UPPSALA
UNIVERSITET

22038

Examensarbete 15 hp
Juni 2022

Exploring and extending eigensolvers for Toeplitz(-like) matrices

A study of numerical eigenvalue and eigenvector computations combined with matrix-less methods

Fredrik Cers
Oliver Groth
Martin Knebel

Abstract

Exploring and extending eigensolvers for Toeplitz(-like) matrices

F. Cers, O. Groth, M. Knebel

We implement an eigenvalue solving algorithm proposed by Ng and Trench, specialized for Toeplitz(-like) matrices, utilizing root finding in conjunction with an iteratively calculated version of the characteristic polynomial. The solver also yields corresponding eigenvectors as a free bi-product. We combine the algorithm with matrix-less methods in order to yield eigenvector approximations, and examine its behavior both regarding demands for time and computational power.

The algorithm is fully parallelizable, and although solving of all eigenvalues to the bi-Laplacian discretization matrix - which we used as a model matrix - is not superior to standard methods, we see promising results when using it as an eigenvector solver, using eigenvector approximations from standard solvers or a matrix-less method. We also note that an advantage of the algorithm we examine is that it can calculate singular, specific eigenvalues (and the corresponding eigenvectors), anywhere in the spectrum, whilst standard solvers often have to calculate all eigenvalues, which could be a useful feature.

We conclude that - while the algorithm shows promising results - more experiments are needed, and propose a number of topics which could be studied further, e.g. different matrices (Toeplitz-like, full), and looking at even larger matrices.

Handledare: Sven-Erik Ekström
Ämnesgranskare: Ken Welch
Examinator: Martin Sjödin
ISSN: 1401-5757, 22038

Contents

1 Introduction

A Toeplitz matrix has constant diagonal entries, and may be of any shape, however, in this report we always assume that it is square. Toeplitz matrices are named after the mathematician Otto Toeplitz, a German mathematician who was one of the first persons that started studying this type of structured matrices. Toeplitz matrices have important applications in discretization of partial differential equations, system theory, signal processing, and statistics, such as time series analysis, image processing. A Toeplitz matrix, denoted $T_n(f)$, can be written as

$$T_n(f) = [\hat{f}_{i-j}]_{i,j=1}^n = \begin{bmatrix} \hat{f}_0 & \hat{f}_{-1} & \hat{f}_{-2} & \cdots & \cdots & \hat{f}_{1-n} \\ \hat{f}_1 & \hat{f}_0 & \hat{f}_{-1} & \hat{f}_{-2} & & \vdots \\ \hat{f}_2 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \hat{f}_{-2} \\ \vdots & & \ddots & \hat{f}_1 & \hat{f}_0 & \hat{f}_{-1} \\ \hat{f}_{n-1} & \cdots & \cdots & \hat{f}_2 & \hat{f}_1 & \hat{f}_0 \end{bmatrix} \tag{1}$$

where n defines the size of the matrix, $T_n(f) \in \mathbb{C}^{n\times n}$, and f is the so-called generating symbol, or simply the symbol. Here, the entries $\hat{f}_k$ are the Fourier coefficients of the symbol $f(\theta)$,

$$\hat{f}_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(\theta)\mathrm{e}^{-\mathbf{i}k\theta}\mathrm{d}\theta, \quad f(\theta) = \sum_{k=-\infty}^{\infty} \hat{f}_k e^{\mathbf{i}k\theta}, \quad \mathbf{i}^2 = -1, k \in \mathbb{Z}.$$

To give a concrete example of what a Toeplitz matrix may look like, we have the identity matrix,

$$\mathbb{I}_n = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} = T_n(1), \tag{2}$$

In this case, all the diagonal entries are equal to 0, except for the main diagonal, where they are equal to 1. The identity is an example of a banded matrix, that is, only a finite number of Fourier coefficients, $\hat{f}_k$, of the symbol are non-zero, in this case $\hat{f}_0 = 1$, since the symbol $f(\theta) = 1$. Another example of a banded matrix is the Laplacian, given by second order finite difference approximation of the second derivative,

$$T_n(f) = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}, \tag{3}$$

which is generated by the symbol $f(\theta) = 2 - 2\cos(\theta) = -\mathrm{e}^{-\mathbf{i}\theta} + 2 - \mathrm{e}^{\mathbf{i}\theta}$, and thus, $\hat{f}_{-1} = \hat{f}_1 = -1$ and $\hat{f}_0 = 2$. The model matrix that will be used in this report is the bi-Laplacian discretization matrix,

$$T = \begin{bmatrix} 6 & -4 & 1 & & & & \\ -4 & 6 & -4 & 1 & & & \\ 1 & -4 & 6 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 6 & -4 & 1 \\ & & & 1 & -4 & 6 & -4 \\ & & & & 1 & -4 & 6 \end{bmatrix}, \tag{4}$$

where the symbol is $f(\theta) = (2 - 2\cos(\theta))^2 = 6 - 8\cos(\theta) + 2\cos(2\theta)$. This matrix approximates the fourth derivative. An example of a Toeplitz-like matrix is that we add boundary conditions to a matrix, for example instead of Dirichlet-Dirichlet in (3), we have Dirichlet-Neumann in the matrix below

$$A_n = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 1 \end{bmatrix} = T_n(f) + R_n. \tag{5}$$

Notice the 1 in the bottom right corner. This matrix can be written as a sum of a Toeplitz matrix, $T_n(f)$, and a low-rank correction matrix, R_n, where $f(\theta) = 2 - 2\cos(\theta)$ and R_n is the zero matrix with a -1 in the bottom right corner.

For every matrix A_n defined above (including $A_n = T_n(f)$), we can define a sequence of matrices of increasing size $\{A_n\}_n$. Most results in this report can be applied to the wider class of so-called generalized locally Toeplitz (GLT) sequences [3], but all numerical experiments only concerns the pure Toeplitz case, in particular (4).

The theory of GLT sequences [3], describes the connection by the symbol $f(\theta)$ and the eigenvalues $\lambda_j(A_n)$; informally, we can say that for an Hermitian sequence of matrices $\{A_n\}_n$, with an associated symbol $f(\theta)$, we can approximate the eigenvalues of a matrix A_n for a fixed n as

$$\lambda_j(A_n) \approx f(\theta_{j,n}), \quad j = 1, \dots, n,$$

where $\theta_{j,n}$ is an equispaced grid on $[0, \pi]$, e.g., $\theta_{j,n} = j\pi/(n+1)$.

1.1 Objective and purpose of the project

The aim of this project is first and foremost to implement the so called Ng-Trench algorithm for eigenvalue and eigenvector calculation described in [4]. Thereafter, we wish to combine it with other methods of approximation, parallelize it, and compare it with existing eigenvalue solvers. We wish to look at both eigenvalues and eigenvectors (which both are given by the Ng-Trench algorithm). The goal is to, for large Toeplitz matrices (size at least $n = 10^3$), gain some edge over the standard solvers. Eigenvalue solvers that are found in LAPACK and MKL have been optimized for around half a century, making beating them far from trivial. Our hope is that we, by using the special properties of Toeplitz matrices, can construct an eigenvalue and eigenvector solver "better" than these. It would of course be optimal to both be accurate and fast, however, it is possible that we have to sacrifice accuracy for time efficency. Being able to efficiently approximate eigenvectors and eigenvalues is of course also important.

1.2 Outcome of the project

Although we had a clear objective and problem formulation, the focus of the project shifted as it progressed. After successful implementation, analysis of results from numerical experiments, combined with other methods, the project took a path where eigenvector calculation and approximation became the focus. We found success both time and precision wise when combining our algorithm with standard eigenvalue solvers to find exact eigenvectors, as well as when combining the Ng-Trench algorithm with an eigenvalue approximation method in order to quickly approximate eigenvectors.

2 Theory

The term matrix was introduced by the mathematician James Sylvester in the mid 1800s and his colleague Arthur Cayley started applying matrices to the study of systems of linear equations [5]. Today, matrices have many applications and there exists many different classes of matrices. A way to write systems of linear equations on matrix form is

$$Ax = b, \tag{6}$$

where A is called the coefficient matrix and b is called the constant matrix. (6) can also be used to solve linear partial differential equations. For (6) to be valid, A must have the same number of rows as the solution vector x have columns. This means that $A \in \mathbb{C}^{m\times n}$, $x \in \mathbb{C}^{n\times 1}$ and $b \in \mathbb{C}^{m\times 1}$. The matrix A can also be called a linear transformation on the vector x. If x is a nonzero vector, there exists some scalar λ [8] for which, this equation is true:

$$Ax = \lambda x, \tag{7}$$

where λ is the eigenvalue of A with corresponding eigenvector x. This can be rewritten as:

$$(A - \lambda \mathbb{I})x = 0. \tag{8}$$

This method for calculating eigenvalues and eigenvectors is easy to use when solving very small matrices by hand. When the matrices become larger, the method is not as effective and becomes expensive if a computer were to solve it. This would be a waste for Toeplitz-like matrices because due to their characteristic structure, there exists more efficient methods to compute eigenvalues and eigenvectors for Toeplitz-like matrices.

2.1 Introduction to the Ng-Trench algorithm

In the introduction, we mention that the goal of the project is to implement the Ng-Trench algorithm [4]. This algorithm gives a numerical solution of the eigenvalue problem for Toeplitz-like matrices, and also the corresponding eigenvectors. In this section, we describe and and explain the different parts of the Ng-Trench algorithm.

