On the computation of probabilities and eigenvalues for random and non-random matrices

by

Vignesh Peruvamba Sundaresh

Submitted to the The School of Engineering
in partial fulfillment of the requirements for the degree of

Masters of Science in Computation for Design and Optimization

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Jun 2009

Author . . . .
The School of Engineering
May 8, 2009

Certified by . . . .
Alan Edelman
Professor of Applied Mathematics
Thesis Supervisor

Accepted by . . . .
Jaime Peraire
Professor of Aeronautics and Astronautics
Director, Computation for Design and Optimization Program

On the computation of probabilities and eigenvalues for random and non-random matrices

by

Vignesh Peruvamba Sundaresh

Submitted to the The School of Engineering
on May 8, 2009, in partial fulfillment of the
requirements for the degree of
Masters of Science in Computation for Design and Optimization

Abstract

Can you imagine doing hundreds of millions of operations on non-integers and not obtaining a single round-off error? For $n < 12$, the algorithm used in this thesis does exactly that. We took advantage of a floating point property that we have not seen used before. If only we had quad precision we could have gone even further and extended the algorithm without round-off error for higher values of 'n'.

The problem in question concerns whether the eigenvalues are real or complex. The eigenvalues of an n-by-n real random matrix whose elements are independent random variables with standard normal are examined. An exact expression to determine the probability $p_{n,k}$ that exactly k eigenvalues are real are derived in [1]. This expression was used to compute the probabilities $p_{n,k}$, but the computation was achieved only up to $n = 9$. For higher values of n, the symbolic expressions generated during the course of an algorithm to compute an exact probability as expressed in Mathematica code requires large amounts of memory. In this thesis, we target development of a more efficient algorithm. The symbolic algorithm implemented in Mathematica is converted into an equivalent numerical version and is implemented using MATLAB. After implementing the serial code in MATLAB, the code is parallelized using a client-server parallel computing platform named Star-p. This modified code implementation along with superior hardware in terms of better processor speeds and larger memory, has enabled the probability evaluation for all values of k up to n=11, and for certain k values for $n = 12$ and 13. An expression for the expected number of real eigenvalues $\sum_{k=0}^{n} kp_{n,k}$ is obtained in paper [2]. Results relating the rational and irrational parts of the summations $\sum_{k=0}^{n} kp_{n,k}$, $\sum_{k=0}^{n} \binom{k}{2} p_{n,k}$ and $\sum_{k=0}^{n} \binom{k}{3} p_{n,k}$ are conjectured.

Three eigenvalue algorithms, the block Davidson, the block KrylovSchur and the Locally optimal Block Pre-conditioned Conjugate Gradient Method (LOBPCG) are analyzed and their performance on different types of matrices are studied. The performance of the algorithms as a function of the parameters , block size, number of blocks and the type of preconditioner is also examined in this thesis. The block Krylov Schur Algorithm for the matrices which are used for the experiments have proved to much superior to the others in terms of computation time. Also its been more efficient in finding eigenvalues for matrices representing grids with Neumann boundary conditions which have at least one zero eigenvalue. There exists one optimal combination of block size and number of blocks at which the time for eigenvalue computation is minimum. These parameters have different effects for different cases. The block Davidson algorithm has also been incorporated with the locking mechanism and this implementation is found to be much superior to its counterpart without the locking mechanism for matrices which have at least one zero eigenvalue.

Thesis Supervisor: Alan Edelman
Title: Professor of Applied Mathematics

Acknowledgments

I am deeply indebted to my advisor, Professor Alan Edelman for his constant support. Without his help, this work would not be possible. I would like to sincerely thank my co-advisor Sivan Toledo for his great help. I would like to thank Thornquist Heidi, Lead Developer of Anasazi, Chris Baker,Developer of Anasazi and Ulrich Hetmaniuk Developer of Anasazi for their tremendous technical support for Trilinos, a software package which was very new to me. I would like to thank Alan Edelman's student Yeelok Wong and Sudarshan Raghunathan at Interactive Supercomputing for their help with Star-p.

I would like to thank Jonathan Kelner, Assistant Professor, Department of Mathematics for letting me use his personal computer to run my code. I would like to thank my Academic Administrator Laura Koller for her assistance in aiding me at various junctures of my Master's program and her frequent important emails regarding various deadlines. I would also like to thank Jaime Peraire, the Director and Robert Freund the Ex Director of the Computation for Design and Optimization (CDO) program.

I would like to thank my friends Dipanjan Sen, Vivek Inder Sharma, Srikanth Patala and Sukant Mittal for their help. I also would like to thank my friend Aravindan Vijayaraghavan, Princeton University who has provided me with a lot of help with linux, latex and with my C++ codes. I would like to thank all the members of my family for their guidance and support and enabling me to join this great institute.

Chapter 1

Introduction and motivation

Let A be an n-by-n real random matrix whose elements are independent random variables with standard normal distributions. The eigenvalues of the matrix for finite 'n' was examined and an exact expression to determine the probability $p_{n,k}$ that exactly k eigenvalues are real was derived in [1]. This expression was used to compute the probabilities $p_{n,k}$, but however the computation was achieved only for up to $n = 9$. The probability distribution of the real eigenvalues is derived by first factoring the matrix into some form of the Real Schur Decomposition, then interpreting this decomposition as a change of variables and finally performing a wedge product derivation of the Jacobian of this change of variables. This derivation is explained in detail in [1]. The algorithm to implement the formula in [1] to calculate probabilities involves the generation of a large number of symbolic expressions. The size of these expressions are a function of 'n'(matrix size) and they get very large during the course of the algorithm. Hence for the evaluation of probabilities for larger 'n' greater memory is required. A code devoid of symbolic variables is expected to be superior, both memory and computation wise over one with large number of symbolic expressions. Hence in an attempt to improve the performance, the code was converted to a purely numerical one without the use of symbols.

With improved processors and the ability to parallelize codes, greater computational power is available. A symbolic algorithm previously implemented in Mathematica is implemented numerically in MATLAB with no round-off error up to n=11. The calculation for the $n = 11$ and $k = 1$ case which was evaluated using the MATLAB code, involves about fifty million floating point operations. Since all the fractions are dyadic rationals (rational numbers with power of 2 in the denominator), this computation was possible. We are not aware of any other computations that have taken advantage of this special property. In the presence of quad precision, the computation which at this stage has been achieved only for up to $n < 12$ without round-off error can be extended to higher values of 'n'.

Further, the MATLAB code is parallelized using a client-server parallel computing platform, Star-p which is designed to work with very high level languages such as MATLAB and Python. The first chapter is organized as follows. The main results of the thesis are presented in the very first section of the thesis. The next section discusses how the algorithm has been implemented numerically using MATLAB. The scaling of the algorithm as a function of the matrix size 'n' is also explained. All the probability values for up to n=11 and certain values of k for $n = 12$ and 13 is presented in a table. The next section deals with the comparison of the different codes used to obtain the probabilities. We then present conjectures involving the summations $\sum_{k=0}^{n} \binom{k}{2} p_{n,k}$ and $\sum_{k=0}^{n} \binom{k}{3} p_{n,k}$ and an expression for the asymptotic variance of the real number of eigenvalues.

The chapter titled "Eigenvalue Algorithms for sparse symmetric matrices" pertains to eigensolver algorithms for non-random matrices. In linear algebra, one of the most important problems is designing efficient and stable algorithms for finding the eigenvalues of a matrix. These eigenvalue algorithms may also find eigenvectors. In this thesis, three important eigensolver algorithms the block Davidson, block KrylovSchur and the Locally optimal Block Pre-conditioned Conjugate Gradient Method (LOBPCG)are studied and their performance on different matrices is analyzed. All of these algorithms are implemented with the help of the Trilinos project. These algorithms are explained in detail in [10].

The Trilinos Project is an effort to develop and implement robust algorithms and enabling technologies using modern object-oriented software design, while still leveraging the value of established libraries such as SuperLU, the BLAS and LAPACK.

Different parameters associated with these algorithms can be changed and their effect on the algorithm performance is analyzed. All are used with preconditioners in order to improve their performance. The preconditioner ‘P’ of a matrix ‘A’ is a matrix such that P^{-1}A has a smaller condition number than A. Preconditioners are useful especially when we are using iterative methods to solve a large sparse linear system. The most efficient preconditioner is indeed the matrix itself since it reduces the condition number to one and it enables speedy computation of the eigenvalues. All the matrices used in this thesis are extremely sparse and the use of preconditioners improve the performance of the algorithm significantly. The preconditioners used in this thesis are of two types- either an accurate or an approximate representation of the matrix inverse, both of which have their own advantages. While it takes more time to build the accurate matrix inverse using the Amesos-Klu method rather than the approximate inverse using the Multi-Level Method, solving for the eigenvalues takes lesser time using the accurate matrix inverse.

For certain combinations of iterative methods and linear systems, the error at each iteration projected onto the eigenfunctions has components that decay at a rate proportional to the corresponding eigenvalue. Multilevel methods exploit this property by projecting the linear system onto a hierarchy of increasingly coarsened “meshes” so that each error component decreases on at least one coarse “mesh”. The multi-grid methods used for all our results is the Multi-grid V cycle. Though other type of cycles do exist, we found that this method was the most efficient for our set of matrices. The approximate matrix inverse is calculated using this method. The exact matrix inverse is calculated through the Amesos-Klu method. Detailed explanation about this method can be found on [13]. Calculation of the exact inverse is of order N^3 complexity where N is the size of the dense matrix. In this thesis, all the matrices which we are using for the numerical simulation are sparse in nature.

The block size and the number of blocks are other parameters which can also be varied and their effects on the performance are studied. The Algorithms are implemented for the solution of the generalized eigenvalue problem Ax $=\lambda$Bx where A and B are large sparse symmetric matrices. All the Algorithms are implemented in C++.

Table 1.1: Types of Preconditioners

Multi-grid Method	ML Preconditioner
Amesos Klu Method	INV Preconditioner

Also the algorithms can be implemented with or without locking mechanisms. Since we are dealing with the computation of multiple eigenvalues, a locking mechanism will indeed serve to be useful. Let us consider a problem of computing a number of eigenvectors of a symmetric matrix by simultaneous iterations. When computing several eigenvectors simultaneously it is often observed that some eigenvectors converge faster than the others. To avoid unnecessary computational work, it is common to "lock" the eigenvectors that have already converged within a required tolerance while continue iterating other eigenvectors. The chapter begins by introducing the various matrices used for the experiments. After which the different algorithms and their corresponding results are explained. We finish the chapter comparing the various algorithms and under what conditions each of them are superior to the other.

Block methods are more efficient for multiple or clustered eigenvalues. Moreover, a block method is the natural choice when more than one good initial vector is available. This situation is very common for the self-consistent-loop in electronic structure calculations where one obtains several good initial guesses from former loops. One other advantage of a block method over a non-block method is better utilization of cache and better memory performance.

The eigensolver package in Trilinos is called Anasazi. Anasazi is a framework for developing large-scale iterative eigensolvers. When developing such a framework, or even a collection of different solvers, one will notice a large amount of overlap among different implementations. For example, two separate eigensolvers may utilize the same orthogonalization methods, whereas two different implementations of a particular eigensolver may utilize different orthogonalization routines. Dividing the different tasks associated with an iterative eigensolver into multiple routines enables increased code reuse and eases code maintenance. Consider the block Davidson iteration. The essence of the algorithm can be distilled as follows: a preconditioner is applied to the current residual vectors, and the resulting vectors are used to expand a subspace from which the next eigenvector approximations (and their residuals) are computed. However, a multitude of choices abound in implementing this simple iteration: the size of the computed basis, whether and how the method is restarted, the convergence criterion for the algorithm, and the use of deflation (locking) mechanisms, etc. [12].

Chapter 2

On the computation of probabilities associated with real eigenvalues of random matrices

2.1 Main Results

For an n-by-n real random matrix, $p_{n,k}$ represents the the probability that exactly k of the eigenvalues are real. In [1], $p_{n,k}$ is evaluated only for up to $n = 9$. The additional probability values for n = 10,11 and certain values for n = 12 and n = 13 are presented in this section. Two conjectures one relating the summations in equations 2.1 and other pertaining to the variance of the number of real eigenvalues of the random matrix are in 2.2, 2.3, 2.4 and 2.5.

Also the memory requirements of the mathematica code in [1]have been analyzed and an estimate of how much memory is required for further calculations is known. A new floating point MATLAB code to compute $p_{n,k}$ that can perform hundreds of millions of operations without round-off error is presented in Section 2.2.2.

