Block Lanczos Tridiagonalization of Complex Symmetric Matrices

By:
Guohong Liu

Computer Science 4ZP6 Project

Supervised by Dr. Sanzheng Qiao
Professor, Department of Computing and Software

Department of Computing and Software
McMaster University
Hamilton, Ontario L8S 4K1
2005

Abstract

The project is involved with the numerical analysis scheme, the tridiagonalization of complex symmetric matrices by using block Lanczos algorithm.

The project is required to implement a numerical software using C language to reduce a complex symmetric matrix to the complex symmetric and tridiagonal form. The software is given any complex symmetric matrix A of order n. It computes a unitary $Q \in C^{n\times n}$ and the diagonals of the complex symmetric and tridiagonal matrix $T \in C^{n\times n}$ such that

$$Q^{\mathrm{H}} A \bar{Q} = T$$

This project group is made up of two students. Guohong Liu is responsible for the high level design, implementation and verification on Windows system and these corresponding documentations. Ivan Mak is responsible for the requirement part of the final report, the test of the implementation on UNIX system and the testing report.

This project is supervised by Professor Qiao. Guohong Liu did the summer research on block Lanczos algorithm under the seasoned guidance of Professor Qiao. The research and project show Guohong Liu the wonderful world of numerical analysis. We would like to thank Professor Qiao for his help throughout the project. The project couldn't be successfully completed without the gaidance of Professor Qiao.

Contents

Chapter 1

Algorithm

We would like to present block Lanczos algorithm in details. As we know, Lanczos method suffers from the loss of orthogonality of the computed Q in the presence of roudning error. Orthogonalization is necessary for practical Lanczos method. In fact orthogonalization is the critical component of the block Lanczos algorithm. Two orthogonalization detection schemes are discussed in S.Qiao[12]. In this chapter we present only one of them, the componentwise scheme. In Section2.2, we will present the reason for the selection of componentwise scheme.

The block Lanczos tridiagonalization algorithm consists of two stages: block tridiagonalization and tridiagonalization. A complex symmetric matrix is first reduced to complex symmetric and block tridiagonal form. The second stage reduces the block tridiagonal complex symmetric matrix to complex symmetric tridiagonal. We describe the two stages of block Lanczos tridiagonalization algorithm respectively.

1.1 Block Tridiagonalization

Let A be an n-by-n complex symmetric matrix, and assume $n = k \times b$. Then there exists the decomposition

$$Q^H A \bar{Q} = J = \begin{bmatrix} M_1 & B_1^T & & \dots & 0 \\ B_1 & M_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & B_{k-1}^T \\ 0 & \dots & & B_{k-1} & M_k \end{bmatrix} \tag{1.1}$$

where

$$Q = [Q_1, Q_2, \dots, Q_k] \qquad Q_i \in C^{n \times b}$$

is orthonormal, $M_i \in C^{b\times b}$ are symmetric, and $B_i \in C^{b\times b}$ upper triangular. Rewriting (1.1) as

$$A\bar{Q} = QJ \tag{1.2}$$

and comparing the jth block columns on both sides of (1.2), we have

$$A\bar{Q}_j = Q_{j-1}B_{j-1}^T + Q_jM_j + Q_{j+1}B_j \qquad Q_0B_0 = Q_{k+1}B_k = 0$$

for $j = 1...k$, which leads to the block Lanczos outer iteration:

$$Q_{j+1}B_j = A\bar{Q}_j - Q_jM_j - Q_{j-1}B_{j-1}^T. \tag{1.3}$$

From the orthogonality of Q we have

$$M_j = Q_j^HA\bar{Q}_j$$

for $j = 1...k$. Let $R_j = A\bar{Q}_j - Q_jM_j - Q_{j-1}B_{j-1}^T \in C^{n\times b}$, then $Q_{j+1}B_j = R_j$ is a QR factorization of R_j.

Algorithm 1 (Block Tridiagonalization) *Given an n-by-b starting matrix S of orthonormal columns and a subroutine for matrix-matrix multiplication Y = AX for any X, where A is an n-by-n complex symmetric matrix. This algorithm computes the diagonal blocks of the block tridiagonal complex symmetric matrix J in (1.1) and a unitary Q such that* $J = Q^HA\bar{Q}$

```
$k = n/b$;
$Q_0 = 0$; $B_0 = 0$;
$Q_1 = S$;
for $j = 1$ to $k$
    $Y = A\bar{Q}_j$;
    $M_j = Q_j^HY$;
    $R_j = Y - Q_jM_j - Q_{j-1}B_{j-1}^T$;
    $Q_{j+1}B_j = R_j$;    (QR factorization of $R_j$)
end.
```

1.2 Tridiagonalization

We follow Berry [5] to adopt the single vector Lanczos tridiagonalization recursion for reducing the block tridiagonal complex symmetric matrix to tridiagonal form.

Let J be the k-by-k block tridiagonal complex symmetric matrix resulted from Algorithm 1. We can find a unitary $P = [\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n]$, $\mathbf{p}_i \in C^{n\times 1}$, such that

$$P^H J\bar{P} = T = \begin{bmatrix} \alpha_1 & \beta_1 & & \dots & 0 \\ \beta_1 & \alpha_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_{n-1} \\ 0 & \dots & & \beta_{n-1} & \alpha_n \end{bmatrix} \tag{1.4}$$

Rewriting (1.4) as

$$J\bar{P} = PT \tag{1.5}$$

and comparing the jth columns on both sides of (1.5), we have

$$J\bar{\mathbf{p}}_j = \beta_{j-1}\mathbf{p}_{j-1} + \alpha_j\mathbf{p}_j + \beta_j\mathbf{p}_{j+1} \qquad \beta_0\mathbf{p}_0 = 0$$

for j = 1:n - 1, which leads to the inner Lanczos recursion:

$$\beta_j\mathbf{p}_{j+1} = J\bar{\mathbf{p}}_j - \alpha_j\mathbf{p}_j - \beta_{j-1}\mathbf{p}_{j-1} \tag{1.6}$$

From the orthogonality of P we have

$$\alpha_j = \mathbf{p}_j^H J\bar{\mathbf{p}}_j$$

for $j = 1...n$. Let $\mathbf{r}_j = J\bar{\mathbf{p}}_j - \alpha_j\mathbf{p}_j - \beta_{j-1}\mathbf{p}_{j-1} \in C^{n\times 1}$, then $\beta_j = \pm\|\mathbf{r}_j\|_2$ and $\mathbf{p}_{j+1} = \mathbf{r}_j/\beta_j$ if $\mathbf{r}_j \neq 0$. The symmetric block tridiagonal structure of J is exploited in the calculation of $J\bar{\mathbf{p}}_j$ for efficiency.

Algorithm 2 (Tridiagonalization) *Given a starting vector* $\mathbf{b}$ *and a subroutine for symmetric block tridiagonal matrix-vector multiplication* $\mathbf{y} = J\mathbf{x}$ *for any* $\mathbf{x}$*, where* J *is the complex symmetric and block tridiagonal matrix in (1.1), this algorithm computes the diagonals of the complex symmetric and tridiagonal matrix* T *and unitary* P *in (1.4).*

```
$\mathbf{p}_0 = 0$; $\beta_0 = 0$;
$\mathbf{p}_1 = \mathbf{b}/\|\mathbf{b}\|_2$;
for $j = 1$ to $n$
    $\mathbf{y} = J\bar{\mathbf{p}}_j$;    (symmetric block tridiagonal matrix-vector multiplication)
    $\alpha_j = \mathbf{p}_j^{\mathrm{H}}\mathbf{y}$;
    $\mathbf{r}_j = \mathbf{y} - \alpha_j\mathbf{p}_j - \beta_{j-1}\mathbf{p}_{j-1}$;
    $\beta_j = \|\mathbf{r}_j\|_2$;
    if $\beta_j = 0$, quit; end
    $\mathbf{p}_{j+1} = \mathbf{r}_j/\beta_j$;
end.
```

1.3 Model of Estimating Orthogonality

The algorithms described in the previous sections assume exact arithmetic. In the presence of rounding error, however, the computed Q loses orthogonality. Thus, orthogonalization is necessary for practical Lanczos methods. Obviously, to detect the loss of orthogonality we must measure the orthogonality. Although Q^HQ can be a measurement for the orthogonality, it is too expensive to compute in every iteration. In this section, we present a model of estimating the orthogonality of Q computed by our block tridiagonalization algorithm. Denoting $W_{k,j} = Q_k^H Q_j$, we propose an efficient recurssion of $W_{k,j}$ including rounding errors without explicitly computing $Q_k^H Q_j$.

Incorporating rounding errors into the jth iteration of (1.3), we have

$$Q_{j+1}B_j + F_j = A\bar{Q}_j - Q_jM_j - Q_{j-1}B_{j-1}^T, \qquad j = 1, ..., k, \tag{1.7}$$

where F_j represents the rounding error at step j. From (1.7), we have

$$Q_{j+1}B_j = A\bar{Q}_j - Q_jM_j - Q_{j-1}B_{j-1}^T - F_j$$

and

$$Q_{k+1}B_k = A\bar{Q}_k - Q_kM_k - Q_{k-1}B_{k-1}^T - F_k.$$

Premultiplying the above two equations with Q_k^H and Q_j^H respectively, we get

$$W_{k,j+1}B_j = Q_k^H A\bar{Q}_j - W_{k,j}M_j - W_{k,j-1}B_{j-1}^T - Q_k^H F_j \tag{1.8}$$

and

$$W_{j,k+1}B_k = Q_j^H A\bar{Q}_k - W_{j,k}M_k - W_{j,k-1}B_{k-1}^T - Q_j^H F_k. \tag{1.9}$$

From (1.9) we get

$$Q_j^H A\bar{Q}_k = W_{j,k+1}B_k + W_{j,k}M_k + W_{j,k-1}B_{k-1}^T + Q_j^H F_k. \tag{1.10}$$

Since the transpose of $Q_j^H A\bar{Q}_k$ is $Q_k^H A\bar{Q}_j$ in (1.8) and $W_{k,j}^T = \bar{W}_{j,k}$, substituting $Q_k^H A\bar{Q}_j$ in (1.8) with (1.10) results in

$$W_{k,j+1}B_j = B_k^T\bar{W}_{k+1,j} + M_k\bar{W}_{k,j} + B_{k-1}\bar{W}_{k-1,j} - W_{k,j}M_j - W_{k,j-1}B_{j-1}^T + G_{k,j}, \tag{1.11}$$

where $W_{j-1,j-1} = \bar{W}_{j,j} = I$, for $k = 1, ..., j-1$, where $G_{k,j} = F_k^T\bar{Q}_j - Q_k^H F_j$ represents the local rounding error. The above equation (1.11) shows that $W_{k,j+1}$ in iteration $j+1$ can be obtained by $W_{k-1,j}$, $W_{k,j}$, and $W_{k+1,j}$ in iteration j and $W_{k,j-1}$ in iteration $j-1$. In this model, $W_{k,j}$ measures the orthogonality $Q_k^H Q_j$ including rounding error. In the following two sections, we show how to use $W_{k,j}$ to detect the loss of orthogonality.

1.4 Componentwise Detection

Let $w_{x,y}$ be the (x,y)-entry of $W_{k,j+1}$, then $w_{x,y} > \sqrt{\epsilon}$ means that the orthogonality between $\mathbf{q}_x$ in Q_k and $\mathbf{q}_y$ in Q_{j+1} is lost. Thus it detects loss of orthogonality more accurately than the normwise scheme.

Our componentwise orthogonalization scheme is based on the model (1.11). For the term $G_{k,j}$ in (1.11), we use a block version of modified partial reorthogonalization scheme in [1]:

$$G_{k,j} = \epsilon(B_k + B_j)(\Theta_{\rm r} + i\Theta_{\rm i}), \quad \Theta_{\rm r},\ \Theta_{\rm i} \in N(0, 0.3), \tag{1.12}$$

where $(B_k + B_j)(\Theta_{\rm r} + i\Theta_{\rm i})$ means that each entry in $B_k + B_j$ is multiplied with a normally distributed random number with zero mean and variance 0.3. Also, we set $W_{j,j+1}$ so that

$$W_{j,j+1}B_j = b\epsilon B_1(\Psi_{\rm r} + i\Psi_{\rm i}), \quad \Psi_{\rm r},\ \Psi_{\rm i} \in N(0, 0.6). \tag{1.13}$$

Now that we have described the computation of $W_{k,j+1}$ for $k = 1, ..., j$, in the following we discuss the determination of the orthogonalization intervals. Analogous to the normwise method, when the absolute value of a component $w_{x,y}$ of $W_{k,j+1}$ exceeds $\sqrt{\epsilon}$, we find the largest interval $[l_k, u_k]$ such that $k \in [l_k, u_k]$ and $W_{i,j+1}$ has a component larger than a tolerance for all $i \in [l_k, u_k]$. Based on our experiments, we chose $\epsilon^{7/8}$, a value between ϵ and $\sqrt{\epsilon}$, as the tolerance. Our experiments have also shown that for each j, there is usually only one interval, if it exists, and the lower end of the interval is close to one and the upper end is near j. Thus, to save search time, we always set the lower end to 1 and search for the upper end starting from j.

To incorporate rounding error, after the columns of Q_{j+1} are orthogonalized against the columns of Q_k, we set

$$W_{k,j+1} = \epsilon(\Omega_{\rm r} + i\Omega_{\rm i}), \quad \Omega_{\rm r},\ \Omega_{\rm i} \in N(0, 1.5), \tag{1.14}$$

that is the entries of $W_{k,j+1}$ are normally distributed random numbers with zero mean and variance 1.5.

Whenever we perform the modified partial orthogonalization in iteration j, we always carry out orthogonalization in the subsequent iteration $j+1$ so that the next block generated by the recurrence is almost orthogonal to its predecessors. Suppose that the orthogonalization interval in iteration j is $[1, u]$. As shown in (1.11), the computation of $W_{k,j+1}$ requires $W_{k-1,j}$, $W_{k,j}$, and $W_{k+1,j}$, thus we expand the orthogonalization inteval $[1, u]$ to $[1, u+1]$ for the subsequent iteration.

Algorithm 3 (Componentwise Detection) *Given an n-by-b starting matrix S of orthonormal columns and a subroutine for matrix-matrix multiplication $Y = AX$ for* *any X, where A is an n-by-n complex symmetric matrix. This algorithm computes the diagonal blocks of the block tridiagonal complex symmetric matrix J in (1.1) and a unitary Q such that* $J = Q^H A\bar{Q}$

```
$steps = n/b$;
$Q_0 = 0$; $B_0 = 0$;
$Q_1 = S$;
$doOrtho = 0$;
$second = 0$;
for $j = 1$ to $steps$
    $Y = A\bar{Q}_j$;
    $M_j = Q_j^H Y$;
    $R_j = Y - Q_j M_j - Q_{j-1} B_{j-1}^T$;
    $Q_{j+1} B_j = R_j$;    (QR factorization of $R_j$)

    if $second == 0$
        $k = 1$;
        while $(k \leq j)$ and $(doOrtho \mathrel{!=} 1)$
            if $1 \leq k \leq j-1$
                Compute $W_{k,j+1}$ using (1.11) (1.12);
            else
                Compute $W_{j,j+1}$ using (1.13);
            end
            if one component of $W_{k,j+1}$ exceeds $\sqrt{\epsilon}$
                $doOrtho = 1$;
            end
            $k = k + 1$;
        end

        if $doOrtho == 1$
            $thresh = 0$;
            $k = j$;
            while $(k \geq 2)$ and $(thresh \mathrel{!=} 1)$
                if $1 \leq k \leq j-1$
                    Compute $W_{k,j+1}$ using (1.11) (1.12);
                else
                    Compute $W_{j,j+1}$ using (1.13);
                end
                Search every component of just computed $W_{k,j+1}$;
                if one component of $W_{k,j+1}$ exceeds $\epsilon^{7/8}$
```

```
                    $up = k$;
                    $thresh = 1$;
                end
                $k = k - 1$;
            end
        end
    end

    if ($doOrtho == 1$) or ($second == 1$)
        Orthogonalize $R_j$ against $Q_1, ..., Q_{up}$;
        Reset $W_{1,j+1}, ..., W_{up,j+1}$ using (1.14);
        Recalculate QR factorization $Q_{j+1}B_j = R_j$;
        if ($second == 1$)
            $second = 0$;
        else
            $second = 1$;
            $doOrtho = 0$;
            Adjust the neighborhood to $[1, up + 1]$ for the next iteration;
        end
    end
end.
```

1.5 Orthogonalization in Tridiagonalization

Analogous to the orthogonalization algorithm for the block tridiagonalization, we present a modified partial orthogonalization scheme for the tridiagonalization. We first identify mechanisms whereby orthogonality is lost and then apply a model of the loss of orthogonality to determine when to correct the situation. This algorithm follows the modified partial orthogonalization algorithm of S.Qiao [1].

