). For an expression of X_6 in terms of X_1 only the first

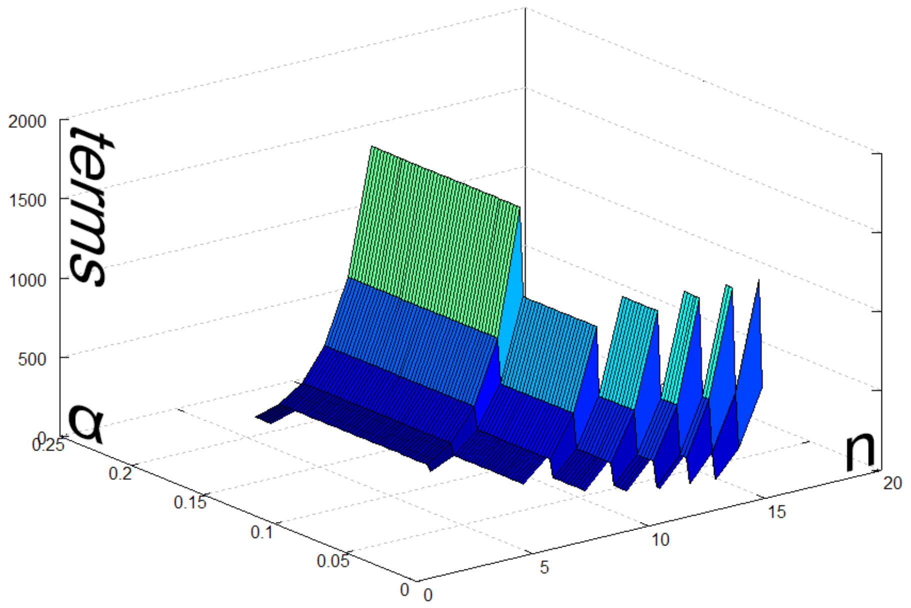

Figure 2.2: Plot of the number of terms that have magnitude greater than 2^{-16} in terms of n and α.

54 out of 64 terms of equation (2.1) need to be added to get the ∞-norm of the error of the approximate solution down to less than 2^{-16}. The plot of the number of terms that have magnitude greater than 2^{-16} can be seen in Figure 2.2 where $\alpha \in [0; 0.25]$ on one horizontal axis and $n \in [0; 20]$ on the other one. It was generated using the code provided in Appendix A.

Some amount of parallelism in the algorithm could be extracted by doing the summing independently once you have the terms of the sum ready. Also, even though there is a lot dependence among the terms themselves, each one can be computed in multiple ways. For example, the vector term $(X_1A)^8X_1b$ can be found by applying the matrix X_1A to X_1b eight times, or multiplying X_1A by itself three times to get $((X_1A)^2)^2)^2 = (X_1A)^8$ and then multiplying that by X_1b. Whether that is something worth doing needs further investigation.

2.6 Problems With This Method

The biggest problem is that the terms in the sequence grow extremely large and this results in catastrophic cancellation error that causes significant loss of accuracy in the last digits of the computed solution. For larger values of n and larger condition numbers, zero digits of accuracy can be computed. One idea to mitigate this cancellation error could be to orthogonalize the set of basis vectors $(X_1A)^kX_1b$ for $k = 0, 1, 2...$ used in the expansion of the series. However, all this does is it moves the main problem of computing an alternating series with coefficients of large magnitude for the solution of $Ax = b$ to computing the coefficients for the expansion in the new basis. The coefficient for the first vector in this basis set is now $\sum_{k=1}^{2^n} \binom{2^n}{k}(-1)^{k-1}\alpha^k c_i$ where $c_i = ((X_1A)^iX_1b)^TX_1b$. It is another alternating series with coefficients of large magnitude, but its result is now a scalar, not a vector. It is not clear if it is possible to compute this numerically in a practical way and currently I am not aware of any way solve the cancellation error problem.

Another disadvantage is that other, more popular Krylov subspace iterative methods, that will be analysed in more detail in the following chapters, reach the desired accuracy much quicker compared to the described algorithm and do it in a stable way. By "stable" we mean that the solution will not diverge.

Chapter 3

Conjugate Gradient Method

The Conjugate Gradient method is an iterative algorithm for finding a solution to a large and sparse linear system of equations whose corresponding matrix A is real, symmetric ($A^T = A$) and positive-definite ($x^T A x > 0$ for all non-zero vectors x). In this chapter the algorithm will be described briefly and a way to reduce the need for synchronization will be introduced.

3.1 The Algorithm

Given a matrix A, a right-hand side vector b and an initial guess at a solution x_0 (usually $x_0 = b$ or $x_0 = 0$), the algorithm is trying to minimize the quadratic function $f(x) = \frac{1}{2}x^T A x - x^T b$. The point that minimizes this function is the actual solution to $Ax = b$. At each iteration of the algorithm, a search direction d_i is chosen so that it is A-orthogonal to and linearly independent from all the other previous search directions. A-orthogonality means that $d_i^T A d_j = 0$ for all $j < i$.

After we find a suitable direction vector d_k, a step in that direction is taken $x_{k+1} = x_k + \alpha_k d_k$ using an α_k that minimizes the quadratic function $f(x_{k+1})$. Thus, a sequence $x_0, x_1, ..., x_k$ is generated and $x_k \to x$ as $k \to n-1$, where x is the actual solution to $Ax = b$. Theoretically, the algorithm reaches the exact solution after n iterations because the search directions $d_0, d_1, ..., d_{n-1}$ cover the whole search space. In practice, however, the floating point arithmetic used in computers prevents us from finding the exact solution, but we can get sufficiently close. Pseudocode for the conjugate gradient method is shown in Algorithm 1.

Algorithm 1 Find a solution to Ax = b using the Conjugate Gradient method

```
Input: matrix $A$, vector $b$
Output: solution $x_n$
$x_0 := 0$
$n :=$ row count of $A$
$r_0 := b - Ax_0$
$d_0 := r_0$
$k := 0$
**while** $k < n$ **do**
    $\alpha_k := \frac{r_k^T r_k}{d_k^T A d_k}$
    $x_{k+1} := x_k + \alpha_k d_k$
    $r_{k+1} := r_k - \alpha_k A d_k$
    $\beta_{k+1} := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $d_{k+1} := r_{k+1} + \beta_{k+1} d_k$
    $k := k + 1$
**end while**
```

Usually the algorithm is stopped after much fewer than n iterations. The magnitude of the residual r_k can be a good indicator of when to halt the program when it drops below some set threshold. For more information about the stopping criterion as it applies to the finite element method, see [12].