Here, a Toeplitz-like matrix is denoted as A_n where index n refers to the size of the matrix, which is $n \times n$. The matrix A_n is assumed to be Hermitian, which means that A_n is the same as the transpose of the complex conjugate of A_n,

$$A_n = \overline{A_n^T}.$$

Furthermore, in this report A_n is always real-valued. In order to use Toeplitz-like matrices in the Ng-Trench algorithm, the matrix A_n must satisfy the following equation,

$$A_n Z_n - Z_n A_n = G_n H_n^T \tag{9}$$

where Z_n is a shift matrix,

$$Z_n = \begin{bmatrix} 0 & 0 & ... & 0 & 0 \\ 1 & 0 & ... & 0 & 0 \\ 0 & 1 & ... & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & ... & 1 & 0 \end{bmatrix} \tag{10}$$

and G_n and H_n have the size $n \times \alpha$, where α is assumed to be $\alpha \ll n$ and the displacement rank of A_n, which can be calculated using singular value decomposition. Matrix A_n can be decomposed into the weighted sum of separable matrices, which means that A_n can be written as the outer product of two vectors $A_n = u \otimes v$. A_n can than be decomposed into a sum of columns $\sum_i A_n^{(i)} = \sum_i \sigma_i U_i \otimes V_i$, where σ_i are the ordered singular values, and α is the number of non-zero σ [9]. In the case that A_n is a pure Toeplitz matrix we have $\alpha = 2$ [4] and

$$G_n^T = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & \hat{f}_{n-1} & \cdots & \hat{f}_2 & \hat{f}_1 \end{bmatrix},$$
$$H_n^T = \begin{bmatrix} \hat{f}_1 & \hat{f}_2 & \cdots & \hat{f}_{n-1} & 0 \\ 0 & 0 & \cdots & 0 & -1 \end{bmatrix}.$$

Computing the eigenvalues for an $n \times n$ Hermitian, non-sparse matrix usually requires $\mathcal{O}(n^3)$ floating-point operations per seconds, or flops, while the following algorithm theoretically only would require $\mathcal{O}(n^2)$ flops per eigenvalue [6]. Since the matrix A_n is an $n \times n$-matrix, there are n different eigenvalues $\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$. To be able to write the Ng-Trench algorithm in a concise fashion, the different variables are explained and shown under this statement. A_m is now denoted as

$$A_m = [a_{i,j}]_{i,j=1}^m, \quad 1 \leq m \leq n.$$

With $1 \leq m \leq n$, the matrix can be scaled down into smaller fragments, for example when $m = 1$, A_m only consists of one value, $A_1 = a_{1,1}$. This will prove to be useful in later calculations since there will be iterations over the whole interval 1 to n.

The most common way to calculate eigenvalues algebraically for small matrices is with the following equation: $\det(A_m - \lambda \mathbb{I}_m) = 0$, if this equation is satisfied the eigenvalues are found. Instead of solving this equation for every iteration, this equation is denoted as $p_m(\lambda)$ since it's a very expensive way to calculate eigenvalues for bigger matrices,

$$p_m(\lambda) = \det(A_m - \lambda \mathbb{I}_m), \quad 1 \leq m \leq n$$

where $p_0(\lambda) = 1$. This notation is important for the next step, since the p from the former iteration is used in the following equation,

$$q_m(\lambda) = \frac{p_m(\lambda)}{p_{m-1}(\lambda)}, \quad 1 \leq m \leq n. \tag{11}$$

Since $p_0(\lambda) = 1$, this can be done for every iteration, which is helpful for the cost of the Ng-Trench algorithm because calculations from previous iterations are used, instead of making new calculations. The next step is to define the column just outside matrix A_m,

$$v_m = \begin{bmatrix} a_{1,m+1} \\ a_{2,m+1} \\ \vdots \\ a_{m,m+1} \end{bmatrix}, \quad 1 \leq m \leq n-1$$

v_m is the solution of the following equation, where $w_m(\lambda)$ is the solution,

$$(A_m - \lambda \mathbb{I}_m) w_m(\lambda) = v_m, \quad 1 \leq m \leq n, \tag{12}$$

where

$$w_m(\lambda) = \begin{bmatrix} w_{1m}(\lambda) \\ w_{2m}(\lambda) \\ \vdots \\ w_{mm}(\lambda) \end{bmatrix}, \quad 1 \leq m \leq n-1.$$

Here, $w_m(\lambda)$ is replaced with another column vector. This column vector is w from the previous iteration. The column vector is defined as y_m,

$$y_m(\lambda) = \begin{bmatrix} w_{1m-1}(\lambda) \\ w_{2m-1}(\lambda) \\ \vdots \\ w_{m-1m-1}(\lambda) \\ -1 \end{bmatrix}, \quad 2 \leq m \leq n-1. \tag{13}$$

Since w_{m-1} is one row shorter than w_m, the last row is valued as -1, which is important for the last calculation. Without -1, this method would not work. Here, v_m is also replaced with q_m and $e_m = \begin{bmatrix} 0 & 0 & ... & 0 & 1 \end{bmatrix}^T$ which gives the equation:

$$(A_m - \lambda \mathbb{I}_m) y_m(\lambda) = q_m(\lambda) e_m, \quad 2 \leq m \leq n.$$

When writing this equation on paper, this becomes a system of equations. If w_m is used instead of y_m, this system of equations is not possible to solve, since there would be $m+1$ (the whole column vector w_m plus λ) variables and m equations, but because y_m is used now, there are m variables since the last row in the eigenvector y_m is a constant. The following equation is a way to rewrite q_m with some of the matrices that's been introduced above

$$q_m(\lambda) = a_{mm} - \lambda - v^*_{m-1} w_{m-1}(\lambda), \quad 1 \leq m \leq n. \tag{14}$$

Even if this only is the introduction to the Ng-Trench algorithm it is very difficult for the reader to understand and grasp what's happening. As a way for the reader to get a deeper understanding of what is happening, the different formulas above is used for an $n \times n$ test matrix A_n,

$$A_n = \begin{bmatrix} 6 & -4 & 1 & & & & \\ -4 & 6 & -4 & 1 & & & \\ 1 & -4 & 6 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 6 & -4 & 1 \\ & & & 1 & -4 & 6 & -4 \\ & & & & 1 & -4 & 6 \end{bmatrix} \tag{15}$$

To make it as easy to follow as possible, $m = 4$, which creates the the 4×4 matrix A_m,

$$A_4 = \begin{bmatrix} 6 & -4 & 1 & 0 \\ -4 & 6 & -4 & 1 \\ 1 & -4 & 6 & -4 \\ 0 & 1 & -4 & 6 \end{bmatrix}.$$

A_4 is used to create $p_4(\lambda)$,

$$p_4(\lambda) = \det(A_4 - \lambda I_4) = \lambda^4 - 24\lambda^3 + 166\lambda^2 - 328\lambda + 105.$$

Together with $p_3 = -\lambda^3 + 18\lambda^2 - 75\lambda + 50$ is $p_4(\lambda)$ used to create $q_4(\lambda)$,

$$q_4(\lambda) = \frac{p_4(\lambda)}{p_3(\lambda)} = \frac{\lambda^4 - 24\lambda^3 + 166\lambda^2 - 328\lambda + 105}{-\lambda^3 + 18\lambda^2 - 75\lambda + 50}.$$

Then the column outside the matrix is defined as

$$v_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ -4 \end{bmatrix},$$

and the solution vector $w_4(\lambda)$ is defined as

$$w_4(\lambda) = \begin{bmatrix} w_{14}(\lambda) \\ w_{24}(\lambda) \\ w_{34}(\lambda) \\ w_{44}(\lambda) \end{bmatrix}.$$

As seen in (12), this yields the equation

$$\begin{bmatrix} 6-\lambda & -4 & 1 & 0 \\ -4 & 6-\lambda & -4 & 1 \\ 1 & -4 & 6-\lambda & -4 \\ 0 & 1 & -4 & 6-\lambda \end{bmatrix} \begin{bmatrix} w_{14}(\lambda) \\ w_{24}(\lambda) \\ w_{34}(\lambda) \\ w_{44}(\lambda) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ -4 \end{bmatrix}.$$

Eigenvector $w_4(\lambda)$ is then replaced by $y_4(\lambda) = \begin{bmatrix} w_3(\lambda) \\ -1 \end{bmatrix}$ and v_4 is replaced by $-q_4(\lambda)e_4^T$, which provides the equation

$$\begin{bmatrix} 6-\lambda & -4 & 1 & 0 \\ -4 & 6-\lambda & -4 & 1 \\ 1 & -4 & 6-\lambda & -4 \\ 0 & 1 & -4 & 6-\lambda \end{bmatrix} \begin{bmatrix} w_{13}(\lambda) \\ w_{23}(\lambda) \\ w_{33}(\lambda) \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{\lambda^4 - 24\lambda^3 + 166\lambda^2 - 328\lambda + 105}{-\lambda^3 + 18\lambda^2 - 75\lambda + 50} \end{bmatrix}.$$