Incorporate roundoff errors into the kth Lanczos iteration (1.6), we write

$$\beta_j \mathbf{p}_{j+1} + \mathbf{f}_j = J\bar{\mathbf{p}}_j - \alpha_j \mathbf{p}_j - \beta_{j-1}\mathbf{p}_{j-1}, \quad j = 1, .., k, \tag{1.15}$$

where $\mathbf{f}_j$ represents roundoff errors. From (1.15), we have

$$\begin{aligned} \beta_j \mathbf{p}_{j+1} &= J\bar{\mathbf{p}}_j - \alpha_j \mathbf{p}_j - \beta_{j-1}\mathbf{p}_{j-1} - \mathbf{f}_j \\ \beta_k \mathbf{p}_{k+1} &= J\bar{\mathbf{p}}_k - \alpha_k \mathbf{p}_k - \beta_{k-1}\mathbf{p}_{k-1} - \mathbf{f}_k \end{aligned}$$

Premultiplying the above two equations with $\mathbf{p}_k^{\mathrm{H}}$ and $\mathbf{p}_j^{\mathrm{H}}$ respectively and denoting

$\omega_{k,j} = \mathbf{p}_k^{\mathrm{H}} \mathbf{p}_j$, we get

$$\begin{aligned}\beta_j \omega_{k,j+1} &= \mathbf{p}_k^{\mathrm{H}} A \bar{\mathbf{p}}_j - \alpha_j \omega_{k,j} - \beta_{j-1} \omega_{k,j-1} - \mathbf{p}_k^{\mathrm{H}} \mathbf{f}_j \\ \beta_k \omega_{j,k+1} &= \mathbf{p}_j^{\mathrm{H}} A \bar{\mathbf{p}}_k - \alpha_k \omega_{j,k} - \beta_{k-1} \omega_{j,k-1} - \mathbf{p}_j^{\mathrm{H}} \mathbf{f}_k\end{aligned}$$

Since J is symmetric, $\mathbf{p}_k^{\mathrm{H}} J \bar{\mathbf{p}}_j = \mathbf{p}_j^{\mathrm{H}} J \bar{\mathbf{p}}_k$. Thus, subtracting the above two equations and noting that $\omega_{k,j} = \bar{\omega}_{j,k}$, we have the following recursion on the orthogonalities of the Lanczos vectors:

$$\beta_j \omega_{k,j+1} = \beta_k \bar{\omega}_{k+1,j} + \alpha_k \bar{\omega}_{k,j} - \alpha_j \omega_{k,j} + \beta_{k-1} \bar{\omega}_{k-1,j} - \beta_{j-1} \omega_{k,j-1} + \mathbf{p}_j^{\mathrm{H}} \mathbf{f}_k - \mathbf{p}_k^{\mathrm{H}} \mathbf{f}_j$$

We define

$$\psi_j = \omega_{j,j+1}$$

$$\theta_{k,j} = \mathbf{p}_j^{\mathrm{H}} \mathbf{f}_k - \mathbf{p}_k^{\mathrm{H}} \mathbf{f}_j$$

Using these notations, we get

$$\omega_{k,j+1} = \beta_j^{-1} (\beta_k \bar{\omega}_{k+1,j} + \alpha_k \bar{\omega}_{k,j} - \alpha_j \omega_{k,j} + \beta_{k-1} \bar{\omega}_{k-1,j} - \beta_{j-1} \omega_{k,j-1}) + \theta_{k,j} \tag{1.16}$$

for $k = 1, ..., j-1$, with

$$\beta_0 = \omega_{0,j} = 0, \quad \omega_{j,j+1} = \psi_j \quad \text{and} \quad \omega_{j+1,j+1} = 1.0$$

Let ϵ be the unit of roundoff, we propose that

$$\psi_j = n\epsilon \frac{\beta_1}{\beta_j} (\Psi_{\mathrm{r}} + i\Psi_{\mathrm{i}}), \quad \Psi_{\mathrm{r}},\ \Psi_{\mathrm{i}} \in N(0, 0.6) \tag{1.17}$$

where $N(0, v)$ means normal distribution with zero mean and variance v, and

$$\theta_{k,j} = \epsilon (\beta_k + \beta_j)(\Theta_{\mathrm{r}} + i\Theta_{\mathrm{i}}), \quad \Theta_{\mathrm{r}},\ \Theta_{\mathrm{i}} \in N(0, 0.3) \tag{1.18}$$

To aleviate the problem caused by isolated reorthogonalization, when $\omega_{k,j+1}$ exceeds the tolerance *tol* for some k, we perform modified partial orthogonalization scheme in which $\mathbf{r}_j$ is orthogonalized against only $\mathbf{p}_k$ and its neighboring vectors, not against all $\mathbf{p}_1$,...,$\mathbf{p}_j$ like partial orthogonalization scheme does. Suppose $[l_k, u_k]$ is the neighborhood of k, then it is the largest interval such that $k \in [l_k, u_k]$ and $|\omega_{i,j+1}| \geq tol$ for all i between l_k and u_k. The tolerance *tol* should be some value between *epsilon* and $\sqrt{\epsilon}$. We choose $tol = \epsilon^{3/4}$. Theoretically, after the reorthogonalization $\omega_{k,j+1} = 0$ for $k = 1, ..., j$. To incorporate the rounding errors, we set

$$\omega_{k,j+1} = \epsilon (\Omega_{\mathrm{r}} + i\Omega_{\mathrm{i}}), \quad \Omega_{\mathrm{r}},\ \Omega_{\mathrm{i}} \in N(0, 1.5) \tag{1.19}$$

As Algorithm 3, we always perform a reorthogonalization in the subsequent iteration $j+1$. As shown in (1.16), since the computation of $\omega_{k,j+1}$ requires $\omega_{k-1,j}$, $\omega_{k,j}$, and $\omega_{k+1,j}$, we expand each inteval $[l_k, u_k]$ to $[l_k - 1, u_k + 1]$ for the subsequent iteration.

Algorithm 4 (Orthogonalization in Tridiagonalization) *Given a starting vector* $\mathbf{b}$ *and a subroutine for symmetric band matrix-vector multiplication* $\mathbf{y} = J\mathbf{x}$ *for any* $\mathbf{x}$*, where* J *is a* k*-by-*k *block complex symmetric matrix, and every block is* b*-by-*b*. Suppose* $n = k \times b$*. Using the modified partial orthogonalization, this algorithm computes the diagonals of the complex symmetric and tridiagonal matrix* T *in (1.4) and a unitary* P *such that* $T = P^{\mathrm{H}} A \bar{P}$*.*

```
$\mathbf{p}_0 = 0$; $\beta_0 = 0$; $\omega_{1,1} = 1$;
$\mathbf{p}_1 = \mathbf{b}/\|\mathbf{b}\|_2$;
for $j = 1$ to $n$
    $\mathbf{y} = J\bar{\mathbf{p}}_j$;    (symmetric band matrix-vector multiplication)
    $\alpha_j = \mathbf{p}_j^{\mathrm{H}}\mathbf{y}$;
    $\mathbf{r}_j = \mathbf{y} - \alpha_j\mathbf{p}_j - \beta_{j-1}\mathbf{p}_{j-1}$;
    $\beta_j = \|\mathbf{r}_j\|_2$;

    Compute $\omega_{k,j+1}$ for $k = 1, ..., j-1$ using (1.16);
    Set $\omega_{j,j+1}$ to $\psi_j$ using (1.17);
    Set $\omega_{j+1,j+1} = 1$;
    $k = 1$;
    while $k \leq j$
        if $|\omega_{k,j+1}| > \sqrt{\epsilon}$
            Find the neighborhood $[l_k, u_k]$ of $k$;
            $k = u_k + 1$;
        else
            $k = k + 1$;
        end
    end

    for each inteval $[l_k, u_k]$
        Orthogonalize $\mathbf{r}_j$ against $\mathbf{p}_{l_k}, ..., \mathbf{p}_{u_k}$;
        Reset $\omega_{l_k,j+1}, ..., \omega_{u_k,j+1}$ using (1.19);
        Adjust the neighborhood to $[l_k - 1, u_k + 1]$ for the next iteration;
    end
    if orthogonalization was performed
        Recalculate $\beta_j = \|\mathbf{r}_j\|_2$;
    end
    if $\beta_j = 0$, quit; end
    $\mathbf{p}_{j+1} = \mathbf{r}_j/\beta_j$;
end.
```

Chapter 2

High Level Design

Implementing a complicated numerical software involves in many kinds of basic linear algebra computations. In Section 2.1 we describe the process of selecting existing numerical software packages to perform these basic computations and speed up the development. As we said in Chapter 2, We select one of two schemes to detect the loss of orthogonality of the computed matrix. In Section 2.2 we present our reason for the selection. In Section 2.4 we describe the design of the user interface of the implementation. In Section 2.5 we present the hierarchy of modules. Our design of the data structure is described In Section 2.6.

2.1 Selection of Numerical Packages

The block Lanczos project has many basic matrix and vector computations, for example, the matrix-matrix multiplication and division, the transpose or conjugate of one complex matrix and QR factorization, etc. These routines are critical for the performance of the project. It is very important to select one software package which meets the requirement of the project, and which is highly optimized.

LAPACK (Linear Algebra PACKage) is a free, portable and standard library of Fortran 77 routines for solving the most common problems in numerical linear algebra. It is designed to be efficient on a wide range of high-performance computers, under the proviso that the hardware vendor has implemented an efficient set of BLAS (Basic Linear Algebra Subroutines). LAPACK is a natural choice for the project.

LAPACK++ is an object-oriented C++ extension of the LAPACK library. The advantages of an object-oriented approach include the ability to encapsulate various matrix representations, hide their implementation details, reduce the number of subroutines, simplify their calling sequences, and provide an extendible software framework that can incorporate future extensions of LAPACK. LAPACK++ seems to be a better choice than LAPACK. Unfortunately, the LAPACK++ users' guide [11] makes the statement that complex support disabled due to the transitory support among C++ compilers, and new developments and ongoing efforts have shifted to the Template Numerical Toolkit (TNT) for Linear Algebra project, based on the Standard Template Library (STL).

The Template Numerical Toolkit (TNT), the successor to the LAPACK++, is a collection of interfaces and reference implementations of numerical objects useful for scientific computing in C++. The toolkit defines interfaces for basic data structures commonly used in numerical applications. The goal of this package is to provide reusable software components that address many of the portability and maintenance problems with C++ codes. But this package provides few useful functionalities besides accessing arrays. I haven't found QR factorization solution in it yet. According to some web sites it is at an early stage of development.

So at present only LAPACK fits our purposes well. The project is module oriented, not object oriented because we select LAPACK as our numerical software library, and the complicated block Lanczos algorithms focus on procedural.

2.2 Selection of Algorithms

In the presence of rounding error the computed Q loses orthogonality. Thus, orthogonalization is necessary for the stable block Lanczos tridiagonalization algorithm. Based on a model of estimating the orthogonality of Q, two schemes are proposed in S.Qiao[12] for detecting the loss of orthogonality for block tridiagonalization stage. One is normwise detection involving norm computations. Another is componentwise detection. These two algorithms and the algorithm in the tridiagonalization stage have been implemented in MATLAB. The following numerical experiments made by this MATLAB implementation have shown that componentwise scheme detects loss of orthogonality more accurately than the normwise scheme.

Example 1. Block Lanczos algorithm in which normwise detection was applied in block tridiagonalization stage was run on a random complex symmetric matrix of order 2048. Table 2.1 shows the total number of orthogonalizations and errors for various block sizes.

Example 2. Block Lanczos algorithm in which componentwise detection was applied in block tridiagonalization stage was run on a random complex symmetric matrix of order 2048. Table 2.2 shows the total number of orthogonalizations and errors for various block sizes. This example shows that the componentwise algorithm performed fewer orthogonalizations than the normwise alogrithm.

Example 3. To compare the performance, we generated random complex symmetric

block size	total number of orthogonalizations	error in orthogonality	error in factorization
2	602542	$2.66E-13$	$2.44E-13$
4	633830	$1.34E-13$	$1.27E-13$
8	662991	$8.81E-14$	$8.38E-14$
16	677998	$2.79E-14$	$2.63E-14$
32	810810	$3.53E-14$	$3.21E-14$

Table 2.1: Efficiency and accuracy of block Lanczos algorithm in which normwise detection was applied on a complex symmetric matrix of order 2048.

block size	total number of orthogonalizations	error in orthogonality	error in factorization
2	437969	$8.13E-14$	$7.92E-14$
4	444056	$1.32E-12$	$2.28E-13$
8	425837	$8.00E-14$	$7.72E-14$
16	433081	$1.40E-13$	$1.38E-13$
32	413307	$1.64E-13$	$1.52E-13$

Table 2.2: Efficiency and accuracy of block Lanczos algorithm in which componentwise detection was applied on a complex symmetric matrix of order 2048.

matrices of various sizes and ran the single vector algorithm and the two block algorithms. Figure 2.1 depicts the run times which are normalized so that the execution time of the single vector algorithm is 100. In all cases, the block size in the block algorithms is 32. This example shows that the block algorithm with componentwise detection has the best performance for large matrices.

Since componentwise detection is more efficient than the normwise detection, we select componentwise detection scheme as the algorithm in the block tridiagonalization stage.

2.3 LAPACK Functions

We decide to follow LAPACK style to design and document the data structure and interface of the project. To use LAPACK from C in real application, it is important to keep in mind that LAPACK is written in Fortran 77. There are differences between Fortran and C programming:

- Fortran uses only call by reference. To call a Fortran routine from C, it is

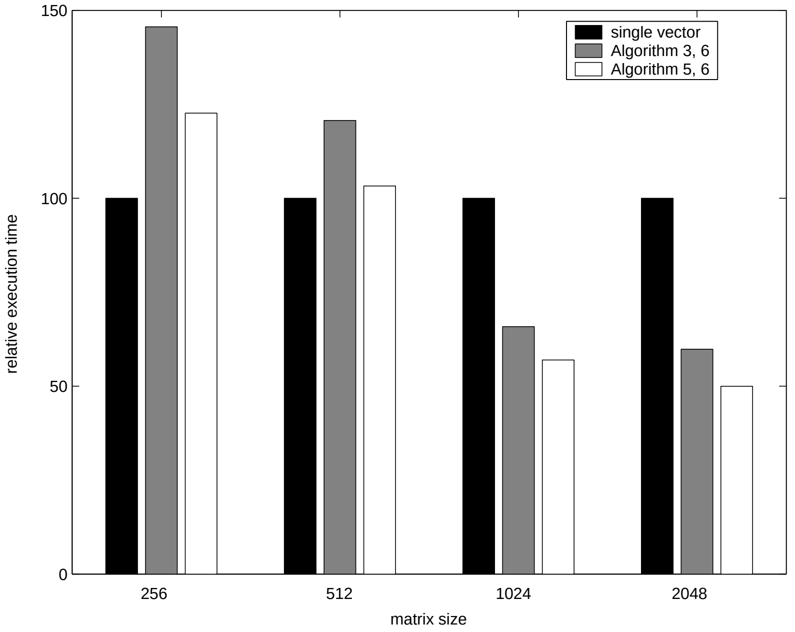

Figure 2.1: Comparison of the efficiency of block Lanczos algorithms. The y axis shows the execution times normalized to the execution time of the single vector algorithm.

necessary to pass pointers to the relevant variables, not the variables themselves.

- When invoking a compiled Fortran library subroutine, it is necessary to put the name in lower case followed by an underscore. The reason is that the Fortran compiler adds the underscore to the routine name when it generates the compiled code.
- Two-dimensional arrays in Fortran are stored by columns, in the opposite order from C.
- Fortran arrays start indexing at 1 by default; C arrays start at 0.
- Dynamic allocation, commonly used in C, is part of the Fortran 90 standard but not the Fortran 77 standard, so Fortran LAPACK rountines typically require the passing of work arrays.
- In C, a function may or may not return a value. In Fortran, it must.

The name of each LAPACK routine is a coded specification of its function. All driver and computational routines have the names of the form XYYZZZ. The first letter, X, indicates the data type. The next two letters, YY, indicate the type of matrix. The last three letters ZZZ indicate the computation performed. For example, the following is a LAPACK routine to reduce a complex Hermitian matrix A to real symmetric tridiagonal form T by a unitary similarity transformation:

```
int zhetrd_ (char *uplo,
             integer *n, doublecomplex *a, integer *lda,
             doublereal *d, doublereal *e,
             doublecomplex *tau, doublecomplex *work,
             integer *lwork, integer *info)
```

2.4 User Interface

Based on the rules of LAPACK, we design the following procedural which user invokes to perform the tridiagonalization of a complex symmetric matrix:

```
int zcstrd_ (integer *n, doublecomplex *A,
             integer *bs, doublecomplex *S,
             doublecomplex *a, doublereal *b,
             doublecomplex *Q, doublecomplex *P,
             integer *info)
```

The first letter, *z*, indicates double complex. The next two letters, *cs*, indicate the complex symmetric matrix. The last three letters, *trd*, indicate the tridiagonalization reduction of one matrix. We adopt the data type in LAPACK, like integer, double-complex, and doublereal, etc. The idea of the design is to allow users to seamlessly invoke the block Lanczos algorithm routine with other LAPACK routines.

