Survey of Numerical Methods

An Investigation of Four Relevant Problems in Numerical Analysis

Ishan Muzumdar IHM5018@PSU.EDU
Advanced Numerical Methods
Pennsylvania State University
Department of Computer Science

Abstract

With advances in data mining and technology, scientific computing has become a useful tool in solving several important quantitative problems. Many computational tools in science, particularly in mechanics and statistics, require the use of advanced numerical methods to achieve fast convergence of solutions and reliable parameter estimates. In this report, we examine four important and relevant problems in numerical analysis. We first provide a brief motivation of each problem. Then, we explore some methods and techniques to address these problems, providing some code samples (when applicable) and commentary on how these methods improve on previous work. In completing this report, we hope that readers develop a firmer grasp on the surveyed numerical methods and how to apply them in a practical setting.

Keywords: Non-linear least squares, divide and conquer algorithms, eigenvalue problems, generalized Euler's method, orthogonal polynomials, nonhomogeneous ODEs

1 Overview

Below are the four problems we examine in this paper:

1. Nonlinear least square problems: Implementing the Levenberg-Marquart method.
2. Integrals of orthogonal polynomials: Developing recurrence relations for integrals to generate orthogonal polynomials.
3. Eigenvalue problems for tri-diagonal matrices: Deriving efficient algorithms that make use of the special structure.
4. Stability and convergence analysis: Evaluating important properties of generalized Euler's method to solve nonhomogeneous ODEs

2 Non-Linear Least Square Problems

Non-linear least squares estimation (LSE) is a branch of least squares methods that seeks to fit a set of *m* observations to a nonlinear model with *n* parameters. Non-linear least squares commonly arises in statistics, specifically within several regression problems, including Box-Cox regressor transformation, logistic link models, and probit regression. It also emerges in many minimization and equilibrium problems across academia. For example, economists employ non-linear least squares to calculate equilibria prices for markets with *n* competing goods.

Unlike many linear LSE problems, non-linear least squares does not have an explicit formula to estimate model parameters. Instead, iterative methods are commonly used to calculate parameter values. The most traditional method for non-linear LSE is Newton's Method, which is a standard approach to solve problems with non-linear systems of equations. The derivation for Newton's Method relies on calculating the gradient of the sum of squared errors and finding parameters such that this gradient is zero.

Below is MATLAB code for Newton's method. Here, we have that *r* is a vector of residuals, *x0* is the initial guess, and *J* is the Jacobian of *r*. Notice how the iteration is very similar to the normal equation found in linear LSE. In a sense, we are applying an iterative "normal equation" to solve this problem.

```
function [x,err] = newtons_method(r, x0, J, tol, maxIter)
    err = [];
    iter = 0;
    x = x0;
    while iter < maxIter
        A = J(x);
        mat = transpose(A) * A;
        b = transpose(-A) * r(x);
        v = linsolve(mat, b);
        x = x + transpose(v);
        err_1 = norm(x - x0);
        err = [err,err_1];
        if err_1 < tol
           break
        end
        iter = iter + 1;
        x0 = x;
    end
    disp("reached max number of iterations")
end
```

Although Newton's method is usually fast reliable, it sometimes can fail to converge. In general, Newton's method is fairly susceptible to local minima and poorly suited for ill-conditioned matrices. To address these issues, we can apply the Levenberg-Marquadt Algorithm (LMA). LMA is very similar to Newton's Method but applies a slightly different iterative formula, which provides a correction to the direction of each descent throughout the algorithm (highlighted in yellow). As a result, LMA is less prone to local minima in the optimization space. Furthermore, LMA features an additional hyperparameter, λ, which acts as a regularization term for the algorithm. The regularization effect of lambda is particularly useful for ill-conditioned matrices, as we will soon see.