3.1.1 Convergence

After k iterations of the Algorithm 1, an approximate solution x_k to $x = A^{-1}b$ is chosen from the Krylov subspace $K_k(A, r_0) := span(\{r_0, Ar_0, A^2r_0, ..., A^{k-1}r_0\})$ such that the error $e_k = x - x_k$ is minimized over the A-norm $||e_k||_A = e_k^T Ae_k$. Expand x_k in the form

$$x_k = \sum_{j=0}^{k-1} \alpha_j A^j r_0 \tag{3.1}$$

given some coefficients $\alpha_0, \alpha_1, ..., \alpha_{k-1}$. We can express this as $x_k = q_{k-1}(A)r_0$ where $q_{k-1}(\zeta)$ is a polynomial of the form

$$q_{k-1}(\zeta) = \sum_{j=0}^{k-1} \alpha_j \zeta^j. \tag{3.2}$$

Since $x_0 = 0$, $e_0 = x - x_0 = x$, $r_0 = b - Ax_0 = b$ and we can express e_k as

$$e_k = e_0 - x_k = e_0 - q_{k-1}(A)r_0 = p_k(A)e_0, \tag{3.3}$$

where $p_k(\zeta) = 1 - \zeta q_{k-1}(\zeta)$ is a polynomial of degree k.

As previously stated, $||e_k||_A$ is minimized over all the vectors in the Krylov subspace $K_k(A, r_0)$ and this gives us a way to characterize the polynomial $p_k(\zeta)$ as a polynomial of order k with a constant term equal to 1 that minimizes

$$||e_k||_A = \min_{p_k \in P_k, p_k(0)=1} ||p_k(A)e_0||_A, \tag{3.4}$$

where P_k is the set of all real polynomials of degree k.

An upper bound on $||e_k||_A$ can be found if we expand e_0 in the basis of eigenvectors v_i of A, where $Av_i = \lambda_i v_i$:

$$e_0 = \sum_{j=1}^{n} \gamma_j v_j. \tag{3.5}$$

Combining the last two equations we can bound the error e_k:

$$\begin{aligned} \|e_k\|_A &= \min_{p_k \in P_k, p_k(0)=1} \| \sum_{j=1}^{n} \gamma_j p_k(\lambda_j) v_j \|_A \\ &\leq \min_{p_k \in P_k, p_k(0)=1} \| \max_j |p_k(\lambda_j)| \sum_{j=1}^{n} \gamma_j v_j \|_A \\ &= \min_{p_k \in P_k, p_k(0)=1} \max_j |p_k(\lambda_j)| \|e_0\|_A. \end{aligned} \tag{3.6}$$

If we can find an interval $[a, b]$ such that $\lambda_j \in [a; b]$ for all j (optimally $a = \min_j \lambda_j, b = \max_j \lambda_j$), then

$$\frac{\|e_k\|_A}{\|e_0\|_A} \leq \min_{p_k \in P_k, p_k(0)=1} \max_{x \in [a;b]} |p_k(x)|. \tag{3.7}$$

Polynomials $p_k(x)$ that minimize the upper bound of equation (3.7) are known (see [14]) and they are a scaled and translated version of Chebyshev polynomials of the first kind:

$$\chi_k(x) = \frac{\tau_k\left(\frac{b+a}{b-a} - \frac{2x}{b-a}\right)}{\tau_k\left(\frac{b+a}{b-a}\right)}. \tag{3.8}$$

Standard Chebyshev polynomials $\tau_0, \tau_1, ...$ are given by $\tau_k(x) = \cos(k \cos^{-1} x)$ and it can be shown that

$$\tau_k(x) = \frac{1}{2}\left[(t + \sqrt{t^2 - 1})^k + (t - \sqrt{t^2 - 1})^k\right]. \tag{3.9}$$

If $a = \min_j \lambda_j$ and $b = \max_j \lambda_j$, then the condition number of A is $\kappa = \frac{\max_j \lambda_j}{\min_j \lambda_j} = \frac{b}{a}$ and using equation (3.9) we get

$$\tau_k\left(\frac{b+a}{b-a}\right) = \tau_k\left(\frac{\kappa+1}{\kappa-1}\right) = \frac{1}{2}\left[\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k + \left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k\right] \geq \frac{1}{2}\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k. \quad (3.10)$$

The magnitude of the numerator in equation (3.8) is always less than or equal to one for $x \in [a; b]$ and combining equation (3.7) with (3.8) and (3.10) we can establish the following upper bound on $\|e_k\|_A$:

$$\|e_k\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|e_0\|_A. \quad (3.11)$$

This shows that for well-conditioned matrices the conjugate gradient method converges very quickly. More on the convergence of the conjugate gradient method can be found in [6], pages 72-75.

3.2 Synchronization Reduction

3.2.1 Theory

The conjugate gradient method is highly synchronous. The computation of d_k at iteration k depends on having already computed β_k and r_k, β_k depends on r_k, r_k depends on α_{k-1} and α_{k-1} depends on d_{k-1}. In this section we will introduce a way to compute the search direction d_k without directly relying on already having r_k or x_k.

Assume you are doing the k-th iteration so you already have k A-orthogonal search directions

$d_1, d_2, ..., d_k$. Then the next search direction can be computed as follows:

1. Compute $q = Ad_k$.
2. A-orthogonalize q and d_k to get q' ($q' = q - \frac{d_k^T Aq}{d_k^T Ad_k} d_k$).
3. A-orthogonalize q' and d_{k-1} to get d_{k+1} ($d_{k+1} = q' - \frac{d_{k-1}^T Aq}{d_{k-1}^T Ad_{k-1}} d_{k-1}$).

d_{k+1} is automatically A-orthogonal to $d_1, d_2, ...d_{k-2}$ because Ad_k is A-orthogonal to $d_1, d_2, ...d_{k-2}$. We can prove the A-orthogonality of Ad_k with $d_j, j < k-1$ as follows:

Lemma 3.2.1. *Assume we are working in infinite precision arithmetic. Then either the set of vectors $K_k = \{b, Ab, A^2b, ..., A^{k-1}b\}$ are linearly independent or the solution to $Ax = b$ has already been found at the k-th iteration of the conjugate gradient method.*