The following are descriptions of arguments of the procedural:

Input:	
n	The order of the matrix A. $n >= 0$
A	Double complex array, size $= n * n$
bs	Size of the block. $bs > 0$.
S	Double complex array, size $= n * bs$ The starting orthonormal columns matrix.
Output :	
a	Double complex array, size $= n$ The diagonal elements of the tridiagonal matrix T.
b	Double real array, size $= n - 1$ The off-diagonal elements of the tridiagonal matrix T.

Q Double complex array, size = $n * n$
Q is computed in block tridiagonalization stage.
P Double complex array, size = $n * n$
P is computed in tridiagonalization stage.
$(QP)^{\mathrm{H}}A(\bar{QP}) = T$
info 0 Successful exit
< 0 If info = $-i$, the i-th argument had an illegal value
> 0 Exception is thrown

Return:
always 0

Matirx A is not stored in packed form which LAPACK supports due to the sake of performance which the project requires. It would not be hard to implement the tridiagonalization of complex symmetric matrices in packed storage. Matrix Q and P are matrices which are computed in block tridiagonalization stage and tridiagonalization stage respectively. The return value of this routine is always 0 due to the reason related to the differences between Fortran and C.

We studied LAPACK's routines to figure out what arguments LAPACK checks for illegal values. We follow its way to check the arguments of our project:

- The order of matrix A is illegal if it is less than or equal to 0.
- The size of one block is illegal if it is less than or equal to 0.
- The order of matrix A can't be divided by the block size. Then block size is regarded illegal.

2.5 Modules

We design several layers for the modules implemented in the project. Figure 2.2 shows the hierarchy of modules.

We follow software design principles to design the modules. There is a weak coupling between modules and a strong cohesion within modules. Each module depends on as few other modules as possible. User of the block Lanczos tridiagonalization package maybe interested in only how to invoke the routine to perform tridiagonalization of a complex symmetric matrix. The block Lanczos algorithm user interface is what he can see and use. User can include only one file, *zcstrd.h*, which has the prototype of the routine, *zcstrd_*, in his C code, and then link the block lanczos tridiagonalization package. The user interface module hides the secret of the algorithm and the implementation.

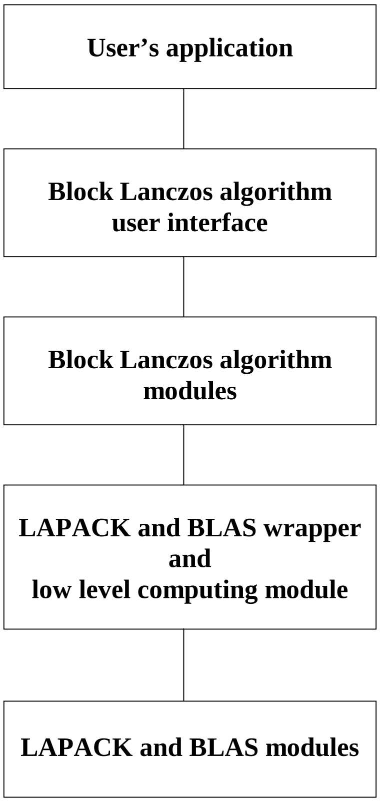
User's application
Block Lanczos algorithm user interface
Block Lanczos algorithm modules
LAPACK and BLAS wrapper and low level computing module
LAPACK and BLAS modules

Figure 2.2: hierarchy of modules

Block Lanczos algorithm modules are only related to the algorithm itself. They don't invoke any LAPACK or BLAS routine explicitly. The algorithm itself places the most important role in the performance of the project. If the algorithm has the possibility to improve, then only the block Lanczos algorithm modules should be changed without the change of other modules.

LAPACK and BLAS wrapper module implements the computations which block Lanczos algorithm modules require by calling LAPACK and BLAS routines. For example QR factorization, matrix-matrix operations and matrix-vector operations. Block Lanczos algorithm requires some low level computations which LAPACK and BLAS don't provide explicitly. These requirements are implemented in one separate module, low level computing module. The LAPACK and BLAS wrapper and the low level computing module are at the same layer in the module hierarchy. The changes to these computing modules should not affect the block Lanczos algorithm modules provided the interface between block Lanczos algorithm modules and the computing modules is not changed.

LAPACK and BLAS have the incomparable performance. But explicitly calling any one of LAPACK and BLAS routines in the block Lanczos algorithm modules would be a nightmare. Take a look at the prototype of one LAPACK routine which is frequently called in block Lanczos algorithm modules. It computes the solution of A/B, where B is upper triangular:

```
int ztrsm_ (char *side, char *uplo,
            char *transa, char *diag, integer *m, integer *n,
            doublecomplex *alpha, doublecomplex *a, integer *lda,
            doublecomplex *b, integer *ldb)
```

After we studied the LAPACK and BLAS routines, we developed the LAPACK and BLAS wrapper as the beginning of the project implementation. We generalized what kinds of computations may be used in the block Lanczos algorithm modules, then implemented them. For example, one of level 3 BLAS functions, *zgemm_*(), implements the computation of $\alpha op(A)op(B) + \beta op(C)$ where $op(X) = X, X^T, X^H$. We found out that we only need $op(A)op(B) + op(C)$ in block Lanczos algorithm after carefully generalizing this matrix-matrix operation. So we developed a function, *mmult*(), to wrap this LAPACK function to meet our specific need. The caller of *mmult*() needn't worry about arguments α and β any longer.

Due to the importance of the accuracy of the wrapper, we took considerable time to make stress tests to verify the wrapper. We used as many cases as possible to test every path in the routine of the wrapper. We must guarantee the wrapper's accuracy before we started the implementation of block Lanczos algorithm modules.

The fact proved that this work is valuable and even more than the project requires. You can imagine we have a lot of computation troubles in the block Lanczos algorithm modules. But we never worried much about whether these troubles came from the LAPACK and BLAS wrapper and low level computing modules. We concentrated on the block Lanczos algorithm modules to check errors. This bottom-up strategy guaranteed the smooth development.

After we finished and verified the whole project implementation, we figured out there were areas we could improve to achieve as best performance as possible. The LAPACK and BLAS wrapper was one of them. The wrapper was so general that it could handle more cases than the project requires. Many cases never happen in the block Lanczos algorithm modules. But the tests of these cases affected a lot on the performance of the project. So at the last stage of the project we modified the LAPACK and BLAS wrapper to handle the cases that may happen only in the block Lanczos algorithm. We found out the performance of the project was greatly improved. When we made these modification, we never touched other modules, like block Lanczos algorithm modules.

If LAPACK and BLAS are not selected as the support package in the future, only the LAPACK and BLAS wrapper needs to be replaced. The change of the wrapper is transparent to other modules.

The low level computing module deals with computations like element-by-element product of two complex matrices and multiplication and addition of complex numbers. We developed them by referring to the implementation of the related BLAS routines. Professor Qiao considered to use GNU package to replace them in the future work. This replacement can be done in only one module without any modification of other modules.

At the bottom of the hierarchy of the modules are the LAPACK and BLAS routines.

It has other advantages to design the modules as this hierarchy. Users of the block Lanczos tridiagonalization package can understand the sizable block Lanczos algorithm implementation more easily. Although the code of the project is implemented all by Guohong Liu, it could be implemented by several programmers. Each programmer develops a separate module.

2.6 Data Structure

We design the following structure for a m-by-n complex matrix,

```
typedef struct doubleComplexMat {
    doublecomplex *mat;
    int m;
    int n;
} DoubleComplexMat;
```

This data structure wraps the LAPACK data type, doublecomplex, with the dimension of a matrix. It is used between modules to simplify the arguments of the routines. For example, LAPACK's QR factorization module is like the following by using doublecomplex,

```
void qr (int m, int n,
         doublecomplex *A,
         doublecomplex *Q, doublecomplex *R)
```

The same routine can be changed to a simpler and more readable form by using the data type we define,

```
void qr (DoubleComplexMat *A,
         DoubleComplexMat *Q, DoubleComplexMat *R);
```

We would like to use a example to show that the LAPACK and BLAS wrapper and the data structure can make the implementation of the project more easily understood. It is related to LAPACK's matrix-matrix multiplication routine which is widely used in the project. The prototype of it is as the following,

```
int zgemm_ (char *transa, char *transb, integer *m, integer *n,
            integer *k, doublecomplex *alpha,
            doublecomplex *a, integer *lda,
            doublecomplex *b, integer *ldb,
            doublecomplex *beta, doublecomplex *c__,
            integer *ldc)
```

There is a corresponding routine in LAPACK wrapper to perform the computation $R = op(A)op(B) + C$ or $R = -op(A)op(B) + C$. The prototype and the description of arguments of the routine are as the following,

```
int mmult (char *signa, DoubleComplexMat *A, char *transa,
           DoubleComplexMat *B, char *transb,
           DoubleComplexMat *C, DoubleComplexMat *R)
```

signa	” + ”,	$op(A) * op(B) + C$
	” − ”,	$-op(A) * op(B) + C$
A	Matrix.	
transa	”N”,	$op(A) = A$
	”T”,	$op(A) = A^{\mathrm{T}}$
	”H”,	$op(A) = A^{\mathrm{H}}$
B	Matrix.	
transb	”N”,	$op(B) = B$
	”T”,	$op(B) = B^{\mathrm{T}}$
	”H”,	$op(B) = B^{\mathrm{H}}$
	”C”,	$op(B) = \bar{B}$
C	Matrix. if C = NULL, then $R = op(A) * op(B)$ or $R = -op(A) * op(B)$	
R	Matrix.	

Suppose there are four matrices, $DoubleComplexMat * A, *Q, *R, *M$. We would like to perform the following computation in the block Lanczos algorithm:

$$R = A\bar{Q}$$
$$M = Q^{\mathrm{H}}R$$
$$R = R - QM$$

Then the corresponding C implementation would be:

```
mmult (” + ”, A, ”N”, Q, ”C”, NULL, R);
mmult (” + ”, Q, ”H”, R, ”N”, NULL, M);
mmult (” − ”, Q, ”N”, M, ”N”, R, R);
```

It has simpler and more understandable form than LAPACK routines. More importantly it decreases the chances of introducing errors when writing down so many arguments in one LAPACK routine. It is headache to read the manual of LAPACK, and to be very careful about the arguments. This design allows us to concentrate on the business of block Lanczos algorithm, not disturbed by LAPACK routines any more once the LAPACK and BLAS wrapper is believed to be correctly implemented.

By the same consideration we define the following data types for double complex vector and double real vector:

```
typedef struct doubleComplexVec {
      DoubleComplex *vec;
      int n;
} DoubleComplexVec;

typedef struct doubleRealVec {
      DoubleReal *vec;
      int n;
} DoubleRealVec;
```

Chapter 3

Implementation

We describe the implementation of block Lanczos tridiagonalization algorithm in details. Section 3.1 describes the hierarchy of the source code and the description of every files. Section 3.2 is about the naming standard in the project. Section 3.3 presents some important functions we implemented for the project. Section 3.4 presents the data type and the technology to access FORTRAN arrays that LAPACK expects. The important variables used in block tridiagonalization stage and tridiagonalization stage are described in Section 3.5. We describe the memory operation principles in Section 3.7. The optimization we made for the project is described in Section 3.8. The we present some samples of code and comment in Section 3.9, 3.10, 3.11 and 3.12.

3.1 Project Source Files

There are 16 C files we developed in the project. Figure 3.1 shows the hierarchy of these files. They are listed as the following:

- BlkLanApp.c BlkLanApp.h
  Sample application of the block Lanczos tridiagonalization algorithm. User can refer to this sample to invoke our routine to perform the tridiagonalization of a complex symmetric matrix.

- zcstrd.h
  The prototype of the routine user can invoke to perform the tridiagonalization of a complex symmetric matrix.

- zcstrd.c
  Implement the user interface of block Lanczos tridiagonalization algorithm. It

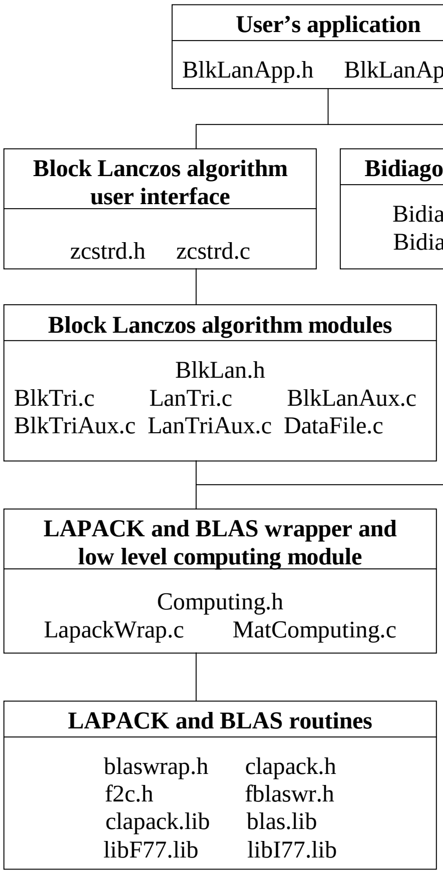

Figure 3.1: hierarchy of modules

invokes routines implemented in block Lanczos algorithm modules to perform the tridiagonalization of a complex symmetric matrix.

- Bidiagonal.c Bidiagonal.h
  Implement the bidiagonalization of general complex matrices by invoking LAPACK's routines. Bidiagonalization of a complex matrix is used to compare the performance with the block Lanczos tridiagonalization algorithm.

- BlkLan.h
  Define prototypes of all routines implemented for block Lanczos algorithm.

- BlkTri.c
  Implement routines in the block tridiagonalization stage.

- BlkTriAux.c
  Perform initialization for the block tridiagonalization. It is also responsible for dealing with exceptions thrown in this stage, and releasing resources when block tridiagonalization is done.

- LanTri.c
  Implement the procedurals used in the tridiagonalization stage.

- LanTriAux.c
  Perform initialization in the tridiagonalization stage. It also deals with exceptions thrown in this stage. It releases resources when tridiagonalization is done.

- BlkLanAux.c
  Implement some auxiliary routines used in block Lanczos algorithm.

- DataFile.c Implement the functionality to read one data file which includes the specific matrices and vectors coming from MATLAB. It is for verification purpose.

- Computing.h
  Define the data structures and the prototypes of routines in LAPACK and BLAS wrapper and low level computing module.

- LapackWrap.c
  Implement routines of LAPACK and BLAS wrapper.

- MatComputing.c
  Implement computing routines which are required by the block Lanczos algorithm, but are not provided by LAPACK or BLAS explicitly.

3.2 Rules of Naming and Comment

The naming of variables and functions follows conformed rules. We referred to the naming standard of LAPACK, BLAS and MATLAB. The name of the matrix begins with the big capital, the vector with the small capital. In LAPACK and BLAS wrapper the input matrix is specified as A, B or C. The solution of the computation is R if it is a matrix. The input vector is specified as x, y or z. The solution of the computation is r if it is a vector. The constant in the LAPACK and BLAS wrapper is called alpha or beta.

In tridiagonalization stage, two kinds of vectors, double real vector and double complex vector, are involved in computation. 'D' or 'd' in the name of a variable or a function means double, 'Z' or 'z' means complex. This is LAPACK's naming standard.

The names of functions in the LAPACK and BLAS wrapper and low level computing module are all in small capital. The names of functions in other modules are mixed with big capital and small capital characters. For the arguments of functions, the input arguments are in front of output arguments, and in the order they are in the computing. The following is the sample of a routine which performs matrix-vector operation $r = op(A) * x + y$, where $op(A) = A$ or A^T:

```
int mvmult (DoubleComplexMat *A, char *transa,
            DoubleComplexVec *x, DoubleComplexVec *y,
            DoubleComplexVec *r)
```

We adopt MATLAB style in the comment of the code. $Q.'$ means Q^T. Q' means Q^H, and $conjg(Q)$ means $\bar{Q}$.

3.3 Functions

We describe some important routines implemented in block Lanczos algorithm modules, LAPACK and BLAS wrapper and low level computing module respectively.

