Spectral decomposition using the Lanczos method

E. Biegert and N. Konopliv

June 11, 2014

Contents

1 Introduction

This project was about partitioning a set of particles with equal radii in such a way that if the particles were moving in random directions there would be very few imminent collisions between particles across partitions. Each particle was defined by the location of its center. To accomplish

the partitioning, we used the particle positions to generate a graph with vertices corresponding to particles and edge weights based on the distance between particles. If the particles were further apart than a certain threshold, no edge was created. The Fiedler vector of the laplacian of this graph was then used to partition the particles in half.

We calculated the Fiedler vector by using the Lanczos algorithm with partial reorthogonalization and the QR algorithm for finding the eigenvalues and eigenvectors of a tridiagonal matrix. Lanczos was implemented in parallel with MPI and QR was implemented in serial. We experimented with different edge weights and different levels of partial reorthogonalization. We also tested the performance of our code with problem size and number of processors.

2 Generating the laplacian matrix

2.1 Data distribution

Our code is fed an ASCII input file containing the total number of particles on the first line and one particle position (x,y) on each line after that. Each MPI process, one at a time, opens the input file and reads a certain subset of particle positions into memory, based on the the total number of particles, the total number of MPI processes and its rank. Particles are divided as evenly as possible among processors. There is no particular pattern of particle positions assigned to each processor. It is completely random. We thought about assigning particles to processors based on particle position so that each processor would have neighbors and would only need to communicate with a subset of the other processors. However, if the particles are moving and being re-assigned to different processors over time, there is no longer any set pattern that would define what a neighboring processor is. Also, if the particles had a non-uniform distribution in space, it is conceivable that a processor would be assigned two clumps of particles that are far apart, with a larger clump in between assigned to another processor. Because we wanted flexibility, the particles were initially assigned to processors without regard to spatial location.

2.2 Calculating edge weights

The Laplacian matrix in our code is implemented as a linked list, with one entry for every non-zero location away from the diagonal. The diagonal is stored separately in a contiguous block of memory. The matrix is divided among the processors by blocks of rows. Each processor first calculates off-diagonal entries corresponding to pairs of particles that it owns. For each pair of particles that are a distance apart that is less than the threshold distance, two off-diagonal entries are added. Each time an off-diagonal entry is added, the weight is added to the corresponding diagonal entry. After the processors are done looking for edges between particles that they own, a carousel algorithm is used to calculate the remaining edges in parallel. Each processor passes its particles to the processor with rank + 1 and the highest rank processor passes to rank 0. The processors look for edges between particles that they own and the guest particles that the just received. When that is finished, the guest particles are passed and the process repeats until each processor has checked for edges between its particles and every other particle. This process could be made more efficient by up to a factor of 2. For example, as it stands, processors 0 and 1 would

both look for edges between processor 0 particles and processor 1 particles. A possible change would be to have processors only look at at half of the particles owned by the processor with lower rank. For example when processor 1 has guest particles from processor 0, it would only look at the first half of processor 0 particles and send information on the edges it found back to processor 0 at the end. When processor 0 has guest particles from processor 1, it would look at the other half of processor 0 particles and send information on the edges it found back to processor 1. We did not implement this because we ran out of time.

2.3 Edge weight functions

The edge weights were given by a formula of distance between the particles:

$$w(d) = \frac{1}{(d - 0.9)^n} \tag{1}$$

where n could be chosen. The radius of the particles in our tests was 0.5, so the minimum value of d was 1. The volume fraction of the particles in the domain was always 0.3 in our tests. If the distance between the particles was greater than some threshold, chosen to be 3, no edge was created. The particles were partitioned in half using the Fiedler vector. Those with a value in the Fiedler vector greater than the median were assigned to one half and the others were assigned to the other half. Assuming a number of processors equal to a power of 2, this process can be repeated until there are N divisions, where N is the number of processors. We would have $N/2$ processors work on half of the particles from the first partition and $N/2$ processors work on the other half. These halves would then be partitioned in half and $N/4$ processors assigned to each group. The process would repeat until there were N partitions. In our tests, we only did the first partition in half because we ran out of time and found plenty of other interesting things to look at.

The effect of chosing n can be seen in figures 1 through 5. Chosing a larger n makes edges between particles that are close have weights that are much larger than for particles that are far apart. In general, larger values of n give better partitioning results. Intuitively, this was because it was harder to cut edges between particles that were close together while it was easier to cut edges between particles that were far apart.

When n was larger, it was tempting to lower the threshold since the edge weights at distances close to 3 were small. The problem with this was that the graph would become disconnected if the threshold was too small. It was necessary to have some very weak edges to keep the graph connected. A threshold of 3 was probably a little too high for the particle density chosen, but it ensured we always had a connected graph. The weight function and threshold together set the range of weights that were possible in the laplacian. The ratio of the largest to smallest possible weights in the laplacian was a good indicator of how difficult it would be to find the Fiedler vector. When the ratio was large, the Fiedler value took a long time to converge. The Fiedler vector was considered converged when the Fiedler value converged. For large ratios, the Lanczos vectors also had a higher tendency to lose orthogonality. The wide range of values in the laplacian made roundoff errors more harmful. Partial reorthogonalization was less effective - when reorthogonalizations occurred, the vector had to reorthogonalized against more of the previous Lanczos vectors. A more complete discussion can be found in section 3.3. When n was 5, the ratio was order 10^7. This meant that our code had to do more work, but choosing $n = 5$ did give noticeably better partitions.

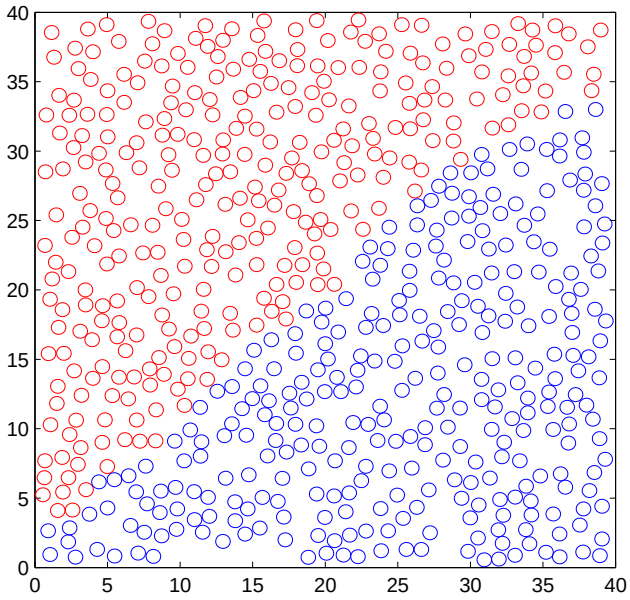

Figure 1: Particle partitions with $n = 1$ in the weight function (1). The number of particles here is 661.

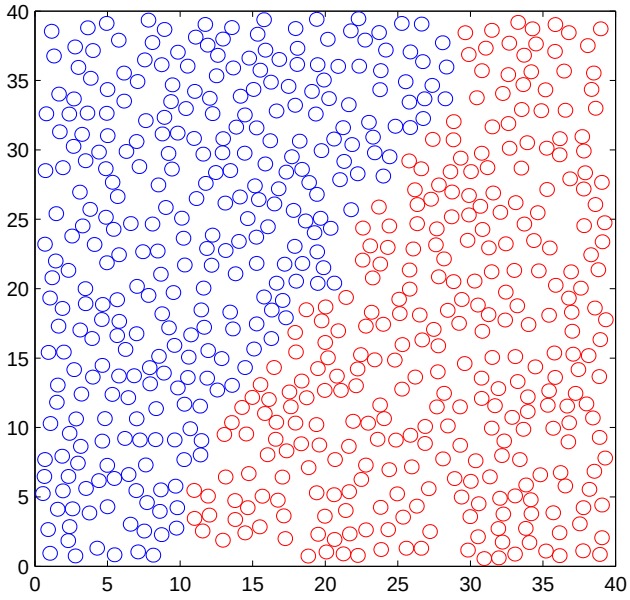

Figure 2: Particle partitions with $n = 2$ in the weight function (1). The number of particles here is 661.

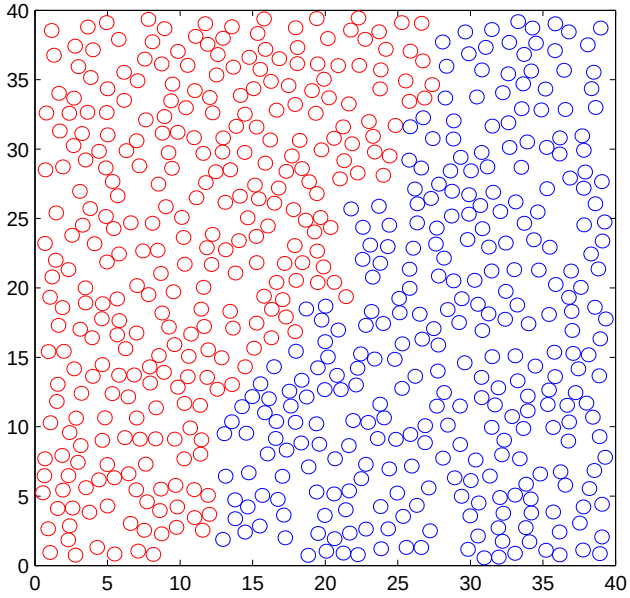

Figure 3: Particle partitions with $n = 3$ in the weight function (1). The number of particles here is 661.

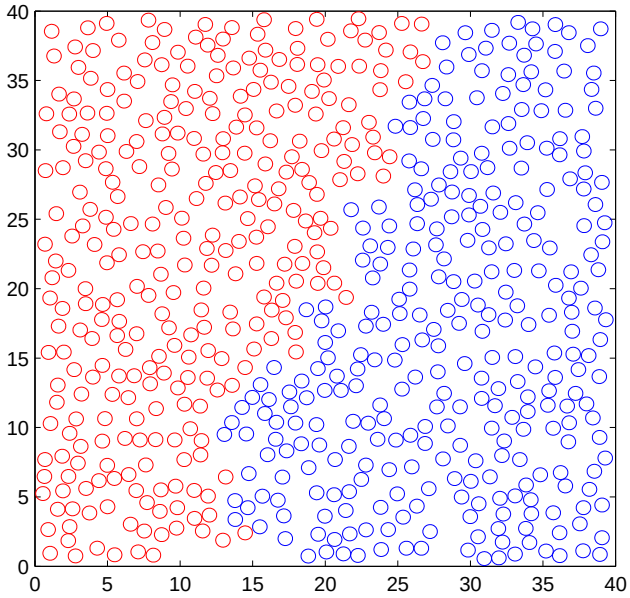

Figure 4: Particle partitions with $n = 4$ in the weight function (1). The number of particles here is 661.

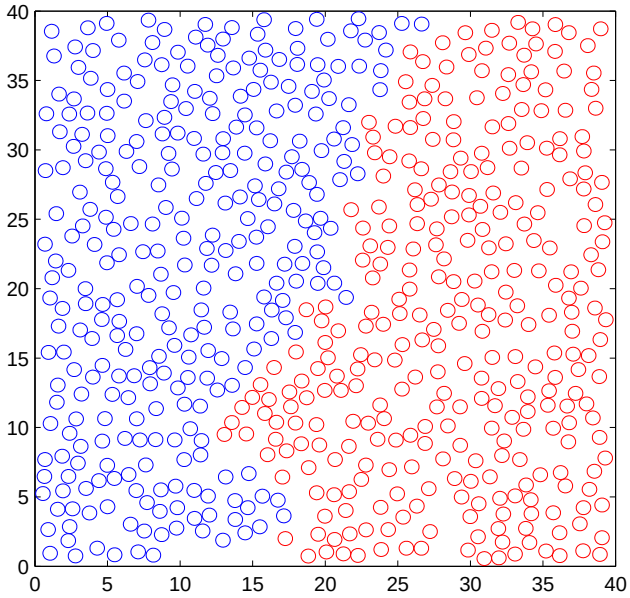

Figure 5: Particle partitions with $n = 5$ in the weight function (1). The number of particles here is 661.

3 Calculation of the Fiedler value

3.1 Lanczos algorithm

The Lanczos algorithm provides an inexpensive first step towards calculating the eigenvalues and eigenvectors of a matrix. Given an $m \times m$ symmetric matrix $\mathbf{A}$, this algorithm can be carried out for n steps, yielding

$$\mathbf{A} = \mathbf{Q}^T \mathbf{T} \mathbf{Q}$$

where $\mathbf{T}$ is an $n \times n$ tridiagonal matrix

$$\mathbf{T} = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix}$$

and the $m \times n$ matrix $\mathbf{Q}$ contains the Lanczos vectors $\mathbf{q}_j$, which are orthogonal to one another:

$$\mathbf{Q} = [\mathbf{q}_1 \quad \mathbf{q}_2 \quad \dots \quad \mathbf{q}_n]$$

If we can diagonalize the tridiagonal matrix $\mathbf{T}$ as

$$\mathbf{T} = \mathbf{V}^T \mathbf{D} \mathbf{V}$$

then we will have diagonalized the starting matrix:

$$\mathbf{A} = \mathbf{Q}^T\mathbf{T}\mathbf{Q} = \mathbf{Q}^T\mathbf{V}^T\mathbf{D}\mathbf{V}\mathbf{Q} = \mathbf{W}^T\mathbf{D}\mathbf{W}$$

An approximate subset of the eigenvalues and eigenvectors of $\mathbf{A}$ are then given by $\texttt{diag}(\mathbf{D})$ and the corresponding columns of $\mathbf{W}$, respectively. This method converges to the largest and smallest eigenvalues first, working its way towards the middle eigenvalues with additional steps. There cannot be more Lanczos steps than there are elements along one dimension of the matrix ($m \geq n$).

