Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

Note

# Finding the Largest Eigenvalues of Large Matrices

Note no No number
Author Geir-Arne Fuglstad
Date 10th July 2009

Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

## The author

Student from NTNU on summer job

## Norwegian Computing Center

Norsk Regnesentral (Norwegian Computing Center, NR) is a private, independent, non-profit foundation established in 1952. NR carries out contract research and development projects in the areas of information and communication technology and applied statistical modeling. The clients are a broad range of industrial, commercial and public service organizations in the national as well as the international market. Our scientific and technical capabilities are further developed in co-operation with The Research Council of Norway and key customers. The results of our projects may take the form of reports, software, prototypes, and short courses. A proof of the confidence and appreciation our clients have for us is given by the fact that most of our new contracts are signed with previous customers.

**Norsk Regnesentral**
Norwegian Computing Center
Postboks 114, Blindern
NO-0314 Oslo, Norway

**Besøksadresse**
*Office address*
Gaustadalléen 23
NO-0373 Oslo, Norway

**Telefon** · telephone
(+47) 22 85 25 00
**Telefaks** · telefax
(+47) 22 69 76 60

**Internett** · internet
www.nr.no
**E-post** · e-mail
nr@nr.no

| | |
|---|---|
| **Title** | **Finding the Largest Eigenvalues of Large Matrices** |
| **Author** | **Geir-Arne Fuglstad** |
| Date | 10th July 2009 |
| Publication number | No number |


## Abstract

The Lanczos iteration for finding the largest eigenvalues has been implemented in C++. This implementation has been tested against ARPACK which is created for this purpose, but written in Fortran. The test were with regards to both speed and precision.



| | |
|---|---|
| Keywords | |
| Target group | |
| Availability | |
| Project | |
| Project number | |
| Research field | |
| Number of pages | 17 |

# Contents

# 1 Introduction

The process of finding all eigenvalues of a matrix becomes very time consuming for large matrices. Therefore iterative methods such as implicitly restarted Arnoldi iterations has been created for finding some part of the spectre of a matrix. Here the focus will be on finding the largest eigenvalues, and only symmetric matrices will be considered. For symmetric matrices there exists a faster variant of the implicitly restarted Arnoldi method called the implicitly restarted Lanczos method. The implementation of this in C++ is described in the following sections.

In addition to the implementation, this report describes the tests done. These tests explore the dependency on run time versus the size of the matrix, the number of eigenvalues calculated, the number of vectors used to calculate the eigenvalues and the density of the eigenvalues. Here density will be determined by what the closest eigenvalues are.

From this point on the abbreviation IRLI will be used for implicitly restarted Lanczos iteration.

# 2 Background on IRLI

Let $A$ be the $n \times n$ matrix for which one wants to compute the eigenvalues. Then the IRLI method for finding the $k$ largest eigenvalues of $A$ is based on the Krylov space of $A$ corresponding to some start vector, say $v_0$. The Krylov space of $A$ of size $m$ corresponding to $v_0$ is

$$K_m(A, v_0) = \text{span}(v_0, Av_0, \ldots, A^{m-1}v_0).$$

If one consider $v_0$ as a linear combination of the eigenvectors of $A$, one would expect the components of the largest eigenvectors to dominate the vector $A^{m-1}v_0$ as $m$ increases. However, taking this product is potentially numerically unstable.

This is solved by using the Lanczos iterations, this algorithm is a stable way of creating an orthonormal basis for $K_m(A, v_0)$. Furthermore the IRLI is a way of creating a iterative method from the Lanczos iterations in which each iteration improves the estimates for the eigenvalues and the eigenvectors of the largest eigenvalues. The implicitly restarted part of the name refers to the fact that after a fixed number of steps, say $m$, the iterations are stopped and the basis is reduced to size $k$ again and thereafter expanded to size $m$ again. This allows for the use of less storage and less orthogonalizations.