The following are the important routines implemented for the block tridiagonalization stage:

- blkTri
  The main procedural to perform block tridiagonalization.
- detectW
  Compute W's, the detectors of the loss of orthogonality in block tridiagonalization stage.
- orthInterval
  Determine the orthogonalization intervals
- completeW
  In the second orthogonalization stage, compute W's which are not computed in last iteration due to orthogonalization.
- orthR
  Orthogonalize Q_{j+1}, current Q block, against all Q's blocks which are in the orthogonalization intervals.

- blkTriInit
  Perform initialization for block tridiagonalization operation. Allocate necessary memories for computations.

- blkTriEnd
  Release memories after block tridiagonalization is done.

- blkTriExcept
  Handle exceptions thrown in the block tridiagonalization stage. Then release memories one after another.

The following are important routines implemented for tridiagonalization stage:

- lanTri
  Main procedural in tridiagonalization stage.

- detectw
  Compute w's, the detectors of the loss of orthogonality in tridiagonalization stage.

- orthr
  Carry out orthogonalization in tridiagonalization stage.

- sbmvmul
  Complex symmetric and block tridiagonal matrix-vector multiplication, $r = J\bar{P}_j$, where J is complex symmetric and block tridiagonal whose main diagonal blocks are matrices M, and subdiagonal blocks are matrices B. Matrices M and B are computed in the block tridiagonalizaition stage.

- lanTriInit
  Perform initialization for tridiagonalization operation. Allocate necessary memories for computation.

- lanTriEnd
  Release memories after tridiagonalization is done.

- lanTriExcept
  Handle exceptions thrown in the tridiagonalization stage. Then release memories one after another.

The routines implemented in LAPACK and BLAS wrapper are listed below:

- wrapperInit
  Allocate work memories for the routines of Lapack wrapper.

- freeWrapper
  Release work memories occupied by LAPACK and BLAS wrapper.

- qr
  Perform QR factorization.

- orthogonalize
  Apply Gram-Schmidt method to orthogonalizing A against B.

- mmult
  $\pm op(A) * op(B) + C$

- bmmult
  This function is the same as $mmult()$ except matrices A and B are two bigger matrices, n-by-n for example. $mmult()$ has better performance than this one. 'b' means bigger. This function is only used to check errors in factorization and orthogonality of block Lanczos algorithm.

- mdiv
  $R = A/B$

- mplus
  $R = A + B$

- mscal
  $R = \alpha * A$,
  where α is a real constant.

- matnorm
  Compute the norm of a matrix.

- mvmult
  $r = op(A) * x + y$

- symvmult
  $r = A * x + y$,
  where A is a complex symmetric matrix.

- vmult
  $r = x^T * y$ or $r = x^H * y$

- vztplus
  $y = \pm\alpha * x + y$,
  where α is a complex scalor. 'z' means complex. 't' means times.

- vdtplus
  $y = \pm\alpha * x + y$,
  where α is a real scalor. 'd' means real. 't' means times.

- vplus
  $y = x + y$

- vscal
  $r = \alpha * x$,
  where α is a real scalor.

- vecnorm
  Compute the norm of a vector.

- randComplexMat
  Generate a random complex matrix. The real and imaginary part of each element are in uniform (-1,1) distribution.

- randOrthMat
  Generate starting matrix of orthonormal columns by using QR factorizatoin.

- randComplexVec
  Generate a random vector. The real and imaginary part of each element are in uniform (-1,1) distribution.

- randComplexNum
  Generate a random complex number.

Routines implemented in low level computing module are listed below:

- memult
  $R = \alpha * A. * B + C$
  'e' means element.

- vzemult
  $r = x. * y + z$, $r = x. * \bar{y} + z$

- vdemult
  $r = \alpha * x. * y + z$, or $r = \alpha * x. * \bar{y} + z$,
  where α is real number, and x is a real vector

- zzmult
  $\pm a * b + c$ or $\pm a * \bar{b} + c$
  a, b, c are complex numbers.

- dzmult
  $\pm a * b + c$ or $\pm a * \bar{b} + c$
  a is a real number, b and c are complex number.

- zdiv
  $r = a/b$
  a is a complex number, b is a real number.

- conjg
  Conjugate of a complex number.

3.4 Fortran Array

Fortran stores the two-dimensional arrays by columns, in the opposite order from C. We must use a one-dimensional C array of size $m * n$ to store a matrix of size $m * n$. As an example of accessing Fortran-style arrays in C, the following code shows how to allocate memory for double matrix A of size $m * n$ which LAPACK will be expecting, and initialize A so that all of column j has the value j:

```
double *A;
A = (double *)malloc(m * n * sizeof (double));

for (j = 0; j < n; j++) {
    for (i = 0; i < m; i++) {
        A[j * m + i] = j
    }
}
```

Note the loop over the row index i is the inner loop, since column entries are contiguous.

3.5 Important Variables

The project is implemented on the basis of corresponding block Lanczos algorithm [12]. Appendix C.1 C.2 and C.3 are the main source codes of MATLAB implementation. The project strictly follows the procedural of MATLAB implementation. The important variables in the project have the corresponding variables in the MATLAB implementation. Table 3.1 3.2 lists the variables used in block tridiagonalization and tridiagonalization stage respectively. Table 3.3 lists the work buffers used in these

MATLAB Variable	C Variable	Variable Size
A	DoubleComplexMat *A	$n * n$
Q	DoubleComplexMat *Q DoubleComplex *Q_m	$n * n$
M	DoubleComplexMat *M DoubleComplex **M_m	$bs * bs$
B	DoubleComplexMat *B DoubleComplex **B_m	$bs * bs$
W(:,:,:,cur)	DoubleComplexMat *WCur DoubleComplex **WCur_m	$bs * bs$
W(:,:,:,old)	DoubleComplexMat *WOld DoubleComplex **WOld_m	$bs * bs$
R	DoubleComplexMat *R	$n * bs$

Table 3.1: Variables in block tridiagonalization stage in MATLAB and C implementations. n is the order of given matrix A. bs is the block size.

two stages. There is ”$\_m$” in some of the names of variables which means the memory of the matrix or vector.

Some variables need to be further explained. In the following examples which all come from the project, we suppose n is the order of given matrix, bs is the block size. Please note the index to any array is supposed to start from 1 not from 0 like in C language tradition. In the block Lanczos algorithm Q is a matrix of n-by-n. However usually one of its blocks of n-by-bs is involved in computing. Another variable, DoubleComplex $*Q\_m$, represents the pointer to the data of the matrix Q of n-by-n. The following code performs $R = A\bar{Q}_j$ where Q_j means the jth block of Q.

```
Q->m = n;
Q->n = bs;
Q->mat = Q_m + j * bs;
mmult ("+", A, "N", Q, "C", NULL, R);
```

Usually $Q \rightarrow m$ and $Q \rightarrow n$ are set only once. Only $Q \rightarrow mat$ is dynamically set to a new memory address before invoking wrapper routines.

Some variables represent a sequence of matrices. These matrices look like different blocks, for example DoubleComplex $**M\_m$. Variable DoubleComplex $*M$ is responsible for being a representative of one of these blocks to perform computation. Suppose variable DoubleComplexMat $*Q$ is already set. Then the following code computes $R = R - Q_j M_j$:

MATLAB Variable	C Variable	Variable Size
P	DoubleComplexMat *P DoubleComplex *P_m	n
a	DoubleComplexVec *a, DoubleComplex *a_m	n
b	DoubleRealVec *b DoubleReal *b_m	$n-1$
wCur	DoubleComplexVec *wCur DoubleComplex *wCur_m	n
wOld	DoubleComplexVec *wOld DoubleComplex *wOld_m	n
r	DoubleComplexVec *r	n

Table 3.2: Variables in tridiagonalization stage in MATLAB and C implementations. n is the order of given matrix A

Variable	Variable Size
DoubleComplexMat *Work1	$bs * bs$
DoubleComplexMat *Work2	$bs * bs$
DoubleComplexVec *workZ, DoubleComplex *workZ_m	n
DoubleComplexVec *workD DoubleComplex *workD_m	n

Table 3.3: The work buffers in the block Lanczos algorithm.

```
M->m = bs;
M->n = bs;
M->mat = M_m[j];
mmult ("-", Q, "N", M, "N", R, R);
```

There are other variables similar to DoubleComplex $**M\_m$ in the block tridiagonalization stage, like DoubleComplex $**B\_m$, $**WCur\_m$, $**WOld\_m$.

In tridiagonalization stage sometimes one part of vector, or one element of vector is involved in computing. Suppose variables DoubleComplexVec $*a$, $*wCur$ and $*workZ$ represent vectors of size n, variablies DoubleComplex $*a\_m$, $*wCur\_m$ and $*workZ\_m$ are the pointers to the memory of these vectors. Then the following code performs computation which involves element multiplication and addition of vectors, $workZ = workZ + a(2:j-1).*conjg(wCur(2:j-1))$

```
a->vec = &a_m[2];
a->n = j - 2;
wCur->vec = &wCur_m[2];
wCur->n = j - 2;
workZ->vec = &workZ_m[1];
workZ->n = j - 2;
vzemult (a, wCur, "C", workZ, workZ);
```

3.6 Index to Array

In C language standard, the index to an array starts from 0. In MATLAB programming, however, the index is from 1. We found out that LAPACK and BLAS uses a simple technique to make the index start from 1. Conforming to MATLAB implementation of block Lanczos algorithm can reduce the opportunity of introducing new errors and new unnecessary troubles. So we adopt this technique to make the index to the array start from 1.

At the beginning of the procedural to perform the block tridiagonalization, we decrease the memory pointers used in this stage by 1 as the following,

```
--M_m;
--B_m;
--WCur_m;
--WOld_m;
```

Of course before the procedural runs into the next stage to perform tridiagonalization, we increase the pointers:

```
++M_m;
++B_m;
++WCur_m;
++WOld_m;
```

In the procedural to perform tridiagonalization we use the same technique to adjust the memory pointers.

3.7 Memory Operations

The user interface of block Lanczos algorithm, *zcstrd_*(), has already shown the principle of the memory operations in the project. The caller of the function is responsible for providing the memory to store the result. For example user should provide memories for matrix Q and P to store the result of block tridiagonalization and tridiagonalization stage before *zcstrd_*() is invoked. The callee may allocate local work buffers. It is callee's responsibility to release the work buffers before it returns to the caller.

From Table 3.1, 3.2 and 3.3 you can find out that there are lots of memories that need to be allocated and freed. The project should be robust to handle memory operations. we pay much attention on the cases when the program fails to allocate memory for a variable. It can happen after some memories have already been successfully allocated before the failure. We don't invoke C function *exit*() to simply return to the operating system. That would not be the user's expectation. We use the systematic steps to handle this case. Suppose the caller A calls function B to perform some kinds of operations, and function B needs to allocate memories. If function B is successful to allocate memories it needs, then B is responsible for releasing all memories it has allocated before it returns to caller A. If it fails, it should release the already allocated memories before the failure. Function B must guarantee that there is no memory which is not released. Caller A should check the return code from function B.

We also specify that function B doesn't necessarily check if the pointer to the memory, transferred to function B as an argument, is NULL or not, or the size of memory is correct. It is function A's responsibility to guarantee that the pointer is not NULL, and the memory of correct size is allocated. By this strategy function B saves many memory correctness tests, like "$if(ptr == NULL)$". It is one of our methods to achieve as best performance as we can. We don't find memory correctness check in LAPACK either.

We list several code samples in block tridiagonalization stage to show our memory operations. The memory operations in Lanczos tridiagonalization has similar procedural. The following code is to allocate memory for DoubleComplexMat $**M$ in Table 3.1, where n is the order of given matrix, bs is the block size:

```
if (!(M=(DoubleComplexMat *)malloc(sizeof(DoubleComplexMat)))) {
   blkTriExcept("blkTriInit", "Malloc for *M failed");
   return 1;
}
if (!(M_m=(DoubleComplex **)malloc(steps*sizeof(DoubleComplex *)))) {
   blkTriExcept("blkTriInit", "Malloc for **M_m failed");
   return 1;
}
for (i = 0; i < steps; i++) {
   if (!(M_m[i]=(DoubleComplex *)malloc(bs*bs*sizeof(DoubleComplex)))) {
       blkTriExcept("blkTriInit", "Malloc for M_m[i] failed");
```

```
        return 1;
    }
}
```

We use the following code to release DoubleComplexMat $**M$.

```
    free (M);
    M = NULL;
    for (i = 0; i < blks; i++) {
        free (M_m[i]);
        M_m[i] = NULL;
    }
    free (M_m);
    M_m = NULL;
```

They are many memories need to be allocated and freed in the project. We release the memories in the order of their allocation. So we don't lose any memory which needs to be released.

In the above example function $blkTriExcept\ (char\ *pos,\ char\ *msg)$ is called whenever exception is thrown in the block tridiagonalization stage. The memories at this stage is listed in Table 3.1. Function $blkTriExcept()$ releases memories allocated before the exception, and prints in which function the exception was thrown and the reason. The implementation of this function is as the following:

```
void blkTriExcept (char *pos, char *msg) {
    int i;

    printf("%s: %s\n", pos, msg);

    if (A != NULL) {
        free (A);
        A = NULL;
    }

    if (Q != NULL) {
        /* Q->mat is assigned by user instead, So don't free Q->mat*/
        free (Q);
        Q = NULL;
    }

    if (M != NULL) {
```

```
    free (M);
    M = NULL;
}
if (M_m != NULL) {
    for (i = 0; i < steps; i++) {
        if (M_m[i] != NULL) {
            free (M_m[i]);
            M_m[i] = NULL;
        }
    }
    free (M_m);
    M_m = NULL;
}

if (B != NULL) {
    free (B);
    B = NULL;
}
if (B_m != NULL) {
    for (i = 0; i < steps - 1; i++) {
        if (B_m[i] != NULL) {
            free (B_m[i]);
            B_m[i] = NULL;
        }
    }
    free (B_m);
    B_m = NULL;
}

if (WCur != NULL) {
    free (WCur);
    WCur = NULL;
}
if (WCur_m != NULL) {
    for (i = 0; i < steps; i++) {
        if (WCur_m[i] != NULL) {
            free (WCur_m[i]);
            WCur_m[i] = NULL;
        }
    }
    free (WCur_m);
    WCur_m = NULL;
}

if (WOld != NULL) {
    free (WOld);
    WOld = NULL;
}
if (WOld_m != NULL) {
    for (i = 0; i < steps; i++) {
        if (WOld_m[i] != NULL) {
            free (WOld_m[i]);
            WOld_m[i] = NULL;
        }
    }
    free (WOld_m);
    WOld_m = NULL;
}

if (R != NULL) {
    if (R->mat != NULL) {
        free (R->mat);
        R->mat = NULL;
    }
    free (R);
    R = NULL;
}

if (Work1 != NULL) {
    if (Work1->mat != NULL) {
        free (Work1->mat);
        Work1->mat = NULL;
    }
    free (Work1);
    Work1 = NULL;
}
if (Work2 != NULL) {
    if (Work2->mat != NULL) {
        free (Work2->mat);
        Work2->mat = NULL;
        }
        free (Work2);
        Work2 = NULL;
    }
}
```

To allocate, release memory and handle exceptions in the project is so complicated that the functions in two files, blkTriAux.c and lanTriAux.c, are supposed to do nothing but these operations in the block tridiagonalization and tridiatgonalization stage.

3.8 Optimization of LAPACK and BLAS Wrapper

The performance is one requirement of the project. It is mainly determined by the block Lanczos algorithm. On the other hands, some design and implementation strategies also play very important role.

After the implementation is almost done and fully verified, we compared our performance with LAPACK's routine of bidiagonalization of general complex matrices. See Section 3 for the reason. Our implementation is faster than the LAPACK's bidiagonalization routine. But we had the feeling that we could improve the performance even further. We examined the LAPACK and BLAS wrapper implementation and figured out that the wrapper might be a little bit too general. For example function *mmult*(), which performs $R = op(A)op(B) + C$, is frequently called in block Lanczos algorithm. Its performance affects a lot on the efficiency of the whole software. We took care of the case when $op(A) = \bar{A}$. However this case is believed not to happen in block Lanczos algorithm. The test of this case in function *mmult*() has no use but to waste time. Function *mdiv*() is another example. It computes $R = A/B$. Our implementation even checks whether the result of computing is specified to store not in matrix R, but in matrix B, ie $B = A/B$. We found out this case can never happen in the algorithm. We had such similar useless checks when we implemented other functions of the wrapper. We examined and modified every function in the wrapper for this issue.