The algorithm, presented in Appendix A.1, generates the tridiagonal matrix $\mathbf{T}$ with less computational cost than that required by Householder or Givens rotations [2]. A problem with this method, however, is that it depends on the Lanczos vectors $\mathbf{q}_j$ maintaining their orthogonality. Because the vector $\mathbf{q}_i$ only depends on $\mathbf{q}_{i-1}$ and $\mathbf{q}_{i-2}$, round-off errors can build up over several steps.

When this "deorthogonalization" happens, one or more Ritz vectors may diverge from the vector to which they were converging, moving instead towards another vector and resulting in an eigenvalue duplicity. Furthermore, these "shifts" will continue to happen so that, depending on the convergence criteria and behavior of the matrix, the expected set of eigenvalues and eigenvectors is never found within a given tolerance.

Reorthogonalization, that is, enforcing that $\mathbf{q}_i$ be orthogonal to all previous vectors, prevents the aforementioned problems from occuring. The next section explains how implement partial reorthogonalization, in which we reorthogonalize only against the "least orthogonal" vectors.

3.2 Partial Reorthogonalization

In order to prevent the deorthogonalization of Lanczos vectors, one of several strategies can be taken. The most simple—and expensive—strategy, which we refer to as full reorthogonalization, would be to reorthogonalize the new Lanczos vector against all the previously generated vectors. However, it turns out that deorthogonalization occurs for a subset of the previous vectors. Thus, we could instead reorthogonalize only against this subset. To find this subset, we could employ a strategy such as the following:

- Reorthogonalize $\mathbf{q}_{j+1}$ against $\mathbf{q}_k$ if $\mathbf{q}_k^T\mathbf{q}_{j+1} >= \texttt{TOL}$ for $k = 1, \ldots, j$
- Do nothing with $\mathbf{q}_{j+1}$ and $\mathbf{q}_k$ if $\mathbf{q}_k^T\mathbf{q}_{j+1} < \texttt{TOL}$ for $k = 1, \ldots, j$

for some given tolerance $\texttt{TOL}$.

However, calculating the inner product between $\mathbf{q}_{j+1}$ and $\mathbf{q}_k$ is expensive, particularly if we have to do it for all $k = 1, \ldots, j$ for each Lanczos step (this also gets us halfway to full reorthogonalization anyway).

Simon [4] studied the magnitudes of the inner products $\mathbf{q}_k^T\mathbf{q}_{j+1}$ and noticed a pattern from which he was able to formulate an algorithm. Representing the inner products with an approximation

$$\omega_{j,k} \approx \mathbf{q}_j^T\mathbf{q}_k = \mathbf{q}_k^T\mathbf{q}_j$$

he saw a dependence of $\omega_{j+1,k}$ on previously known values of ω. This algorithm, presented in Appendix A.2, mimics the propagation of round-off errors in these inner products. As we shall see in the results section, this method works quite well, albeit with some caveats.

The real beauty of Simon's partial reorthogonalization (PRO) is that we can keep track of which Lanczos vectors are losing their orthogonality without explicitly calculating their inner products. Thus, we can maintain the spirit of the Lanczos algorithm by making inexpensive updates of our variables using only information from the previous step or two.

Simon's algorithm decides which vectors to reorthogonalize q_{j+1} against by searching through $|\omega_{j+1,k}|$ for values greater than $\sqrt{\epsilon}$, where ϵ is the machine precision. According to Simon, the threshold of $\sqrt{\epsilon}$ is enough to ensure the accurate calculation of eigenvalues and eigenvectors. If no values exceeding the threshold of $\sqrt{\epsilon}$ are found, no reorthogonalization is performed. If some values do exceed the threshold, $|\omega_{j+1,k}|$ is searched in an outward fashion from those places. When those searches hit values lower than a specified value, η, the search stops and the reorthogonalization is done on all vectors that ended up in one of the search radii. Simon recommended $\eta = \epsilon^{0.75}$, but we found we needed an even smaller η.

3.3 Selection of η

The choice of η in the partial reorthogonalization process had an important impact on the results. Choosing a smaller η led to more reorthogonalizations, which was more work, but sometimes necessary to ensure accurate results. Because the estimates of the loss of orthogonality had a random component, it was possible for a worse than usual Lanczos vector to appear that was not seen in the estimate. For example, say that the 100th Lanczos vector generated was perfectly orthogonal to every other Lanczos vector except the 20th one, which it was still mostly orthogonal with but had an unusually high dot product with. If this error was not captured in the orthogonality estimate, the partial reorthogonalization would not orthogonalize against the 20th vector and Lanczos would lose orthogonality. We found that most of the time, the orthogonality estimates were very reliable. Figures 6 through 8 show plots of estimated and actual orthogonality. For timed runs, we didn't calculate the actual orthogonality - it was just for these tests. These figures along with figures 9 and 10 show that usually the orthogonality estimate had the same pattern as the actual orthogonality, but was slightly larger, as it should have been. Occaisionally, this is not true as can been seen obviously in figure 6 and at the right side of figure 7. Doing a full reorthogonalization every time the algorithm decides to reorthogonalize can solve this problem, but making η smaller can also help by making the reorthogonalization closer to a full reorthogonalization.

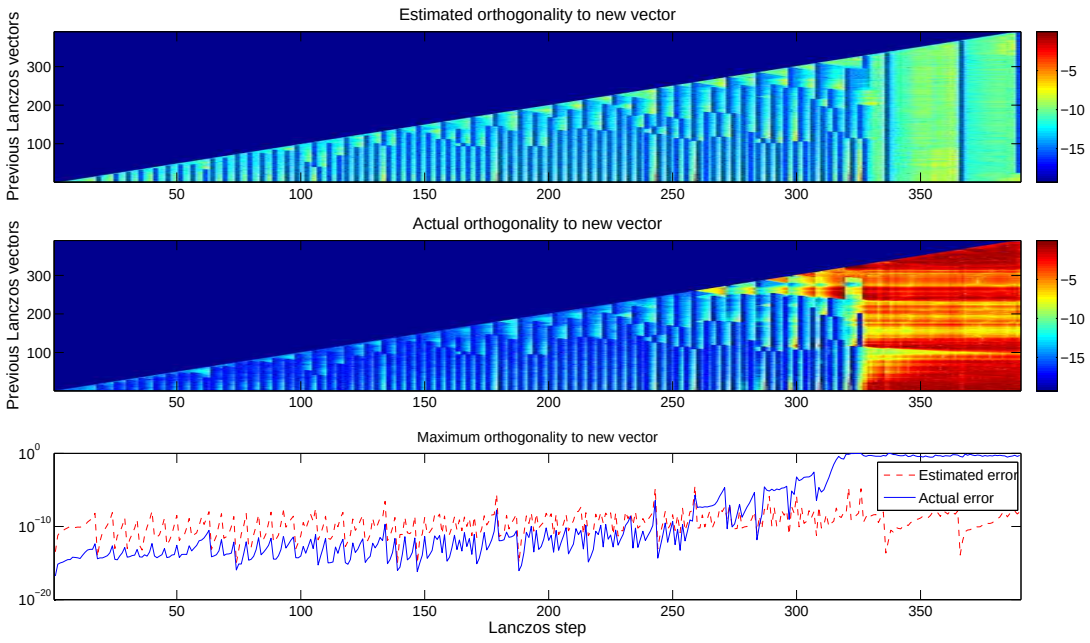

Figure 6: Loss of orthogonality, both actual and estimated, vs. Lanczos step for $\eta = \epsilon^{0.80}$. The base 10 log of the magnitude of the orthogonality is shown in the colormaps. Reorthogonalizations are visible as vertical blue streaks in the colormaps.

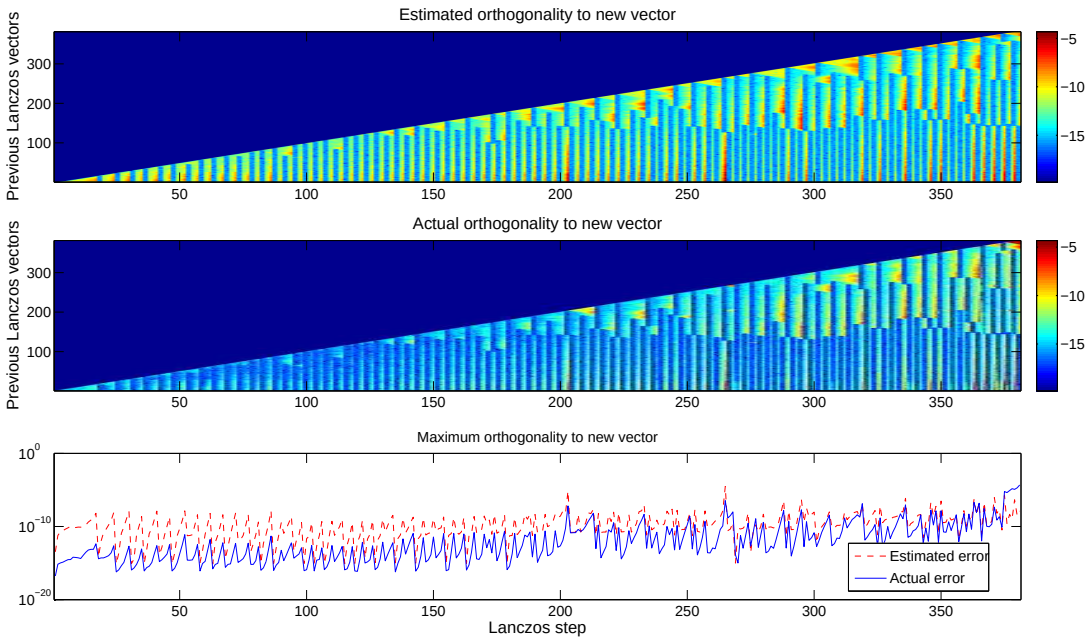

Figure 7: Loss of orthogonality, both actual and estimated, vs. Lanczos step for $\eta = \epsilon^{0.85}$. The base 10 log of the magnitude of the orthogonality is shown in the colormaps. Reorthogonalizations are visible as vertical blue streaks in the colormaps.

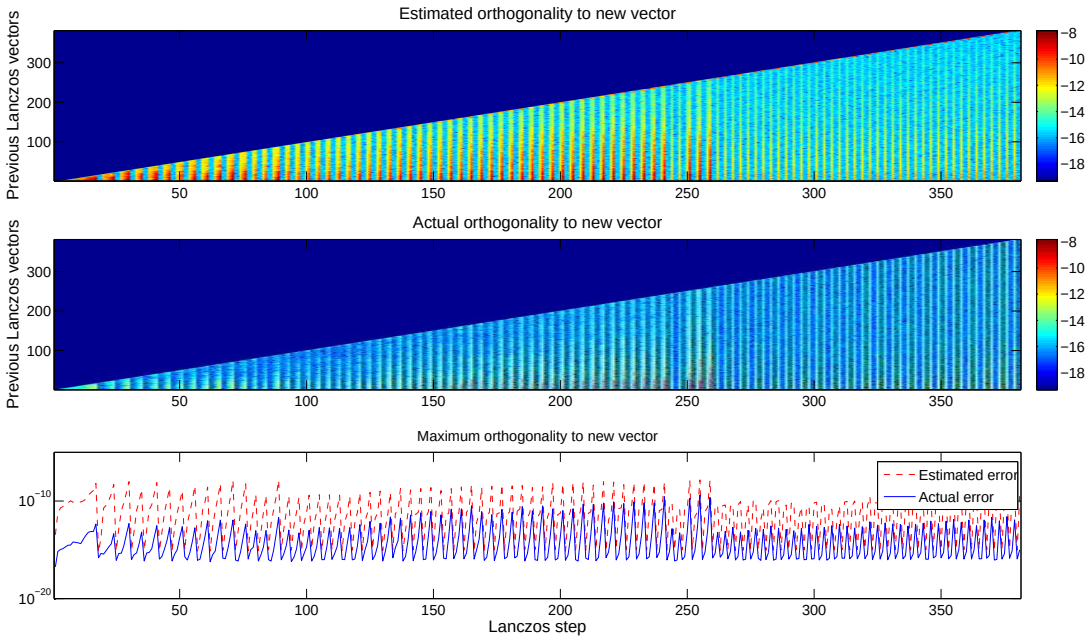

Figure 8: Loss of orthogonality, both actual and estimated, vs. Lanczos step for $\eta = 0$, or full reorthogonalization (not after every Lanczos step though). The base 10 log of the magnitude of the orthogonality is shown in the colormaps. Reorthogonalizations are visible as vertical blue streaks in the colormaps.

