An Implementation of a Parallel Rational Krylov Algorithm

Daniel Skoogh

Department of Computing Science
1996

An Implementation of a Parallel Rational Krylov Algorithm

Daniel Skoogh

Department of Computing Science
Göteborg University and
Chalmers University of Technology
Göteborg, Sweden
E-mail: skoogh@cs.chalmers.se
May, 1996

Department of Computing Science
1996

A dissertation for the
Licentiate Degree in Numerical Analysis at
Chalmers University of Technology

Department of Computing Science
S-412 96 Göteborg
ISBN 91-7197-326-5
CTH, Göteborg, 1996

Abstract

An implementation of a parallel rational Krylov method for the generalised matrix eigenvalue problem is discussed. The implementation has been done on a MIMD computer and a cluster of workstations. The Rational Krylov algorithm is an extension of the shifted and inverted Arnoldi method where several shifts are used to compute basis vectors for one subspace. In this parallel implementation, the different shifted matrices are factorised each on one processor and then the iteration vectors are generated in parallel.

Keywords: eigenvalues, eigenvectors, sparse, parallel, rational, Krylov, shift, invert, Arnoldi
AMS 1980 subject classification 65F15, 65F50, 65W05

Contents

Chapter 1

Introduction

Eigenproblems occur frequently in science and engineering. Examples of everyday eigenproblems are vibrations in a car and in a stretched string on a piano. Other areas where eigenproblems occur are hydrodynamic stability, magnetohydrodynamics, Ising spin model, modes in a waveguide, economics and operation research, just to name a few. For an overview of eigenproblems see the paper by Bai, Day, Demmel and Dongarra [4].

Example 1. The modelling of different modes of a Transverse Magnetic wave (TM) travelling in the z-direction in a hollow-waveguide with arbitrary cross-section is done by the differential equation

$$-\nabla^2 e_z = k_c^2 e_z, \; e_z = 0 \; on \; the \; boundary$$

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

where the eigenfunction e_z is related to the electric field in the z-direction by $E_z = e_z e^{j\beta z}$, β is the propagation constant. The eigenvalue k_c is related to the cutoff frequency f_c by the relation

$$f_c = \frac{c \cdot k_c}{2\pi}$$

where c is the speed of light. The cutoff frequency f_c is the lowest frequency the mode corresponding to k_c propagates. □

The solution of a practical eigenproblem can be composed into three subtasks, modelling of the problem, discretisation of the model and solving the discrete problem. This report deals with the last subtask.

Modelling

Mathematical modelling deals with how to describe reality in mathematical terms, for example setting up the differential equations for the waveguide.

Discretisation

The discretisation process consists of reducing the continuous problem into a matrix problem by approximating the required unknown function by a vector representing the value of the function at a discrete set of points. Common discretisation processes are the finite difference and the finite element methods. They are used to discretise differential equations and can be used to discretise the waveguide problem. The matrices that the finite difference and the finite element methods generate are sparse i.e most elements are zero. The matrices often become large when they approximate complicated continuous problems in order to make a good approximation.

Solving the Discrete Problem

When we have obtained a discrete problem, we need to put it into a matrix problem that can be solved with a computer and find an algorithm that solves it. This report deals with an algorithm, the parallel rational Krylov algorithm, that solves the matrix eigenproblem with a parallel computer. The algorithm is not dependent of the origin of the matrices. They can come from a discretisation of a continuous problem, be random numbers or model a discrete problem like an electric power network. However the algorithm is more or less suitable for different types of problems.

Parallel Computers

A parallel computer is a computer that can use several processors to work together on the same problem. The main reasons to use a parallel computer instead of a sequential computer are to solve existing problems faster or to solve larger problems. The type of parallel computer that we have used is a computer where each processor has its own memory and the processors communicate through a network. The processors are relatively fast and have relatively large memory.

Eigenproblems

The kind of eigenproblems that can be solved with the parallel rational Krylov algorithm discussed in this report is

$$\boldsymbol{A}\boldsymbol{u} = \lambda \boldsymbol{B}\boldsymbol{u},\ \boldsymbol{A}, \boldsymbol{B} \in \mathcal{C}^{n\times n},\ \boldsymbol{u} \in \mathcal{C}^{n},\ \lambda \in \mathcal{C} \tag{1.1}$$

The algorithm is suitable for eigenproblems where several eigenvalues in a specific region in the complex plane are desired, together with the corresponding eigenvectors, and where the matrices are large and sparse.

The rational Krylov algorithm is an extension to the shift and invert Arnoldi algorithm. A good introduction to large eigenvalue computation and to the Arnoldi method is given in the book by Saad [17]. An implementation of the shift and invert Arnoldi is given in the paper by Kooper, Van Der Vorst and Goedbloed [10]. A parallel Arnoldi method is implemented by Booten, Meijer, te Riele and Van Der Vorst [5].

The Parallel Rational Krylov Algorithm

Let p shifts $\mu_1, \ldots, \mu_p$ be given in a region in the complex plane, where we want eigenvalues and corresponding eigenvectors. Construct an orthonormal basis $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m$ for the space

$$\begin{aligned}\mathcal{K}_m \equiv span\{\boldsymbol{v}_1, &(\boldsymbol{A}-\mu_1\boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{v}_1, \ldots, ((\boldsymbol{A}-\mu_1\boldsymbol{B})^{-1}\boldsymbol{B})^{m_1}\boldsymbol{v}_1,\\ &(\boldsymbol{A}-\mu_2\boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{v}_1, \ldots, ((\boldsymbol{A}-\mu_2\boldsymbol{B})^{-1}\boldsymbol{B})^{m_2}\boldsymbol{v}_1,\\ &\ldots\\ &(\boldsymbol{A}-\mu_s\boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{v}_1, \ldots, ((\boldsymbol{A}-\mu_s\boldsymbol{B})^{-1}\boldsymbol{B})^{m_s}\boldsymbol{v}_1\},\end{aligned}$$

where m_i are positive integers, $m = m_1 + m_2 + \ldots + m_s + 1$ and $Dim(\mathcal{K}_m) \leq m$. Approximate eigenvalues and eigenvectors are computed from the restriction of the matrix pencil (1.1) to the subspace $\mathcal{K}_m$.

There are two different major ways to parallelise the rational Krylov algorithm. The first one is to parallelise the matrix operation $\boldsymbol{r} = (\boldsymbol{A}-\mu_i\boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{v}_j$ and the process of orthogonalising $\boldsymbol{r}$ against the basis. This has been done for the Arnoldi process. This report deals with a second way that uses different shifts in the matrix operator $\boldsymbol{r} = (\boldsymbol{A}-\mu_i\boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{v}_j$ on different processors as follows.

Perform p factorisations in parallel, one on each processor i.

$$\boldsymbol{L}_i\boldsymbol{U}_i = (\boldsymbol{A}-\mu_i\boldsymbol{B}),\ i = 1, \ldots, p$$

Then let processor number i perform the multiplications

$$\boldsymbol{r}_{i+1} = \boldsymbol{U}_i^{-1}\boldsymbol{L}_i^{-1}\boldsymbol{B}\boldsymbol{v}_1$$

and later

$$\boldsymbol{r}_{j+p} = \boldsymbol{U}_i^{-1}\boldsymbol{L}_i^{-1}\boldsymbol{B}\boldsymbol{v}_j,\ j = qp + i + 1,\ q = 0, 1, \ldots$$

$\boldsymbol{r}_{j+p}$ is orthogonalised against all previous basis vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{j+p-1}$ either on the same processor or on a master processor. The remaining part is normalised and put into $\boldsymbol{v}_{j+p}$. In the orthogonalisation process, the different processors need to exchange data with each other.

The rational Krylov algorithm is suited for medium sized problems and the algorithm discussed is intended for problems where speed is more important than to solve the largest possible problem that can fit onto a parallel computer.

Summary

The objective of this work is to implement a parallel rational Krylov method, as described in the second way above, and to analyse speedup and convergence performance. In order to solve very large problems with the algorithm, we need to combine the two different ways above of parallelisation the algorithm. However this is outside the scope of this report.

The implementation is experimental, and is not yet intended for distribution. More work needs to be done in analysing the numerical stability and in the area of parallel sparse general LU factorisation. Shift strategies are an important part of a commercial or public domain software of the (parallel) rational Krylov method. However this report does not deal with shift strategies.

With the best implementation of the algorithm we got a speedup of 5.0 using 6 processors, see chapter 5.

The parallel rational Krylov method as implemented in this report has some numerical problems that the sequential version does not have, these numerical problems need to be investigated further, see the tests.

The (parallel) rational Krylov method is likely to perform better than the shifted and inverted Arnoldi method when the region with the desired eigenvalues differs to a great extent from the typical circular convergence region of the shift and invert Arnoldi method.

Outline

In chapter 2 we introduce basic concepts about parallel computers and parallel algorithms that we need for the discussion of the parallel rational Krylov method. In chapter 3 we first discuss subspace methods for eigenvalue computations and then we go on to the Arnoldi method. The Arnoldi method is a subspace method that gives the basic idea for the rational Krylov method. In the rest of the chapter we discuss the rational Krylov method and the parallel rational Krylov method. In chapter 4 we give two different major idea programs of the parallel rational Krylov algorithm and discuss some implementation details. In chapter 5 we give the test results of the aspects speed performance and numerical behaviour. We also compare the (parallel) rational Krylov method to the shift and invert Arnoldi method. At the end of the chapter we discuss some possible shift strategies.

Notation

Matrices are written with upper case bold letters like $\boldsymbol{A}$, vectors are written with lower case bold letters like $\boldsymbol{v}$ and scalars are written with lower case italic letters like μ. With $h_{i,j}$ we mean the element $\boldsymbol{H}(i,j)$ and with $\boldsymbol{h}_j$ we mean the j:th column of $\boldsymbol{H}$. With $\boldsymbol{V}_j$ we mean the first j columns of the matrix $\boldsymbol{V}$.

Acknowledgements

First, I would like to thank my supervisor Axel Ruhe for introducing me to eigenvalue computation and the fruitful discussion behind this report. I would also like to thank my assistant adviser Thomas Ericsson for helping me with everything from finding bugs in my programs to discussion of the numerical behaviour of the algorithms in this report.

Other people have also influenced this report and I would like to thank them. Richard B. Lehoucq read an early version of the manuscript during a visit to the department and made some suggestions, Magnus Bondesson helped me with some issues in the chapter Parallel Computing and Setta Svensson corrected the English.

This work was partly supported by Swedish National Board for Industrial and Technical Development, grant 8902538-5.

I am grateful to the Center for Parallel Computers, Royal Institute of Technology, Stockholm, Sweden for making the parallel computer IBM-SP2 available for computations.

Chapter 2

Parallel Computing

Before we discuss the parallel eigenvalue algorithms, we need some basic understanding of parallel computers and algorithms.

2.1 Introduction

The need to solve larger problems or decrease the execution time for existing ones will always demand more powerful computers. One way to satisfy the needs is to build faster one-processor machines. Since the first computer was built there has been a tremendous increase in computing power, however we will sooner or later reach the physical limit of how fast one-processor can run. If we wish to increase the performance with existing processor technology, one way to do it is to use several processors at the same time i.e. parallel computers. In a parallel computer several processors are put together in such a way that they can work independently and exchange data with each other. In this report we are mainly interested in parallel algorithms. With a parallel algorithm we mean an algorithm that can be divided into several sub-parts that can be treated in parallel. The sub-parts may have to exchange information with each other at certain points of time.

In order to discuss parallel algorithms we need some basic knowledge of parallel computers. We look at three different ways of classifying computers; instruction and data streams, granularity of the operations and memory localisation.

2.1.1 Instruction and Data Streams

The most common way to describe computer architectures is Flynn's taxonomy see [7, 6]. The classification is based on the number of instructions and data streams that can be processed simultaneously.

- (SISD) Single Instruction stream -Single Data stream
- (SIMD) Single Instruction stream -Multiple Data stream
- (MISD) Multiple Instruction stream -Single Data stream
- (MIMD) Multiple Instruction stream -Multiple Data stream

SIMD and MIMD are the existing parallel computer types.

SISD

SISD is the usual one-processor computer.

SIMD

SIMD computers have only one instruction stream. The different processors execute the same operation at every moment of time, but each of them with different data. Not all parallel algorithms can be implemented on an SIMD successfully. An SIMD computer usually has several thousands of processors. One example of a problem that can use an SIMD is matrix addition, $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$. First $\boldsymbol{A}$ and $\boldsymbol{B}$ are distributed over the processors in a such a way that for each index pair i, j $a_{i,j}$ and $b_{i,j}$ are located on the same processor, then a particular processor just adds the part of $\boldsymbol{A}$ and $\boldsymbol{B}$ that belongs to its domain, $c_{i,j} = a_{i,j} + b_{i,j}$. Examples of SIMD computers are the Connection Machines, CM-1 and CM-2. Another SIMD computer is the Maspar MP-1. For information about these computers see [18].

MISD

No computer has ever been build after this model according to [18].

MIMD

MIMD is the most flexible computer type. Each processor runs its own process/processes and communicates with the other processors when needed. An example of an MIMD computer is the IBM-SP2. An MIMD computer usually has fewer processors than an SIMD machine but each of them is most often much faster.

2.1.2 Granularity

A categorisation of the inherent parallelism of an algorithm is the grain size. Typically the grain size can be divided into fine grain, medium grain and coarse grain parallelism according to if units, tens or hundreds and more operations are performed between each communication point. SIMD computers are most suited for fine grain parallelism, while MIMD computers are most suited for coarse grain parallelism. The level of granularity a computer is suited for depends on how much time each operation takes compared with how much time it takes to exchange data with other processors.

The algorithms discussed in this report are relatively coarse grain and they are implemented on an MIMD computer (IBM-SP2), so we will concentrate on MIMD computers.

2.1.3 Memory Localisation

MIMD computers (and SIMD) can be divided into two major subclasses

- Shared memory computers

- Message passing computers

Shared Memory

The processors share the main memory, but usually each processor has its own cache. The communication is done through the shared main memory.

Message Passing

Each processor has its own memory. Communication is done through messages that are sent on an interconnecting network. This is the subclass on which we will concentrate.

2.1.4 Parallel Virtual Machine

Several heterogeneous computers can be connected through a network and act as a parallel computer. PVM (parallel virtual machine) is a software package that makes this possible, see [8]. PVM uses messages to exchange data between processes, and the two programming languages C and Fortran are supported by PVM. Several Parallel MIMD computers also support PVM as a way to exchange data between processors.

The main difference, between a parallel MIMD computer with distributed memory and a cluster of workstations connected through a standard network acting as a parallel computer, is the speed of communication between the processors. A parallel computer has a fast dedicated network with much higher performance than a standard network.

The algorithms in this report are implemented using PVM and Fortran. The programs run on an IBM-SP2, a MIMD parallel computer with distributed memory and a cluster of SUN workstations connected through an Ethernet.

2.2 Measuring Program Performance

If we implement a parallel algorithm using p processors, the best we can hope for is a program that runs p times faster than if it was to run on a single processor. However this is seldom reached in practice. Below we will discuss some performance measures and why the optimum speed is rarely reached.

Speedup

The speedup S_p on p processors, is given by

$$S_p = T_1/T_p,$$

where T_1 is the execution time for the sequential program on one processor and T_p is the execution time for the parallel program on p processors. The theoretical peak performance is $S_p = p$.

Efficiency

The Efficiency E_p is given by

$$E_p = 100\frac{S_p}{p}\%$$

The theoretical peak performance is $E_p = 100\%$.

The main reasons why E_p and S_p rarely reach their peak values are

- it takes time to transfer data between processors
- all processors do not have the same load
- parts of the program have no natural parallelism in them

Communication Time

The total time it takes to transfer b bytes from one processor to another is

$$t = L + b/B,$$

where L is the latency, that is the time it takes to set up the communication link. The bandwidth B is the rate of transfer in byte per seconds after a communication link has been established.

Example 2. The process of sending a message with PVM is composed into the sub-parts of initiating a sending buffer, packing data into the buffer and sending the buffer. An arriving message is first stored into a temporary buffer and is later unpacked by the program that receives the message.

To estimate the communication speed we note that on the IBM-SP2 the whole process of sending a double precision number from processor A to processor B and back to A again with PVMe (IBMs version of PVM) takes 400μs, just the send operation takes 42μs. The same procedure with a double precision array of length 10000 takes 12ms and 3.3ms respectively.

The computation speed on the other hand can be seen from that the calculation of the inner product between two vectors of length 10000 takes 870μs on one processor. The computation speed varies depending on how data is transferred between different memory hierarchy.

The latency L can be approximated by 42μs, that is the time for the send operation for one double precision number. The bandwidth B for a double precision vector of length $n = 10000$ is $8 \cdot 10000/6 \cdot 10^{-3} = 13.3$M byte per second, or differently expressed $13.3/8 = 1.7$ M double precision numbers per second. The number of floating point operations per second (flops) for the scalar product is $2 \cdot 10000/870 \cdot 10^{-6} = 23$M flops. In this case the compute communicate ratio is $23/1.7 = 13.5$

The above example shows that in order to write efficient parallel programs on the IBM-SP2 and similar computers, the number of floating point numbers sent from one processor to another should be much smaller than the number of floating point operations.

Load Balancing

In order to achieve high efficiency in a parallel program, it is important to keep the time spent waiting for data from other processors to a minimum and the time spent doing useful work to a maximum. The process of allocating work to processors in such a way as to keep the processors' idle time as short as possible, is called load balancing.

Sequential Parts of a Program

Not all algorithms can be successfully implemented on a parallel computer because a large portion of the algorithm has no parallelism in it. Even parallel programs may have parts in them that are not parallel, for example control sequences. Some parallel parts in a parallel algorithm may be more efficiently implemented sequentially due to communication time. Amdahl's law gives a relation between the speedup S_p, the parallel part r i.e execution time of the program and the sequential part s of the program as follows

$$S_p \leq \frac{1}{s + \frac{r}{p}},\ s + r = 1$$

where p is the number of processors.

Example 3. If the sequential part $s = 0.5$, the parallel part $r = 0.5$ and the the number of processors $p = 5$ then

$$S_5 \leq \frac{1}{0.5 + \frac{0.5}{5}} = 1.67$$

The same procedure with $s = 0.1$, $r = 0.9$ and $p = 5$ gives

$$S_5 \leq \frac{1}{0.1 + \frac{0.9}{5}} = 3.57$$

In order for an algorithm to be successfully implemented on a parallel computer the sequential part s needs to be much smaller than the parallel part r.

For a more complete discussion on Parallel computers and parallel algorithms than given above, see for example [7, 6].

Chapter 3

Krylov Subspace Methods

3.1 Subspace Methods

In a subspace method for the eigenproblem,

$$\boldsymbol{A}\boldsymbol{u} = \lambda \boldsymbol{u}, \tag{3.1}$$

we seek an approximate eigenpair $(\tilde{\lambda}, \tilde{\boldsymbol{u}})$ in such a way that the approximate eigenvector belongs to a subspace S and the difference between the approximate eigenpair $(\tilde{\lambda}, \tilde{\boldsymbol{u}})$ and the correct eigenpair $(\lambda, \boldsymbol{u})$ is small by some measure.