# 3 Implementation of IRLI

## 3.1 Step 1: Initial Lanczos iteration

The desire is to create a Krylov space in which information of the largest eigenvalues can be found, therefore if no information exists a random starting vector is as good as can be chosen.

The implementation may be provided with a start vector, if not a random starting vector $v_1$ is created. This starting vector is then normalized to a norm of 1, and will provide the starting point of the iterations. The other input needed is $m$ the number of steps of the Lanczos iterations to do. This will determine the size of the matrix $V$ and $H$ obtained from the process. The process will give a $m \times m$ symmetric tridiagonal matrix $H$ and a $n \times m$ matrix $V$, with orthonormal columns.

Set $v_0 = 0$ and $\beta_0$ = 0, then the algorithm used may be represented as follows. At step $i = 1$ let $\hat{f}_i = Av_i$ and calculate $\alpha_i = \hat{f}_i^T v_i$. Then orthogonalize $\hat{f}_i$ against the two previous vectors $v_i$ and $v_{i-1}$ by using the calculated $\alpha_i$ and $\beta_{i-1}$ by setting

$$f_i = \hat{f}_i - \alpha_i v_i - \beta_{i-1} v_{i-1}.$$

This is theoretically enough to create a vector $f_i$ which is orthogonal to all previous vectors $v_j$ for $j = 1, \ldots, i$ due to the symmtery of $A$, however, few steps are required in finite arithmetic before the loss of orthogonality becomes too big. Therefore an additional step is added here, $f_i$ is orthogonalized against all previous vectors. This is called a full reorthogonalization scheme and is done at each iteration.

The vector is then normalized and used as the next vector, by calculating $\beta_i = ||f_i||$ and setting $v_{i+1} = \frac{f_i}{\beta_i}$. This is done for $i = 1, \ldots, k$, but for the last iteration $i = k$ the vector $v_{k+1}$ is not calculated. If the norm of $f$ becomes smaller than the tolerance during the iteration, the iteration is stopped and the discovered set of vectors are consider to span a invariant subset of $A$ and are expected to give error estimates smaller than the tolerance for all eigenvalues of $h$.

After completion the set $\{v_1, \ldots, v_k\}$ constitute a set of orthonormal vectors and the calculated values may be used to represent $A$ in a near tridiagonal form. Let

$$V = \begin{bmatrix} v_1 & \ldots & v_k \end{bmatrix}$$

and

$$H = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \beta_{k-1} & \alpha_k \end{bmatrix}.$$

Then one can write

$$AV = VH + f_k e_k^T, \tag{1}$$

where $e_k^T = [0 \ \dots \ 0 \ 1]$. From this one can see that if $(\theta, s)$ satisfies

$$Hs = \theta s,$$

then

$$AVs = VHs + f_k e_m^T s = \theta Vs + f_k e_m^T s.$$

Thus if we can make the last term on the right hand side small, $(\theta, Vs)$ will be a good approximation to an eigenpair of $A$. Additionally the columnspace of $V$ is $K_k(A, v_1)$, so we would expect the large eigenvalues of $A$ to dominate the eigenvalues of $H$. One can also see this as $H$ represents the projection of $A$ onto the krylov space. To drive the error on the right hand side towards zero the iteration may be continued until $||f||$ becomes small at the cost of storage space and increasingly more expensive orthogonalizations, or the iterations may be implicitly restarted as explained in the following subsections.

At each step $i$ of the iterations a matrix vector product of complexity $O(n^2)$ and a full reorthogonalization of complexity $O(in)$ are required. This gives a total complexity of $O(kn^2 + k^2 n)$.

## 3.2 Step 2: Calculate eigenvalues and find error bounds

From step 1 one has the from given in equation (1). Let $m$ be the number of iterations that were done in step 1 and $k$ the number of eigenvalues of $A$ that are desired.