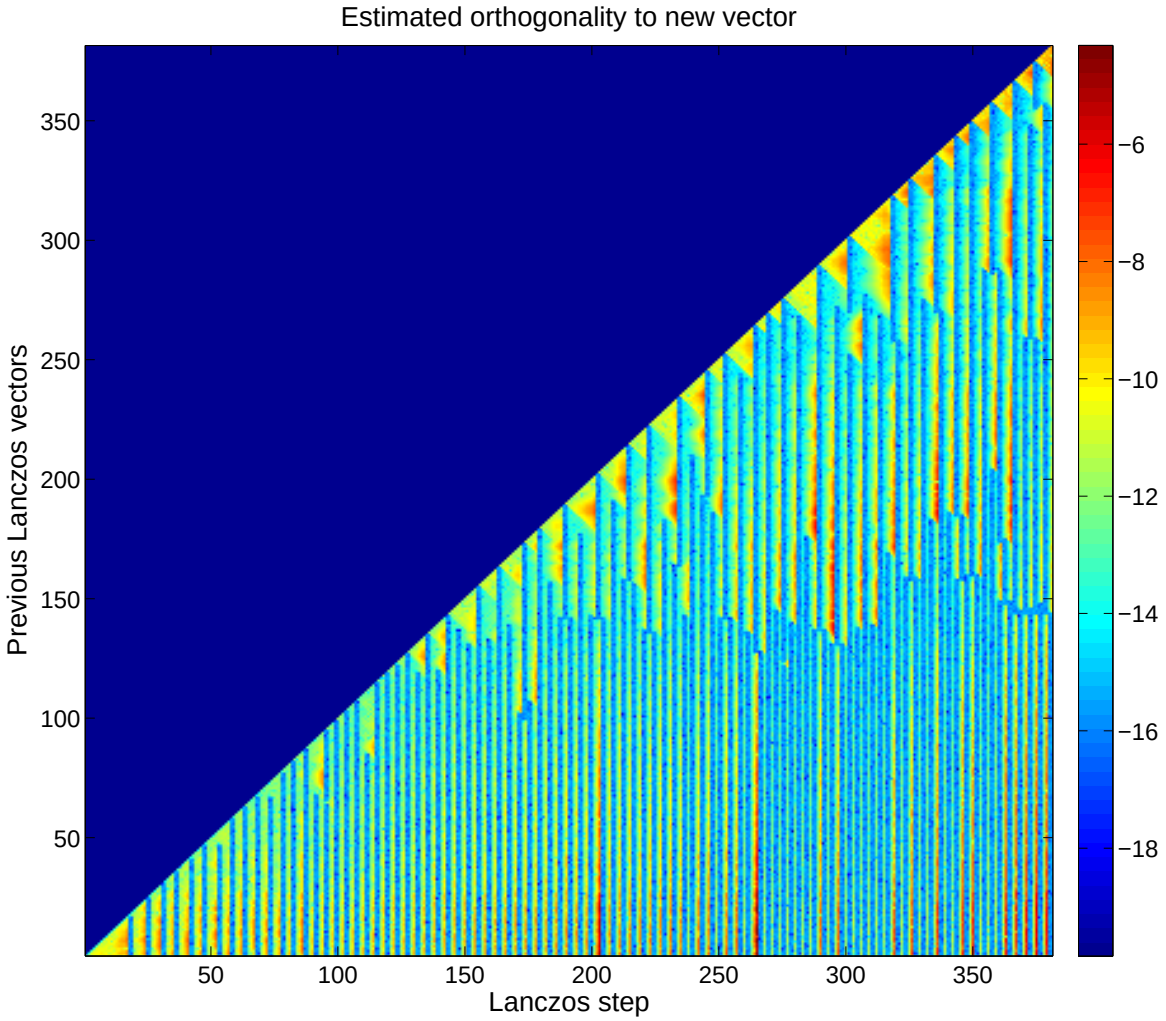

Figure 9: A larger version of the approximated loss of orthogonality from figure 7.

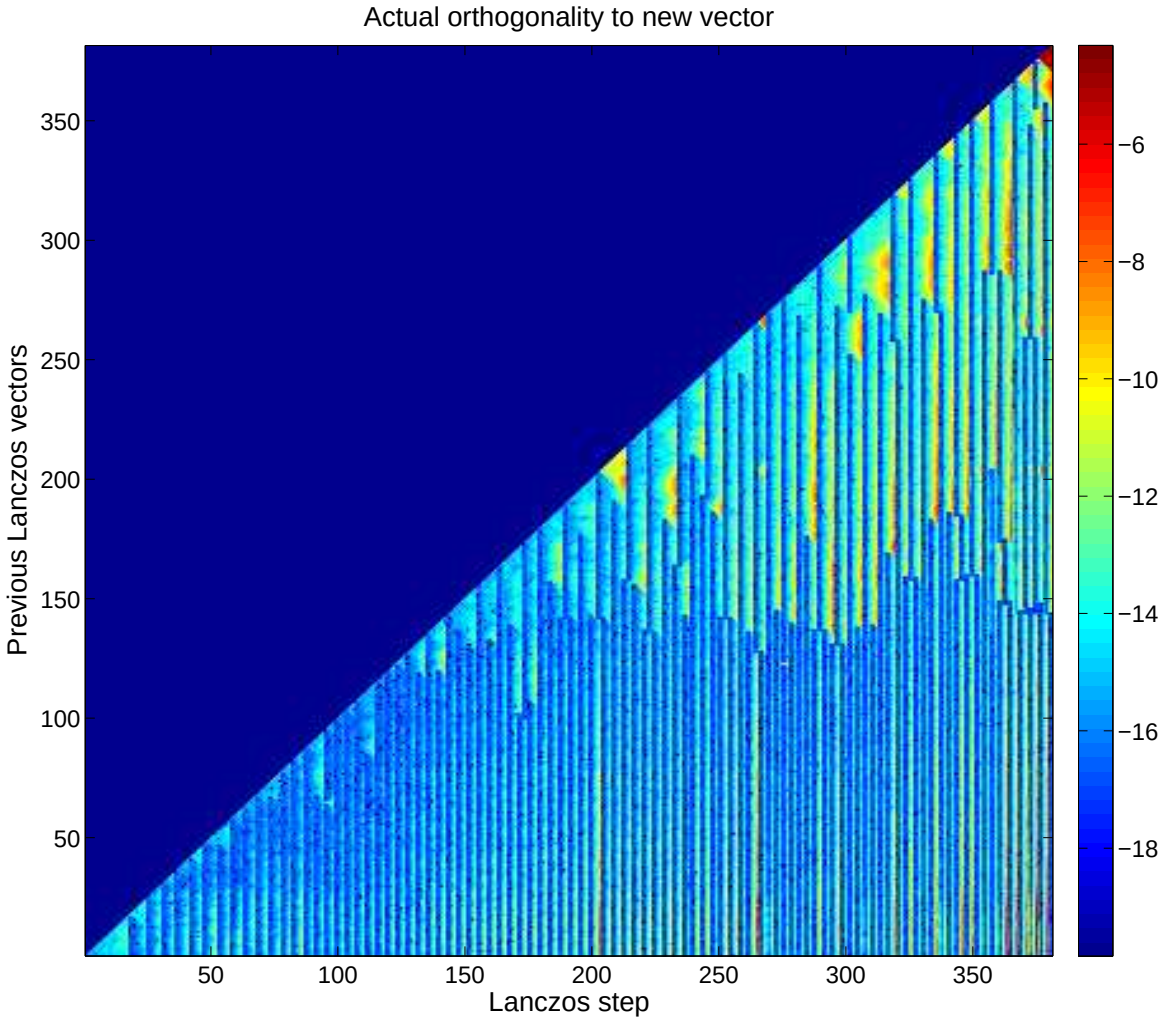

Figure 10: A larger version of the actual loss of orthogonality from figure 7.

3.4 QR solution of tridiagonal matrix

We must now find the eigenvalues and eigenvectors of the tridiagonal matrix $\mathbf{T}$. To do so, we carry out the method described in [3]. The principle behind this method is that, given the QR transformation of $\mathbf{T}$

$$\mathbf{T} = \mathbf{V}\mathbf{R}$$

where $\mathbf{R}$ is an upper-triangular matrix, the sequence

$$\mathbf{T}_j = \mathbf{V}_j\mathbf{R}_j \tag{2a}$$

$$\mathbf{T}_{j+1} = \mathbf{R}_j\mathbf{V}_j \qquad \left(= \mathbf{V}_j^T\mathbf{T}_j\mathbf{V}_j\right) \tag{2b}$$

will converge to the diagonal matrix $\mathbf{T}_j \to \mathbf{D}$ as $j \to \infty$. For a tridiagonal matrix, the QR transformation is carried out through Givens rotations $\mathbf{P}_i$:

$$\mathbf{V}_j^T = \mathbf{P}_1^{(j)}\mathbf{P}_2^{(j)}\cdots\mathbf{P}_{n-1}^{(j)}$$

where

$$\mathbf{P}_i^{(j)} = \begin{array}{c} \\ i \\ i+1 \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & & i & i+1 & & \end{array} \\ \begin{bmatrix} \ddots & & & & & & \\ & 1 & & & & & \\ & & & c & -s & & \\ & & & s & c & & \\ & & & & & 1 & \\ & & & & & & \ddots \end{bmatrix} \end{array}$$

with s on row i and column $i+1$. The values c and s are designed to eliminate the lower-diagonal entry $\mathbf{T}_j(i+1, i)$, with the additional constrain that $s^2 + c^2 = 1$.

The convergence of the QR iterative method can be slow. We can speed up the convergence, however, by estimating the eigenvalues of the matrix and then operating on the *shifted* matrix $\mathbf{T} - k\mathbf{I}$, where k is an estimated eigenvalue and $\mathbf{I}$ is the identity matrix.

The numerical recipes book [3] goes on to show that, if the last row of $\mathbf{V}_j^T$ is known, then all the other rows are can be determined from that row. In other words, if $\mathbf{P}_{n-1}^{(j)}$ is known, then the other $\mathbf{P}_i^{(j)}$ are also known. This fact is used to show that the following method works:

1. The [tridiagonal] matrix $\mathbf{T}_j$ may be split into several blocks, each of which are separated where off-diagonal elements are zero. For each block $\mathbf{B}$ of dimensions $b \times b$:

$$\mathbf{B} = \begin{bmatrix} \alpha_1 & \beta_1 & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \beta_{b-2} & \alpha_{b-1} & \beta_{b-1} & \\ & & & \beta_{b-1} & \alpha_b & \end{bmatrix}$$

(a) Estimate a shift value k by finding the eigenvalue of the 2×2 matrix in the far lower-right corner which is closest to α_b:

$$\lambda_1, \lambda_2 = \text{eig}\left(\begin{bmatrix}\alpha_{b-1} & \beta_{b-1} \\ \beta_{b-1} & \alpha_b\end{bmatrix}\right)$$
$$k = \min(|\lambda_1 - \alpha_b|, |\lambda_2 - \alpha_b|)$$

(b) Operate a symmetric Givens rotation $\mathbf{P}_1$ on $\mathbf{B}$ as if we were eliminating β_1 of the shifted matrix $\mathbf{B} - k\mathbf{I}$:

$$\mathbf{B}' = \mathbf{P}_1^T \mathbf{B} \mathbf{P}_1 = \begin{bmatrix} \alpha_1' & \beta_1' & \gamma' & & \\ \beta_1' & \alpha_2' & \beta_2' & & \\ \gamma' & \beta_2' & \alpha_3 & \beta_3 & \\ & & \ddots & \ddots & \ddots \\ & & & \beta_{b-1} & \alpha_b \end{bmatrix}$$

where

$$\mathbf{P}_1 = \begin{bmatrix} c & -s & & & \\ s & c & & & \\ & & 1 & & \\ & & & \ddots & \end{bmatrix}$$

and γ' is the off-off-diagonal term created by the rotation. This step thus creates a "bulge" in the tridiagonal matrix.

(c) Execute $b-2$ additional rotations $\mathbf{P}_2, \ldots, \mathbf{P}_{b-1}$, each of which seeks to eliminate the newly-generated γ:

$$\mathbf{B}^{(i+1)} = \mathbf{P}_{i+1}^T \mathbf{B}^{(i)} \mathbf{P}_{i+1} = \begin{bmatrix} \ddots & \ddots & 0 & & \\ \ddots & \alpha_{i+1}'' & \beta_{i+1}'' & \gamma^{(i+1)} & \\ 0 & \beta_{i+1}'' & \alpha_{i+2}'' & \beta_{i+2}'' & \\ & \gamma^{(i+1)} & \beta_{i+2}'' & \alpha_{i+3} & \ddots \\ & & & \ddots & \ddots \end{bmatrix}$$

with

$$\mathbf{P}_{i+1} = \begin{array}{c} \\ i+1 \\ i+2 \end{array} \begin{array}{c} \begin{array}{cccccc} & & i+1 & i+2 & & \end{array} \\ \begin{bmatrix} \ddots & & & & & \\ & 1 & & & & \\ & & c & -s & & \\ & & s & c & & \\ & & & & 1 & \\ & & & & & \ddots \end{bmatrix} \end{array}$$

This step thus "chases the bulge down the diagonal."

(d) The resulting matrix $\widetilde{\mathbf{B}}$ will be tridiagonal

$$\widetilde{\mathbf{B}} = \mathbf{P}^T\mathbf{B}\mathbf{P}$$

where

$$\mathbf{P} = \mathbf{P}_1\mathbf{P}_2\cdots\mathbf{P}_{b-1}$$

2. If $\mathbf{P}_i$ is the resulting matrix found for block $\mathbf{B}_i$, $i = 1, \ldots, l$, then

$$\mathbf{V}_j = \mathbf{P}_1\mathbf{P}_2\cdots\mathbf{P}_l$$
$$\mathbf{T}_{j+1} = \mathbf{V}_j^T\mathbf{T}_j\mathbf{V}_j$$

represents one iteration of the sequence described by (2).

3. Iterating on this procedure will cause $\mathbf{T}_{j+1}$ to converge to a diagonal matrix $\mathbf{D}$:

$$\mathbf{V} = \mathbf{V}_1\mathbf{V}_2\ldots\mathbf{V}_J$$
$$\mathbf{D} = \mathbf{V}^T\mathbf{T}\mathbf{V}$$

where convergence has occurred after J iterations.

3.5 Convergence of λ_2

When monitoring a general value to check for convergence, the most obvious criteria to determine if it is converged is to wait until the change between two iterations falls below a certain value. This can lead to false convergence if the convergence is slow, which it can be when calculating the Fiedler value. The convergence is especially slow if the weight function is chosen in such a way that the matrix is not "nice", as discussed in section 2.3. Additionally, it is inefficient to run the QR algorithm to get the Ritz values after every Lanczos step. Both of these issues were resolved by looking at an estimate of the true error and then guessing how many more Lanczos steps would be needed to achieve convergence.