Function *mmult*() deals with the case $op(B) = \bar{B}$. The LAPACK's function, *zlacgv_*(), which is used to compute the conjugate of matrix B, would store the conjugate of matrix B in B itself. So it is necessary to allocate a memory to keep the original matrix B. This operation is frequently performed. It makes sense to avoid the dynamic memory allocation. We asked ourselves the question about how many memory blocks are common between the wrapper's functions, and what is the size of every such memory. After careful generalization we introduced several global work buffers which are allocated in advance before the computations. The following is definition:

```
/**************************************************************************
 *
 * Global variablies
 *
 *   For the sake of performance, specify work buffers which frequently
 *   involve in computing.
 *     *mat     Work buffer for a matrix. Max size = n * bs
 *     *tau     Work buffer for QR fact.  Max size = n
 *     *work    Work buffer for QR fact.  Max size = n
 *
 **************************************************************************/

 DoubleComplex *mat = NULL;
 DoubleComplex *tau = NULL;
 DoubleComplex *work = NULL;
```

During the initialization of block Lanczos algorithm, a function in the LAPACK and BLAS wrapper, *wrapperInit* (*int n*, *int bs*), is called to allocate memories and assign these global variables the pointers to the allocated memories. At the end of block Lanczos algorithm, we call function $freeWrapper$ () to release these memories. Every function in the wrapper can freely use these three memories provided it doesn't require larger size than the maximum of these memories.

The numerical experiments have shown that we improved the performance of our project greatly by these strategies. We compared the run time of our implementation before and after the optimization. Please note in these experiments we multiplied the matrix Q which is computed in block tridiagonalization stage by matrix P which is computed in tridiagonalization stage to generate a final matrix. The run time demonstrated below includes this procedural. The first example is the run time of tridiagonalization of a random complex symmetric matrix of size 1024 before the optimization:

```
Block Lanczos tridiagonalization: 438.80 seconds
Bidiagonalization: 476.00 seconds
```

Next is the run time of our optimized implementation for another random complex symmetric matrix of size 1024:

```
Block Lanczos tridiagonalization: 399.77 seconds
Bidiagonalization: 482.77 seconds
```

The three auxiliary buffers have specific sizes. This limitation makes the functions of LAPACK and BLAS wrapper not suitable for all general cases. We make comments about the limitation for every function of the wrapper in our implementation. The following comment of function *mmult*() is a typical one:

```
/***************************************************************************
 *
 * Abstract
 *   Compute the multiplication and addition of matrices,
 *      R =  op(A) * op(B) + C,
 *      R = -op(A) * op(B) + C,
 *   BLAS doesn't support conjg(B), we must implement it.
 *
 * Input
 *   signa   "+", op(A) * op(B) + C
 *           "-", -op(A) * op(B) + C
 *   A       Matrix. Size of op(A) = m * k
 *   transa  "N", op(A) = A
 *           "T", op(A) = A.'          nonconjugate transpose.
 *           "H", op(A) = A'           conjugate transpose
 *   B       Matrix. Size of op(B) = k * n
 *   transb  "N", op(B) = B
 *           "T", op(B) = B.'          nonconjugate transpose.
 *           "H", op(B) = B'           conjugate transpose
 *           "C", op(B) = conjg (B)    just conjugate.
 *   C       Matrix. Size = m * n
 *           if C = NULL, then R = (+, -) op(A) * op(B)
 *
 * Output
 *   R       Matrix. Size = m * n.
 *
 * Return
 *   0       Success exit
 *   1       Exception occurs
 *
 * NOTE
 *   (1) The function assumes matrix R is not matrix A or B to
 *       achieve the best performance. That means the result of
 *       computing should be put in C or another matrix.
 *   (2) The function doesn't deal with the case that op(A) = "C"
 *       because block Lanczos method doesn't has this operation.
 *   (3) Only one of the matrices, A, B or C, can be n-by-n size
 *       if one of them involves "T", "H" or "C" operation.
 *       Otherwise call routine bmmult (), where 'b' means big
 *       matrix-matrix operation. If there is no such operations
 *       at all, matrix A and B  can be n-by-n size.
 *
 *****************************************************************************/
```

We also paid attention on some trivial details to get better performance. Function call is time-consuming, so we try to avoid calling a function if possible. For example, the conjugate function in the wrapper is simple, and is found to be used only by function *mmult*(). We added the conjugate function implementation into function *mmult*(), then deleted the conjugate function from the wrapper.

3.9 Code Sample 1

We list some parts of implementation of the project here. The MATLAB implementation of the first example is as the following,

```
R=Tmp-Q(:,qLow:qUp)*M(:,:,j)-Q(:,(qLow-bs):(qUp-bs))*B(:,:,j-1).';
[Q(:,(qLow+bs):(qUp+bs)),B(:,:,j)]=qr(R, 0);
```

Suppose we define DoubleComplexMat Q_pre and Q_next as local variables to represent two blocks of matrix Q, $Q(:,(qLow-bs):(qUp-bs))$ and $Q(:,(qLow+bs):(qUp+bs))$, respectively. The C implementation is as the following:

```
Q->m = n;
Q->n = bs;
Q->mat = Q_m;

M->m = bs;
M->n = bs;
B->m = bs;
B->n = bs;
```

```
Q_pre.m = Q->m;
Q_pre.n = Q->n;
Q_pre.mat = Q->mat - Q->m * bs;
Q_next.m = Q->m;
Q_next.n = Q->n;
Q_next.mat = Q->mat + Q->m * bs;

M->mat = M_m[j];
/* R = R - Q(j) * M(j) - Q(j-1) * B(j-1).'  */
mmult ("-", Q, "N", M, "N", R, R);
B->mat = B_m[j-1];
mmult ("-", &Q_pre, "N", B, "T", R, R);

B->mat = B_m[j];
qr (R, &Q_next, B);
```

3.10 Code Sample 2

MATLAB implementation of the second sample is as the following:

```
W(:,:,k,old) = M(:,:,k)*conj(W(:,:,k,cur)) ...
               - W(:,:,k,cur)*M(:,:,j) ...
               + (eps*0.3)*((B(:,:,1)+B(:,:,2)) ...
               .*(randn(bs,bs)+IM*randn(bs,bs)));
```

Suppose we define DoubleComplexMat $B2$ as a local variable. Then the corresponding C implementation is:

```
M->m = bs;
M->n = bs;
WCur->m = bs;
WCur->n = bs;
WOld->m = bs;
WOld->n = bs;
B->m = bs;
B->n = bs;
B2->m = bs;
B2->n = bs;
Work1->m = bs;
```

```
Work1->n = bs;
Work2->m = bs;
Work2->n = bs;

M->mat = M_m[k];
WCur->mat = WCur_m[k];
WOld->mat = WOld_m[k];
mmult ("+", M, "N", WCur, "C", NULL, WOld);
M->mat = M_m[j];
mmult ("-", WCur, "N", M, "N", WOld, WOld);
B->mat = B_m[1];
B2.mat = B_m[2];
mplus (B, &B2, Work1);
randComplexMat (Work2);
WOld->mat = WOld_m[k];
memult (eps * 0.3, Work1, Work2, WOld, WOld);
```

3.11 Code Sample 3

Next is the MATLAB code in tridiagonalization stage:

```
wOld(2:j-1) = (b(2:j-1).*conj(wCur(3:j)) ...
              + a(2:j-1).*conj(wCur(2:j-1)) ...
              - a(j)*wCur(2:j-1) ...
              + b(1:j-2).*conj(wCur(1:j-2)) ...
              - b(j-1)*wOld(2:j-1))/b(j) ...
              + eps*0.3*(b(2:j-1)+b(j)*ones(j-2,1)) ...
              .*(randn(j-2,1) + IM*randn(j-2,1));
```

To implement this one line of MATLAB code, C seems cumbersome:

```
/* workZ = b(2:j-1) .* conjg(wCur(3:j))
 *         + a(2:j-1) .* conjg(wCur(2:j-1))
 *         - a(j) * wCur(2:j-1)
 *         + b(1:j-2) .* conjg(wCur(1:j-2))
 */
workZ->vec = &workZ_m[1];
workZ->n = j - 2;
b->vec = &b_m[2];
b->n = j - 2;
```

```
wCur->vec = &wCur_m[3];
wCur->n = j - 2;
vdemult (1.0, b, wCur, "C", NULL, workZ);

a->vec = &a_m[2];
a->n = j - 2;
wCur->vec = &wCur_m[2];
vzemult (a, wCur, "C", workZ, workZ);

vztplus ("-", &a_m[j], wCur, workZ, workZ);

b->vec = &b_m[1];
wCur->vec = &wCur_m[1];
vdemult (1.0, b, wCur, "C", workZ, workZ);

/* workZ = workZ - b(j-1)*wOld(2:j-1)    */
wOld->vec = &wOld_m[2];
wOld->n = j - 2;
vdtplus ("-", b_m[j-1], wOld, workZ, workZ);

/* wOld(2:j-1) = workZ / b(j)     */
vscal (1 / b_m[j], workZ, wOld);

/* workD = b(2:j-1)+b(j)*ones(j-2,1) */
workD->vec = &workD_m[1];
workD->n = j - 2;
setRealArray (workD->vec, workD->n, b_m[j]);
b->vec = &b_m[2];
b->n = j - 2;
vplus (b, workD, workD);

/* workZ = randn(j-2,1) + IM*randn(j-2,1) */
randComplexVec (workZ);
/* wOld(2:j-1) = wOld(2:j-1) + eps*0.3*workD.*workZ  */
vdemult (eps*0.3, workD, workZ, "N", wOld, wOld);
```

3.12 Comment Sample

The following is the comment for function *blkTriInit*. There is a notice to describe some important issues:

```
/*****************************************************************************
 *
 * Abstract:
 *   Perform initialization for block tridiagonalization operation.
 *   Allocate necessary memories.
 *
 * Input:
 *   na     Order of matrix A.
 *   A_p    Pointer to the memory of matrix A.
 *   blk    Block size.
 *   S_p    Pointer to the starting matrix S.
 *
 * Output:
 *   a_p    Pointer to the memory of vector a.
 *   b_p    Pointer to the memory of vector b.
 *   Q_p    Pointer to the memory of matrix Q.
 *   P_p    Pointer to the memory of matrix P.
 *
 * Return:
 *   0      Success exit
 *   1      Memory allocation exception occurs
 *
 * NOTE:
 *   There is a little difference in the definition of MIDEPS in
 *   C implementation from the paper and MATLAB code. Matlab code:
 *      SQRTEPS = sqrt(eps)
 *      MIDEPS = sqrt(sqrt(SQRTEPS))^7
 *
 *   Later in the detection of loss of orthogonality,
 *      max(abs(W(:, colW, k, old))) >= SQRTEPS
 *   Supporse w is maximum element of matrix W, then
 *   max(abs(W(:, colW, k, old))) > = SQRTEPS means
 *      sqrt(w.r^2 + w.i^2) >= sqrt(eps)
 *
 *   For the sake of performance, C implementation will be,
 *       MIDEPS = sqrt(sqrt(eps))^7
 *   Supporse w is one element of matrix W, then use the following code
 *   to test if this element loses orthogonality,
 *       w.r^2 + w.i^2 >= eps
 *
 *****************************************************************************/
```

Chapter 4

Verification

The project's last stage, verification, is described in this chapter. First our test plan is present in Section 4.1. Then the verification of LAPACK and BLAS wrapper, and block Lanczos algorithm implementation are present in Section 4.2 and 4.3. The format of the data file, which we used for verification, is present in Section 4.4. In Section 4.5 and 4.6 we present two typical test cases. We compared our block Lanczos tridiagonalization implementation wiht LAPACK's routine of bidiagonalization of complex matrices. Section 4.7 shows the comparison result. The targets the project achieved are described in Section 4.8.

4.1 Test Plan for Block Lanczos Algorithm

We have the following test plans:

- A random complex symmetric matrix generator is developed since the project requires the software to be able to handle any complex symmetric matrix. The generator sets every element of the matrix to be uniformly distributed.
- The starting orthonormal columns matrix S is generated from QR factorization of a random complex n-by-b matrix.
- The error in the orthogonality of Q was measured by:

  $$\|I - Q^H Q\|_{\mathrm{F}}/n^2$$

  The error in the tridiagonalization $T = Q^H A\bar{Q}$ was measured by:

  $$\|Q^H A\bar{Q} - T\|_{\mathrm{F}}/n^2$$

- Compare the performance with LAPACK's routine of bidiagonalization of general complex matrices. We supposed to compare with LAPACK's routine of tridiagonalization of general complex matrices. Unfortunately LAPACK supports only the tridiagonalization of Hermitian matrices, not general complex matrices. With the advice of Professor Qiao, we compared with LAPACK'S routine of bidiagonalization of general complex matrices, $zgebrd_{()}$.
- Test our implementation on the matrices of large size, say, 1024.
- Perform stress testing to determine the ability of the implementation to cope with large matrices or prolonged computing, examine whether there exists memory leak danger.

4.2 Verification of LAPACK and BLAS Wrapper

The accuracy of LAPACK and BLAS wrapper and the low level computing module play important role in the success of the project. We applied three methods to test. One is to compute the error. For example, function $qr()$ computes QR factorization of given matrix A, it generates two matrices, Q and R. We computed $\|QR - A\|_{\mathrm{F}}/n^2$ to test the accuracy of the function. Using MATLAB is another method. We gave MATLAB and our function the same data, then compared their result. Sometimes we applied both the error computing and MATLAB to the full test of a function to make sure that it is correct and robust. Some functions need to handle many cases. The frequently used function *mmult*() computes $R = op(A) * op(B) + C$ or $R = -op(A) * op(B) + C$, where $op(X)$ maybe $X, X^T or X^H$. We developed test programs to go through every path in *mmult*().

4.3 Verification of Block Lanczos Algorithm

When we began to develop the block Lanczos algorithm on the basis of LAPACK and BLAS wrapper and the low level computing module, we faced one problem: how do we know the computing in every step of every iteration in block tridiagonalization and tridiagonalization stage is correct? In the final testing we can check errors in orthogonality and factorization. In the development stage we need a fast and efficient way to locate and correct errors. MATLAB is the big help for this purpose. By the same method as we applied in the verification of LAPACK and BLAS wrapper, we gave MATLAB and our C implementation the same data to compute. We compared the result of every trivial computation made by C implementation with MATLAB implementation. MATLAB helped us efficiently find all errors in the block Lanczos algorithm implementation. We also developed several routines which print out the data of matrices and vectors. After we corrected one error or changed the functionality, both MATLAB and these print routines helped us check the computation more easily and quickly.

4.4 Data File for Developing and Testing

In order to input the data coming from MATLAB into C program, we first stored matrix A, starting matrix S, one random matrix and one random vector generated by MATLAB into a text file. Then we developed a C function to read the text file into the corresponding memories.