Below is MATLAB code for the LMA. All the variables are the same as noted above, with our hyperparameter, lambda. Notice that when lambda = 0, we recover Newton's method.

```
function [x,err] = levenberg_marquadt(r, x0, J, tol, maxIter, lambda)
    err = []; iter = 0;x = x0;
    while iter < maxIter
        A = J(x);
        mat = transpose(A) * A;
        Matrix = mat + lambda*diag(diag(mat));
        b = transpose(-A) * r(x);
        v = linsolve(Matrix, b);
        x = x + transpose(v);
        err_1 = norm(x - x0);
        err = [err,err_1];
        if norm(x - x0) < tol
           return
        end
        iter = iter + 1;
        x0 = x;
        disp(err_1)
    end
    disp("reached max number of iterations")
end
```

We now apply these methods to two different problems. First, we consider a small set of data set: $(t_i, y_i) = \{(1,3), (2,5), (2,7), (3,5), (4,1)\}$ fit to the model $y = c_1 e^{-c_2(t-c_3)^2}$. Here, the c_i values are our model parameters. This problem is example 4.25 of Sauer's Numerical Analysis.

```
%textbook example
r = @(c)[c(1)*exp(-c(2)*(1-c(3))^2) - 3; c(1)*exp(-c(2)*(2-c(3))^2) - 5;
         c(1)*exp(-c(2)*(2-c(3))^2) - 7; c(1)*exp(-c(2)*(3-c(3))^2) - 5;
         c(1)*exp(-c(2)*(4-c(3))^2) - 1];

j_row = @(c,t)[exp(-c(2)*(t-c(3))^2),
               -c(1)*(t-c(3))^2 * exp(-c(2)*(t-c(3))^2),
                2*c(1)*c(2)*(t-c(3)) * exp(-c(2)*(t-c(3))^2)];

J = @(c)[j_row(c,1);j_row(c,2);j_row(c,2);j_row(c,3);j_row(c,4)];

lambda=25;
maxIter=10000000;
tol=1e-5;
x0=[1,1,1];
lamdba = 0;

[newton_soluation_tb, err_nm] = newtons_method(r, x0, J, tol, maxIter);
[lm_solution_tb, err_lm] = levenberg_marquadt(r, x0, J, tol, maxIter, 0);
```

When we apply Newton's method, our solution diverges because the matrix $A^T A$ is ill-conditioned. This results in the following output:

```
Warning: Matrix is singular, close to singular or badly scaled. Results may be
inaccurate. RCOND = NaN.
```

However, when we run LMA on the same data, we obtain a convergent solution, determining that: $\vec{c} = (c_1,\ c_2,\ c_3) = (6.3001, 0.5087, 2.2487)$. In the following figure, we plot the error of LMA over each iteration.

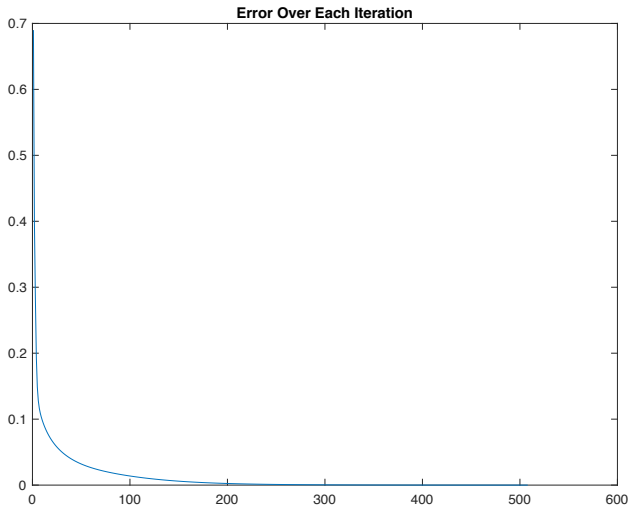


Figure 1: *Error of LMA for problem 1*

For the second problem, we apply Newton's method and LMA to triangulate the location of a GPS receiver using three satellite signals, while using a fourth satellite to account for the delay in satellite transmissions. The details of the problem are available on page 238 of Sauer's Book.