Let $\boldsymbol{A} \in \mathcal{C}^{n \times n}$, S be a subspace of $\mathcal{C}^n$, $S \subseteq \mathcal{C}^n$ and $\boldsymbol{x}_i, i = 1, \ldots, m$ be a basis of S. Now let S be invariant under $\boldsymbol{A}$,

$$\boldsymbol{x} \in S \Rightarrow \boldsymbol{A}\boldsymbol{x} \in S.$$

We seek an eigenvector, eigenvalue pair such that

$$\boldsymbol{A}\boldsymbol{u} = \lambda \boldsymbol{u}, \boldsymbol{u} \in S, \lambda \in \mathcal{C}.$$

Especially if we let $\boldsymbol{A}$ operate on the vectors that build up the basis of S

$$\boldsymbol{A}\boldsymbol{x}_j = \sum_{i=1}^{m} \boldsymbol{x}_i h_{ij},$$

and if we take

$$\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m], \boldsymbol{H} = [h_{ij}],$$

we will get

$$\boldsymbol{A}\boldsymbol{X} = \boldsymbol{X}\boldsymbol{H}. \tag{3.2}$$

If $(\lambda, \boldsymbol{y})$ is an eigenpair of $\boldsymbol{H}$ then $(\lambda, \boldsymbol{X}\boldsymbol{y})$ is an eigenpair of $\boldsymbol{A}$. Multiply (3.2) by $\boldsymbol{y}$

$$\begin{aligned} \boldsymbol{A}(\boldsymbol{X}\boldsymbol{y}) &= \boldsymbol{X}\boldsymbol{H}\boldsymbol{y} \\ &= \lambda(\boldsymbol{X}\boldsymbol{y}). \end{aligned}$$

In general, in a numerical subspace method we do not have invariant subspaces. So the relation (3.2) does not hold. Instead we get

$$\boldsymbol{W} = \boldsymbol{A}\boldsymbol{X} - \boldsymbol{X}\boldsymbol{H}, \tag{3.3}$$

where $\boldsymbol{W}$ is the residual matrix. Thus if $(\tilde{\lambda}, \tilde{\boldsymbol{y}})$ is an eigenpair of $\boldsymbol{H}$ then $(\tilde{\lambda}, \boldsymbol{X}\tilde{\boldsymbol{y}})$ is usually not an eigenpair of $\boldsymbol{A}$. In designing a subspace method we want both the error in the eigenvalue $\| \lambda - \tilde{\lambda} \|$ and the angle to the eigenvector $\min_{\boldsymbol{u} \in S_\lambda} (\| \boldsymbol{u} - \tilde{\boldsymbol{u}} \| / \| \boldsymbol{u} \|)$ to be small. Here S_λ denotes the eigenspace corresponding to λ. Usually we can not measure these quantities in a good way. On the other hand we can take $\tilde{\boldsymbol{u}} = \boldsymbol{X}\tilde{\boldsymbol{y}}$ as the approximate eigenvector, and get the residual vector $\boldsymbol{w}$

$$\begin{aligned} \boldsymbol{w} &= \boldsymbol{A}\tilde{\boldsymbol{u}} - \tilde{\lambda}\tilde{\boldsymbol{u}} \\ &= \boldsymbol{A}\boldsymbol{X}\tilde{\boldsymbol{y}} - \boldsymbol{X}\boldsymbol{H}\tilde{\boldsymbol{y}} \\ &= \boldsymbol{W}\tilde{\boldsymbol{y}}, \end{aligned}$$

and see if we can get its scaled length $\| \boldsymbol{w} \| / \| \boldsymbol{u} \|$ to be small.

3.1.1 Projection

A projection matrix $\boldsymbol{P} \in \mathcal{C}^{n \times n}$ onto a subspace $S \subseteq \mathcal{C}^n$ satisfies

$$\boldsymbol{P}^2 = \boldsymbol{P}, \ \boldsymbol{P}\boldsymbol{x} \in S \ \forall \boldsymbol{x} \in \mathcal{C}^n, \tag{3.4}$$

Every vector $\boldsymbol{x} \in \mathcal{C}^n$ can be composed into one part in S and one in its complement.

$$\boldsymbol{x} = \boldsymbol{P}\boldsymbol{x} + (\boldsymbol{I} - \boldsymbol{P})\boldsymbol{x}$$

An orthogonal projection $\boldsymbol{P}$ satisfies in addition to (3.4) the following condition

$$(\boldsymbol{I} - \boldsymbol{P})\boldsymbol{x} \in S^-$$

Proposition 1. *A projector is orthogonal if and only if it is hermitian.*

For proof see [17]. Assume that $m = Dim(S)$, $m \leq n$ and that $\boldsymbol{v}_1, \ldots \boldsymbol{v}_m$ are orthonormal vectors such that $\boldsymbol{v}_i \in \mathcal{C}^n$ and

$$S = span\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m\},$$

then $\boldsymbol{P}$ can be written as

$$\boldsymbol{P} = \boldsymbol{V}_m \boldsymbol{V}_m^*,$$

where $\boldsymbol{V}_m = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m]$.

Proposition 2. *Let $\boldsymbol{P}$ be an orthogonal projector onto the subspace S, $x \in \mathcal{C}^n$ and $y \in S$, then*

$$\| \boldsymbol{x} - \boldsymbol{P}\boldsymbol{x} \|_2 = \min_{y \in S} \| \boldsymbol{x} - \boldsymbol{y} \|_2$$

Projections are often used to measure how close an eigenvector $\boldsymbol{u}$ is to the subspace S.

3.1.2 Subspace Methods

We will now give a basic subspace method. Consider the eigenproblem (3.1) and a subspace S of dimension m. We want to find approximate eigenvectors $\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \in S, i = 1, \ldots, m$ and corresponding eigenvalues $\tilde{\lambda}_i^{(m)}, i = 1, \ldots, m$, usually $m \ll n$ where n is the dimension of the matrix $\boldsymbol{A}$.

Let $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m$ be an orthonormal basis for S and $\boldsymbol{V}_m = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m]$. Now we want the approximate eigenpairs $(\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})}, \tilde{\lambda}_i^{(m)}), i = 1, \ldots, m$ to satisfy

$$(\boldsymbol{A} - \tilde{\lambda}_i^{(m)} \boldsymbol{I}) \tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} - S, \ \tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \in S, i = 1, \ldots, m,$$

or with matrix notation,

$$\boldsymbol{V}_m^* (\boldsymbol{A} - \tilde{\lambda}_i^{(m)} \boldsymbol{I}) \tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} = 0. \tag{3.5}$$

This can also be expressed with a projection $\boldsymbol{P}_m = \boldsymbol{V}_m \boldsymbol{V}_m^*$ as

$$\boldsymbol{P}_m (\boldsymbol{A} - \tilde{\lambda}_i^{(m)} \boldsymbol{I}) \tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} = 0.$$

Now $\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \in S$ so $\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})}$ can be written as a linear combination of the basis vectors $\boldsymbol{v}_i$

$$\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} = \boldsymbol{V}_m \tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \tag{3.6}$$

Put (3.6) in (3.5) and get

$$\boldsymbol{V}_m^* (\boldsymbol{A} \boldsymbol{V}_m \tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} - \tilde{\lambda}_i^{(m)} \boldsymbol{V}_m \tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})}) = 0$$

or

$$\boldsymbol{V}_m^* \boldsymbol{A} \boldsymbol{V}_m \tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} = \tilde{\lambda}_i^{(m)} \tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})}.$$

or

$$\boldsymbol{H} \tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} = \tilde{\lambda}_i^{(m)} \tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})},$$

where

$$\boldsymbol{H} = \boldsymbol{V}_m^* \boldsymbol{A} \boldsymbol{V}_m.$$

This leads up to the simple algorithm

Algorithm 1. *(basic idea)*

1. Compute the orthonormal basis $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m$
2. Compute $\boldsymbol{H} = \boldsymbol{V}_m^* \boldsymbol{A} \boldsymbol{V}_m$
3. Compute eigenpairs $(\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})}, \tilde{\lambda}_i^{(m)}), i = 1, \ldots, m$ to $\boldsymbol{H}$
4. Take $\tilde{\lambda}_i^{(m)}$ as approximate eigenvalue and $\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} = \boldsymbol{V}_m \tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})}$ as approximate eigenvector to $\boldsymbol{A}$.

The above algorithm should only be considered as a basic idea, not for numerical computation.

3.2 Krylov Subspaces

Krylov subspace methods are built around the sequence of subspaces,

$$\mathcal{K}_m \equiv span\{\boldsymbol{v}, \boldsymbol{A}\boldsymbol{v}, \boldsymbol{A}^2\boldsymbol{v}, \ldots, \boldsymbol{A}^{m-1}\boldsymbol{v}\} \tag{3.7}$$

The Krylov methods calculate approximate eigenvectors $\tilde{\boldsymbol{u}}_i \in \mathcal{K}_m, i = 1, \ldots, m$ and corresponding approximate eigenvalues $\tilde{\lambda}_i, i = 1, \ldots, m$, to the eigenvalue problem

$$\boldsymbol{A}\boldsymbol{u} = \lambda\boldsymbol{u}, \boldsymbol{A} \in \mathcal{C}^{n\times n}, \boldsymbol{u} \in \mathcal{C}^n, \lambda \in \mathcal{C} \tag{3.8}$$

Some of the Krylov subspace methods are

1. The Arnoldi method
2. The Hermitian Lanczos method

The rational Krylov method uses a variation of the subspace (3.7). We will discuss this later.

In the following we state some properties about the Krylov subspaces. A more complete discussion can be found in Saad [17].

Proposition 3. *A vector $\boldsymbol{x} \in \mathcal{K}_m$ can be written as $\boldsymbol{x} = p(\boldsymbol{A})\boldsymbol{v}$, where p is a polynomial of degree not exceeding m-1.*

Proposition 4. *Let μ be the lowest degree of a polynomial p such that $p(\boldsymbol{A})\boldsymbol{v} = 0$. Then $\mathcal{K}_\mu$ is invariant under $\boldsymbol{A}$ and $\mathcal{K}_m = \mathcal{K}_\mu$ for all $m \geq \mu$.*

This means that $\mathcal{K}_\mu$ is the largest possible Krylov subspace starting with $\boldsymbol{v}$. The vectors $\boldsymbol{v}, \boldsymbol{A}\boldsymbol{v}, \boldsymbol{A}^2\boldsymbol{v}, \ldots, \boldsymbol{A}^{\mu-1}\boldsymbol{v}$ are linearly independent, but $\boldsymbol{v}, \boldsymbol{A}\boldsymbol{v}, \boldsymbol{A}^2\boldsymbol{v}$, $\ldots, \boldsymbol{A}^{m-1}\boldsymbol{v}$ will be dependent for any $m > \mu$. The subspace $\mathcal{K}_\mu$ is invariant and all approximations to eigenvectors from $\mathcal{K}_\mu$ will be exact. Any Krylov algorithm will have to stop before linear dependence is reached among the basis vectors.

3.3 Arnoldi's Method

Krylov proposed his method already in 1931, but the most notable works were done by Lanczos [11] and Arnoldi [3] around 1950. Their methods build up an orthonormal basis $\boldsymbol{V}_m = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m]$ for the subspace $\mathcal{K}_m$ one vector at a time. In each step the matrix is applied to the most recent basis vector, and then the result is orthogonalised to those vectors that have already been computed.

Algorithm 2. Arnoldi

I **Start**. Choose a vector $\boldsymbol{v}_1$ of norm 1.

II **for** $j = 1, 2, \ldots, m$ **do**

1. $\boldsymbol{r} := \boldsymbol{A}\boldsymbol{v}_j$ — Operate
2. $h_{ij} := \boldsymbol{v}_i^*\boldsymbol{r}, i = 1, 2, \ldots, j$
3. $\boldsymbol{r} := \boldsymbol{r} - \sum_{i=1}^{j} h_{ij}\boldsymbol{v}_i$, — Orthogonalise
4. $h_{j+1,j} := \| \boldsymbol{r} \|_2$ — Normalise
5. **if** $h_{j+1,j} = 0$ **stop**
6. $\boldsymbol{v}_{j+1} := \boldsymbol{r}/h_{j+1,j}$ — Get new vector
7. Compute approximate solution and test for convergence

By construction, the vectors $\boldsymbol{v}_j$ are orthonormal. They also span $\mathcal{K}_m$.

Let $\boldsymbol{V}_m = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m]$ and $\boldsymbol{H}_{m,m} = [h_{ij}], i, j = 1, \ldots, m$ be computed by this algorithm. Then the relation

$$\boldsymbol{A}\boldsymbol{V}_m = \boldsymbol{V}_m\boldsymbol{H}_{m,m} + h_{m+1,m}\boldsymbol{v}_{m+1}\boldsymbol{e}_m^* \tag{3.9}$$

holds. If $h_{m+1,m} = 0$ then the vectors $\boldsymbol{v}_j$ span an invariant subspace under $\boldsymbol{A}$. Thus if $(\lambda, \boldsymbol{y})$ is an eigenpair of $\boldsymbol{H}$then $(\lambda, \boldsymbol{V}_m\boldsymbol{y})$ is an eigenpair of $\boldsymbol{A}$.

In general, we stop before we get an invariant subspace. In step II.7 the $m \times m$ eigenproblem

$$\boldsymbol{H}_{m,m}\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} = \tilde{\lambda}_i^{(m)}\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})}$$

is solved and we take

$$\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} = \boldsymbol{V}_m\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \tag{3.10}$$

as the approximate eigenvector. Now we get the residuals

$$\begin{aligned}
(\boldsymbol{A} - \tilde{\lambda}_i^{(m)}\boldsymbol{I})\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} &= \boldsymbol{A}\boldsymbol{V}_m\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} - \tilde{\lambda}_i^{(m)}\boldsymbol{V}_m\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \\
&= \boldsymbol{V}_m\boldsymbol{H}_{m,m}\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} + h_{m+1,m}\boldsymbol{e}_m^*\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})}\boldsymbol{v}_{m+1} \\
&\quad - \tilde{\lambda}_i^{(m)}\boldsymbol{V}_m\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \\
&= h_{m+1,m}\boldsymbol{e}_m^*\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})}\boldsymbol{v}_{m+1}.
\end{aligned}$$

The first equality follows from (3.10), the second from (3.9), and the third is a consequence of that $\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})}$is an eigenvector of $\boldsymbol{H}_{m,m}$.

3.3.1 Invariant Subspace

Under what condition is the subspace $\mathcal{K}_m$ invariant? Assume that the matrix $\boldsymbol{A}$ is diagonalisable, then

$$\boldsymbol{v} = \sum_{i=1}^{m}\alpha_i\boldsymbol{u}_i, \ \| \boldsymbol{v} \|_2 = 1$$

where the vectors $\boldsymbol{u}_i, i = 1, \ldots, m$ are eigenvectors to the matrix $\boldsymbol{A}$. The Krylov subspace $\mathcal{K}_m$ can be written as

$$\mathcal{K}_m \equiv span\{\sum_{i=1}^{m}\alpha_i\boldsymbol{u}_i, \sum_{i=1}^{m}\alpha_i\lambda_i\boldsymbol{u}_i, \sum_{i=1}^{m}\alpha_i\lambda_i^2\boldsymbol{u}_i, \ldots, \sum_{i=1}^{m}\alpha_i\lambda_i^{m-1}\boldsymbol{u}_i\}$$

Suppose first that the eigenvalues $\lambda_i, i = 1 \ldots, m$ to the corresponding eigenvectors $\boldsymbol{u}_i, i = 1, \ldots, m$ are all distinct. Then the eigenvectors $\boldsymbol{u}_i, i = 1, \ldots, m$ span the Krylov subspace $\mathcal{K}_m$ of dimension m and the subspace $\mathcal{K}_m$ is invariant under $\boldsymbol{A}$.

Suppose now that we have two eigenvectors $\boldsymbol{u}_{m-1}$ and $\boldsymbol{u}_m$ to the same eigenvalue $\lambda_{m-1} = \lambda_m$ then

$$\alpha_{m-1}\lambda_{m-1}^k\boldsymbol{u}_{m-1} + \alpha_m\lambda_m^k\boldsymbol{u}_m = \lambda_{m-1}^k(\alpha_{m-1}\boldsymbol{u}_{m-1} + \alpha_m\boldsymbol{u}_m)$$

and $\alpha_{m-1}\boldsymbol{u}_{m-1} + \alpha_m \boldsymbol{u}_m$ is also an eigenvector to the matrix $\boldsymbol{A}$ with the eigenvalue $\lambda_{m-1} = \lambda_m$. In this case the subspaces $\mathcal{K}_{m-1}$ and $\mathcal{K}_m$ are equal and the dimension of the subspace $\mathcal{K}_{m-1}$ is $m-1$. Furthermore the eigenvectors $u_i, i = 1, \ldots, m-2$ and the eigenvector $\alpha_{m-1}\boldsymbol{u}_{m-1} + \alpha_m \boldsymbol{u}_m$ span the subspace $\mathcal{K}_{m-1}$ and $\mathcal{K}_{m-1}$ is invariant under $\boldsymbol{A}$. This shows that it is only possible to obtain one eigenvector to a eigenvalue with the Arnoldi method. This is also true for the related methods rational Krylov method and shift and invert Arnoldi method discussed later.

In the non diagonalisable case we may get principal vectors of higher grade, but we will only get one eigenvector.

3.3.2 Convergence of Arnoldi

In the following subsection we use projection operators as means to measure the distance between an eigenvector $\boldsymbol{u}_i$ and the Krylov subspace $\mathcal{K}_m$. In the hermitian case these bound the errors in the eigenvalues and eigenvectors, but also in the general case they are used to measure convergence see [17].

Proposition 5. *Let $\boldsymbol{P}_m$ be the orthogonal projector onto $\mathcal{K}_m$ i.e. $\boldsymbol{P}_m = \boldsymbol{V}_m \boldsymbol{V}_m^*$. Assume that $\boldsymbol{A}$ is diagonalisable and that the vector $\boldsymbol{v}_1$ in Arnoldi's method can be written as $\boldsymbol{v}_1 = \sum_{k=1}^{k=m} \alpha_k \boldsymbol{u}_k$, where $\boldsymbol{u}_k$ are normalised eigenvectors of $\boldsymbol{A}$ in which $\alpha_i \neq 0$. Then the following inequality holds*

$$\| \, (\boldsymbol{I} - \boldsymbol{P}_m)\boldsymbol{u}_i \, \|_2 \leq \xi_i \epsilon_i^{(m)}$$

where

$$\xi_i = \sum_{k=1, k\neq i}^{n} \frac{|\alpha_k|}{|\alpha_i|}$$

and

$$\epsilon_i^{(m)} \equiv \min_{p \in \mathcal{P}^*_{m-1}} \max_{\lambda \in \sigma(A) - \lambda_i} |p(\lambda)|.$$

where $\mathcal{P}^*_{m-1}$ is the set of all polynomials with a degree less than or equal to $m-1$ such that $p(\lambda_i) = 1$. For proof see [17].

An estimate for $\epsilon_1^{(m)}$ is given below.