Let λ_2^* be the true Fiedler value and λ_2^k be the second smallest Ritz value after k Lanczos steps that will converge to λ_2^*. Then the true error can be written

$$\lambda_2^k - \lambda_2^* = \lambda_2^k - \lambda_2^{k+1} + \lambda_2^{k+1} - \lambda_2^{k+2} + \lambda_2^{k+2} - \ldots - \lambda_2^* \tag{3}$$

by adding and subtracting λ_2^{k+1}, λ_2^{k+2}, λ_2^{k+3}, etc. Using the triangle inequality gives

$$|\lambda_2^k - \lambda_2^*| \leq |\lambda_2^{k+1} - \lambda_2^k| + |\lambda_2^{k+2} - \lambda_2^{k+1}| + \ldots \tag{4}$$

An estimate for the rate of convergence is then used:

$$\rho = \frac{|\lambda_2^k - \lambda_2^{k-1}|}{|\lambda_2^{k-1} - \lambda_2^{k-2}|} \tag{5}$$

This ρ is assumed to be constant for all k. If ρ is close to 1, λ_2 is converging slowly because it is changing by about the same amount after each step. Figure 11 shows a plot of ρ vs. Lanczos step

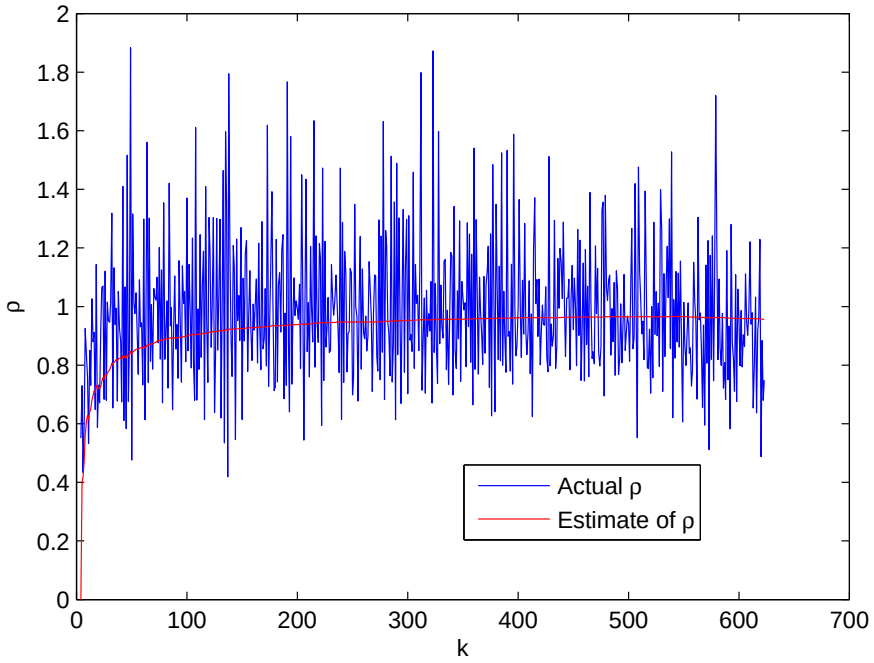

Figure 11: Rate of convergence ρ after k Lanczos steps. The actual ρ is evaluated using (5) and the estimate is evaluated using (8). The size of the matrix was 763.

for matrix of size 763, and figure 12 shows the corresponding convergence of λ_2. The data for ρ is quite noisy, but seems to hover around a constant value. Using (5) allows (4) to be written

$$|\lambda_2^k - \lambda_2^*| \leq \rho|\lambda_2^k - \lambda_2^{k-1}| + \rho^2|\lambda_2^k - \lambda_2^{k-1}| + \rho^3|\lambda_2^k - \lambda_2^{k-1}| + ... \tag{6}$$

The geometric sum on factors of ρ yields the estimate of the true error

$$|\lambda_2^k - \lambda_2^*| \leq \frac{\rho}{1-\rho}|\lambda_2^k - \lambda_2^{k-1}| \tag{7}$$

It is easy to get $|\lambda_2^k - \lambda_2^{k-1}|$ if the Ritz values are calculated after two steps in a row. This means to check the error, we calculate the Ritz values for two consecutive steps. An estimate of ρ is given by

$$\rho = \left(\frac{|\lambda_2^k - \lambda_2^{k-1}|}{|\lambda_2^3 - \lambda_2^2|} \right)^{1/(k-3)} \tag{8}$$

as recommended by Ascher and Petzold [1], except that λ_2^2 is the earliest λ_2^k used instead of λ_1. This is because λ_2^1 is basically a random number. We could have chosen other pairs of λ_2's for the denominator, but this seemed to work well. This estimate of ρ is nice because there is always a fairly large number in the denominator, so division by small numbers producing large errors is not a concern here. It also helps to smooth out *rho*, as shown in figure 11. The estimate of ρ appears to approach a fairly constant value that is close to 1, indicating slow convergence.

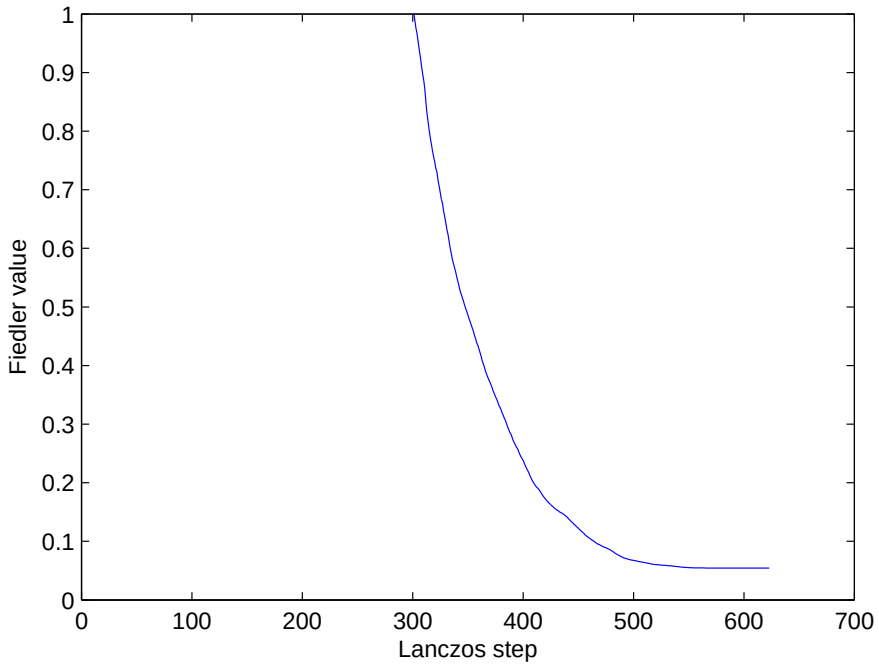

Figure 12: Convergence of λ_2 for the same test run as in figure 11. It takes a large number of Lanczos steps to get convergence of λ_2. The size of the matrix was 763. The y-axis has been zoomed in to show that λ_2 was still converging after many Lanczos steps.

Given a desired error tolerance, the estimate of the true error can also be used to guess how many more Lanczos steps are needed. Setting (7) equal to an error tolerance after m more steps gives

$$\text{ETOL} = \frac{\rho}{1-\rho}\rho^m|\lambda_2^k - \lambda_2^{k-1}| \tag{9}$$

The factor of ρ^m appears because $\rho^m|\lambda_2^k - \lambda_2^{k-1}|$ is an estimate of what $|\lambda_2^k - \lambda_2^{k-1}|$ will be after m more steps. This can be solved for m:

$$m = \log_\rho \frac{\text{ETOL}}{\text{EST}} \tag{10}$$

where

$$\text{EST} = \frac{\rho}{1-\rho}|\lambda_2^k - \lambda_2^{k-1}| \tag{11}$$

is the error estimate that has already been calculated. In our code, instead of taking m more steps, we took a more conservative $0.8m$ more steps. There is a tradeoff here between taking a higher percentage of m more steps (possibly more than 100%) and doing less QR algorithm work, but overshooting and taking more Lanczos steps than necessary. This would have been interesting to examine, but in the interest of time we decided other things were more interesting. Of course, we checked to ensure that we never take more Lanczos steps than the size of the original matrix.

4 Performance

We performed a brief analysis of how our code scales with problem size and number of processors. Due to the non-deterministic nature of the code, we have conducted each test a number of times until we obtained 10 runs that produced the correct solution. However, we have only tested one graph for each problem size. A variety of graphs for each problem size would give a better basis for analyzing the code, but our time constraints prohibited us from pursuing this analysis.

Figures 13 and 14 show how the code scales with problem size. In Figure 13, we see that the QR portion of the code accounts for the vast majority of the time, while the Lanczos algorithm accounts for a smaller portion. The initialization routines that generate the sparse matrix are negligible compared to these other parts, so we will discard it for the rest of this analysis.

In Figure 14, we see that the parallel efficiency increases for a time with increasing problem size, peaking around $N = 1200$. In fact, the parallel efficiency is above 1.0 for all the larger problem sizes. Looking at Figure 15, we can hypothesize the reason for this observation. Here, we see that the QR algorithm accounts for most of the program's runtime for a small number of processors, but this time decreases sharply as the number of processors is increased. Looking at our code, we realized this must be a caching problem. We noticed that our algorithm moves through the slowly-varying index when updating the eigenvector matrix $\mathbf{Q}$. Thus, when there are enough processors, and $\mathbf{Q}$ is spread thinly enough among them, this caching problem disappears.

This caching issue would explain the observed parallel efficiency being greater than one, allowing us to better interpret Figure 14. For small problem sizes, a single processor does not encounter as many caching issues and thus lowers the speedup relative to larger problem sizes. However, for problem sizes larger than $N = 1200$, we see the parallel efficiency decrease. This may be due to the

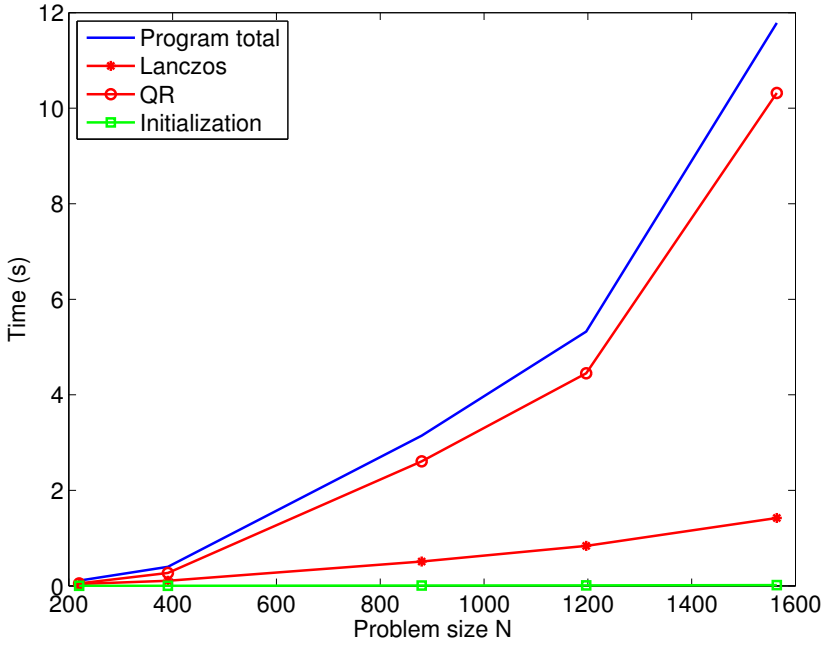

Figure 13: Time individual program components took to run on 8 processors for different problem sizes.

problem size reaching the cache limit on 8 processors. If the problem size continues to increase, we may see another increase in the parallel efficiency when the single processor hits the level in cache size.

Another interesting result is that, for a fixed (low) number of processors, the QR algorithm dominates the program runtime (Figure 13), while for a fixed problem size, the QR algorithm dominates the runtime for a low number of processors, while the Lanczos algorithm dominates for a large number of processors (Figure 15). This result is likely due to communication slowdowns. The Lanczos algorithm requires several communications among the processors, while the QR algorithm, as currently written, runs independently on each processor. We can see this result reflected in Figure 16, where the QR algorithm maintains a parallel efficiency around 1, which decreases slightly for a larger number of processors (because every processor is still calculating the tridiagonal entries of the matrix, whose work does not decrease with the number of processors).

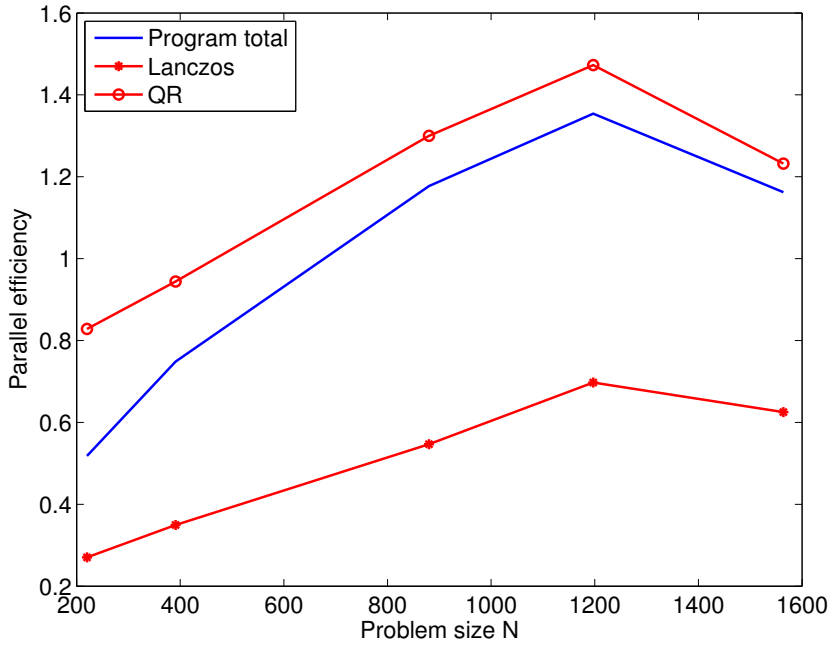

Figure 14: Parallel efficiency of individual program components on 8 processors for different problem sizes.

