Understanding *A Fast Contour-Integral Eigensolver for Non-Hermitian Matrices*

Austin Gill
Karen Braman

May 19, 2018

Abstract

We examine a new complex contour integral-based eigensolver algorithm using ideas from functional analysis. This eigensolver algorithm extends previous eigensolvers that typically required some special structure, to an eigensolver that works on a more general class of complex-valued matrices.

The algorithm consists of two stages. First, it searches the complex plane for regions dense with eigenvalues. Second, it combs through those regions to find the eigenvalues using a modified form of the FEAST algorithm.

Due to time constraints we were only able to examine the first stage in detail, and for completeness provide a description for the second stage. Our work was exploratory in nature, which in practice consisted of building and testing a Matlab implementation of the first stage.

Contents

1 Introduction

1.1 The Eigenvalue Problem

All undergraduate mathematics students know that matrices are *linear transformations* as illustrated in Figure 1. That is, they are transformations that take in a set of vectors and transform (or map) them to another set of vectors. The Eigenvalue Problem asks the question

"Given a transformation, what vectors map to scaled versions of themselves?"

That is, if A is a transformation acting on a vector $\vec{x}$ in some vector space, can we find vectors $\vec{x}$ and scalars λ that satisfy the following relationship?

$$A\vec{x} = \lambda\vec{x} \tag{1.1}$$

For our purposes, $A \in \mathbb{C}^{n\times n}$ will be a non-Hermitian matrix with no particular structure, and the eigenvalues $\lambda \in \mathbb{C}$ will be complex scalars, and the eigenvectors $\vec{x} \in \mathbb{C}^n$ will be complex-valued vectors.

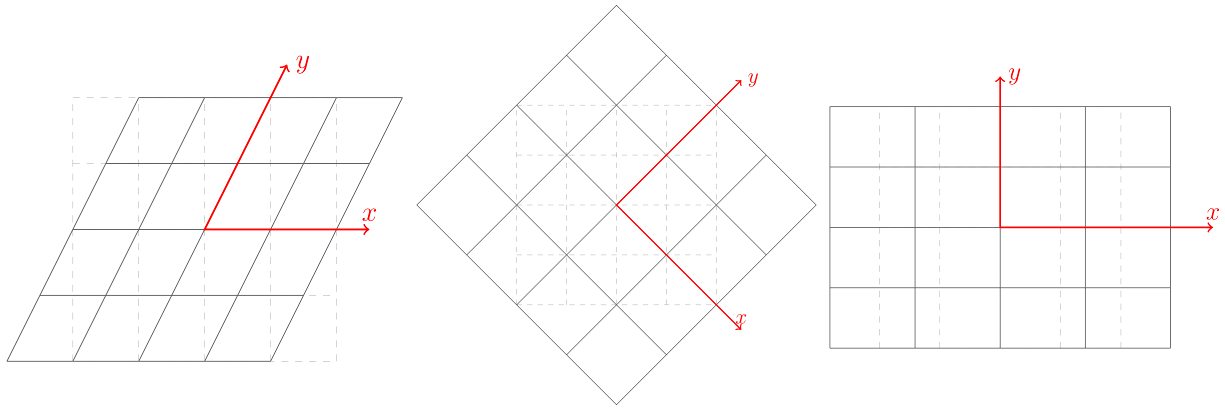


Figure 1: Some different linear transformations

Definition 1.1. The **conjugate transpose** of a matrix A, denoted by any of the following

$$A^* = A^H = \left(\overline{A}\right)^T = \overline{A^T}$$

takes the elementwise complex conjugate of A^T. Observe that if $A \in \mathbb{R}^{n\times n}$, then $A^H = A^T$ (the complex conjugate of a real number is that real number).

Definition 1.2. A matrix A is **Hermitian** if $A = A^H$. Observe that if $A \in \mathbb{R}^{n\times n}$ we would say A is *symmetric*. Hermitian matrices always have real eigenvalues. It is appropriate to think of Hermitian matrices as the complex extension of real symmetric matrices.

The fact that A is non-Hermitian is significant because the algorithms we will discuss extend an existing method specific to Hermitian matrices to a more general method for non-Hermitian matrices.

1.2 A Naïve Solution

Before we can beat the eigenvalue problem with big sticks, we must first beat the problem with little sticks. So consider,

$$\begin{aligned} A\vec{x} &= \lambda\vec{x} \\ A\vec{x} - \lambda\vec{x} &= 0 \\ (A - \lambda I)\,\vec{x} &= 0 \end{aligned} \tag{1.2}$$

Now we have translated the question

> "Given a transformation, what vectors map to scaled versions of themselves?"

into the question stated in Equation 1.2

> "What vectors map to 0 under the transformation $A - \lambda I$?"

or rather, "What is the null space of $A - \lambda I$?" So we take the determinant $\det(A - \lambda I)$, set it equal to zero, and solve for λ. I.e., we find the roots of the characteristic polynomial

$$p(z) = \det(A - zI) \tag{1.3}$$

We have thus reduced our original problem to that of finding the roots of a polynomial, and declare ourselves done. However, root-finding is a well-studied problem, so what makes this approach so naïve?

First, finding the roots of the characteristic polynomial requires *finding* the characteristic polynomial. For large problems, this may be intractable. It requires finding the determinant for every λ, which is a costly operation, ranging in complexity from $\mathcal{O}(n!)$ to $\sim \mathcal{O}(n^3)$ in special cases.

Second, after the characteristic polynomial has been found, we must find its roots, which adds additional complexity. In fact, Trefethen's *Numerical Linear Algebra*[1] claims that root-finding problems are ill-conditioned, even when the underlying eigenvalue problem is not. Observe also that, for matrices 5×5 and bigger, there is no closed form solution for the roots of the characteristic polynomial, so it must be approximated. Additionally, there is the numerical stability of root-finding algorithms to account for.

So what should we do instead?

1.3 A Proposed Solution

There are many existing methods for solving the eigenvalue problem, but Sakurai and Sugiura [2] proposed a *new* method for solving the eigenvalue problem in 2003 using *the power of math*™. This work prompted an entirely new class[1] of eigensolvers. Each of these new eigensolvers (ab)uses ideas from complex and functional analysis to use complex contour integrals to find the eigenpairs of a matrix, typically requiring the matrix to follow some special structure to make the problem easier.

One of the new algorithms that this paper inspired was the FEAST [4] algorithm, which solves the generalized eigenvalue problem

$$A\vec{x} = \lambda B\vec{x} \tag{1.4}$$

for a Hermitian matrix A, and a symmetric positive definite matrix B.

[1] You can read a summary of several such algorithms in the paper *A map of contour integral-based eigensolvers for solving generalized eigenvalue problems* [3].

Definition 1.3. A symmetric matrix $A \in \mathbb{R}^{n\times n}$ is said to be **positive definite** if the scalar

$$\vec{z}^T A\vec{z}$$

is positive for every nonzero column vector $\vec{z}$ of A.

The paper we examined, *A Fast Contour-Integral Eigensolver for non-Hermitian Matrices* [5] extends the FEAST algorithm to the Structured FEAST (SFEAST) algorithm and adds a precomputation stage to comb through the complex plane looking for feasible search regions. This new SFEAST algorithm relaxes some of the requirements (e.g. that A should be Hermitian) of the older eigensolvers.

The overall eigensolver algorithm developed in [5] can roughly be spit into two stages:

Stage 1 Search the complex plane for regions that contain some specified number of eigenvalues. Mark these regions as target regions.