Proposition 6. *Let $m < n$. Then there exist m eigenvalues of A which can be labelled $\lambda_2, \lambda_3, \ldots, \lambda_{m+1}$ such that*

$$\epsilon_1^{(m)} = \left(\sum_{j=2}^{m+1} \prod_{k=2, k\neq j}^{m+1} \frac{|\lambda_k - \lambda_1|}{|\lambda_k - \lambda_j|} \right)^{-1}$$

For proof see [17].

If λ_1 is in the outer part of the spectrum then $|\lambda_k - \lambda_1|$ will generally be larger than $|\lambda_k - \lambda_j|$. Thus $\epsilon_1^{(m)}$ will be small compared to the average of $\epsilon_i^{(m)}$ and $\tilde{\lambda}_i^{(m)}$ will converge faster than most eigenvalues.

3.3.3 Shift And Invert Arnoldi

If we substitute the matrix $\boldsymbol{A}$ by the shifted and inverted matrix $(\boldsymbol{A}-\mu\boldsymbol{I})^{-1}$ in the Arnoldi algorithm, the eigenvalues of $\boldsymbol{A}$ which are closest to μ will converge fastest.

$$\boldsymbol{A}\boldsymbol{x} = \lambda\boldsymbol{x}$$

$$(\boldsymbol{A}-\mu\boldsymbol{I})\,\boldsymbol{x} = (\lambda-\mu)\,\boldsymbol{x}$$

$$\frac{1}{(\lambda-\mu)}\boldsymbol{x} = (\boldsymbol{A}-\mu\boldsymbol{I})^{-1}\,\boldsymbol{x}$$

Thus if λ is an eigenvalue of $\boldsymbol{A}$ then $\frac{1}{\lambda-\mu}$ is an eigenvalue of $(\boldsymbol{A}-\mu\boldsymbol{I})^{-1}$. If λ is close to μ then $\frac{1}{\lambda-\mu}$ will be an eigenvalue in the outer part of the spectrum of $(\boldsymbol{A}-\mu\boldsymbol{I})^{-1}$ and thus converge fast.

3.4 Rational Krylov Algorithm

The rational Krylov method is a generalisation of the shift and invert Arnoldi method. In the shift and invert Arnoldi we choose one shift μ in the complex plane where we want the eigenvalues to converge fast, while in the rational Krylov we choose several shifts $\mu_1,\ldots,\mu_s$. The rational Krylov method was developed by Ruhe see [13, 14, 15, 16] . The space $\mathcal{K}_m$ in the rational Krylov method is

$$\begin{aligned}\mathcal{K}_m \equiv span\{&\boldsymbol{v}_1,(\boldsymbol{A}-\mu_1\boldsymbol{I})^{-1}\boldsymbol{v}_1,\ldots,(\boldsymbol{A}-\mu_1\boldsymbol{I})^{-m_1}\boldsymbol{v}_1,\\ &(\boldsymbol{A}-\mu_2\boldsymbol{I})^{-1}\boldsymbol{v}_1,\ldots,(\boldsymbol{A}-\mu_2\boldsymbol{I})^{-m_2}\boldsymbol{v}_1,\\ &\ldots\\ &(\boldsymbol{A}-\mu_s\boldsymbol{I})^{-1}\boldsymbol{v}_1,\ldots,(\boldsymbol{A}-\mu_s\boldsymbol{I})^{-m_s}\boldsymbol{v}_1\},\end{aligned} \tag{3.11}$$

where $m = m_1+m_2+\ldots+m_s+1$ and $Dim(\mathcal{K}_m)\le m$.

In Arnoldi's method every vector $\boldsymbol{x}\in\mathcal{K}_m$ can be expressed as

$$\boldsymbol{x} = p_j(\boldsymbol{A})\boldsymbol{v}_1 \quad , j\le m-1,$$

where p_j is a polynomial of degree j. But in the rational Krylov method every vector $\boldsymbol{x}\in\mathcal{K}_m$ can be written as

$$\boldsymbol{x} = r(\boldsymbol{A})\boldsymbol{v}_1$$

where

$$\begin{aligned}r(\lambda) =& \frac{p_j(\lambda)}{(\lambda-\mu_1)^{j_1}(\lambda-\mu_2)^{j_2}\ldots(\lambda-\mu_s)^{j_s}}\\ =& c_0+\sum_{k=1}^{j_1}\frac{c_{1,k}}{(\lambda-\mu_1)^k}+\sum_{k=1}^{j_2}\frac{c_{2,k}}{(\lambda-\mu_2)^k}\\ &+\ldots+\sum_{k=1}^{j_s}\frac{c_{s,k}}{(\lambda-\mu_s)^k},\end{aligned}$$

$j = j_1 + j_2 + \ldots + j_s$, $j_1 \leq m_1, j_2 \leq m_2, \ldots, j_s \leq m_s$.

In matrix analyses in general we have

$$\boldsymbol{B}\boldsymbol{A} \neq \boldsymbol{A}\boldsymbol{B}, \quad \boldsymbol{A}, \boldsymbol{B} \in \mathcal{C}^{n \times n},$$

but for different functions of the same matrix, note that

$$(\boldsymbol{A} - \mu_i \boldsymbol{I})^{-1}(\boldsymbol{A} - \mu_j \boldsymbol{I})^{-1} = (\boldsymbol{A} - \mu_j \boldsymbol{I})^{-1}(\boldsymbol{A} - \mu_i \boldsymbol{I})^{-1}$$

and

$$p(\boldsymbol{A})(\boldsymbol{A} - \mu_i \boldsymbol{I})^{-1} = (\boldsymbol{A} - \mu_i \boldsymbol{I})^{-1} p(\boldsymbol{A}).$$

Example 4.

$$\begin{aligned} r(\lambda) &= \frac{\lambda^2 - 3\lambda + 1}{(\lambda - 2)(\lambda - 3)} \\ &= \frac{1}{(\lambda - 2)} + \frac{1}{(\lambda - 3)} + 1 \\ r(A) &= (\boldsymbol{A}^2 - 3\boldsymbol{A} + \boldsymbol{I})(\boldsymbol{A} - 2\boldsymbol{I})^{-1}(\boldsymbol{A} - 3\boldsymbol{I})^{-1} \\ &= (\boldsymbol{A} - 2\boldsymbol{I})^{-1}(\boldsymbol{A} - 3\boldsymbol{I})^{-1}(\boldsymbol{A}^2 - 3\boldsymbol{A} + \boldsymbol{I}) \\ &= (\boldsymbol{A} - 2\boldsymbol{I})^{-1} + (\boldsymbol{A} - 3\boldsymbol{I})^{-1} + \boldsymbol{I} \end{aligned}$$

In creating a basis for $\mathcal{K}_m$ (3.11) it does not matter in which order the operators $(\boldsymbol{A} - \mu_1 \boldsymbol{I})^{-1}, \ldots, (\boldsymbol{A} - \mu_s)^{-1}$ are applied. This is the key to the parallel algorithm, but first we will discuss the sequential algorithm.

We will discuss the rational Krylov method for the generalised eigenproblem

$$\boldsymbol{A}\boldsymbol{u} = \lambda \boldsymbol{B}\boldsymbol{u} \tag{3.12}$$

In the case, $\boldsymbol{B}$ is invertible. We substitute $\boldsymbol{A}$ with $\boldsymbol{B}^{-1}\boldsymbol{A}$ in (3.11) and the operators become $(\boldsymbol{B}^{-1}\boldsymbol{A} - \mu_i \boldsymbol{I})^{-1} = (\boldsymbol{A} - \mu_i \boldsymbol{B})^{-1}\boldsymbol{B}$. Below we describe the sequential version of rational Krylov method for the generalised eigenproblem.

Algorithm 3. RKS 1

I **Start.** Choose a vector $\boldsymbol{v}_1$ of norm 1

II **for** $j = 1, 2, \ldots, m$ **do**

1. $\boldsymbol{r} := \boldsymbol{V}_j \boldsymbol{t}_j$ — Choose starting combination
2. $\boldsymbol{r} := (\boldsymbol{A} - \mu_j \boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{r}$ — Choose μ_j and Operate
3. $h_{ij} := \boldsymbol{v}_i^* \boldsymbol{r}, i = 1, 2, \ldots, j$
4. $\boldsymbol{r} := \boldsymbol{r} - \sum_{i=1}^{j} h_{ij}\boldsymbol{v}_i$ — Orthogonalise
5. $h_{j+1,j} := \| \boldsymbol{r} \|_2$ — Normalise
6. **if** $h_{j+1,j} = 0$ **stop**
7. $\boldsymbol{v}_{j+1} := \boldsymbol{r}/h_{j+1,j}$ — Get new vector
8. Compute approximate solution and test for convergence

We will now derive relations between $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{V}_j$ and $h_{i,j}$. Later on we will describe how to calculate approximate eigensolutions and residuals.

Put II.1 and II.2 together and we get

$$\boldsymbol{r} = (\boldsymbol{A} - \mu_j \boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{V}_j \boldsymbol{t}_j.$$

From II.4 we rewrite

$$\sum_{i=1}^{j} h_{ij}\boldsymbol{v}_i = \boldsymbol{V}_j\boldsymbol{h}_j, \;\; \boldsymbol{h}_j = \begin{bmatrix} h_{1,j} \\ \vdots \\ h_{j,j} \end{bmatrix}, \;\; \boldsymbol{V}_j = [\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_j]$$

Use step II.4 and II.7 together with the relations above we get

$$\boldsymbol{v}_{j+1}h_{j+1,j} = (\boldsymbol{A} - \mu_j\boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{V}_j\boldsymbol{t}_j - \boldsymbol{V}_j\boldsymbol{h}_j$$

substitute $\boldsymbol{h}_j = \begin{bmatrix} \boldsymbol{h}_j \\ h_{j+1,j} \end{bmatrix}$

$$\boldsymbol{V}_{j+1}\boldsymbol{h}_j = (\boldsymbol{A} - \mu_j\boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{V}_j\boldsymbol{t}_j$$

multiply with $(\boldsymbol{A} - \mu_j\boldsymbol{B})$

$$(\boldsymbol{A} - \mu_j\boldsymbol{B})\boldsymbol{V}_{j+1}\boldsymbol{h}_j = \boldsymbol{B}\boldsymbol{V}_j\boldsymbol{t}_j$$

Separate terms with $\boldsymbol{A}$ and $\boldsymbol{B}$, substitute also $\boldsymbol{t}_j$ with $\begin{bmatrix} \boldsymbol{t}_j \\ 0 \end{bmatrix}$ to get the relation at the j:th step,

$$\boldsymbol{A}\boldsymbol{V}_{j+1}\boldsymbol{h}_j = \boldsymbol{B}\boldsymbol{V}_{j+1}(\boldsymbol{h}_j\mu_j + \boldsymbol{t}_j).$$

Put $\boldsymbol{H}_{m+1,m} = [\boldsymbol{h}_1, \ldots, \boldsymbol{h}_m]$, $\boldsymbol{T}_{m+1,m} = [\boldsymbol{t}_1, \ldots, \boldsymbol{t}_m]$ with appropriate zeros added to the bottom of each $\boldsymbol{h}_j$, $\boldsymbol{t}_j$. Introduce the new matrix

$$\boldsymbol{K}_{m+1,m} = \boldsymbol{H}_{m+1,m}diag(\mu_i) + \boldsymbol{T}_{m+1,m}. \tag{3.13}$$

Note that both $\boldsymbol{H}_{m+1,m}$ and $\boldsymbol{K}_{m+1,m}$ are Hessenberg matrices. We finally get the relation

$$\boldsymbol{A}\boldsymbol{V}_{m+1}\boldsymbol{H}_{m+1,m} = \boldsymbol{B}\boldsymbol{V}_{m+1}\boldsymbol{K}_{m+1,m}. \tag{3.14}$$

3.4.1 Approximate Eigensolution

We will first show that if $\mathcal{K}_m$ is invariant under $(\boldsymbol{A}-\mu_i\boldsymbol{B})^{-1}\boldsymbol{B}$ and $\boldsymbol{A}$ is invertible then, $\mathcal{K}_m$ is invariant under $\boldsymbol{A}^{-1}\boldsymbol{B}$.

If $\boldsymbol{x} \in \mathcal{K}_m$, and $(\boldsymbol{A} - \mu_i\boldsymbol{B})^{-1}\boldsymbol{B}$ is invariant under $\mathcal{K}_m$, then

$$\boldsymbol{z} = (\boldsymbol{A} - \mu_i\boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{x}, \;\; \boldsymbol{z} \in \mathcal{K}_m$$

multiply with $(\boldsymbol{A} - \mu_i\boldsymbol{B})$ and separate $\boldsymbol{A}$ and $\boldsymbol{B}$

$$\boldsymbol{A}\boldsymbol{z} = \boldsymbol{B}(\boldsymbol{x} + \mu_i\boldsymbol{z})$$

and thus if $\boldsymbol{A}$ is invertible then $\mathcal{K}_m$ is invariant under $\boldsymbol{A}^{-1}\boldsymbol{B}$.

If $\mathcal{K}_m$ is invariant under $(\boldsymbol{A} - \mu_i\boldsymbol{B})^{-1}\boldsymbol{B}$ then $\boldsymbol{v}_{m+1} = 0$ and the relation (3.14) becomes,

$$\boldsymbol{A}\boldsymbol{V}_m\boldsymbol{H}_{m,m} = \boldsymbol{B}\boldsymbol{V}_m\boldsymbol{K}_{m,m} \tag{3.15}$$

If

$$\boldsymbol{K}_{m,m}\boldsymbol{y} = \lambda\boldsymbol{H}_{m,m}\boldsymbol{y} \tag{3.16}$$

then put it in (3.15) and we get

$$\boldsymbol{A}\boldsymbol{V}_m\boldsymbol{H}_{m,m}\boldsymbol{y} = \lambda\boldsymbol{B}\boldsymbol{V}_m\boldsymbol{H}_{m,m}\boldsymbol{y}.$$

Thus if $(\lambda, \boldsymbol{y})$ is an eigenpair to (3.16) then

$$\boldsymbol{A}\boldsymbol{u} = \lambda\boldsymbol{B}\boldsymbol{u}, \quad \boldsymbol{u} = \boldsymbol{V}_m\boldsymbol{H}_{m,m}\boldsymbol{y}$$

Now assume that $\mathcal{K}_m$ is not invariant, which will be the most usual case. Let $(\tilde{\lambda}_i^{(m)}, \tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})})$ be an eigenpair of (3.16) and take

$$\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} = \boldsymbol{V}_{m+1}\boldsymbol{H}_{m+1,m}\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \tag{3.17}$$

as the approximate eigenvector and $\tilde{\lambda}_i^{(m)}$ as the approximate eigenvalue to (3.12). The residual will be

$$\begin{aligned}
(\boldsymbol{A} - \tilde{\lambda}_i^{(m)}\boldsymbol{B})\tilde{\boldsymbol{u}}_{\boldsymbol{i}}^{(\boldsymbol{m})} &= (\boldsymbol{A} - \tilde{\lambda}_i^{(m)}\boldsymbol{B})\boldsymbol{V}_{m+1}\boldsymbol{H}_{m+1,m}\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \\
&= \boldsymbol{B}\boldsymbol{V}_{m+1}(\boldsymbol{K}_{m+1,m} - \tilde{\lambda}_i^{(m)}\boldsymbol{H}_{m+1,m})\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \\
&= \boldsymbol{B}\boldsymbol{v}_{m+1}(k_{m+1,m} - \tilde{\lambda}_i^{(m)}h_{m+1,m})\boldsymbol{e}_m^*\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})} \\
&= \boldsymbol{B}\boldsymbol{v}_{m+1}(\mu_m - \tilde{\lambda}_i^{(m)})h_{m+1,m}\boldsymbol{e}_m^*\tilde{\boldsymbol{y}}_{\boldsymbol{i}}^{(\boldsymbol{m})}.
\end{aligned} \tag{3.18}$$

The first equality comes from (3.17), the second from (3.14), the third from (3.16) and the fourth from (3.13). Note that the residual is $\boldsymbol{B}^{-1}$ orthogonal against $\boldsymbol{V}_m$.

3.4.2 Shifts

Assume that we use s different shifts, $\mu_1, \ldots, \mu_s$ and that i:th shift is used j_i times. For memory reasons we can usually have only one LU-factorisation of each shifted matrix $(\boldsymbol{A}-\mu_i\boldsymbol{B})$ in memory at a certain time. The LU-factorisation usually costs more than to solve $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{v}, \boldsymbol{U}\boldsymbol{r} = \boldsymbol{x}$. So in step II.2 we want

- Factor $\boldsymbol{L}_i\boldsymbol{U}_i = (\boldsymbol{A} - \mu_i\boldsymbol{B})$ one time.
- Keep the same factor for j_i steps.

If we use iterative solvers this may not apply.

3.5 Parallel Rational Krylov Algorithm

The key to the parallel algorithm is that it does not matter in exact arithmetic in which order the operators $(\boldsymbol{A} - \mu_i\boldsymbol{B})^{-1}\boldsymbol{B}$ are applied in building a basis for the subspace $\mathcal{K}_m$ (3.11).

Algorithm 4. RKS Parallel

I **Start**. Choose a vector $\boldsymbol{v}_1$ of norm 1 **and set** $j := 1$

II **do while** $j \leq m$

1. $\boldsymbol{r}_k := \boldsymbol{V}_{j+k-1}\boldsymbol{t}_{j+k-1}$, $k = 1, 2 \ldots, p$ Choose starting combinations
2. $\boldsymbol{r}_k := (\boldsymbol{A} - \mu_k \boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{r}_k$, $k = 1, 2 \ldots, p$ Choose μ_k,$k = 1, 2 \ldots, p$
3. Orthogonalise and get new vectors

   for $k = 1, 2, \ldots, p$ **do**

   (a) $h_{ij} := \boldsymbol{v}_i^* \boldsymbol{r}_k, i = 1, 2, \ldots, j$

   (b) $\boldsymbol{r}_k := \boldsymbol{r}_k - \sum_{i=1}^{j} h_{ij}\boldsymbol{v}_i$

   (c) $h_{j+1,j} :=\| \boldsymbol{r}_k \|_2$

   (d) **if** $h_{j+1,j} = 0$ **stop**

   (e) $\boldsymbol{v}_{j+1} := \boldsymbol{r}_k / h_{j+1,j}$

   (f) $j := j + 1$

   end

4. Compute approximate solution and test for convergence

end

In step II.1 note that $\boldsymbol{v}_{j+k-1}$, $k > 1$ has not yet been computed so all starting combinations in the vector $\boldsymbol{t}_{j+k-1}$ are not available.

The algorithm above leaves some choices on how to implement it. We have chosen two different ways. They differ in the way orthogonalisation is done. Both programs use p different processors to compute $\boldsymbol{r}_k := (\boldsymbol{A} - \mu_k \boldsymbol{B})^{-1}\boldsymbol{B}\boldsymbol{r}_k$, $k = 1, \ldots, p$. The first program uses an additional processor to do the orthogonalisation. The second program lets each processor orthogonalise its own vector.