The algorithm then computes the eigenvalues of $H$ and sorts them in to the $k$ largest $\sigma_1(H)$ and the $p = m - k$ undesired eigenvalues $\sigma_2(H)$. For each desired eigenvalue the error is estimated as

$$||AVs - \lambda Vs|| = ||f||||e_m^T s|.$$

The eigenvalues $\lambda$ is considered converged when this size is less than tol$\lambda$. If tol is less than machine precision, the machine precision is used as tolerance instead.

Finding the eigenvalues of a symmetric tridiagonal matrix has a complexity of $O(m^2)$, however, as eigenvectors are also computed it has a complexity of $O(m^3)$.

## 3.3 Step 3: Shifting unwanted eigenvalues

Let $m$ be the size of the Lanczos iteration created, $k$ the desired number of eigenvalues and $n$ the size of $A$.

To this step some theory is required. A QR-shift of $H$ with shift $\lambda$ means to set $H = Q^T HQ$ where $Q$ is the $Q$ from the QR factorization of $H - \lambda I$. The implicit

QR-shift with shift $\lambda$ does the same, however, $H - \lambda I$ is never formed. This reduces errors due to subtracting a possibly small number from each diagonal element and allowes an implementation of it with complexity $O(m)$ by using Givens rotations. The implementation of this is based on the "tqli" algorithm in "Numerical Recipes in C". This algorithm also computes $V = VQ$ and $\delta$ required in updating recidual. This increases the complexity of the implict QR-shift to $O(mn)$. The implementation of the implicit QR-shift is based on a $Q_1^T = P_{1,2} \cdots P_{m,m-1}$ as a product of Givens rotation, as such $e_m^T Q_1 = [0 \ \ldots \ 0 \ \delta_1 \ *]$. And the next shift will give $e_m^T Q_1 Q_2 = [0 \ \ldots \ 0 \ \delta_1 \delta_2 \ * \ *]$.

Using this one can reduce the form in equation (1) to

$$AVQ = VQQ^T HQ + f e_m^T Q,$$

where $Q = Q_1 \cdots Q_p$ and the $Q_i$ matrices corresponds to the implicit QR-shifts. This means if one sets $V^+ = VQ$, it will still have orthonormal columns, $H^+ = Q^T HQ$ is still symmetric tridiagonal and $e_m^T Q = [(\delta_1 \cdots \delta_p) e_k^T \ * \ \ldots \ *]$. Thus setting

$$f^+ = f\delta_1 \cdots \delta_p + \beta_k v_{k+1},$$

gives

$$AV^+ = V^+ H^+ + f^+ e_k^T,$$

a Lanczos iteration of $k$ steps with a new starting vector.

With the theory explained the description of the algorithm will continue. From the starting form from step 1 and the eigenvalues from step 2, an implicit QR-shift is done for each of the undesired eigenvalues, where for each shift $V = VQ$ and $\delta_i$ is computed. After the $p$ implicit QR-shifts the new $f$ is calculated and a $k$ step Lanczos iteration is achived.

After this the desired eigenvalues are still eigenvalues of $H^+$.

Each shift has a complexity of $O(mn)$ and a total of $p$ is required. Finally calculating the new $f$ has a complexity of $O(n)$. Thus the total complexity is $O(pmn)$.

## 3.4 Step 4: Expand basis

Let $m$ be the total size of the Lanczos iteration used, $k$ the number of eigenvalues, $p = m - k$ and $n$ the size of $A$.

This step is the same as step 1, the only difference is that the iterations are not startet at $i = 1$. The implementation is equal. After this step go to step 2.

As in step 1 iteration $i$ requires $i$ orthogonalizations with complexity of $O(in)$ and a matrix vector product of complexity $O(n^2)$. Thus iterations $i = k, \ldots, m$ has a complexity of $O(pn^2 + m^2 n)$.