To solve this problem, we record the locations of each satellite in $\mathbb{R}^3$ by (A_i, B_i, C_i) and the transmission time for each satellite's signal by $t_i, i = 1,2,3,4$. Below is the non-linear system of equations we are solving. The variables of interest are *(x, y, z, d)*.

$$r_1(x,y,z,d) = \sqrt{(x-A_1)^2 + (y-B_1)^2 + (z-C_1)^2} - c(t_1 - d) = 0$$
$$r_2(x,y,z,d) = \sqrt{(x-A_2)^2 + (y-B_2)^2 + (z-C_2)^2} - c(t_2 - d) = 0$$
$$r_3(x,y,z,d) = \sqrt{(x-A_3)^2 + (y-B_3)^2 + (z-C_3)^2} - c(t_3 - d) = 0$$
$$r_4(x,y,z,d) = \sqrt{(x-A_4)^2 + (y-B_4)^2 + (z-C_4)^2} - c(t_4 - d) = 0$$

We now apply both Newton's method and LMA and compare the final errors of each method, along with the number of iterations required for convergence. Note that to find a suitable choice for λ here, we apply Armijo's method. The MATLAB code for this problem is below.

One downside of LMA, which we will quickly see, is that the regularization term results in LMA often requiring far more iterations to converge to a final solution than Newton's method. This property, in conjunction with LMA's higher time complexity, sometimes makes LMA less suitable for non-linear LSE than Newton's method.

```
%Satellite Example
r = @(x)[sqrt((x(1)-15600)^2 + (x(2)-7540)^2 + (x(3)-20140)^2) -
299792.458*(.07074-x(4));
         sqrt((x(1)-18760)^2 + (x(2)-2750)^2 + (x(3)-18610)^2) -
299792.458*(.07220-x(4));
         sqrt((x(1)-17610)^2 + (x(2)-14630)^2 + (x(3)-13480)^2) -
299792.458*(.07690-x(4));
         sqrt((x(1)-19170)^2 + (x(2)-610)^2 + (x(3)-18390)^2) -
299792.458*(.07242-x(4))];

j_row = @(x,pos)[(x(1)-pos(1))/sqrt((x(1)-pos(1))^2 + (x(2)-pos(2))^2 + (x(3)-
pos(3))^2)...
                 (x(2)-pos(2))/sqrt((x(1)-pos(1))^2 + (x(2)-pos(2))^2 + (x(3)-
pos(3))^2)...
                 (x(3)-pos(3))/sqrt((x(1)-pos(1))^2 + (x(2)-pos(2))^2 + (x(3)-
pos(3))^2)...
                 299792.458];

pos1 = [15600,7540,20140,.07074];
pos2 = [18760,2750,18610,.07220];
pos3 = [17610,14630,13480,.07690];
pos4 = [19170,610,18390, .07242];

J = @(x)[j_row(x,pos1);j_row(x,pos2);j_row(x,pos3);j_row(x,pos4)];

maxIter=10000000;
tol=1e-7;
x0=[0,0,6370,0];

[satelite_sol_nm, err_satelite_nm] = levenberg_marquadt(r, x0, J, tol, maxIter,
0);
[satelite_sol,err_satelite_lm] = levenberg_marquadt(r, x0, J, tol, maxIter,
lambda);
disp(satelite_sol);
```

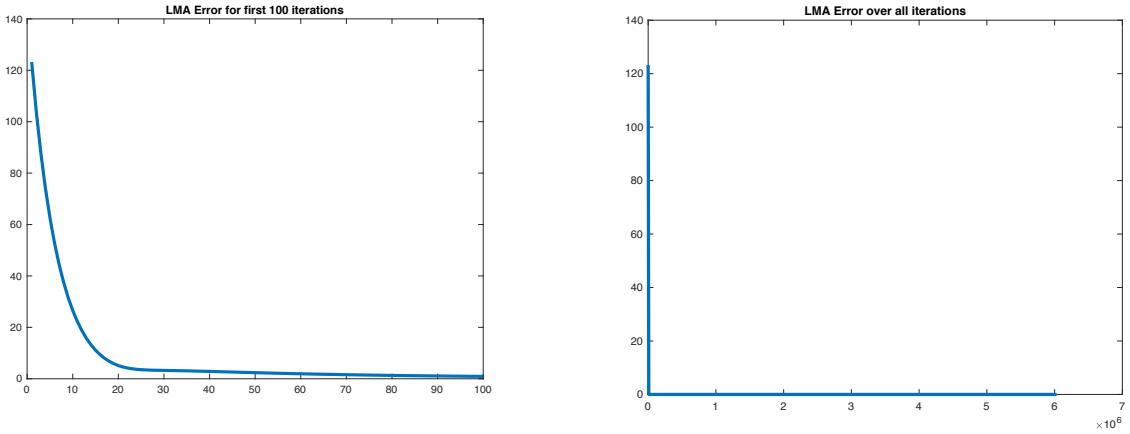
From the above code, we obtain the following solutions:

$$Newton's\ Method: (x, y, z, d) = (-41.7727, -16.7891, 6370.0595, -3.2015 * 10^{-3})$$
$$LMA: (x, y, z, d) = (-41.7681, -16.7875, 6370.0613, 3.2015 * 10^{-3})$$

We can see that both solutions are very close to the true solution listed in the problem:

$$(-41.7727, -16.7891, 6370.0596, -3.201566 * 10^{-3}).$$

However, what's particularly interesting is the speed of convergence. From the above code, we obtain that Newton's method converges in 4 iterations, while LMA requires 6029145 iterations.



Figures 2 and 3: *plots of the error in the LMA method for the first 100 iterations (left) and over all iterations (right). Notice how quickly the error slope flattens, leading to slow convergence.*

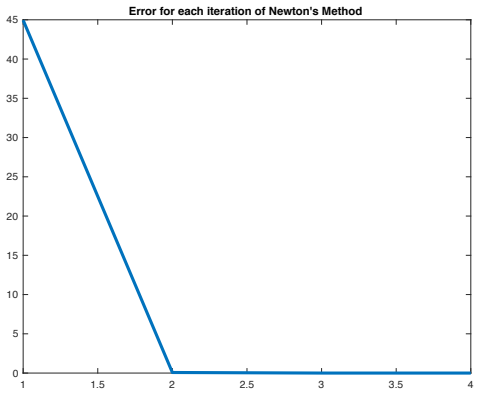


Figure 4: *plots of the error in Newton's method, which converges after 4 iterations.*

One reason for this extremely stark difference in convergence rates is the initial guess. Notice that the initial guess (highlighted in the MATLAB code) is very close to the true solution. When the initial guess is fairly strong, Newton's method is able to converge far more quickly. Secondly, the regularization parameter in LMA results in the method essentially "dancing" around the true solution, yielding very slow convergence.

3 Integrals and Recurrence Relations for Orthogonal Polynomials

For our second problem, we examine a set of orthogonal polynomials, which are sets of polynomials which are mutually orthogonal over some given inner product. Orthogonal polynomials are incredibly powerful mathematical tools, with a variety of applications in differential equations, probability density estimation, numerical quadrature, and even deep learning. Some well-known collections of orthogonal polynomials are Laguerre, Legendre, Hermite, and Jacobi polynomials.

For this particular problem, we consider orthogonal polynomials over the interval [0,1] with weight function $w(x) = \sqrt{x}$. This particular weight function is common in iterative weighted least squares to model heteroskedastic data, or data with nonconstant variance. We first solve for the first 6 orthogonal polynomials in this set. Then, we consider some useful properties of these polynomials results that simplify their calculation. Here, our inner product is defined as

$$\langle f, g \rangle_w = \int_0^1 f(x)g(x)\sqrt{x}dx$$

We apply the standard Gram-Schmidt procedure to determine the first six orthogonal polynomials. We start with

$$\boldsymbol{\phi_0(x) = 1}$$

Then, we have

$$\phi_1(x) = x + c \; such \; that \; \langle \phi_1, \phi_0 \rangle_w = 0$$

$$0 = \langle \phi_1, \phi_0 \rangle_w = \int_0^1 1 * (x + c)\sqrt{x}dx = \int_0^1 \left(x^{\frac{3}{2}} + c\sqrt{x}\right)dx = \frac{2}{5} + \frac{2}{3}c \xrightarrow{rearrangement} c = -\frac{3}{5}$$

$$\boldsymbol{\phi_1(x) = x - \frac{3}{5}}$$

For the remaining polynomials, we use the Gram-Schmidt recurrence

$$\phi_k(x) = (x - B_k)\phi_{k-1} - c_k\phi_{k-2}$$

Where, by orthogonality,

$$B_k = \frac{\langle x\phi_{k-1}, \phi_{k-1} \rangle_w}{\langle \phi_{k-1}, \phi_{k-1} \rangle_w}; c_k = \frac{\langle x\phi_{k-1}, \phi_{k-2} \rangle_w}{\langle \phi_{k-2}, \phi_{k-2} \rangle_w}$$