In the implementations of the algorithm we have used direct solvers. So in step II.2 on each processor, we factorise once for each shift and keep the shifts the same number of iterations on all processors.

The choice of starting combinations in step II.1 is as follows: in the first parallel iteration each processor uses $\boldsymbol{v}_1$. Then in the following parallel iterations each processor starts with its own orthogonalised $\boldsymbol{r}_k$. With a parallel iteration we mean an iteration of the `while loop` in step II.

Chapter 4

Implementation Details

4.1 Parallel Program 1

This program has been implemented in a Master/Slave way. The Master program does all program control and all orthogonalisation. The Slave programs do all operations $\boldsymbol{r}_k := (\boldsymbol{A} - \mu_k \boldsymbol{B})^{-1} \boldsymbol{B} \boldsymbol{r}_k, k = 1, \ldots, p$

program Master
Choose
    No of slave processors p
    The p first shifts $\mu_i, i = 1, \ldots, p$
    The start vector $\boldsymbol{v}_1$
Start p Slave programs
Initialise the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$
Send$(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{v}_1)$ to all slaves
Send μ_i to slave No i,$i = 1, \ldots, p$
$j := 0$
do until convergence
(Decide the status for all slaves
all slaves have the same status
status ={ New Shift, Same Shift , STOP })
    if status = New Shift **then**
        Choose p new shifts $\mu_i, i = 1, \ldots, p$
    end if
    do $i = 1, p$
        $j := j + 1$
        Receive vector $\boldsymbol{r}$ from slave no i
        Orthogonalise $\boldsymbol{r}$ against all previous $\boldsymbol{v}_i$ $i = 1, \ldots, j$
        if $\| \boldsymbol{r} \|_2 = 0$ **then**
            status := STOP
        else
            $\boldsymbol{v}_{j+1} := \boldsymbol{r} / \| \boldsymbol{r} \|_2$
        end if
        Send status to slave no i

```
        if status = STOP then
            Compute approximate solution and convergence
            STOP
        end if
        Send v_{j+1} and μ_i to slave no i
    end do
    Compute approximate solution and check convergence
end until
end program Master

program Slave
Receive(A, B, μ, r) from Master
status := New shift
do while status ≠ STOP
    if status = New Shift then
        factor LU := (A − μB)
    end if
    r := U^{-1} L^{-1} B r
    send r to master
    receive status from master
    if status = STOP then
        STOP
    end if
    receive r, μ from master
end while
end program Slave
```

The time the master program spends in communication in each iteration is

$$t_M = Cnp + Lp$$

where C is a constant which depends on the network and which precision is used in the program (single precision, double precision, complex ,...). p is the number of slave processors, L is the latency and n is the dimension of $\boldsymbol{A}$. The time each slave program spends in communication in each iteration is

$$t_S = Cn + L.$$

The main problem with this implementation is that the slave processors became idle while orthogonalisation is done in the Master program. The time the Master processor spends in each orthogonalisation grows linearly with j. The Master processor is idle while the slaves do the factorisation. We will comment more on this in the tests.

The advantage of this implementation is that the communication grows linearly with the number of processors and the total communication time is smaller than in the second implementation. This will be an advantage on a slow network. If the operations $\boldsymbol{r}_k := (\boldsymbol{A} - \mu_k \boldsymbol{B})^{-1} \boldsymbol{B} \boldsymbol{r}_k, k = 1, \ldots, p$ takes relatively long time, so that the Master processor has time to orthogonalise while the slaves work, this implementation will work relatively well.

4.2 Parallel Program 2A

The second implementation can be described as same program multiple data (SPMD). The same programs are started on p different processors. Processor number k, $1 \leq k \leq p$ operate $\boldsymbol{r}_k := (\boldsymbol{A} - \mu_k \boldsymbol{B})^{-1} \boldsymbol{B} \boldsymbol{r}_k$ and orthogonalises $\boldsymbol{r}_k$ against $\boldsymbol{v}_i, i = 1, \ldots, j$. For this to be possible, the processors exchange the orthogonalised vectors with each other. Note also that the orthogonalisation is split up in two parts.

program SPMD
get the number of started programs: p
determine which program I am: me $(1 \leq me \leq p)$
Choose
    The p first shifts $\mu_i, i = 1, \ldots, p$
    The start vector $\boldsymbol{v}_1$
status := New shift
j:= 1
$\boldsymbol{r} := \boldsymbol{v}_1$
do until convergence
    if status = New Shift **then**
        factor $\boldsymbol{LU} := (\boldsymbol{A} - \mu_{me} \boldsymbol{B})$
    end if
    $\boldsymbol{r} := \boldsymbol{U}^{-1} \boldsymbol{L}^{-1} \boldsymbol{B} \boldsymbol{r}$
    Orthogonalise $\boldsymbol{r}$ against $\boldsymbol{v}_i, i = 1, \ldots, j$
    do $i = 1, me - 1$
        $j := j + 1$
        receive $\boldsymbol{v}_j$ from program i
    end do
    orthogonalise $\boldsymbol{r}$ against $\boldsymbol{v}_i, i = j - me + 2, j$
    if $\| \boldsymbol{r} \|_2 = 0$ **then**
    signal to other processors to STOP
    Compute approximate solution and STOP
    else
        $\boldsymbol{v}_{j+1} := \boldsymbol{r} / \| \boldsymbol{r} \|_2$
    end if
    $j := j + 1$
    send $\boldsymbol{v}_j$ to all other programs
    do $i = me + 1, p$
        $j := j + 1$
        receive $\boldsymbol{v}_j$ from program i
    end do
    Compute approximate solution and check convergence
    decide status for next iteration
    if status = STOP **then**
        STOP
    else if status = New Shift **then**
        Choose p new shifts $\mu_i, i = 1, \ldots, p$
    end if
end until
end program SPMD

Every processor sends and receives $p-1$ vectors of length n. A particular processor waits while the others send and receive. So every processor slows down with the system communication time,

$$t_c = Cp(p-1)n + Lp(p-1).$$

Besides the communication time, every processor has to wait while the others do the second orthogonalisation, this is

$$t_{ortho} = (1 + 2 + \ldots + p - 1)D$$

where D is the time it takes to orthogonalise against one vector of length n.

This implementation works well up to around 6 processors on the IBM-SP2, see the test results. Because of the quadratic growth in communication time as p increases this algorithm will never work on too many processors.

4.3 Parallel Program 2B

With a reorder of `receive` and `send` in the algorithm above it should be possible that each processor has a linear growth in communication, as the number of processors increases, while the system of processors has quadratic growth.

program SPMD B
get the number of started programs: p
determine which program I am:me $(1 \leq me \leq p)$
Choose
    The p first shifts $\mu_i, i = 1, \ldots, p$
    The start vector $\boldsymbol{v}_1$
status := New shift
j:= 1
$\boldsymbol{r} := \boldsymbol{v}_1$
do while status $\neq$ STOP
    if status = New Shift **then**
        factor $\boldsymbol{LU} := (\boldsymbol{A} - \mu_{me}\boldsymbol{B})$
    end if
    $\boldsymbol{r} := \boldsymbol{U}^{-1}\boldsymbol{L}^{-1}\boldsymbol{B}\boldsymbol{r}$
    Orthogonalise $\boldsymbol{r}$ against $\boldsymbol{v}_i, i = 1, \ldots, j$
    if $j > 1$ **then**
        do $i = me + 1, p$
            $j := j + 1$
            receive $\boldsymbol{v}_j$ from program i
        end do
    end if
    do $i = 1, me - 1$
        $j := j + 1$
        receive $\boldsymbol{v}_j$ from program i
    end do
    orthogonalise $\boldsymbol{r}$ against $\boldsymbol{v}_i, i = j - p + 2, j$
    if $\| \boldsymbol{r} \|_2 = 0$ **then**
    signal to other processors to STOP

Compute approximate solution and STOP
else
$\boldsymbol{v}_{j+1} := \boldsymbol{r} / \parallel \boldsymbol{r} \parallel_2$
end if
$j := j + 1$
send $\boldsymbol{v}_j$ to all other programs
Compute approximate solution and check convergence
decide status for next iteration
if status = New Shift **then**
Choose p new shifts $\mu_i, i = 1, \ldots, p$
end if
end while
do $i = me + 1, p$
$j := j + 1$
receive $\boldsymbol{v}_j$ from program i
end do
end program SPMD B

Every processor sends and receives $p - 1$ vectors of length n. So every processor slows down with communication time

$$t_c = C(p-1)n + L(p-1).$$

on the other hand, the system has a communication time of,

$$t_c = Cp(p-1)n + Lp(p-1).$$

The idea behind this version is to receive the data first when you need it, and work while the other processors communicate.

The process of sending a message with PVM is composed into (1) initiating a sending buffer, (2) packing data into the buffer and (3) sending the buffer. An arriving message first is stored into a temporary buffer and later is unpacked by the program that receives the message.

That the sending process is blocking means that it returns first when the receiving process has unpacked the data. In order for the above program to work properly the `send` operation should be non-blocking, so the sending process can do useful work and do not have to wait until the receiving process unpacks the data. However, in this particular implementation on the IBM-SP2 with Fortran and PVMe, it has seemed like that the `send` operation is blocking, even though it should not be blocking according to the manual, see [9].

4.4 Implementation Details

4.4.1 Solvers

We have used direct band solvers from LAPACK [2]. Other solvers are possible like iterative solvers and general sparse solvers. However there is a problem with using general sparse solvers. The factorisation on many general sparse solvers can be split into analysing the nonzero structure and numeric factorisation. The analysing phase needs to be done only once for factorisation of different matrices with the same structure, like them we have in this report $(\boldsymbol{A} - \mu_i \boldsymbol{B})$.

The analysing phase can take for example 3 to 10 times as much time as the numerical factorisation. In order for the algorithm to give good speedup results when using direct general sparse solvers, the analysing phase needs to be done in parallel on all processors, and then be distributed to the different processors. To the author's knowledge this has not yet been done by anybody.

4.4.2 Subroutines in Linear Algebra

It is a nontrivial task to write efficient and accurate subroutines for solving equations and doing vector and matrix operations. These subroutines are needed in order to implement the parallel rational Krylov algorithm. Whenever appropriate we have used LAPACK and BLAS routines, see [2].

The BLAS routines is a collection of subroutines in basic matrix operations like matrix matrix multiplications, matrix vector multiplications and scalar products. Many computer manufactories have their own optimised BLAS routines. The speed of these optimised routines can be significantly higher than matrix operations implemented in the naive way, due to that modern computers use memory with different access time in a hierarchy.

The LAPACK routines is a collection of linear algebra routines like equation solvers and eigenvalue solvers. LAPACK uses the BLAS routines. The main work in this implementation is done in the factorisation, solving and multiplication routines, see the table below for the specific routines used.

Subroutines in Linear Algebra			
	double complex	double precision	library
Matrix vector product	zgemv	dgemv	BLAS
scalar product	zdotc	ddot	BLAS
Norm	dznrm2		BLAS
Scaling	zdscal	dscal	BLAS
general band factorise	zgbtrf	dgbtrf	LAPACK
general band solve	zgbtrs	dgbtrs	LAPACK
generalised eigenvalue solver	zgegv	dgegv	LAPACK

4.4.3 Orthogonalisation

To ensure that the basis is orthonormal to working precision every vector $\boldsymbol{r}_j$ is orthogonalised twice against the basis, as suggested by Kahan, see [12].

4.4.4 Program Code

The code is experimental and is only intended for research purposes.

The code for the IBM-SP2 and the cluster of SUN workstations is basically the same. The different versions used by the different computer configurations are made by the preprocessor cpp from a generic program.

On the cluster of SUN workstations, the programs are written in Fortran77 together with the libraries BLAS, LAPACK and PVM. The nonstandard Fortran77 statements `malloc` and `pointer` are used for memory management.

On the IBM-SP2 the xlf fortran compiler is used. The xlf compiler provide maximum compatibility with existing Fortran77 programs and uses many features in Fortran90. The Fortran90 feature `allocate` is used for memory management in the code. The libraries used are LAPACK, BLAS from the ESSL library and PVMe, the IBM version of PVM.

All programs have a double complex version. Program 1 is the only program with a double precision version.

MATLAB is used to process the data from the tests and to draw the graphs. A semi-parallel code for convergence tests is implemented in MATLAB.

Chapter 5

Test Results

The algorithms described in the previous chapter have been tested on IBM-SP2, and on a cluster of SUN workstations. The main aspects tested are efficiency, speedup and convergence. Efficiency and speedup are related to the speed performance of the algorithm. In chapter 3 of this report we predicted that it should not matter in which order the operations $\boldsymbol{r}_k := (\boldsymbol{A} - \mu_k \boldsymbol{B})^{-1} \boldsymbol{B} \boldsymbol{r}_k$, $k = 1, \ldots, p$ are applied, we should get the same approximation if we do it in parallel or sequentially, thus we should get similar convergence behaviour. Is this true on a computer with floating point arithmetic ? We will discuss this issue later.

5.1 Test Sites

The IBM-SP2 was chosen because it was the best available MIMD computer in Sweden for research purposes. The SUN configuration was chosen because it was available at the department.

The IBM-SP2 is a distributed memory MIMD computer. The machine used at the Center for Parallel Computers, Royal Institute of Technology, Stockholm, Sweden, had 55 nodes(the configuration is now upgraded). Each node contains a processor and memory. The nodes share data via message passing over a high performance switch. There are two types of nodes, thin nodes with 128Mbyte memory and wide nodes with 512Mbyte memory. We have only used thin nodes in our tests.

The SUN configuration contains SUN ELC workstations connected through an Ethernet network.

In the table below we give some information about the nodes and network used.

	IBM-SP2	SUN
Processor	RS/6000	SPARC
Architecture	POWER2	SPARC
Clock frequency	66.7MHZ	33MHZ
Floating point operations per cycle	4	
Peak performance	266Mflops	10Mflops
Main memory	128Mbyte	24Mbyte
Data Cache	64Kbyte	
Instruction Cache	32Kbyte	
Bandwidth network	35Mbyte/s	1Mbyte/s
Latency	40μs	2ms

The table above gives peak performance, in the table below we give measured performance. For the test of the communication bandwidth we have used PVM on the SUN workstations and PVMe on IBM-SP2 (PVMe is IBM version of PVM). The data transferred in the tests are double precision vectors of length 10000. The number of flops is measured for a scalar product (ddot in the BLAS).

	IBM-SP2	SUN
Performance	23Mflops	2.4Mflops
Bandwidth network	13.3Mbyte/s	0.27Mbyte/s

5.2 Test Problem

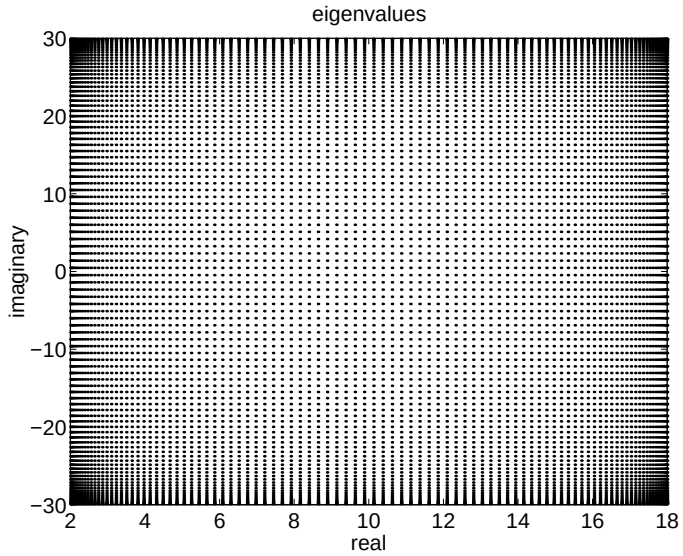
The test matrices consist of a finite difference discretisation of the differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{qy}{h}\frac{\partial u}{\partial y} = \lambda u$$

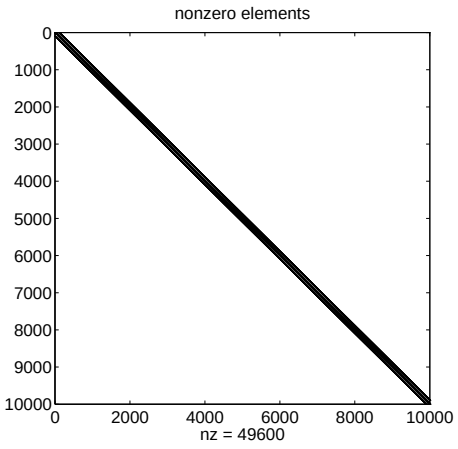
The test problem is artificial but related to a convection diffusion problem, it is intended for testing convergence and efficiency. This problem has been chosen for the following reasons; it is easy to vary the size of the problem, the eigenvalues are known and the eigenvalues can be arbitrary ill or well conditioned. The matrix $\boldsymbol{A}$, generated from the discretisation is a sparse matrix with 5 diagonals and the matrix $\boldsymbol{B}$ is the identity matrix. The matrix $\boldsymbol{A}$ is real and non-symmetric. Some facts about the test problems for the different configurations are given in the table below.

	IBM-SP2	SUN
Upper and lower bandwidth	100	100
Dimension	10000	1000
Number of nonzero diagonals	5	5
Amount of memory for the LU factor	48M byte	4.8M byte
Amount of memory for the basis (m=150)	24M byte	2.4M byte
Precision	double complex	double complex

The larger test problem chosen for the IBM-SP2 would not fit on the cluster of SUN workstations due to less amount of memory on the SUN workstations. The bandwidth of the matrix $\boldsymbol{A}$ on the SUN workstations is chosen so that the same time ratio for orthogonalise / solving the equations as in the IBM-SP2 is obtained.



The exact eigenvalue distribution for the larger test problem is given in the graph above.



The nonzero elements from the larger test matrix.

5.3 Test of The Sequential Algorithm

The tests of the sequential algorithm are done in order to obtain data to compare the sequential and parallel algorithms.

Shifts

We use every shift the same number of iterations in each test, this is because we want to compare these tests with the tests on the parallel algorithms where it is unpractical to have different numbers of iterations with different shifts.