2.1.1 Conjectures

On observing the summation values in Table 2.9, the result is true for n=1 to 10. However for n=11, the first conjecture relating the rational part is only approximately true. At this stage we aren't certain if n=11 is an aberration or if the result is untrue for higher values of 'n'. Let the set of variables involved the conjecture be defined by the set of the equations in 2.1. For n even, k is the set of all even numbers lesser than 'n' and similarly for n odd. r_j^n and s_j^n are dyadic rationals for all 'n'.

$$r_j^{(n)} + s_j^{(n)}\sqrt{2} = \sum_{k=0}^{n} \binom{k}{j} p_{n,k} \tag{2.1}$$

We conjecture that for n even,

$$r_2^{(2n)} = r_3^{(2n)} \ and \ s1^{(2n)} = 2 * s_2^{(2n)} \tag{2.2}$$

While for n odd

$$s_1^{(2n-1)} = \frac{2}{3} s_2^{(2n-1)} \tag{2.3}$$

and for certain n odd

$$r_2^{(2n-1)} = \frac{1}{2} r_3^{(2n-1)} \tag{2.4}$$

2.1.2 Variance of the number of real eigenvalues

If σ_n^2 denotes the variance of the number of real eigenvalues of an n-by-n random matrix, then

$$\lim_{n\to\infty} \frac{\sigma_n^2}{\sqrt{n}} = \sqrt{\frac{1}{2}} \tag{2.5}$$

Table 2.1: Results for n=10, 11 and 12

n	k	$p_{n,k}$	
10	10	$\frac{1}{2^{28}}\sqrt{2}$	0.00000
	8	$\frac{236539}{2^{29}} - \frac{5}{2^{23}}\sqrt{2}$	0.00044
	6	$\frac{-236539}{2^{27}} + \frac{35098479}{2^{30}}\sqrt{2}$	0.04447
	4	$\frac{149206217}{2^{28}} - \frac{105292877}{2^{30}}\sqrt{2}$	0.41712
	2	$\frac{-148733139}{2^{27}} + \frac{1216831949}{2^{30}}\sqrt{2}$	0.49453
	0	$\frac{834100651}{2^{29}} - \frac{1146637039}{2^{30}}\sqrt{2}$	0.04341
11	11	$\frac{1}{2^{28}}\sqrt{2}$	0.00000
	9	$\frac{333213}{2^{33}} + \frac{-160}{2^{33}}\sqrt{2}$	0.00004
	7	$\frac{-20823}{2^{27}} + \frac{60262315}{2^{33}}\sqrt{2}$	0.00977
	5	$\frac{510394091}{2^{31}} + \frac{-180786305}{2^{33}}\sqrt{2}$	0.20791
	3	$\frac{-1020121891}{2^{31}} + \frac{6423679969}{2^{33}}\sqrt{2}$	0.58254
	1	$\frac{10629845251}{2^{33}} + \frac{-6303155851}{2^{33}}\sqrt{2}$	0.19975
12	12	$\frac{1}{2^{33}}$	0.00000
	10	$\frac{-3}{2^{32}} + \frac{3781485}{2^{41}}\sqrt{2}$	0.00000
[0]	8	$\frac{27511372605}{2^{44}} - \frac{18907425}{2^{41}}\sqrt{2}$	0.00155
[0]	6	$\frac{-27511352125}{2^{42}} + \frac{126455775487}{2^{41}}\sqrt{2}$	0.07507
[0]	4	$\frac{6237846960567}{2^{43}} - \frac{379291696761}{2^{41}}\sqrt{2}$	0.46523
	2	$\frac{-6182824264509}{2^{42}} + \frac{356179603371}{2^{38}}\sqrt{2}$	0.42669
	0	$\frac{29930323227453}{2^{44}} + \frac{-1298292889877}{2^{40}}\sqrt{2}$	0.03145

2.2 Codes used for Probability Evaluation

2.2.1 Mathematica Code

An original Mathematica code to implement the above algorithm is shown in [1]. The probability $p_{n,k}$ is always of the form r+s$\sqrt{2}$ where r and s are rational numbers. Depending on the values of n and k , different polynomials are generated during the course of the algorithm and after taking certain moments we obtain the probability. These results and the derivation of the expression to evaluate the probability is presented in [1].

Table 2.2 examines the the memory requirement of the mathematica code as a

[0]Results for k=4,6 and 8 may not be exact because they have been obtained using conjecture 2.2

function of ‘n’ and ‘k’. For large values of ’n’ the code was run on a machine with 128GB of memory. However even this large RAM size was insufficient to compute certain probabilities for n=11 and 12. The scaling of memory requirements gives an indication of how much memory is required by the mathematica code for larger values of ‘n’. The case n=10 and k=6 requires over 10GB of memory and considering the scaling across consecutive even numbers, 128 GB of RAM is not sufficient for the n=12 and k=6 case. The next $p_{n,k}$ which can be computed with a memory enhancement of about 100GB is for $n = 12$ and $k = 4$ case. For a fixed n, when $k = n$, $p_{n,k} = 2^{-n(n-1)/4}$. This result is presented in [1]. Hence the calculation of $p_{n,n}$ is not required. When $k = n - 2$, the computation takes up much lesser memory rather than when k is close to $\frac{n}{2}$.

2.2.2 Matlab Code-Algorithm

We implemented an algorithm that begins with the generation of a polynomial resulting from the expansion of a determinant. It is this polynomial that will be referred to as the starting polynomial in all further discussions. The determinant is both a function of the integers n (size of the Matrix) and k (number of real eigenvalues), and hence so is the starting polynomial. The starting polynomial equals "1" for the k=0 case. Numerical computation is computationally more efficient and achieves memory savings in comparison to symbolic computation. This was the primary motivation to convert the mathematica code to an equivalent MATLAB code without the use of symbols. As explained before, limited memory was the primary deterrent in the computation of probabilities for larger sized matrices. During the course of the algorithm, the starting polynomial is multiplied with many other polynomials resulting in the generation of larger sized polynomials.

In order to keep the MATLAB code symbol free, the starting polynomial and all other polynomials to be multiplied during the course of the algorithm are transformed to an equivalent numerical form. This transformation is explained below through a simple example. For a given polynomial, let the number of monomials be represented by the integer p and let the number of distinct variables the polynomial is a function of, be indicated by the integer q. Then the resulting matrix which will be representative of the above polynomial will be of size p-by-q. The entries of the matrix are the exponents of the variables in the polynomial. For the k=0 and n=even case the variables which occur in the polynomial are $x_1, y_1,$ ..., $x_{n/2}, y_{n/2}$. There are many more variables involved when k is non-zero and when n is an odd integer.

A simple example is as follows. $p(x_1, x_2, y_1, y_2) = x_1^2 y_1^2 + x_2^3 + 2x_1^3 y_2$. The matrix representation for the above polynomial is shown in Table 2.3

Since the total number of distinct variables involved is four, the width of the matrix is 4 and number of monomials equals the length of the matrix. The coefficient vector is [1 1 2], where the first element of the vector corresponds to the coefficient

Table 2.2: Memory Usage for n=6 to 12

n	k	Memory(MB)
6	4	0.0371
	2	0.0520
	0	0.2904
7	5	0.6556
	3	1.327
	1	0.9531
8	6	3.999
	4	40.11
	2	6.426
	0	13.21
9	7	21.61
	5	695.4
	3	107.42
	1	54.22
10	8	113.0
	6	11316.24
	4	5378.71
	2	278.6
	0	551.85
11	9	571.32
	7	Not Sufficient Memory
	5	Not Sufficient Memory
	3	3764.01
	1	3157.97
12	10	2944.47
	8	Not Sufficient Memory
	6	Not Sufficient Memory
	4	Not Sufficient Memory
	2	13973.22
	0	24399.43

Table 2.3: Symbolic expression to numerical matrix

x1	x2	y1	y2
2	0	2	0
0	3	0	0
3	0	0	1

of row 1 of the matrix and the second element of the vector corresponds to row 2 of the matrix and so on. This transformation can be automated for any n and k, and this would be essential especially when the starting polynomials are of large length. This however has been carried out manually in this thesis.

The starting polynomial which is now in the form of a matrix, is multiplied sequentially to polynomials which are functions of x_i, y_i and x_j, y_j, and the number of such polynomials will be the number of distinct combinations of i and j. All these polynomials are also converted to their matrix form before hand. Since all these polynomials are of the exact same type, except for a change in the variable indices, their matrix representations will be permutations of each other. The multiplication results in larger matrices as both the number of distinct variables and number of monomial terms increases. It is essential to understand the complexity of this algorithm. For n=12,and k=0 case the length of the matrix representing the polynomial at every stage is shown in Table 2.5. As can be seen in the algorithm, when all the polynomials which contain a certain pair of variables, usually of the form (x_i, y_i), have been multiplied, then that variable pair is removed from the matrix. Hence the columns containing the exponents of those variables in the matrix are transformed to a different numerical value and the coefficient of the monomials are updated. Due to the reduction in the number of variables the polynomial is a function of, the number of columns and rows of the matrix reduces. This is explained using the following simple example. Let the current polynomial be $3x_1^2y_1y_2x_2^2 + 4x_1^3y_1^2y_2x_2^2$. Let the transformation be $x_1^2 \rightarrow 2!$, $y_1 \rightarrow 1$, $x_2^2 \rightarrow 2!$, $y_2^2 \rightarrow 2$ and $x_1^3 \rightarrow 3!$. The variables are transformed to the factorial of the exponent. After the transformation the polynomial simplifies to $54y_2x_2$. At this stage those variables are removed from the matrix representation of the polynomial. The resultant polynomial has lesser number of terms.

Table 2.4: Matrix length for n=10 and k=0

Polynomial Multiplied	Size of Matrix
p2	169-by-10
p3	1902-by-10
p4	20561-by-10
Variable Reduction	3321-by-8
p5	16875-by-8
p6	76398-by-8
p7	321118-by-8
Variable Reduction	7875-by-6
p8	24644-by-6
p9	72832-by-6
Variable Reduction	1225-by-4
p10	2987-by-4

Table 2.5: Matrix length for n=12 and k=0

Polynomial Multiplied	Size of Matrix
p2	169-by-12
p3	1902-by-12
p4	20561-by-12
p5	216474-by-12
Variable Reduction	29646-by-10
p6	150498-by-10
p7	683163-by-10
p8	2825779-by-10
p9	10975176-by-10
Variable Reduction	195625-by-8
p10	611147-by-8
p11	1788818-by-8
p12	4985644-by-8
Variable Reduction	58996-by-6
p13	143261-by-6
p14	330607-by-6
Variable Reduction	3321-by-4
p15	6913-by-4

Table 2.6: Largest Matrix Size for different n for k=0

n	Size of largest matrix
6	231-by-4
8	8899-by-6
10	321118-by-8
12	10975176-by-10

Tables 2.4 and 2.5 show the length of the matrix corresponding to the polynomial at that instance. p_j's refer to the polynomial which is multiplied at stage 'j'. While Table 2.4 correspond to the matrix sizes during the evaluation of $p_{12,0}$, table 2.5 correspond to $p_{10,0}$. The starting polynomial in both cases consists of fourteen monomials and is a function of twelve variables. Hence it is converted to a matrix of size 14-by-12. After the multiplication of the polynomial p5, a transformation rule is applied to a couple of variables and the number of variables the polynomials is now a function of is reduced by two. Similar reductions in number of variables is evident from the width of the matrix. It is essential to understand the scaling of the algorithm for higher values of n. In the Table 2.6, the size of the largest matrix which occurs in the Algorithm is shown as a function of n for $k = 0$. The Factor of scaling of the matrix sizes across two consecutive even numbers is over forty times. Probability evaluation for most k values for $n = 11$ and $n = 12$ was not possible on a single 4GB ram processor due to memory restrictions and hence the computation had to be carried out in parallel with Star-p. Twelve 4GB ram processors were used for the computation. This large scaling has indeed been the limiting factor in computation of the probabilities for larger values of n.

2.2.3 Results

The $p_{n,k}$ values for $n = 1$ to 11 and certain values for $n = 12$ and 13 are presented Table 2.7. In [1], results for $p_{n,k}$ were obtained only for up to n=9. A single processor with 4GB ram was sufficient to compute some more values for the n=10 and 11 case with the mathematica code. After obtaining the values for k=1,3,9 and 11 the other two values for n=11 were obtained by using the result for the expected number of real eigenvalues in [2] and using the fact that the sum of the probabilities equals one.