Stage 2 Run the Structured FEAST algorithm given in [5] on each target region.

The paper [5] notes that they had a 40% speedup when they added the first stage. The overall complexity of the algorithm presented is $\mathcal{O}(rn^2) + \mathcal{O}(nr^2)$, where A is $n \times n$ and r is the maximum off diagonal rank as discussed later in Definition 3.2.

1.4 Our Goals

Our goals for this research project have been narrowed down to understanding and partially implementing the first stage in Matlab. Our initial goals were to understand, implement, and then improve (possibly with parallelization) the overall eigensolver. However, as detailed later, we ran into several problems just in implementing the first stage, so we moved the goalposts and focused on understanding and implementing a smaller piece of the puzzle.

1.5 Our Results

We have been able to implement the `EigenCounter` algorithm introduced in Section 2 with fair amounts of accuracy. Note that, as discussed later, the lower levels of accuracy ($\sim$ 75%) are not an issue because the `EigenCounter` algorithm is intended to be ran recursively on a region. In the larger scheme of things, when the algorithm hits its divergence condition, it quadsects the region and tries again on the four sub-regions. The following Matlab output is from one of our test scripts that generates 20 random 100×100 matrices and runs `EigenCounter` on the circular region centered at $4 - 7i$ with radius 3.

```
>> full_test_counter
(1) actual number of eigenvalues with radius 3 centered at 4-7i: 1
(1) estimated number of eigenvalues: 2.5307+1.5953i magnitude: 2.991571
(1) number of iterations: 68
(2) actual number of eigenvalues with radius 3 centered at 4-7i: 0
(2) estimated number of eigenvalues: -0.036668+0.021258i magnitude: 0.042384
(2) number of iterations: 35
(3) actual number of eigenvalues with radius 3 centered at 4-7i: 3
(3) estimated number of eigenvalues: 0.33933+2.8355i magnitude: 2.855749
(3) number of iterations: 52
(4) actual number of eigenvalues with radius 3 centered at 4-7i: 4
(4) estimated number of eigenvalues: 5.9749+1.742i magnitude: 6.223682
(4) number of iterations: 12
(5) actual number of eigenvalues with radius 3 centered at 4-7i: 0
(5) estimated number of eigenvalues: -0.063731-0.10088i magnitude: 0.119328
```

```
(5) number of iterations: 55
(6) actual number of eigenvalues with radius 3 centered at 4-7i: 3
(6) estimated number of eigenvalues: 2.6293+0.4573i magnitude: 2.668784
(6) number of iterations: 30
(7) actual number of eigenvalues with radius 3 centered at 4-7i: 0
(7) estimated number of eigenvalues: 0.022556-0.014095i magnitude: 0.026597
(7) number of iterations: 83
(8) actual number of eigenvalues with radius 3 centered at 4-7i: 0
(8) estimated number of eigenvalues: 0.12518-0.43354i magnitude: 0.451249
(8) number of iterations: 32
(9) actual number of eigenvalues with radius 3 centered at 4-7i: 2
(9) estimated number of eigenvalues: 1.1087+2.2777i magnitude: 2.533175
(9) number of iterations: 26
(10) actual number of eigenvalues with radius 3 centered at 4-7i: 1
(10) estimated number of eigenvalues: 1.1493-0.057231i magnitude: 1.150735
(10) number of iterations: 77
(11) actual number of eigenvalues with radius 3 centered at 4-7i: 2
(11) estimated number of eigenvalues: 3.2814+1.7323i magnitude: 3.710637
(11) number of iterations: 33
(12) actual number of eigenvalues with radius 3 centered at 4-7i: 3
(12) estimated number of eigenvalues: 4.8141-3.8737i magnitude: 6.179077
(12) number of iterations: 30
(13) actual number of eigenvalues with radius 3 centered at 4-7i: 2
(13) estimated number of eigenvalues: 2.0705+0.052601i magnitude: 2.071201
(13) number of iterations: 60
(14) actual number of eigenvalues with radius 3 centered at 4-7i: 1
(14) estimated number of eigenvalues: 0.73213+1.6857i magnitude: 1.837825
(14) number of iterations: 68
(15) actual number of eigenvalues with radius 3 centered at 4-7i: 1
(15) estimated number of eigenvalues: -0.26446+0.15268i magnitude: 0.305368
(15) number of iterations: 53
(16) actual number of eigenvalues with radius 3 centered at 4-7i: 4
(16) estimated number of eigenvalues: 5.3726-0.65104i magnitude: 5.411903
(16) number of iterations: 22
(17) actual number of eigenvalues with radius 3 centered at 4-7i: 1
(17) estimated number of eigenvalues: 1.1844-0.14814i magnitude: 1.193604
(17) number of iterations: 40
(18) actual number of eigenvalues with radius 3 centered at 4-7i: 0
(18) estimated number of eigenvalues: 0.039362+0.082194i magnitude: 0.091133
(18) number of iterations: 136
(19) actual number of eigenvalues with radius 3 centered at 4-7i: 0
(19) estimated number of eigenvalues: 0.15834-0.31701i magnitude: 0.354353
(19) number of iterations: 39
(20) actual number of eigenvalues with radius 3 centered at 4-7i: 2
(20) estimated number of eigenvalues: 1.7425+1.4218i magnitude: 2.248937
(20) number of iterations: 6
```

2 Overview

Suppose we are interested in the eigenvalues and eigenvectors of a non-Hermitian matrix $A \in C^{n\times n}$. As mentioned in Section 1.3, the eigensolver algorithm in [5] may be split into two stages. Stage 1 is a coarse comb-through of the complex plane, searching for regions where eigenvalues are dense (or at least above some threshold). Stage 2 combs through these regions at a much higher resolution to actually *find* the eigenvalues.

More specifically, Stage 1 consists of running the `EigenCounter` algorithm, and Stage 2 the `SFeast` algorithm. These two stages, along with some glue, make up the `FastEig` algorithm. Each algorithm is explained in more detail in a later section.

2.1 Stage 1

Given an initial search region, run the `EigenCounter` routine to come up with an eigenvalue count estimate. The `EigenCounter` routine can fail (yield a count estimate too big), in which case quadsect the given region and try again on each of the four subregions.

If the `EigenCounter` routine succeeds, mark the region as a target, and save it and its estimated eigenvalue count for Stage 2 to use. Figure 2 illustrates this process on a contrived example with tolerance $k = 2$.

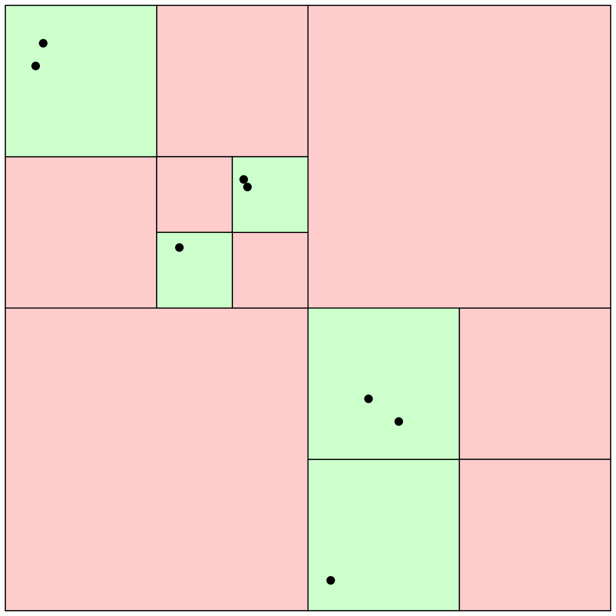


Figure 2: Quadsection target regions shown in green

Note that the use of the word "tolerance" is slightly counter intuitive, because we do not terminate our recursive quadsection once our count estimates dip *under* the threshold, rather, we quadsect *further* when our count estimate is *greater than* the specified tolerance.