This equation finds the eigenvector $w_3(\lambda)$ from the previous iteration. The last equation is (14), which in this case is

$$\begin{aligned} q_4(\lambda) = 6-\lambda - \begin{bmatrix} 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} 6-\lambda & -4 & 1 \\ -4 & 6-\lambda & -4 \\ 1 & -4 & 6-\lambda \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 1 \\ -4 \end{bmatrix} \\ = 6-\lambda - \frac{-17\lambda^2+172\lambda-195}{(\lambda-5)(\lambda^2-13\lambda+10)} = 6-\lambda-\frac{-17\lambda^2+172\lambda-195}{\lambda^3-18\lambda^2+75\lambda-50} \\ = \frac{\lambda^4-24\lambda^3+166\lambda^2-328\lambda+105}{-\lambda^3+18\lambda^2-75\lambda+50} = \frac{p_4(\lambda)}{p_3(\lambda)} = 0. \end{aligned}$$

This method by Ng-Trench is not practical for general Hermitian matrices, because the cost for calculating (12) is $\mathcal{O}(n^3)$ for each eigenvalue λ. It can be combined with some other methods in order to compute individual eigenvalues and eigenvectors of Toeplitz-like matrices. As mentioned in the introduction of the Ng-Trench algorithm, G_n and H_n have dimensions $n \times \alpha$, where α is much smaller than n. For the Ng-Trench algorithm, these matrices are scaled down by dropping the last rows to change the dimensions to $m \times \alpha$. This is done with the following equations

$$G_m = U_{mn}G_n, \quad H_m = U_{mn}H_n,$$

where U_{mn} is the identity matrix, but with dimension $m \times n$, since the last rows are dropped. The matrix U_{mn} is used on the left hand side of (9) in order to transform the left hand side into the correct dimensions, which is done with the following equation

$$U_{mn}A_nZ_nU_{mn}^T = A_mZ_m + v_me_m^T, \quad U_{mn}Z_nA_nU_{mn}^T = Z_mA_m.$$

These equations lead to the final form of the equation

$$A_mZ_m - Z_mA_m = G_mH_m^T - v_me_m^T, \quad 1 \le m \le n-1. \tag{16}$$

The columns of G_m are denoted as

$$g_j^{(m)} = \begin{bmatrix} g_{1j} \\ g_{2j} \\ \vdots \\ g_{mj} \end{bmatrix},$$

which means that G_m can be rewritted as

$$G_m = \begin{bmatrix} g_1^{(1)} & g_2^{(m)} & \dots & g_\alpha^{(m)} \end{bmatrix}. \tag{17}$$

All these equations and notations lay the foundation to the Ng-Trench algorithm that follows. The Ng-Trench algorithm prodives a way to calculate (12) with the cost $\mathcal{O}(\alpha n^2)$ for every eigenvalue and eigenvector instead of the cost $\mathcal{O}(n^3)$.

2.2 The Ng-Trench algorithm

Here, specific start values are required for q, w and f_j which are computed by the following equations

$$q_1 = a_{11} - \lambda, \quad w_1(\lambda) = \frac{a_{12}}{q_1(\lambda)}, \quad f_j^{(1)}(\lambda) = \frac{g_{1j}}{q_1(\lambda)}, \quad 1 \le j \le \alpha.$$

For $2 \le m \le n$ the equation for computing $q_m(\lambda)$ is the same as (14) that was used in the introduction of the Ng-Trench algorithm. Computing the j:th column for $f^{(m)}(\lambda)$ is done with equation

$$f_j^{(m)}(\lambda) = \begin{bmatrix} f_{j-1}^{(m-1)}(\lambda) \\ 0 \end{bmatrix} - \frac{(g_{mj} - v_{m-1}^* f_{j-1}^{(m-1)}(\lambda))}{q_m(\lambda)} y_m(\lambda), \quad 1 \le j \le \alpha, \tag{18}$$

where y_m is calculated in (13) and $\frac{(g_{mj} - v_{m-1}^* f_{j-1}^{(m-1)}(\lambda))}{q_m(\lambda)}$ is a scalar. In order to calculate the eigenvector $w_m(\lambda)$ in a way that isn't as costly as in (12), (16) is rewritten by adding and subtracting λZ_m on the left hand side and both sides are muliplied by $y_m(\lambda)$, which gives

$$(A_m - \lambda \mathbb{I}_m) Z_m y_m - Z_m (A_m - \lambda \mathbb{I}_m) y_m = G_m H_m^T y_m - v_m e_m^T y_m, \quad 2 \le m \le n. \tag{19}$$

As seen in (13), the last row of the eigenvector is -1, which means that $e_m^T y_m(\lambda) = -1$. Since Z_m is a shift matrix, seen in (10), $Z_m(A_m - \lambda \mathbb{I}_m) y_m(\lambda) = Z_m q_m(\lambda) e_m = 0$. This also yields that

$$Z_m y_m(\lambda) = \begin{bmatrix} 0 \\ w_{m-1}(\lambda) \end{bmatrix}.$$

These results gives the resulting version of (19) which is

$$(A_m - \lambda \mathbb{I}_m) \begin{bmatrix} 0 \\ w_{m-1}(\lambda) \end{bmatrix} = G_m H_m^T y_m(\lambda) + v_m, \quad 2 \le m \le n.$$

The final step to find an equation for $w_m(\lambda)$ is to multiply both sides with $(A_m - \lambda \mathbb{I}_m)^{-1}$ since it was established in (12) that $(A_m - \lambda \mathbb{I}_m)^{-1} v_m = w_m(\lambda)$. At last an equation for $w_m(\lambda)$ is provided

$$w_m(\lambda) = \begin{bmatrix} 0 \\ w_{m-1}(\lambda) \end{bmatrix} - (A_m - \lambda \mathbb{I}_m)^{-1} G_m H_m^T y_m(\lambda).$$

Here, $F_m(\lambda) = (A_m - \lambda \mathbb{I}_m)^{-1} G_m$ and $F_m(\lambda)$ is written as columns

$$F_m(\lambda) = \begin{bmatrix} f_1^{(m)}(\lambda) & f_2^{(m)}(\lambda) & \dots & f_\alpha^{(m)}(\lambda) \end{bmatrix}$$

This means that (17) can be rewritten for the different columns as

$$(A_m - \lambda \mathbb{I}_m) f_j^{(m)}(\lambda) = g_j^{(m)}, \quad 1 \le j \le \alpha$$

The final version of the equation for the solution vector w_m is as follows

$$w_m(\lambda) = \begin{bmatrix} 0 \\ w_{m-1}(\lambda) \end{bmatrix} - \begin{bmatrix} f_1^{(m)}(\lambda) & f_2^{(m)}(\lambda) & \dots & f_\alpha^{(m)}(\lambda) \end{bmatrix} H_m^T y_m(\lambda).$$

With this algorithm, it is possible to find specific eigenvalues. In order to find eigenvalue λ_i, an interval (α, β) is given where λ_i is the only eigenvalue in the interval. This can be shown by introducing $Neg_n(\lambda_i)$, which is the number of eigenvalues before λ_i and the number of negative values in the q-vector, since q is negative if a value that's bigger than it's corresponding eigenvalue is given and positive if the given value is smaller than it's corresponding eigenvalue. This means that $q_i(\alpha) > 0$ and $q_i(\beta) < 0$. This also means that the counter Neg_n will be different for α and β. $Neg_n(\alpha) = i - 1$ and $Neg_n(\beta) = i$.

2.3 Matrix-less method

The introduction mentioned symbols briefly. Symbols are very important when using matrix-less methods to calculate eigenvalues. A matrix-less method is not the same the Ng-Trench algorithm seen before but it is also a viable option for approximating eigenvalues for Toeplitz-like matrices and then compute eigenvectors by inserting them in Ng-Trench. As mentioned previously in the introduction of the Ng-Trench algorithm, it is assumed that the Toeplitz-like matrix is Hermitian. Assumed a real matrix, it is here symmetric and banded with bandwidth p. This means that if the function f originally is written as

$$f(\theta) = \sum_{k=-p}^{p} \hat{f}_k e^{\mathbf{i}k\theta}, \tag{20}$$

it can instead be written as

$$f(\theta) = \hat{f}_0 + \sum_{k=1}^{p} \hat{f}_k (e^{\mathbf{i}k\theta} + e^{-\mathbf{i}k\theta}), \tag{21}$$

since $\hat{f}_k = \hat{f}_{-k}$. The other relation that's useful here, is that $e^{\mathbf{i}k\theta} + e^{-\mathbf{i}k\theta} = 2\cos(k\theta)$, which means that function f can once again be rewritten as

$$f(\theta) = \hat{f}_0 + 2\sum_{k=1}^{p} \hat{f}_k \cos(k\theta)$$

To show an example of what it can look like, the bi-Laplacian matrix in (4) is used again, where $\hat{f}_0 = 6$, $\hat{f}_1 = \hat{f}_{-1} = -4$ and $\hat{f}_2 = \hat{f}_{-2} = 1$, which results in the following symbol for the matrix in (4)

$$f(\theta) = 6 - 8\cos(\theta) + 2\cos(2\theta).$$

In order to use a matrix-less method, the symbol for the Toeplitz matrix is needed and with the simplification from (21), a matrix-less method becomes much more viable. For matrix-less methods, the eigenvalues of a matrix $T_n(f)$ is approximated by sampling the symbol of the matrix and adding it with the errors of the approximations of order $\mathcal{O}(h)$, where h is the grid-size related to n as $h = \frac{1}{n+1}$.