### 3.5 Possible shortcomings

If some of the largest eigenvalues are equal, the implementation will not be able to detect all of them. This is because all components of eigenvectors corresponding to the same eigenvalue will increase by the same factor. But the difference required to detect close eigenvalues can be made small by setting the tolerance to a low value. As an example consider the $1000 \times 1000$ identity matrix. At a tolerance of 1e-15 the implementation is only able to detect one of the eigenvalues, but if one adds random values of order 1e-14 to the diagonal elements it is able to detect multiple eigenvalues. ARPACK accomplishes the detection of multiple eigenvalues of the identity matrix, so if it becomes important to detect extremely close eigenvalues it should be possible to do so. An idea might be to allow the Lanczos iteration to continue when $f$ becomes small, but do more reorthogonalizations to ensure orthogonality, and stop only when it will underflow the double values.

The full orthogonalization at each step of the Lanczos iteration might not always be necessary. Some articles seems to suggest a partial reorthogonalization scheme, in which the degree of orthogonality is monitored by a simple recursion relation, with the logic that semiorthoogonality to preserve the desired properties of the iterations. This will not decrease the total number of matrix vector product required, but might reduce the number of reorthogonalizations.

## 4 Tests of speed and accuracy

### 4.1 Summary

For this section let the size of $A$ be $n$, the number of eigenvalues to calculate $k$ and the number of vectors used $m$. The tolerance will be called tol, $T_C$ will refer to the time used by the implementation in C++ and $T_A$ refer to the time used by ARPACK, both times will be measured in ms.

The test that follow will test the dependence of the time used on the size of the matrix $n$, $k$ when there is some fixed relationship between $k$ and $m$ and the density of the eigenvalues. Here the density will be considered to be how large the ratios between close eigenvalues are. To eigenvalues will be considered close when the ratio between them is close to 1.

Table 1. $T_C$ and $T_A$ as a function of $n$, and the greatest relative error between eigenvalues calculated by ARPACK and C++ implementation $E$ for each $n$.

| $n$ | $T_C$ | $T_A$ | $E$ |
|---|---|---|---|
| 215 | 90 | 70 | 6.7e-15 |
| 278 | 160 | 130 | 1.7e-14 |
| 359 | 290 | 260 | 1.3e-14 |
| 464 | 530 | 470 | 3.7e-14 |
| 599 | 940 | 870 | 4.1e-14 |
| 774 | 1900 | 1790 | 4.6e-14 |
| 1000 | 3750 | 3400 | 5.3e-14 |

## 4.2 Geometric progression with factor 1.0001

### 4.2.1 Dependency on the the size of the initial matrix

In this test the eigenvalues of the matrix $A$ will be $\lambda_i = 1.0001^{10000-i}$. In other words as the size of $A$ icreases more eigenvalues are added from the progression. The following parameters were used for the test

| **Parameter** | **Value** |
|---|---|
| $k$ | 3 |
| $m$ | 29 |
| tol | 1e-15 |

Table 1 shows how $T_C$ and $T_A$ increases as $n$ increases. A linear regression of the logarithm of time and the logarithm of $n$ gives a coefficient of 2.414 for this implementation and 2.524 for ARPACK. Both of these are higher than the value of $2.0$ one would expect if the number of restarts were constant. Since the ratio between succesive eigenvalues are kept constant one would perhaps expect this to be the case, but a possible cause might be the fact that the ratio is quite small.

### 4.2.2 Dependency on the number of eigenvalues

In this test the eigenvalues of the $1000 \times 1000$ matrix $A$ will be $\lambda_i = 1.0001^{10000-i}$. This test will consider what happens when $k$ is increased while the relationship $m = 2k + 23$ is used. This more or less arbitrary chosen for this test. Parameters used for the test

| **Parameter** | **Value** |
|---|---|
| $n$ | 1000 |
| $m$ | $2k + 23$ |
| tol | 1e-15 |

Table 2 shows how $T_C$ and $T_A$ depends on $k$ under the relation $m = 2k + 23$. It is not easy to draw any useful conclusion based on these data, however, one may

Table 2. $T_C$ and $T_A$ as a function of $k$, and the greatest relative error between eigenvalues calculated by ARPACK and C++ implementation $E$ for each $k$.

