## Final Project

**Problem Description:**
Bisection and divide-and-conquer: Implement two codes for computing eigenvalues of a symmetric tridiagonal matrix A:
(a) Method of bisection: Find all the eigenvalues in a given interval [a, b], using a standard bisection method and Sturm sequences.
(b) Divide-and-conquer: Find all the eigenvalues of A, by solving the secular equation using Newton?s method.
Run tests and timings that verify the expected computational complexities of your codes, and compare the accuracy with MATLAB's eig function.

**Code:** Included in the submission, not in this report.

**Bisection:**

**a:** High level idea:
Given the nice interlace property of eigenvalues of symmetric diagonal matrix, we can compute the Sturm sequence with any shift S of a matrix in linear time, and use the number of sign changes in the sequence to determine the number of eigenvalues between $(-\infty, S)$.
With this technique in hand, we can quickly find the number of eigenvalues between $[a, b)$, which is already of great value. Furthermore, since we can generally find number of eigenvalues between any interval, we can bisect the given interval [a,b] and find the exact eigenvalues by just bisecting subintervals until subintervals (that contains exactly one eigenvalue) are so small that we can treat any point in the interval as an eigenvalue of A.
• Input: A, a, b, tolerance
• Return: eigenvalues of A in interval [a,b], with relative error under tolerance
• precondition: A is symmetric, tridiagonal. The eigenvalues of A must be unique in context of tolerance. (in other words, the method breaks down with huge tolerance, since the algorithm will stop when there are still several eigenvalues in the interval.)

1.Check if b is an eigenvalue of A.
2.Bisect [a,b) into [a,(a+b)/2) and [(a+b)/2,b), the eigenvalues in [a,b) are just the union of eigenvalues of two subintervals.
3.Find eigenvalues of subintervals recursively by repeating 2, until subinterval is small enough.

**b:** Choices I have made:
• When calculating the Sturms sequence with large matrices and large shift, the numbers will actually overflow or underflow. So instead of calulating the exact sequence, I scale some terms if they appears too large. I did not implement the underflow case, as I think we can pre-scale the matrix to convert that into an overflow problem, which is much easier to handle.
• The reason why underflow is hard for me is that, in my calculations of Sturms, if the last term of the sequence is really close to zero, I have a good reason to believe the shift is almost an eigenvalue. Since there seems to be a cumulated precision error, when matrix size is large, the algorithm is prone to error when trying to find sign changes near zero. Thus, I will recalculate the Sturm

sequence with a slightly larger shift to avoid this problem.
• The way to calculate the number of sign changes follows closely to the convention introduced in Trefethen/Bau.
• To Avoid dynamic memory allocation, I pre-allocated memory for eigenvalues. This can be buggy if the precondition do not met and my algorithm cannot find enough number of eigenvalues in the subinterval. However, if precondition is satisfied, no error will occur.
• tolerance is user-chosen. The algorithm will break if tolerance is smaller than machine error.
• Sub intervals that has 0 eigenvalues in it will not be examed.
• When bisecting [a,b), I will calculate the number of eigenvalues from $(-\infty, a)$,$(-\infty, \frac{a+b}{2})$,$(-\infty, b)$. Note that, these values can be reused when bisecting $[a, \frac{a+b}{2})$, $[\frac{a+b}{2}, b)$. Thus, I will pass these values into smaller function calls, which nearly doubles the speed.

**d:** Test Problem, Results and Analysis:
**1.** Random Normal A,find all eigenvalues.

```
m = 512; %4,8,16,32,64,128,256,512
Error=0;
for i=1:10
    A = randn(m);
    B = hess(A'*A)*10^40; % huge numbers
    %B = hess(A'*A);       % normal numbers

    E = eig(B);
    %ANS = bisection(B,0,10000,10e-14,-1,-1);% normal numbers
    ANS = bisection(B,0,10000*10^40,10e-14,-1,-1);% huge numbers
    if size(ANS,1)~=m
        fprintf('BAD!\n');
    end
    Error= Error + norm(E(1:size(ANS,1))-ANS);
end
Error=Error/10
```

**Results:** See next page. I ran each problem ten times to average out some outliers and included the total errors. I also included the number of function calls and running time of bisection and Sturm.

