1 Introduction

At the moment, the first stage of the project, namely, laying out the theory and prototyping the main algorithms in MATLAB is complete. This report provides the details of the numerical algorithms used in the project. The MATLAB prototypes for these algorithms are listed in the Appendix.

2 Theory

The central subject of study in this project is Lanczos iteration (Listing 1 in the Appendix). This algorithm allows to obtain some eigenvalues and eigenvectors of a square symmetric matrix $A$. Starting from a randomly chosen vector $q_0$, it builds a basis of the Krylov subspace $\text{span}(Q_k) = K(A, q_0)$ and, in parallel, builds an orthogonal transformation $T_k = Q_k^T A Q_k$ of $A$ into a tridiagonal matrix $T_k$. Eigenvalues of $T_k$ are equal to some$^1$ of the eigenvalues of $A$, and the $j$’th eigenvector $V_j^A$ of $A$ can be obtained from the $j$’th eigenvector $V_i$ of $T_k$ as $V_j^A = Q_i^T V_j$.

Lanczos iteration as described above does not work well in practice. Due to accumulation of rounding errors in the process of building a basis $Q_k$ of the Krylov subspace, its vectors gradually lose orthogonality. As a result, we cannot get an orthogonal transformation of $A$ and find its eigenvectors to the desired precision. This situation is alleviated by periodically reorthogonalizing the already computed columns of $Q_k$. Reorthogonalization against all the previously computed

$^1$In the original Lanczos iteration without shifting and inverting, the first eigenvalues (and eigenvectors) to converge are the ones with the largest magnitude.
vectors is too costly. Thus, the best strategy is to use \textit{selective orthogonalization}. The theory says that it is enough to reorthogonalize only against the already converged Ritz vectors.

Selective orthogonalization imposes the requirement to find eigenvalues and eigenvectors of a symmetric tridiagonal matrix $T_k$. The standard way to do it is to use \textit{QR iteration} (Listing 2 in the Appendix). The central idea of QR iteration is to iteratively QR-decompose a matrix ($T_i$) and build its decomposition into “an almost Schur form" $T_i = V\Lambda V^T$, where $V$ is unitary and $\Lambda$ is \textit{almost} upper-triangular. This iterative process of QR-decomposition of $T_i$ converges very slowly if $T_i$’s eigenvalues are not well-separated. To fight this slow convergence, instead of decomposing $T_i$ (or the result of its decomposition on the previous steps), a shifted matrix ($T_i - \mu I$) is decomposed, where $\mu$ is a \textit{shift}. A wisely chosen shift $\mu$ increases the separation of eigenvalues and increases the convergence. For example, a Wilkinson’s shift delivers a cubic rate of convergence.

Finally, in order to perform QR iteration, we should be able to perform \textit{QR decomposition} (Listing 3 in the Appendix). The main idea is to find a series of transformations of a matrix to nullify its subdiagonal elements. The product of these transformations will constitute matrix $Q$, while the transformed upper-triangular matrix will represent $R$. If we deal with an arbitrary matrix, the first step is usually to use Householder reflections, which can nullify several matrix elements per one reflection. The result of multiple reflections is a tridiagonalized matrix. However, we are starting with an already tridiagonal matrix $T_i$ and only need to nullify the elements located at its bottom off-diagonal.

The common way to nullify a few elements in a matrix is to use \textit{Givens rotations} $G(i, k, \theta)$ – a rotation by the angle $\theta$ in the plane containing axes $(i, k)$ (Listings 4 and 5 in the Appendix). The angle $\theta$ can be chosen so that the rotation applied to a vector (or a matrix) nullifies its particular element. Thus, having applied a Givens rotation to each subdiagonal element of a tridiagonal matrix, we are getting an upper triangular matrix, $R$. The product of used Givens rotations becomes $Q$. There exists a more sophisticated way to perform QR iteration – implicit QR with Wilkinson’s shift $\mu$, but it will be implemented only if time permits.

\section{Future Work}

Having prepared the theoretical base and the MATLAB prototypes of each required algorithm, the roadmap\footnote{Such milestones as “completing the project report” or “presenting a demo in class” are deliberately omitted from the roadmap.} is to

- Implement the algorithms serially in C++;
- Parallelize the serial code using Cilk++;
- Test for correctness and performance.
Having completed this second stage, the third stage is to

- Parallelize the code for distributed-memory systems either directly via MPI or indirectly via CombBLAS;
- Optionally, implement an efficient QR iteration and parallelize it;
- Integrate the obtained eigensolver into a real-world data-analysis application for the purpose of testing correctness and performance.

While completing the third stage is desirable, the project will be considered successful even without this last stage being completed.