| $k$ | $T_C$ | $T_A$ | $E$ |
|---|---|---|---|
| 2 | 3930 | 3600 | 5.7e-14 |
| 4 | 3480 | 3320 | 5.0e-14 |
| 8 | 3530 | 3010 | 3.0e-14 |
| 16 | 3520 | 3210 | 7.8e-14 |
| 32 | 4350 | 3620 | 7.0e-15 |

Table 3. $T_C$ and $T_A$ as a function of $m$, and the greatest relative error between eigenvalues calculated by ARPACK and C++ implementation $E$ for each $m$.

| $m$ | $T_C$ | $T_A$ | $E$ |
|---|---|---|---|
| 40 | 3620 | 3420 | 5.1e-14 |
| 80 | 3370 | 3140 | 1.4e-14 |
| 160 | 5360 | 3350 | 2.0e-14 |
| 320 | 5480 | 2510 | 8.2e-16 |
| 640 | 25690 | 6320 | 4.9e-16 |

note that it appears to be faster to find 4 eigenvalues than 2 eigenvalues, this is due to the arbitrary nature of the relation between $k$ and $m$. Better choices of $m$ will give better times, but it is not easy to know what the proper choice is.

### 4.2.3 Dependency on the size of the iteration

In this test the eigenvalues of the $1000 \times 1000$ matrix $A$ will be $\lambda_i = 1.0001^{10000-i}$. This test will consider what happens when $m$ is increased while the $k = 3$ is used. Parameters used for the test

| **Parameter** | **Value** |
|---|---|
| $n$ | 1000 |
| $k$ | 3 |
| tol | 1e-15 |

From table 3 one can see that there are some values of $m$ that are better than others. However for values not far from the optimal value the difference in time is not large, but for very bad choices there may be significant difference. Further one can see that ARPACK has a much better time for large values of $m$. The interesting values here are all except the last one, a value of $m$ that large would be larger than the one required to converge with only one Lanczos iteration. One can see that ARPACK has a time usage that does not appear to depend to much upon $m$ unless it is chosen extremely large, or too small for convergence. This implemenation seems to possess stages at which it keeps a nearly constant value

Table 4. $T_C$ and $T_A$ as a function of $n$, and the greatest relative error between eigenvalues calculated by ARPACK and C++ implementation $E$ for each $n$.

| $n$ | $T_C$ | $T_A$ | $E$ |
|---|---|---|---|
| 215 | 30 | 20 | 4.7e-15 |
| 278 | 50 | 30 | 5.2e-15 |
| 359 | 70 | 60 | 3.7e-15 |
| 464 | 120 | 110 | 4.1e-15 |
| 599 | 170 | 170 | 2.7e-15 |
| 774 | 290 | 300 | 4.5e-15 |
| 1000 | 500 | 490 | 2.4e-15 |

and large increases. This could indicate inefficient access to the stored arrays and that the parts of the algorithm with cubic dependency on $m$ grows too large. Another possible cause could be that the cost of the full reorthogonalization grows very large.

## 4.3 Geometric progression with factor 1.05

### 4.3.1 Dependency on the size of the inititial matrix

In this test the eigenvalues of the matrix $A$ will be $\lambda_i = 1.05^{1000-i}$. In other words as the size of $A$ icreases more eigenvalues are added from the progression. The following parameters were used for the test

| **Parameter** | **Value** |
|---|---|
| $k$ | 3 |
| $m$ | 15 |
| tol | 1e-15 |

Table 4 shows how $T_C$ and $T_A$ depens on $n$ in this example. A linear regression on the logarithm on $n$ and the logarithm of the time gives a coefficient of 1.791 for C++ implementation and a coefficient of 2.126 for ARPACK. These values are around the value 2 which one would expect as the ratio between succesive eigenvalues is kept constant.