Proof. Suppose the set of vectors K_k are not linearly independent for some k and some vector A^ib for $i \in \{0, 1, 2, ..., k-1\}$ can be expressed as a linear combination of vectors in $\{A^jb\}_{j=0,2..i-1} = K_i$. This implies that $A^{i+1}b = AA^ib$ can also be expressed in terms of vectors in K_i and $span(K_{i+1}) = span(K_i)$. The conjugate gradient method chooses x_i to be a linear combination of vectors in K_k such that the residual r_k is minimized. The CG algorithm solves the problem in a finite number of iterations n, $r_n = 0$. As the span of K_k is the same as of K_i, r_i must be zero and the solution has already been found at iteration i or the vectors in K_k are linearly independent. □

Theorem 3.2.2. *Assume the solution has not yet been found at the k-th iteration of the conjugate gradient method. Then Ad_k is A-orthogonal to $d_1, d_2, ...d_{k-2}$ where d_i is the search direction used at i'th iteration.*

Proof. Take any $d_i, i \in \{1, 2, ..., k-2\}$. Express Ad_i in terms of $d_1, d_2, \ldots, d_{k-1}$: $Ad_i = a_1d_1 + a_2d_2 + \cdots + a_{k-1}d_{k-1}$. This can be done because the d_i's span the Krylov subspace $\{d_1, Ad_1, ...A^{i-1}d_1\}$. Since the set of vectors $\{d_1, d_2, ...d_{k-1}\}$ is linearly independent by Lemma 3.2.1, the expansion is unique and $a_j = 0$ for $j > i+1$. Now $(d_i, Ad_k)_A = d_i^T AAd_k = (Ad_i)^T Ad_k$ because A is symmetric.

$$(Ad_i)^T Ad_k = (a_1d_1 + a_2d_2 + \cdots + a_{k-2}d_{k-2})Ad_k = a_1d_1Ad_k + a_2d_2Ad_k + \cdots + a_{k-1}d_{k-1}Ad_k = 0$$

because d_i is A-orthogonal to d_j when $i \neq j$. Thus, Ad_k is A-orthogonal to d_i. □

Now use this new search direction d_{k+1} to update x_{k+1} and r_{k+1} :

$\alpha_{k+1} = (d_{k+1}, r_i)/(d_{k+1}, Ad_{k+1})$

$x_{k+1} = x_k + \alpha_{k+1}d_{k+1}$

$r_{k+1} = r_k - \alpha_{k+1}Ad_{k+1}$.

The nice thing about this is that the search directions can be computed independently of any x_i and r_i. The computations can be done in parallel and the search vectors used as soon as they are available to minimize the residual. This allows the algorithm to run in a slightly more parallel way, improving its performance on modern multi-core processors.

3.2.2 Results

The algorithm was tested on six different real symmetric positive-definite matrices taken from the University of Florida Sparse Matrix Collection [4] that had the right hand side b in $Ax = b$ bundled together. The results are summarized in table 3.1. A visual plot of the

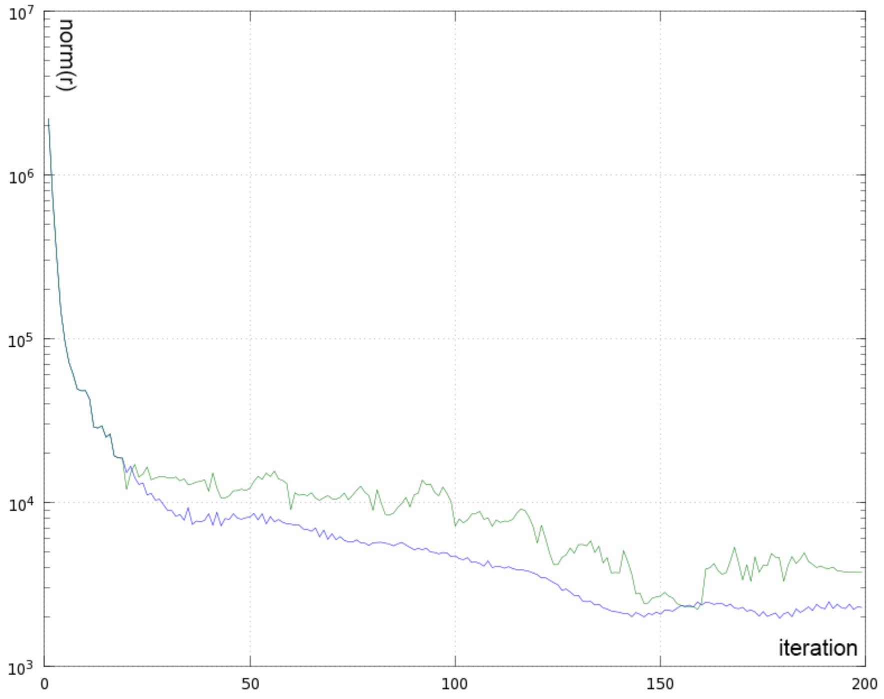

Figure 3.1: Residual plot for "af_3_k101" matrix.

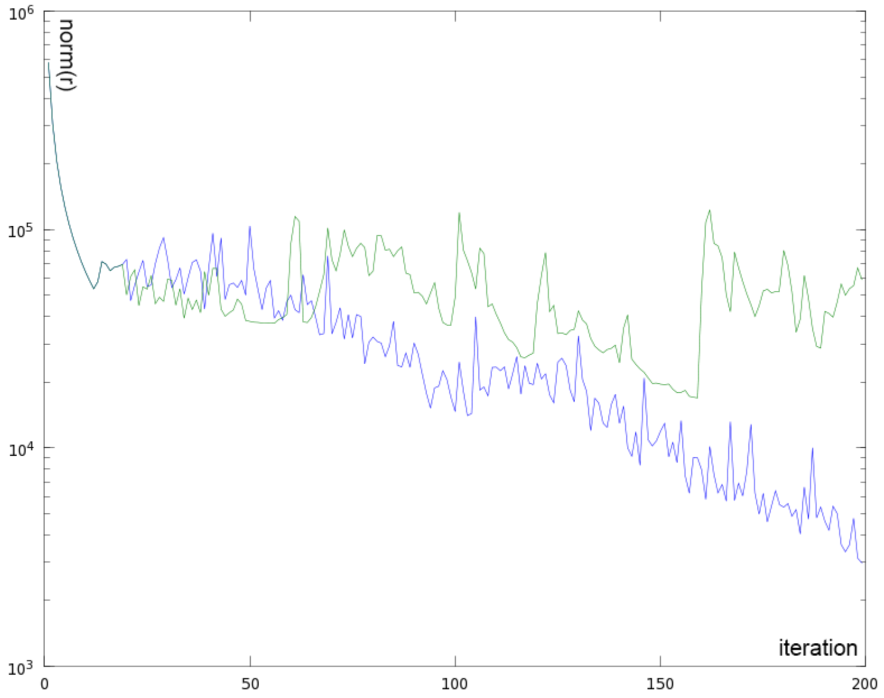

Figure 3.2: Residual plot for "nasa4704" matrix.