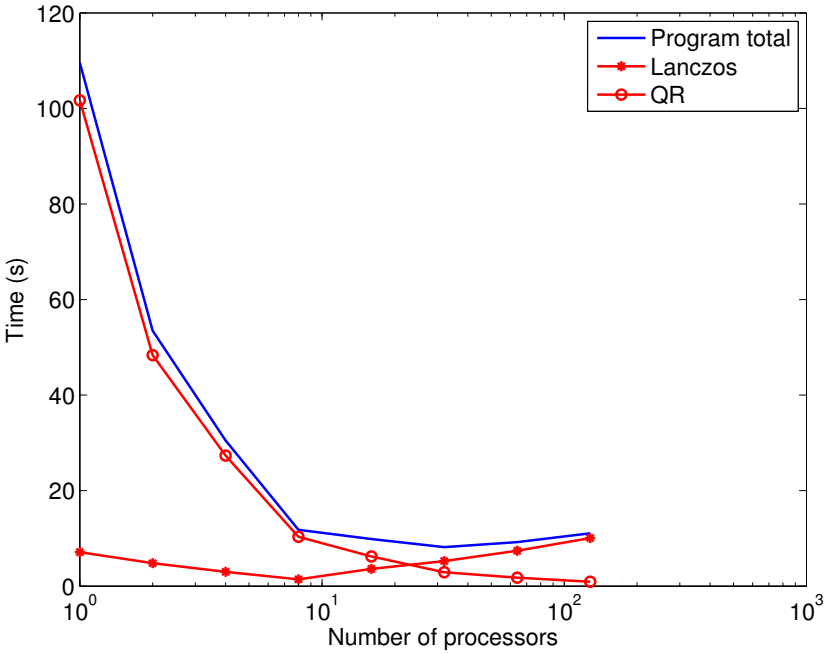

Figure 15: Time individual program components took to run on different numbers of processors for the problem size of $N = 1564$.

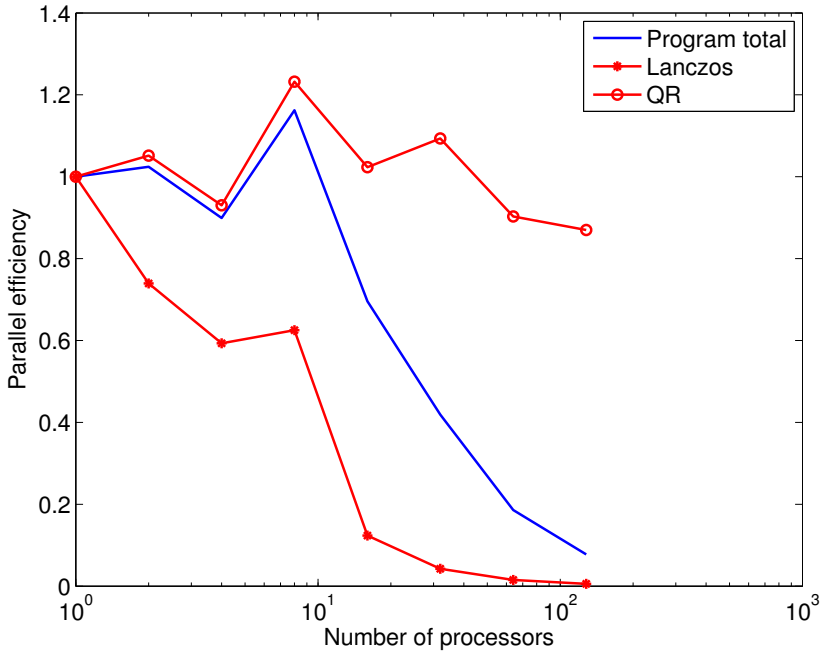

Figure 16: Parallel efficiency of individual program components on different numbers of processors for the problem size of $N = 1564$.

5 Sample Results

Sample partitioning results are shown here for systems with varying numbers of particles. The weight function used was

$$w(d) = \frac{1}{(d - 0.9)^5} \tag{12}$$

A good partition has very few instances where collisions between particles assigned to different partitions are imminent. Only two partitions are shown here, but the partitioning can be recursively repeated to obtain more partitions. The partitions are perfect, but they are pretty good overall. The spectral partitioning did a good job of keeping clumps of particles on a single partition.

We checked our results for accuracy by comparing with the output of Matlab's `eig()` function. When our results were wrong, it was always because of a loss of orthogonality. We would fix this by either trying with another starting vector or lowering η.

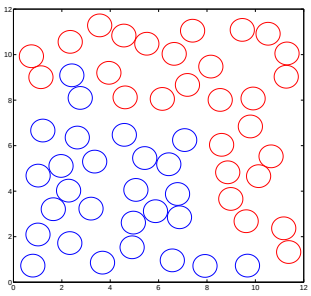

Figure 17: Partition for 55 particles

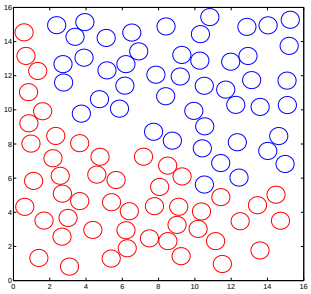

Figure 18: Partition for 97 particles

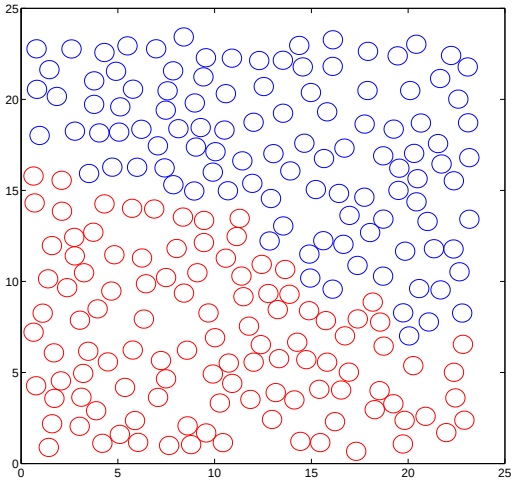

Figure 19: Partition for 220 particles

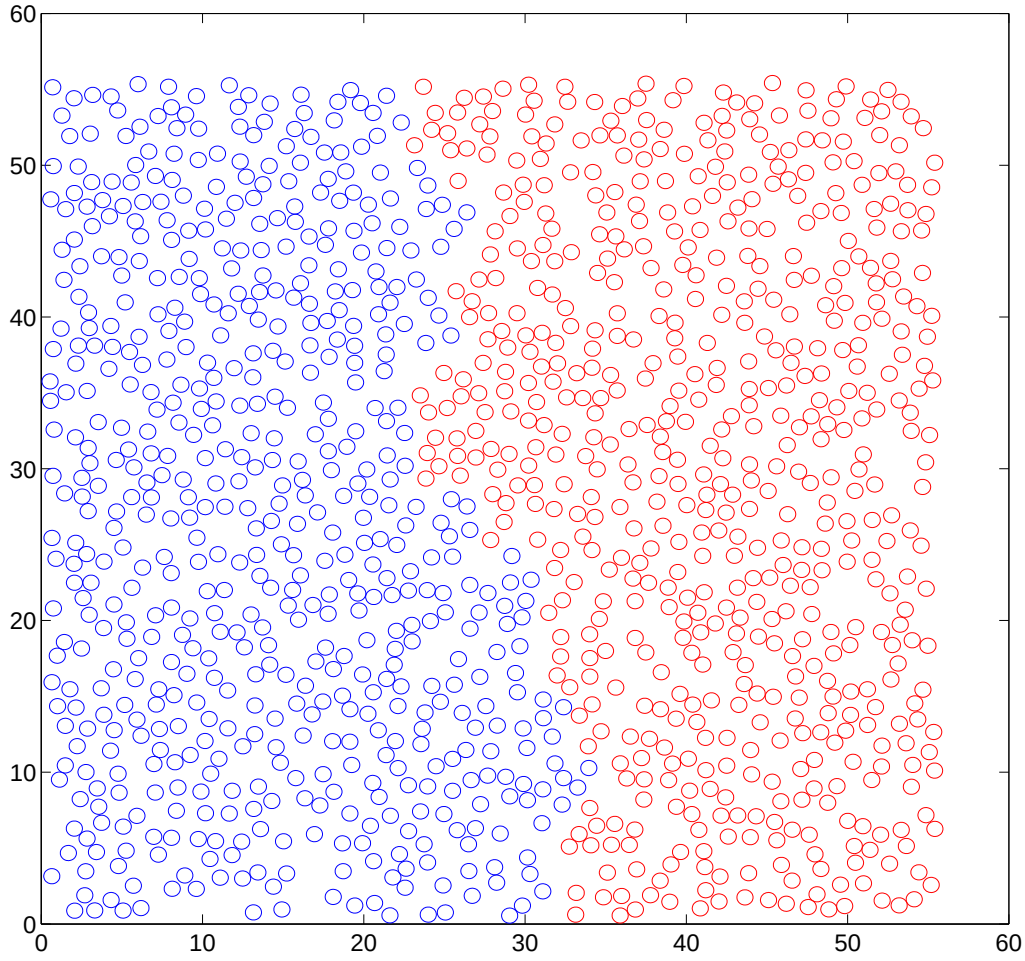

Figure 20: Partition for 1197 particles

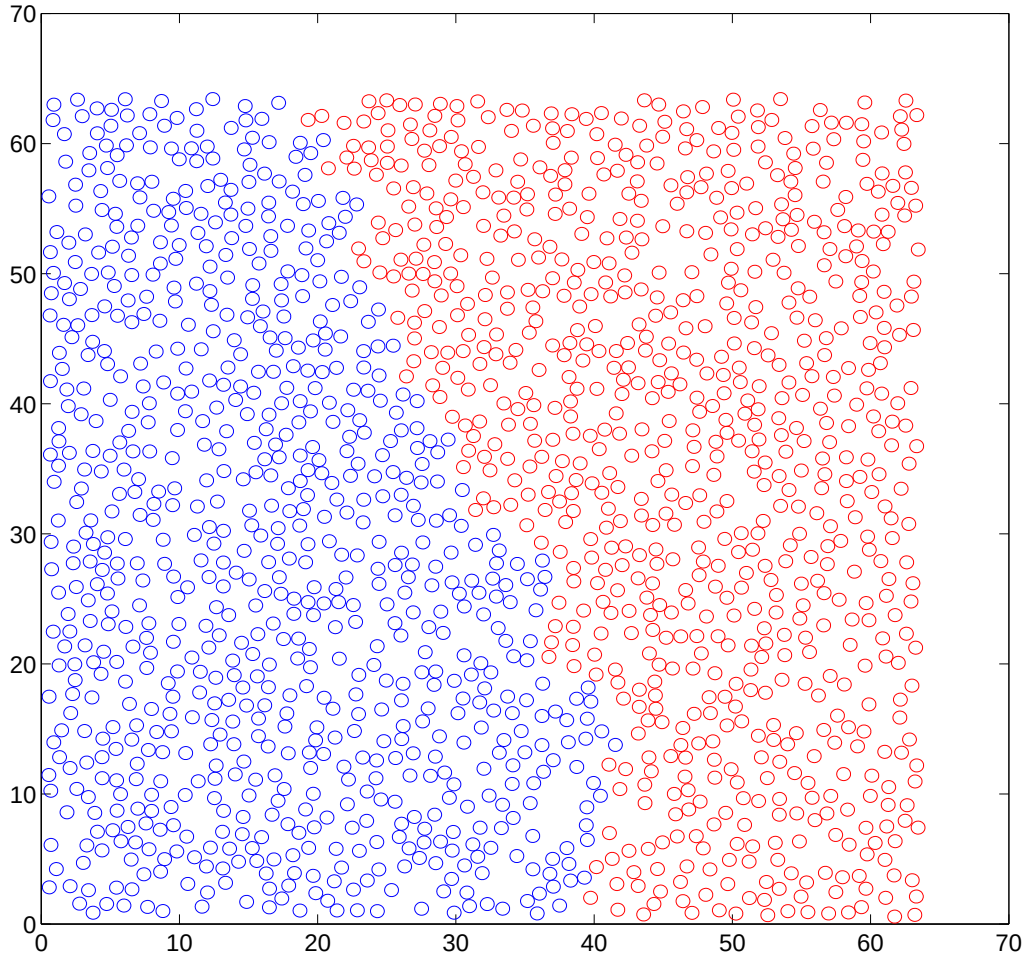

Figure 21: Partition for 1564 particles

A Lanczos method with partial reorthogonalization

A.1 Lanczos algorithm

Given an $m \times m$ matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, we will carry out the Lanczos method for n steps, giving us n approximate eigenvalues and corresponding eigenvectors.

1. Choose a random starting vector $\mathbf{q}_1$ and let $\mathbf{q}_0 = \mathbf{0}$.
2. `for j = 1:n`

   $\mathbf{v} = \mathbf{A}\mathbf{q}_j$
   $\alpha_j = \mathbf{v}^T \mathbf{q}_j$
   $\mathbf{v} = \mathbf{v} - \alpha_j \mathbf{q}_j - \beta_{j-1}\mathbf{q}_{j-1}$
   $\beta_j = ||\mathbf{v}||$
   Partially reorthogonalize $\mathbf{v}$ against $\mathbf{q}_1, \ldots, \mathbf{q}_j$
   $\mathbf{q}_{j+1} = \dfrac{\mathbf{v}}{\beta_j}$

This method will produce the matrices $\mathbf{Q}$ and $\mathbf{T}$ such that

$$\mathbf{A} = \mathbf{Q}^T \mathbf{T} \mathbf{Q}$$

where $\mathbf{T} \in \mathbb{R}^{n \times n}$ is a tridiagonal matrix

$$\mathbf{T} = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix}$$

and $\mathbf{Q} \in \mathbb{R}^{m \times n}$ contains the vectors $\mathbf{q}_j$:

$$\mathbf{Q} = [\mathbf{q}_1 \quad \mathbf{q}_2 \quad \ldots \quad \mathbf{q}_n]$$

All that is needed to carry out Lanczos are matrix vector multiplications. With the sparse matrix data structure discussed in section 2, matrix-vector multiplications are easy. To make things as simple as possible, the data for the vector is communicated to all processors so that all processsors contain the entire vector in memory. Each processor can then calculate its part of the product vector by looping through the linked list containing the off-diagonal entries and then applying the diagonal entries.