$$\lambda_j(T_n(f)) = f(\theta_{j,n}) + E_{j,n},$$

where j is the index of the grid points $\theta_{j,n}$, $j = 1, ..., n$ and n, as mentioned before, is the size of the matrix. The error $E_{j,n}$ is dependent on the grid-size and can be expanded into different functions $c^{(k)}(\theta_{j,n})$ that's also depend on grid-size

$$E_{j,n} = \sum_{k=1}^{\alpha} c^{(k)}(\theta_{j,n}) h^k + E_{j,n,\alpha},$$

where error $E_{j,n,\alpha}$ is of order $\mathcal{O}(h^{\alpha+1})$. With this new of writing $E_{j,n}$, a new equation for the eigenvalues is given

$$\lambda_j(T_n(f)) = f(\theta_{j,n}) + \sum_{k=1}^{\alpha} c_k(\theta_{j,n})h^k + E_{j,n,\alpha},$$

The approximation of $c_k(\theta)$, using the matrix-less method, is possible by carefully choosing matrix sizes n_k and subsets of eigenvalues j_k,

$$n_k = 2^{k-1}(n_1 - 1) + 1, \quad j_k = 2^{k-1}j_1,$$

where $j_1 = 1, 2, \ldots, n_1$, such that $\theta_{j_1,n_1} = \theta_{j_k,n_k}$ for all k. What this enables, is that Julia's own eigenvalue-solver can be used to calculate the eigenvalues for small matrices in order to compute the values of the different functions $c_k(\theta_{j,n})$. Since they are constant, the eigenvalues are also approximately constant for some grid points. But since the the order of the matrix becomes bigger, there are more eigenvalues to compute. These eigenvalues that are not constants can instead be computed using interpolation. Interpolation is a statistical method that uses known values to estimate unknown values between these known values. Since the eigenvalues are approximately the same for some grid points and the symbol f looks the same, no matter what order the matrix is, the error E is also the same for these grid points. This means that all the functions $c_k(\theta_{j_1,n_1})$, $k = 1, ..., \alpha$. This can be written as a matrix function

$$E = HC, \tag{22}$$

where

$$E = \begin{bmatrix} E_{j_1,n_1} \\ E_{j_2,n_2} \\ \vdots \\ E_{j_\alpha,n_\alpha} \end{bmatrix}, \quad H = \begin{bmatrix} h_1 & h_1^2 & ... & h_1^\alpha \\ h_2 & h_2^2 & ... & h_2^\alpha \\ \vdots & \vdots & \ddots & \vdots \\ h_\alpha & h_\alpha^2 & ... & h_\alpha^\alpha \end{bmatrix}, \quad C = \begin{bmatrix} c_1(\theta_{j_1,n_1}) \\ c_2(\theta_{j_1,n_1}) \\ \vdots \\ c_\alpha(\theta_{j_1,n_1}) \end{bmatrix}.$$

Since $E_{j,n,\alpha}$ is an error of order $\mathcal{O}(h^{\alpha+1})$, $E_{j,n,\alpha}$ is assumed to be negligible for $\alpha = 3$ and is therefore excluded when computing C. In order to compute C, both sides are multiplied with H^{-1}

$$\begin{bmatrix} h_1 & h_1^2 & h_1^3 \\ h_2 & h_2^2 & h_2^3 \\ h_3 & h_3^2 & h_3^3 \end{bmatrix}^{-1} \begin{bmatrix} E_{j_1,n_1} \\ E_{j_2,n_2} \\ E_{j_3,n_3} \end{bmatrix} = \begin{bmatrix} c_1(\theta_{j_1,n_1}) \\ c_2(\theta_{j_1,n_1}) \\ c_3(\theta_{j_1,n_1}) \end{bmatrix}. \tag{23}$$

This equation is solved using

$$C = H \setminus E.$$

With these functions, it is possible to use interpolation and extrapolation to approximate the rest of the eigenvalues for a matrix of arbitrary size $n \times n$.

3 Method

The Ng-Trench algorithm, and all additional tools regarding the spectral analysis, are implemented in the JULIA programming language [2]. We first implement the main algorithm, after which we produce a method for finding α and β via bisection, allowing us to find the eigenvalues as roots to $q_n(\lambda)$. Next, we build a matrix-less method for approximating the eigenvalues and combine this with our algorithm in different ways. We finally parallelize the solver and compare both its accuracy and efficiency to standard solvers. We study both the eigenvalues and the eigenvectors, which are a "free" biproduct of the Ng-Trench algorithm.

3.1 Implementing the Ng-Trench algorithm

The Ng-Trench algorithm is purely iterative, starting with a given initial value given by A_n and λ, and thereafter constructing $\{q_i\}_{i=1}^n$ (for a $n \times n$ matrix A_n), step by step. In order to get "clean" and easily separable code, we write each line of the Ng-Trench algorithm as an independent function, each being called by our actual function for finding $\{q_i\}_{i=1}^n$. For example, we implement the function

$$y_m(\lambda) = \begin{bmatrix} w_{m-1}(\lambda) \\ -1 \end{bmatrix}$$

as the following in JULIA.

```
function y(Ww)
        # Uses Ww (w_{m-1} (a vector) to create y_m (a vector))
        return [Ww;-1]
end
```

That is, we take the w from the previous iteration (w_{m-1}) and append -1 to it. A more advanced snippet is the function creating the F matrix, where each column is composed of the column vectors $f_j^{(m)}(\lambda)$, where $1 \leq j \leq \alpha$. Specifically, we create the $m \times \alpha$ matrix

$$\begin{bmatrix} f_1^{(m)}(\lambda)\ f_2^{(m)}(\lambda)\ \dots\ f_j^{(m)}(\lambda) \end{bmatrix}$$

This is done, since although the the Ng-Trench algorithm gives a way of computing α $m \times 1$ vectors in (18), they are always used as a $m \times \alpha$ matrix. The following code is the JULIA implementation:

```
function F(F_old,G,vv,qq,yy)
        # Uses F_old ((m-1) x alpha matrix), G ((m-1) x alpha matrix
        # to extract g_{mj} as
        # bottom value at column j),
        #vv (vector v_{m-1}), qq (value q_m)
        # and yy (vector y_m)
        T = eltype(G)
        m = (size(F_old)[1]+1)
        alpha = size(F_old)[2]
        Fm = zeros(T,m,alpha)
        gg = G[end,1]
        if m-1 == 1
                Fm[:,1] = [F_old[:,1];0] -
                yy*(gg-(vv'*F_old[:,1])[1])/qq
        else
                Fm[:,1] = [F_old[:,1];0] -
                yy*(gg-vv'*F_old[:,1])/qq
        end

        for j in range(2,alpha)
                gg = G[end,j]

                if m-1 == 1
                        Fm[:,j] = [F_old[:,j-1];0] -
                        yy*(gg - ((vv'*F_old[:,j-1])[1]))/qq
                else
                        Fm[:,j] = [F_old[:,j-1];0] -
                        yy*(gg - (vv'*F_old[:,j-1]))/qq
                end
        end
        return Fm
end
```

The principle is the same as for the short `y(Ww)`-function. We take the necessary values from the current or previous iteration as input parameters, and perform operation on and with them as specified by the Ng-Trench algorithm. The matrix-element-syntax is basically identical to MATLAB's, for an $m \times m$ matrix $A_m = \{a_{i,j}\}_{i,j=1}^{m}$ we find element $a_{r,p}$ as `A[r,p]`. Also, just like MATLAB, we get column r as `A[:,r]` and row p as `A[p,:]`.

Only the functions are not enough to calculate the q-vector though, we also need something driving the whole iterative procedure. The basic principle is that, given the function being called for some $n \times n$ matrix A and value λ (not necessarily an eigenvalue), start by calculating the given initial values, and using a `for` loop ranging from 2 to n, we calculate each value one by one. We have to consider some special cases, like the fact that we cannot create v_m when $m = n$ since that would require an $n+1$ element long vector from an $n \times n$ matrix. This is however easily solved by an `if`-statement.

However, implementing the bulk of the Ng-Trench algorithm is just the beginning. As we have stated, eigenvalues are found as roots to $q_n(\lambda)$, and finding these roots is not trivial.

3.1.1 Finding the eigenvalues as roots

As was stated in Section 2, in order to find the i-th eigenvalue, we must first find an interval (α, β) containing exactly one eigenvalue to A_n, and none from A_{n-1}. The first condition is ensured by checking that $Neg_n(\alpha) = i - 1$ and that $Neg_n(\beta) = i$, if there is one less eigenvalue at α compared with β, there must be exactly one between α and β. Remember from section 2 that $Neg_n(\lambda)$ could be found as the amount of negative values in the $\{q_m(\lambda)\}_{m=1}^{n}$. Furthermore, the second condition is that $q_n(\alpha) > 0$ and $q_n(\beta) < 0$. All four of these conditions can be checked with two calls to the q-finding function, one for α and one for β.

The interval finding algorithm requires a starting guess for (α, β), which in principle only requires that the eigenvalue $\lambda \in (\alpha, \beta)$. Using bisection we can use this initial interval and narrow it down until it satisfies all the conditions we impose on it. The idea can be seen in Algorithm 1.