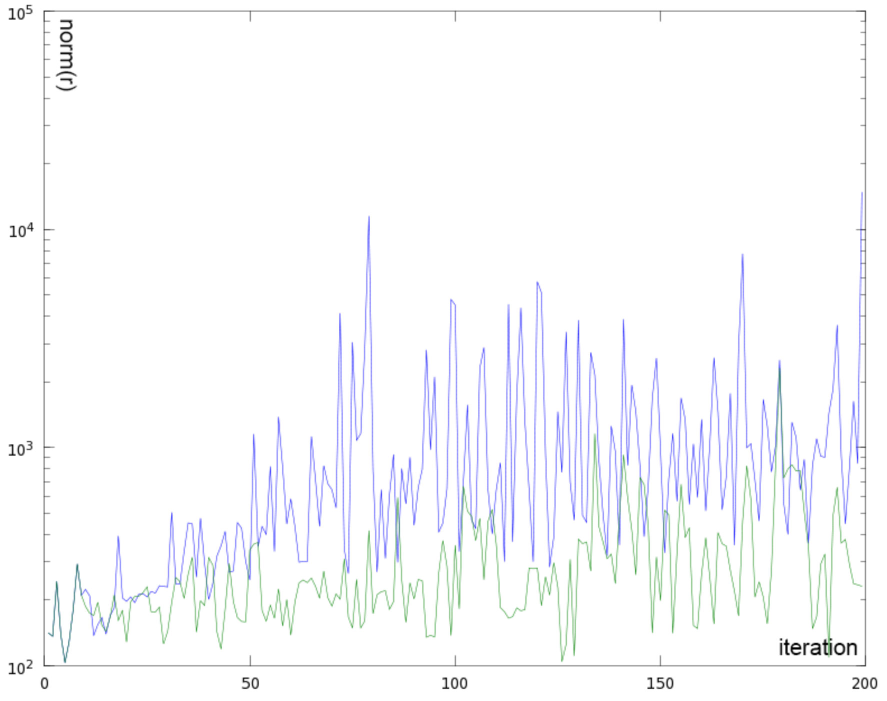

Figure 3.3: Residual plot for ”olafu” matrix.

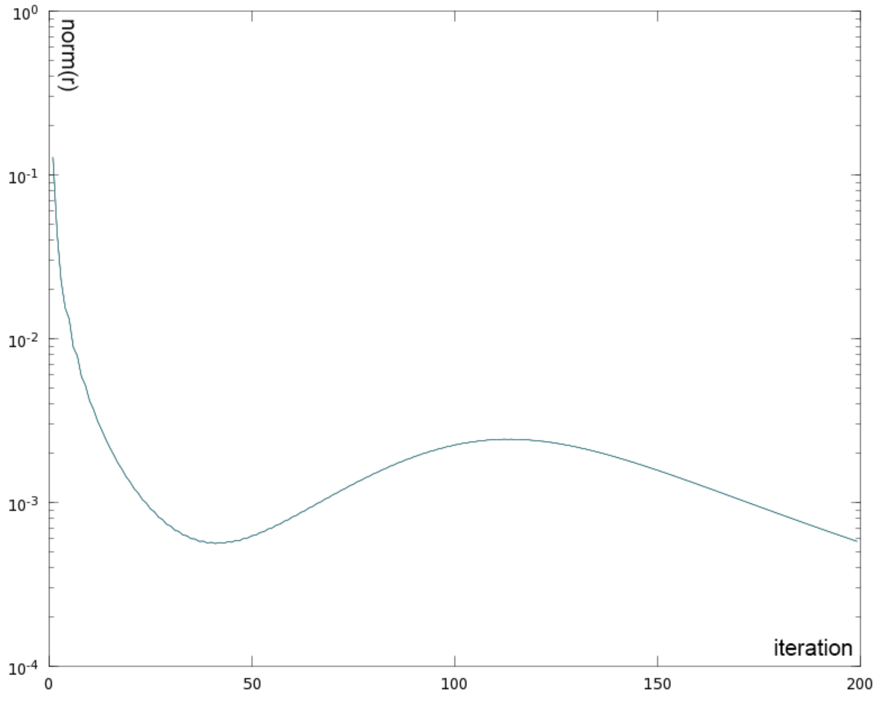

Figure 3.4: Residual plot for ”parabolic_fem” matrix.

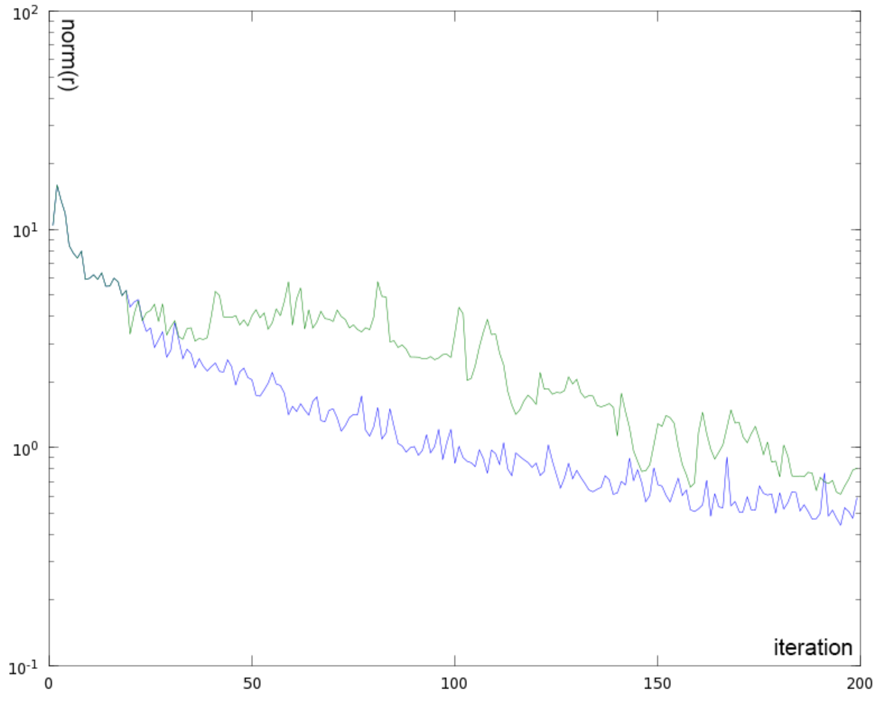

Figure 3.5: Residual plot for "smt" matrix.