It should also be noted that when a reorthogonalization is done, it must be done in two consecutive steps. Otherwise roundoff errors from the previous Lanczos vector will still propagate forward.

A.2 Partial reorthogonalization

To avoid expensive schemes, such as orthogonalizing the new vector $\mathbf{q}_{j+1}$ against all previous vectors, we have implemented the method of Horst D. Simon [4]. He devises a method which seeks to approximate the inner products of the Lanczos vectors by

$\omega_{j+1,j+1} = 1$
$\omega_{j+1,j} = \psi_j$
`for k = 1:j`

$$\omega_{j+1,k} = \frac{1}{\beta_j}\left[\beta_k\omega_{j,k+1} + (\alpha_k - \alpha_j)\omega_{j,k} + \beta_{k-1}\omega_{j,k-1} - \beta_{j-1}\omega_{j-1,k} + \theta_{j,k}\right]$$

where

$$\psi_j = \epsilon\frac{\beta_1}{\beta_j} m \,\texttt{randn(0, 0.6)}$$

$$\theta_{j,k} = \epsilon\frac{\beta_k}{\beta j}\,\texttt{randn(0, 0.3)}$$

and `randn(m, s)` generates a random number in a normal distribution with mean `m` and standard deviation `s`.

When a reorthogonalization was performed, the $\omega_{j+1,k}$ corresponding to the vectors which were orthogonalized against were set to `randn(0, 1.5`ϵ`)`.

B Code

B.1 structs.h

```
#ifndef _STRUCTS
#define _STRUCTS

typedef struct
{
double x;
double y;
int id;
} PARTICLE;

typedef struct
{
// MPI rank and number of processors
int rank, nproc;

// Debug file
FILE *dbg;
char real_filename[50], approx_filename[50], flag_filename[50];

// Particle data
```

```
PARTICLE *particles;
int Np;
int *Np_local;
int *vecstart;
int Np_local_max;
} LDATA;

typedef struct spmat_entry_t
{
int row; //local
double weight;
int column; //global
struct spmat_entry_t *next;
} SPMAT_ENTRY;

typedef struct
{
double *diag;
SPMAT_ENTRY *root;
} SPMAT;

typedef struct
{
int start;
int steps;
double **Q;
double *alpha;
double *beta;

// Variables necessary for restarting
double **w, *w0;
int *flag_reorth;
int firststep;
} LANCZOS;

#undef DEBUG
#define CALC_REORTH

#endif
```

B.2 main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <mpi.h>
#include <math.h>
#include <time.h>
```

```
#include "structs.h"
#include "matvec.h"
#include "lanczos.h"
#include "qr_tridiag.h"

#define min(a, b) ((a)<(b)) ? (a) : (b)

#define ETOL 1e-6


/*****************************************************************************/
/*
*/
/*****************************************************************************/
void save_vector(const char *filename, double *lvec, LDATA *data)
{
int i;
int signal;
MPI_Status status;
FILE *fp;

if (data->rank != 0)
{
MPI_Recv(&signal, 1, MPI_INT, data->rank-1, 0, MPI_COMM_WORLD, &status);
fp = fopen(filename, "a");
}
else
{
fp = fopen(filename, "w");
}

for (i = 0; i < data->Np_local[data->rank]; i++)
fprintf(fp, "%.14e\n", lvec[i]);

fclose(fp);
if (data->rank != data->nproc-1)
MPI_Send(&signal, 1, MPI_INT, data->rank+1, 0, MPI_COMM_WORLD);
}



/*****************************************************************************/
/*
 */
/*****************************************************************************/
void get_id_startstop(LDATA *data, int *id_start, int *id_stop)
{
int rank = data -> rank;
int nproc = data -> nproc;
int Np = data -> Np;
```

```
//divide the particles evenly among processors, assign "leftover" particles to low rank processors so some low rank processors g
if (rank < Np % nproc) //extra particle processors
{
*id_start = rank * (Np / nproc + 1);
*id_stop = *id_start + Np / nproc;
}
else
{
*id_start = rank * (Np / nproc) + (Np % nproc);
*id_stop = *id_start + Np / nproc - 1;
}
}


/**********************************************************************/
/*
 */
/**********************************************************************/
void load_particles(char *filename, LDATA *data)
{
FILE *fp;
char junk[100];
int i;
int signal = 0;
int id_start, id_stop;
MPI_Status status;

int rank = data -> rank;
int nproc = data -> nproc;

if (rank != 0)
MPI_Recv(&signal, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &status);

fp = fopen(filename, "r");
fscanf(fp, "%d\n", &data->Np);
get_id_startstop(data, &id_start, &id_stop);

data->Np_local = (int *) malloc(nproc * sizeof(int));
data->Np_local[rank] = id_stop - id_start + 1;

//skip to starting position
for (i = 0; i < id_start; i++)
fgets(junk, 100, fp);

//read the local particles
data->particles = (PARTICLE *) malloc(data->Np_local[rank] * sizeof(PARTICLE));
for (i = 0; i < data->Np_local[rank]; i++)
{
data->particles[i].id = id_start + i;
fscanf(fp, "%lf %lf\n", &data->particles[i].x, &data->particles[i].y);
}
```

```
fclose(fp);

if (rank != nproc-1)
MPI_Send(&signal, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
}


/*************************************************************************/
/*
 */
/*************************************************************************/
void share_Np_local(LDATA *data)
{
int i;
int rank = data -> rank;
int nproc = data -> nproc;

MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, data->Np_local, 1, MPI_INT, MPI_COMM_WORLD);

data->Np_local_max = data->Np_local[0];

data->vecstart = (int *) malloc(nproc*sizeof(int));
data->vecstart[0] = 0;

for (i = 1; i < nproc; i++)
{
if (data->Np_local_max < data->Np_local[i])
data->Np_local_max = data->Np_local[i];

data->vecstart[i] = data->vecstart[i-1] + data->Np_local[i-1];
}
}


/*************************************************************************/
/*
 */
/*************************************************************************/
void clean_data(LDATA *data)
{
free(data->particles);
free(data->vecstart);
free(data->Np_local);
}



/*************************************************************************/
/*
 */
```

```
/*****************************************************************************/
void clean_lancz(LANCZOS *lancz)
{
free(lancz -> w0);
free(&(lancz -> w[lancz->steps - 2]));
free(lancz -> flag_reorth);
free(lancz -> alpha);
free(lancz -> beta);
free_mat(lancz -> Q);
}



/*****************************************************************************/
/*
 */
/*****************************************************************************/
MPI_Datatype make_mpi_particle(void)
{
MPI_Datatype mpi_type;
MPI_Datatype types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
int nitems = 3;
int blocklengths[3] = {1, 1, 1};
MPI_Aint offsets[3];
offsets[0] = offsetof(PARTICLE, x);
offsets[1] = offsetof(PARTICLE, y);
offsets[2] = offsetof(PARTICLE, id);
MPI_Type_create_struct(nitems, blocklengths, offsets, types, &mpi_type);
MPI_Type_commit(&mpi_type);
return mpi_type;
}



/*****************************************************************************/
/*
 */
/*****************************************************************************/
int main(int argc, char **argv)
{
int rank, nproc;
FILE *dbg;
char dbg_filename[100];
int i, j;
int seed;
MPI_Datatype mpi_particle;

LDATA data;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
data.rank = rank;
data.nproc = nproc;

double T0, T1, T2;
double t_total, t_initialize;
double t_lanczos = 0.0;
double t_qr = 0.0;

T0 = MPI_Wtime();

// Initialize random number generator with same seed among all processors
//  if (rank == 0)
//  seed = time(0);
//  MPI_Bcast(&seed, 1, MPI_INT, 0, MPI_COMM_WORLD);
//  srand(seed);

sprintf(dbg_filename, "debug/rank_%02d.txt", rank);
dbg = fopen(dbg_filename, "w");
data.dbg = dbg;

#ifdef CALC_REORTH
if (rank == 0) {
FILE *fid;
sprintf(data.real_filename, "real.txt");
sprintf(data.approx_filename, "approx.txt");
sprintf(data.flag_filename, "flag.txt");
fid = fopen(data.real_filename, "w");
fclose(fid);
fid = fopen(data.approx_filename, "w");
fclose(fid);
fid = fopen(data.flag_filename, "w");
fclose(fid);
}
#endif

SPMAT L;
T1 = MPI_Wtime();
// fprintf(dbg, "Begin initializing\n"); fflush(dbg);
load_particles(argv[1], &data);
share_Np_local(&data);
init_spmat(&L, &data);
// fprintf(dbg, "End initializing\n"); fflush(dbg);
MPI_Barrier(MPI_COMM_WORLD);

mpi_particle = make_mpi_particle();
// fprintf(dbg, "Made mpi_particle\n"); fflush(dbg);
gen_matrix_intra(&L, &data);
// fprintf(dbg, "Made intra matrix\n"); fflush(dbg);
gen_matrix_inter(&L, &data, mpi_particle);
// fprintf(dbg, "Made inter matrix\n"); fflush(dbg);
```

```
T2 = MPI_Wtime();
t_initialize = T2 - T1;

int count = 0;
int N[2];
double *alpha, *beta, **Q;
int ilam = 0;
double past_lam[4] = {0.0, 0.0, 0.0, 0.0};
int past_steps[4];
double minritz;
int lam_ind, minritz_ind;
double rho;
double error;
int more;
int single_iteration;
LANCZOS lancz;
lancz.start = 0;

// Number of initial steps to take
if (argc > 2) {
single_iteration = 1;
lancz.steps = atoi(argv[2]);
}
else {
single_iteration = 0;
lancz.steps = 2;
}

error = ETOL+1.0;
while (error > ETOL)
{

// Clear results of tridiagonal solution from last iteration
if (count > 0) {
free(alpha);
free(beta);
free_mat(Q);
}

if (rank == 0)
printf("steps = %d\n", lancz.steps);

//-----------------------------------------------------------------------
// Run Lanczos algorithm
//-----------------------------------------------------------------------
T1 = MPI_Wtime();
lanczos(L, &data, &lancz);
T2 = MPI_Wtime();
t_lanczos += T2 - T1;

 #ifdef DEBUG
```

```
fprintf(dbg, "alpha = \n");
for (i = 0; i < lancz.steps; i++)
fprintf(dbg, "%2d %f\n", i, lancz.alpha[i]);
fprintf(dbg, "beta = \n");
for (i = 0; i < lancz.steps-1; i++)
fprintf(dbg, "%2d %f\n", i, lancz.beta[i]);
fflush(dbg);
 #endif


//-------------------------------------------------------------------
// Solve eigenvals and eigenvecs for tridiagonal system
//-------------------------------------------------------------------
alpha = (double *)malloc(lancz.steps * sizeof(double));
beta = (double *)malloc(lancz.steps * sizeof(double));
Q = alloc_mat(lancz.steps, data.Np_local[rank]);

// Copy 'alpha,' 'beta,' and 'Q' from Lanczos results
memcpy(alpha, lancz.alpha, lancz.steps * sizeof(double));
memcpy(beta, lancz.beta, lancz.steps * sizeof(double));
memcpy(Q[0], lancz.Q[0], lancz.steps * data.Np_local[rank] * sizeof(double));

// Dimensions of 'Q'
N[0] = data.Np_local[rank];
N[1] = lancz.steps;

// Solve for eigenvalues and eigenvectors
T1 = MPI_Wtime();
tridiag(alpha, beta, Q, N);
T2 = MPI_Wtime();
t_qr += T2 - T1;

// Get Fiedler value and which Q index that is
minritz = alpha[0];
minritz_ind = 0;
for (i = 0; i < lancz.steps; i++)
{
if (minritz > alpha[i])
{
minritz = alpha[i];
minritz_ind = i;
}
}
lam_ind = (minritz_ind + 1) % lancz.steps;
past_lam[ilam] = alpha[lam_ind];
for (i = 2; i < lancz.steps; i++)
{
j = (minritz_ind + i) % lancz.steps;
if (alpha[j] < past_lam[ilam])
{
past_lam[ilam] = alpha[j];
```

```
lam_ind = j;
}
}

// Stop checking error if lanczos steps = size of original matrix
if (lancz.steps == data.Np || single_iteration)
break;

// Determine error and how many more steps to take
past_steps[ilam] = lancz.steps;
lancz.start = lancz.steps - 1;
ilam++;
if (ilam > 3) // enough info to evaluate error
{
rho = pow(fabs((past_lam[3] - past_lam[2])/(past_lam[1] - past_lam[0])), 1.0/(past_steps[2] - past_steps[0]));
error = rho/(1.0-rho)*fabs(past_lam[3]-past_lam[2]);
more = ceil(0.8*log(ETOL/error)/log(rho));
if (more < 1)
break;
lancz.steps += more;
lancz.steps = min(lancz.steps, data.Np-1);
ilam = 2;
}
else
{
lancz.steps++;
}

count++;
}

#ifdef DEBUG
fprintf(dbg, "alpha_tridiag = \n");
for (i = 0; i < lancz.steps; i++)
fprintf(dbg, "%2d %f\n", i, alpha[i]);
fprintf(dbg, "beta_tridiag = \n");
for (i = 0; i < lancz.steps-1; i++)
fprintf(dbg, "%2d %f\n", i, beta[i]);
fflush(dbg);
fprintf(dbg, "Q_tridiag = \n");
for (j = 0; j < data.Np_local[rank]; j++) {
for (i = 0; i < lancz.steps; i++) {
fprintf(dbg, "%f  ", Q[i][j]);
}
fprintf(dbg, "\n");
}
fflush(dbg);
#endif