**Analysis:**
The number of function calls for each eigenvalue do not vary with scaling of both numbers and size of the problem, which is expected. In fact, with problem size being larger, the average number of functions calls for each eigenvalue goes down. However, huge numbers requires slightly more time because I have to scale more frequently to keep the Sturm sequence small.
The relative error is the same for different scalings. (Multiplying by $10^{40}$ gives $10^{40}$ bigger errors)
The time here is quadratic with the problem (matrix) size. Based on the number of function calls, as size of the problem double, we call double amount of Sturm, which is a O(m) function. This means the implementation is $O(m^2)$ indeed.

TABLE 1. Test1

| bisection 10 times. | | | | | | | |
|---|---|---|---|---|---|---|---|
| Scale | Size(m) | Error | # bisection called/time | | # Sturm called/time(s) | | # of Sturm per eigenvalue |
| Normal | 512 | 3.60E-10 | 199911 | 4.218 | 194815 | 40.465 | 38.04980469 |
| Normal | 256 | 1.21E-10 | 102533 | 1.836 | 99996 | 10.492 | 39.0609375 |
| Normal | 128 | 4.48E-11 | 52579 | 0.819 | 51329 | 2.756 | 40.10078125 |
| Normal | 64 | 1.63E-11 | 26952 | 0.389 | 26332 | 0.776 | 41.14375 |
| Normal | 32 | 5.30E-12 | 13837 | 0.211 | 13545 | 0.285 | 42.328125 |
| Normal | 16 | 1.98E-12 | 7120 | 0.128 | 6983 | 0.117 | 43.64375 |
| Normal | 8 | 6.37E-13 | 3705 | 0.139 | 3645 | 0.065 | 45.5625 |
| Normal | 4 | 2.13E-13 | 1935 | 0.06 | 1929 | 0.028 | 48.225 |
| | | | | | | | |
| Huge | 512 | 3.55E+30 | 199852 | 4.171 | 194753 | 45.918 | 38.03769531 |
| Huge | 256 | 1.23E+30 | 102575 | 1.546 | 100061 | 11.339 | 39.08632813 |
| Huge | 128 | 4.52E+29 | 52590 | 0.662 | 51330 | 2.974 | 40.1015625 |
| Huge | 64 | 1.51E+29 | 26980 | 0.382 | 26361 | 0.909 | 41.1890625 |
| Huge | 32 | 5.01E+28 | 13851 | 0.291 | 13551 | 0.356 | 42.346875 |
| Huge | 16 | 1.92E+28 | 7118 | 0.152 | 6979 | 0.115 | 43.61875 |
| Huge | 8 | 4.84E+27 | 3700 | 0.106 | 3640 | 0.072 | 45.5 |
| Huge | 4 | 1.95E+27 | 1935 | 0.039 | 1915 | 0.04 | 47.875 |
| * Everything multiplied by 10^40 | | | | | | | |

**2.** Random Normal A, find eigenvalues in a range.

```
m = 64; %4,8,16,32,64,128,256,512
Error=0;
a = 1;
b = 512; %2, 4,8,16,32,64,128,256
Eigenvalues_found =0;
for i=1:10
    A = randn(m);
    B = hess(A'*A);        % normal numbers
    E = eig(B);
    ANS = bisection(B,a,b,10e-14,-1,-1);% normal numbers

    E_in_range=zeros(0,1);
    for j=1:m
        if (E(j)>=a && E(j)<=b)
            E_in_range(end+1,1)=E(j);
        end
    end

    if size(ANS,1)~=size(E_in_range,1)
        fprintf('BAD! %d %d \n',size(ANS,1),size(E_in_range,1));
        ANS
        E_in_range
    end
    Eigenvalues_found = Eigenvalues_found + size(ANS,1);
    Error= Error + norm(E_in_range-ANS);
end
Error=Error/10
Eigenvalues_found
```