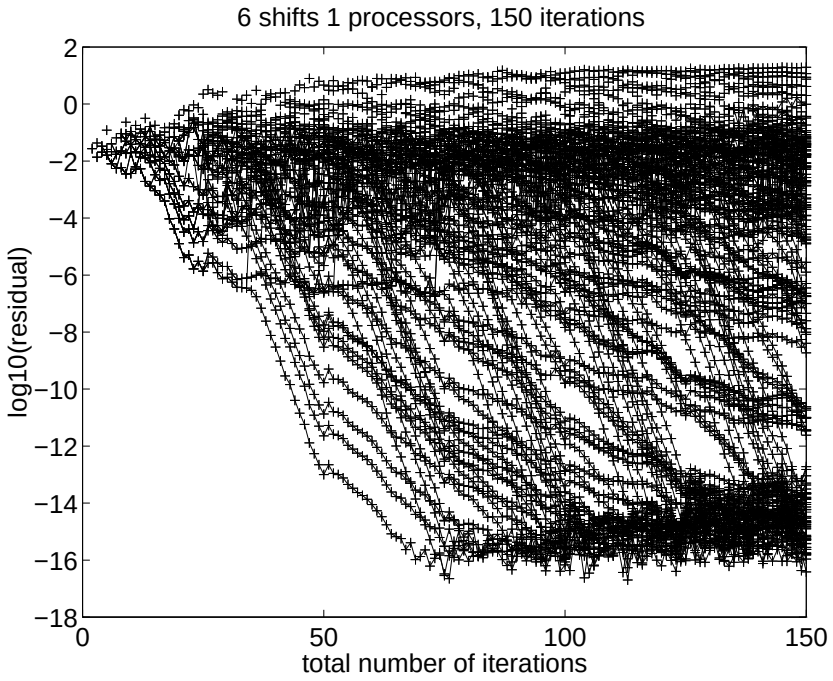
Timing

We have kept time on the different parts of the algorithm. In these tests we have not timed the computation of the approximate solution step. The algorithms can roughly be divided into factorisation of $\boldsymbol{LU} = (\boldsymbol{A} - \mu\boldsymbol{B})$, multiplication $\boldsymbol{r} = \boldsymbol{U}^{-1}\boldsymbol{L}^{-1}\boldsymbol{B}\boldsymbol{v}$, and orthogonalisation. The time it takes to orthogonalise a vector against the basis grows linearly with the number of iterations.

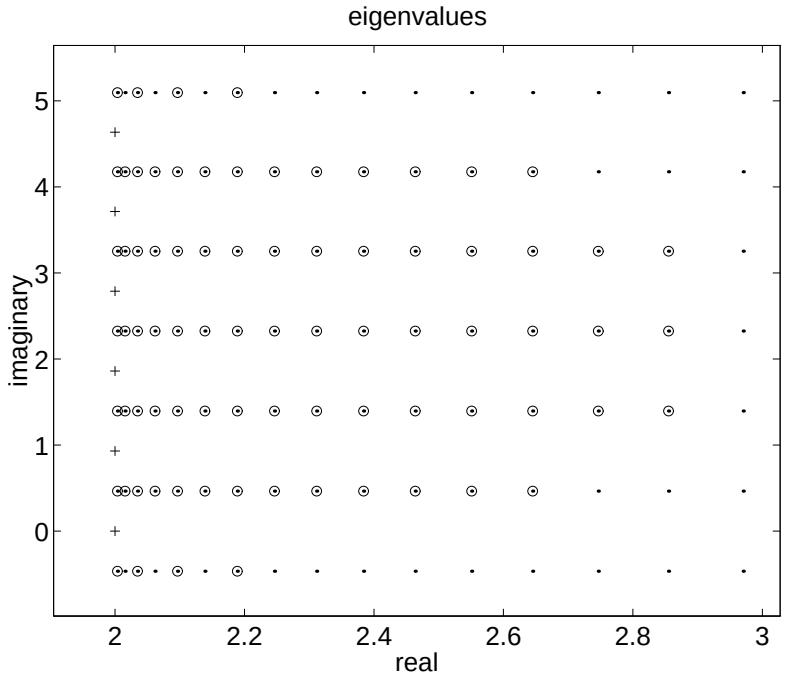
Test Results				
Configuration	IBM-SP2	IBM-SP2	SUN	SUN
Test No	1	2	3	4
Shifts	6	3	6	3
Iterations with each shifts	25	30	25	30
Total no of Iterations	150	90	150	90
Timing Results (in seconds)				
One factorisation	16.6	17.0	22.8	24.3
One multiplication	0.483	0.485	0.656	0.648
Factorisation total	99.6	51.0	136.8	73.0
Multiplication total	72.45	43.65	98.4	58.3
Orthogonalisation total	103.5	38.42	113	40.9
Other	0	0	3.6	3.6
Total program	275.6	133.1	351.8	175.8

Convergence

The convergence is measured by the norm of the residual see (3.18). In the plot below the norm of the residuals are plotted in each iteration with a plus +. Lines are drawn between residuals corresponding to the same eigenvalue in different iterations. The plot is for test number 1.



In the figure below the exact eigenvalues are plotted with a dot $\cdot$, the approximate eigenvalues with a residual less than 10^{-5} are plotted with circle $\circ$ and the shifts are plotted with a plus $+$. The plot is for test number 1.



5.4 Parallel Program 1

How well this implementation works, depends on how much time it spends in orthogonalisation, compared to factorisation and multiplication. The problem with this algorithm is that the processors became idle. The communication time is negligible in the test compered to the idle time.

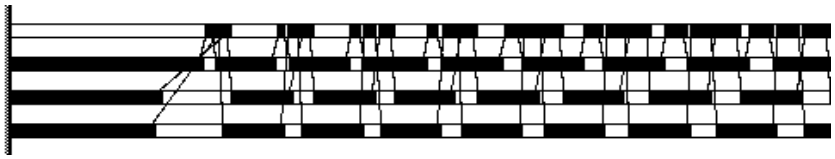
The advantage of this implementation is that the basis is only located on one processor and the slaves only have to keep the matrices and the factorisation. The communication grows linearly with the number of processors so the total communication time is short compered to the other parallel algorithms.

Test Results Parallel Program 1		
Configuration	IBM-SP2	SUN
Test No	5	6
Slaves	3	3
Master	1	1
Shifts	3	3
Iterations with each shifts	30	30
Total no of Iterations	90	90
Timing Results On One Slave (in seconds)		
One factorisation	17.0	24.2
One multiplication	0.485	0.66
Factorisation total	17.0	24.2
Multiplication total	14.5	19.8
Communication estimated	0.72	3.6
Idle time	29.18	27.7
Total time	61.4	75.3
Timing Results On Master (in seconds)		
Orthogonalisation total	38.4	39.3
Communication estimated	2.16	10.8
Idle time	20.84	25.2
Total time	61.4	75.3
Timing Results Total (in seconds)		
Parallel Program	61.4	75.3
Sequential Program	133	176
Speedup	2.2	2.3
Efficiency	54%	58%

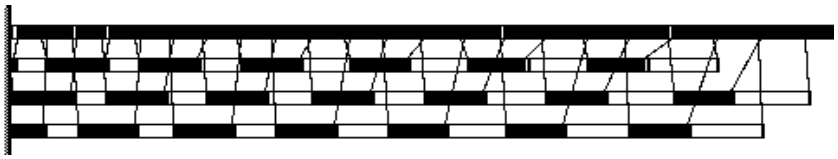
The graph below shows idle time, working time and communication for the SUN configuration. For this particular test we have used 15 iterations ($3 \cdot 15 = 45$ total iterations), 3 shifts, 1 master and 3 slaves. The white area in the bars corresponds to idle time and the black to working time. The lines between the bars symbolise communication. In the first iteration the master is idle while the slaves do the factorisations, only part of the factorisation time is shown in the figure. In the second iteration a particular slave is idle while the master orthogonalise its vector and the master is idle while it is waiting

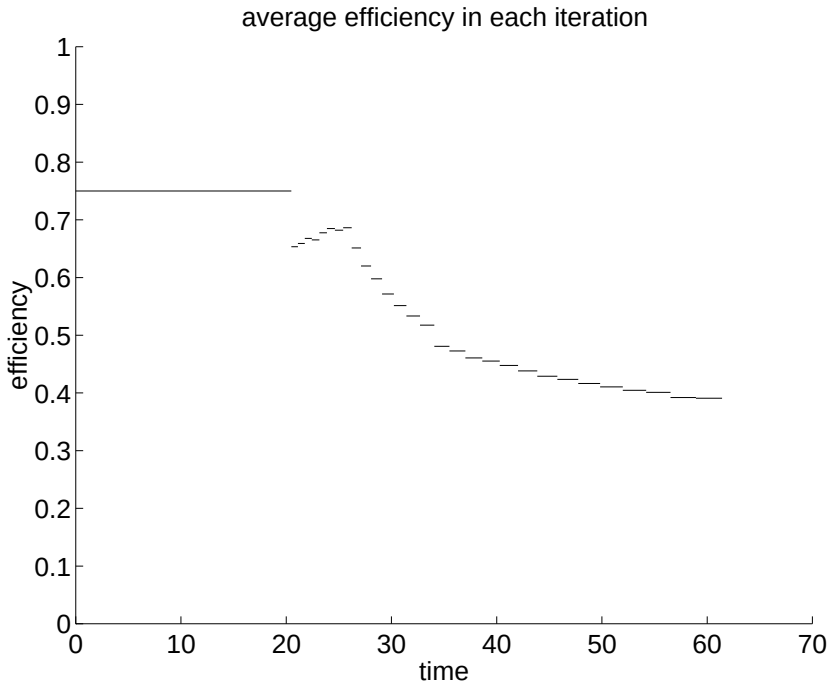
for the next slave. After a number of iterations the master has no idle time. Orthogonalisation grows linearly with the number of iterations and so do the slaves idle time. The speedup and efficiency will depend on the total number of iterations.

The first iterations



The last iterations





The graph above shows the average efficiency in each iteration for test no 5.

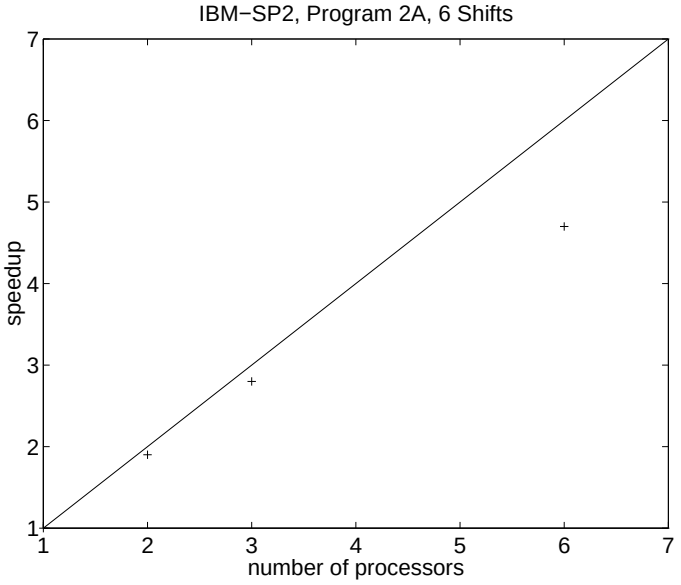
The tests shows low efficiency for this algorithm. The algorithm can work rather well if the time spent in factorisation and multiplication is large compared to the orthogonalisation time. If the opposite is true this algorithm will achieve poor results.

5.5 Parallel Program 2A

This algorithm gives much better performance than the previous one. The algorithm communicates more and the idle time is considerable less. It was the idle time that gave the previous algorithm poor performance.

Test Results Parallel Program 2A				
Configuration	IBM	IBM	IBM	IBM
Test No	7	8	9	10
No of Processors	3	6	3	2
Shifts	3	6	6	6
Iterations with each shifts	30	25	25	25
Total no of Iterations	90	150	150	150
Timing Results On One Processor (in seconds)				
One factorisation	18.6	18.65	17.44	17.05
One multiplication	0.4933	0.4908	0.486	0.485
Factorisation total	18.6	18.65	34.88	51.15
Multiplication total	14.8	12.27	24.3	36.37
Orthogonalisation total	12.4	16.8	34.41	51.55
Communication + idle time	3.43	11.07	5.48	3.56
Total time	49.23	58.79	99.07	142.6
Sequential Program	133	275.6	275.6	275.6
Speedup	2.7	4.7	2.8	1.9
Efficiency	90%	78%	93%	97%

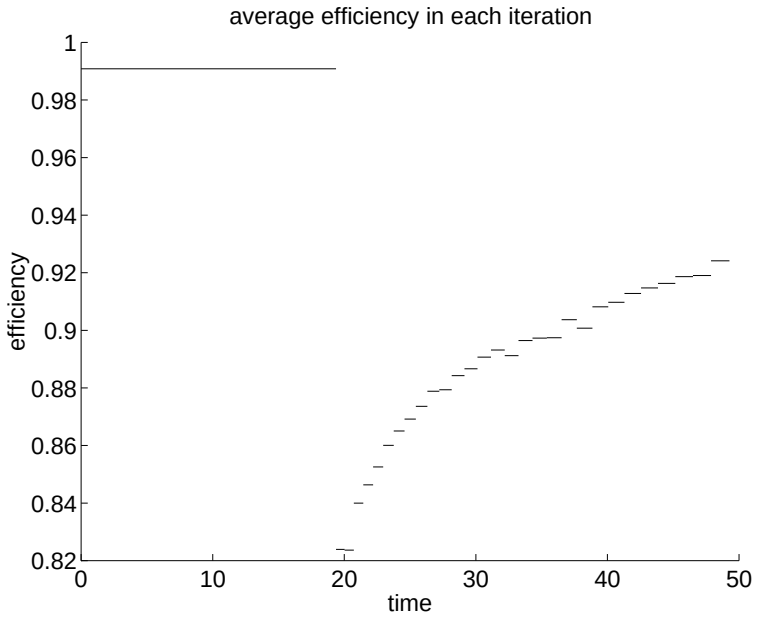
The plot below shows speedup for the tests using 6 shifts on the IBM-SP2. The straight line is the ideal speedup, the + is the speedup from the test.



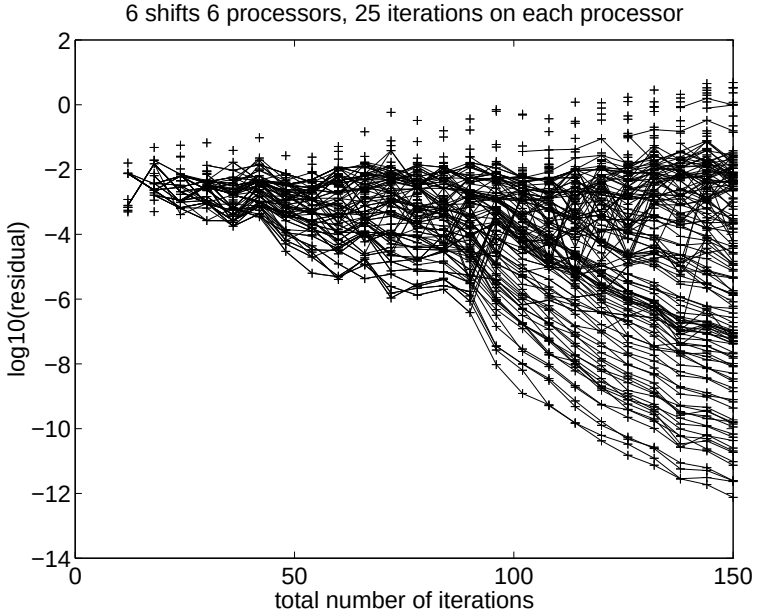
Test Results Parallel Program 2A				
Configuration	SUN	SUN	SUN	SUN
Test No	11	12	13	14
No of Processors	3	6	3	2
Shifts	3	6	6	6
Iterations with each shifts	30	25	25	25
Total no of Iterations	90	150	150	150
Timing Results On One Processor (in seconds)				
One factorisation	26.4	24.3	24.2	22.8
One multiplication	0.675	0.658	0.661	0.662
Factorisation total	26.4	24.3	48.4	68.4
Multiplication total	20.3	16.5	33.1	49.7
Orthogonalisation total	13.1	17.8	36.6	54.8
Communication + idle time	10.9	33.2	18.9	16.7
Other	1.0	1.0	1.8	2.6
Total time	71.7	92.8	138.8	192.2
Sequential Program	176	351.8	351.8	351.8
Speedup	2.45	3.79	2.53	1.86
Efficiency	81.8%	63.2%	84.5%	92.8%

The graph below shows idle time, working time and communication for the SUN configuration. First every processor factorises, only part of the factorisation time is shown in the figure. Then at each iteration each processor multiplies, orthogonalises its own vector and exchanges the vector with the other processors. Every processor has to wait while the others communicate.

The graph below shows the average efficiency at each iteration for test 7. In the first iteration most time is spent in factorisation, then in the second iteration the efficiency drops. In the following iterations the efficiency grows due to the orthogonalisation time grows in each iteration. Note that the scaling for the efficiency axis is different from the corresponding graph in the tests for parallel program 1.



The convergence is measured by the norm of the residual see (3.18). In the plot below, the norms of the residuals are plotted in each iteration with a plus +. Lines are drawn between residuals corresponding to the same eigenvalue in different iterations. Note that the residuals come down together because we use different shifts on different processors at the same time compared to the sequential algorithm where they come down at each shift. The levels of the residuals are not so good as in the sequential algorithm. The parallel algorithms have some problems with the numerical computations that the sequential does not have. We will discuss the difference in numerical computation between the parallel and sequential algorithm later.



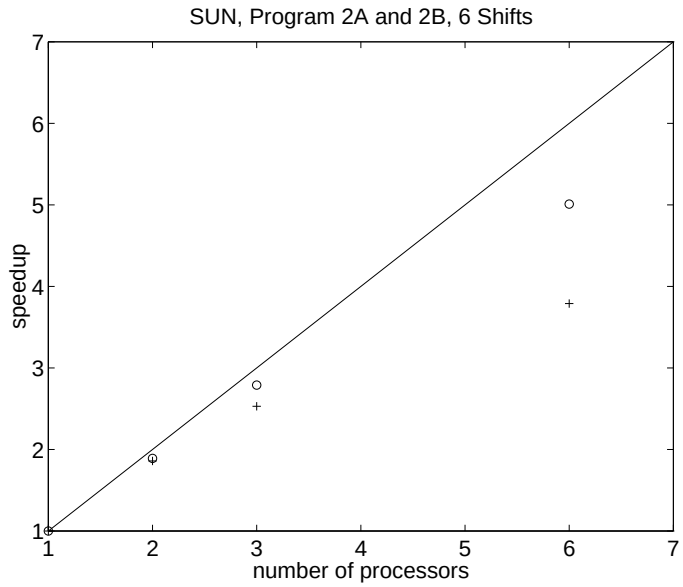
This algorithm gives much better results than the previous one even though the total communication time is longer in this one. The next program obtains even better results.

5.6 Parallel Program 2B

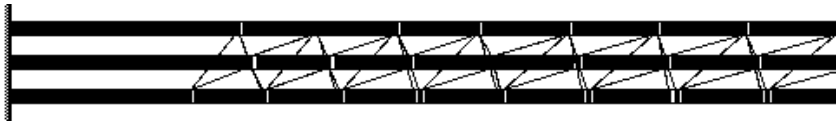
The difference between the parallel program 2A and 2B is how the vectors are exchanged. In 2A the processors waited for each other while the data were exchanged. In 2B a arriving vector is stored in a temporary buffer and the processor unpacks it first when it is needed. The communication measured in the table below is the sending and receiving operations. However there are hidden communication, when a vector arrives to a processor it is stored in a temporary buffer, this takes processor time. The multiplication and orthogonalisation seems to take longer time as the operations are measured by a real clock. This algorithm did not work on the IBM-SP2. It looks like that it is due to a bug in the PVMe, the implementation of PVM on the IBM-SP2.