// Save Fiedler vector
save_vector("fiedler_vec.txt", Q[lam_ind], &data);
```

```
if (rank == 0) {
if (single_iteration)
printf("Fiedler value = %.14e\n", past_lam[0]);
else
printf("Fiedler value = %.14e\n", past_lam[3]);
}

T2 = MPI_Wtime();
t_total = T2 - T0;

// Timing results
if (rank == 0) {
printf("Total time: %g\n", t_total);
printf("Initialization time: %g\n", t_initialize);
printf("Lanczos algorithm time: %g\n", t_lanczos);
printf("QR tridiagonal time: %g\n", t_qr);
}

MPI_Barrier(MPI_COMM_WORLD);

clean_lancz(&lancz);
clean_data(&data);
clean_spmat(&L);

free(alpha);
free(beta);
free_mat(Q);

fclose(dbg);

MPI_Finalize();

return 0;
}
```

B.3 matvec.h

```
#ifndef _MATVEC
#define _MATVEC

#include <mpi.h>
#include "structs.h"

extern double thresh_conn;

double **alloc_mat(int m, int n);
void free_mat(double **mat);
double get_weight(double dist);
SPMAT_ENTRY *add_spmat_entry(SPMAT_ENTRY *root, int row, double w, int column);
void gen_matrix_intra(SPMAT *mat, LDATA *data);
```

```
void gen_matrix_inter(SPMAT *mat, LDATA *data, MPI_Datatype mpi_particle);
void init_spmat(SPMAT *mat, LDATA *data);
void clean_spmat(SPMAT *mat);
void matvec(LDATA *data, SPMAT mat, double *lvec, double *prod);

#endif
```

B.4 matvec.cpp

```
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "matvec.h"

double thresh_conn = 3.0;

/*************************************************************************/
/*
 Allocates space for a 2-D double array, where 'm' and 'n' are the dimensions of
 the slow- and fast-varying indices, respectively
 */
/*************************************************************************/
double **alloc_mat(int m, int n) {

int i;
double **mat;

mat = (double **)malloc(m * sizeof(double *));
mat[0] = (double *)malloc(m * n * sizeof(double));
for (i = 1; i < m; i++)
mat[i] = mat[0] + i * n;

return mat;
}


/*************************************************************************/
/*
 Frees matrix generated by alloc_mat()
 */
/*************************************************************************/
void free_mat(double **mat) {

free(mat[0]);
free(mat);
}


/*************************************************************************/
```

```
/*
 */
/*****************************************************************************/
double get_weight(double dist)
{
//  return 0.9*0.5*(1.0-erf(10.0*(dist-1.15))) + 0.1*exp(-3.0*(dist-1.0));
// return 1.0/(dist-1.0);
return pow((dist - 0.9), -5.0);
}


/*****************************************************************************/
/*
 */
/*****************************************************************************/
SPMAT_ENTRY *add_spmat_entry(SPMAT_ENTRY *root, int row, double w, int column)
{
SPMAT_ENTRY *entry;

if (root == NULL)
{
root = (SPMAT_ENTRY *) malloc(sizeof(SPMAT_ENTRY));
entry = root;
}
else
{
entry = root;
while (entry->next != NULL)
entry = entry->next;
entry->next = (SPMAT_ENTRY *) malloc(sizeof(SPMAT_ENTRY));
entry = entry->next;
}

entry->row = row;
entry->weight = w;
entry->column = column;
entry->next = NULL;

return root;
}


/*****************************************************************************/
/*
 */
/*****************************************************************************/
void gen_matrix_intra(SPMAT *mat, LDATA *data)
{
int i, j, k;
double dx, dy, dist, weight;
```

```
int rank = data -> rank;

k = data->vecstart[rank];

for (i = 0; i < data->Np_local[rank]; i++)
{
for (j = i+1; j < data->Np_local[rank]; j++)
{
dx = data->particles[i].x - data->particles[j].x;
dy = data->particles[i].y - data->particles[j].y;
dist = sqrt(dx * dx + dy * dy);
if (dist < thresh_conn)
{
weight = get_weight(dist);
mat->root = add_spmat_entry(mat->root, i, -weight, k+j);
mat->root = add_spmat_entry(mat->root, j, -weight, k+i);
mat->diag[i] += weight;
mat->diag[j] += weight;
}
}
}
}


/*****************************************************************************/
/*
 */
/*****************************************************************************/
void gen_matrix_inter(SPMAT *mat, LDATA *data, MPI_Datatype mpi_particle)
{
int step;
int send, recv;
int i, j, k;
PARTICLE *sendbuf, *recvbuf, *tempbuf;
int sendbufN, recvbufN;
MPI_Status status;
double dx, dy, dist, weight;

int rank = data -> rank;
int nproc = data -> nproc;

sendbuf = (PARTICLE *) malloc(data->Np_local_max * sizeof(PARTICLE));
recvbuf = (PARTICLE *) malloc(data->Np_local_max * sizeof(PARTICLE));

send = (rank + 1) % nproc;
recv = (rank - 1 + nproc) % nproc;

memcpy(sendbuf, data->particles, data->Np_local[rank] * sizeof(PARTICLE));
sendbufN = data->Np_local[rank];

//carousel algorithm
```

```
for (step = 1; step < nproc; step++)
{
MPI_Sendrecv(&sendbufN, 1, MPI_INT, send, 0, &recvbufN, 1, MPI_INT,
             recv, 0, MPI_COMM_WORLD, &status);
MPI_Sendrecv(sendbuf, sendbufN, mpi_particle, send, 1,
             recvbuf, recvbufN, mpi_particle, recv, 1, MPI_COMM_WORLD, &status);

k = data->vecstart[(rank-step+nproc) % nproc];

for (i = 0; i < data->Np_local[rank]; i++)
{
for (j = 0; j < recvbufN; j++)
{
dx = data->particles[i].x - recvbuf[j].x;
dy = data->particles[i].y - recvbuf[j].y;
dist = sqrt(dx*dx + dy*dy);
if (dist < thresh_conn)
{
weight = get_weight(dist);
mat->root = add_spmat_entry(mat->root, i, -weight, k+j);
mat->diag[i] += weight;
}
}
}

tempbuf = sendbuf;
sendbuf = recvbuf;
recvbuf = tempbuf;
sendbufN = recvbufN;
}

free(sendbuf);
free(recvbuf);
}


/*****************************************************************************/
/*
 */
/*****************************************************************************/
void init_spmat(SPMAT *mat, LDATA *data)
{
int rank = data -> rank;
mat->diag = (double *) malloc(data->Np_local[rank] * sizeof(double));
memset(mat->diag, 0, data->Np_local[rank] * sizeof(double));
mat->root = NULL;
}


/*****************************************************************************/
/*
```

```
 */
/*****************************************************************************/
void clean_spmat(SPMAT *mat)
{
SPMAT_ENTRY *one, *two;

free(mat->diag);

one = mat->root;
while (one->next != NULL)
{
two = one->next;
free(one);
one = two;
}
}


/*****************************************************************************/
/*
 */
/*****************************************************************************/
void matvec(LDATA *data, SPMAT mat, double *lvec, double *prod)
{
SPMAT_ENTRY *node;
int i, k;
double *vec;

int rank = data -> rank;
FILE *dbg = data -> dbg;

k = data->vecstart[rank];
//  fprintf(dbg, "vecstart = %d\n", k); fflush(dbg);

//  fprintf(dbg, "vec = \n");
//  for (i = 0; i < data->Np; i++)
//  fprintf(dbg, "%2d %f\n", i, vec[i]);
//  fflush(dbg);

vec = (double *) malloc(data->Np * sizeof(double));
// memcpy(&vec[k], lvec, data->Np_local[rank] * sizeof(double));
MPI_Allgatherv(lvec, data->Np_local[rank], MPI_DOUBLE,
               vec, data->Np_local, data->vecstart, MPI_DOUBLE, MPI_COMM_WORLD);

//  fprintf(dbg, "vec = \n");
//  for (i = 0; i < data->Np; i++)
//  fprintf(dbg, "%2d %f\n", i, vec[i]);
//  fflush(dbg);

memset(prod, 0, data->Np_local[rank]*sizeof(double));
```

```
//  fprintf(dbg, "my part of prod = (before matvec op)\n");
//  for (i = 0; i < data->Np_local[rank]; i++)
//  fprintf(dbg, "%2d %f\n", data->particles[i].id, prod[i]);
//  fflush(dbg);

//  fprintf(dbg, "diagonal entries = \n");
for (i = 0; i < data->Np_local[rank]; i++)
{
prod[i] += mat.diag[i]*vec[k+i];
//  fprintf(dbg, "%2d %f\n", data->particles[i].id, mat.diag[i]);
}
//  fflush(dbg);

//  fprintf(dbg, "off-diagonal entries = \n");
node = mat.root;
while (node != NULL)
{
//  fprintf(dbg, "row = %2d, column = %2d, weight = %f\n", k+node->row, node->column, node->weight);
prod[node->row] += node->weight * vec[node->column];
node = node->next;
}
//  fflush(dbg);

free(vec);
}
```

B.5 lanczos.h

```
#ifndef _LANCZOS
#define _LANCZOS

double dot(double *x, double *y, int lN);
void lanczos(SPMAT mat, LDATA *data, LANCZOS *lancz);