For the n=12 and k=4 case, the size of the matrix after the multiplication of the fourth polynomial is of the order of $5 * 10^9$-by-15. The existing set of twelve processors of 4GB ram each was not sufficient to run this code. Hence probability values for $k = 4, 6$ and 8 for n=12 could not be obtained using the code. They were however calculated using the results 2.7, 2.8 and 2.2. While 2.7 and 2.8 are exact, 2.2 is only a conjecture and hence there is some uncertainty with respect to these three values. Round-off erro is always an issue once the numbers get very large. The zeros at the end of $p_{n,k}$ values for the n=12 case cause us to suspect that these values maybe associated with some round-off error. The IEEE standard provides flags to trap when the "INEXACT EXCEPTION" occurs. We are not aware if this is reachable through MATLAB, but in principle we could know from that flag alone whether any rounding errors have occurred and what higher precision computations could still be performed in floating point and reach an exact answer. The values obtained from the MATLAB code were corroborated with results from the Mathematica code run on a processor with 128GB RAM.

Table 2.7: $p_{n,k}$ for n=1 to 12

n	k	$p_{n,k}$		n	k	$p_{n,k}$	
1	1	1	1	9	9	$\frac{1}{262144}\sqrt{2}$	0.00000
2	2	$\frac{1}{2}\sqrt{2}$	0.70711		7	$\frac{-1}{65536}+\frac{5297}{2097152}\sqrt{2}$	0.00256
	0	1-$\frac{1}{2}\sqrt{2}$	0.29289		5	$\frac{82347}{524288}-\frac{15891}{2097152}\sqrt{2}$	0.14635
3	3	$\frac{1}{4}\sqrt{2}$	0.35355		3	$\frac{-82339}{262144}-\frac{1345555}{2097152}\sqrt{2}$	0.59328
	1	1-$\frac{1}{4}\sqrt{2}$	0.64645		1	$\frac{606625}{524288}-\frac{1334961}{2097152}\sqrt{2}$	0.25681
4	4	$\frac{1}{8}$	0.125	10	10	$\frac{1}{2^{28}}\sqrt{2}$	0.00000
	2	$\frac{-1}{4}+\frac{11}{16}\sqrt{2}$	0.72227		8	$\frac{236539}{2^{29}}-\frac{5}{2^{23}}\sqrt{2}$	0.00044
	0	$\frac{9}{8}-\frac{11}{16}\sqrt{2}$	0.15273		6	$\frac{-236539}{2^{27}}+\frac{35098479}{2^{30}}\sqrt{2}$	0.04447
5	5	$\frac{1}{32}$	0.03125		4	$\frac{149206217}{2^{28}}-\frac{105292877}{2^{30}}\sqrt{2}$	0.41712
	3	$\frac{-1}{16}+\frac{13}{32}\sqrt{2}$	0.51202		2	$\frac{-148733139}{2^{27}}+\frac{1216831949}{2^{30}}\sqrt{2}$	0.49453
	1	$\frac{33}{32}-\frac{13}{32}\sqrt{2}$	0.45673		0	$\frac{834100651}{2^{29}}-\frac{1146637039}{2^{30}}\sqrt{2}$	0.04341
6	6	$\frac{1}{256}\sqrt{2}$	0.00552	11	11	$\frac{1}{2^{28}}\sqrt{2}$	0.00000
	4	$\frac{271}{1024}-\frac{3}{256}\sqrt{2}$	0.24808		9	$\frac{333213}{2^{33}}+\frac{-160}{2^{33}}\sqrt{2}$	0.00004
	2	$\frac{-271}{512}+\frac{107}{128}\sqrt{2}$	0.65290		7	$\frac{-20823}{2^{27}}+\frac{60262315}{2^{33}}\sqrt{2}$	0.00977
	0	$\frac{1295}{1024}-\frac{53}{64}\sqrt{2}$	0.09350		5	$\frac{510394091}{2^{31}}+\frac{-180786305}{2^{33}}\sqrt{2}$	0.20791
7	7	$\frac{1}{2048}\sqrt{2}$	0.00069		3	$\frac{-1020121891}{2^{31}}+\frac{6423679969}{2^{33}}\sqrt{2}$	0.58254
	5	$\frac{355}{4096}-\frac{3}{2048}\sqrt{2}$	0.24808		1	$\frac{10629845251}{2^{33}}+\frac{-6303155851}{2^{33}}\sqrt{2}$	0.19975
	3	$\frac{-355}{2048}+\frac{1087}{2048}\sqrt{2}$	0.65290	12	12	$\frac{1}{2^{33}}$	0.00000
	1	$\frac{4451}{4096}-\frac{1085}{2048}\sqrt{2}$	0.09350		10	$\frac{-3}{2^{32}}+\frac{3781485}{2^{41}}\sqrt{2}$	0.00000
8	8	$\frac{1}{16384}$	0.00006	[2]	8	$\frac{27511372605}{2^{44}}-\frac{18907425}{2^{41}}\sqrt{2}$	0.00155
	6	$\frac{-1}{4096}+\frac{3851}{262144}\sqrt{2}$	0.02053	[2]	6	$\frac{-27511352125}{2^{42}}+\frac{126455775487}{2^{41}}\sqrt{2}$	0.07507
	4	$\frac{53519}{131072}-\frac{11553}{262144}\sqrt{2}$	0.34599	[2]	4	$\frac{6237846960567}{2^{43}}-\frac{379291696761}{2^{41}}\sqrt{2}$	0.46523
	2	$\frac{-53487}{65536}+\frac{257185}{262144}\sqrt{2}$	0.57131		2	$\frac{-6182824264509}{2^{42}}+\frac{356179603371}{2^{38}}\sqrt{2}$	0.42669
	0	$\frac{184551}{131072}-\frac{249483}{262144}\sqrt{2}$	0.0621	[3]	0	$\frac{1915540686556992}{2^{50}}+\frac{-664725959617024}{2^{49}}\sqrt{2}$	0.03145
				13	13	$\frac{1}{2^{39}}$	0.00000
					11	$\frac{-3}{2^{38}}+\frac{5396403}{2^{46}}\sqrt{2}$	0.00000

[2]Results for k=4,6 and 8 may not be exact because they have been obtained using conjecture 2.2

[3]The MATLAB code for n=12 and k=0 yielded $\frac{1915540686556995}{2^{50}}+\frac{-664725959617025}{2^{49}}\sqrt{2}$. The difference in result is due to the round-off error.

Table 2.8: Times for the n=8 and k=0 case

n	Code Type	Time(s)
8	Mathematica code	3.41
	MATLAB serial code 1	17.76
	MATLAB serial code 2	0.1158
10	Mathematica code	258.14
	MATLAB serial code 2	3.88

2.2.4 Performance comparison

We use the times for the n=8 and k=0 case as a mode for comparison. Times for the 3 different codes enumerated below are presented in Table 2.8. These codes were run on an Intel Core 2 Duo 2.66GHz processor with 4GB ram. The mathematica code as already mentioned before is available in [1]. There are 2 implementations of the serial MATLAB code, a non-optimized and an optimized version of the same algorithm. The parts of the code which have been implemented differently alone are shown in Appendix A.

1. Serial Mathematica Code [1]

2. Serial MATLAB Code before optimization.

3. Serial MATLAB Code after optimization

The starting polynomial for the k=0 case is 1. The polynomial which is to be multiplied to the starting polynomial to the starting algorithm is represented by the variable Mat1 in Appendix A. There are three operations which occur repeatedly in the algorithm. Their implementation has been optimized in the second serial code. These operations have been numbered segments 1 to 3 in Appendix A. This optimized implementation has improved the performance significantly.

The first segment shows how the process of polynomial multiplication has been optimized. Matrix Mat2 represents a polynomial which is multiplied to the polynomial Mat1. Polynomial multiplication is equivalent to the Matrix addition operation shown in segment 1. The many vector vector additions in the loop are replaced with one Matrix addition reducing it to a one dimensional loop rather than a two dimensional loop which was used in the first serial code. There are a total of 5 such polynomial multiplications in this code and the same optimization procedure has been incorporated to all of them. As the length of the matrix gets larger, the one dimensional reduction of the loop amounts to a very significant reduction in time.

After two polynomials are multiplied which amounts to repetitive addition of matrices, there are going to be many rows which are equal which represent the same monomial. Hence these rows are replaced by a single row, and the corresponding coefficient is obtained by taking the sum of the coefficients of the equal rows. In the non-optimized code, the rows are initially sorted column-wise and equal rows are identified one by one. This procedure is shown in Segment 2. This is not an effective algorithm because row equality is verified only for the next 12 rows and it involves the use of many for loops. The choice of the number 12 though may appear random isn't quite the case. The number has been chosen after some kind of an optimization procedure. A larger number will increase the number of computations in the loops but will aid the identification of a greater number of similar rows and hence reduce the size of the resultant matrix. The opposite will be true for a number smaller than 12. Taking these 2 contradicting factors into consideration, the number 12 has been chosen. In the optimized serial code, this inefficient algorithm has been avoided by the use of the sparse command. The row of exponents is split into two sequence of numbers. Each sequence is considered to be representative of a single integer in the base of the largest number occurring in the matrix plus one. After obtaining two integers which are representative of each row in the matrix, the sparse command is used to combine all rows which have this same set of numbers. This allows the combination of the coefficients of rows which are equal. This procedure doesn't involve the use of "for loops" and the reduction procedure is accurate.

All matrix elements which represent the exponents of the variables in the polynomial are replaced by their specific transformation rule corresponding to the variable they represent and the value of the exponent. This transformation is shown in segment 3. The "for loop" which runs along the length of the matrix, has been replaced by using a logical variable and the transformation can be carried out in a single step.

It is due to these reasons , that we have been able to obtain a speed-up of almost 100 times. The key differences are enumerated in the following tables. The key difference has been the removal of multiple “for loops” all through the code.

Significant reduction in time has primarily been obtained through the removal of the for loops. Hence the usage of “for loops” especially in MATLAB needs to be avoided. The optimized serial code is about 10 times faster than the mathematica. A numerical approach is computationally cheaper than a symbolic algorithm. After we have obtained the optimized serial code, the code is parallelized using the star-p software. The Star-p software was connected to a cluster with twelve processors. The parallelized code hasn't been optimized yet and its results aren't presented.

2.2.5 Expected number of real eigenvalues

If E_n denotes the expected number of real eigenvalues of a n-by-n random matrix, then the Asymptotic number of real eigenvalues is given by equation 2.6.

$$\lim_{n\to\infty}\frac{E_n}{\sqrt{n}} = \sqrt{\frac{2}{\pi}} \tag{2.6}$$

The expected number of real eigenvalues can be obtained accurately through the evaluation of the following expression in equation 2.7.

For n even,

$$\sum_{k=0}^{n} kp_{n,k} = \sqrt{2}\sum_{k=0}^{n/2-1}\frac{(4k-1)!!}{(4k)!!} \tag{2.7a}$$

For n odd,

$$\sum_{k=1}^{n} kp_{n,k} = 1+\sqrt{2}\sum_{k=1}^{(n-1)/2}\frac{(4k-3)!!}{(4k-2)!!} \tag{2.7b}$$

These results are presented in [2]. In an attempt to obtain the probabilities for all values of k for a fixed n , its not necessary to evaluate each of them. The probability is of the form r+s$\sqrt{2}$ and hence has two unknowns, the rational part and the irrational part. For the n=11 case, a total of six probabilities need to be evaluated and this amounts to obtaining twelve unknowns. The sum of all the probabilities for a fixed 'n' equals 1. (equation 2.8 amounts to two equations, as we can equate the rational part to 1 and the irrational part to 0). Similarly the equations governing the expected number of real eigenvalues also amounts to two equations. Hence at this stage the task was reduced to the calculation of just four probabilities rather than six. If more summation formulaes could be obtained, probability evaluation could be further simplified since the evaluation of actual probabilities is computationally more expensive.

$$\sum_{k=1}^{n} p_{n,k} = 1 \tag{2.8}$$

Conjectures

In the very first section of this chapter conjectures relating the rational and irrational parts in equations 2.1 were presented in 2.2,2.3 and 2.4. These results were obtained by observing the data in the table 2.9. Though a formula to compute the summations r_2,s_2,r_3 and s_3 haven't been obtained these conjectures relate these summations. These relations haven't been proved as yet, and at this stage are backed only by the data in Table 2.9. As already mentioned before, the first conjecture is only approximately true for the n=11 case.

2.2.6 Variance of the number of real eigenvalues

Experiments were conducted to predict the asymptotic variance of the number of real eigenvalues of a random matrix. The variance of the number of real eigenvalues of a large number of matrices of different sizes are examined in tables 2.10, 2.11, 2.12 and 2.13. The number of random matrices used for the experiment is represented as "Number of trials" in tables 2.10, 2.11, 2.12 and 2.13.