2.2 Stage 2

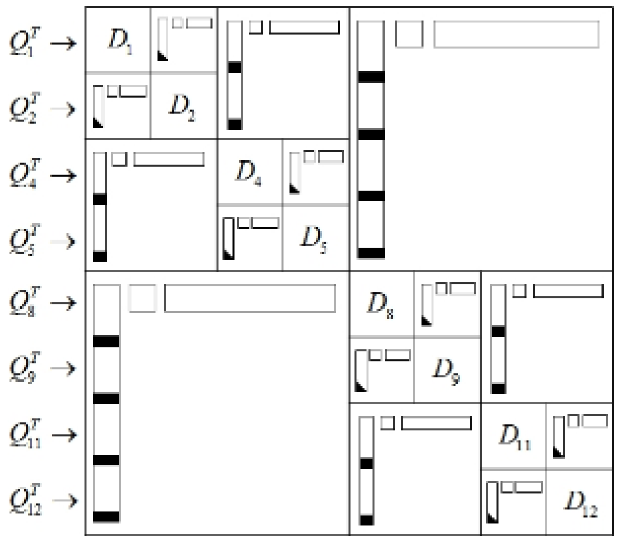
Given a collection of target regions, along with their count estimates, run the `SFeast` algorithm on each target region, saving the approximated eigenvalues and eigenvectors for each region.

2.3 Hierarchically Semi-Separable Matrices?

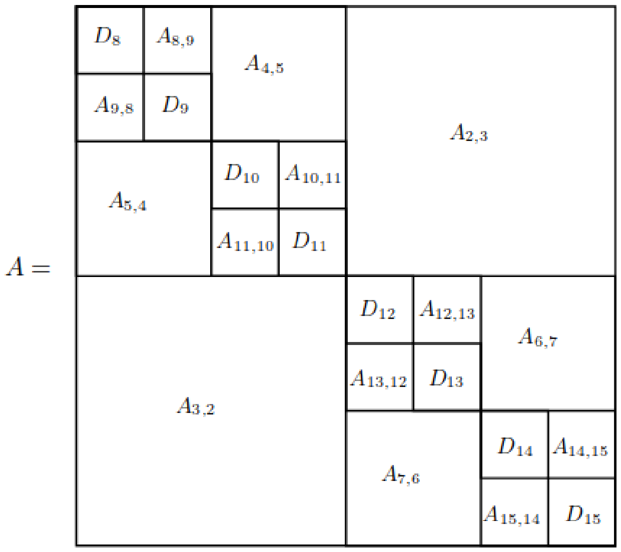
Both of the above stages do not operate directly on the matrix A. Instead, we approximate A by a Hierarchically Semi-Separable (HSS) matrix $\widetilde{A}$. Stage 1 uses a precomputed, low accuracy approximation of A, because all we care about is *how many* eigenvalues there are within a region, and not *where* in that region they exist. Stage 2 uses a much higher accuracy approximation to take information about how many eigenvalues there are and come up with actual eigenvalue approximations.

Using the HSS approximation allegedly saves computation time when solving the system given later in Equation 3.7.

One of the authors of the paper [5] has a Matlab library [6] for HSS computations posted on his website. The code is quite dense, so we have gladly pressed our big, red "I Believe" buttons, and have proceeded without much of a detailed understanding of how HSS approximations work.



(a) HSS illustration found in Figure 4.2 of [7]



(b) HSS illustration found in Figure 3.1 of [8]

Figure 3: Illustrations of HSS matrix approximations.

Understanding and implementing efficient HSS computations will be a challenge for any future work in this area. This is notably true if the given algorithms are to be parallelized.

The HSS library provides the following two functions

`mat2hss` Converts a given matrix[2] A to an HSS approximation $\widetilde{A}$ consisting of tree-like collections of matrix sub-blocks D, U, R, B, W, and V.

`hssulvsol` Quickly solves the equation $\widetilde{A}\vec{x} = \vec{b}$, using the algorithm presented in [9].

Because our use of the provided library was limited, we felt this blind use is appropriate.

[2]Along with a number of metaparameters to tweak the HSS approximation.

3 Details

3.1 The Theoretical Background

Suppose λ_j for $j = 1 \dots s$ are the eigenvalues of A inside some Jordan curve Γ in the complex plane. Now consider the contour integral

$$\phi(z) = \frac{1}{2\pi\mathbf{i}} \int_\Gamma \frac{1}{\mu - z} \mathrm{d}\mu \tag{3.1}$$

where $\mathbf{i}$ is the imaginary unit and $z \notin \Gamma$.[3] A spectral projector[4] $\mathbf{\Phi}$ to the desired eigenspace

$$\operatorname{span}\{x_1, x_2, \dots, x_s\}$$

may be constructed using Cauchy's residue theorem

$$\begin{aligned}
\mathbf{\Phi} \equiv \phi(A) &= \frac{1}{2\pi\mathbf{i}} \int_\Gamma (\mu I - A)^{-1} \mathrm{d}\mu \\
&= \frac{1}{2\pi\mathbf{i}} \int_\Gamma \left(\mu I - X\Lambda X^{-1}\right)^{-1} \mathrm{d}\mu \\
&= X \left(\frac{1}{2\pi\mathbf{i}} \int_\Gamma (\mu I - \Lambda)^{-1} \mathrm{d}\mu \right) X^{-1} \\
&= X \begin{pmatrix} I_s & \\ & 0 \end{pmatrix} X^{-1}
\end{aligned} \tag{3.2}$$

However, we do not explicitly form $\mathbf{\Phi}$, and instead approximate it using randomization. The paper [5] under consideration states (emphasis mine)

> Instead, the basis of the eigenspace can be extracted with randomization, where the product of $\mathbf{\Phi}$ and an *appropriately chosen random matrix Y* is computed

So we form Z as

$$Z = \mathbf{\Phi} Y = \frac{1}{2\pi\mathbf{i}} \int_\Gamma (\mu I - A)^{-1} Y \mathrm{d}\mu \tag{3.3}$$

where we will repeatedly generate the random matrix Y and numerically evaluate the integral. This numerical approximation of Z is the meat of the `FastEig` algorithm.

Interestingly, one of the big differences between the approach given in this paper and previous work in this area is that the authors spent a lot of effort proving that integrating via the trapezoid rule provides optimal convergence. Previous work used Gaussian quadrature almost exclusively.

[3]This assumption is significant, but ignored in practice.

[4]While the mathematics behind spectral theory is interesting, understanding the paper from a functional analysis standpoint was not the goal of this project. Thus I'm going to gloss over some of the details here, because for our purposes, the meat of this paper, as in the paper [5], lies in the numerical computation.

3.2 Building the HSS Approximation of A

Building an HSS approximation $\widetilde{A}$ of A must be done as a precomputation step before either stage of the overall `FastEig` algorithm. In a word, an HSS approximation imposes *rank structure* on an arbitrary matrix. Rank structure is vital to the stability and convergence of the algorithms discussed in the paper.