Algorithm 1 Finding the interval

```
Require:
    $\alpha, \beta$                                        ▷ Starting guess
    $i$                                         ▷ Which eigenvalue we want to find
    $A$                                                    ▷ Our matrix
1: while $(\alpha, \beta)$ does not satisfy the conditions do
2:     $\gamma \leftarrow (\alpha + \beta)/2$
3:     if $Neg_n(\gamma) \leq i - 1$ then
4:         $\alpha \leftarrow \gamma$
5:     else
6:         $\beta \leftarrow \gamma$
7:     end if
8: end while
9: return $(\alpha, \beta)$
```

Via the symbol, one can compute maximal and minimal eigenvalues to the matrix[1]. For a 100×100 matrix this might be a fine initial interval, since the eigenvalues are relatively widely spaced, and the matrix operations are not too costly. However, with an increase in matrix size we quickly increase both how compactly the eigenvalues are distributed (of course, if we squeeze more values into $(0, 16)$, the space between two consecutive values decreases), and also the cost of each operation since the matrices are larger. If we use just the minimal and maximal eigenvalues as α and β, respectively, the computational cost of finding the interval that actually satisfy the conditions will rise rapidly. We would thus like a more efficient manner of initializing these values. Such a method has been implemented, and will be discussed after the section on root finding and matrix-less methods. Below follow quick JULIA-like pseudo-code for finding α and β for the i-th eigenvalue to the matrix A,

```julia
function abFinder(a,b,i,A)
    q_a = qFind(a)
    q_b = qFind(b)
    neg_a = count(x->x<0,q_a) # Amount of negative
    # numbers per def.
    neg_b = count(x->x<0,q_b) # As above
    while # Some maximum of iterations
        qn_a = q_a[end] #Values at m=n
        qn_b = q_b[end]
        if Neg_a == i-1 && Neg_b == i && qn_a > 0 && qn_b < 0
            return [a,b] # interval found
        else
            c = (a+b)/convert(T,2)
```

[1]For example, the bi-Laplacian matrix with symbol $f(\theta) = 6 - 8\cos(\theta) + 2\cos(2\theta)$ has a maximum value $f(\pi) = 16$ and a minimal value $f(0) = 0$, and thus all eigenvalues are distributed in the open interval $(0, 16)$ [3]

```
            Neg_c = count(x->x<-0,qFind(c))
            if Neg_c <= i-1
                a = c
            elseif Neg_c >= i
                b = c
            else
                println("error")
end
```

3.1.2 Finding the eigenvalues

We now have a method for computing the q-vectors, and a separate one for finding the intervals containing our sought values. As we know from the Theory section, the eigenvalues are found as the roots to $q_n(\lambda)$, i.e the eigenvalues are the values λ such that $q_n(\lambda) = 0$. There are many optimized root finding algorithms implemented in most computing science languages, and JULIA is no exceptions. Using (in our case) the ROOTS.JL package[2], we can find the roots to $q_n(\lambda)$. All we have to do to find a given eigenvalue is plug in the function with the matrix specified (e.g., by using an anonymous function with only λ as and input parameter), and our interval. Optionally, we could also specify which method to use (the default is bisection). The following lines are some pseudocode for finding the i-th eigenvalue to the matrix A,

```
[a,b] = abFinder(a_start,b_start,i,A)       # Find interval
E = find_zero(lmb->qFinder(A,lmb),(a,b))    # qFinder runs the Ng-
                                            # Trench algorithm for
                                            # given lambda (lmb) and A
```

The `lmb->qFinder(A,lmb)` treats `qFinder(A,lmb)` as a function only depending on `lmb` since `A` is already specified. Finding all (or a section of all) eigenvalues is not more difficult than putting the code for finding one eigenvalue inside a for loop, looping over i for the sought values.

3.2 Implementing the matrix-less method

There are two (major) issues with the Ng-Trench algorithm so far regarding its computational speed. It has to do with both the finding of (α, β), and the root finding procedure. Root finding requires many function calls in order to converge to a root (each demanding for large matrices), and (α, β) currently starts by covering the entire spectrum for each eigenvalue, ensuring no one can be left out. In other words, the interval is unnecessarily big, and it would be advantageous if we could make it smaller and specific to each eigenvalue. As it turns out, we can. By implementing matrix-less methods to approximate the eigenvalues via higher order symbols we can take this eigenvalue approximation, say $\tilde{\lambda}$, and let our interval $(\alpha, \beta) = (\tilde{\lambda} - \epsilon, \tilde{\lambda} + \epsilon)$. We can by using this approximation shift the initial guess of the interval from covering the entire spectrum, to e.g. only covering an interval in the order of 10^{-4}.

Regarding the root finding procedure, matrix-less also has a role to play. As has been stated, one of our method's advantages is that it gives the eigenvector belonging to a certain eigenvalue as a bi-product. Finding eigenvectors to Toeplitz-like matrices is many times desired. We could, with a good matrix-less approximation of the eigenvalue, plug that approximation in and get an

[2]From `https://github.com/JuliaMath/Roots.jl`

eigenvector approximation in just one function call per eigenvector. This is also fully parallelizable, and the higher order symbols need only be calculated once for a certain matrix. Interpolation is inexpensive, and we thus have a potentially very fast way for approximating eigenvectors. Now we will briefly look at the implementation of matrix-less methods.

To use matrix-less, we need two things: a method for numerically computing the higher order symbols, and a way to interpolate the values for larger n.

3.2.1 Calculation and interpolation of higher order symbols

As we described in the section on Theory, we can via (23), which is a direct consequence of (22), determine our C-matrix, where the j-th column are the α higher order symbols evaluated in the j-th point. Implementing this in JULIA is a as easy as writing `C = H\E`.

Algorithm 2 Approximate higher order symbols

```
Require:
    $f(\theta)$                                   ▷ Matrix's symbol
    $\alpha$                                      ▷ No. of symbols to compute
    $A$                                           ▷ Our initial matrix
 1: $n_1 \leftarrow \text{rows}(A)$
 2: $j_1 \leftarrow 1 : n_1$
 3: $\theta_{j_1} \leftarrow j_1 \cdot \pi/(n_1 + 1)$          ▷ Here we define our grid points
 4: for $k = 1 : \alpha$ do
 5:     $n_k \leftarrow 2^{k-1}(n_1 + 1) - 1$
 6:     $j_k \leftarrow 2^{k-1} j_1$
 7:     $h_k \leftarrow 1/(n_k + 1)$
 8:     $E_{j_k,n_k} \leftarrow \lambda_{j_k} - f(\theta_{j_1})$    ▷ Subtract exact eigenvalues by approximated
 9:     Store $E_{j_k,n_k}$ as $k$-th row in $E$
10: end for
11: for $i = 1 : \alpha,\ j = 1 : \alpha$ do
12:     Store $(h_i)^j$ in $H_{i,j}$                  ▷ $h_i$ to the power of $j$
13: end for
14: return $C = H \setminus E$
```

Having calculated our matrix C, we can interpolate the values for each symbol (i.e. each row), by fitting a polynomial to it, and - if we want to approximate eigenvalues to an $n_f \times n_f$-matrix - evaluating it in n_f places along our eigenvalue distribution[3].

3.3 Combining the Ng-Trench algorithm with the matrix-less method

Up until now we have in this section presented ways of implementing our eigenvalue (and eigenvector) solver, using it to find intervals containing exactly **one** eigenvalue, finding eigenvalues as roots to a function, and constructing matrix-less approximations of the eigenvalues. Simply stated, we desire as few function calls as possible in order to utilize our algorithm. With the unmodified algorithm, both the interval finding and the root finding require many function calls. Both root finding and interval

[3]For bi-Laplace this would be in $(0, \pi)$

finding are iterative procedures. We would thus like a somewhat "direct" method of computing these instead. Such can be implemented - although both have their respective trade-offs. We will expand on two ideas below, one being to use a matrix-less approximation of the eigenvalue to construct a narrow interval for (α, β). The other is to directly input the eigenvalue approximations gotten from matrix-less, or other eigenvalue solvers, in order to get an eigenvector approximation. Here we are also interested in how the eigenvectors error depends on the error of the eigenvalue approximation we passed the Ng-Trench algorithm.

3.3.1 Constructing intervals with the matrix-less method

We have already discussed constructing eigenvalue approximations with matrix-less. It is tempting to just input these directly - as an initial guess - into root finding procedure. This does not work. It is to easy ending up in a singular point where we have an eigenvalue to A_{n-1}, made evident by (11). We thus need to approximate an interval, as narrow as possible, containing the eigenvalue we are looking for. This is simply done by, given a set of eigenvalue approximation $\{\tilde{\lambda}_i\}_{i=1}^n$, letting our interval (α, β) for the eigenvalue λ_i be $(\tilde{\lambda}_{i-1}, \tilde{\lambda}_{i+1})$. This is in contrast to letting the interval be the minimal and maximal eigenvalues, given by the symbol and discussed briefly in section 3.1.1.

3.3.2 Approximating eigenvalues with the matrix-less method

This is not more complicated by computing the set of eigenvalue approximations $\{\tilde{\lambda}_i\}_{i=1}^n$ and taking an eigenvector approximation $\tilde{v}_i$ as

$$\tilde{v}_i = y_n(\tilde{\lambda}_i),$$

in accordance with $y_n(\lambda)$ being the eigenvector and a "free" bi-product of the Ng-Trench algorithm.