We can see that the variance normalized with the factor $\sqrt{n}$ approximates 0.5. Based on these results we claim that if σ_n^2 denotes the variance of the number of real

Table 2.9: Summation

n	$r_1 + s_1\sqrt{2}$	$r_2 + s_2\sqrt{2}$	$r_3 + s_3\sqrt{2}$
1	$[1, 0]$	$[0, 0]$	$[0, 0]$
2	$[0, 1]$	$[0, \frac{1}{2}]$	$[0, 0]$
3	$[1, \frac{1}{2}]$	$[0, \frac{3}{4}]$	$[0, \frac{1}{4}]$
4	$[0, \frac{11}{8}]$	$[\frac{1}{2}, \frac{11}{16}]$	$[\frac{1}{2}, 0]$
5	$[1, \frac{13}{16}]$	$[\frac{1}{8}, \frac{39}{32}]$	$[\frac{1}{4}, \frac{13}{32}]$
6	$[0, \frac{211}{128}]$	$[\frac{271}{256}, \frac{211}{256}]$	$[\frac{271}{256}, \frac{1}{32}]$
7	$[1, \frac{271}{256}]$	$[\frac{355}{1024}, \frac{813}{512}]$	$[\frac{355}{512}, \frac{273}{512}]$
8	$[0, \frac{1919}{1024}]$	$[\frac{53471}{2^{15}}, \frac{1919}{2048}]$	$[\frac{53471}{2^{15}}, \frac{3851}{32768}]$
9	$[1, \frac{2597}{2048}]$	$[\frac{82335}{2^{17}}, \frac{7791}{4096}]$	$[\frac{82335}{2^{16}}, \frac{171505}{2^{18}}]$
10	$[0, \frac{67843}{32768}]$	$[\frac{18562075}{2^{23}}, \frac{67843}{65536}]$	$[\frac{18562075}{2^{23}}, \frac{35097199}{134217728}]$
11	$[\frac{8589934592}{2^{33}}, \frac{12485787648}{2^{33}}]$	$[\frac{8158310504}{2^{33}}, \frac{18728681472}{2^{3}3}]$	$[\frac{16316622448}{2^{33}}, \frac{6724989784}{2^{3}3}]$

Table 2.10: Matrix Size 50-by-50

Number of trials	Variance (σ^2)	$\frac{\sigma^2}{\sqrt{n}}$
10^5	3.5876	0.5074
10^6	3.5823	0.5066
10^7	3.5866	0.5073

Table 2.11: Matrix Size 100-by-100

Number of trials	Variance (σ^2)	$\frac{\sigma^2}{\sqrt{n}}$
10^5	4.864	0.4864
10^6	4.981	0.4981
10^7	4.967	0.4967

Table 2.12: Matrix Size 200-by-200

Number of trials	Variance (σ^2)	$\frac{\sigma^2}{\sqrt{n}}$
10^5	6.9982	0.4949
10^6	6.9074	0.4884

Table 2.13: Real Eigenvalue distribution of Large sized Matrices

Matrix Size	Number of trials	Variance (σ^2)	$\frac{\sigma^2}{\sqrt{n}}$
800*800	10^5	6.9074	0.4824
1000*1000	10^4	14.9415	0.4725
1000*1000	10^5	14.9415	0.4756

eigenvalues of a n-by-n random matrix, then

$$\lim_{n\to\infty}\frac{\sigma_n^2}{\sqrt{n}}=\sqrt{\frac{1}{2}}$$

Chapter 3

Eigenvalue Algorithms for sparse symmetric matrices

3.1 Matrices and Processor used for the algorithms

In this thesis a total of 10 matrices are used for the numerical simulations. Since we are solving the generalized eigenvalue problem of the form $Ax = \lambda Bx$, two matrices are required per simulation. The 5 sets of matrices used are shown in Table 3.1

The extension "-n" and "-d" refer to Neumann and Dirichlet boundary conditions. While the Dirchlet matrices are positive definite, the Neumann matrices are positive semi-definite with at least one zero eigenvalue. Matrices with the cube keyword refer to a three dimensional cube of side length 20 voxels and in the case of the ball it refers to a ball with radius of twenty. From the density($\frac{nnz}{s^2}$ where 'nnz' refers to the number of non-zeros in the matrix and s is the size of the matrix) column in table

Table 3.1: Size and density of the Square Matrices used for the experiments

Matrix Names	Matrix Pair	Size(s)	Density (%)
cube20-n-A cube20-n-B	cube20-n	62181	0.0687
cube20-d-A cube20-d-B	cube20-d	50179	0.084
ball20-n-A ball20-n-B	ball20-n	738593	0.1056
ball20-d-A ball20-d-B	ball20-d	28631	0.1464
qa8fm qa8fk	qa8f	66127	0.038

3.1 its quite clear that all the matrices we are dealing with are very sparse. The mass and stiffness matrix qa8fm and qa8fk were downloaded from the Tim Davis matrix collection and more details about these matrices can be found at [14]. These matrices are of the largest size but much sparser than the rest. The two matrices which are used for a single problem are presented in each line of the table 3.1. They constitute a matrix pair and will be referred that way for the rest of the thesis. Figure 5-2, 5-3 and 5-1 shows the sparsity pattern of the matrices ball20-d-A, cube20-n-A and qa8fm. The sparsity pattern of the all the Dirichlet matrices used in this thesis are identical. Hence the figure 5-2 will serve to be representative of all Dirichlet matrices. The same is true for the the Neumann matrices as well.

All the codes used in this thesis, can be easily parallelized using the message passing interface which is already incorporated in the codes. These codes were run on a single Intel Core 2 Duo 2.66 GHz processor with 4GB ram.

3.2 BlockDavidson

The Davidson method is an eigenvalue technique aimed at computing a few of the extreme (i.e., leftmost or rightmost) eigenpairs of large sparse symmetric matrices. This method has gained quite an interest in quantum chemistry where it emanated. However, for the classical Davidson method to be suitable, the matrix dealt with must be strongly diagonally dominant (in the sense that its eigenvectors are close to the canonical vectors). The algorithm then uses the diagonal as preconditioner. Other single and block versions of the Davidson method have already been investigated, with numerical results and comparisons of different variants. For instance, Liu [3] and Kosugi [4] have suggested block versions of the Davidson method and a resulting implementation was proposed in Stathopoulos and Fisher [5]. A number of other implementations of the classical Davidson method have been published, see, for example, Weber et al. [6] and Cisneros and Bunge [7]. However, these cited works, unlike the study done here, are confined to a diagonal-type preconditioning. They do not consider general user-supplied preconditioners, nor do they incorporate deflation and locking. A block version of the Davidson method has been implemented in Trilinos. In [11] various multilevel preconditioners are employed to improve the convergence and memory consumption of the JacobiDavidson algorithm and of the locally optimal block preconditioned conjugate gradient (LOBPCG) method. This algorithm has been implemented with both the ML based preconditioner and INV preconditioner as mentioned in the introductions section.

Block Davidson Algorithm

1. apply Preconditioner N to the current residual: $H = NR$
2. use H to expand the current basis V
3. use new V to compute a projected eigenproblem
4. solve the projected eigen problem and form the Ritz vectors X and the Ritz values ϕ
5. compute the new Residuals R

In implementing the block Davidson method, this iteration repeats until the basis V is full (in which case it is time to restart) or some stopping criterion has been satisfied.

3.2.1 Eigenvalue computation times for varying block size and number of blocks

Its essential to understand how greatly the preconditioners affect the performance of the algorithm in the computation of eigenvalues. We present in Figure 5-4 the computation times of eigenvalues for the following 2 methods.

1. Block Davidson algorithm with Multi-Level Preconditioner.
2. Block Davidson algorithm without any Preconditioner.

Its apparent from the figure 5-4 that in the presence of a preconditioner, the algorithm takes an order of magnitude less to compute the eigenvalues. The results have been presented only for the cube20-d matrix pair. Similar results were observed for the other matrix pairs and has confirmed the significant improvement of the performance of the algorithm in the presence of a preconditioner. In the figure 5-4, associated with every eigenvalue is only one time for computing those many number of eigenvalues (N_v). For every N_v, associated with the different Block Sizes (B_s) and the Number of Blocks (N_b), there are many corresponding computation times. The data presented in the figure is the least of these computation times in a sense the point of optimality.

Block Davidson Algorithm with the ML Preconditioner

Understanding the performance of the algorithm as a function of the parameters B_s and N_b is key to this thesis and is presented in every section for all the three algorithms. The eigenvalue computation times for varying B_s and N_b is shown in figures 5-5 and 5-6. Figure 5-5 and 5-6 shows the times for computing 10 and 25 eigenvalues respectively. From these figures, it is clear that there exists an optimal set of values for B_s and N_b at which the computation time is minimum. However since the times have been plotted only for discontinuous values of B_s and N_b, where adjacent values plotted differ by a magnitude of 5, it isn't possible to obtain the exact point of minima. However based on the trends from the figures 5-5, 5-6, 5-11 and 5-12 it is reasonable to claim that the optimal point occurs in the range of the B_s and N_b used in these figures. The extreme values for B_s is the number of requested eigenvalues on the lower side and extreme values of N_b are 10 and 40. For values of B_s, lower than N_v, the algorithm did not converge. The basis for the claim is the fact that the optimal point is enclosed between the extreme bounds of the parameter. For higher values of N_b, immediately outside the bound the computation time increases in most cases and for further outside the bound, computation isn't possible due to additional memory requirements. The same is true for values greater than the maximum B_s value used in the figures corresponding to each N_v. For the purpose of this thesis, the least computation time in each of the graphs will be referred to as the optimal computation time. This exercise was carried out for different values of number of eigenvalues (NV) and the optimal point from all those graphs were used in the figure 5-4. So in actuality we are comparing the best times for computing a certain number of eigenvalues by both the methods.

Similar results are presented for the Cunningham matrix. Though the matrix is larger in size, its more sparse than cube20-d matrix pair. Just as with the cube20-d matrix pair, the eigenvalue computation times for varying block sizes and number of blocks is shown in figures 5-7 and 5-8.

Block Davidson Algorithm with the INV Preconditioner

Just like the results for the blockdavidson algorithm with the ML Preconditioner was presented in figures 5-6 and figure 5-5 in the earlier section, the corresponding values for the INV Preconditioner for the ball20-d matrix pair is presented in figures 5-11 and 5-12. The preconditioner of the generalized eigenvalue problem will take the form $A^{-1}B$. Hence after computing the exact inverse using the Amesos-Klu method, matrix B is applied to the inverse. The existence of an optimal set of values in this case also is clearly visible from these figures. The extreme values for B_s and N_b are the same as that in the previous subsection. The algorithm was found to converge for $B_s < N_v$. However this din't occur for all values of N_b in the range. Hence they haven't been presented. For the ball20d matrix pair, the optimal time occurs for $B_s < N_v$. These results haven't been presented in this section. However the optimal time will come into consideration in the last section of this chapter. The results for the same Cunningham matrix as in the previous section with the INV-Preconditioner is presented in Figures 5-9 and 5-10.

Neumann Matrices

There are three matrix pairs with at least one zero eigenvalue. Both the Neumann matrix pairs and the qa8f matrix pair. Eigenvalue computation results for the qa8f matrix pair has been shown in the previous sections. For the cube20-n matrix pair, the block Davidson algorithm with the ML Preconditioner is unable to compute more than 20 eigenvalues. We have arrived at this conclusion after waiting for almost ten times

Table 3.2: Computation time for 30 eigenvalues for varying block size and block number for the "INV case"

N_b B_s	10	12	15
30	Doesn't Converge in $>$ 600 s	143.67s	Doesn't Converge in $>$600 s
35	Doesn't Converge in $>$ 600 s	156.62	Doesn't Converge in $>$ 600 s

the amount of time which the other algorithms took to compute the same number of eigenvalues. Though it may be possible that a larger waiting time may eventually calculate the eigenvalues, from an application point of view that verification isn't important since other algorithms have been able to obtain the eigenvalues in much less time. The results for this matrix pair with the INV Preconditioner is presented in Table 3.2.