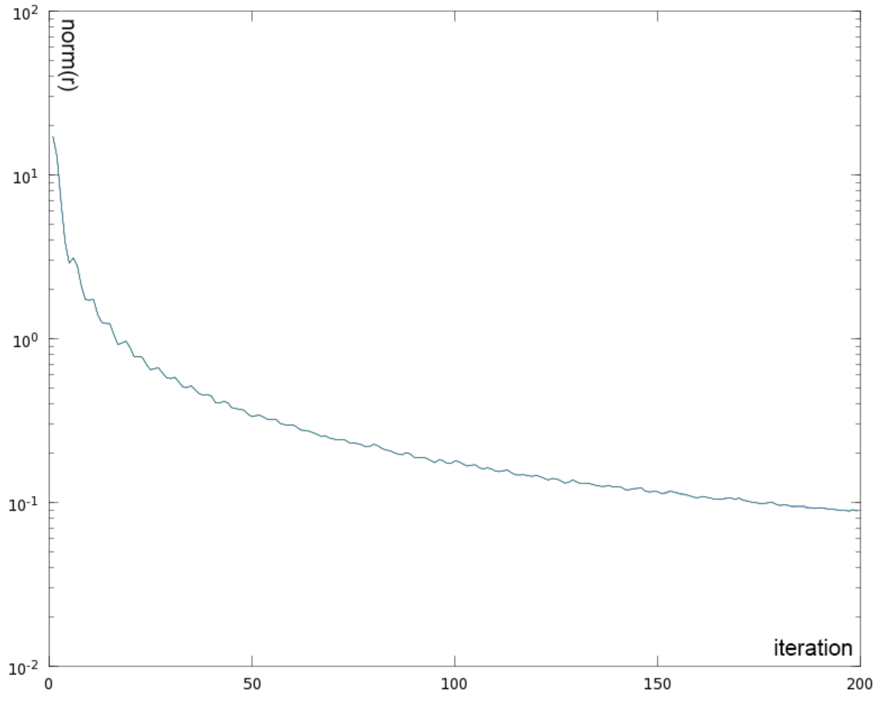

Figure 3.6: Residual plot for "thermal1" matrix.

actual norm of the residual after each iteration can be seen in figures 3.1 to 3.6, where the horizontal axis is the index of the iteration and the vertical axis is the norm. The decision about whether the algorithm converges or not was made by visually inspecting the slope of the graph - if the slope was negative, it was assumed that the estimate of the solution would eventually converge to the actual solution.

For some of the matrices, the modified algorithm outputs the exact same results as the original algorithm (matrices "parabolic_fem" and "thermal1"). However, for others some numerical instability causes the modified algorithm to explode and produce solution values outside the normal range (matrices "af_3_k101", "nasa4704", "olafu" and "smt"). The modified algorithm can be fixed by resetting the estimate of the residual and the search direction when the solution explodes by setting $residual = b - Ax, search\_direction = residual$ for the last known valid solution estimate x, but that makes a significant difference in the output of the modified algorithm compared to the original algorithm. The source of the numerical instability is not known right now and needs further investigation.

Matrix name	Size	Non-zero entries	Original	Modified
af_3_k101	503625×503625	17550675	converges	converges, $residual$ = NaN after 28 iterations, reset every 20 iterations
nasa4704	4704×4704	104756	converges	converges, $residual$ = NaN after 25 iterations, reset every 20 iterations
smt	25710×25710	3749582	converges	converges, $residual$ = NaN after 28 iterations, reset every 20 iterations
olafu	16146×16146	1015156	does not converge	does not converge, $residual$ = NaN after 22 iterations, reset every 10 iterations
parabolic_fem	525825×525825	3674625	converges	converges
thermal1	82654×82654	574458	converges	converges

Table 3.1: A description of a sample of matrices taken from the University of Florida matrix library and the results of running the modified Conjugate Gradient method on them.

Chapter 4

Soft Errors and Conjugate Gradient Method

As mentioned in the introductory chapter, sometimes computation can be corrupted by soft errors - one time events that do not halt the algorithm and need extra care to be detected. In this chapter we will try to modify the standard conjugate gradient method to fix the erroneous results caused by soft errors once they have been detected.

4.1 Errors as Perturbations to the Matrix

Iterative methods have the property that they might never find the solution in a finite number of steps but should get closer to the real solution at every step. That is why, when running an iterative method like the conjugate gradient method, we would want to stop after some number of iterations and check for soft errors. If no errors have occurred then we may continue. On the other hand, if there is at least one soft error in the computation of matrix-vector or inner vector products involved in the computation, we would like to be able to fix it in an efficient way as opposed to restarting the whole algorithm from the beginning or, if we took care of storing the intermediate results, a point where everything was still correct.

Suppose we have an equation $Ax = b$. We would like to construct a sequence of matrices $A_1, A_2, ..., A_n$ such that $A_i \to A$ as $i \to \infty$ and for every i, $A_i x_i = b_i$, where x_i is the computed solution at the i'th iteration of the algorithm and b_i is some $n \times 1$ vector that might differ at each iteration, but is also approaching b. Now, if one soft error occurs during the computation of the solution and we can detect it using the method described in chapter 1.3, we would like to be able to fix it. If we stopped after j iterations to check for errors, we have our current erroneous matrix (call it A_j^e) and erroneous approximation to the solution (call it x_j^e) that should satisfy $A_j^e x_j^e = b_j$. We would like to be able to find a rank-1 matrix $\delta A = uv^T$ such that $A_j^e + \delta A = A_j$. In that case we could find the actual solution at step j with the help of the Sherman-Morrison formula

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}. \tag{4.1}$$

Our goal is to find such a sequence of matrices $A_1, A_2, ...A_n$ that would result in an easy-to-compute rank-1 matrix update δA in case of one soft error.