We set the configuration of MATLAB so that it prints a matrix or a vector in the command window in "long g" format and compact form. We decreased the size of MATLAB window so that MATLAB can print out any size of matrix and vector in only one column. The following is a typical MATLAB output of a matrix:

```
>> A
A =
  Column 1
         0.0716881986569971 -   0.00767011476577342i
         -0.016811636657866 +    0.0134989080664658i
         ...... ......
         ...... ......
        -0.0991974048156785 +    0.0957366734282219i
        -0.0587775546633594 +    0.0229676758117651i
  Column 2
         -0.016811636657866 +    0.0134989080664658i
         0.0684827583322591 -    0.0106944620780606i
         ...... ......
         ...... ......
        -0.0287528136042179 -   0.00673593437374433i
         0.0373572489045463 +       0.0215571535732i
  Column 3
         ...... ......
         ...... ......
  Column 128
        -0.0587775546633594 +    0.0229676758117651i
         0.0373572489045463 +       0.0215571535732i
         ...... ......
```

```
...... ......
0.0391611913210441 +    0.0271795819142423i
0.0209261831926791 +    0.0396159133305231i
```

The data file which is input into C program has the following format:

```
n bs
matrix A
starting matrix S
random block matrix
random vector
```

The following is a sample data file. The line starting with "%" means it is comment. This sample shows that it requires as least work as possible to edit the data from MATLAB to generate the file. We just changed MATLAB's "Column" to be comment, then put several matrices and vectors together into one file.

```
% Matrix and block size.
15 3

% Matrx A. Size = n * n ------------------------------------------
  % Column 1
         0.0478589394560227 -    0.0428073419593041i
          0.283574922665358 +    0.0264951601674461i
         ...... ......
         ...... ......
         -0.180698010996064 +   0.00319292523110796i
         -0.140536270515727 -      0.21671273149041i
  % Column 2
          0.283574922665358 +    0.0264951601674461i
         -0.102703286196567 -     0.108509759880881i
         ...... ......
         ...... ......
         -0.203457543499649 -    0.0556994850689247i
          0.109460499990275 -     0.204705548917621i
  % Column 3
         ...... ......
         ...... ......
  % Column 15
         -0.140536270515727 -      0.21671273149041i
```

```
          0.109460499990275 -     0.204705548917621i
         ...... ......
         ...... ......
         0.0488464556736896 -    0.0155762855663472i
          0.137928067945508 +    0.0961066192839481i

% Start matrx S. size = n * bs -------------------------------------
  % Column 1
        -0.387614354984968
       -0.0119714591590236
         ...... ......
         ...... ......
        -0.209863793806297
       -0.0665062941010241
  % Column 2
         0.492110768859258
         -0.43587178334902
         ...... ......
         ...... ......
        -0.239240659896391
       -0.0106846213254676
  % Column 3
       -0.0615874188133977
       -0.0461140236170074
         ...... ......
         ...... ......
         0.353201583530759
         0.131974824336085

% Random block matrix. Size = bs * bs ------------------------------
  % Column 1
          0.528743010962225 +     0.591282586924176i
          0.219320672667622 -     0.643595202682526i
         -0.921901624355539 +      0.38033725171391i
  % Column 2
          -2.17067449430526 -      1.00911552434079i
        -0.0591878245211912 -    0.0195106695302893i
          -1.01063370647425 -    0.0482207891453123i
  % Column 3
          0.614463048895481 +  4.3191841625545e-005i
          0.507740785341986 -     0.317859451247688i
           1.69242987019052 +      1.09500373878749i

% Random Vector. Size = n * 1 -------------------------------------
         -0.432564811528221 +      0.11393131352081i
           -1.6655843782381 +      1.06676821135919i
         ...... ......
         ...... ......
            2.1831858181971 -      1.44096443190102i
         -0.136395883086596 +     0.571147623658178i
```

4.5 Test Case 1

The data file shown in Section **??** was used from the beginning of block Lanczos algorithm implementation after LAPACK and BLAS wrapper and low level computing module were finished. The matrix A in the data file is deliberately specified to be 15-by-15, the block size is specified to be 3. Matrix A is not big, but it is enough to help us go through every path of the algorithm to find all possible errors. This data file played a very important role in the completeness of the project. The following is output of our C implementation on this data file:

```
==================== Block Lanczos Algorithm ====================

Order of matrix A = 15. Block size = 3.

Matrix A
 0.0478589395 -0.0428073420i ... -0.1405362705 -0.2167127315i
 0.2835749227 +0.0264951602i ...  0.1094605000 -0.2047055489i
 0.2126834343 +0.1010314229i ...  0.0728396097 -0.0321435091i
 0.0340301898 -0.1170237713i ...  0.1198303864 +0.2465204200i
 ...... ......
 ...... ......
 0.0930588095 -0.0108819143i ... -0.0638638363 +0.0725416028i
 0.1279936798 -0.0006970399i ...  0.0467403617 -0.1217547535i
-0.1806980110 +0.0031929252i ...  0.0488464557 -0.0155762856i
-0.1405362705 -0.2167127315i ...  0.1379280679 +0.0961066193i
```

```
Starting matrix S
-0.3876143550 +0.0000000000i ... -0.0615874188 +0.0000000000i
-0.0119714592 +0.0000000000i ... -0.0461140236 +0.0000000000i
-0.2314101541 +0.0000000000i ... -0.2587480752 +0.0000000000i
-0.0856510685 +0.0000000000i ... -0.1363600296 +0.0000000000i
 ...... ......
 ...... ......
-0.1492539258 +0.0000000000i ... -0.1036389505 +0.0000000000i
-0.1948120134 +0.0000000000i ...  0.3645636182 +0.0000000000i
-0.2098637938 +0.0000000000i ...  0.3532015835 +0.0000000000i
-0.0665062941 +0.0000000000i ...  0.1319748243 +0.0000000000i

==================== BlkTri Iteration #4 ====================

Matrix Q block #5
-0.0237943123 -0.0230355910i ...  0.2144107268 -0.1121143141i
-0.4447218553 +0.0205433687i ...  0.0753316986 -0.2005011249i
 0.0346898896 -0.0462275251i ... -0.1301028749 -0.1127984841i
-0.0857878112 +0.0762725490i ...  0.3313906096 -0.1729874419i
 ...... ......
 ...... ......
 0.1172680257 -0.0649908712i ... -0.1971375633 +0.1700256952i
 0.1371619099 -0.3726888987i ... -0.2716242849 +0.1334174181i
 0.0819607628 +0.1199584190i ...  0.1380740139 -0.0654531821i
 0.0863491842 -0.0725421391i ... -0.0503035860 +0.1019349581i

Matrix M block #4
 0.3114919166 +0.1436544414i ...  0.2231779007 +0.0286534411i
 0.0644497831 -0.0056412862i ...  0.0577689467 -0.0712097797i
 0.2231779007 +0.0286534411i ...  0.0074985421 -0.2149166194i

Matrix B block #4
 0.3521671481 +0.0000000000i ... -0.2174270369 -0.0977944839i
 0.0000000000 +0.0000000000i ... -0.0759708992 +0.0387181555i
 0.0000000000 +0.0000000000i ...  0.2608633275 +0.0000000000i

==================== LanTri Iteration #14 ====================

Matrix P column #15
```

```
-0.2384590400 -0.3009164102i  -0.0954178108 -0.1969373769i
-0.0737626662 -0.0585505882i   0.1968172044 -0.1068801589i
 0.1706240239 +0.1771808314i   0.0396924044 +0.2207635223i
-0.2979288003 +0.1057655403i  -0.2936485570 +0.0351849059i
 ...... ......
 ...... ......
-0.0428570719 -0.0254166999i   0.2347595726 +0.1999805341i
 0.3400777653 +0.0472979508i  -0.1825663841 -0.0854940114i
 0.0666418432 +0.0904082649i   0.0608288089 +0.2285311562i
 0.1151987079 -0.3309174602i

Vector a
 0.4132803969 -0.1607432363i  -0.2191385078 -0.1845246417i
 0.1738547341 -0.0359881760i  -0.0621378994 -0.0384280844i
-0.0884579866 +0.0802786025i  -0.0806427002 +0.1349532193i
 0.1383766603 +0.2654783824i   0.0955196537 +0.2948483760i
 ...... ......
 ...... ......
 0.0790878959 +0.0115411736i   0.3767681478 +0.0248356685i
-0.0653626625 -0.3457474798i  -0.1716503934 -0.4603941507i
-0.0823728291 -0.3353493908i   0.2536401538 -0.5080346759i

Vector b
 0.6557260600   0.3122593127   0.5351583737   0.4362215701
 0.5870279458   0.3991668196   0.3685844057   0.4496510986
 0.4384279361   0.2769169967   0.3888436648   0.3983887749
 0.3131608130   0.3442559780

Block Lanczos tridiagonalization:  0.01 seconds

Error in orthogonality: 2.364e-016
Error in factorization: 1.770e-016

=================== LAPACK Bidiagonalization ===================

Bidiagonalization:  0.01 seconds

Error in orthogonality of Q: 1.079e-017
Error in orthogonality of P: 1.098e-017
Error in factorization: 7.490e-018
```

Next is the output of MATLAB implementation. It is clear that our C implementation of block Lanczos algorithm has exactly the same computation as MATLAB implementation.

```
 ==================== Block Lanczos Algorithm (MATLAB) ====================

Order of matrix A = 15, block size = 3

Matrix A:
 0.04785893945602 -0.04280734195930i...-0.18069801099606 +0.00319292523111i
 0.28357492266536 +0.02649516016745i...-0.20345754349965 -0.05569948506892i
 0.21268343429027 +0.10103142293987i... 0.17211543844081 -0.05425759812987i
 0.03403018976937 -0.11702377129074i... 0.08657862532985 +0.09501768369865i
 ...... ......
 ...... ......
 0.09305880952240 -0.01088191434569i...-0.05762133793361 +0.29714182101118i
 0.12799367975477 -0.00069703991728i... 0.22165793911916 -0.16196028035689i
-0.18069801099606 +0.00319292523111i... 0.01462084819687 -0.01229791026297i
-0.14053627051573 -0.21671273149041i... 0.04884645567369 -0.01557628556635i

Starting Matrix S:
-0.38761435498497 +0.00000000000000i... 0.49211076885926 +0.00000000000000i
-0.01197145915902 +0.00000000000000i...-0.43587178334902 +0.00000000000000i
-0.23141015405530 +0.00000000000000i...-0.25663699776637 +0.00000000000000i
-0.08565106851322 +0.00000000000000i...-0.20604272597600 +0.00000000000000i
 ...... ......
 ...... ......
-0.14925392576933 +0.00000000000000i... 0.03791128365443 +0.00000000000000i
-0.19481201342857 +0.00000000000000i...-0.27609893979653 +0.00000000000000i
-0.20986379380630 +0.00000000000000i...-0.23924065989639 +0.00000000000000i
-0.06650629410102 +0.00000000000000i...-0.01068462132547 +0.00000000000000i

============================== BlkLan iter #4 ==============================

Matrix Q block #5
-0.02379431234652 -0.02303559100384i... 0.07279969475170 -0.17329927703256i
-0.44472185530834 +0.02054336871827i...-0.16139888159055 -0.02083455626194i
 0.03468988955630 -0.04622752512405i...-0.33568530829753 +0.00592836963793i
```

```
-0.08578781119378 +0.07627254900486i... 0.39549556149190 +0.42236929197806i
 ...... ......
 ...... ......
 0.11726802574352 -0.06499087119617i... 0.19891528042628 -0.16570248772914i
 0.13716190987916 -0.37268889873881i... 0.16017101854865 +0.21437164588139i
 0.08196076279288 +0.11995841899699i... 0.00159121810166 -0.07466791161679i
 0.08634918416893 -0.07254213910534i... 0.08836184601459 +0.16134487613958i

Matrix M block #4
 0.31149191661281 +0.14365444135118i... 0.06444978313071 -0.00564128622986i
 0.06444978313071 -0.00564128622986i...-0.01497473580488 -0.39712600214300i
 0.22317790070853 +0.02865344108820i... 0.05776894668822 -0.07120977972422i

Matrix B block #4
 0.35216714806776 +0.00000000000000i... 0.16510896897047 +0.08466700528600i
 0.00000000000000 +0.00000000000000i... 0.29954496787334 +0.00000000000000i
 0.00000000000000 +0.00000000000000i... 0.00000000000000 +0.00000000000000i

============================ LanTri iter #14 ============================

Matrix P column #15
-0.23845904002821 -0.30091641020487i  -0.09541781082930 -0.19693737694301i
-0.07376266620061 -0.05855058819109i   0.19681720438358 -0.10688015888984i
 0.17062402386220 +0.17718083137892i   0.03969240438029 +0.22076352225411i
-0.29792880034058 +0.10576554031181i  -0.29364855702353 +0.03518490586966i
-0.04285707188984 -0.02541669993110i   0.23475957259174 +0.19998053407358i
 0.34007776529120 +0.04729795078215i  -0.18256638413998 -0.08549401137922i
 0.06664184320247 +0.09040826488977i   0.06082880886547 +0.22853115622808i
 0.11519870787504 -0.33091746024892i

Vector a
 0.41328039687186 -0.16074323630622i  -0.21913850780044 -0.18452464169751i
 0.17385473405649 -0.03598817600496i  -0.06213789941431 -0.03842808438983i
-0.08845798661262 +0.08027860245079i  -0.08064270022562 +0.13495321932444i
 0.13837666031634 +0.26547838236782i   0.09551965368060 +0.29484837601359i
 0.07908789588384 +0.01154117362022i   0.37676814783093 +0.02483566852594i
-0.06536266250309 -0.34574747975281i  -0.17165039344675 -0.46039415067483i
-0.08237282908419 -0.33534939080096i   0.25364015384886 -0.50803467587901i

Vector b
```

```
 0.65572606003296   0.31225931266052   0.53515837366451   0.43622157009151
 0.58702794578677   0.39916681957557   0.36858440573782   0.44965109860803
 0.43842793611461   0.27691699670491   0.38884366483228   0.39838877493217
 0.31316081300382   0.34425597796063

Error in orthogonality = 7.31e-016
Error in factorization = 6.76e-016
```

4.6 Test Case 2

In this section we present another example to show that both C and MATLAB implementation have the same computation. The matrix A is 256-by-256, and block size is specified as 8. The following is the content of the data file.

```
% Matrix and block size.
256 8

% A. Size = n * n
  % Column 1
         0.0312026273811561 -    0.0277405655728896i
        -0.0317817413376722 +    0.0395099713223415i
        ...... ......
        ...... ......
       -0.00956079401462568 +   0.00789967560148603i
        0.00916347590916092 -   0.00763644962180457i
  % Column 2
        -0.0317817413376722 +    0.0395099713223415i
        -0.0175922919556229 +    0.0194548558352152i
         ...... ......
         ...... ......
         0.0338716739966944 +    0.0181251095130932i
         0.0585077801039869 -    0.0501774347262309i
  % Column 3
         ...... ......
         ...... ......
  % Column 256
        0.00916347590916092 -   0.00763644962180457i
         0.0585077801039869 -    0.0501774347262309i
         ...... ......
         ...... ......
         0.0334029993869039 -    0.0422870261996952i
        -0.0103104449515434 +    0.0691266791184281i

% S. size = n * bs
  % Column 1
         -0.02300868931295
      -0.00693418671303187
        ...... ......
       -0.0890764745322856
       -0.0139155303592929
  % Column 2
        0.0256088915957461
       -0.0412761315010268
        ...... ......
        ...... ......
       -0.0698434042045308
       -0.0548597794552349
  % Column 3
        ...... ......
        ...... ......
  % Column 8
        0.0328690052645199
        0.0581720559916915
        ...... ......
        ...... ......
        0.0544384942394926
       -0.0147017582092159

% RandMat. Size = bs * bs
  % Column 1
        0.465214197711276 +     0.327334626514537i
        0.402821302367635 +     0.206260223527035i
        ...... ......
        ...... ......
          0.2129783190453 +     0.235910570652193i
         0.61452401389579 +     0.744343321451221i
 % Column 2
```

```
       0.639227876381669 +     0.355846858016115i
       0.453382407108533 +     0.246128622853403i
       ...... ......
       ...... ......
        0.29726598873376 +       0.4614816018226i
       0.938337127933372 +     0.605022500009319i
% Column 3
       ...... ......
       ...... ......
% Column 8
       0.323728455810956 +     0.178763140465727i
       0.946928018208132 +       0.3269335623422i
       ...... ......
       ...... ......
      0.0682468105769109 +    0.0890911004036379i
       0.864204556312917 +     0.577133836408033i

 % randVec. Size = n * 1
       0.984370873595866 +     0.833936168042149i
        0.60502217897123 +    0.0382491752681705i
        ...... ......
        ...... ......
        0.84288173280989 +     0.856360969833307i
       0.516026572249778 +     0.239707840407361i
```