Test Results Parallel Program 2B				
Configuration	SUN	SUN	SUN	SUN
Test No	11	12	13	14
No of Processors	3	6	3	2
Shifts	3	6	6	6
Iterations with each shifts	30	25	25	25
Total no of Iterations	90	150	150	150
Timing Results On One Processor (in seconds)				
One factorisation	23.6	23.8	22.7	23.1
One multiplication	0.73	0.808	0.712	0.691
Factorisation total	23.6	23.8	45.4	69.2
Multiplication total	22.1	20.2	35.6	51.8
Orthogonalisation total	13.1	19.4	37.2	55.2
Communication measured	3.1	5.2	5.6	6.7
Other	1.5	1.6	2.5	3.5
Total time	63.4	70.2	126.3	186.4
Sequential Program	175.8	351.8	351.8	351.8
Speedup	2.77	5.01	2.79	1.89
Efficiency	92.4%	83.5%	92.9%	94.4%

The plot below shows speedup from the tests using 6 shifts on the SUN workstations. The straight line is the ideal speedup, the pluses + are the speedup from the tests on program 2A and the circles ∘ are the speedup from the tests on program 2B.



The graph below shows idle time, working time and communication for the SUN configuration. First every processor factorises, only part of the factorisation time is shown in the figure. Then at each iteration each processor multiplies, orthogonalises its own vector and exchanges the vector with the other processors. A particular processor unpacks a vector first when it is needed, The idle time is reduced by this procedure compared to the parallel program 2A.



In comparing the timing results with parallel program 2A this implementation get considerably better results due to the different way to exchange the vectors as discussed before.

5.7 Convergence

In this section we will discuss the convergence aspects of the algorithms. The different parallel algorithms differ mainly in the communication aspects. The only difference in the numerical aspects is in the orthogonalisation process. The different parallel algorithms get almost the same numerical behaviour. The big difference is between the sequential algorithm and the parallel algorithms. Here we will discuss the difference in the numerical aspects between the parallel algorithms and the sequential algorithm.

Test Matrices

In these tests we have chosen $\boldsymbol{A} = diag(1:500)$, $\boldsymbol{B} = \boldsymbol{I}$.

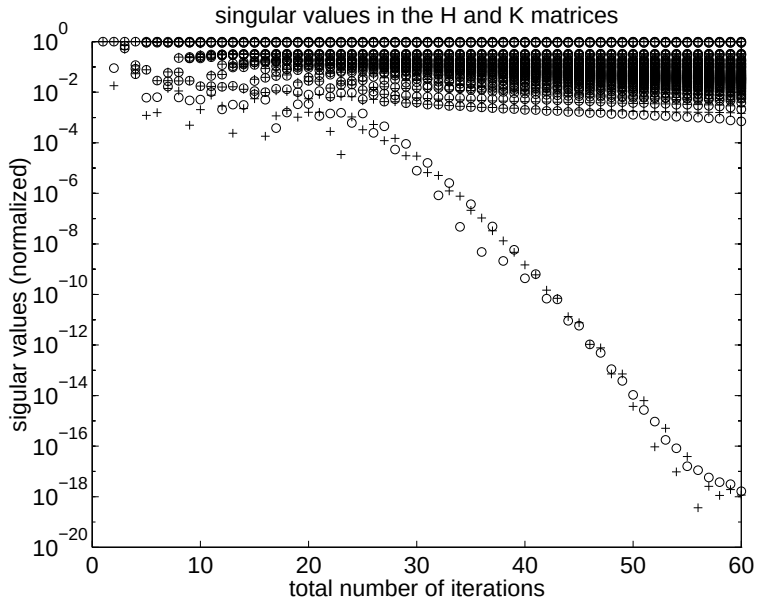
Tests

The tests shows that the matrices $\boldsymbol{K}$ and $\boldsymbol{H}$ are more unstable numerically in the parallel algorithm than in the sequential algorithm.

As the first two tests we choose 2 shifts, one sequential test and one parallel test.

Tests		
Test No	15	16
No of Processors	2	1
Shifts	2	2
Iterations with each shifts	30	30
Total no of Iterations	60	60
μ_1	100.5	100.5
μ_2	110.5	110.5

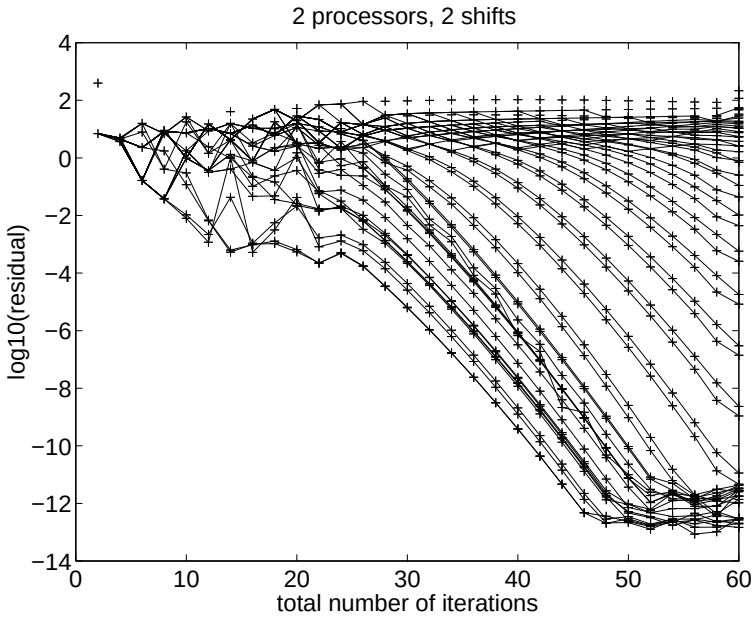
The graph below shows the normalised singular values for the matrices $\boldsymbol{H}$ and $\boldsymbol{K}$ as the number of iterations progresses for test 15, the parallel test. The pluses $+$ are the singular values of the matrix $\boldsymbol{H}$ and the circles $\circ$ are the singular values of the matrix $\boldsymbol{K}$.



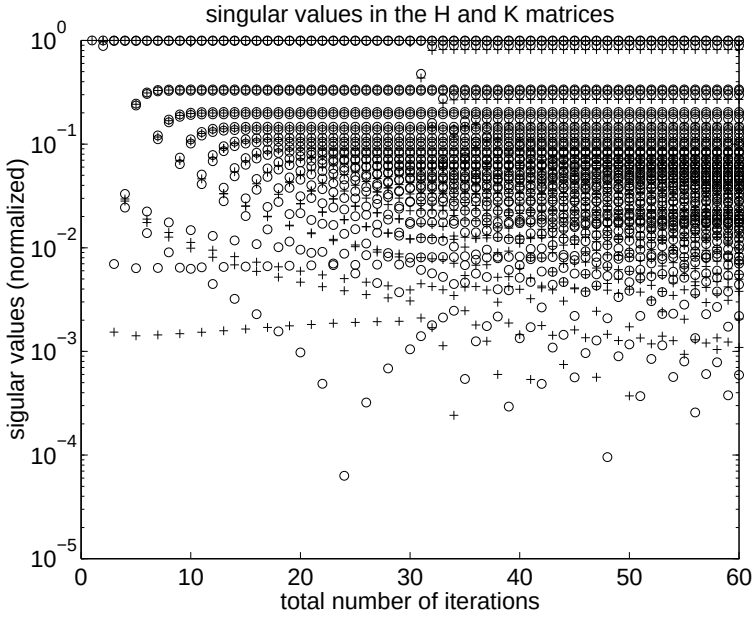
The test shows that the matrices $\boldsymbol{H}$ and $\boldsymbol{K}$ get a null space of dimension one in this case. A singular value decomposition of the matrices shows that they get the same null space in this case.

However if we take into consideration the null space when the approximate eigenvalues are calculated we get the same convergence as in the sequential case, see appendix how this is done.

The graph below shows the convergence for the parallel case.



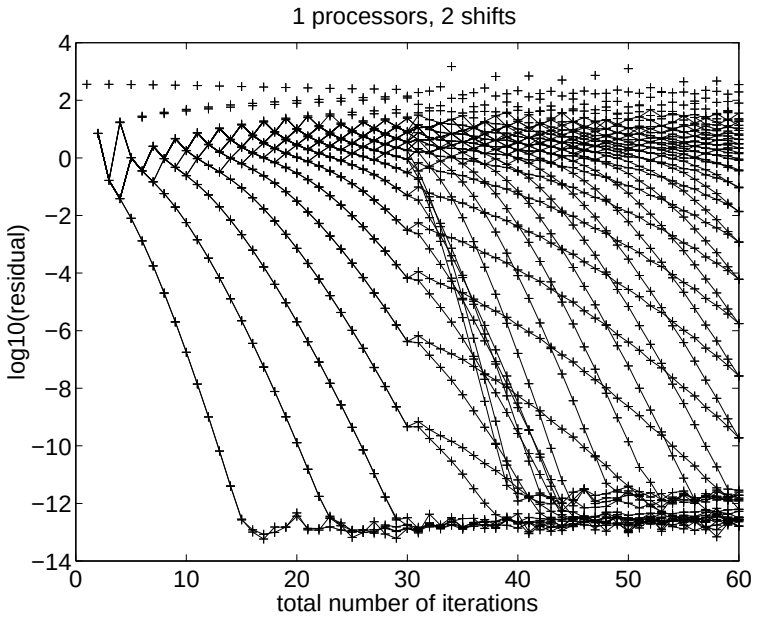
The graph below shows the normalised singular values for the $\boldsymbol{H}$ and the $\boldsymbol{K}$ matrices as the number of iterations progresses for test 16, the sequential test. The + sign are the singular values of $\boldsymbol{H}$ and ∘ are the singular values of $\boldsymbol{K}$.



Note in the graph above that we do not get a null-space in the matrices $\boldsymbol{H}$

and $\boldsymbol{K}$.

The graph below shows the convergence for the sequential case.

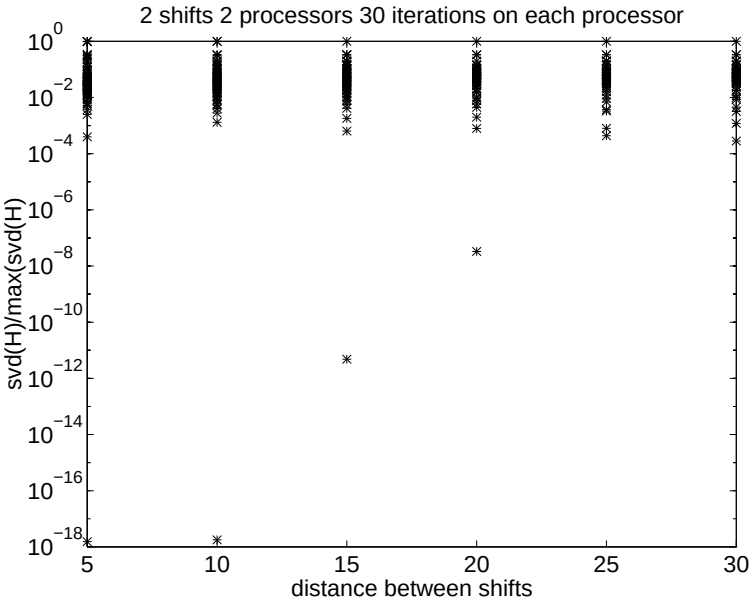


From the convergence graphs for the sequential and parallel algorithms we can conclude that we get almost the same convergence. The null-space in the $\boldsymbol{H}$ and $\boldsymbol{K}$ matrices in the parallel algorithm does not affect the convergence if dealt with properly in this case.

Distance Between the Shifts

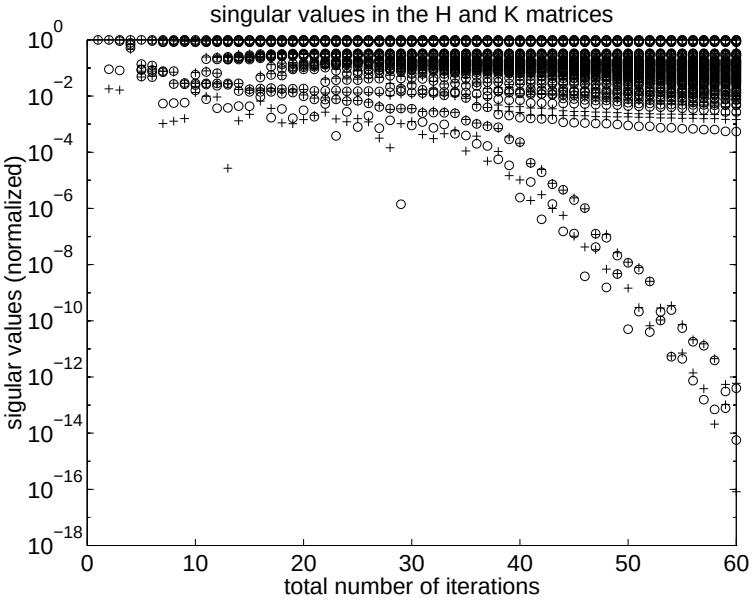
How does the distance between the shifts affect the singular values of the matrices if the number of iterations is kept constant? If the distance is zero we get the same shifts and linear dependence between the basis vectors.

In the graph below we have tested how the singular values in the $\boldsymbol{H}$ matrix for the parallel algorithm depends on the distance between the shifts. The shifts are $\mu_1 = 100.5$ and $\mu_2 = 100.5 + 5 * j, j = 1, \ldots, 6$. The total number of iterations is 60, 30 iterations on each processor.



As we can see in the above graph, if we fix the number of iterations and increase the distance between the shifts, the smallest singular value of $\boldsymbol{H}$ increases in the parallel algorithm.

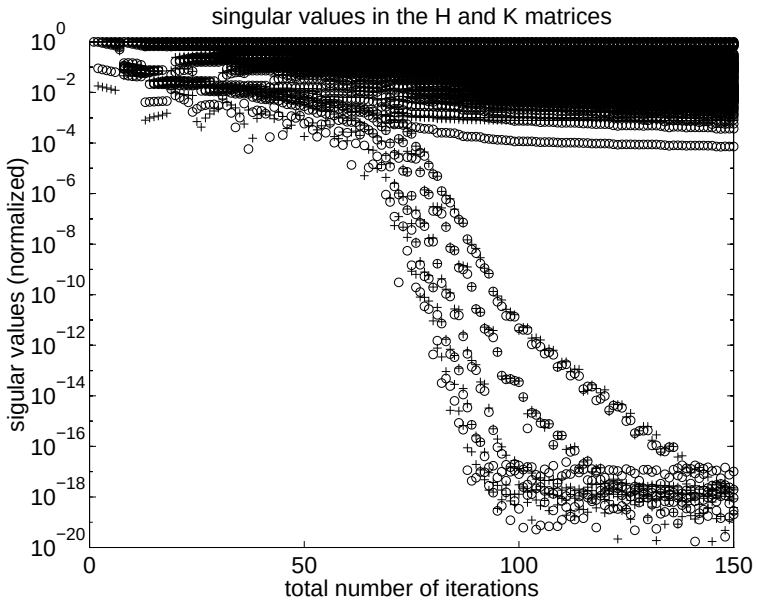
How large is the null space of $\boldsymbol{H}$ and $\boldsymbol{K}$ as the number of processors increases. In the graph below we have used 3 processors, 3 shifts and 20 iterations on each processor. The shifts are $\mu_1 = 100.5$, $\mu_2 = 110.5$ and $\mu_3 = 120.5$. The $+$ sign are the singular values of $\boldsymbol{H}$ and $\circ$ are the singular values of $\boldsymbol{K}$.



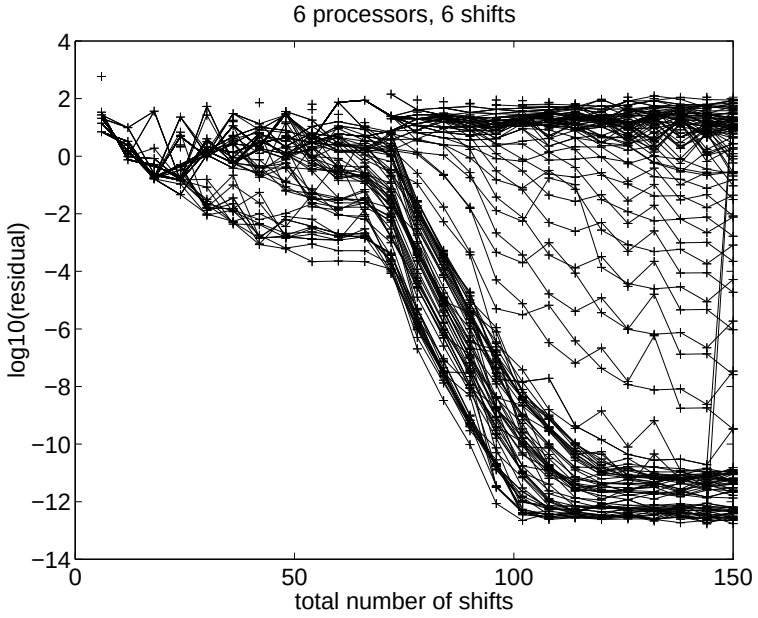
From the above test we get a null space of dimension two for the matrices $\boldsymbol{H}$ and $\boldsymbol{K}$. A singular value decomposition shows that the null space of $\boldsymbol{H}$ and

$\boldsymbol{K}$ are the same for this case.

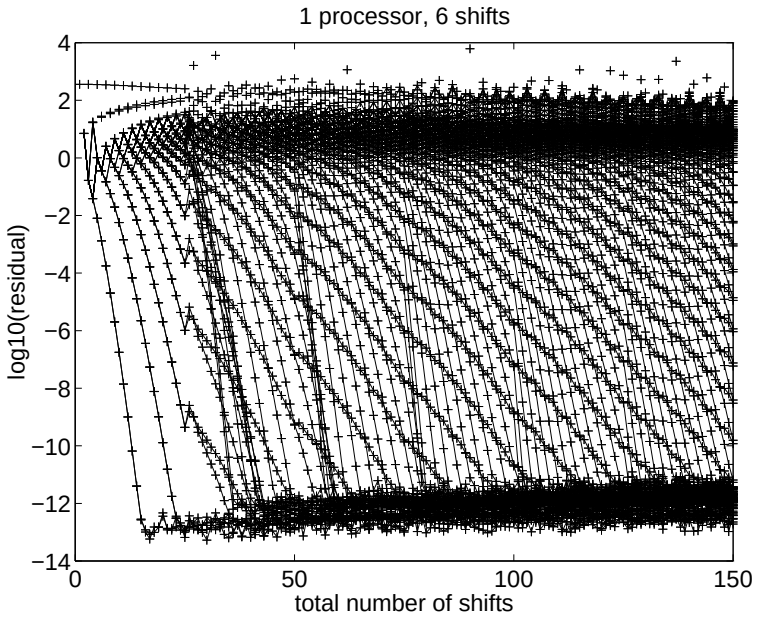
If we use 6 processors, how large is the null space for $\boldsymbol{H}$ and $\boldsymbol{K}$? In the graph below we have used 6 processors 6 shifts and 25 iterations with each shift. The shifts are $\mu_i = 100.5 + 10 * (j-1),\ j = 1, \ldots, 6$.