Results for the cube20-n matrix have been difficult to obtain. For N_v=30, for each block size, the following Number of Blocks were considered 10,12 and 15. The algorithm was found to converge only for N_b=12. A certain minimum value for Number of Blocks is required for the Algorithm to converge. Usually when the Number of Blocks is too large for the machine too handle, then it throws a memory error. In this case for values larger than 12, we don't find the algorithm producing any eigenvalues. The time for computing the INV Preconditioner is 647 seconds. Even though the algorithm was found to converge only for one value of N_b in the range considered, it is quite reasonable to conclude that 790.67 seconds is close to the optimal value for the total time.

For the ball20-n matrix pair, just like in the previous case, the algorithm was unable to compute large number of eigenvalues. when the algorithm was implemented with the ML Preconditioner. However no such problems were faced when the algorithm with the INV Preconditioner was used to calculate the eigenvalues. The results for this matrix with the INV preconditioner are presented in figure 5-14. Though the eigenvalues can be computed for $B_s < N_v$, the algorithm doesn't converge for all values of N_b. Hence these values haven't been presented in the figure 5-14. However the point of optimality does occur for values of $B_s < N_v$ which are not in the figure.

B_s has to be greater than a certain threshold value for each N_v to be able to compute the required number of eigenvalues.

3.2.2 Locking

If a Ritz value θ has converged (meaning $\|Ax - xB\theta\| < \epsilon_D$) and is thought to be a member of the wanted set of eigenvalues, we wish to declare it converged, decouple the eigenpair (x, θ), and continue to compute remaining eigenvalues with no further alteration of x or θ. This process is called locking. More about deflation and locking mechanisms can be found at [15]. As mentioned in the introductory section, Trilinos enables the use of locking mechanisms. The block davidson algorithm implemented on the qa8f matrix pair, were of two types, one with locking and one without. Figure 5-13 show the effects of incorporating the locking mechanism into the algorithm. The optimal time for computing a certain number of eigenvalues are presented in the figures. The performance with locking is far superior to the one without. The effect of locking is more prominent for this matrix pair than for the cube20-d matrix pair. About locking, there is a significant improvement in performance only when the matrix pair has a zero eigenvalue. The effect of locking is also significant in the cube20n-A matrix pair. With so many parameters, its possible to improve computation times for eigenvalues by large magnitudes. Combination of these varying techniques which are otherwise individually present in Trilinos, is the focus of this chapter of the thesis.

Locking mechanism has been incorporated to all algorithms. In order to solve the neumann matrices, it was essential to incorporate this mechanism into the algorithm. Also while adjusting the parameters, the Block Davidson and LOBPCG Algorithm can compute eigenvalues for $B_s < N_v$ only with the locking mechanism.

3.2.3 Comparing Preconditioners

In the beginning of this section, the performance of the blockdavidson algorithm with and without a preconditioner were compared and the comparison highlighted the function of preconditioners in improving the performance significantly. Now we seek to compare the conditions under which the different preconditioners discussed in the earlier section are more effective. From the figure 5-15, it is clear that the ML-Prec method is more effective when the number of eigenvalues requested isn't very large and the INV method is more effective in the other eigenvalue regime. The accurate inverse preconditioner takes a lot of time to be built in comparison to the ML based preconditioner. However time for solving for the eigenvalues is lesser in the INV Prec case. For this matrix pair, from the slope of two graphs in figure 5-15, it is clear that after a certain number of eigenvalues, the ML method will take more time to compute the eigenvalues. The large preconditioner build time, is compensated by the quick estimation of the eigenvalues. An important concern of this method, is that it gets very difficult to compute the inverse when the matrix is not very sparse. Since the matrix we are using has a very small density of "0.084" , this method turns out to be superior for the estimation of a large number of eigenvalues. The results for the qa8f matrix pair presented in 5-17 are quite different from those of the other two matrices. The time for computing the INV Preconditioner takes about 177 seconds and the corresponding value for the ML Preconditioner is about 0.5 seconds . Also unlike in the figures 5-15 and 5-16, the slope of both the lines are approximately equal. This implies that even when we seek to compute larger number of eigenvalues, the performance with the ML Preconditioner is superior to that with the INV Preconditioner. An explanation for why an accurate matrix inverse is less efficient than an approximate one is difficult to explain. The extreme sparsity of the matrix could be responsible for this odd behavior.

3.3 Block KrylovSchur

The KrylovSchur method belongs to the implicit restart category, i.e., the restarting vector is obtained not by explicit polynomial filtering but by implicit filtering. Sorensen [3] achieved the implicit polynomial filtering by utilizing the property of the shifted QR algorithm. A general rule on block methods based on Krylov-type decompositions, e.g., block Lanczos or Arnoldi decomposition, is that a large block size does not yield good efficiency. Besides hardware concerns (different cache size for different architecture, etc), and the already mentioned less significant gain for a block method on the matrix-vector products for large sparse matrices, we point out that in a restarted Krylov subspace method, if the maximum degree of the Krylov polynomial of a single vector method is k_deg, then the maximum degree of the Krylov polynomial of a block size b method using the same dimension subspace is floor(k_deg/b. If b is relatively large, it means the block method always applies a low degree Krylov polynomial, which may be rather inefficient for the Krylov subspace method. Just like in the block Davidson case, this algorithm is also implemented in Trilinos with the provision to modify all the parameters discussed above.

3.3.1 Results

An important objective of this thesis is to compare the various algorithms and which is more suited for the matrices in consideration. We begin with the results of the block KrylovSchur Algorithm, for the dirichlet matrices. In order to reduce redundancy, we present only one set of results for INV Preconditioner case. The eigenvalue computation times for 10 and 25 eigenvalues vs Number of Blocks for different Block Sizes with the INV Preconditioner is shown in 5-18 and 5-19respectively. While 5-18 corresponds to the ball20d matrix pair, figure 5-19 presents the results for the cube20d matrix pair.

During the course of the Algorithm, the matrix equation in 3.1 needs to be solved using an iteration technique. A refers to the first matrix in the matrix pair, and b is the right hand side which is created during the course of Algorithm. This system can be solved using a direct factorization by obtaining the accurate matrix inverse using the "Amesos-Klu" method or by estimating an approximate matrix inverse using the the "ML method". The advantages in either case has already been explained in previous sections. In this thesis, the results are presented only for the INV Preconditioner case.

$$Ax = b; \tag{3.1}$$

After presenting the results for the dirichlet matrices, we proceed to present the results for Neumann matrices ball20-n and cube20-n matrices. The Neumann matrices has at least 1 zero eigenvalue. The eigenvalue computation times for the Neumann matrices is plotted in figure 5-20 and 5-21. These results are similar to the results for the ball20d-matrix pair. There exists optimal values at which the computation time is minimal. Increase in the block size, does improve the performance. An increase in number of blocks improves the performance until a certain value, following which the computation time increases. The striking feature of these figures is that fact that, the least computation time occurs for very small values of B_s.

For the ball20-n matrix pair, we can see that the values of N_b begin from 10 in both the graphs 5-20 and 5-21. The computation times for N_b=5 haven't been included in the graph, since the algorithm did not converge for the B_s=2 case. Those results are presented in Table 3.3. Even for $B_s = 5$ and 10, it takes a lot of time to compute the eigenvalues because the resulting basis size is quite small. However the small basis size alone can't explain the the reason for large computation time since by using a different combination of B_s and N_b, even though the resulting basis was only 15, the computation time was around 27 seconds. However the only thing which can be concluded from these experiments is that, a certain minimum basis size is required, to be able to compute the required number of eigenvalues.

For the cube20-n matrix pair, the results are shown in Table 3.4. The size of the matrices are larger than the Dirichlet matrices and hence for $N_b = 25$ and greater, the memory is insufficient for $B_s >= 30$. Similarly when $B_s = 40$, memory is insufficient when N_b exceeds 20. Also for $N_b <= 10$, the algorithm takes a very long time to converge (effectively indeterminant). This implies that a certain minimum number of blocks are required to obtain the eigenvalues in less than three minutes. The least time required to compute 30 eigenvalues is about 130.15 seconds. Computation of the inverse takes 646 seconds and hence the total time required is 776.15 seconds which closely compares to the total time taken by the block Davidson Algorithm.

Table 3.3: Computation time for 10 eigenvalues for ball20-n with $N_b = 5$

Block Size	Time	Basis Size
2	Doesn't Converge	
5	262.59	25
10	535.09	5
15	279.68	15

Table 3.4: Computation time for 30 eigenvalues for cube20n with BK algorithm

N_b / B_s	10	15	20	25
30	Doesn't Converge in > 600 s	130.15	180.73	MI
40	Doesn't Converge in > 600 s	177.35	Memory Insufficient(MI)	MI

3.4 LOBPCG-Locally optimal Block Pre-conditioned Conjugate Gradient Method

The LOBPCG method has recently attracted attention as a potential competitor to the Lanczos and Davidson methods due to its simplicity, robustness and good convergence properties. In LOBPCG for computing a single eigenpair of the pencil A-λB, the new iterate is determined by the Rayleigh Ritz method on a three-dimensional subspace, which includes the previous iterate in addition to the current iterate and the preconditioned residual T(Ax-λBx). Since the Algorithm doest depend on the number of blocks the variation of eigenvalue computation time is plotted only with different block sizes. We can see a significant improvement in computation time as we increase the block size over the number of requested eigenvalues especially for large number of eigenvalues.

3.4.1 Results

In this section the LOBPCG algorithm coupled with the ML and INV preconditioner is used to compute the eigenvalues for the same set of matrices. Figure 5-22and 5-23 shows the effect of block size variation for different values of N_v for the cube20-d and ball20-d matrix respectively with the ML Preconditioner. Figures 5-24and 5-25 shows

Table 3.5: Computation time for 30 eigenvalues for varying block size for INV Case"

	Block Size	
Preconditioner Type	30	40
INV	640 + 718.39 = 1358.39	640+963.28 = 1603.28
ML	0.6+748.22=748.82	0.6+1015.41 = 1016.01

the effect of block size variation for different values of N_v for the cube20-d and ball20-d matrix respectively with the INV Preconditioner. These plots are quite different from the others. The values on the horizontal axis is a function of the variable N_v , the number of eigenvalues which are computed. For N_v=5, the computational times are for B_s = 5,10 and 15. Similarly for N_v=20, the corresponding B_s values are 20,25 and 30. For all the figures, the reduction in computation times as the block size is increased is quite significant for larger number of eigenvalues. There isn't significant reduction in computation time for the $N_v = 5$ eigenvalues case. In the previous section for the Block KrylovSchur Algorithm, we observed that the optimal time was obtained for very small values of B_s. This algorithm also converges to some of the eigenvalues for these small value of B_s, however it is unable to compute all the required eigenvalues. Hence the optimal computation time is obtained for values of $B_s > N_v$.

LOBPCG algorithm takes over twelve minutes to compute 30 eigenvalues for the cube-20-n matrix pair as shown in Table 3.5. The results don't improve for increasing values of Block Size. The two times refer to the time for computing the preconditioner which in this case is the inverse of the matrix and the time, for actually computing the eigenvalues. The cumulative time is larger for the "INV case".

3.5 Comparison of the 3 Algorithms

The results for the 3 Algorithms with the Inverse preconditioner have been shown in Figures 5-26, 5-27 and 5-28. In terms of time, the block KrylovSchur Algorithm is superior to the other two algorithms for these set of matrices. In order to obtain the optimal time for the computation of a certain number of eigenvalues, the eigenvalues are computed for a variety of Block Sizes(B_s) and Number of Blocks (N_b). Due to time constraints, only discontinuous values of B_s and N_b were considered. While comparing the computation times for a fixed N_v and the same B_s and N_b, all 3 algorithms perform on par with each other. For the Block Krylov Schur Algorithm, the optimal computation time for $N_v = 5$ to 20 occurs at $B_s = 2$ and for $N_v > 20$ occurs at $B_s = 3$. The Block Davidson algorithm is unable to converge to the required number of eigenvalues for this small value of B_s and the LOBPCG algorithm is unable to compute all the eigenvalues for these small values of B_s. The Block Davidson algorithm produces the eigenvalues for all values of N_b when the B_s is greater than or equal to N_v. For values of B_s lesser than N_v, though for some values of N_b, it produces the optimal computation times, for other values of N_b it doesn't converge to the eigenvalues at all. Another important aspect which needs to be taken into consideration is that the fact, the eigenvalues are computed only for discontinuous set of B_s's and N_b's. Hence the optimum time shown for each N_v, might not be the exact optimum time as explained in the earlier section.

Chapter 4

Conclusion and Future Work

In the chapter titled “On the computation of probabilities associated with real eigenvalues of random matrices” , the symbolic algorithm used to calculate the probability $p_{n,k}$ that ‘k’ real eigenvalues of a random matrix are real was implemented as a purely numerical one. The numerical algorithm helped make computation cheaper, improved computation times and enabled the computation of $p_{n,k}$ for higher values of ‘n”.