### 4.3.2 Dependency on the number of eigenvalues

In this test the eigenvalues of the $1000 \times 1000$ matrix $A$ will be $\lambda_i = 1.05^{1000-i}$. This test will consider what happens when $k$ is increased while the relationship $m = 2k + 9$ is used. Parameters used for the test

| **Parameter** | **Value** |
|---|---|
| $n$ | 1000 |
| $m$ | $2k + 9$ |
| tol | 1e-15 |

Table 5. $T_C$ and $T_A$ as a function of $k$, and the greatest relative error between eigenvalues calculated by ARPACK and C++ implementation $E$ for each $k$.

| $k$ | $T_C$ | $T_A$ | $E$ |
|---|---|---|---|
| 2 | 530 | 530 | 3.6e-15 |
| 4 | 530 | 460 | 3.7e-15 |
| 8 | 510 | 520 | 3.5e-15 |
| 16 | 650 | 610 | 3.0e-15 |
| 32 | 940 | 850 | 1.7e-15 |
| 64 | 1390 | 1380 | 6.6e-15 |
| 128 | 3860 | 2880 | 1.2e-13 |

Table 6. $T_C$ and $T_A$ as a function of $m$, and the greatest relative error between eigenvalues calculated by ARPACK and C++ implementation $E$ for each $m$.

| $m$ | $T_C$ | $T_A$ | $E$ |
|---|---|---|---|
| 20 | 600 | 480 | 6.3e-15 |
| 40 | 570 | 520 | 1.2e-15 |
| 80 | 680 | 560 | 1.1e-15 |
| 160 | 1740 | 1140 | 1.1e-15 |
| 320 | 5470 | 2540 | 2.7e-15 |
| 640 | 27740 | 6650 | 3.0e-15 |

From table 5 one can see the same development as in the the previous section. ARPACK is more efficient than this implementation when the number of vectors to create increase. This is the same behavior experienced in the previous subsection.

### 4.3.3 Dependency on the size of the iteration

In this test the eigenvalues of the $1000 \times 1000$ matrix $A$ will be $\lambda_i = 1.05^{1000-i}$. This test will consider what happens when $m$ is increased while the $k = 3$ is used. Parameters used for the test

| **Parameter** | **Value** |
|---|---|
| $n$ | 1000 |
| $k$ | 3 |
| tol | 1e-15 |

Table 6 shows how the time used depends on the number of vectors used $m$ for $k = 3$ eigenvalues. As in the previous subsection the difference from the time used by ARPACK becomes larger as $m$ increases. This behavoir is the same as experienced in the previous subsection.

## 4.4 Larger matrices

The reason why the matrices has been kept to a maximum size of 1000 is that it becomes much more time consuming to construct matrices with a desired set of eigenvalues when the size becomes big. Therefore some test will be done here with matrices where not all eigenvalues are known.

Let $A$ be a $10000 \times 10000$ matrix generated by the rand() function in C++ with 1000 added to one of the diagonal elements. The reason why 1000 is added is because then the matrix will then have two eigenvalues around 1000 and the rest clustered around 0 with magnitude less than 10. With $k = 2$ and $m = 5$ this gives $T_C = 9170$ and $T_A = 11120$. However, with $m = 40$ one gets $T_C = 26580$ and $T_A = 27520$. This indicates that the choice of $m$ can be quite important with respect to time used.

For this example let $A$ be as in the previous paragraph, if an accuracy of the order of $10^{-15}$ is not required, the use of a higher tolerance can decrease the used time. With a tolerance of 1e-5 and $m = 5$ the times used are $T_C = 5240$ and $T_A = 5890$.

## 4.5 Conclusions

The tests show that the precision of this implementation is on the level of the precision of ARPACK. Further that for small $m$ the time used is close to the time used by ARPACK, however, as $m$ increases the deviation in time increases. This indicates that the parts of the algorithm which are of complexity $O(m^3)$ should be given closer consideration. This is mainly the algorithm for finding eigenvalues and eigenvectors of the tridiagonal matrix. Finding all the eigenvectors seems might be unecessary and it would be desirable to calculate the eigenvalues in $O(m^2)$ floating point operation and measure error in another way. Further the full reorthogonalization requires $O(m^2 n)$ floating point operations and a partial reorthogonalization could decrease the number of full orthogonalization done.