Note that the graph above that the nullspace is of dimension 5 for the test. The graph below shows the convergence for the above test.



The graph below shows the convergence for the corresponding sequential algorithm.



If we compare the residuals from the tests we get 57 residuals with a norm less than 10^{-10} for the parallel algorithm and 78 for the sequential algorithm.

Below we give a summary and conclusions for the convergence tests. There remains work to done before theorems and proofs can be stated from these conclusions.

1. If $\mu_1, \ldots, \mu_p$, $\mu_i \neq \mu_j$ shifts are chosen and applied in parallel with the parallel rational Krylov method with the same number of iterations with each shift then the matrices $\boldsymbol{H}$ and $\boldsymbol{K}$ get $p-1$ singular values that are significantly smaller than the others. As the number of iterations progresses, these singular values can be considered zero and the matrices $\boldsymbol{H}$ and $\boldsymbol{K}$ have the same null space of dimension $p-1$.

2. If $\mu_1, \ldots, \mu_p$, $\mu_i \neq \mu_j$ shifts are chosen initially and applied in parallel with the parallel rational Krylov method and the number of iterations are fixed and equal on each processor. Then the matrices $\boldsymbol{H}$ and $\boldsymbol{K}$ will have $p-1$ singular values that are significantly smaller than the others when the shifts are close enough. As the distance between the shifts increases the smallest singular value increases.

5.7.1 Some Possible Explanations of the Convergence of the Parallel Algorithm

Why do the parallel and the sequential algorithm behave differently? According to the theory, in exact arithmetic they build up the same space. Somehow the floating point arithmetic make them behave differently.

One possible explanation is in the way the basis is built up. In the parallel version different Krylov spaces are intermixed to build up a larger space, while in the sequential version one Krylov space at a time is added to build up a larger space. If the shifts are the same, then the parallel version stops in the orthogonalisation step in the first iteration because the different processors produce identical vectors. But for the sequential version if the shifts are the same the algorithm becomes the shifted and inverted Arnoldi algorithm and does not break down. What happens when the shifts are close but not the same? We have some idea from the tests.

Another explanation lies in the orthogonalisation process. The sequential algorithm builds up the basis one vector at a time, while the parallel version builds up the basis with p vectors at a time. The parallel version operate on one processor

$$\boldsymbol{r}_{i+p} = (\boldsymbol{A} - \mu_i \boldsymbol{B})^{-1} \boldsymbol{B} \boldsymbol{v}_i$$

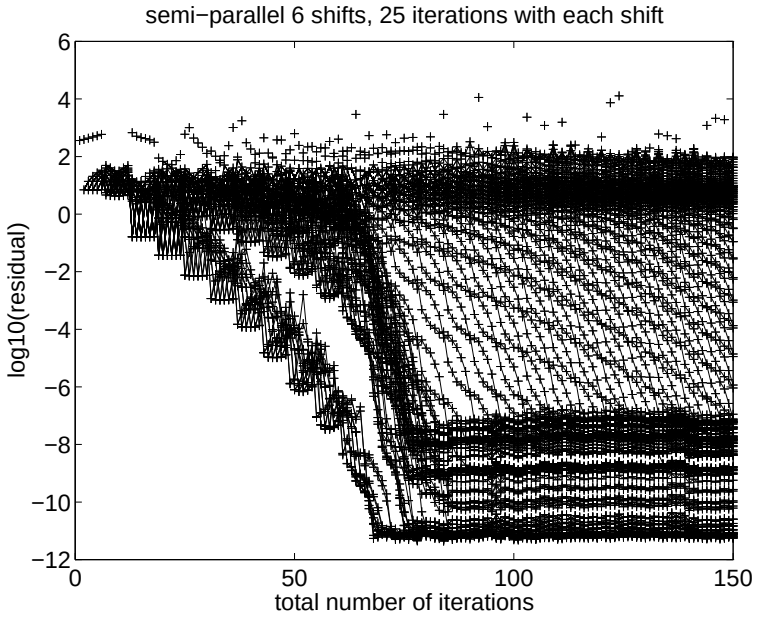
$\boldsymbol{r}_{i+p}$ is orthogonalised against $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{i+p-1}$. The vectors $\boldsymbol{v}_{i+1}, \ldots, \boldsymbol{v}_{i+p-1}$ cannot be included in the the starting combinations because they are not calculated when the processor needs them. This process could cause numerical instability.

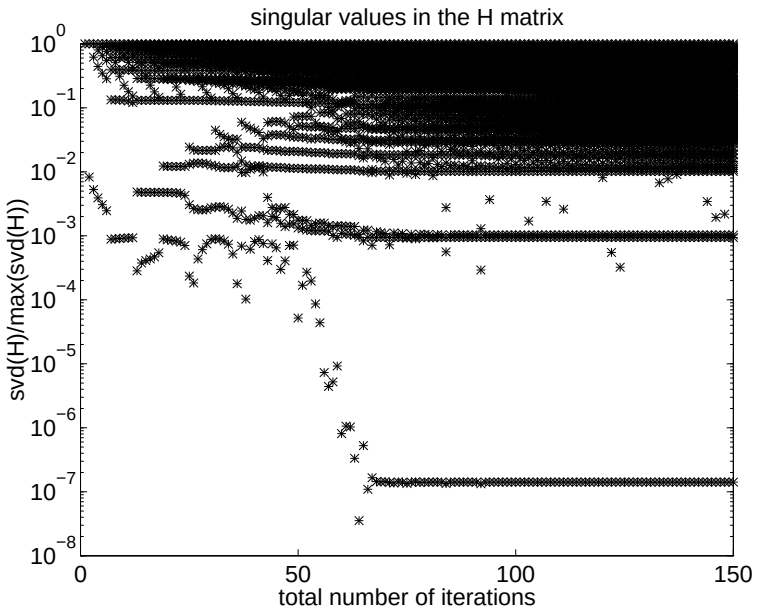
Semi Parallel Algorithm

These tests were done after a suggestion by R.B. Lehoucq [1]. Here we use the same intermixing of the spaces as the parallel algorithm and the same starting combination as the sequential algorithm. We operate

$$\boldsymbol{r}_{i+1} = (\boldsymbol{A} - \mu_i \boldsymbol{B})^{-1} \boldsymbol{B} \boldsymbol{v}_i$$

and diag(μ)= diag($\mu_1, \ldots, \mu_p, \mu_1, \ldots, \mu_p, \ldots$). In the tests below we have used 6 semi processors and 6 shifts $\mu_i = 100.5 + 10 * (j-1)$, $j = 1, \ldots, 6$. in the semi parallel algorithm



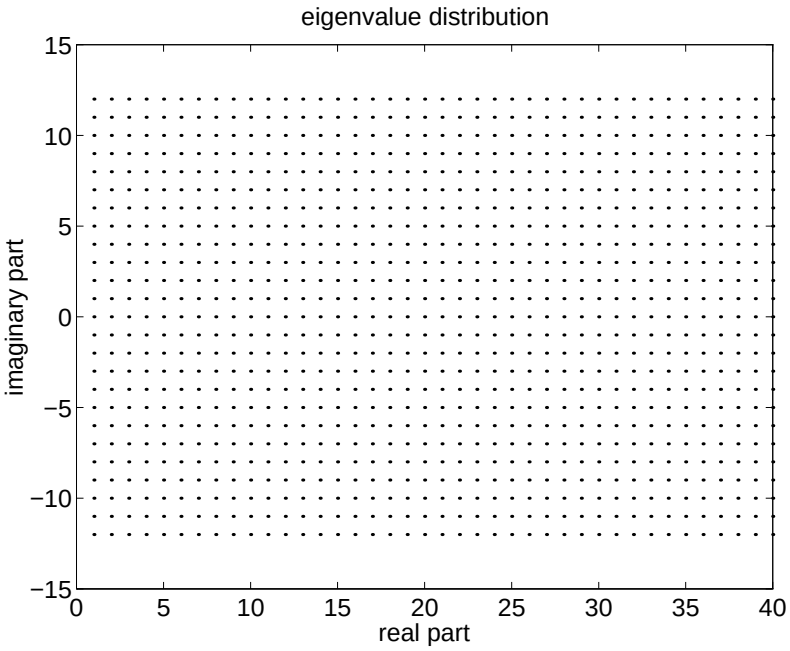


If we compare the plots above with the corresponding plots for the sequential algorithm and the parallel algorithm we can see that the semi parallel program has the largest residuals and the $\boldsymbol{H}$ matrix for the semi parallel algorithm do not have the $p-1$ small singular values as the $\boldsymbol{H}$ matrix for the parallel program has.

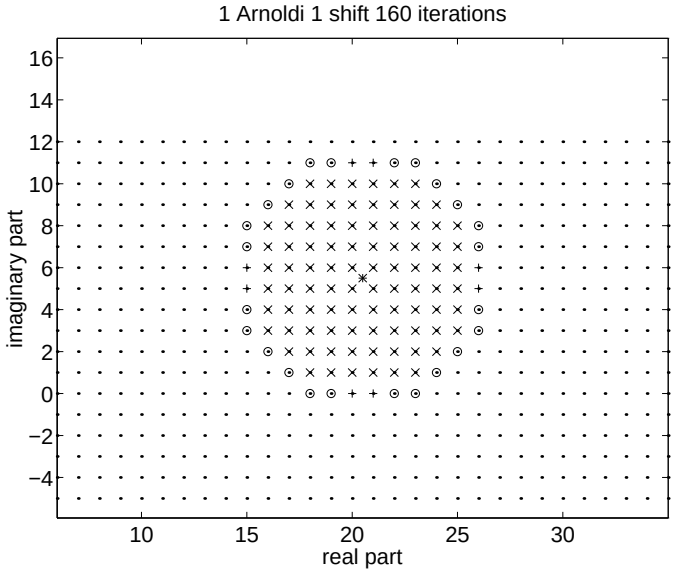
5.8 The Rational Krylov Method Compared to the Shift and Invert Arnoldi Method

In this section we compare the rational Krylov method with the shift and invert Arnoldi method. We compare the number of converged eigenvalues and the shape of the convergence region in the complex plane with respect to the total number of iterations. We compare the methods in two different ways. First we compare the rational Krylov method with s shifts to shift and invert Arnoldi method with one shift. Second we compare the rational Krylov method with s shifts to applying shifted and inverted Arnoldi method s times with the same shifts. The algorithms are run the same number of steps with all the shifts. In most tests the total number of steps is 160.

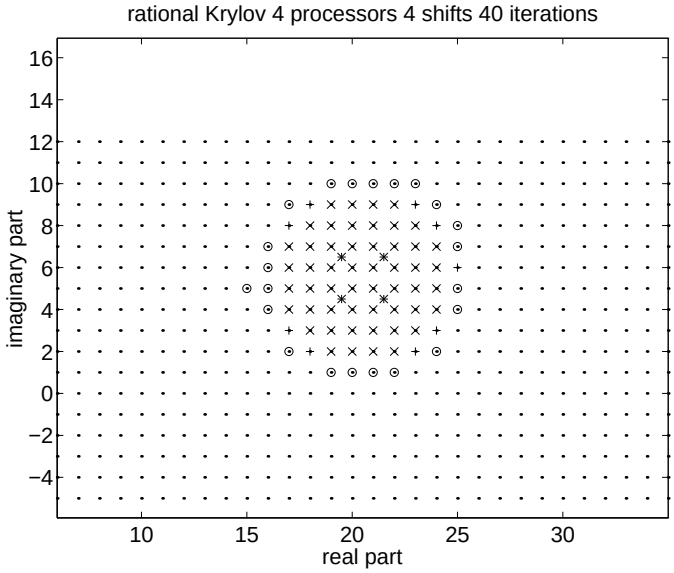
In these tests, we have used a diagonal matrix $\boldsymbol{A}$ and unit $\boldsymbol{B}$ with eigenvalues in the complex integers.



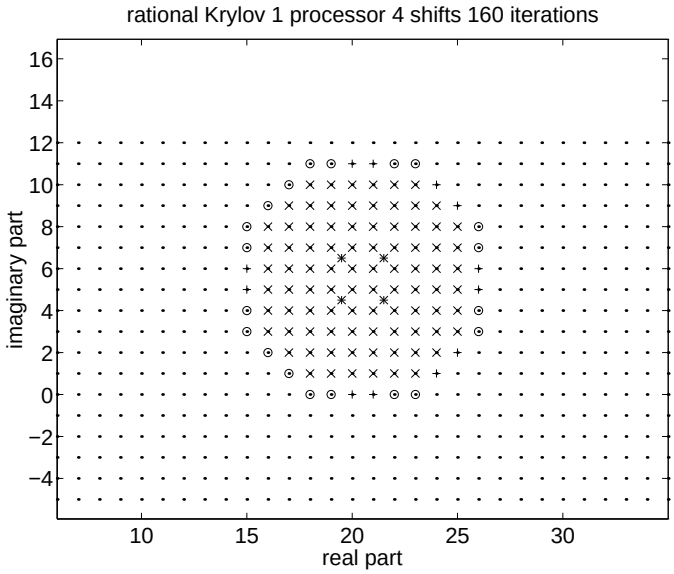
The graph below shows the converged eigenvalues for one shift and invert Arnoldi method with 160 iterations. Approximate eigenvalues with a relative error between 10^{-5} and 10^{-8} are plotted with a circle $\circ$. Approximate eigenvalues with a relative error between 10^{-8} and 10^{-11} are plotted with a plus $+$. Approximate eigenvalues with a relative error below 10^{-11} are plotted with a cross $\times$. The shift is plotted with a star $*$. The exact eigenvalues are plotted with a dot $\cdot$.



The graph below shows converged eigenvalues for the parallel rational Krylov method.

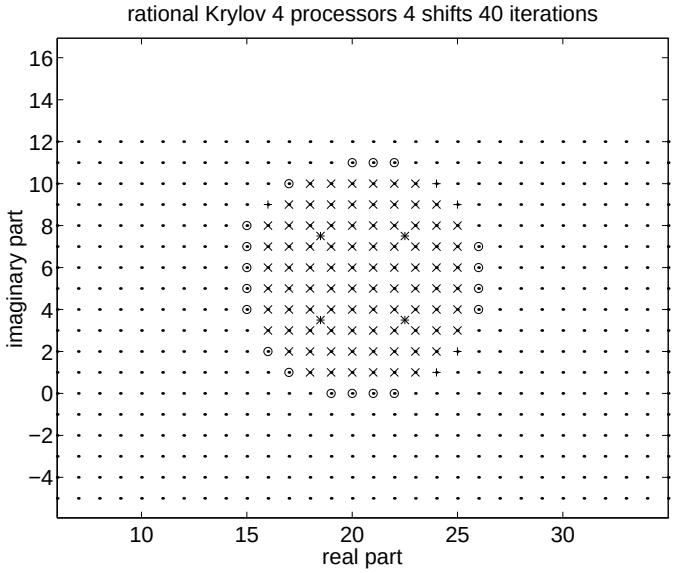


The graph below shows converged eigenvalues for the sequential rational Krylov method.



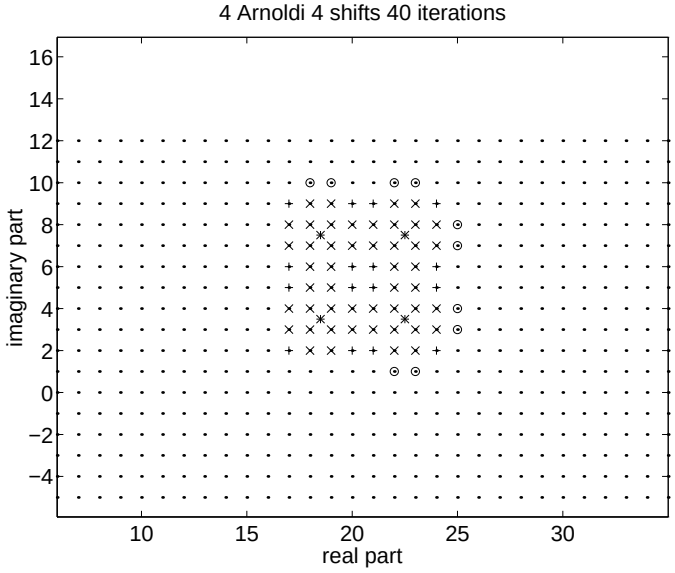
If we compare the three graphs above we note that the Arnoldi method and the sequential rational Krylov method obtains almost the same number of converged eigenvalues. The parallel rational Krylov method obtains considerable fewer converged eigenvalues in this case.

Next we run the parallel rational Krylov method with a larger separation between the shifts than above.

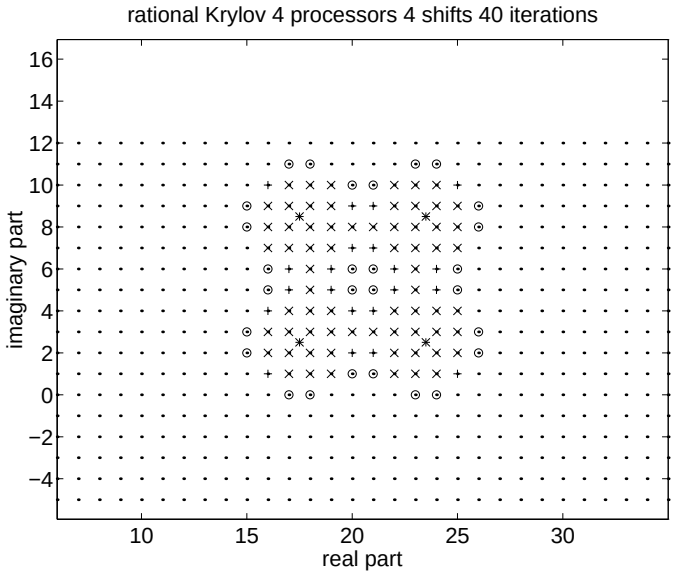


In this case the parallel rational Krylov method obtains similar convergence as the Arnoldi method and the sequential rational Krylov method (graph not shown).

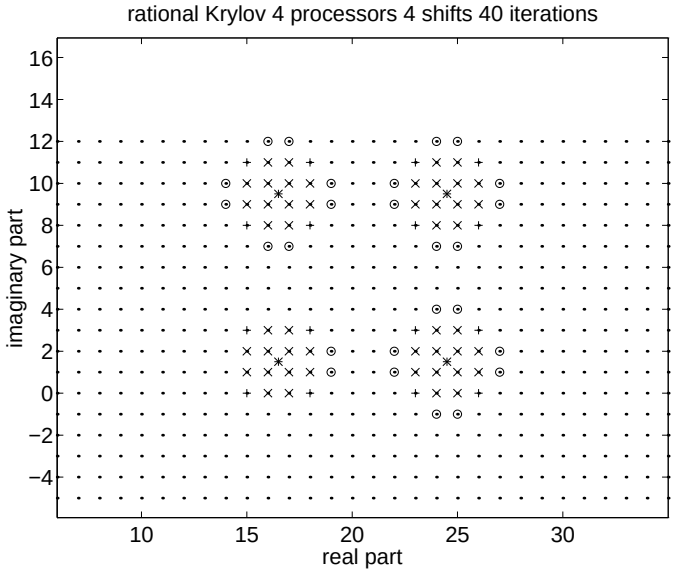
Now let us compare the rational Krylov method with 4 shifts to 4 separate runs of shifted and inverted Arnoldi with the same shifts. The convergence is as plotted below.