4.2 A Sequence of Sums

Since $x_i = \alpha_1 d_1 + \alpha_2 d_2 + ... + \alpha_i d_i$ is a sum of scaled search vectors d_i and d_i is A-orthogonal to d_j (meaning that $d_i^T A d_j = 0$) for $i \neq j$, we could try decomposing $A \in \mathbb{R}^{n\times n}$ in $Ax = b$ into a sum of rank-1 matrices of the form $q_j(Ad_j)^T$ for some vector q_j so that

$$A = \sum_{j=1}^{n} q_j (Ad_j)^T. \tag{4.2}$$

Let us define A_i as the matrix constructed by taking only the first i terms of this sum in equation (4.2), specifically $A_i = \sum_{j=1}^{i} q_j(Ad_j)^T$. Then the product $A_i x_i$ takes the form:

$$\begin{aligned} (q_1(Ad_1)^T + q_2(Ad_2)^T + ... + q_i(Ad_i)^T)(\alpha_1 d_1 + \alpha_2 d_2 + ... + \alpha_i d_i) &= \\ = \alpha_1 q_1 (d_1, d_1)_A + \alpha_2 q_2 (d_2, d_2)_A + ... + \alpha_i q_i (d_i, d_i)_A &= b_i \end{aligned} \tag{4.3}$$

where $(d_j, d_j)_A = d_j^T A d_j$ for $x, y \in \mathbb{R}^{n\times 1}$. Vector q_i can be found by multiplying equation (4.2) from the right by d_i:

$$Ad_i = (\sum_{j=1}^{i} q_j (Ad_j)^T) d_i = q_i (d_i, d_i)_A \tag{4.4}$$

$$q_i = \frac{Ad_i}{(d_i, d_i)_A}. \tag{4.5}$$

A_i does converge to A for small values of n, but my experiments have shown that the sum diverges for n much greater than ten.

Suppose we stop and check for errors at iteration k. In case of an error occurring at iteration

$j, k \geq j$, the rank of δA, the matrix that fixes A_k, is $2 \times (k - j + 1)$ as the rank grows by two at each iteration for which we do not fix the error. The rank is growing because at each step the approximation A_k to the matrix A is not getting closer to the actual matrix A in addition to moving away from it. The computation of δA is highly involved and it is of much higher rank than we want for use with the Sherman-Morrison formula. A different option to try would be a product of matrices (a factorization) instead of a sum.

4.3 A Sequence of Products

Suppose we could construct a series of matrices $A_1, A_2, ..., A_i$ such that $A_i = \Delta A_i \cdots \Delta A_2 \Delta A_1$ and

$$A_i[x_1, x_2, ..., x_i] = [Ax_1, Ax_2, ..., Ax_i], 1 \leq i \leq n. \tag{4.6}$$

In this case one could take $\Delta A_i = I + q_i d_i^T$, where $q_i = \frac{Ad_i - A_i d_i}{d_i^T A_i d_i}$.

Unfortunately this decomposition into a product of matrices does not help with the computation of the rank-1 perturbation matrix in case of an error. The problem is that an error at step i affects not only the matrix ΔA_i, but also $\Delta A_{i+1}, \Delta A_{i+2}, ...$ and so on.

Another approach we can try is computing the rank-1 perturbation ΔA to matrix A as $\Delta A = uv^T$ where $u = \alpha e^j, v = \frac{d_i^T A}{d_i^T A d_i}$, j is the location of the error in the vector $w_i = Ad_i$ that is computed in the conjugate gradient method (lines 8 and 10 in Algorithm 1), α is the magnitude of the error, d_i is the search vector in the Algorithm 1 at step i and e^j is the unit vector with value one in location j and zero everywhere else. The erroneous vector is then equal to $w_i^e = w_i + \alpha e^j = (A + \Delta A)d_i$.

The problem with this approach is that it requires all the vectors d_i to be A-orthogonal to each other. However, once an error is introduced at step k, all the subsequent vectors d_{k+1}, d_{k+2}... are not A-orthogonal to the first $i-2$ vectors any more.
An attempt has been made to apply this solution to the GMRES method (see [8] p.548), but this factorization works only when the search vectors are all A-orthogonal to each other. The orthogonality property $d_i^T d_j = 0$ when $i \neq j$ does not seem to be sufficient.

4.4 Key Problems

There are a couple of reasons why the ideas presented do not seem to work in practice. I will list some of them:

1. Once a soft error occurs, $A + \Delta A$ may not be positive definite any more even if A is and this is a necessary property for the conjugate gradient method to work properly. This is not a problem for methods that work with indefinite matrices such as GMRES, but attempts to apply these ideas to those methods have failed as well.

2. If you want to use the Sherman-Morrison formula to fix the error, you still have to compute $A^{-1}u$, where $u = \alpha e^j$. This is usually no easier than computing the actual solution to the main problem $Ax = b$.

3. The condition number of $A+\Delta A$ becomes very high even for errors of small magnitude.

But the biggest issue, in my opinion, is that the error e_i in the solution is, in general, not in the Krylov subspace of A at iteration i and can not be expressed using the search vectors

$d_1, d_2, ..., d_i$ that have already been computed.

Chapter 5

A Monte Carlo Method

As computers become faster, but at the same time less reliable, entirely different classes of algorithms might become useful for solving large scale linear algebra problems. In this chapter we are going to discuss a stochastic algorithm that was inspired by particle filters, also known as the sequential Monte Carlo method. The algorithm attempts to guess what the solution might be and then refines the guesses until it gets close enough to the actual solution.

5.1 Particle Filters

Particle filtering [11],[3] is a Monte Carlo method for performing statistical inference in models where the state of the system is evolving with time and information about the state is collected via noisy measurements made at each time step. However, in our case the state of the system is constant and we will simply perform noisy measurements to estimate the state vector, the true location of the solution to the linear system of equations. We call the measurements, or guesses, particles and the act of estimating the state vector - filtering.