Definition 3.1. The **rank** of a matrix is the dimension of the vector space spanned by its columns, or equivalently its rows. Rank is thus a measure of the "non-degenerateness" of the matrix.

Definition 3.2. A matrix is **rank structured** if all its off-diagonal blocks have small ranks. That is, the singular values of the off-diagonal blocks decay quickly. Here, by saying a matrix is rank structured, we mean it can be accurately approximated by a compact HSS form [5].

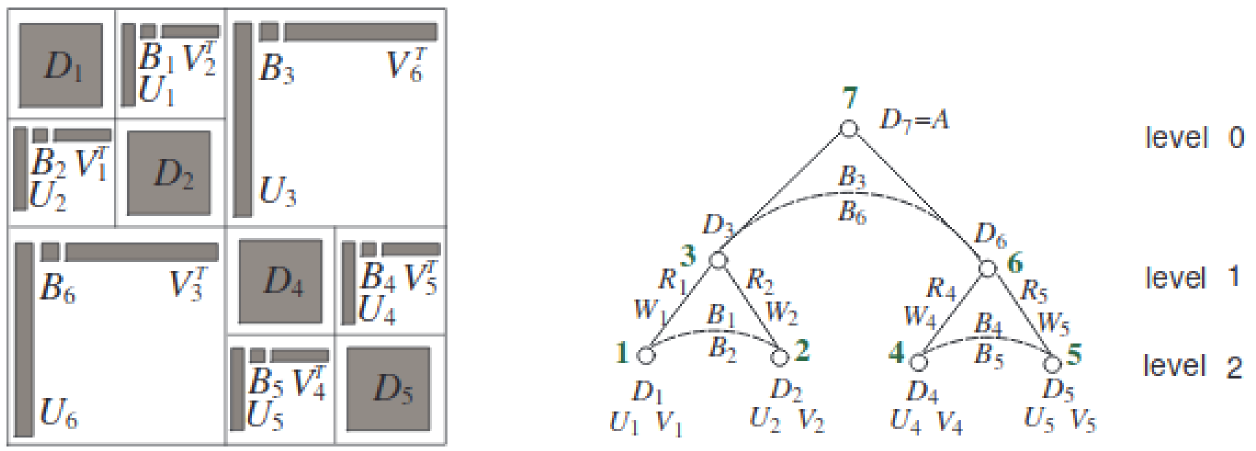


FIG. 2.1. *A 3-level HSS matrix and its corresponding HSS tree.*

Figure 4: An HSS approximation of A from Figure 2.1 of [10]

The `mat2hss` function from [6] is used as follows

```
% A is an 802 × 802 test matrix
load A.mat
% Define several metaparameters of the HSS approximation
n = 802
hss_block_row_size = 16;
tolerance = 1e-6;
[hss_tree, m] = npart(n, hss_block_row_size);

% Convert A to its HSS approximation Ã
[D, U, R, B, W, V, nflops] = mat2hss(A, hss_tree, m, 'tol', tolerance);
```

where the returned D, U, R, B, W, and V are tree-like collections of the subblocks seen in Figure 4 saved in Matlab `cell` arrays.

3.3 The EigenCounter Algorithm and its Implementation

The `EigenCounter` algorithm estimates how many eigenvalues there are inside the given circular region centered at z_0 with threshold k. If the estimated number of eigenvalues is much greater than[5] k, the `EigenCounter` algorithm should be ran again on each of its quadsected subregions.

3.3.1 The EigenCounter Algorithm

The `EigenCounter` algorithm is given below for reference in the following sections.

Algorithm 1 `EigenCounter`

```
1: function EigenCounter($\widetilde{A}$, $k$, region)              ▷ region is circular and centered at $z_0$
2:     $m \leftarrow$ some small integer                               ▷ initial number of random vectors
3:     $Y \leftarrow$ an $n \times m$ random matrix
4:     $q \leftarrow$ some number of trapezoid rule nodes
5:     Precompute $c_j = w_j(z_j - z_0)$ for $j = 1 \dots q$           ▷ Weighted nodes of the trapezoid rule
6:     Update the HSS factors of $\widetilde{A}$ to get those of $z_j I - \widetilde{A}$ for $j = 1 \dots q$
7:     repeat
8:         for $j = 1 \dots q$, compute $S_j \leftarrow (z_j I - \widetilde{A})^{-1} Y$     ▷ HSS ULV solution
9:         $\widetilde{Z} \leftarrow \frac{1}{2} \sum_{j=1}^{q} c_j S_j$                   ▷ An approximation of (3.3)
10:        $t \leftarrow \text{trace}\left(Y^T \widetilde{Z}\right)$
11:        $s \leftarrow \frac{t}{m}$                                          ▷ The current count estimate
12:        if $s$ remains the same for some number of consecutive steps then
13:            return s                                                      ▷ Count estimate identified
14:        else
15:            Append new random vector to $Y$                              ▷ Multiple vectors may be appened
16:            $m \leftarrow m + 1$
17:        end if
18:    until $s$ is much larger than $k$                                   ▷ Further quadsection needed
19: end function
```

3.3.2 Implementation Notes on the EigenCounter Algorithm

Each of the individual components of the `EigenCounter` algorithm took more time to get right than we expected. Some of the details had to be scraped together from various theorems and proofs given in the paper. Others we had to infer, and still others were blind guesses.

In hindsight, all of the pieces below make sense now, but building up to that point took quite some time. We had most of our code written at the end of the Fall semester, but did not feel we were making much progress (our eigenvalue count estimates were off by three to five orders of magnitude). The first month or two of the Spring semester exposed several assumptions and mistakes we made, and started to show our first real results. As the Spring semester progressed, we began to have more and more "ahah!" moments with an ever-increasing frequency.

The Algorithm Inputs

The `EigenCounter` function takes in a low accuracy HSS approximation of A, along with some region and a threshold, but it was useful to pass in some more information. We take in the HSS approximation along with several related metaparameters, as well as the eigenvalue count threshold, the region, the number of trapezoid rule nodes to use, and the initial number of random vectors that compose the matrix Y given in Equation 3.3. So we have

[5] What does "much greater than" mean?

```
function [count_estimate, num_iters] = Eigencounter(D, U, R, B, W, V, hss_tree,
↪  hss_subblock_size, n, threshold, region, num_rand_vectors, num_trap_nodes)
   ...
end
```

The Optimal Stopping Threshold

Note that the optimal threshold k is $\mathcal{O}(r)$ where r is the maximum off-diagonal rank as discussed in Definition 3.2. As of yet, we have only manually set the threshold, but a full implementation would need to automatically determine k from the HSS approximation.

Computing the Trapezoid Nodes and Weighted Coefficients

After generating the "appropriately chosen" random matrix[6] Y, we then compute the nodes z_j and weighted coefficients c_j using the trapezoid rule as stated in line 6 of `EigenCounter`. The paper [5] parameterizes q nodes on a circle centered at z_0 with radius r as

$$z_j = z_0 + re^{\mathbf{i}\pi t_j} \tag{3.4}$$

where

$$t_j = -1 + \frac{2(j-1)}{q} \tag{3.5}$$

for $j = 1, \ldots, q$. Then the corresponding weights w_j for each node is just $\frac{2}{q}$, and we have

$$c_j = w_j(z_j - z_0) \tag{3.6}$$

so all together we have

```
j = 1:num_points;
% See Equation 3.5
tj = -1 + 2 * (j - 1) / num_points;

% See Equation 3.4
nodes = center + radius * exp(pi * 1i * tj);
weights = (2 / num_points);

% See Equation 3.6
cj = weights * (nodes - center);
```

This was one of the "ahah!" moments that started to give us more promising results, and the details of the parameterization were buried (seemingly as an afterthought) in the proof of a theorem about the superiority of the trapezoid rule.