The following is the computation of C implementation:

```
=================== Block Lanczos Algorithm ===================

Order of matrix A = 256. Block size = 8.

Matrix A
 0.0312026274 -0.0277405656i ...  0.0091634759 -0.0076364496i
-0.0317817413 +0.0395099713i ...  0.0585077801 -0.0501774347i
 0.0542177053 +0.0367214293i ...  0.0092103173 -0.0393992628i
-0.0119076565 +0.0041115583i ... -0.0139858036 -0.0257877252i
 ...... ......
 ...... ......
 0.0147562430 +0.0017638672i ... -0.0340190268 -0.0767703354i
 0.0358197067 +0.0047915540i ...  0.0393892014 -0.0043480459i
-0.0095607940 +0.0078996756i ...  0.0334029994 -0.0422870262i
 0.0091634759 -0.0076364496i ... -0.0103104450 +0.0691266791i

Starting matrix S
-0.0230086893 +0.0000000000i ...  0.0328690053 +0.0000000000i
-0.0069341867 +0.0000000000i ...  0.0581720560 +0.0000000000i
-0.0762014162 +0.0000000000i ...  0.0134324304 +0.0000000000i
-0.0198493799 +0.0000000000i ... -0.0782806744 +0.0000000000i
 ...... ......
 ...... ......
-0.0549302269 +0.0000000000i ...  0.0842830277 +0.0000000000i
-0.0194649000 +0.0000000000i ...  0.0610104961 +0.0000000000i
-0.0890764745 +0.0000000000i ...  0.0544384942 +0.0000000000i
-0.0139155304 +0.0000000000i ... -0.0147017582 +0.0000000000i

==================== BlkTri Iteration #31 ====================

Matrix Q block #32
-0.0367529915 +0.0118624123i ... -0.0106191623 +0.0578109763i
-0.0157152654 +0.0443186412i ... -0.0425620476 -0.0512112398i
 0.0191917817 +0.0232078928i ... -0.0529285443 +0.0233952072i
-0.0048907069 -0.0204786374i ... -0.0448977035 -0.0231779171i
 ...... ......
 ...... ......
-0.0034118633 +0.0216172700i ...  0.0723035285 -0.0196731434i
 0.0053284497 -0.0254471937i ...  0.0147565666 +0.0193208947i
 0.0461914204 -0.0110963994i ... -0.0221963302 -0.0504245577i
-0.0487913557 +0.0234821221i ...  0.0100788353 +0.0373870456i

Matrix M block #31
-0.0322006397 +0.0240044678i ... -0.0041488165 +0.0083286961i
-0.0169865395 -0.0200360229i ...  0.0083210202 -0.0164958005i
 0.0240500315 +0.0124625461i ... -0.0181730052 +0.0143416164i
-0.0105348760 +0.0107974974i ...  0.0046209777 +0.0176032251i

Matrix B block #31
-0.1334418091 +0.0000000000i ... -0.0116578014 +0.0028999693i
 0.0000000000 +0.0000000000i ... -0.0104809789 -0.0237587636i
 0.0000000000 +0.0000000000i ...  0.0210302696 -0.0338663519i
 0.0000000000 +0.0000000000i ... -0.0465052624 +0.0524685621i

===================== LanTri Iteration #255 =====================

Matrix P column #256
 0.0044883997 -0.0321760159i   0.0226965581 -0.0328469249i
-0.0465114062 -0.0578798600i   0.0100612894 +0.0073166057i
 0.0088722111 +0.0045145654i   0.0205656249 +0.0120150000i
 0.0178407619 -0.0402421794i   0.0673424290 -0.0562659985i
 ...... ......
 ...... ......
-0.0247010270 -0.1277335779i   0.1285968206 +0.0831373779i
 0.0189975284 -0.0831392346i   0.0698593469 -0.0151709710i
-0.0653907993 +0.0316690225i   0.2527139619 -0.1395290052i
-0.2152443200 +0.0654564457i   0.1874963574 +0.0492414659i

Vector a
 0.1254847031 -0.0009226228i  -0.0530484165 -0.0246746118i
 0.0328794864 -0.0184175065i  -0.0327814264 -0.0483311172i
-0.0504806317 +0.0074421816i   0.0116692834 -0.0046485868i
-0.0542514500 +0.0000087815i   0.0048199762 +0.0330069680i
 ...... ......
 ...... ......
 0.1234371497 +0.1599291800i  -0.0511678042 +0.0331034644i
-0.0471556243 -0.0601698617i   0.0533264394 -0.0591544711i
 0.0033431951 -0.0016256022i   0.0075518086 -0.0133600886i
-0.0534723062 +0.0181474936i   0.0621931972 +0.0260588735i

Vector b
 0.6141465790   0.5555112757   0.4593802073   0.4736957027
 0.5166386590   0.4944338974   0.4954400789   0.4930741077
 0.4894921335   0.5158897457   0.4730315743   0.5113437820
 0.4963722428   0.4794938214   0.5113726439   0.4725432549
 ...... ......
 ...... ......
 0.1900443994   0.1215158865   0.1642722445   0.0976237551
 0.1149252681   0.1423279784   0.0812899000   0.0972691640
 0.0893512289   0.1264892956   0.1509597134   0.0513350618
 0.0330792103   0.0264689137   0.0251211572   0.0053627372
 Block Lanczos tridiagonalization:  5.66 seconds

 Error in orthogonality: 1.300e-013
 Error in factorization: 1.162e-013

 =================== LAPACK Bidiagonalization ===================

 Bidiagonalization: 14.09 seconds

 Run time comparison: Block Lanczos/Bidiagonalization =  40.2%
 Error in orthogonality of Q: 3.740e-019
 Error in orthogonality of P: 3.805e-019
 Error in factorization: 2.788e-019
```

The output of MATLAB implemenation is as the following:

```
==================== Block Lanczos Algorithm (MATLAB) ====================

Order of matrix A = 256, block size = 8

Matrix A:
 0.03120262738116 -0.02774056557289i...-0.00956079401463 +0.00789967560149i
-0.03178174133767 +0.03950997132234i... 0.03387167399669 +0.01812510951309i
 0.05421770534143 +0.03672142927183i...-0.01335588861391 +0.00450656448502i
-0.01190765646822 +0.00411155832224i...-0.03044945264622 -0.00088589742111i
 ...... ......
 ...... ......
 0.01475624303577 +0.00176386717331i...-0.02466496544510 +0.00081845524647i
 0.03581970672308 +0.00479155395313i...-0.02664679883110 -0.01569299287080i
-0.00956079401463 +0.00789967560149i... 0.04406692139138 +0.00733241251491i
 0.00916347590916 -0.00763644962180i... 0.03340299938690 -0.04228702619970i

Starting Matrix S:
-0.02300868931295 +0.00000000000000i... 0.12623543839259 +0.00000000000000i
-0.00693418671303 +0.00000000000000i... 0.05572760631818 +0.00000000000000i
-0.07620141622993 +0.00000000000000i... 0.06016153644789 +0.00000000000000i
-0.01984937988051 +0.00000000000000i... 0.00480904747401 +0.00000000000000i
```

```
 ...... ......
 ...... ......
-0.05493022688580 +0.00000000000000i... 0.06295294994872 +0.00000000000000i
-0.01946489996336 +0.00000000000000i... 0.10315374305325 +0.00000000000000i
-0.08907647453229 +0.00000000000000i...-0.04481759111243 +0.00000000000000i
-0.01391553035929 +0.00000000000000i...-0.03559041983095 +0.00000000000000i

============================= BlkLan iter #31 =============================

Matrix Q block #32
-0.03675299151360 +0.01186241230621i... 0.03226939120892 +0.05664427099199i
-0.01571526536330 +0.04431864121262i... 0.02372888727725 +0.05268720213871i
 0.01919178170733 +0.02320789278627i...-0.04504448669993 -0.02044385364875i
-0.00489070687917 -0.02047863742156i...-0.08959896745769 +0.02279466980650i
 ...... ......
 ...... ......
-0.00341186330521 +0.02161726999385i... 0.07604634833103 -0.04244922102971i
 0.00532844974443 -0.02544719365573i...-0.00746866237971 -0.01765106030741i
 0.04619142043060 -0.01109639938237i...-0.01876130613818 +0.01461011488604i
-0.04879135565755 +0.02348212208675i... 0.06951558668193 -0.01120721292890i

Matrix M block #31
-0.03220063966021 +0.02400446778255i... 0.00110614780808 -0.00053369979192i
-0.01698653945203 -0.02003602294069i...-0.01664071155632 +0.02505836213542i
 0.02405003153725 +0.01246254606456i...-0.02201679897366 -0.00532312081674i
-0.01053487595451 +0.01079749739880i... 0.00612400500969 -0.01318830528272i

Matrix B block #31
-0.13344180907058 +0.00000000000000i... 0.03762344668029 -0.00045770686708i
 0.00000000000000 +0.00000000000000i... 0.02210499374265 +0.00019261378627i
 0.00000000000000 +0.00000000000000i... 0.00570340109414 -0.04154561528219i
 0.00000000000000 +0.00000000000000i...-0.05987770212611 +0.01602531098780i

============================= LanTri iter #255 ============================

Matrix P column #256
 0.00448839972115 -0.03217601588943i   0.02269655814886 -0.03284692489117i
-0.04651140617880 -0.05787985999630i   0.01006128942190 +0.00731660571955i
 0.00887221112310 +0.00451456542510i   0.02056562491731 +0.01201500001683i
 0.01784076189626 -0.04024217940410i   0.06734242900742 -0.05626599848490i
 ...... ......
 ...... ......
-0.02470102705042 -0.12773357790423i   0.12859682058425 +0.08313737792711i
 0.01899752838514 -0.08313923457857i   0.06985934691182 -0.01517097101719i
-0.06539079928162 +0.03166902245374i   0.25271396190746 -0.13952900517727i
-0.21524431999132 +0.06545644572476i   0.18749635744771 +0.04924146585988i

Vector a
 0.12548470313001 -0.00092262284008i  -0.05304841647537 -0.02467461182453i
 0.03287948639787 -0.01841750649283i  -0.03278142644982 -0.04833111716616i
-0.05048063168846 +0.00744218161159i   0.01166928335368 -0.00464858681930i
-0.05425144998647 +0.00008781145896i   0.00481997617850 +0.03300696801871i
 ...... ......
 ...... ......
 0.12343714966975 +0.15992918004993i  -0.05116780420383 +0.03310346438776i
-0.04715562432670 -0.06016986170335i   0.05332643944328 -0.05915447113377i
 0.00334319507784 -0.00162560224557i   0.00755180859916 -0.01336008857342i
-0.05347230624802 +0.01814749363475i   0.06219319717875 +0.02605887354005i

Vector b
 0.61414657900984   0.55551127566723   0.45938020729813   0.47369570272906
 0.51663865903613   0.49443389740628   0.49544007893108   0.49307410765742
 0.48949213352516   0.51588974574209   0.47303157430025   0.51134378195326
 0.49637224283247   0.47949382143015   0.51137264388339   0.47254325487993
 ...... ......
 ...... ......
 0.19004439941232   0.12151588653826   0.16427224452253   0.09762375512978
 0.11492526813316   0.14232797835112   0.08128990001094   0.09726916401558
 0.08935122894726   0.12648929561624   0.15095971335330   0.05133506181854
 0.03307921026819   0.02646891367048   0.02512115720075   0.00536273716332

Error in orthogonality = 2.99e-013
Error in factorization = 2.54e-013
```

4.7 Performance Comparison

We compared the performance of block Lanczos tridiagonalization implementation with LAPACK's routine of bidiagonalization of general complex matrices, *zgebrd_*().

*zgebrd_*() doesn't support the bidiagonalization of complex symmmetric matrices. It treats complex symmetric matrices as general complex matrices to perform bidiagonalization. The first comparison was made on complex symmetric matrix of 1024-by-1024. The block size is 4.

```
D:\Project\Debug>blklanapp 1024 4

=================== Block Lanczos Algorithm ===================

Block Lanczos tridiagonalization: 231.87 seconds

Error in orthogonality: 9.185e-014
Error in factorization: 4.142e-012

=================== LAPACK Bidiagonalization ===================

Bidiagonalization: 539.33 seconds

Run time comparison: Block Lanczos/Bidiagonalization =  43.0%

Error in orthogonality of Q: 9.190e-020
Error in orthogonality of P: 9.120e-020
Error in factorization: 3.051e-018
```

Next is another comparison on complex symmetric matrix of size 1280. The block size is 4.

```
D:\Project\Debug>blklanapp 1280 4

=================== Block Lanczos Algorithm ===================

Block Lanczos tridiagonalization: 448.00 seconds

Error in orthogonality: 7.774e-014
Error in factorization: 2.737e-012

=================== LAPACK Bidiagonalization ===================
Bidiagonalization: 1042.34 seconds

Run time comparison: Block Lanczos/Bidiagonalization =  43.0%

Error in orthogonality of Q: 8.979e-020
Error in orthogonality of P: 9.075e-020
Error in factorization: 3.024e-018
```

4.8 Overall Success and Achieved Targets

The block Lanczos tridiagonalization of complex symmetric matrices C package achieved the accuracy and performance requirements of the project. Table 4.1 lists the result of numerical experiment on accuracy. Figure 4.1 shows the performance comparison with LAPACK's bidiagonalization routine.

matrix order	block size	error in orthogonality	error in factorization
512	4	$4.750E-013$	$1.612E-011$
1024	4	$9.185E-014$	$4.142E-012$
1024	8	$7.473E-014$	$3.334E-012$
1024	16	$8.338E-014$	$3.891E-012$
1280	4	$7.774E-014$	$2.737E-012$

Table 4.1: Accuracy of block Lanczos tridiagonalization implementation.

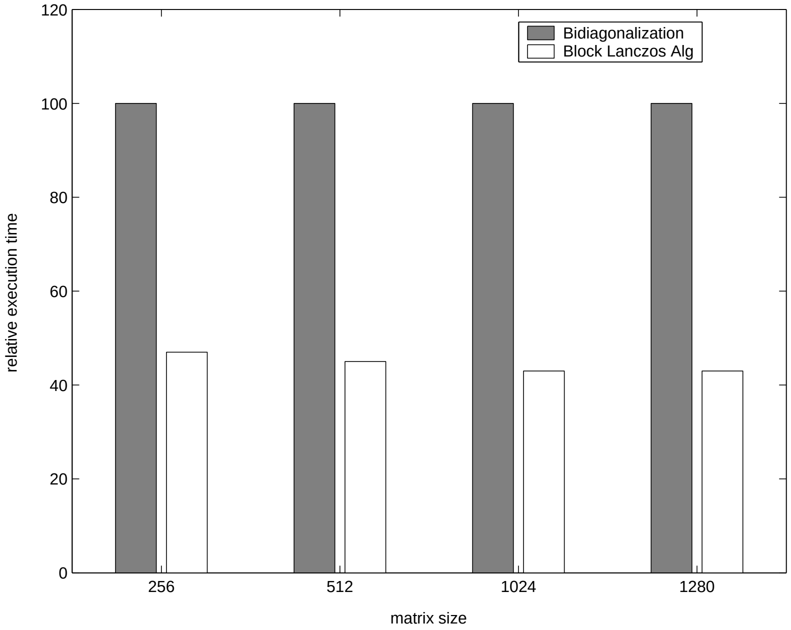

Figure 4.1: Comparison of the efficiency of block Lanczos tridiagonalization implementation. The y axis shows the execution times normalized to the execution time of LAPACK's bidiagonalization of complex matrices

Appendix A

Compile Preprocessor Definitions

Two preprocessor definitions are listed as the following:

- INFO
  Information at the midpoint of iterations in block tridiagonalization stage and tridiagonalization stage is printed out. Section 4.5, 4.6 are the samples of output of the executive program made by this preprocessor definition.

  Run command: blklanapp *n* *bs*
  n: the order of matrix A.
  bs: the block size. *n* should be divisible by *bs*.

- MATLAB_COMP
  Some specific data, which is needed in block Lanczos alogrithm like matrix A and starting matrix S, etc., come from a data file. The data file is edited from the output of MATLAB. Section **??** describes the format of the file.

  Run command: blklanapp datafile
  datafile: the name of the data file.

Appendix B

User Manual of Package

The manual follows LAPACK documentation style.

```
int zcstrd_ (integer *n, doublecomplex *A,
             integer *bs, doublecomplex *S,
             doublecomplex *a, doublereal *b,
             doublecomplex *Q, doublecomplex *P,
             integer *info)
```

n	(input) integer * The order of the matrix A. $n >= 0$.
A	(input) double complex array, dimension (n, n) The complex symmetric matrix A.
bs	(input) integer * The order of one block. $bs >= 0$.
S	(input) double complex array, dimension (n, bs) The starting matrix of orthonormal columns.
a	(output) double complex array, dimension (n) The diagonal elements of the tridiagonal matrix T.
b	(output) double real array, dimension $(n-1)$ The off-diagonal elements of the tridiagonal matrix T.
Q	(output) double complex array, dimension $(n * n)$

It is computed in block tridiagonalization stag.
$(QP)^{\mathrm{H}}A(\bar{QP}) = T$

P (output) double complex array, dimension $(n * n)$
It is computed in tridiagonalization stag.
$(QP)^{\mathrm{H}}A(\bar{QP}) = T$

info (output) integer *
$= 0$: successful exit
< 0: if info = -i, the ith argument had an illegal value.
> 0: Exception is thrown.