5.2 The Algorithm

The algorithm will proceed as follows:

1. First we make a set $S_1 = \{x_1^i\}$ of N random guesses at the solution to the system of equations.

2. Afterwards, we compute the residual $r_1^i = b - Ax_1^i$ for each guess x_1^i.

3. The third step is to make a set $T = \{x_2^i\}$ of N new guesses by sampling from the set S_1 without replacement. The probability $P(x_1^i)$ of picking x_1^i is proportional to some function $g(x)$ of the magnitude of the residual, that is $P(x_1^i) = \frac{g(||r_1^i||)}{\sum\limits_{j=1}^{N} g(||r_1^j||)}$. We would like to sample the guesses with smaller residuals more often so we can pick $g(x) = e^{-\beta x}$ for some scalar β. In the experiment below, $\beta = 10000$.

4. The final step is to apply a small perturbation ϵ_2^i to each $x_2^i \in T$ to form a new set $S_2 = \{x_2^i + \epsilon_2^i : x_2^i \in T\}$. This allows us to search for better solutions in the vicinity of the current estimate x_2^i.

If x_{exact} is the solution to $Ax_{exact} = b$, then the overall error $E_2 = \sum\limits_{i=1}^{N} ||x_{exact} - x_2^i||$ of the guesses in S_2 should be smaller than of guesses in S_1 and the best solution in S_2 should be closer to the actual solution than the best solution in S_1. During the next iteration of the algorithm if we use S_2 in place of S_1, we should be able to compute an even more accurate

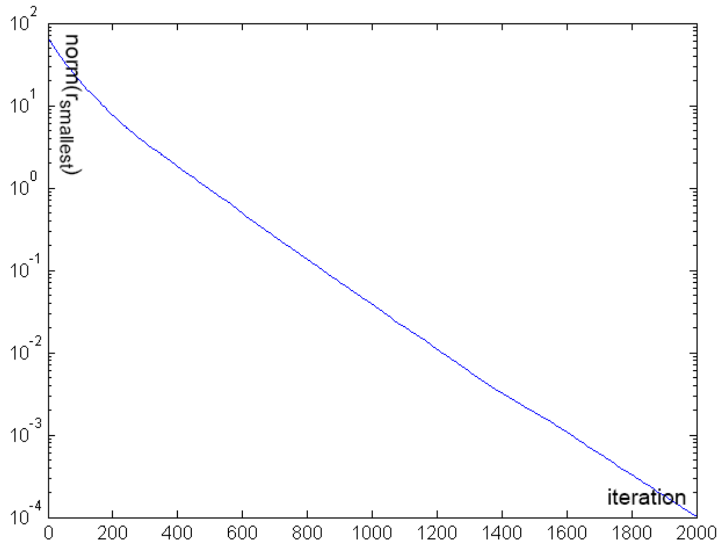

Figure 5.1: Logarithmic plot of the norm of the smallest residuals at every iteration

solution. We can use the magnitude of the smallest residual to figure out when to stop the algorithm, just as with the conjugate gradient method discussed in Chapter 3.

5.3 Results

The test was performed using a matrix $A = \frac{1}{10}B + I \in \mathbb{R}^{200\times 200}$ which is a sum of a matrix $B \in \mathbb{R}^{200\times 200}$ with uniformly random elements in the range [0;1) and the 200×200 identity matrix I. A logarithmic plot of the smallest norm of the residuals after each iteration can be seen in Figure 5.1. The magnitude of the smallest residual is steadily decreasing and this implies that we are computing more and more accurate solutions. Code used for the experiment can be found in the Appendix C.

5.4 Further Work

Careful analysis of the algorithm has not been carried out yet. An investigation into the convergence bounds similar to the one presented in Chapter 3.1.1 on the conjugate gradient method would be useful. Also, it is not clear what an optimal function $g(x)$ of the norm of the residual in step 3 of the algorithm could be.

The algorithm could be potentially useful in cases where a lot of errors occur during execution time and some form of fault-tolerance is required as it does not matter if some of the solutions have large errors in them because they will be discarded during the re-sampling step and replaced with more accurate ones. Also, each sample is independent of every other sample so a lot of operations could be performed in parallel per sample.

Chapter 6

Summary

Even though the modification of the hyperpower method presented in Chapter 2 does not require any matrix-matrix multiplications to find the solution as opposed to the original algorithm of finding the inverse of the matrix first, it suffers from cancellation error and is not usable for large matrices of moderate rank.

The modification of the original Conjugate Gradient method in Chapter 3 reduces some synchronization and allows the algorithm to run in a more parallel fashion arriving at an answer quicker on multi-core machines. However, it does fail sometimes as shown during the experiments.

Recovering from soft errors as formulated in Chapter 4 should allow for faster recovery. However, I was not able to build a proper procedure for recovery because of some serious theoretical problems. Also, simply restoring the last known good solution and restarting the standard algorithm from that point is a simple solution that is hard to improve on.

The Monte Carlo algorithm presented in Chapter 5 does allow easy recovery from soft errors, but is not very efficient considering how many operations it performs for a set level of solution accuracy. However, it might be practical for processors with very high rates of soft errors.

6.1 Suggestions for Further Work

Fault-tolerance for iterative methods for solving large and sparse linear systems of equations is still an active topic of research. At the moment the best way to recover from a soft error after detecting it is restarting from a checkpoint. It would be beneficial to have methods that are able to recover from one or more soft errors without throwing away all the results computed after a checkpoint.

Stochastic Monte Carlo methods such as the one presented in Chapter 5 could be fairly resilient to soft errors because we can treat these errors as just another source of randomness, but converge slower than the more popular algorithms such as the conjugate gradient method and are more computationally intensive. Convergence bounds for the method in chapter 5 need to be established and there are a lot of parameters that need investigating and tuning.

Bibliography

[1] *754-2008 - IEEE Standard for Floating-Point Arithmetic*. Aug. 29, 2008. ISBN: 978-0-7381-5752-8.

[2] M. Altman. “An optimum cubically convergent iterative method of inverting a linear bounded operator in Hilbert space”. In: *Pacific Journal of Mathematics* 10 (4 1960), pp. 1107–1113.

[3] S. Arulampalam et al. “A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking”. In: *IEEE Transactions on Signal Processing* 50 (2 2002), pp. 174–188.

[4] Tim Davis and Yifan Hu. *The University of Florida Sparse Matrix Collection.* `http://www.cise.ufl.edu/research/sparse/matrices/`. 2014.