Shifting the HSS Factors

The algorithm then states to update the HSS factors D, U, R, B, W, and V of $\widetilde{A}$ to get those of $z_jI - \widetilde{A}$ for $j = 1, \ldots, q$. However, for "notational convenience", the paper only states how to update the HSS factors to get those of $\widetilde{A} - zI$.

[6]The paper never says what "appropriately chosen" means. We used a matrix with entries from the standard normal distribution.

All generators of $\widetilde{A} - zI$ are the same as $\widetilde{A}$, except for the diagonal D_i subblocks, which get z subtracted off of each D_i's main diagonal. E.g., $D_i \leftarrow D_i - zI$. Finding the updated subblocks for $zI - \widetilde{A}$ is equivalent to updating each $D_i \leftarrow zI - D_i$, along with negating each entry in the rest of the subblocks U_i, R_i, B_i, W_i, and V_i. We confirmed this by shifting the HSS approximation of A and comparing the results to the HSS approximation of $zI - A$.

Now, as noted, this must be done for each trapezoidal node z_j, and stored for future use. The code to shift the D subblocks is shown below

```
% Returns a list of shifted collections D of the D_i generators
function Dj = UpdateHssFactors(D, zj, hss_subblock_size)
  Dj = {};

    for z = zj
        % Append a new updated cell to the cell array
        D_shifted = UpdateD(D, z, hss_subblock_size);
        Dj = [Dj, {D_shifted}];
    end
end

% Updates a single collection D of D_i generators
function D_shifted = UpdateD(D, z_j, hss_subblock_size)
  zjI = z_j * eye(hss_subblock_size);
    D_shifted = {};

    % D is a cell array of cell arrays.
    for d_i = D
        % Each cell contains one matrix.
        subblock = d_i{1};
        if isempty(subblock)
            % Do nothing
        else
            % Shift the subblock
            subblock = zjI - subblock;
        end
        % Append the shifted subblock to the shifted cell array.
        D_shifted = [D_shifted, {subblock}];
    end
end
```

With these functions defined, we can shift the D subblocks, but now we must negate the rest of the HSS generators.

```
% Update HSS factors of Ã. Returns a cell array of cell arrays of the newly shifted D
↪   generators.
D_generators = UpdateHssFactors(D, nodes, subblock_size);

% Negate the rest of the entries of the matrix
Un = {}; Rn = {}; Bn = {}; Wn = {}; Vn = {};
for u = U
    Un = [Un, {-u{1}}];
end
for r = R
```

```
    Rn = [Rn, {-r{1}}];
end
for b = B
    Bn = [Bn, {-b{1}}];
end
for w = W
    Wn = [Wn, {-w{1}}];
end
for v = V
    Vn = [Vn, {-v{1}}];
end
```

Computing S_j
We then need to compute

$$S_j = \left(z_j I - \widetilde{A}\right)^{-1} Y \tag{3.7}$$

for each $j = 1, \ldots, q$. This is equivalent to solving $\left(z_j I - \widetilde{A}\right) S_j = Y$, which can be done column-wise. We can solve the system $AS = Y$ as follows in pseudocode

Algorithm 2 Columnwise solution of $AS = Y$

```
for $\vec{y}$ in $Y$ do
    Solve $A\vec{s} = \vec{y}$
    Append $S \leftarrow \vec{s}$
end for
```

As with the shifted HSS factors, this must be done for each z_j and stored for future use.

```
% The cell array of the q matrices.
S_matrices = {};
for j = 1:num_points
    Sj = [];
    % Solve the matrix system (z_j I - Ã) Y = S_j columnwise.
    for y_col = Y
        % The only thing that changes for each j is the jth D_i generators.
        s_col = hssulvsol(hss_tree, D_generators{j}, Un, Rn, Bn, Wn, Vn,
        ↪  length(hss_tree), y_col);
        % Append the column to S_j.
        Sj = [Sj, s_col];
    end
    % Append the new S_j matrix to the list of q matrices.
    S_matrices = [S_matrices, {Sj}];
end
```

Building the Spectral Projector Approximation $\widetilde{Z}$
After the collection of S_js have been computed, we can finally build our approximation of Equation 3.3:

$$\widetilde{Z} = \frac{1}{2} \sum_{j=1}^{q} c_j S_j \tag{3.8}$$

where c_j is the jth weighted trapezoid node, and S_j as above. We do this by

```
Z = 0;
for j = 1:num_points
    Z = Z + (coefficients(j) * S_matrices{j});
end
Z = 0.5 * Z;
```

Updating the Eigenvalue Count Estimate

Now we compute our current-step eigenvalue count estimate $\frac{t}{m}$ where $t = \text{trace}\left(Y^T \widetilde{Z}\right)$.

```
total_trace = trace(Y' * Z);
count_estimate = total_trace / num_rand_vectors;
```

This is a departure we made from the algorithm as stated in the paper [5]. The paper states that we should increment $t \leftarrow t + \text{trace}\left(Y^T \widetilde{Z}\right)$ on each iteration, but we are quite certain this is a typo[7].

Note also that this count estimate is a complex number due to computing the trace of a complex-valued matrix. It is not usually possible to have a complex number of eigenvalues, so we take the magnitude `abs(count_estimate)` as our estimated eigenvalue count.

Stopping Conditions

This is essentially the end of the algorithm. All that is left is to tie up some loose ends related to the stopping conditions. The algorithm as stated is quite loose with its stopping conditions. This is something that we have played with and have yet to find a good answer to.

Because the count estimates are `complex double`s, implementing the paper's suggested stopping condition of checking for repeated estimates inherently requires some kind of tolerance.

However, the actual estimates tend to not repeat, even if they do settle down and appear to converge. Thus we use the following stopping condition

```
error = abs(count_estimate - old_count_estimate) / abs(count_estimate);
old_count_estimate = count_estimate;

if error < 0.01
    return;
else
    Y = [Y, randn(n, 1)];
    num_rand_vectors = num_rand_vectors + 1;
end
```

but this was intended to be a temporary solution.

3.4 SFeast

The **SFeast** algorithm takes in a suitable search region, along with the estimated number of eigenvalues within that region, and returns the estimated eigenvalues inside that search region. The algorithm requires a higher

[7]There have been other small typos and grammatical errors in the paper, and it doesn't make sense to add up the traces repeatedly. Numerical experiments also confirm this belief, as does Equation (4.8) on page 18 of the paper: $\#_\Lambda(A, \mathcal{C}_\gamma(z_0)) \approx \frac{1}{m}\text{trace}\left(Y^T \widetilde{Z}\right)$. Note that there is no indication that the trace should be repeatedly added to itself as the algorithm progresses.

accuracy HSS approximation of A to be precomputed before using the algorithm, similar to the `EigenCounter` algorithm.

We did not implement this algorithm due to our struggles implementing the `EigenCounter` algorithm. The `SFeast` algorithm has just as many pieces missing, that need to be filled in by the surrounding context[8], as what we have done so far.