Appendix C

MATLAB Implementation

C.1 Block Tridiagonalization

```
function [M,B,Q,steps,nVec] = BlkTri(A,R,steps)
% [M,B,Q,steps,nVec] = BlkTri(A,R,steps)
%
% Lanczos block tridiagonalization of a complex-symmetric matrix.
% Componentwise orthogonalization scheme is applied.
%
% Input
%   A     complex-symmetric matrix.
%   R     starting matrix of orthonormal columns, assuming the
%         number of columns is the block size.
%   steps number of iterations
% Output
%   M     3-d array, M(:,:,i) are the main diagonal blocks of the
%         resulting  block tridiagonal
%   B     3-d array, B(:,:,i) are the subdiagonal blocks of the
%         resulting block tridiagonal
%   Q     unitary
%   steps number of iterations actually run
%   nVec  number of vectors selected for reorthogonalization
% so that
%            A = Q*J*Q.'
% where J is block tridiagonal whose main diagonal blocks are
% M(:,:,i) and subdiagonal blocks are B(:,:,i).
%
```

```
[n,bs] = size(R);               % get the size of the starting matrix
if steps > n/bs
    steps = n/bs;
end
%
% constants
IM = sqrt(-1); SQRTEPS = sqrt(eps);
MIDEPS = sqrt(sqrt(SQRTEPS))^7; % between sqrt(eps) and eps
%
Q(:,1:bs) = R;                  % initial block in Q
M = zeros(bs,bs,steps);         % initialize M and B
B = zeros(bs,bs,steps-1);
W = zeros(bs,bs,steps,2);       % orthogonality estimates
%
up = 0;                         % upper bound
doOrtho = 0;                    % if do orthogonalization
second = 0;                     % if this is the second ortho
nBlk = 0;                       % number of blocks for reortho
%
% start the first block Lanczos iteration
cur = 1;                        % indices of W for the recursion
old = 2;
qLow = 1;                       % initial low and up indices of Q
qUp = bs; Tmp = A*conj(Q(:,qLow:qUp));
M(:,:,1) = Q(:,qLow:qUp)'*Tmp;               %M(1)=Q(1)'*A*conj(Q(1))
R = Tmp - Q(:,qLow:qUp)*M(:,:,1);            %R(1)=A*conj(Q(1))-Q(1)M(1)
[Q(:,(qLow+bs):(qUp+bs)),B(:,:,1)]=qr(R, 0); %QR. Q(2)B(1)=R
W(:,:,1,cur) = (eps*bs*0.6*B(:,:,1) ...      %compute W(:,:,1,2)
                .*(randn(bs,bs)+IM*randn(bs,bs)))/B(:,:,1);
%
% following block Lanczos iterations
cur = 2; old = 1; for j = 2:steps-1
    qLow = qLow + bs;           % update low and up indices of Q
    qUp = qUp + bs;
    Tmp = A*conj(Q(:,qLow:qUp));
    M(:,:,j) = Q(:,qLow:qUp)'*Tmp;
    R = Tmp - Q(:,qLow:qUp)*M(:,:,j) ...
        - Q(:,(qLow-bs):(qUp-bs))*B(:,:,j-1).';
    [Q(:,(qLow+bs):(qUp+bs)), B(:,:,j)] = qr(R, 0);
%
```

```
    % block orthogonalization
    if (second == 0)            % not second orthogonalization
        k = 1;
        while ((k <= j) & (doOrtho ~= 1))   %compute W(k,j+1),k=1...j
            if (k == j)
                W(:,:,k,old) = (eps*bs*0.6)*(B(:,:,1) ...
                                .*(randn(bs,bs)+IM*randn(bs,bs)));
            else % k<j
                W(:,:,k,old) = M(:,:,k)*conj(W(:,:,k,cur)) ...
                               - W(:,:,k,cur)*M(:,:,j) ...
                               + (eps*0.3)*((B(:,:,1)+B(:,:,2)) ...
                               .*(randn(bs,bs)+IM*randn(bs,bs)));
                if (j > 2)
                    if (k > 1)
                        W(:,:,k,old) = W(:,:,k,old) ...
                                    + B(:,:,k-1)*conj(W(:,:,k-1,cur));
                    end % if (k>1)
                    if (k < j-1)
                        W(:,:,k,old) = W(:,:,k,old) ...
                              + ((B(:,:,k).')*conj(W(:,:,k+1,cur)) ...
                              - W(:,:,k,old)*(B(:,:,j-1).'));
                    end % if (k<j-1)
                end % if (j>2)
            end % if (k=j)
            W(:,:,k,old) = W(:,:,k,old)/B(:,:,j);
%
            % find the first W which loses orthogonality
            for colW = 1:bs            % each column of W(:,:,k,j+1)
                if (max(abs(W(:,colW,k,old))) >= SQRTEPS)
                    doOrtho = 1;       % found loss of ortho
                    up = k;
                    break;
                end
            end % for colW
            k = k + 1;
        end % while ((k <= j) & (doOrtho ~= 1))
%
        if ((doOrtho == 1) & (up < j))
            % if loss of ortho was found and not the last W
            thresh = 0;         % flag if MIDEPS is found
            k = j;              % search from the last W
            while ((k >= 2) & (thresh ~= 1))
                if (k == j)
                    W(:,:,k,old) = (eps*bs*0.6)*(B(:,:,1) ...
                                   .*(randn(bs,bs)+IM*randn(bs,bs)));
                else % k<j
                    W(:,:,k,old) = M(:,:,k)*conj(W(:,:,k,cur)) ...
                                + B(:,:,k-1)*conj(W(:,:,k-1,cur)) ...
                                - W(:,:,k,cur)*M(:,:,j) ...
                                + (eps*0.3)*((B(:,:,k)+B(:,:,j)) ...
                                   .*(randn(bs,bs)+IM*randn(bs,bs)));
                    if (k < j - 1)
                        W(:,:,k,old) = W(:,:,k,old) ...
                              + (B(:,:,k).')*conj(W(:,:,k+1,cur)) ...
                              - W(:,:,k,old)*(B(:,:,j-1).');
                    end % if (k<j-1)
                end % if (k=j)
                W(:,:,k,old) = W(:,:,k,old)/B(:,:,j);
%
                % check if W(:,:,k,j+1) exceeds MIDEPS
                for colW = 1:bs         % for each column of W
                    if (max(abs(W(:,colW,k,old))) >= MIDEPS)
                        thresh = 1;     % found a W
                        up = k;
                        break;
                    end
                end % for colW
                k = k - 1;
            end % while ((k >= 2) & (thresh ~= 1))
        end % if ((doOrtho == 1) & (up ~= j))
    else        % second orthogonalization
        if (up < j)             % compute Ws in [up j]
            for k = up:j
                if (k == j)
                    W(:,:,k,old) = (eps*bs*0.6)*(B(:,:,1) ...
                                   .*(randn(bs,bs)+IM*randn(bs,bs)));
                else % k<j
                    W(:,:,k,old) = M(:,:,k)*conj(W(:,:,k,cur)) ...
                                + B(:,:,k-1)*conj(W(:,:,k-1,cur)) ...
                                - W(:,:,k,cur)*M(:,:,j) ...
                                + (eps*0.3)*(B(:,:,k)+B(:,:,j)) ...
                                  .*(randn(bs,bs)+IM*randn(bs,bs));
                    if (k < j-1)
                        W(:,:,k,old) = W(:,:,k,old) ...
                              + (B(:,:,k).')*conj(W(:,:,k+1,cur)) ...
                              - W(:,:,k,old)*(B(:,:,j-1).');
                    end % if (k<j-1)
                end % if (k=j)
                W(:,:,k,old) = W(:,:,k,old)/B(:,:,j);
            end % for k = up:j
        end % if (up < j)
    end % if (second == 0)
%
    tmp = old;                  % swap indices old and cur
    old = cur;
    cur = tmp;
%
    if ((doOrtho == 1) | (second == 1))
        % orthogonalize Q(j+1) against Q(k), k is inside the interval
        qLow2 = 1 - bs;         % initial low and up indices
        qUp2 = 0;               % for Q which is reorthogonalized
        for (k = 1:up)
            qLow2 = qLow2 + bs;
            qUp2 = qUp2 + bs;
            for (colR = 1:bs)
                for (colQ = qLow2:qUp2)
                    R(:,colR) = R(:,colR) ...
                              - (Q(:,colQ)'*R(:,colR))*Q(:,colQ);
                end
            end
            %reset orthogonality estimates
            W(:,:,k,cur) = eps*1.5*(randn(bs,bs) + IM*randn(bs,bs));
        end % for (k = 1:up)
        [Q(:,(qLow+bs):(qUp+bs)), B(:,:,j)] = qr(R, 0);
        nBlk = nBlk + up;       % update number of blocks selected
                                % for orthogonalization
%
        if (second == 1)
            second = 0;
            up = 0;             % clear upper bounds for next iter
        else
            second = 1;
            doOrtho = 0;
            up = min(j + 1, up + 1);    % adjust for second ortho
        end
    end % if ((doOrtho == 1) | (second == 1))
end % for j = 2:steps-1
%
% the last iteration
qLow = qLow + bs; qUp = qUp + bs; Tmp = A*conj(Q(:,qLow:qUp));
M(:,:,steps) = Q(:,qLow:qUp)'*Tmp;
%
nVec = nBlk * bs;               % update the number of vectors
                                % selected for orthogonalization
```

C.2 Tridiagonalization

```
function [a,b,P,nVec] = LanTri(M,B,r)
% [a,b,P,nVec] = LanTri(M,B,r)
%
% Lanczos tridiagonalization of a complex-symmetric and block
% tridiagonal matrix. Modified partial orthogonalization is
% applied if required.
%
% Input
%   M     M(:,:,i) are main diagonal blocks of the block tridiagonal
%   B     B(:,:,i) are subdiagonal blocks of the block tridiagonal
%   r     starting vector
% Outputs
%   a     main diagonal of the tridiagonal
%   b     subdiagonal of the tridiagonal
%   P     unitary
%   nVec  number of vectors selected for reorthogonalization
% so that
%       J = P*(diag(a) + diag(b,1) + diag(b,-1))*P.'
% where J is block tridiagonal whose main diagonal blocks are
% M(:,:,i) and the subdiagonal blocks are B(:,:,i).
%
% Dependency
%    ./sbmvmul.m    symmetric block tridiagonal matrix and vector
%                   multiplication
%
n = length(r);                  % dimension of starting vector
%
% constants
IM = sqrt(-1); SQRTEPS = sqrt(eps);
MIDEPS = sqrt(SQRTEPS)^3;       % between sqrt(eps) and eps
%
% initialize two column vectors for diagonals
a = zeros(n,1); b = zeros(n-1,1);
wOld = zeros(n,1);              % orthogonality estimates
wCur = zeros(n,1); wOld(1) = 1.0;
%
up = ones(n,1);                 % upper and lower bounds for
low = ones(n,1);                % orthogonalization intervals
```

```
interNum = 0;                   % orthogonalization interval number
doOrtho = 0;                    % if do orthogonalization
second = 0;                     % if this is the second partial ortho
nVec = 0;                       % number of vectors for reortho
%
P(:,1) = r/norm(r);             % set the first column of P
%
for j=1:n
    tmp = sbmvmul(M,B,conj(P(:,j)));    % J*conj(p(j)).Band multiply
    a(j) = P(:,j)'*tmp;                 % a(j) = p(j)'*J*conj(p(j))
    % calculate r = J*conj(p(j)) - a(j)*p(j) - b(j-1)*p(j-1)
    if j == 1
        r = tmp - a(j)*P(:,j);
    else
        r = tmp - a(j)*P(:,j) - b(j-1)*P(:,j-1);
    end
%
    if (j < n)
        b(j) = norm(r);
%
        if (j > 2)          % compute orthogonality estimates
            wOld(1) = (b(1)*conj(wCur(2)) + a(1)*conj(wCur(1)) ...
                       - a(j)*wCur(1) - b(j-1)*wOld(1))/b(j) ...
                       + eps*(b(1)+b(j))*0.3*(randn + IM*randn);
            wOld(2:j-1) = (b(2:j-1).*conj(wCur(3:j)) ...
                           + a(2:j-1).*conj(wCur(2:j-1)) ...
                           - a(j)*wCur(2:j-1) ...
                           + b(1:j-2).*conj(wCur(1:j-2)) ...
                           - b(j-1)*wOld(2:j-1))/b(j) ...
                           + eps*0.3*(b(2:j-1)+b(j)*ones(j-2,1)) ...
                           .*(randn(j-2,1) + IM*randn(j-2,1));
%
            % swap wOld and wCur
            tmp = wOld(1:j-1);
            wOld(1:j-1) = wCur(1:j-1);
            wCur(1:j-1) = tmp;
            wOld(j) = 1.0;
        end % if j>2
        wCur(j) = eps*n*(b(1)/b(j))*0.6*(randn + IM*randn);
        wCur(j+1) = 1.0;
```

```
%
        if (second == 0)      % not second time,determine intervals
            doOrtho = 0;      % initialization
            interNum = 0;
            k = 1;
            while k <= j
                if (abs(wCur(k)) >= SQRTEPS)    % lost ortho
                    doOrtho = 1;
                    interNum = interNum + 1;
                    % find the upper bound
                    p = k + 1;
                    while ((p < (j + 1)) & (abs(wCur(p)) >= MIDEPS))
                        p = p + 1;              % nearly lost ortho
                    end % while
                    up(interNum) = p - 1;
                    % find the lower bound
                    p = k - 1;
                    while ((p > 0) & (abs(wCur(p)) >= MIDEPS))
                        p = p - 1;          % nearly lost orthogonality
                    end % while
                    low(interNum) = p + 1;
%
                    k = up(interNum) + 1;   % continue search
                else
                    k = k + 1;
                end % if lost orthogonality
            end % while k
        end % if not second time
%
        if ((doOrtho == 1) | (second == 1)) % now we have intervals,
            % carry out orthogonalization
            for (k = 1:interNum)            % for each interval
                for (i = low(k):up(k))
                    r = r - (P(:,i)'*r)*P(:,i); % do ortho
                    % reset ortho estimates
                    wCur(i) = eps*1.5*(randn + IM*randn);
                end % for i
%
                nVec = nVec + up(k) - low(k) + 1;
                % count the number of vectors selected
                if (second == 1)            % this is the second time
                    second = 0;             % reset
                    low(k) = 0;
                    up(k) = 0;
                else
                    second = 1;             % do second time
                    doOrtho = 0;            % reset
                    % adjust ortho intervals for the second time
                    low(k) = max(1, low(k) - 1);
                    up(k) = min(j + 1, up(k) + 1);
                end % if
            end % for k
            b(j) = norm(r);                 % recalculate b(j)
        end % if
%
        if (abs(b(j)) < eps)                % b(j)=0, quit
        a = a(1:j);
            b = b(1:j-1);
        return
        else
            P(:,j+1) = r/b(j);
        end % if
    end
end
```

C.3 Matrix-Matrix Multiplication

```
function [r] = sbmvmul(M,B,v)
% [r] = sbmvmul(M,B,v)
%
% Complex-symmetric and block tridiagonal matrix-vector
% multiplication. Inner product version.
%
% Input
%   M   M(:,:,i) are the main diagonal blocks
%   B   B(:,:,i) are the subdiagonal blocks
%   v   column vector
% Outputs
%   r   product vector so that
%         r = J*v,
% where J is complex-symmetric and block tridiagonal whose main
% diagonal blocks are M(:,:,i) and subdiagonal blocks are B(:,:,i).
%
[b,b,k] = size(M);              % block size, number of diagonal blocks
n = b*k;                        % matrix size
%
r = zeros(n,1);                 % initialize a column vector
%
r(1:b) = M(:,:,1)*v(1:b) + B(:,:,1).'*v(b+1:2*b);
%
for i = 1:k-2
    low = i*b +1;               % lower bound
    up = (i+1)*b;               % upper bound
    r(low:up) = B(:,:,i)*v((low-b):(low-1)) ...
              + M(:,:,i+1)*v(low:up) ...
              + B(:,:,i+1).'*v(up+1:up+b);
end
%
r(n-b+1:n) = B(:,:,k-1)*v((n-2*b+1):(n-b)) + M(:,:,k)*v(n-b+1:n);
```

Bibliography

[1] S. Qiao. Orthogonalization Techniques for the Lanczos Tridiagonalization of Complex Symmetric Matrices.

[2] Horst D. Simon. The Lanczos algorithm with partial reorthogonalization. *Mathematics of Computation.* **42** (1984), 115-142.

[3] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[4] G. H. Golub and C. F. Van Loan. *Matrix Computations*, 3rd Ed. The Johns Hopkins University Press, Baltimore, MD, 1996.

[5] M. Berry, T. Do, G. O'Brien, V. Krishna, and S. Varadhan. SVDPACKC (version 1.0) user's guide. Technical Report CS-93-194, University of Tennessee, Department of Computer Science, 1993

[6] F. T. Luk and S. Qiao. A fast singular value algorithm for Hankel matrices. *Fast Algorithms for Structured Matrices: Theory and Applications, Contemporary Mathematics 323*, Editor V. Olshevsky, American Mathematical Society. 2003. 169–177.

[7] A. Bunse-Gerstner and W. B. Gragg. Singular value decompositions of complex symmetric matrices. *Journal of Computational and Applied Mathematics*, **21** (1988) 41–54.

[8] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

[9] T. Takagi. On an algebraic problem related to an analytic Theorem of Carathédory and Fejér and on an allied theorem of Landau. *Japan J. Math.* **1** (1924) 82–93.

[10] E.Anderson, Z.Bai, C.Bischof, J.Demmel, J.Dongarra, J.DuCroz, A.Greenbaum, S.Hammarling, A.McKenney, S.Ostrouchov, and D.Sorensen. *LAPACK Users' Guide, Third edition*. SIAM Publications, Philadelphia, 1999.

[11] Jack Dongarra, Roldan Pozo, David Walker. *LAPACK++ V1.1 High Performance Linear Algebra Users' Guide*. National Institute of Standards and Technology, University of Tennessee, Knoxville, Oak Ridge National Laboratory. 1996.

[12] Guohong Liu, Wei Xu and Sanzheng Qiao. *Block Lanczos Tridiagonalization of Complex Symmetric Matrices*. Technical Report, No. CAS 04-07-SQ Department of Computing and Software, McMaster University. September 2004.