[5] Peng Du, Piotr Luszczek, and Jack Dongarra. *High Performance Linear System Solver with Resilience to Multiple Soft Errors.* `http://www.netlib.org/lapack/lawnspdf/lawn256.pdf`. 2011.

[6] Howard C. Elman, David J. Silvester, and Andrew J. Wathen. *Finite Elements and Fast Iterative Solvers with Applications in Incompressible Fluid Dynamics*. Oxford Univeristy Press, 2005. ISBN: 019852868X.

[7] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". In: *Computing Surveys* (March, 1991).

[8] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Edition)*. Johns Hopkins University Press, 1996. ISBN: 0801854148.

[9] C. L. Lawson et al. "Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Transactions on Mathematical Software (TOMS)* 5 (3 1979), pp. 308–323.

[10] Hans Meuer et al. *Top500 Supercomputing Sites*. `http://www.top500.org/`.

[11] Emin Orhan. *Particle Filtering*. `http://www.cns.nyu.edu/~eorhan/notes/particle-filtering.pdf`. 2012.

[12] M. Picasso. "A stopping criterion for the conjugate gradient algorithm in the framework of anisotropic adaptive finite elements". In: *Communications in Numerical Methods in Engineering* 25 (4 2009), pp. 339–355.

[13] David Pool. *Linear Algebra: A Modern Introduction (Second Edition)*. Thomson Brooks/Cole, 2005. ISBN: 0534405967.

[14] Theodore J. Rivlin. *Chebyshev Polynomials: From Approximation Theory to Algebra and Number Theory*. Wiley-Interscience, 1990. ISBN: 0471628964.

[15] Jonathan Richard Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.* `http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`. 1994.

[16] V. K. Stefanidis and K. G. Margaritis. *Algorithm Based Fault Tolerant Matrix Operations for Parallel and Distributed Systems: Block Checksum Methods.* `http://www.academia.edu/1350674`. 2003.

Appendices

Appendix A

Code for the Number of Terms

Code for GNU Octave to compute the number of terms to solve $Ax = b$:

```
n = 200;            # matrix size n x n
p = 16; # aim for less than 2^-p of absolute error
A = rand(n)*0.005+eye(n);   # The Matrix
b = rand(n,1);

m = eigs(A*A',1,'sm') # smallest eigenvalue
M = eigs(A*A',1,'la') # largest eigenvalue

max_k = 2000; # maximum number of terms that should be added

stepfrac = 400;
steps = 100;
stepsize = 2.0/(M*stepfrac);

normAtA = norm(A'*A,inf);
normAtb = norm(A'*b,inf);
normb = norm(b);

min_it = inf;
min_alpha = inf;
min_terms = inf;

for i=1:steps,# try different alphas
```

```
    realalpha = i*stepsize;
    normT(i) = 1.0 - m*realalpha;
    iter(i) = realalpha;

    for j=1:20,# try different X_j
        prec(j) = j;
        value(i,j) = -inf;

        # can this iteration achieve enough precision? norm(T)^(2^j) > 2^(-p)
        if j < log2((log2(m/normb)-p)/log2(normT(i))), continue; end

        m_t = ceil(log2(realalpha*normAtA)); # bound X*A
        p_t = ceil(log2(realalpha*normAtb)); # bound X*b

        printed_already = 0;
        sum = 0;
        for k=1:2^j,# sum some terms
            if k > max_k, break; end

            # have we found a small enough term already?
            if printed_already == 1, break; end

            # tighter sum (bound) than in the article
            sum += log2((2^j-k+1)/k);
            term_k = sum + m_t*(k-1) + m_t + p_t;

            if term_k <= -p,
                printed_already = 1;
                value(i,j) = k;
                if k <= min_terms,# is this the best solution so far globally?
                    min_terms = k;
                    min_alpha = realalpha;
                    min_it = j;
                end
            end
        end
    end
end

fprintf("Best results for alpha = %e and %d iterations: %d terms\n",
                                          min_alpha, min_it, min_terms);
surf(prec,iter,value)
```

Appendix B

Modified Conjugate Gradient Method

The first part of the loop is just A-orthogonalization of the next two search directions.

An attempt has been made to make the code as readable as possible at the cost of computing some quantities a couple of times.

```
% input: A, b; result: x;
x = b;
k = 1;
r = b - A*x;
p = r;

while k < length(b)+1,
   Ap = A*p;
   % orthogonalize the next search direciton
   q = Ap - ((p'*A*Ap)/(p'*Ap))*p;
   if k > 1
      q = q - ((prev_p'*A*Ap)/(prev_p'*A*prev_p))*prev_p;
   end

   % generate a new search direction used in the next iteration
   next_p = A*q;
   next_p = next_p - ((q'*A*next_p)/(q'*A*q))*q;
   next_p = next_p - ((p'*A*next_p)/(p'*A*p))*p;
   % any r could be used theoretically, but it has to be updated once
   % in a while in practise because of finite precision arithmetic
   % and search directions losing orthogonality
   alpha_1 = (p'*r)/(p'*Ap);
   alpha_2 = (q'*r)/(q'*A*q);

   x = x + alpha_1*p;
   x = x + alpha_2*q;

   % could do without updating r if the search directions would not lose
   % orthogonality, but that is not the case as the number of iterations
   % increases
   r = r - alpha_1*Ap;
   r = r - alpha_2*A*q;

   if norm(r) < 1.0e-13, break, end

   prev_p = q;
   p = next_p;
   k = k + 1;
end
```

Appendix C

Code for the Monte Carlo Method

The algorithm assumes that variables A, b and n have already been initialized.

```
maxSamples = floor(2*n);
samples = (ones(n,maxSamples)-2.0*rand(n,maxSamples))*norm(b);

k = 1;
while k < 2000,
   res = b*ones(1,maxSamples) - A*samples;
   for i=(1:maxSamples)
       r(i) = norm(res(:,i));
   end

   sumR = sum(r);
   w = r/sumR;
   for i=(1:maxSamples),
       w(i) = exp(-w(i)*100000.0);
   end

   choices = randsample(maxSamples,maxSamples,true,w);

   for i=(1:maxSamples)
       guess = (ones(n,1)-2.0*rand(n,1));
       guess = guess/norm(guess);
       newSamples(:,i) = samples(:,choices(i)) + guess*r(choices(i))*0.1;
   end
   samples = newSamples;

   k = k + 1;
end
```