Algorithm 3 `SFeast`

```
1:  function SFEAST($\widetilde{A}$, $s$, region)        ▷ $s$ is the estimated # of eigenvalues inside region
2:      Initialize $\widehat{\Lambda}, \widehat{X}, \widehat{Q} \leftarrow \emptyset$        ▷ $\widehat{Q}$ is the convergent eigenspace, $(\widehat{\Lambda}, \widehat{X})$ the eigenpairs
3:      Generate $Y \leftarrow n \times \left(\frac{2}{3}s\right)$ random matrix
4:      $q \leftarrow$ number of trapezoid rule nodes
5:      Precompute $c_j = w_j(z_j - z_0)$ for $j = 1 \dots q$        ▷ Weighted nodes of the trapezoid rule
6:      Update the HSS factors of $\widetilde{A}$ to get those of $z_j I - \widetilde{A}$ for $j = 1 \dots q$
7:      repeat
8:          for $j = 1 \dots q$, compute $S_j \leftarrow (z_j I - \widetilde{A})^{-1} Y$        ▷ HSS ULV solution
9:          $\widetilde{Z} \leftarrow \frac{1}{2} \sum_{j=1}^{q} c_j S_j$        ▷ An approximation of Equation 3.3
10:         $Q \leftarrow$ basis of $\widetilde{Z}$ orthonormalized w.r.t $\widehat{Q}$
11:         $\widehat{A} \leftarrow Q^T \widetilde{A} Q$        ▷ Reduced problem via HSS matrix-vector multiplication
12:         Solve $\widehat{A} = \widetilde{X}\widetilde{\Lambda}\widetilde{X}^{-1}$        ▷ Solve the reduced eigenvalue problem
13:         $Y \leftarrow Q\widetilde{X}$        ▷ Recovery of approximate eigenvectors of $A$
14:         $\begin{pmatrix}\widehat{\Lambda}_1 & \widehat{\Lambda}_2\end{pmatrix} \leftarrow \widehat{\Lambda}$        ▷ Partition with convergent eigenvalues in $\widehat{\Lambda}_1$
15:         $(Y_1 \quad Y_2) \leftarrow Y$        ▷ Partition with convergent eigenvectors in $Y_1$
16:         $(Q_1 \quad Q_2) \leftarrow Q$        ▷ Partition with convergent eigenspace in $Q_1$
17:         Set $\widehat{\Lambda} \leftarrow \text{diag}\left(\widehat{\Lambda}, \widehat{\Lambda}_1\right)$, $\widehat{X} \leftarrow \begin{pmatrix}\widehat{X} & X_1\end{pmatrix}$, and $\widehat{Q} \leftarrow \begin{pmatrix}\widehat{Q} & Q_1\end{pmatrix}$        ▷ Finding $X_1$ left as exercise to reader
18:         $Y \leftarrow Y_2$
19:     until convergence
20:     return $\widehat{\Lambda}, \widehat{X}$
21: end function
```

3.5 `FastEig`

This is the glue code that ties together the `EigenCounter` and `SFeast` algorithms. We first take an initial search region and look for target regions. Then we take the target regions, along with their estimated eigenvalue counts, and attempt to find each eigenvalue.

[8]Typos aside, I think this may just be a part of the research process.

Algorithm 4 `FastEig`

```
function FastEig($A$, $\Gamma$, $k$)                                   ▷ $\Gamma$ is the initial search region
    Push the initial search $\Gamma$ onto a stack $\mathcal{S}$
    $\mathcal{Q} \leftarrow$ a queue of target regions to process
                                                                        ▷ Stage 1
    Precompute a low accuracy HSS approximation $\widetilde{A}$ from $A$
    while $\mathcal{S} \neq \emptyset$ do
        Pop subregion $\mathcal{R}$ from stack $\mathcal{S}$
        Find the smallest circle $\mathcal{C}_\gamma(z_0)$ that encloses $\mathcal{R}$
        $s \leftarrow$ EigenCounter($\widetilde{A}, \mathcal{C}_\gamma(z_0), k$)
        if $s \leq k$ then                         ▷ No further quadsection needed, mark as target
            Save $\mathcal{R}$, along with $s$, in the queue $\mathcal{Q}$
        else
            Quadsect $\mathcal{R}$, and push the four subregions onto $\mathcal{S}$
        end if
    end while
                                                                        ▷ Stage 2
    Precompute a high accuracy HSS approximation $\widetilde{A}$ from $A$
    Initialize $\Lambda, X \leftarrow \emptyset$                        ▷ The approximated eigenpairs
    for each target region $\mathcal{R}$ and corresponding $s$ in $\mathcal{Q}$ do
        $\widehat{\Lambda}, \widehat{X} \leftarrow$SFeast($\widetilde{A}, \mathcal{R}, s$)      ▷ With "minor" modifications...
        $\Lambda \leftarrow \operatorname{diag}\left(\Lambda, \widehat{\Lambda}\right)$
        $X \leftarrow \begin{pmatrix} X & \widehat{X} \end{pmatrix}$
    end for
    return $\Lambda, X$
end function
```

There is nothing that says we have to run the two stages serially; running the two algorithms concurrently is low hanging fruit that could improve the performance of the overall algorithm. Although some analysis would have to be done on the performance of each algorithm to determine how many instances could be running at a time.

The use of a queue in this algorithm is not stated in the paper (all it says is to save the regions and counts), but makes sense as an implementation detail, especially if these algorithms are to run concurrently to facilitate passing target regions to `SFeast`.

4 Numerical Results

Due to time, hardware, and patience constraints, we limited our tests to relatively small (100×100) matrices. Further, we tested our algorithm on a special type of matrix that the paper [5] refers to as "Cauchy-like".

Definition 4.1. A **Cauchy-like** matrix A is a matrix of the form

$$A_{ij} = \frac{u_i v_j}{s_i - t_j}$$

where $s_i = e^{\frac{2i\pi \mathbf{i}}{n}}$ and $t_j = e^{\frac{(2j+1)\pi \mathbf{i}}{n}}$ are on the unit circle, and $\{u_i\}_{i=1}^n$ and $\{v_j\}_{j=1}^n$ are random. Matrices of this form are known to be rank structured.

Thus we built our test matrices like so

```
function A = TestMatrix(n)
  % Generates 'Cauchy-like' $n \times n$ matrices.
  u = randn(n, 1);
  v = randn(n, 1);

  % $s_i = e^{\frac{2i\pi\mathbf{i}}{n}}$
  s = exp((2 * pi * 1i / n) .* (1:n));

  % $t_j = e^{\frac{(2j+1)\pi\mathbf{i}}{n}}$
  t = exp((2 .* (1:n) + 1) * pi * 1i / n);

  % $A_{kj} = \frac{u_k v_j}{s_k - t_j}$
  for k = 1:n
    for j = 1:n
      A(k, j) = (u(k) * v(j)) / (s(k) - t(j));
    end
  end
end
```

Then to test our implementation of the `EigenCounter` algorithm, we picked (somewhat arbitrarily) a number of the algorithm parameters:

- The matrix size $n = 100$, picked for more timely execution.
- The number $m = 1$ of initial random vectors to start with.
- The number $q = 10$ of trapezoid rule nodes.
- The region of interest. We picked a circular region with radius 3 centered at $4 - 7i$.
- The HSS block row size. We picked 10 somewhat arbitrarily. From what little example code we had of an HSS approximation, this seemed reasonable.
- The HSS approximation tolerance. For the `EigenCounter` algorithm, we use a low accuracy HSS approximation for speed. We used 1×10^{-2} as our HSS tolerance.
- The stopping threshold $k = 60$. We picked this parameter because it was typically above the kinds of values that we saw the `EigenCounter` algorithm converge to.