#endif
```

B.6 lanczos.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include <string.h>

#include "structs.h"
#include "lanczos.h"
#include "matvec.h"

#define PI 3.14159265359
```

```
#define RANK_TEST 1

/*****************************************************************************/
/*
 Determines value of machine precision within a factor of two of its actual
 value
 */
/*****************************************************************************/
double calc_eps()
{
double eps = 1.0;

do {
// printf( "%G\t%.20f\n", eps, (1.0 + eps) );
eps /= 2.0f;
// If next epsilon yields 1, then break, because current
// epsilon is the machine epsilon.
} while ((double)(1.0 + (eps / 2.0)) != 1.0);

return eps;
}




/*****************************************************************************/
/*
 Generates a random number from a normal distribution using the Box-Muller
 method.
     - mu:    mean
     - sigma: standard deviation
 */
/*****************************************************************************/
double randn(double mu, double sigma)
{
double X, U, V;

U = ((double) rand()) / RAND_MAX;
V = ((double) rand()) / RAND_MAX;

X = sqrt(-2.0 * log(fabs(U))) * cos(2.0 * PI * V);

return sigma * X + mu;
}




/*****************************************************************************/
/*
 Computes dot product of two vectors 'x' and 'y' which are split among all the
```

```
 processors.  The vector length on the local processor is 'lN'
 */
/*****************************************************************************/
double dot(double *x, double *y, int lN)
{
int i;
double lans = 0.0;
double ans;

for (i = 0; i < lN; i++)
lans += x[i] * y[i];

MPI_Allreduce(&lans, &ans, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

return ans;
}




/*****************************************************************************/
/*
 Executes Lanczos algorithm to calculate eigenvalues and eigenvectors of a
 sparse matrix defined by 'mat'.

 This method produces a (symmetric) tridiagonal matrix (from a symmetric
 Laplacian matrix):
     - alpha: Diagonal entries, vector of length 'steps'
     - beta: Off-diagonal entries, vector of length 'steps-1'

 The pseudo-eigenvectors are defined as 'Q', having dimensions 'Np' by 'steps':
     - Np: Number of particles, i.e. size of Laplacian matrix
     - steps: Number of Lanczos iterations to perform, yielding that many
              eigenvectors
 On the local processor, 'Q' has dimensions 'Np_local' by 'steps'.  The 'steps'
 dimension is the slowly-varying dimension.

 We are using the partial reorthogonalization method (PRO) of Horst D. Simon
 ['The Lanczos algorithm with partial reorthogonalization', Mathematics of
 Computation, 1984], which estimates the degree of deorthogonalization (which we
 keep track of in 'w') and then reorthogonalizes the new eigenvector against
 only those that have a high degree of deorthogonalization.
 */
/*****************************************************************************/
void lanczos(SPMAT mat, LDATA *data, LANCZOS *lancz)
{
double **Q;
double *alpha, *beta;
double *v;
int i, j, k;
double temp;
```

```
double **w, *w0, *temp_ptr;
int firststep, *flag_reorth;
int lo, up;
double eps, sqrt_eps, psi, theta, eta;

int rank = data -> rank;
FILE *dbg = data -> dbg;

int N_local = data -> Np_local[rank];
int N_global = data -> Np;

int start = lancz -> start;
int steps = lancz -> steps;

#ifdef CALC_REORTH
FILE *fid_real, *fid_approx, *fid_flag;
if (rank == 0) {
fid_real = fopen(data -> real_filename, "w");
fid_approx = fopen(data -> approx_filename, "a");
fid_flag = fopen(data -> flag_filename, "a");
}
#endif

//------------------------------------------------------------------------
// Allocate storage
//------------------------------------------------------------------------
// Matrix of eigenvectors: Q[j][i] = element 'i' of eigenvector 'j'
Q = alloc_mat(steps, N_local);

// Vector of main diagonal elements produced by Lanczos
alpha = (double *) malloc(steps * sizeof(double));

// Vector of off-diagonal elements produced by Lanczos
beta = (double *) malloc(steps * sizeof(double));

// Temporary storage vector for Lanczos algorithm
v = (double *) malloc(N_local * sizeof(double));

// Matrix of approximated inner products.  We change the pointer to 'w' to
// simulate how it changes with 'j'.  Thus, in the Lanczos iterations, we
// can use 'w' as if we were storing the whole matrix, even though we are
// only storing three steps at a time.
w = alloc_mat(3, steps);
w0 = w[0];
w += 1 - start;

// Flag of which vectors to reorthogonalize
flag_reorth = (int *) malloc(steps * sizeof(int));
for (i = 0; i < steps; i++)
flag_reorth[i] = 0;
```

```
//------------------------------------------------------------------------
// Initialize Lanczos
//------------------------------------------------------------------------
// Machine precision
eps = calc_eps();
// Threshold for starting reorthogonalization, p. 126 of reference
sqrt_eps = sqrt(eps);
// Threshold for range of reorthogonalization, p. 129 of reference
// eta = pow(eps, 0.75);
eta = pow(eps, 0.8);
// eta = 0;

// Start Lancozs method
if (start == 0) {
// Random initial vector
  srand(rank);
for (i = 0; i < N_local; i++)
Q[0][i] = ((double) rand())/RAND_MAX - 0.5;

// Reseed random number generator, in case N_local varies among
// processors
int seed = rand();
MPI_Bcast(&seed, 1, MPI_INT, 0, MPI_COMM_WORLD);
srand(seed);

 #ifdef DEBUG
fprintf(dbg, "starting vec = \n");
for (i = 0; i < N_local; i++)
fprintf(dbg, "%.14f\n", Q[0][i]);
fflush(dbg);
 #endif

temp = sqrt(dot(Q[0], Q[0], N_local));
for (i = 0; i < N_local; i++)
Q[0][i] /= temp;

w[0][0] = 1.0;

 #ifdef DEBUG
fprintf(dbg, "starting vec = \n");
for (i = 0; i < N_local; i++)
fprintf(dbg, "%f\n", Q[0][i]);
fprintf(dbg, "w = \n%g\n", w[0][0]);
fflush(dbg);
 #endif

firststep = 1;
}
// Continue Lanczos for more iterations
```

```
else {
// Copy Q matrix into new, larger storage
memcpy(Q[0], lancz->Q[0], (start + 1) * N_local * sizeof(double));
free_mat(lancz->Q);

// Copy alpha vector into new, larger storage
memcpy(alpha, lancz->alpha, (start + 1) * sizeof(double));
free(lancz->alpha);

// Copy beta vector into new, larger storage
memcpy(beta, lancz->beta, (start + 1) * sizeof(double));
free(lancz->beta);

// Copy w matrix into new, larger storage
for (j = start - 1; j <= start; j++) {
for (i = 0; i <= start; i++) {
w[j][i] = lancz->w[j][i];
}
}
free(lancz -> w0);
free(&(lancz -> w[start - 1]));

// Copy flag_reorth into new, larger storage
memcpy(flag_reorth, lancz->flag_reorth, (start + 1) * sizeof(int));
free(lancz->flag_reorth);

// Copy state of firststep
firststep = lancz -> firststep;
}

//------------------------------------------------------------------------
// Lanczos algorithm
//------------------------------------------------------------------------
for (j = start; j < steps-1; j++) {

matvec(data, mat, Q[j], v);

alpha[j] = dot(v, Q[j], N_local);

for (i = 0; i < N_local; i++)
v[i] -= alpha[j] * Q[j][i];

if (j > 0) {
for (i = 0; i < N_local; i++)
v[i] -= beta[j-1] * Q[j-1][i];
}

beta[j] = sqrt(dot(v, v, N_local));

// Estimate loss of orthogonality, eq. (5.1) p. 121 of reference paper
// 'psi' from eq. (5.3) p. 123 of reference paper
```

```
psi = eps * N_global * beta[0] / beta[j] * randn(0, 0.6);
// psi = eps * N_global * randn(0, 1.0);
w[j+1][j+1] = 1.0;
w[j+1][j] = psi;
for (k = 0; k < j; k++) {
// 'theta' from eq. (5.2) p. 123 of reference paper
theta = eps * (beta[k] + beta[j]) * randn(0, 0.3);
// theta = eps * randn(0, KAPPA);
w[j+1][k] = 1.0 / beta[j] * (beta[k] * w[j][k+1] + (alpha[k] - alpha[j]) * w[j][k]
- beta[j-1] * w[j-1][k]) + theta;
if (k > 0)
w[j+1][k] += beta[k-1] / beta[j] * w[j][k-1];
}

// Flag which vectors need to be reorthogonalized
if (firststep) {

// Reset flags
for (k = 0; k <= j; k++)
flag_reorth[k] = 0;

up = 0;
k = 0;
while (k <= j) {

// Find where estimated inner product exceeds threshold
if (fabs(w[j+1][k]) > sqrt_eps) {

firststep = 0;

// Other vectors to be reorthogonalized described on p. 128
// of reference

// Search for lower bound of lesser threshold
lo = k;
while (lo >= up && fabs(w[j+1][lo]) > eta) {
flag_reorth[lo] = 1;
lo--;
}

// Search for upper bound of lesser threshold
up = k + 1;
while (up <= j && fabs(w[j+1][up]) > eta) {
flag_reorth[up] = 1;
up++;
}

k = up;
}

k++;
```

```
}
}
else {
firststep = 1;
}

// Reorthogonalize
for (k = 0; k <= j; k++) {
if (flag_reorth[k]) {
temp = dot(v, Q[k], N_local);
for (i = 0; i < N_local; i++) {
v[i] -= Q[k][i] * temp;
}
w[j+1][k] = eps * randn(0, 1.5);
}
}

beta[j] = sqrt(dot(v, v, N_local));

// Store eigenvector
for (i = 0; i < N_local; i++)
Q[j+1][i] = v[i] / beta[j];

#ifdef CALC_REORTH
if (rank == 0) {
for (i = 0; i <= j; i++) {
fprintf(fid_approx, "%g  ", w[j+1][i]);
fprintf(fid_flag, "%d ", flag_reorth[i]);
}
fprintf(fid_approx, "\n");
fflush(fid_approx);
fprintf(fid_flag, "\n");
fflush(fid_flag);
}
#endif

// Update storage of 'w'
temp_ptr = w[j-1];
w[j-1] = w[j];
w[j] = w[j+1];
w[j+1] = temp_ptr;
w--;
}

// Calculate last alpha
matvec(data, mat, Q[steps-1], v);
alpha[steps-1] = dot(v, Q[steps-1], N_local);

#ifdef CALC_REORTH
// Calculate actual orthogonalities
for (j = 0; j < steps; j++) {
```

```
for (k = 0; k < j; k++) {
temp = dot(Q[j], Q[k], N_local);
if (rank == 0)
fprintf(fid_real, "%g  ", temp);
}
if (rank == 0) {
fprintf(fid_real, "\n");
fflush(fid_real);
}
}
if (rank == 0) {
fclose(fid_real);
fclose(fid_approx);
fclose(fid_flag);
}
#endif

//--------------------------------------------------------------------------
// Save return values
//--------------------------------------------------------------------------
lancz -> Q = Q;
lancz -> alpha = alpha;
lancz -> beta = beta;

lancz -> w = w;
lancz -> w0 = w0;
lancz -> firststep = firststep;
lancz -> flag_reorth = flag_reorth;

free(v);
}
```

B.7 qr_tridiag.h

```
#ifndef _QR_TRIDIAG
#define _QR_TRIDIAG

void tridiag(double *d, double *e, double **z, int *N);

#endif
```

B.8 qr_tridiag.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "qr_tridiag.h"

/*****************************************************************************/
/*
 Calculates eigenvalues of symmetric 2x2 matrix
     - d: Diagonal entries, vector of length 2
     - e: Off-diagonal entry
     - lambda: vector into which the two eigenvalues will be deposited
 */
/*****************************************************************************/
void eig(double *lambda, double *d, double e) {

// Trace
double T = d[0] + d[1];

// Determinant
double D = d[0] * d[1] - e * e;

// Eigenvalues
lambda[0] = 0.5 * T + sqrt(0.25 * T * T - D);
lambda[1] = 0.5 * T - sqrt(0.25 * T * T - D);
}


/*****************************************************************************/
/*
 Computes sqrt(a^2 + b^2) without numerical overflow
 */
/*****************************************************************************/
double pythag(double a, double b) {

static double dsqrarg;
#define DSQR(a) ((dsqrarg=(a)) == 0.0 ? 0.0 : dsqrarg*dsqrarg)

double absa, absb;
absa = fabs(a);
absb = fabs(b);

if (absa > absb)
return absa * sqrt(1.0 + DSQR(absb / absa));
else
return (absb == 0.0 ? 0.0 : absb * sqrt(1.0 + DSQR(absa / absb)));

#undef DSQR
}
```

```
/*****************************************************************************/
/*
 Calculates eigenvalues of symmetric tridiagonal matrix

     - d: Main diagonal of matrix, vector of length 'n'

     - e: Off-diagonal of matrix, vector of length 'n-1,' but memory should be
          allocated as if length 'n'

     - v: Matrix of size 'm-by-n' which will be operated on to yield the
          eigenvectors.  Pass the identity matrix to obtain the eigenvectors of
          the tridiagonal matrix.  Pass another matrix if this is the last step
          in finding the eigenvectors of a matrix (e.g. as part of Lanczos
          algorithm).

     - N: Array describing matrix dimensions, m = N[0], n = N[1]

 */
/*****************************************************************************/
void tridiag(double *d, double *e, double **v, int *N) {

// Is matrix 'v' row-major or column-major form?
#undef ROW_MAJOR

// Indices to mark position within block, i = Is:Ie-1 would access every
// diagonal element in the block
int i, j, Is, Ie;

// Matrix 'v' is m-by-n
int m = N[0];
int n = N[1];

// Flag to indicate that the matrix has not been reduced to 1x1 and 2x2 blocks
int flag;

// Number of iterations across entire matrix
int iter;

// Eigenvalues of 2x2 matrix, shift variable
double lambda[2], mu;

// Rotation matrix coefficients:
// inverse tangent, sine, cosine, sin^2, cos^2, sin * cos
double it, s, c, s2, c2, sc;

// Diagonal and off-diagonal elements
double d0, d1, dd;

// Off-diagonal elements
double e_1, e0, e1;
```

```
// Bulge (off-off diagonal) element
double bulge;

// Eigenvector temp storage
double v0;

Is = 0;
iter = 0;
flag = 1;

while (flag || Is > 0) {

if (Is == 0)
flag = 0;

//---------------------------------------------------------------------
// Find end of block
//---------------------------------------------------------------------
for (i = Is; i < n - 1; i++) {
dd = fabs(d[i]) + fabs(d[i+1]);
if (dd + e[i] == dd) {
break;
}
}
Ie = i + 1;

//---------------------------------------------------------------------
// Block size is greater than 1
//---------------------------------------------------------------------
if (Ie - Is > 1) {
flag = 1;

//---------------------------------------------------------------
// Calculate shift
//---------------------------------------------------------------
eig(lambda, &d[Ie - 2], e[Ie - 2]);
if (fabs(lambda[0] - d[Ie - 1]) < fabs(lambda[1] - d[Ie - 1]))
mu = lambda[0];
else
mu = lambda[1];

//---------------------------------------------------------------
// Create bulge (can assume e[Is] != 0)
//---------------------------------------------------------------
d0 = d[Is];
e0 = e[Is];
d1 = d[Is + 1];
e1 = e[Is + 1];

// Rotation coefficients
```

```
if (e0 == 0) {
fprintf(stderr, "Error: e0 == 0.\n");
exit(0);
}
it = (d0 - mu) / e0;
s  = 1 / sqrt(1 + it * it);
c  = it * s;
s2 = s * s;
c2 = c * c;
sc = s * c;

// Rotate matrix
d[Is]   = c2 * d0 + s2 * d1 + 2 * sc * e0;
d[Is+1] = s2 * d0 + c2 * d1 - 2 * sc * e0;
e[Is]   = sc * (d1 - d0) + (c2 - s2) * e0;
e[Is+1] = c * e1;
bulge   = s * e1;

for (j = 0; j < m; j++) {
#ifdef ROW_MAJOR
v0 = v[j][Is];
v[j][Is]   = c * v0 + s * v[j][Is+1];
v[j][Is+1] = -s * v0 + c * v[j][Is+1];
#else
v0 = v[Is][j];
v[Is][j]   = c * v0 + s * v[Is+1][j];
v[Is+1][j] = -s * v0 + c * v[Is+1][j];
#endif

}

//---------------------------------------------------------------------
// "Squeeze" bulge down diagonal
//---------------------------------------------------------------------
for (i = Is + 1; i < Ie - 1; i++) {

e_1 = e[i - 1];
d0  = d[i];
e0  = e[i];
d1  = d[i + 1];
e1  = e[i + 1];

// Rotation coefficients
if (bulge == 0) {
fprintf(stderr, "Error: bulge == 0.\n");
exit(0);
}
it = e_1 / bulge;
s  = 1 / sqrt(1 + it * it);
c  = it * s;
s2 = s * s;
```

```
c2 = c * c;
sc = s * c;

// Rotate matrix
d[i]   = c2 * d0 + s2 * d1 + 2 * sc * e0;
d[i+1] = s2 * d0 + c2 * d1 - 2 * sc * e0;
e[i-1] = c * e_1 + s * bulge;
e[i]   = sc * (d1 - d0) + (c2 - s2) * e0;
e[i+1] = c * e1;
bulge  = s * e1;

for (j = 0; j < m; j++) {
#ifdef ROW_MAJOR
v0 = v[j][i];
v[j][i]   = c * v0 + s * v[j][i+1];
v[j][i+1] = -s * v0 + c * v[j][i+1];
#else
v0 = v[i][j];
v[i][j]   = c * v0 + s * v[i+1][j];
v[i+1][j] = -s * v0 + c * v[i+1][j];
#endif
}
}

}

//---------------------------------------------------------------------
// Reset of beginning of matrix
//---------------------------------------------------------------------
if (Ie == n) {
Is = 0;
iter++;
}
else
Is = Ie;
}

//  printf("Iterations: %d\n", iter);
}
```

References

[1] U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1st edition, 1998.

[2] C. D. Meyer, editor. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[3] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2nd edition, 1992.

[4] H. D. Simon. The lanczos algorithm with partial reorthogonalization. *Mathematics of Computation*, 42(165):pp. 115–142, 1984.