Star-p, a parallel computation platform was used to calculate $p_{n,k}$. It was connected to a total of twelve 4GB processors. All probability values for $n <= 11$ have been obtained exactly. However for $n = 12$, all the probability values have not been obtained. Both the MATLAB code run on the parallel platform and the mathematica code run on a processor with 128GB ram of memory were not able to compute the probability values for $n = 12$ and $k = 4$, 6 and 8. These values have been obtained however with the use of conjecture 2.2. Hence the exactness of the probability values aren’t known at this stage, even though the decimal equivalent of these results match the answers obtained through a numerical simulation procedure. The algorithm doesn’t scale linearly and hence even with the use of thirty dual processors, almost a fivefold increase in memory availability, further evaluations were not possible with the MATLAB code. For the n=12 and k=4 case, the matrix which needs to be stored is of size $5.4 * 10^9$-by-15. The largest matrix which can be stored with the current available hardware consisting of 12 dual processors is of the order of $8 * 10^7$-by-15. Based on this we require almost thirty times the number of processors which are currently available. However obtaining superior hardware and greater memory isn't the way to move forward. Though an expression to exactly compute the probabilities is now available, simplifying this expression to enable cheaper computation of the probabilities will be an important aspect of the future work. Just like we managed to parallelize the MATLAB code, to be able to do the same with the Mathematica Code would be very helpful for further calculation.

The largest odd number which can be stored in MATLAB without round-off error is $2^{52} - 1$. However some of the numbers which occur during the course of the evaluation for the n=12 case, are a product of a power of two and odd numbers exceeding 2^{52}. Hence there is round-off error associated with the storage of these values. The result for the n=12 and the k=2 case, obtained through the MATLAB code is verified by running the Mathematica code. Up to the n=11 case, there is no round-off error in the probability values. Some of the calculations in the MATLAB code involves over fifty million floating point operations. Since all the fractions are dyadic rationals(rational numbers with power of 2 in the denominator), this computation was possible. In the presence of quad precision, the computation which at this stage has been achieved only for up to $n < 12$ without round-off error can be extended to higher values of 'n'.The use of quad precision will be necessary to calculate the probability values for larger n, since we will come across numbers of larger magnitude. Also through the incorporation of the "INEXACT EXCEPTION" flag, we will be able to determine whether and where the round-off error occurs.

Theorems are always very useful and open new channels of research. In similarity to the result for the expected number of real eigenvalues which was obtained in the paper [2], the results pertaining to the variance of the real number of eigenvalues and the summations in Equations 2.1 will prove to be useful. These equations have been helpful in enabling the computation of probabilities through the solving of a simple system of linear equations to find the unknowns rather than the actual estimation of the probabilities for all values of k for a fixed n. The conjectures if proved will serve the cause of providing 2 more equations to enable the computation of the eigenvalues. More such summation results for higher powers of k will prove to be very useful for the same purpose as well. These conjectures haven't been used as yet for probability estimation as there would be uncertainty associated with the obtained probability values. Future research will focus on proving these conjectures.

In the chapter titled "Eigenvalue Algorithms for sparse symmetric matrices", the eigensolver algorithms were analyzed with many different parameters. The experiments have enabled us to make certain key observation. The computation times are expected to decrease with increasing B_s and N_b. As the number of blocks are increased for a fixed Block Size, the decrease in eigenvalue computation time occurs till a certain value of N_b after which the time for computation increases The reverse trend for larger number of blocks could be due to the fact that more memory is required, due to the storage of a larger number of basis vectors. This could cause a considerable decrease in performance especially because the matrices we are dealing with are quite large in size.

The computation times for the Block Krylov Schur method in most cases is lesser than that of the Block Davidson and LOBPCG algorithms. This is because this method has been able to converge to the eigenvalues for very small Block Sizes as well. The advantage of a preconditioner has been well understood. When the target is to compute a large number of eigenvalues in our case, a large N_b improves performance significantly. The LOBPCG Algorithm is independent of the N_b parameter and we see that there is improvement in computation time as the B_s exceeds N_v.

The advantages of locking are very significant. The Block Davidson and LOBPCG algorithm have been able to compute the eigenvalues for the Neumann matrices only after they have been incorporated with the locking mechanism. Also in the subsection locking, the benefits of locking to solve the qa8f matrix pair which has one zero eigenvalue has been demonstrated. For the cube20-n matrix pair, computation of the eigenvalues was possible for very selected combinations of B_s and N_v.

The code for the Block Krylov Schur method with the ML Preconditioner hasn't been completed. At this stage the LOBPCG Algorithm with the ML Preconditioner seems to be most optimal for computing 30 eigenvalues. However for larger number of eigenvalues, its possible that the methods based on the INV Preconditioners will perform better. Completion of the above mentioned code will also enable further comparisons.

All the analysis in this thesis has been done for computing the smallest eigenvalues of the matrix. Computation of the other extreme eigenvalues is also important and will constitute a significant part of future work . As already mentioned in the introduction section, there are a decisions abound in implementing this algorithm, the size of the computed basis, if and how the method is restarted and the convergence criterion for the algorithm. Another avenue for future research is studying the effect of the above parameters on the performance of the algorithms.

Bibliography

[1] Alan Edelman (1993) "The circular law and the probability that a Random Matrix has k real eigenvalues" Department of Mathematics, University of California, Berkeley

[2] Alan Edelman, Eric Kostlan and Michael Shub (1994) "How many EigenValues of a Random Matrix are real" , Journal of the American Mathematical Society, Volume 7, Number 1, Jan

[3] B. Liu, The simultaneous expansion for the solution of several of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices, in: Numerical Algorithms in Chemistry: Algebraic Method, eds. C. Moler and I. Shavitt (LBL-8158 Lawrence Berkeley Lab., 1978) pp. 49−53

[4] N. Kosugi, Modification of the LiuDavidson method for obtaining one or simultaneously several eigensolutions of a large real-symmetric matrix, Comput. Phys. 55 (1984) 426−436

[5] A. Stathopoulos and C.F. Fischer, A Davidson program for finding a few selected extreme eigenpairs of a large, sparse, real, symmetric matrix, Comput. Phys. Comm. 79(2) (1994), 268−290.

[6] J. Weber, R. Lacroix and G. Wanner, The eigenvalue problem in configuration interaction calculations: a computer program based on a new derivation of the algorithm of Davidson, Comput.Chemistry 4(2) (1980), 55−60.

[7] G. Cisneros and C.F. Bunge, An improved computer program for eigenvectors and eigenvalues of large configuration interaction matrices using the algorithm of Davidson, Comput. Chemistry 8(8) (1986),157−160.

[8] Miloud Sadkane and Roger B. Sidje, Implementation of a variable block Davidson method with deflation for solving large sparse eigenproblems. Numerical Algorithms 20 (1999), 217−240

[9] Yunkai Zhou and Yousef Saad, Block Krylov-schur method for large symmetric eigenvalue problems,Numerical Algorithms 47(4), 341−359(2008)

[10] Peter Arbenz,Ulrich L. Hetmaniuk, Richard B.Lehoucq2 and Raymond S. Tuminaro, A comparison of eigensolvers for large-scale 3D modal analysis using AMG-preconditioned iterative methods, International Journal for Numerical Methods in engineering 2005, 64:204−236

[11] Peter Arbenz and Roman Geus, Multilevel preconditioned iterative eigensolvers for Maxwell eigenvalue problems, Applied Numerical Mathematics 54 (2005), 107−121

[12] `http://trilinos.sandia.gov/packages/docs/r8.0/packages/anasazi/doc/html/group__an`

[13] `http://trilinos.sandia.gov/packages/docs/r5.0/packages/amesos/doc/html/classAmeso`

[14] http://www.cise.ufl.edu/research/sparse/matrices/Cunningham

[15] http://www.cs.utk.edu/ dongarra/etemplates/node123.html

[16] `http://math.ucdenver.edu/~aknyazev/research/conf/cm04_soft_locking/cm04.pdf`

APPENDIX A

```
% Matrix Representation of the polynomial
Mat1=[ 1 1 0 0 2 0 0 0;
       1 1 0 0 0 2 0 0;
       3 1 0 0 0 0 0 0;
       1 3 0 0 0 0 0 0;
       2 2 0 0 0 0 0 0;
       2 0 0 0 2 0 0 0;
       2 0 0 0 0 2 0 0;
       0 2 0 0 0 2 0 0;
       0 2 0 0 2 0 0 0;
       0 0 0 0 2 2 0 0;
       4 0 0 0 0 0 0 0;
       0 4 0 0 0 0 0 0;
       0 0 0 0 4 0 0 0;
       0 0 0 0 0 4 0 0];

% Coefficient Vector
a=[ -4 -4 -4 -4 6 2 2 2 2 -2 1 1 1 1];

%--Multiplication of polynomial Mat1 to polynomial Mat2-Segment 1----------

for i=1:14
     for j=1:14
       fin_exp2(14*(i-1)+j,:)=Mat1(i,:)+Mat2(j,:);
       fin_coeff(14*(i-1)+j)=a(i)*a(j);
     end
end

% Optimized Code
```

```
for j=1:14
    fin_exp2(i*(j-1)+1:i*j,:)=Mat1+ones(i,1)*Mat2(j,:);
    fin_coeff2(i*(j-1)+1:i*j)=a*a(j);
end
%---------------------------------------------------------------------------

%------------Compression of the matrix-Segment 2-------------------------

[e,I]=sortrows(e,[1 2 3 4 5 6 7 8]); coeff=coeff(I);
[exp,c,len]=compresss1(coeff,e);
 for i=1:len
     for j=i+1:i+12
         if j<len+1
             if (e(i,:)==e(j,:) )
                 coeff(i)=coeff(i)+coeff(j);
                 coeff(j)=0;
             end
         end
     end
 end
 t=1;
 for i=1:len
     if abs(coeff(i))~=0
         exp(t,:)=e(i,:);
         c(t,1)=coeff(i);
         t=t+1;
     end
 end
 len=length(exp);
%---------------------------------------------------------------------------
```

```
%----------------------Optimized Code------------------------------------
q=size(fin_exp,2)/2;

hash1=fin_exp(:,1:q)*(13.^(q-1:-1:0)')+1;
hash2=fin_exp(:,q+1:2*q)*(14.^(q-1:-1:0)')+1;

x=zeros(90,1);
x(48:57)=0:9;
x(65:90)=10:35;

sparsec1=sparse(hash1,hash2,fin_coeff);
[i1,j1,c1]=find(sparsec1);
i1=dec2base(i1-1,13);j1=dec2base(j1-1,14);

c=c1;

expo=zeros(length(c1),2*q);
expo(:,1:q)=x(i1+0);
expo(:,q+1:2*q)=x(j1+0);
len=size(expo,1)
%------------------------------------------------------------------------

%-----------Replacing Matrix exponents-Segment 3-------------------------

for i = 1:len
    j=1;
        if mod(exp(i,j),2)~=0
            exp1(j,1:2,1:2)=0;
        else
            exp1(j,1:2,1:2)=eye(2)*dfact(exp(i,j)-1)/2^(exp(i,j)/2);
        end

    j=b;
        if exp(i,j)==1
            exp1(j,1:2,1:2)=(0.5*(-eye(2)+a2));
        else
            switch exp(i,j)
             case 3
            exp1(j,1:2,1:2)=(0.5*(eye(2)-ia));
            case 5
            exp1(j,1:2,1:2)=-eye(2)+ 1.75*ia;
            case 7
            exp1(j,1:2,1:2)=-3*(-eye(2)+ 1.125*ia);
            case 9
            exp1(j,1:2,1:2)= 12*(-eye(2)+1.671875*ia);
            end
        end

   c1(i,:,:)=squeeze(exp1(1,:,:))*squeeze(exp1(b,:,:))*squeeze(c(i,:,:));
end
%----------------------------------------------------------------------------
%--------------Optimized Code----------------------------------------------
j=1;
for l=0:12
    M1 = (exp(:,j)==l);
    if mod(l,2)~=0
        expa(M1)=0;
    else
        expa(M1)=dfact(l-1)/2^(l/2);
    end
end

j=5;
    M1 = (exp(:,j)==1);
    expb1(M1)=-0.5;
    expb2(M1)=0.5;
    M1 = (exp(:,j)==3);
    expb1(M1)=0.5;
    expb2(M1)=-0.25;
    M1 = (exp(:,j)==5);
    expb1(M1)=-1;
    expb2(M1)=0.875;
    M1 = (exp(:,j)==7);
    expb1(M1)=3;
    expb2(M1)=-1.6875;
    M1 = (exp(:,j)==9);
    expb1(M1)=-12;
    expb2(M1)=10.03125;
%---------------------------------------------------------------------------
```