This procedure obtains the following polynomials. We present some sample calculations for the third polynomial to build an intuition for proceeding with calculating higher degree polynomials.

$$B_2 = \frac{\langle x\phi_1, \phi_1 \rangle_w}{\langle \phi_1, \phi_1 \rangle_w}$$

$$\langle x\phi_1, \phi_1 \rangle_w = \int_0^1 x\left(x - \frac{3}{5}\right)^2 \sqrt{x}dx = \frac{8}{175}$$

$$\langle \phi_1, \phi_1 \rangle_w = \int_0^1 \left(x - \frac{3}{5}\right)^2 \sqrt{x} dx = \frac{184}{7875}$$

$$c_k = \frac{\langle x\phi_1, \phi_0 \rangle_w}{\langle \phi_0, \phi_0 \rangle_w}$$

$$\langle x\phi_1, \phi_0 \rangle_w = \int_0^1 x\left(x - \frac{3}{5}\right)\sqrt{x} dx = \frac{8}{175}$$

$$\langle \phi_0, \phi_0 \rangle_w = \int_0^1 \sqrt{x} dx = \frac{2}{3}$$

$$B_2 = \frac{\frac{184}{7875}}{\frac{8}{175}} = \frac{23}{45}; c_2 = \frac{\frac{8}{175}}{\frac{2}{3}} = \frac{12}{75} \xrightarrow{simplification} \boldsymbol{\phi_2 = x^2 - \frac{10}{9}x + \frac{5}{21}}$$

Similarly, we have

$$B_3 = \frac{\langle x\phi_2, \phi_2 \rangle_w}{\langle \phi_2, \phi_2 \rangle_w} = \frac{\frac{7552}{5108103}}{\frac{128}{43659}} = \frac{59}{117}; c_3 = \frac{\langle x\phi_2, \phi_1 \rangle_w}{\langle \phi_1, \phi_1 \rangle_w} = \frac{\frac{128}{43569}}{\frac{8}{175}} = \frac{400}{6237}$$

$$\boldsymbol{\phi_3 = \left(x - \frac{59}{117}\right)\phi_2(x) - \frac{400}{6237}\phi_1(x) = x^3 - \frac{21}{13}x^2 + \frac{105}{143}x - \frac{35}{429}}$$

$$B_4 = \frac{\langle x\phi_3, \phi_3 \rangle_w}{\langle \phi_3, \phi_3 \rangle_w} = \frac{\frac{18944}{203365305}}{\frac{512}{2760615}} = \frac{111}{221}; c_4 = \frac{\langle x\phi_3, \phi_2 \rangle_w}{\langle \phi_2, \phi_2 \rangle_w} = \frac{\frac{512}{2760615}}{\frac{128}{43659}} = \frac{588}{9295}$$

$$\boldsymbol{\phi_4 = \left(x - \frac{111}{221}\right)\phi_3(x) - \frac{588}{9295}\phi_2(x) = x^4 - \frac{36}{17}x^3 + \frac{126}{85}x^2 - \frac{84}{221}x + \frac{63}{2431}}$$

$$B_5 = \frac{\langle x\phi_4, \phi_4 \rangle_w}{\langle \phi_4, \phi_4 \rangle_w} = \frac{\frac{5865472}{1002147721575}}{\frac{32768}{2807136475}} = \frac{179}{357}; c_5 = \frac{\langle x\phi_4, \phi_3 \rangle_w}{\langle \phi_3, \phi_3 \rangle_w} = \frac{\frac{32768}{2807136475}}{\frac{512}{2760615}} = \frac{1728}{27455}$$

$$\boldsymbol{\phi_5 = \left(x - \frac{179}{357}\right)\phi_4(x) - \frac{1728}{27455}\phi_3(x) = x^5 - \frac{55}{21}x^4 + \frac{330}{133}x^3 - \frac{330}{323}x^2 + \frac{55}{323}x - \frac{33}{4199}}$$