TABLE 2. Test2, matrix is 64 by 64

| bisection 10 times | | | | | | |
|---|---|---|---|---|---|---|
| Range | Eig found | Total error | Error per eigenvalue | # Sturm called/time(s) | | # of Sturm per eigenvalue |
| [1,2] | 21 | 4.05E-14 | 1.93E-15 | 911 | 0.06 | 43.38095238 |
| [1,4] | 51 | 1.12E-13 | 2.19E-15 | 2174 | 0.117 | 42.62745098 |
| [1,8] | 93 | 2.39E-13 | 2.57E-15 | 3928 | 0.175 | 42.23655914 |
| [1,16] | 154 | 6.46E-13 | 4.19E-15 | 6453 | 0.251 | 41.9025974 |
| [1,32] | 230 | 1.58E-12 | 6.85E-15 | 9558 | 0.276 | 41.55652174 |
| [1,64] | 339 | 3.63E-12 | 1.07E-14 | 13958 | 0.446 | 41.1740413 |
| [1,128] | 468 | 8.12E-12 | 1.73E-14 | 19146 | 0.633 | 40.91025641 |
| [1,256] | 584 | 1.40E-11 | 2.39E-14 | 23808 | 0.705 | 40.76712329 |

**Analysis:**
As we can see, the error per eigenvalue is invariant with the given parameter a and b (in the sense of relative error, for larger eigenvalues, we do expect slightly larger errors). Furthermore, the number of Sturm called for each eigenvalue is around 40. Given that our input relative tolerance is $10^{-13} \approx 2^{-43}$, the number of function calls are slightly better than expected.

**Divide and Conquer:**

**a:** High level idea of implementation:
- Using the idea of divide and conquer to solve all eigenvalues of A.
- Return: Sorted eigenvalues of A and corresponding eigvector matrix Q
- precondition: A is m by m (m≥2), symmetric, tridiagonal.

0. If A is small, solve using ordinary methods.
1. If A is not irreducible (There are some zeros in the sub diagonal of A), reduce A.
2. Divide: find eigenvalues/eigenvector of upper-half/ lower-half
3. MergeSort two eigenvalues returned in 2
4. Conquer: Find eigenvalue of D ± w*w' by solving secular equations.
5. Conquer: Given eigenvalue, find eigenvector in O($m^2$)

**b:** Choices I have made:
- For the purpose of this project, I will keep dividing A until its size is smaller or equal to 2.
- After I divide A and conquer both top and bottom part, I will use merge-sort to sort their eigenvalues, which is O(m).
- I wrote two files to handle positive and negative case of $\beta$.
- I chose $10^{-10}$ as my tolerence.

**c:** How to the secular equation formed in the Conquer Phase:
- I basically used a bisection structure, with possible speedup with Newton's method when applicable. I will show the logical sequence of the positive $\beta$ case, negative $\beta$ case follows trivially.

Given the eigenvalue D, $\vec{w}$, sorted D called Dsort.
For each i in 1 to m-1, $\lambda_i$ will be bounded by $D_i$ and $D_{i+1}$.
Thus, set **left**=$D_i$ and **right**=$D_{i+1}$, $\lambda_i$= (**left** and **right**)/2.
While True:
——If **left** and **right** are close enough, return $\lambda_i$.
——Else, find:
———$f(\lambda_i)= \vec{w}^T(\text{diag(D)} - \lambda_i I)^{-1}\vec{w}$
———$f'(\lambda_i)= \vec{w}^T(\text{diag(D)} - \lambda_i I)^{-2}\vec{w}$
———**intercept** $= \lambda_i - f(\lambda_i)/f'(\lambda_i)$
———If **intercept** is really close to $\lambda_i$, we have found a good $\lambda_i$, return $\lambda_i$
———If **intercept** is within **left** and **right**, $\lambda_i$=**intercept** and continue
———Else, do bisection, update **left** and **right** according to $f(\lambda_i)$.