Chapter 5

Figures

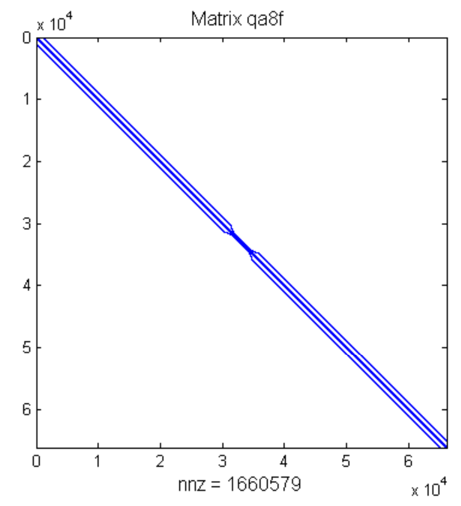

Figure 5-1: Element Distribution in Matrix qa8fm

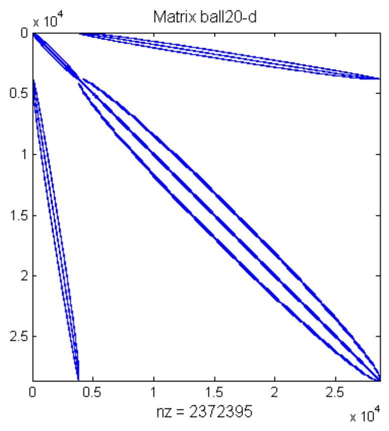

Figure 5-2: Element Distribution in Matrix ball20-d-A

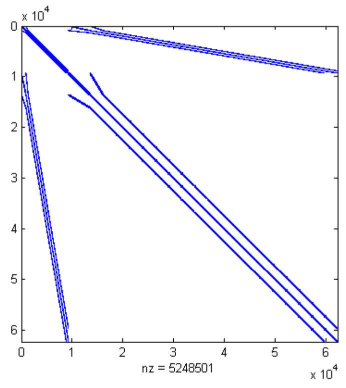

Figure 5-3: Element Distribution in Matrix cube20-n-A

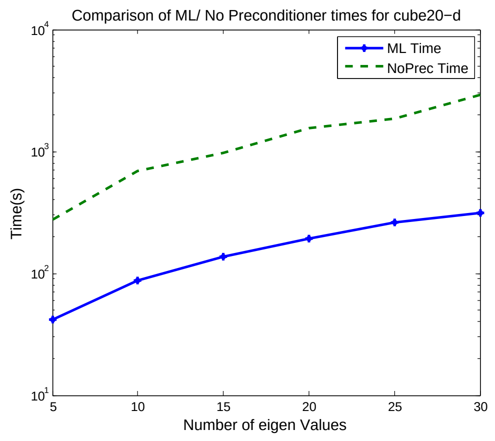

Figure 5-4: ML and No Preconditioner comparison for cube20-d

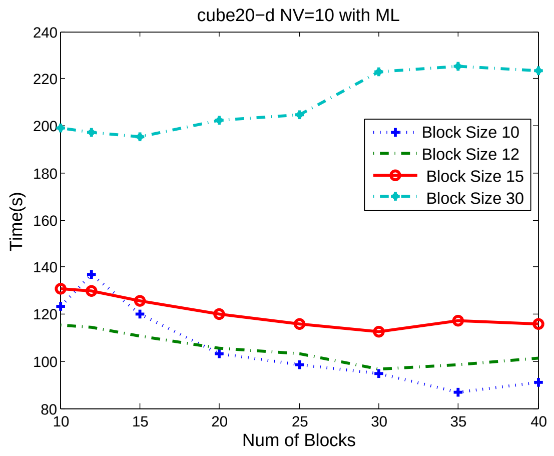

Figure 5-5: ML for NV=10 cube20-d

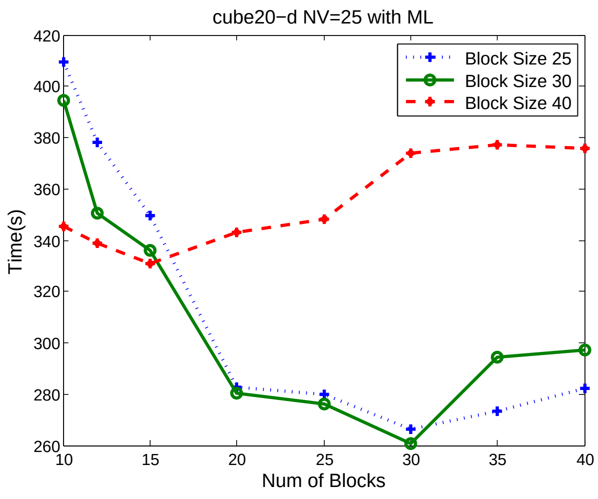

Figure 5-6: ML for NV=25 cube20-d

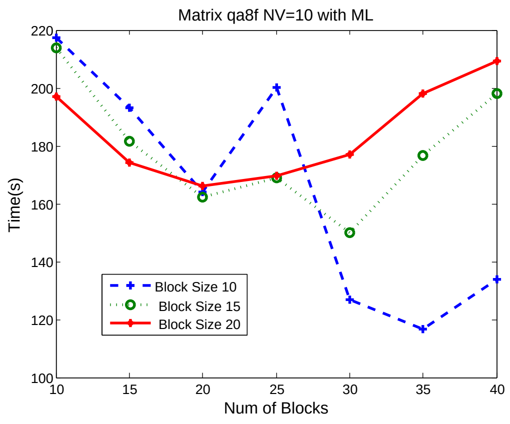

Figure 5-7: ML for NV=10 qa8f

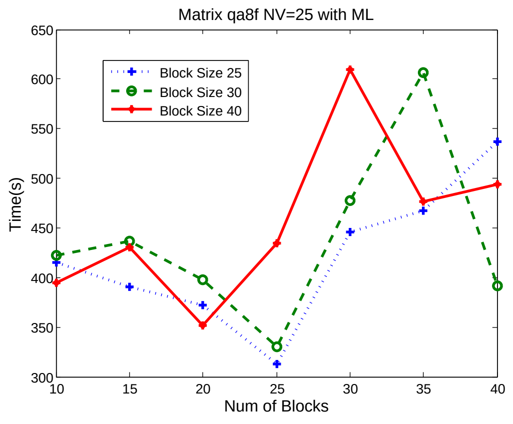

Figure 5-8: ML for NV=25 qa8f

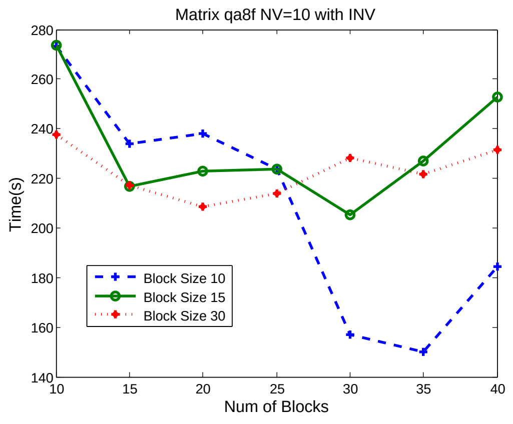

Figure 5-9: INV for NV=10 qa8f

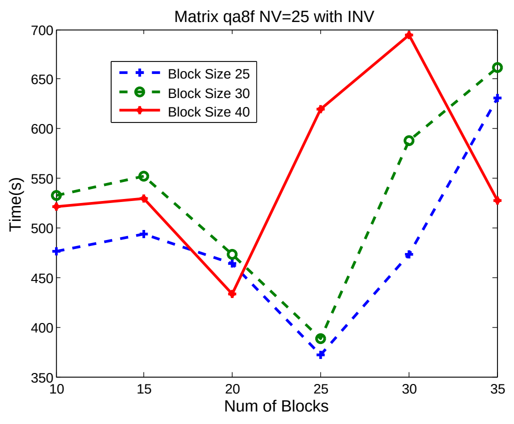

Figure 5-10: INV for NV=25 qa8f

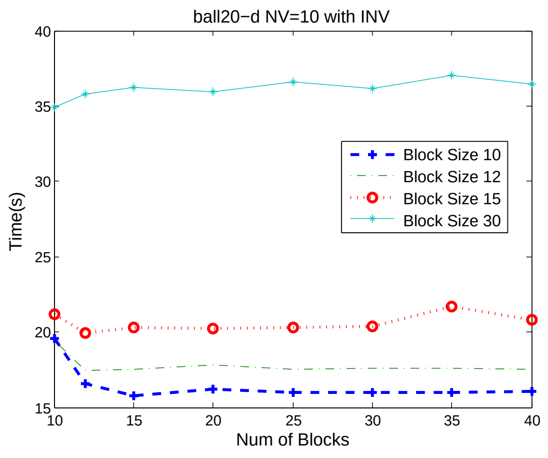

Figure 5-11: INV for NV=10 ball20-d

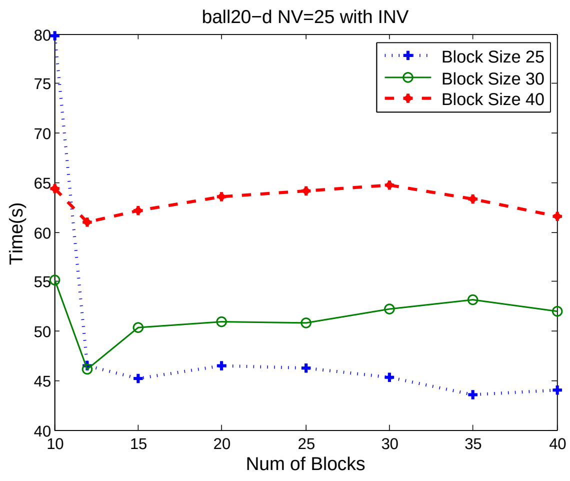

Figure 5-12: INV for NV=25 ball20-d

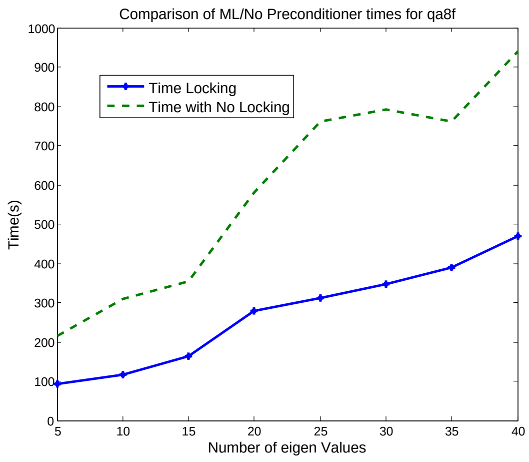

Figure 5-13: With/without Locking with ML Preconditioner for the Cunningham Matrix