# 5 Brief tests on proposed use

## 5.1 Euclidean case

Let $D = [d_{ij}]$ be a distance matrix and construct half-squared distances $A = [-\frac{d_{ij}^2}{2}]$. Then from this construct a centred matrix $B = [a_{ij} - \bar{a}_{i.} - \bar{a}_{.j} + \bar{a}_{..}]$, and find the two eigenpairs with largest eigenvalues.

Using only the Manhattan distance on $21 \times 21$ uniform grid, that is only four possible direction to move at each node, one gets the eigenvalues

73.69
73.69
-20.04
10.80
6.69

Using the eigenvectors of the two largest eigenvalues suitably scaled gives a maximum distance of 2 and a maximum deviation from true distance and Euclidean distance of 0.481.

By also movement to closest diagonal neighbour, that is also allowing 8 different directions to move at each node, one gets the eigenvalues

44.40
44.40
2.32
-1.96
-1.59

Using the eigenvectors corresponding to the largest eigenvalue suitably scaled gives a maximum distance of 1.414 and maximum deviation between true distance and Euclidean distance in the new coordinates of 0.0716.

Further if one uses the true Euclidean distances and solves for eigenvalues one gets the eigenvalues

40.43
40.43
7.6e-14
-2.3e-14
7.2e-15

This gives more or less the exactly the correct distances with an error of only 2e-15, but in new coordinates.

It is interesting to note that refining the grid does not improve the situation, using instead a grid size of $41 \times 41$ on the case where one can move to the 8 closest nodes gives the eigenvalues

162.4
162.4
8.382
-7.167
-5.840

This gives a maximum distance of 1.414 and a maximum deviation from true distances from new Euclidean coordinates of 0.07535.

This shows that a scheme which allows only movement in certain directions at each step cannot improve the distance measurement. To improve this it is necessary to increase the number of angles at which one can move.

## 5.2 Rotating azimuth field

This subsection will use the field $\theta(i,j) = (x_i - y_j)\pi$ and the distance measure

$$d(\mathbf{z}_i, \mathbf{z}_j) = \sqrt{(\mathbf{z}_i - \mathbf{z}_j)^T K (\mathbf{z}_i - \mathbf{z}_j)}$$

with

$$K = \frac{1}{R^2 S^2} \begin{bmatrix} R^2 \cos^2\theta + S^2 \sin^2\theta & (S^2 - R^2)\cos\theta\sin\theta \\ (S^2 - R^2)\cos\theta\sin\theta & R^2 \sin^2\theta + S^2\cos^2\theta \end{bmatrix}$$

The following values will be used, $R = 1$ and $S = \frac{1}{2}$.

The same $21 \times 21$ grid as in the previous subsection, but with the distance measure above for the eight closest points, gives the eigenvalues

84.46
81.52
13.11
-7.84
-5.41

The $41 \times 41$ grid from the previous subsection, with the distance measure above gives the eigenvalues

307.32
296.59
47.72
-28.66
-19.97

Comparing this to the results in the Euclidean case, one sees the same effects. For the $21 \times 21$ grid the Euclidean case has a ratio between the third and second eigenvalue of 0.052 and the rotating field case has a ratio of 0.16. For the $41 \times 41$ grid the Euclidean case has ratio between the third and the second eigenvalue of 0.057 and the rotating field case has a ratio of 0.16. The rotating azimuth field is not a two dimensional Euclidean space, but all the five largest eigenvalues seems to maintain the same relationship as the grid is refined. Further the error seems to be of the same order as the error made in the Euclidean case.

# 6 Closing remarks

The code produced from this project works and finds eigenvalues at a high accuracy in the cases tested, but there is room for improvement of efficiency. However, more tests are required to ensure that it works for a variety of cases.