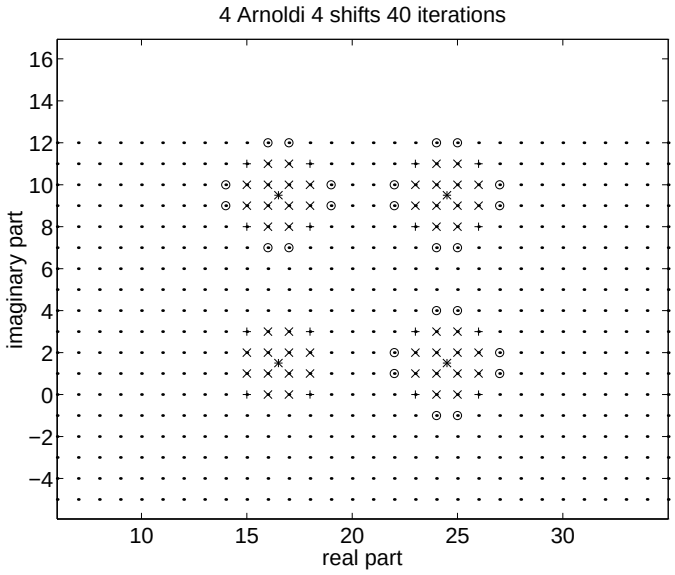
What happens with the rational Krylov method if we increase the distance between the shifts further? The graph below shows convergence for the parallel rational Krylov method. Note that the convergence is not so good in the center as before (the sequential version obtains similar convergence in this case).



If we increase the distance between the shifts further in the parallel rational Krylov method we get 4 different regions with converged eigenvalues, see the graph below (the sequential version obtains similar convergence in this case).

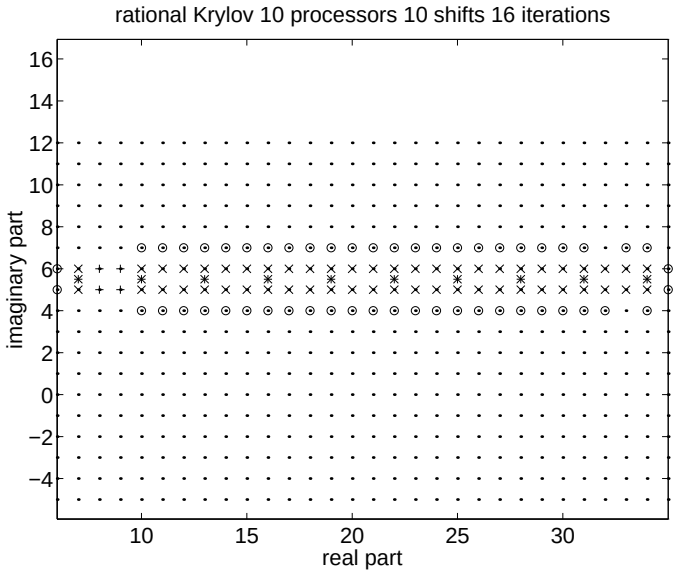


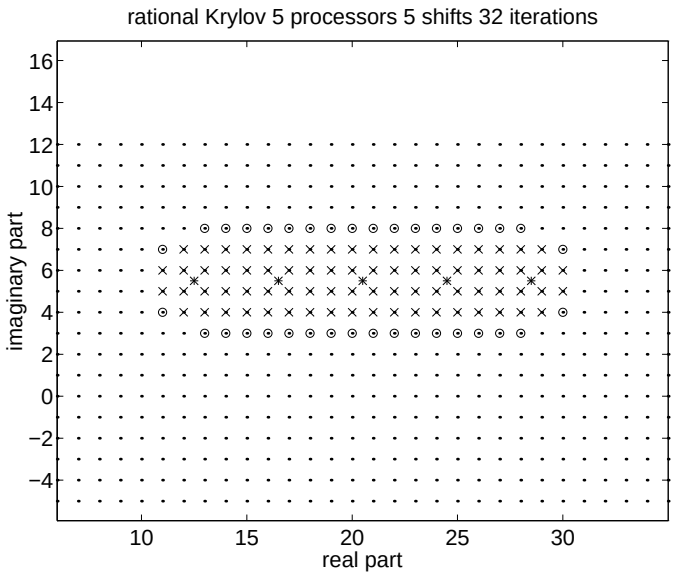
If we compare the graph above for the parallel rational Krylov method with 4 shift and invert Arnoldi methods in the graph below,



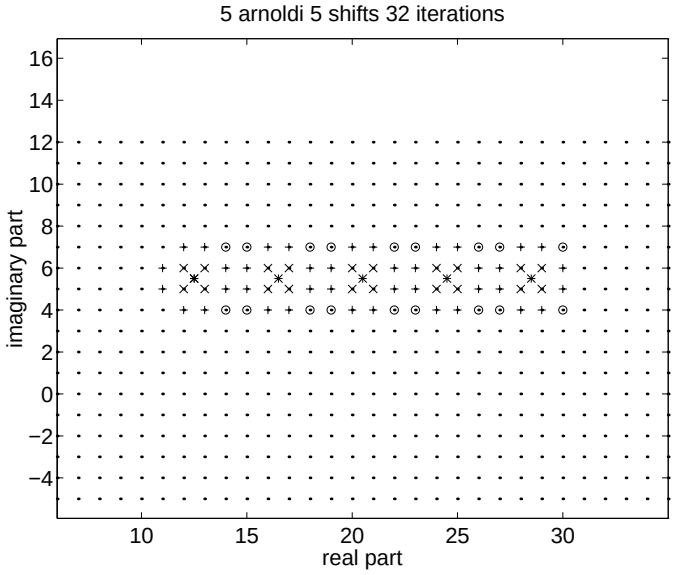
we note that now the rational Krylov method and the Arnoldi method get similar convergence.

One advantage with the (parallel) rational Krylov method is that the shifts can be chosen to mark up other regions than circles.

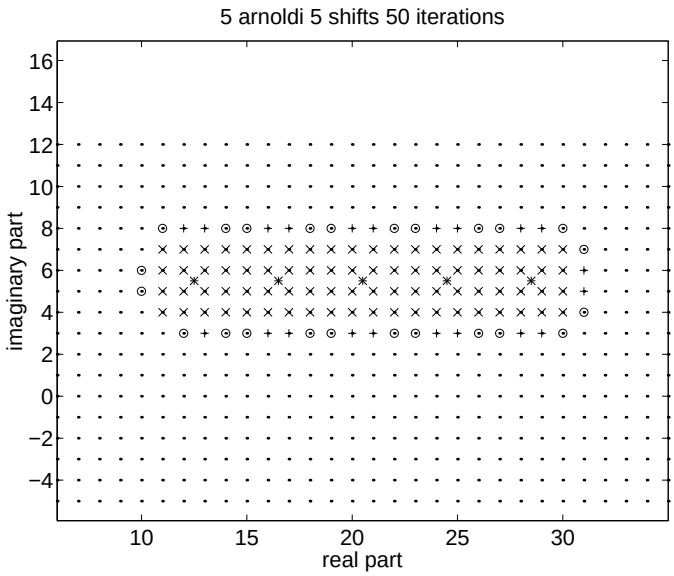




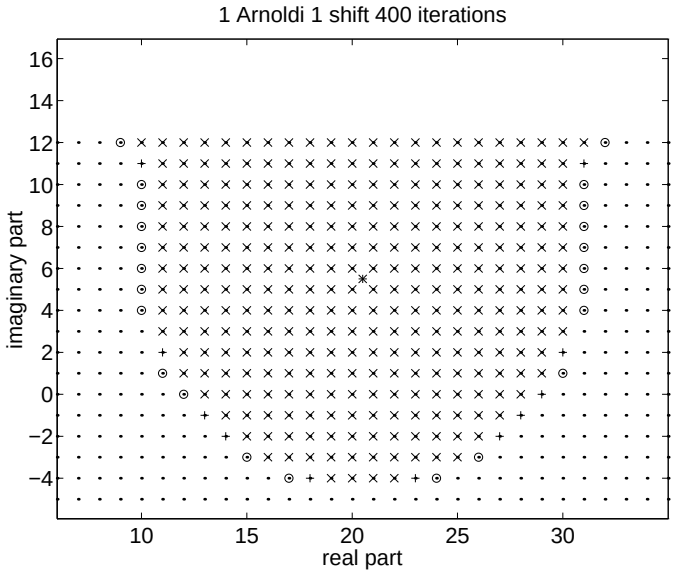
The corresponding convergence test for the 5 shift and invert Arnoldi methods to the test above is shown in the graph below.



If we increase the number of iterations on each processor to 50 for the 5 shift and invert Arnoldi methods we get similar convergence as in the rational Krylov test with 5 shifts and 32 iterations, see the graph below.



We need 400 iterations with one shift and invert Arnolid method in order to obtain a convergence region that encloses the above 5 shift convergence region for the rational Krylov method. In this case with the shift and invert Arnoldi method we get many more eigenvalues than we need in order to obtain them we want, see the graph below.



We can make some conclusions from the tests above.

1. It is possible to obtain similar convergence with several shifts in (parallel) rational Krylov method as with one shift and invert Arnoldi method if the

total number of iterations are the same in both methods.

2. The choice of the shifts is crucial for the convergence of the rational Krylov method. The parallel version is more sensitive to the choice of the shifts than the sequential version. The shifts should be close enough so that they help each other to build a common space and in the parallel version they should not be too close so that the method degenerates.

3. The rational Krylov method has more freedom of choice for obtaining the convergence region than shift and invert Arnoldi.

4. The rational Krylov method with s shifts converges better than the s shift and invert Arnoldi methods with corresponding shifts, if the shifts are close enough that the convergence region is continuous in the rational Krylov method. If the shifts are so far apart so that the convergence region form islands with converged eigenvalues around the shifts in the rational Krylov method, then the rational Krylov method with s shifts and s shift and invert Arnoldi have similar convergence behaviour.

In the tests above, we have compared the convergence with respect to the total number of iterations. Which method is most suitable for a given problem depends on the region where the eigenvalues are desired, which solvers are used and the relation between the orthogonalisation time and multiplication time.

The type of problem where the rational Krylov method is likely to perform better than a standard shift and invert Arnoldi, is where the desired eigenvalue region differ to great extent from the typical circular convergence region to the shift and invert Arnoldi. If a standard shift and invert Arnoldi is applied to such a problem, then a larger convergence region that includes the desired eigenvalues is obtained and thus a larger basis is built than in the rational Krylov method. A large basis takes memory, and it takes time to add a new vector to the basis through the orthogonalisation process. If the rational Krylov method with s shifts performs better than s shift and invert Arnoldi methods depends on the time for multiplications and orthogonalisation. Another advantage is that the rational Krylov method does not need any post sorting of the eigenvalues that the s shift and invert Arnoldi methods need.

5.9 Shift Strategies

The rational Krylov method is suited for problems where we want to compute the eigenvalues and corresponding eigenvectors in a specific region in the complex plane. Shift strategies is an important part of an implementation of the rational Krylov method. With a shift strategy, we want to place the shifts in a specific region in the complex plane in a such a way that the eigenvalues with corresponding eigenvectors are computed in the region to sufficient accuracy. Another important issue with a shift strategy is to avoid to compute too many eigenvalues because this takes time and build up a unnecessarily large basis.

A shift strategy for the sequential rational Krylov method could be like place the first shift close but not to close to the border in the region where the eigenvalues are wanted. After n_{μ_1} eigenvalues with corresponding eigenvectors have converged a new shift could be placed close to the region where we have

converged eigenvalues with corresponding eigenvectors. We keep the second shift until n_{μ_2} additional eigenvalues have converged. The placement of new shift goes on until all eigenvalues with corresponding eigenvectors have converged in the region where we want them. How long one should keep a shift depends on how much time it takes to change the shift compared to the gain with a new shift. If iterative solvers are used the time it takes to change the shift is negligible. But if direct solvers are used the factorisation can take a lot of time. There is no guarantee that all eigenpairs have converged and we get only one eigenvector to eigenvalues with several eigenvectors. See Ruhe [16] for a implementation of a shift strategy.

An implementation of a shift strategy in parallel rational Krylov method is much more difficult. In the sequential version the knowledge about where we have converged eigenvalues and nearly converged eigenvalues is valuable information, when the decision where the new shift should be placed is made. What we want to do in the parallel version is to place them in such a way in the region where the eigenvalues are wanted, that when the convergence region of the different shifts have grown together, all eigenvalues in the region where we want them have converged. But this strategy needs a knowledge of the eigenvalue distribution that we are going to compute. For most cases we do not know the eigenvalue distribution before we compute them, but any information about the eigenvalue distribution could be useful in placing the shifts in a good way. A possible strategy is to place them evenly spread out in the region where we want the eigenvalues and iterate until the convergence regions around the shifts have grown together. Shift strategies has not been emphasised in the implementation discussed in this report. But for a commercial or public domain software of parallel rational Krylov method, shift strategies are important.

Appendix A

Handling the Null Space of the Matrices K and H

From numerical investigations we have that $\boldsymbol{K}_{m,m}$ and $\boldsymbol{H}_{m,m}$ have a common null space of dimension $p-1$, where p is the number of processors. The singular value decomposition of the matrix $\boldsymbol{H}$ is as follows

$$\boldsymbol{U}^* \boldsymbol{H}_{m,m} \boldsymbol{W} = \Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_m) \tag{A.1}$$

where

$$\sigma_1 > \sigma_2 > \ldots > \sigma_{m-p+1} > \sigma_{m-p+2} = \ldots = \sigma_m = 0$$

The nullspace is spaned by $\boldsymbol{w}_{m-p+2}, \ldots, \boldsymbol{w}_m$. Now consider the eigenvalue problem

$$\boldsymbol{K}_{m,m} \boldsymbol{x} = \lambda \boldsymbol{H}_{m,m} \boldsymbol{x} \tag{A.2}$$

$$\boldsymbol{K}_{m,m} \boldsymbol{W} \boldsymbol{W}^* \boldsymbol{x} = \lambda \boldsymbol{H}_{m,m} \boldsymbol{W} \boldsymbol{W}^* \boldsymbol{x}$$

multiply with $\boldsymbol{U}^*$

$$\boldsymbol{U}^* \boldsymbol{K}_{m,m} \boldsymbol{W} \boldsymbol{W}^* \boldsymbol{x} = \lambda \boldsymbol{U}^* \boldsymbol{H}_{m,m} \boldsymbol{W} \boldsymbol{W}^* \boldsymbol{x}$$

set $\hat{\boldsymbol{K}} = \boldsymbol{U}^* \boldsymbol{K}_{m,m} \boldsymbol{W}$ and $\hat{\boldsymbol{x}} = \boldsymbol{W}^* \boldsymbol{x}$ and use (A.1) to get the eigenproblem

$$\hat{\boldsymbol{K}} \hat{\boldsymbol{x}} = \lambda \Sigma \hat{\boldsymbol{x}}$$

where

$$\hat{\boldsymbol{K}} = \begin{bmatrix} \hat{\boldsymbol{K}}_{1,1} & 0 \\ \hat{\boldsymbol{K}}_{2,1} & 0 \end{bmatrix}, \ \hat{\boldsymbol{K}}_{1,1} \in \mathcal{C}^{(m-p+1)\times(m-p+1)}, \ \hat{\boldsymbol{K}}_{2,1} \in \mathcal{C}^{(p-1)\times(m-p+1)}$$

and

$$\Sigma = \begin{bmatrix} \Sigma_{1,1} & 0 \\ 0 & 0 \end{bmatrix}, \ \Sigma_{1,1} = diag(\sigma_1, \ldots, \sigma_{m-(p-1)})$$

We solve the standard eigenproblem

$$\Sigma_{1,1}^{-1} \hat{\boldsymbol{K}}_{1,1} \hat{\boldsymbol{x}}_1 = \lambda \hat{\boldsymbol{x}}_1$$

If $(\hat{\boldsymbol{x}}_1, \lambda)$ is an eigenpair of $\Sigma_{1,1}^{-1} \hat{\boldsymbol{K}}_{1,1}$ then $(\boldsymbol{W}_1 \hat{\boldsymbol{x}}_1, \lambda)$ is an eigenpair to (A.2), where $\boldsymbol{W} = [\boldsymbol{W}_1, \boldsymbol{W}_2]$, $\boldsymbol{W}_1 \in \mathcal{C}^{m\times(m-p+1)}$.

Bibliography

[1] In private communication with R.B Lehoucq Mathematics and Computer Science Division Argone National Laboratory, Argone IL.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Release 2.0.* SIAM, Philadelphia, 1995. 324 pages.

[3] W.E. Arnoldi. The principle of minimized iteration in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9:17–29, 1951.

[4] Z. Bai, D. Day, J. Demmel, and J. Dongarra. The matrix collection (non-hermitian eigenvalue problems). Available by anonymous ftp to ftp.ms.uky.edu in the directory pub/misc/bai/Collection.

[5] J.G.L. Booten, P.M. Meijer, H.J.J. te Riele, and H.A. Van der Vorst. Parallel Arnoldi Method for the Construction of a Krylov Subspace Basis: an Application in Magnetohydrodynamics. In *International conference and exhibition on High Performance computing and networking, Munich, Germany, April 18-20, 1994*. Springer Verlag, 1994.

[6] T. Bräunl. *Parallel Programming An Introduction.* Prentice Hall, 1993.

[7] T.L. Freeman and C. Phillips. *Parallel Numerical Algorithms.* Prentice Hall, 1992.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[9] IBM. *IBM AIX PVMe User's Guide and Subroutine Reference Release 3.1.*

[10] M. N. Kooper, H. A. Van Der Vorst, S. Poedts, and J.P. Goedbloed. Application of the Implicitly Updated Arnoldi Method with a Complex Shift-and-Invert Strategy in MHD. *Journal of Computational Physics*, 118:320–328, 1995.

[11] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research, Nat. Bu. of Standards*, 45:255–282, 1950.

[12] B. N. Parlett. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[13] Axel Ruhe. Rational Krylov sequence methods for eigenvalue computation. *Lin. Alg. Appl.*, 58:391–405, 1984.

[14] Axel Ruhe. Rational Krylov algorithms for nonsymmetric eigenvalue problems, II: Matrix pairs. *Lin. Alg. Appl.*, 197/198:283–296, 1994.2.

[15] Axel Ruhe. The Rational Krylov algorithm for nonsymmetric eigenvalue problems. III: Complex shifts for real matrices. *BIT*, 34:165–176, 1994.3.

[16] Axel Ruhe. Rational Krylov, a practical algorithm for large sparse nonsymmetric matrix pencils. Technical Report UCB/CSD-95-871, Dept Computer Science, University of California Berkeley, 1995.

[17] Y. Saad. *Numerical Methods For Large Eigenvalue Problems.* Manchester University Press, 1992.

[18] Arthur Trew and Greg Wilson. *Past, Present, Parallel A Survey of Available Parallel Computing Systems.* Springer - Verlag, 1991.