Then we repeatedly generated a new test matrix, found its eigenvalues with the built-in (and faster) Matlab `eig()` function, and counted how many eigenvalues are actually within our search region. Then we produced an HSS approximation and ran the `EigenCounter` algorithm on the search region with the above parameters.

So, in code, we had the following.

```
num_trials = 20;
n = 100;
radius = 3;
num_rand_vectors = 1;
num_trap_points = 10;
center = complex(4, -7);
region = CSquareRegion(real(center), imag(center), radius / sqrt(2));
hss_block_row_size = 10;
hss_tol = 1e-2;
threshold = 60;
```

```
correct = 0;
for trial = 1:num_trials
    A = TestMatrix(n);
    E = eig(A);
    % Shift the eigenvalues to the origin and check how many are within `radius'
    num_eigs = sum(abs(E - center) <= radius);
    fprintf('(%d) actual number of eigenvalues: %d\n', trial, num_eigs);

    % Produce the HSS approximation Ã of A
    [hss_tree, m] = npart(n, hss_block_row_size);
    [D, U, R, B, W, V, nflops] = mat2hss(A, hss_tree, m, 'tol', hss_tol);

    [count_estimate, num_iters] = Eigencounter(D, U, R, B, W, V, hss_tree,
    ↪  hss_block_row_size, n, threshold, region, num_rand_vectors, num_trap_points);

    fprintf('(%d) estimated number of eigenvalues: %s\tmagnitude: %f\n', trial,
    ↪  num2str(count_estimate), abs(count_estimate));
    fprintf('(%d) number of iterations: %d\n', trial, num_iters);

    % If the count estimate is close to the actual value, increment the correct counter
    if round(abs(count_estimate)) == num_eigs
        correct = correct + 1;
    end
end
fprintf('error rate: %f\n', 1 - correct / num_trials);
```

Which produced output like the following, given only once for completeness.

```
>> full_test_counter
(1) actual number of eigenvalues: 3
(1) estimated number of eigenvalues: 3.0538-0.93253i magnitude: 3.193000
(1) number of iterations: 46
(2) actual number of eigenvalues: 0
(2) estimated number of eigenvalues: 0.079902-0.093039i magnitude: 0.122640
(2) number of iterations: 43
(3) actual number of eigenvalues: 2
(3) estimated number of eigenvalues: 1.8861-0.29501i magnitude: 1.908994
(3) number of iterations: 52
(4) actual number of eigenvalues: 1
(4) estimated number of eigenvalues: 0.88637+0.013062i magnitude: 0.886465
(4) number of iterations: 20
(5) actual number of eigenvalues: 3
(5) estimated number of eigenvalues: 2.6129-1.6326i magnitude: 3.081016
(5) number of iterations: 33
(6) actual number of eigenvalues: 5
(6) estimated number of eigenvalues: 3.7217-0.11552i magnitude: 3.723505
(6) number of iterations: 27
(7) actual number of eigenvalues: 1
(7) estimated number of eigenvalues: 0.87669+0.87801i magnitude: 1.240754
(7) number of iterations: 79
(8) actual number of eigenvalues: 1
(8) estimated number of eigenvalues: 1.5297-0.32048i magnitude: 1.562865
```

```
(8) number of iterations: 25
(9) actual number of eigenvalues: 2
(9) estimated number of eigenvalues: 1.3507-1.328i magnitude: 1.894232
(9) number of iterations: 31
(10) actual number of eigenvalues: 0
(10) estimated number of eigenvalues: 0.088653-0.22468i magnitude: 0.241540
(10) number of iterations: 57
(11) actual number of eigenvalues: 2
(11) estimated number of eigenvalues: 4.113+0.051333i magnitude: 4.113305
(11) number of iterations: 10
(12) actual number of eigenvalues: 4
(12) estimated number of eigenvalues: 5.8464-1.7098i magnitude: 6.091258
(12) number of iterations: 41
(13) actual number of eigenvalues: 1
(13) estimated number of eigenvalues: 1.6135+0.32967i magnitude: 1.646813
(13) number of iterations: 14
(14) actual number of eigenvalues: 2
(14) estimated number of eigenvalues: 1.6217-1.2509i magnitude: 2.048093
(14) number of iterations: 59
(15) actual number of eigenvalues: 2
(15) estimated number of eigenvalues: 2.4744+2.8212i magnitude: 3.752618
(15) number of iterations: 51
(16) actual number of eigenvalues: 1
(16) estimated number of eigenvalues: 1.0636+0.80696i magnitude: 1.335097
(16) number of iterations: 46
(17) actual number of eigenvalues: 0
(17) estimated number of eigenvalues: -0.074699-0.21963i magnitude: 0.231984
(17) number of iterations: 52
(18) actual number of eigenvalues: 1
(18) estimated number of eigenvalues: 1.4566+0.70049i magnitude: 1.616265
(18) number of iterations: 38
(19) actual number of eigenvalues: 1
(19) estimated number of eigenvalues: 1.253+0.41404i magnitude: 1.319622
(19) number of iterations: 45
(20) actual number of eigenvalues: 0
(20) estimated number of eigenvalues: -0.047901+0.035575i magnitude: 0.059666
(20) number of iterations: 53
error rate: 0.350000
```

Which, in this case, had an accuracy of 65% (13 out of 20 correct) and took 1097 seconds to run all twenty trials.

Running the Matlab profiler (selecting "Run and Time") gave the results summarized in Table 1. The table gives the total amount of time spent inside a given function, as well as the amount of time spent in a function excluding calls to child functions.

As we can see from Table 1, the vast majority of the time spent running the `EigenCounter` algorithm is spent solving the S_j matrix system in Equation 3.7.

So to experiment, I removed the call to `hssulvsol` by converting the shifted HSS approximation $z_j I - \widetilde{A}$ to a matrix A_j and then using the built-in Matlab matrix system solver, i.e., I changed

```
S_matrices = {};
for j = 1:num_points
```

Table 1: A summary of the profiler results

Function Name	Calls	Total Time	Self Time
`full_test_counter`	1	1097.101	0.195
`Eigencounter`	20	1095.863	4.169
`hssulvsol`	208130	1091.628	788.847
`mat2hss`	20	0.544	0.115

```
    Sj = [];
    % Solve the matrix system columnwise.
    for y_col = Y
        s_col = hssulvsol(hss_tree, D_generators{j}, Un, Rn, Bn, Wn, Vn,
        ↪  length(hss_tree), y_col);
        % Append the column to the result.
        Sj = [Sj, s_col];
    end
    % Append the new S matrix.
    S_matrices = [S_matrices, {Sj}];
end
```

to

```
S_matrices = {};
for j = 1:num_points
    Sj = [];
    % Solve the matrix system columnwise.
    for y_col = Y
        Aj = hss2mat(D_generators{j}, Un, Rn, Bn, Wn, Vn, hss_tree);
        s_col = Aj \ y_col;
        % Append the column to the result.
        Sj = [Sj, s_col];
    end
    % Append the new S matrix.
    S_matrices = [S_matrices, {Sj}];
end
```

Note however, this is not a true elimination of the burden of HSS computations, because we still compute the HSS factorization, shift the HSS approximation, and then convert the HSS approximation back into a normal matrix.