3.4 Evaluating the methods

Since our algorithm is fully parallelizable (the solving for each eigenvalue and eigenvector can be done independently), it is relevant to compare it running parallel to standard eigenvalue and eigenvector solvers running sequentially. Basically, calling `eigvals(A)` in Julia on a matrix `A`, calculates all eigenvalues, and not in parallel. With our method, we can split the n eigenvalue calculations onto k threads and theoretically reduce the time by around a factor $1/k$. On basic laptop with two cores, we could reduce the time by a factor of four[4]. Using the **UPPMAX** clusters we can access a great amount of CPUs. As of the writing of this report, the Rackham cluster (which we used) consists of 9720 cores [7].
Parallelizing the Ng-Trench algorithm in Julia is trivial. We simply start Julia on as many threads as we want to use, and then put `@threads` before the loop we want to run parallel. A pseudo code example running a loop on 40 threads:

```
# Start Julia with:
julia --threads=40
# Then:
using Base.Threads
@threads for i=1:n
    function(i)
end
```

[4]Each core consists of two threads

We also use the BenchmarkTools.jl package in Julia to analyze the computational demands of our methods, both regarding time and memory. We save both our answers (to look at the error), and the benchmark, for each test we run. The error can be analyzed by computing very high precision solutions, like **Double64**[5] and **BigFloat** with 1024 bits. This is of course very demanding - especially when computing eigenvectors (takes multiple days for matrices of size in the order of 10^3) - but necessary to evaluate the results.

[5]From https://github.com/JuliaMath/DoubleFloats.jl

4 Results and discussion

Below we present some results from our methods, tested upon the square bi-Laplace discretization matrix. Both errors and benchmarks are shown for various data types (standard `Float64`, `Double64`, `BigFloat256`, `BigFloat1024`). As has been stated before, an advantage of our algorithm is that it - in contrast to standard eigenvalue and eigenvector solvers - is fully parallelizable, meaning that we can calculate multiple eigenvalues at the same time (since determining one is not dependent upon another). Thus, we utilize the **UPPMAX** clusters when gauging the viability of our methods. The amount of cores varies depending on the experiment, but in general 20-40 cores are used. As a reminder, for a fully parallelizable application like this, using k cores could result in a time decrease of a factor $1/k$.

We can right away say that there were issues when attempting to use the eigenvalue solver to just find eigenvalues. We can find exact eigenvalues, but when attempting to find all eigenvalues (with datatype `BigFloat`), to a 1024×1024 matrix, we ran out of time, despite having allocated 10 hours on 40 cores, and thus being able to find 80 eigenvalues parallel. Built in, standard functions, could do it however. It should be said that our algorithm has the ability to calculate a single specific eigenvalue, whilst general solvers return all eigenvalues. In the case where the user is only interested in one, or a few, eigenvalues, our method might still be useful. We have also not looked at even larger matrices. That would be very demanding, and experiments could take weeks or even months. These problems mainly arise due to the root finding procedure, requiring many iterations and thus many function calls in order to converge to an eigenvalue. This does however, not make the Ng-Trench algorithm useless. Apart from being able to calculate one or a few specific eigenvalues, we found great potential in fast eigenvector approximation, since this can be done without having to find roots.

As seen in Figure 1, the estimated values on the eigenvectors for the 1024×1024 matrix are almost identical to the theoretical eigenvectors given by JULIA's own function `eigvecs(A)`, except for the first few eigenvectors. Figure 1 shows how the estimated eigenvectors is easier to calculate than the theoretical eigenvectors. The estimated eigenvectors are calculated by using the matrix-less method from 2 with $n_1 = 64$ and $n = 1024$, which in this case is 3.25 times faster than JULIA's own eigenvector-solver.

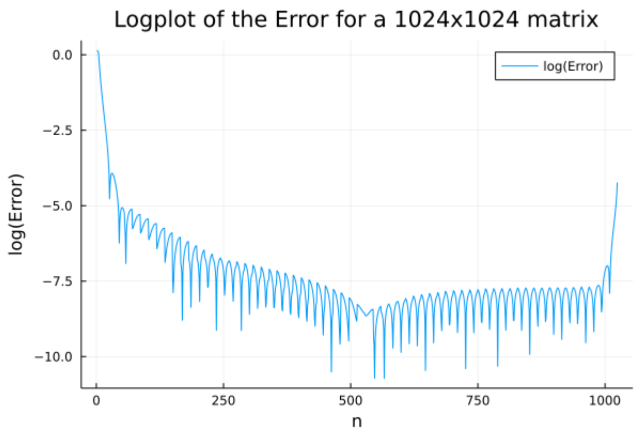
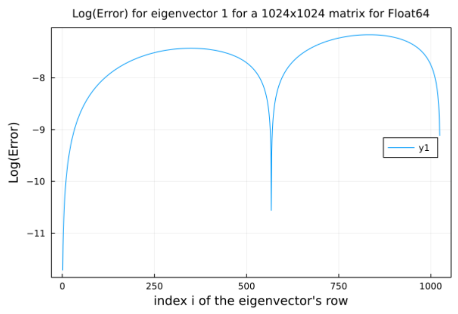


Figure 1: The error for each of our estimated eigenvectors for a 1024×1024 matrix. This graphical description of the error is a logplot.

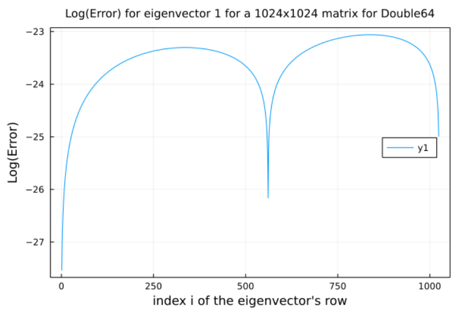
	Time (s)	Memory (GB)
Our solver	1661.194	2184.97
Standard solver	5395.361	2940.89

Table 1: The time it took for the calculations to run and how much memory (estimate of total accessed, not peak memory) that was necessary for our estimated eigenvectors and JULIA's own function `eigvecs(A)`.

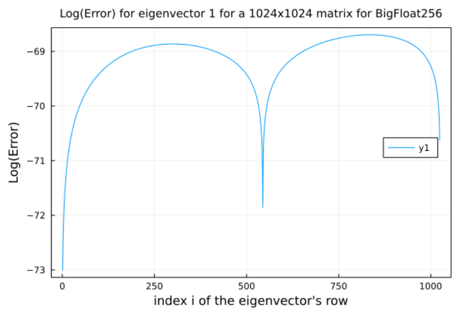
As seen in Figure 1, the error for first eigenvectors are much bigger than the rest of the eigenvectors. This can partly be explained by [1] (due to the fact that the first eigenvalues behave erratic) and the matrix-less method works better for both larger n_1 and larger n. To verify our code, we insert the eigenvalues calculated with JULIA's `eigvals(A)` into `qFinder` in our code to get the eigenvectors. These eigenvectors are then compared to the theoretical eigenvectors, calculated with JULIA's `eigvecs(A)`.



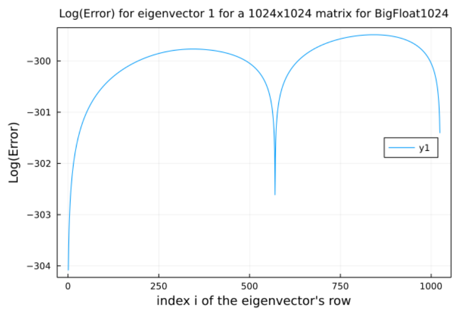
(a) The logarithmic error of all the values in eigenvector 1 for a 1024×1024 matrix with datatype `Float64`.



(b) The logarithmic error of all the values in eigenvector 1 for a 1024×1024 matrix with datatype `Double64`.



(c) The logarithmic error of all the values in eigenvector 1 for a 1024×1024 matrix with datatype `BigFloat256`.

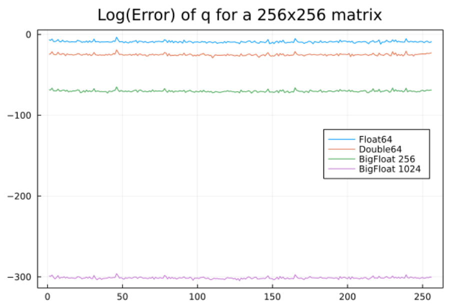


(d) The logarithmic error of all the values in eigenvector 1 for a 1024×1024 matrix with datatype `BigFloat1024`.

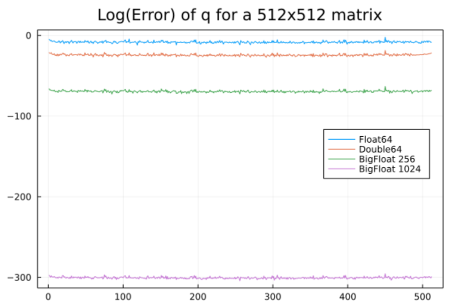
Figure 2: Errors in approximation of the first eigenvector.

Since the first eigenvector was the most accurate eigenvector in Figure 1, we focus on this eigenvector to see if the `qFinder` function can generate better results for theoretical eigenvalues. As seen in Figure 2, the accuracy depend heavily on what datatype that is used. The error is small for all data types but the difference in accuracy is large, especially between `Float64` and `BigFloat1024`. Another thing that's evident is the fact that all graphs are almost identical, which isn't strange since the values are the same, only that the different data types yields different accuracy.