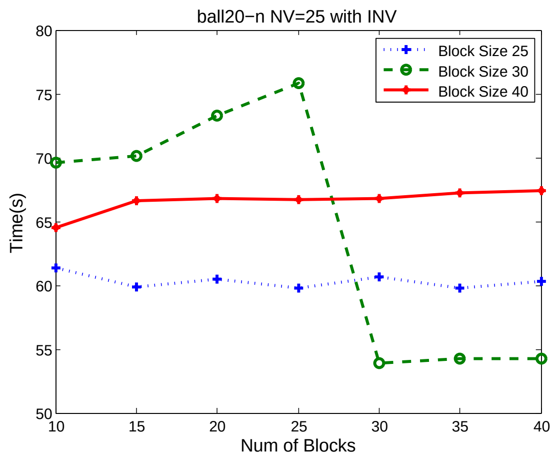

Figure 5-14: INV for NV=25 ball20-n

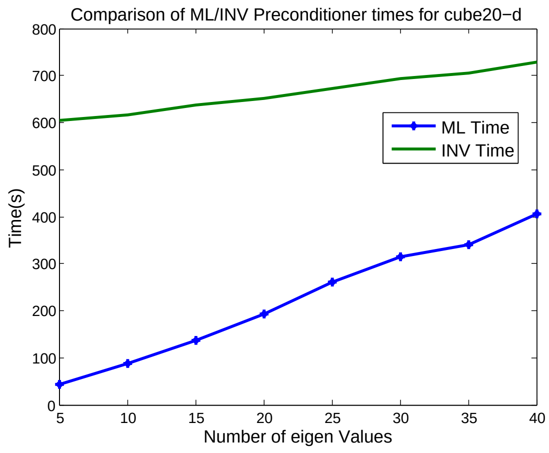

Figure 5-15: ML and INV comparison for cube20-d

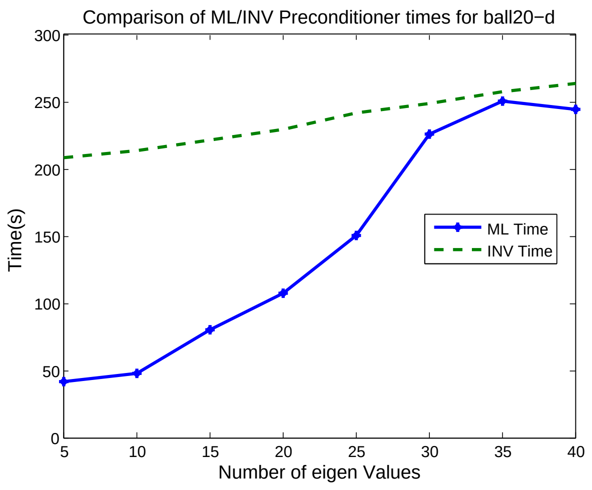

Figure 5-16: ML and INV comparison for ball20-d

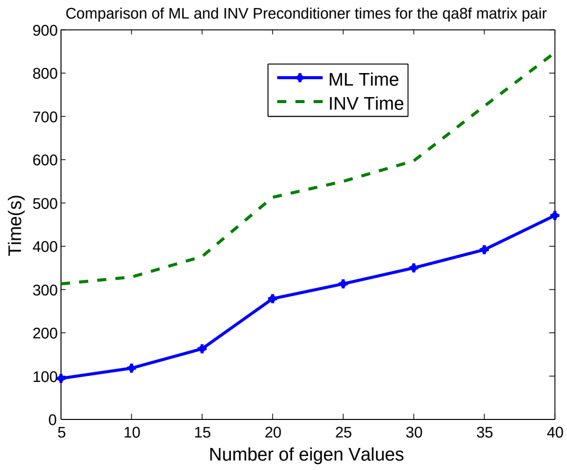

Figure 5-17: ML and INV comparison for ball20-d

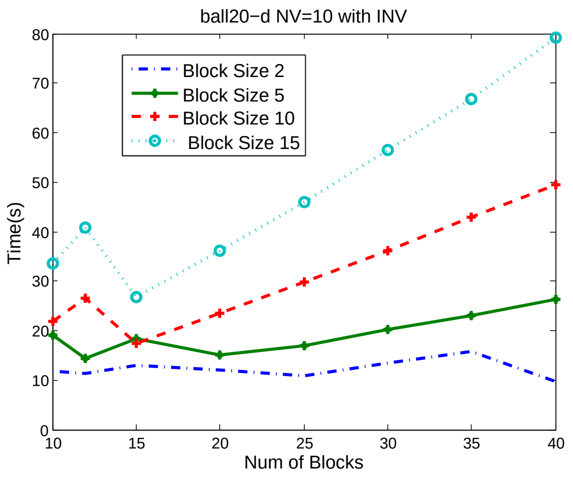

Figure 5-18: INV for NV=10 ball20-d

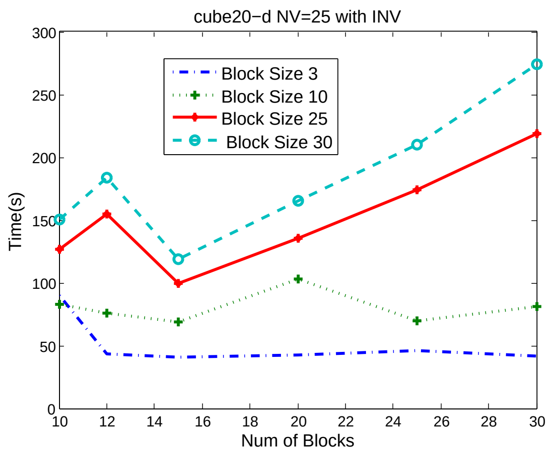

Figure 5-19: INV for NV=25 cube20-d

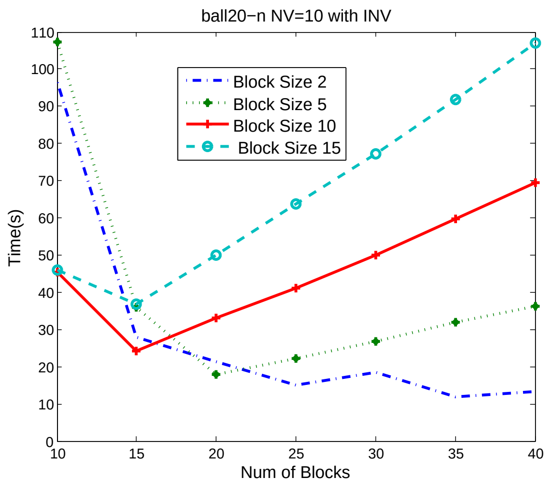

Figure 5-20: INV for NV=10 ball20-n

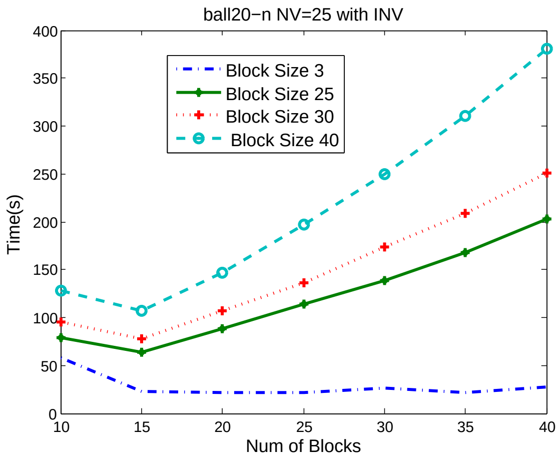

Figure 5-21: INV for NV=25 ball20-n

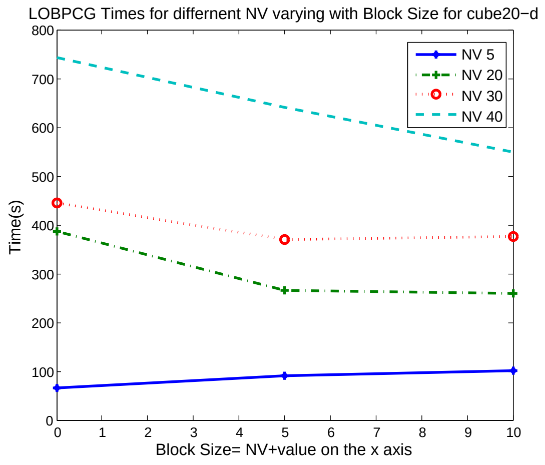

Figure 5-22: LOBPCG Times for different NV varying with Block Size for cube20-d with ML

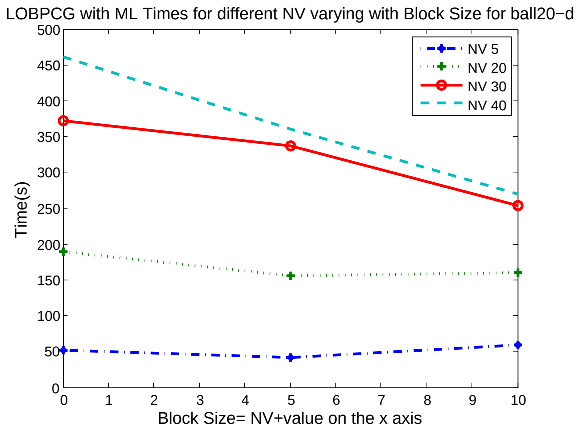

Figure 5-23: LOBPCG Times for different NV varying with Block Size for ball20-with ML

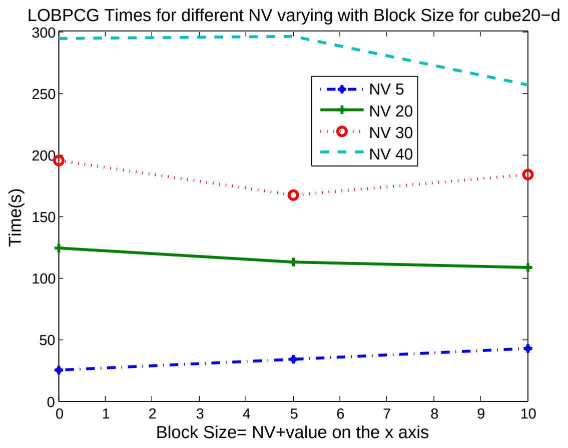

Figure 5-24: LOBPCG Times for different NV varying with Block Size for cube20-d with INV

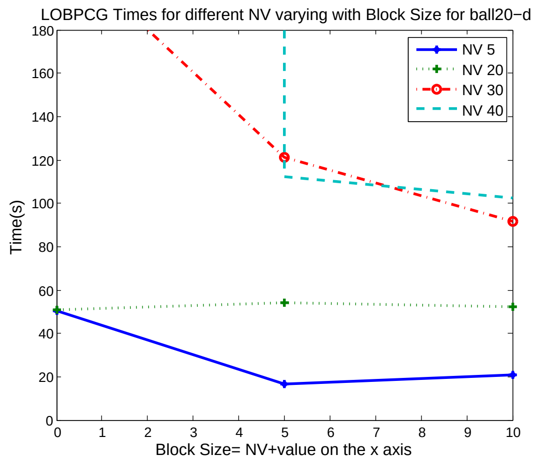

Figure 5-25: LOBPCG Times for different NV varying with Block Size for ball20-d with INV

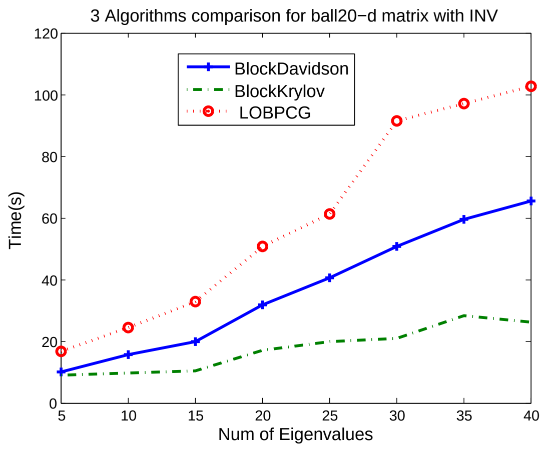

Figure 5-26: Comparison of the 3 Algorithms for the ball20-d matrix pair with INV Preconditioner

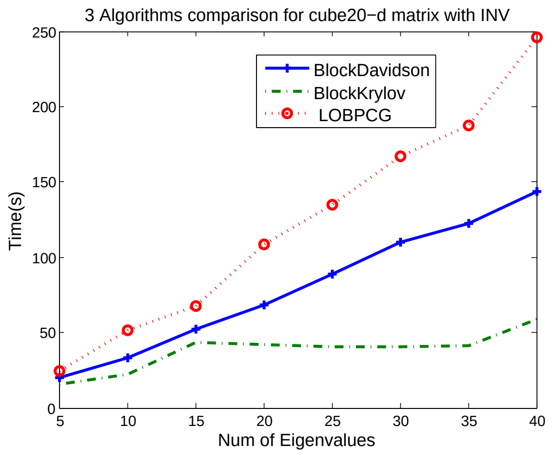

Figure 5-27: Comparison of the 3 Algorithms for the cube20-d matrix pair with INV Preconditioner

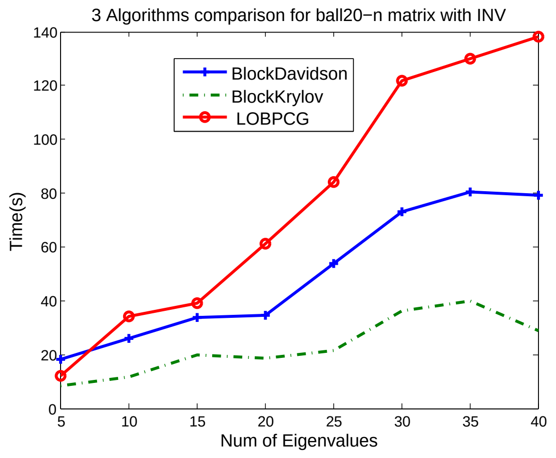

Figure 5-28: Comparison of the 3 Algorithms for the ball20-n matrix pair with INV Preconditioner