It took 617 seconds to run the test script and had an accuracy of 75% (15 out of 20 correct).[9]

The profiler results are summarized in Table 2.

This is not quite a fair comparison though, because we are converting each shifted HSS approximation back to its matrix form *a lot*.

So we did slightly more refactoring and tested the code without any HSS factorizations. I.e., we changed

[9]Note that it's really quite unfair to compare two different implementations of an algorithm (especially one that works via randomization) by only one run of each.

Table 2: Profiler results with HSS computations almost removed

Function Name	Calls	Total Time	Self Time
`full_test_counter`	1	616.909	0.211
`Eigencounter`	20	615.665	111.713
`hss2mat`	214880	503.883	461.239
`mat2hss`	20	0.547	0.110

```
S_matrices = {};
for j = 1:num_points
    Sj = [];
    % Solve the matrix system columnwise.
    for y_col = Y
        Aj = hss2mat(D_generators{j}, Un, Rn, Bn, Wn, Vn, hss_tree);
        s_col = Aj \ y_col;
        % Append the column to the result.
        Sj = [Sj, s_col];
    end
    % Append the new S matrix.
    S_matrices = [S_matrices, {Sj}];
end
```

to

```
S_matrices = {};
for j = 1:num_points
    % $z_j I - A$
    Aj = nodes(j) * eye(n) - A;
    Sj = [];
    % Solve the matrix system columnwise.
    for y_col = Y
        s_col = Aj \ y_col;
        % Append the column to the result.
        Sj = [Sj, s_col];
    end
    % Append the new S matrix.
    S_matrices = [S_matrices, {Sj}];
end
```

along with some additional changes in the glue code to pass A to the `EigenCounter` function instead of the HSS approximation $\widetilde{A}$ and its metaparameters. The results were enlightening.[10] I got similar results to the earlier implementations as far as accuracy and behavior go, yet the test script ran in a measly 73 seconds (as opposed to 1097 seconds). The profiler results are summarized in Table 3.

We can see that the computation time diminished by several orders of magnitude! I suspect that using the HSS approximations become more useful with a faster HSS approximation implementation, as well as for larger matrices where using the Matlab system solver becomes untenable. I *think* that the HSS approximation step produces gains in algorithmic complexity as a heavy performance cost, especially for repeated calls, as

[10] And a bit disheartening. . .

Table 3: Profiler results with HSS computations completely eliminated

Function Name	Calls	Total Time	Self Time
`full_test_counter`	1	73.327	0.208
`Eigencounter`	20	72.651	72.627

well as better stability when considering rank structured matrices. The paper [5] does not state what kind of speedup they were able to achieve by using HSS approximations.

5 Conclusion

Recall from Section 1.4 that our original goals were to understand, implement, and improve on the algorithms given in [5]. However, in our implementation of the `EigenCounter` algorithm, we struggled to get results that made sense. We made substantial progress though, and now have a much better understanding of how the algorithm works. As summarized in Section 3.3.2, we struggled with the following

- Finding the right trapezoid rule node parameterization
- Shifting the HSS factors for each trapezoid rule node
- Updating the eigenvalue count estimate correctly
- Finding the right stopping condition

Our very first numerical results were eigenvalue count estimates several orders of magnitude higher than the actual counts. Not only that, but the count estimates were complex-valued with large magnitudes of both the real and imaginary parts. Parameterizing the trapezoid rule nodes in the manner described by the paper had the biggest impact. Once we changed our parameterization to match the paper's, the magnitudes of the current-step eigenvalue count estimates immediately improved to within the same order of magnitude of the actual counts.

Once we reread the paper several times, we realized the eigenvalue counts did not seem to be converging, and in fact steadily increased with every iteration. This was an immediate clue that we were not updating the eigenvalue count estimate correctly. Incrementing the count estimate as the algorithm listing in the paper [5] indicates did not fit with our intuitive understanding of the algorithm, and did not fit with the equations given throughout the rest of the paper.

Once we changed the implementation of the eigenvalue count updates, we began to see the count estimates appear to converge. Further, they actually began to converge to the actual number of eigenvalues within the contour of interest. The convergence condition as given by the paper [5] is that the count estimate should repeat some number of times. Now, as the count estimates were `complex double`s, they never repeated exactly. Perhaps the paper intended for the estimates to be rounded, and then checked for repeated values. In any case, we ultimately settled on a simple relative error convergence condition.

After overcoming these roadblocks, we were able to test our `EigenCounter` implementation as summarized in Section 4 using a Cauchy-like matrix as defined in Definition 4.1, which as noted, is known to be rank structured. In these tests we saw an average accuracy of $\sim 75\%$, which as noted in Section 1.5, is expected due to the presence of a divergence condition in the `EigenCounter` algorithm.

6 Future Work

Even though we satisfied our revised goals, there is still work that could be done in this area. We have identified the following topics that could be examined in greater depth.

- Future work in this area could go in many different directions. Section 4 detailed the impact HSS approximations had on performance. This is one area that could be researched; understanding how and why we would want to use HSS approximations, or even just developing an efficient library for HSS computations.
- The Structured FEAST algorithm is an extension of the FEAST algorithm developed by Polizzi [4]. The differences between SFEAST and FEAST could be examined.
- We hand-waved our way around the spectral projector in Equation 3.2 and Equation 3.3. Researching the mathematical theory behind our numerical computation could very well be its own research project.
- We picked most of the unspecified parameters based on instinct without much justification. Future work could look at how to choose the right values for these parameters.
- Future work could also look at a full, tested implementation and attempt to more rigorously improve its performance and/or accuracy. This would involve cleaning up our current implementation of `EigenCounter` as well as implementing `SFeast` and `FastEig` in addition to a full suite of tests.

Appendices

A Supplemental Content

The source code for this project may be found at `https://git.agill.xyz/nots/math_research`.

References

[1] L. Trefethen and D. Bau, *Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, 1997.

[2] T. Sakurai and H. Sugiura, "A projection method for generalized eigenvalue problems," *Journal of Computational and Applied Mathematics*, vol. 159, no. 1, pp. 119–128, 2003.

[3] A. Imakura, L. Du, and T. Sakurai, "A map of contour integral-based eigensolvers for solving generalized eigenvalue problems," *Japan Journal of Industrial and Applied Mathematics*, November 2016.

[4] E. Polizzi, "A density matrix-based algorithm for solving eigenvalue problems," *CoRR*, vol. abs/0901.2665, 2009.

[5] X. Ye, J. Xia, R. H. Chan, S. Cauley, and V. Balakrishnan, "A fast contour-integral eigensolver for non-hermitian matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 38, no. 4, pp. 1268–1297, 2017.

[6] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, "A hierarchically semiseparable (hss) matrix package." `https://www.math.purdue.edu/~xiaj/work/hss.zip`, 2010.

[7] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, "Fast algorithms for hierarchically semiseparable matrices," *Numerical Linear Algebra with Applications*, vol. 17, no. 6, pp. 953–976, 2010.

[8] P. G. Martinsson, "A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1251–1274, 2011.

[9] S. Chandrasekaran, M. Gu, and T. Pals, "A fast ULV decomposition solver for hierarchically semiseparable representations," *SIAM Journal on Matrix Analysis and Applications*, vol. 28, no. 3, pp. 603–622, 2006.

[10] Y. Xi and J. Xia, "On the stability of some hierarchical rank structured matrix algorithms," *SIAM Journal on Matrix Analysis and Applications*, vol. 37, no. 3, pp. 1279–1303, 2016.