4 Appendix

4.1 MATLAB Prototype for Lanczos Iteration with Selective Reorthogonalization

```matlab
% This code is partly based on James Demmel’s MATLAB example of Lanczos process.

clear alpha beta;

n = max(size(A));
q = q / norm(q);
Q = zeros(n, m);
Q(:,1) = q;
alpha = zeros(1, m);
beta = zeros(1, m-1);
for i = 1 : m
    z = A * q;
    alpha(i) = q’ * z;
    z = z - alpha(i) * q;
    if(i > 1)
        z = z - beta(i-1) * Q(:,i-1);
    end
    beta(i) = norm(z);
    q = z / beta(i);
    if(i > 1)
        Ti = ...
            diag(alpha(1:i)) + ...
            diag(beta(1:i-1), 1) + ...
            diag(beta(1:i-1), -1);
    else
        Ti = alpha(1);
    end
[V, D] = eigenQR(Ti);
[D, newIndices] = sort(diag(D), 'descend');
V = V(:, newIndices);
errorBounds = abs(beta(i) * V(i,:))’;
```

3
normTi = max(abs(D));
top = sqrt(eps) * normTi;
indsOfConverged = find(errorBounds <= top);

if(~isempty(indsOfConverged))
    for j = indsOfConverged
        ritzVec = Q(:,1:i) * V(:,j);
        z = z - (ritzVec' * z) * ritzVec;
        beta(i) = norm(z);
        q = z / beta(i);
    end
    if i < m
        Q(:,i+1) = q;
    end
end

disp('Indices of converged eigenvalues:');
disp(indsOfConverged);

evals = D(indsOfConverged);
disp('Converged eigenvalues:');
disp(evals);

evecs = Q * V;
evecs = evecs(:,indsOfConverged);
evecs(abs(evecs) < 1.0e-10) = 0;
disp('Eigenveectors of converged eigenvalues are in "evecs"');

Listing 1: Lanczos iteration

4.2 MATLAB Prototypes for QR Iteration

function [evecs, evals] = eigenQR(tridiagA)
n = size(tridiagA, 1);
Qf = eye(n);
A = tridiagA;
% TODO: integrate the proper termination criterion
for i = 1:200
    [Q, R] = tridiagQR(A);
    Qf = Qf * Q;
    A = R * Q;
end
% Now A is transformed and Qf * A * Qf' is almost a
% Schur decomposition of the original matrix A, i.e.
% Qf is unitary and A is upper-triangular.
evals = diag(diag(A));
evecs = Qf;
end

Listing 2: QR iteration for tridiagonal symmetric matrices
function \([Q, R] = \text{tridiagQR}(\text{tridiagA})\)
\[
\begin{align*}
n &= \text{size}(\text{tridiagA}, 1); \\
m &= \text{size}(\text{tridiagA}, 2); \\
\text{if}(n \neq m) \\
&\quad \text{error('Matrix is not tridiagonal')}; \\
\text{end} \\
Q &= \text{eye}(n); \\
R &= \text{tridiagA}; \\
\text{for } i = 1 : n - 1 \\
&\quad [G, R] = \text{givensApply}(R, i + 1, i); \\
&\quad Q = Q \ast G';; \\
\text{end} \\
\end{align*}
\]

Listing 3: QR decomposition of a tridiagonal symmetric matrix

4.3 MATLAB Prototypes for Givens Rotation

function \([G, A] = \text{givensApply}(A, i, j)\)
\[
\begin{align*}
A &= A; \\
\text{sz} &= \text{size}(A); \\
\text{if}(\text{sz}(1) \neq \text{sz}(2)) \\
&\quad \text{error('Matrix is not square')}; \\
\text{end} \\
n &= \text{sz}(1); \\
\text{if}(i \leq 1 || i > n || j \leq 0 || j > n) \\
&\quad \text{error('Invalid index values.');} \\
\text{end} \\
a &= A(i - 1, j); \\
b &= A(i, j); \\
[c, s] &= \text{givensCS}(a, b); \\
\text{for } k = 1 : n \\
&\quad \text{tau1} = A(j, k); \\
&\quad \text{tau2} = A(i, k); \\
&\quad A(j, k) = c \ast \text{tau1} - s \ast \text{tau2}; \\
&\quad A(i, k) = s \ast \text{tau1} + c \ast \text{tau2}; \\
\end{align*}
\]

% Should not be stored explicitly in fast implementation
\[
\begin{align*}
G &= \text{eye}(n); \\
G(i, i) &= c; \\
G(i, j) &= s; \\
G(j, i) &= -s; \\
G(j, j) &= c; \\
\end{align*}
\]

Listing 4: Single Givens rotation

function \([c, s] = \text{givensCS}(a, b)\)
\[
\begin{align*}
\text{if } (b == 0) \\
\end{align*}
\]
\begin{verbatim}
c = 1;
s = 0;
else
  if(abs(b) > abs(a))
    r = -a / b;
    s = 1 / sqrt(1 + r * r);
    c = s * r;
  else
    r = -b / a;
    c = 1 / sqrt(1 + r * r);
    s = c * r;
end
end
\end{verbatim}

Listing 5: Computation of the parameters of a Givens rotation