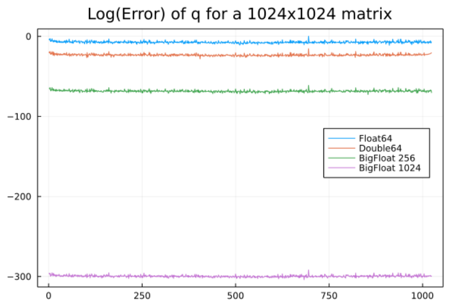
In the Ng-Trench algorithm, there is a way to calculate the eigenvalues using $q_m(\lambda) = 0$, where the eigenvalues are the roots of $q_m(\lambda)$. In the following results, we have done the other way around to test how accurate this method actually is. This is done by using JULIA's own function `eigvals(A)`, which gives the exact eigenvalues, and than using these eigenvalues to see if $q_m(\lambda)$ is equal to zero.



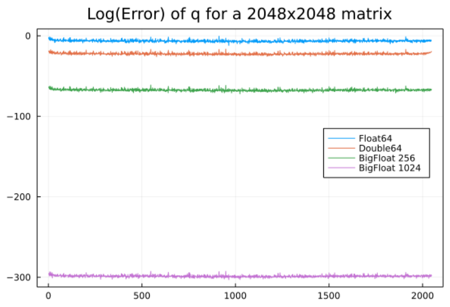
(a) The logarithmic error of q for different data types for a 256×256 matrix.



(b) The logarithmic error of q for different data types for a 512×512 matrix.



(c) The logarithmic error of q for different data types for a 1024×1024 matrix.

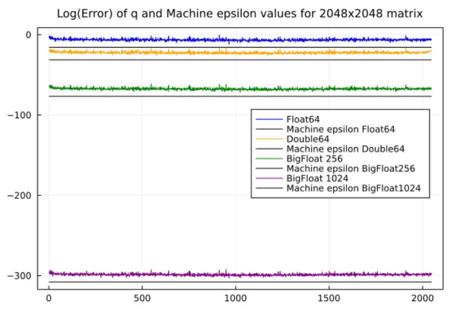


(d) The logarithmic error of q for different data types for a 2048×2048 matrix.

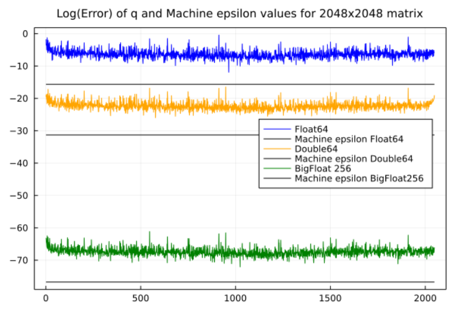
Figure 3: Errors for `q`.

We can also observe that the errors of q for the different data types are almost identical, no matter the size of the matrix. This is a consequence of JULIA's machine epsilon for the data types. We study the 2048×2048 matrix and how the machine epsilons relates to the errors. For reference we mention the different machine epsilons, and memory requirement for storing one value, for different data types.

	`Float64`	`Double64`	`BigFloat256`	`BigFloat1024`
ϵ_m	$2.22 \cdot 10^{-16}$	$4.93 \cdot 10^{-32}$	$1.73 \cdot 10^{-72}$	$1.11 \cdot 10^{-308}$
Bytes	8	16	80	176



(a) The logarithmic error of `q` from the Ng-Trench algorithm for different data types and the machine epsilon values for a 2048×2048 matrix.



(b) A zoomed-in version of 4(a)

Figure 4: Errors for q for different data types.

As seen in Figure 4, the error is always bigger than the machine epsilon for each datatype. What's interesting is that the difference between each datatype and their corresponding machine epsilon is constant. This means that we get correlation between for all the data types and their corresponding machine epsilon. The error can here be written as $\mathcal{O}(10^6 \cdot \epsilon_m)$, where ϵ_m is the machine epsilon for the different data types.

Most of the previous plots have shown how big the errors are for our methods. These results look very promising, but in order for our methods to be justified, our methods should be faster than JULIA's own functions. Figure 5 show us how fast the Ng-Trench algorithm calculate the eigenvectors of a 2048×2048 matrix compared to JULIA's own function `eigen`.

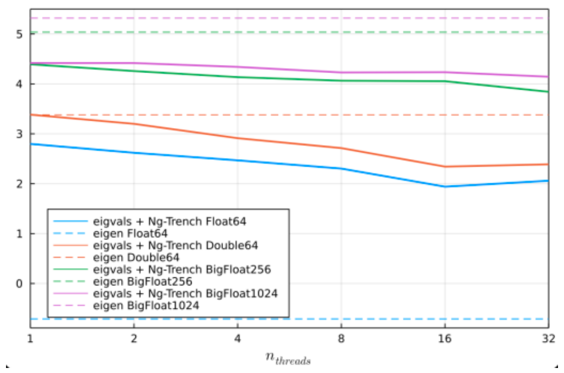
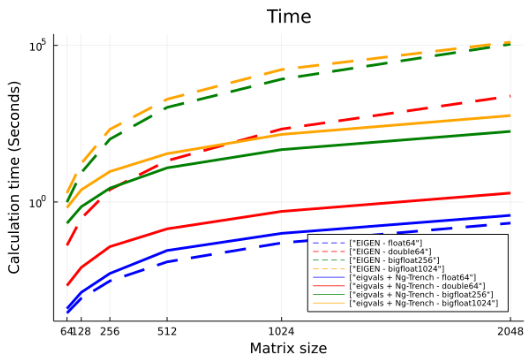
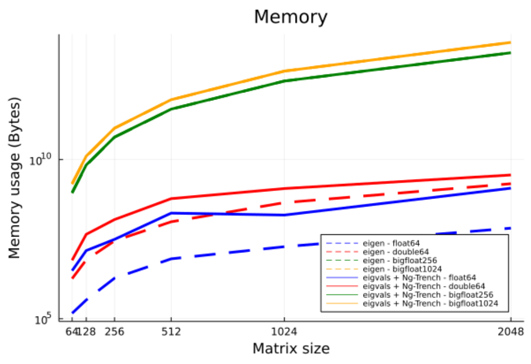


Figure 5: The time it takes for the Ng-Trench algorithm to calculate the eigenvectors for a 2048×2048 compared to `eigen` for different data types. This graphical description of the error is a logplot.

What we see here is a comparison in time between the Ng-Trench algorithm and `eigen`. We observe that for datatype `Float64`, our method is much slower than `eigen` which is very fast in this case, as seen in Figure 5 where the y-axis is logarithmic. However, for all other data types, our method is faster. The largest difference is for `BigFloat1024` where our method is faster by a factor of 10 for only one thread, and then increases when we use multiple threads. The fact that our method can be parallelized is a big advantage, since bigger matrices will take longer, which means that running the code in parallel is of big interest in order to speed up the calculations.



(a) Calculation depending on matrix size for different methods and data types.



(b) Memory usage depending on matrix size for different methods and data types.

Figure 6: Comparisons between `eigen` and `eigvals` combined with Ng-Trench.

In Figure 6 we again compare performance of our eigenvector finder, using eigenvalues from the built-in function `eigvals`, with the built-in function `eigen` that yields both eigenvalues and eigenvectors. Figure 6 panel (a) and (b) show calculation time and memory usage for the two methods, for different matrix sizes and data types. Similar to previous results, the method utilizing the Ng-Trench algorithm outperforms the build-in method in terms of calculation time for all data types except `Float64`. With increasing matrix sizes the difference in calculation time increases up to two orders of magnitude for the `BigFloat` data types. In terms of memory usage the built in methods performs better for both of the lower precision data types, `Float64` and `Double64`. For `BigFloat` data types they perform very similar, results being indistinguishable on the plots.

4.1 Future expansions

Although the implementation of the Ng-Trench algorithm in its current state is far from perfect, we have shown that it has some advantages compared with standard solvers for calculating the eigenvalues, and clear advantages regarding eigenvectors, for Toeplitz matrices. Running the Ng-Trench algorithm in its most basic state has not proved efficient for 1000×1000-matrices, however the size could definitely be increased to see if the standard methods will surpass our algorithm in terms of time required.

An interesting area of expansion is of course to examine how matrix-less methods can improve the Ng-Trench algorithms efficiency, while minimizing the trade off in accuracy. This was done in part, but could of course be looked at much more thoroughly. For example, we know that the error for matrix-less is $O(h^{1+\alpha})$, meaning that if we use larger matrices, we should see a decrease in error[6].

It would also be interesting with an investigation of parallel computing's effect on the algorithm's run time, and why we did not find a linear time decrease when increasing the amount of threads, as seen in Figure 5. Here, more detailed benchmarks and profiling of the algorithm, containing

[6]since $h = 1/(n+1)$ and thus will decrease with increasing n

detailed information regarding e.g. memory allocation, would likely prove useful, both for diagnosing bottlenecks and for optimization.

Throughout this report, we have only experimented with the bi-Laplace matrix. It would be of interest to look at other matrices, ranging from other banded matrices, full[7] matrices, and Toeplitz-like matrices, and investigating if and how this would effect the time and computational demands of the solver, compared with standard methods.

Lastly, one could try to correlate the value of $q_n(\tilde{\lambda})$ for an eigenvalue approximation $\tilde{\lambda}$ with the error of the eigenvalue approximation. For an exact eigenvalue, $q_n(\lambda) = 0$. Seeing how this function behaves for increasing values of error in $\tilde{\lambda}$ could give insights into the Ng-Trench algorithm and how it may be optimized for a given task.