For $\lambda_m$, the last eigenvalue, it is only lower bounded by $D_m$, I used a variation of bisection:
set **left**=$D_i$ and **right**=$\infty$, $\lambda_i$= 2 * ‖ **left** ‖.
While True:
——If **left** and **right** are close enough, return $\lambda_i$.
——Else, find:
———$f(\lambda_i)= \vec{w}^T(\text{diag(D)} - \lambda_i I)^{-1}\vec{w}$
———If $f(\lambda_i)$ is positive, shift **right** to $\lambda_i$ and now we can do normal bisection.
———Else, shift **left** to $\lambda_i$ and $\lambda_i$= 2*‖ **left** ‖.

**d:** Test Problem:

```
m = 512;
Error=0;
for i=1:10
    A = randn(m);
    B = hess(A'*A)*10^20;
    [V1,D1] = eig(B);
    [V2,D2] = DivideAndConquer(B);
    Error = (norm(D1-diag(D2)));
end
Error=Error/10
```

Note that, the unbounded eigenvalue can take huge amount of steps theoretically, but in practice the cost is negligible. Here I use rand normal matrix to prevent this unconsidered factor.

**e:** Results:

TABLE 3. Normal Scale

| Divide and Conquer 10 times (Stable Newton) | | | | | | |
|---|---|---|---|---|---|---|
| Scale | Size(m) | Error | number of DivideAndConquer called/time | | Secular Equations Solved /time(s) | |
| Normal | 512 | 7.29E-01 | 5110 | 12.528 | 2550 | 233.829 |
| Normal | 256 | 4.30E-01 | 2550 | 1.466 | 1270 | 26.213 |
| Normal | 128 | 9.30E-02 | 1270 | 0.302 | 630 | 3.476 |
| Normal | 64 | 2.02E-10 | 630 | 0.152 | 310 | 1.257 |
| Normal | 32 | 1.20E-10 | 310 | 0.083 | 150 | 0.379 |
| Normal | 16 | 1.28E-10 | 150 | 0.052 | 70 | 0.246 |

TABLE 4. Huge Scale

| Divide and Conquer 10 times (Stable Newton) | | | | | | |
|---|---|---|---|---|---|---|
| Scale | Size(m) | Error | number of DivideAndConquer called/time | | Secular Equations Solved /time(s) | |
| huge | 512 | 1.28E+20 | 5110 | 11.659 | 2550 | 188.957 |
| huge | 256 | 1.15E+20 | 2550 | 1.407 | 1270 | 19.663 |
| huge | 128 | 4.77E+19 | 1270 | 0.272 | 630 | 2.982 |
| huge | 64 | 2.13E+14 | 630 | 0.142 | 310 | 0.751 |
| huge | 32 | 7.04E+12 | 310 | 0.075 | 150 | 0.207 |
| huge | 16 | 3.32E+11 | 150 | 0.033 | 70 | 0.078 |
| * Everything multiplied by 10^20 | | | | | | |

**f:** Analysis:

Based on the number of function calls, my recursive step looks correct.

Note that, the error do not scale linearly with the matrix size. For the normal case from 64 to 128, there is a huge jump of errors. I have double checked my code and concluded that the errors are resulting from the slightly incorrect eigenvalues and eigenvectors in the divide step. With large matrix size, the probability of this sudden jump occurring is not negligible and account wholely for the increased error. The possible way to fix is to increase tolerance when solving small matrices. The errors for huge numbers have the same pattern and relative error do not scale.

Even with multiple tests, because of the fluctuation of my computer's performance, the relative time between huge and normal cannot be compared. I expect them to be relatively the same. For each set of tests, normal scale, for example, we expect to see a cubic pattern because my implementation not only finds the eigenvalues, but eigenvectors as well. My implementation shows roughly a cubic pattern. Note that even if eigenvalues may only take $O(m^2)$, I have tried for quite a while but did not figure out a way to perform the algorithm without knowing the eigenvectors of sub matrices. I can speedup my implementation by not computing the last eigenvector but that will be a constant time multiplier.

Also, I have tested Stable Newton and bisection only methods. The difference of time is basically a factor of 2, which means, the algorithm is still performing bisection in most cases. (for full Newton I expect a factor of 5 (40 verses log(40)) or so, which did not show up.)