[7]Non-sparse, i.e. most elements are non-zero

5 Conclusions

The implementation of the eigenvalue solver was successful in the sense that it was able to calculate eigenvalues and eigenvectors to our model matrix, the Bi-Laplace discretization matrix. This could be done fully parallel and thus opening up for large reduction in the time demanded.

Regarding the calculation of eigenvalues, we could see that our method was not superior to standard methods for the size we tried (1024×1024 with datatype `BigFloat`) when finding all eigenvalues. We do however note that our solver is able to find a single, or a section, of eigenvalues, which is not something standard solvers do.

We also combined our algorithm with matrix-less methods in order to compute eigenvector approximations using the eigenvalue approximations gotten from matrix-less. This showed very promising results, with most errors around 10^{-6} and the time required being about a third of what a standard solver needed. What we observe from the results is that our calculations of the eigenvectors are much faster than JULIA's built-in function `eigvecs(A)`. The first comparison is Table 1 where we see that when we use the matrix-less method for a 1024×1024, it is 3.25 times faster than `eigvecs(A)`. In Figure 1 the errors for this method are shown, what we see is that the errors are very small for all eigenvectors (and would decrease as $n \to \infty$), except the first few eigenvectors. This can be explained in [1].
When using the Ng-Trench algorithm to calculate the eigenvectors, we see in Figure 5 the difference in time between our solver and `eigen`. As we can see, there is a big variety in our solver's performance and `eigen` for the four different datatypes. For `Float64`, our solver is much slower than `eigen`, but for `Double64`, `BigFloat256` and `BigFloat1024`, our solver is faster. We observe that when only one thread is used, our solver and `eigen` are equally fast but when more threads are used, our solver becomes faster the more threads that are used. This is the case for both `BigFloat256` and `BigFloat1024` as well, the difference is that our solver is initially faster for these datatypes. The biggest difference in speed is for `BigFloat1024` where our solver is faster by a factor of 10 initially and the difference continues to grow using more threads. Similar results are found when looking at the computing time of these methods for different matrix sizes and data types, which can be seen in Figure 6. Memory usage was however less for the built-in methods when using lower precision data types and similar for higher precision data types such as `BigFloat`.

We conclude that results were promising, and further studies are warranted. We especially need to continue investigating matrix-less methods combined with our algorithm, and test matrices of different size, as well as full matrices and non symmetric matrices, not to mention Toeplitz-like matrices. Also code optimization could further improve the algorithms efficiency, as its current state is just a basic implementation

6 References

References

[1] M. Barrera, A. Böttcher, S. M. Grudsky, and E. A. Maximenko, *Eigenvalues of even very nice Toeplitz matrices can be unexpectedly erratic*, in The Diversity and Beauty of Applied Operator Theory, Operator theory, Springer International Publishing, Cham, 2018, pp. 51–77.

[2] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM Rev. Soc. Ind. Appl. Math., 59 (2017), pp. 65–98.

[3] C. Garoni and S. Serra Capizzano, *Generalized locally Toeplitz sequences: Theory and applications*, Springer International Publishing, Cham, Switzerland, 1 ed., June 2017.

[4] M. K. Ng and W. F. Trench, *Numerical solution of the eigenvalue problem for Hermitian Toeplitz-like matrices*, 1997.

[5] The Editors of Encyclopaedia Britannica, Matrix theory. [Online]. Available from: https://www.britannica.com/science/matrix-mathematics. 1998.

[6] W. F. Trench, *Numerical Solution of the Eigenvalue Problem for Efficiently Structured Hermitlan Matrices*, Trinity University, 1991.

[7] UPPMAX, *The Rackham Cluster*. [Online]. Available from: https://uppmax.uu.se/resources/systems/the-rackham-cluster/. 2021.

[8] E. W. Weisstein, *Eigenvalue*. [Online]. Available from: https://mathworld.wolfram.com/Eigenvalue.html.

[9] Wikipedia, *Singular value decomposition*. [Online]. Available from: https://en.wikipedia.org/wiki/Singular_value_decomposition#Example. 2022.

7 Populärvetenskaplig sammanfattning

Vid lösning av alla möjliga tekniska naturvetenskapliga problem uppkommer matriser. Dessa är en sorts samling med värden, som har sina egna räkneregler. De kan ha godtycklig storlek, och ett exempel är enhetsmatrisen:

$$I = \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & ... & 0 & 1 \end{bmatrix}$$

vilken är matrisernas motsvarighet av en etta[8]. Ofta har matriser inom vissa tillmäpningar återkommande strukturer, så att även om värdena skiljer sig mellan två matriser, så är de ändå placerade utefter något visst mönster. En sådan struktur en matris med konstanta diagonaler, som matrisen ovan som bara har ettor på huvuddiagonalen. En sådan matris kallas för en Toeplitz-matris[9] och förekommer bland annat i signalbehandling och numerisk beräkning av derivator. Alla matriser har vissa kvantifierbara egenskaper som ger information om den, och en sådan egenskap är egenvärdet, som är viktigt för att bland annat avgöra metoders lämplighet. Detta värde har vissa geometriska egenskaper som vi inte går igenom nu, det är helt enkelt ett värde vi är intresserade av.

Egenvärdet kan beräknas analytiskt för en matris, men för riktiga tillämpningar är matriserna inte sällan i storleksordning större än 1000×1000, alltså tusen rader och tusen kolumner, och därmed över en miljon element[10]. För att beräkna egenvärden till matriser större än 10×10 blir papper och penna väldigt plågsamt, och behöver vi numeriska metoder som kan göra det effektivt. Vi kan inte heller kopiera den analytiska metoden och göra den numeriskt, både på grund av tidsåtgång och stabilitet, alltså att vi får felaktiga värden. Som tur är, finns många metoder för egenvärdesberäkning som kan hantera "stora" matriser. Vi kallar dessa för *vanliga* egenvärdeslösare, efter som de kan beräkna egenvärden till matriser med godtycklig struktur. Vi studerar som sagt Toeplitz-matriser, och vill göra en *specialiserad* egenvärdeslösare för dessa. Alltså en lösare som bara hanterar Toeplitz-matriser.

Syftet med detta projekt är att implementera en egenvärdeslösare för Toeplitz-matriser i programmeringsspråket JULIA, vilket är ett "snabbt" programmeringsspråk för numerisk analys. Denna lösare kan räkna ut på egenvärden, och egenvektorer (som är en sorts matris med bara en kolonn, och hör till egenvärdet). Huvudprincipen är att den arbetar fram en funktion steg för steg, och sen hittar egenvärdet som roten[11] till den funktionen. Vi vill därefter effektivera denna algoritm. Vi testar även att kombinera den med en snabb metod för egenvärdesuppskattning, alltså att hitta vad egenvärdet ungefär är, som bygger på att utnyttja särskilda funktioner kopplade till Toeplitz-matrisen.

En fördel är att vi kan köra vår lösare parallellt. Traditionellt, om fyra beräkningar ska göras, görs de en efter varandra. Men genom att beräkna parallellt kan alla fyra beräkningar göras samtidigt, vilket drar ner beräkningstiden med ungefär en faktor fyra. Om en beräkning tar 10 sekunder, skulle det alltså traditionellt ta 40 sekunder. Vi kan dock göra fyra beräkningar på ungefär 10 sekunder

[8]För en matris A är $IA = A$, på samma sätt som $1 \cdot a = a$ för en variabel a
[9]Efter matematikern Otto Toeplitz
[10]Värdena i en matris kallas för element
[11]En rot är ett värde där funktionen är noll. Till exempel är $x = 2$ en rot till $f(x) = 2x - 4$ eftersom $f(2) = 0$

oavsett om vi kör dem parallellt[12]. Detta är ett enkelt exempel, och 10 eller 40 sekunder spelar kanske inte så stor roll, men ponera att varje beräkning tar fem minuter, och du ska göra 1000 beräkningar. Då kan du spara ganska mycket tid genom att göra exempelvis 64 beräkningar parallellt åt gången. Såklart finns begränsningar för hur många processer som kan köras parallellt samtidigt, och dessa bestäms av hårdvaran du räknar på. Men genom att utnyttja stora datorsystem bygda för att göra krävande beräkningar kan vi få tillgång till exempelvis 64 trådar, där varje tråd kör en process.

Slutligen kör vi tester på vår algoritm, där målet såklart är att vara lika snabba och få lika bra svar som de vanliga lösarna. Vi ser att vi kan göra bra uppskattningar av egenvektorerna på mycket kortare tid än de vanliga, men att vi då offrar en del av vår precision, det vill säga får ett lite felaktigt svar. Detta eftersom vi använder de tidigare nämnda uppskattningarna på egenvärdena. Vi kan också använda vanliga lösare för att beräkna exakta egenvärden, och sedan använda dessa för att beräkna exakta egenvektorer, snabbare än en vanlig lösare kan beräkna egenvärden och egenvektorer. För framtden är det intressant att testa algoritmen på större matriser, samt på andra typer matriser än den matris vi testat här. Den måste såklart vara Toeplitz, men det finns fler strukturer en matris kan ha.

[12]Detta är en grov förenkling, mängder av olika faktorer - exempelvis minne - kan spela in och göra att tidsåtgången inte minskar på detta sätt