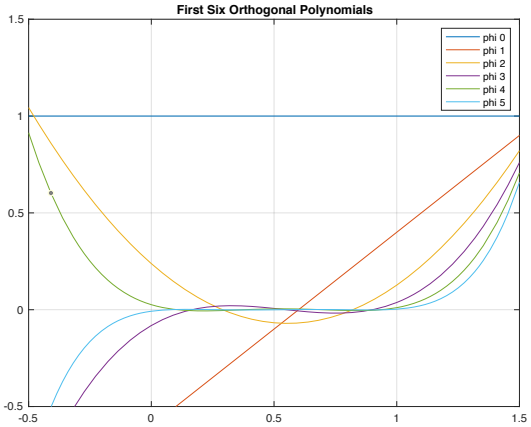


Figure 5: *Plot of first 6 orthogonal polynomials. Observe how on [0,1], the polynomials approach y = 0.*

We can immediately observe some interesting properties of these polynomials. First, they are all monic. One can easily prove that every orthogonal polynomial in this set will be monic by induction. Second, on the k^{th} iteration of deriving these polynomials, we have that both the numerator and denominator of c_k match previously calculated terms. By definition of Gram-Schmidt, we have that the denominator of c_k is the same as the denominator of B_{k-1}. However, we also have that the numerator of c_k is the same as the denominator of B_k. This pattern is, in fact, not a coincidence but a direct result of the monic nature of the polynomials.

Proposition: *For a set of monic orthogonal polynomials* $\{\phi_k\}_{k=0}^{\infty}$, *we have*

$$\langle \phi_k, \phi_k \rangle_w = \langle x\phi_k, \phi_{k-1} \rangle_w$$

Proof: We first not that, by the definition of inner product, that

$$\langle x\phi_k, \phi_{k-1} \rangle_w = \langle \phi_k, x\phi_{k-1} \rangle_w$$

Thus, we now have $x\phi_{k-1}$, *which is a polynomial with degree k. Further, we have* $x\phi_{k-1}$ *is monic because, by assumption,* ϕ_{k-1} *is monic.*

Since orthogonal polynomials form a basis of the inner-product space, we can rewrite any polynomial of degree k as a linear combination of the first k orthogonal polynomials. Hence,

$$x\phi_{k-1} = a_k\phi_k + a_{k-1}\phi_{k-1} + \cdots + a_0\phi_0$$

Note that the only term on the RHS with a degree k term is the leading term in ϕ_k, *which by assumption, has coefficient 1. Hence, equation coefficients for the degree k terms on the LHS and RHS,*

$$x^k = a_k x^k \xrightarrow{simplification} a_k = 1$$

This result yields

$$x\phi_{k-1} = \phi_k + a_{k-1}\phi_{k-1} + \cdots + a_0\phi_0$$

Taking inner product of $x\phi_{k-1}$ *with* ϕ_k, *we obtain*

$$\langle \phi_k, x\phi_{k-1} \rangle_w = \langle \phi_k, \phi_k + a_{k-1}\phi_{k-1} + \cdots + a_0\phi_0 \rangle_w = \langle \phi_k, \phi_k \rangle_w + a_{k-1}\langle \phi_k, \phi_{k-1} \rangle_w + \cdots + a_0\langle \phi_k, \phi_0 \rangle_w$$

But, by orthogonality, we have that $\langle \phi_k, \phi_j \rangle_w = 0$ *for* $j < k$. *Hence, we obtain,*

$$\langle \phi_k, \phi_k \rangle_w = \langle x\phi_k, \phi_{k-1} \rangle_w$$

as desired ■

The previous result suggests that in calculating the k^{th} orthogonal polynomial, we can easily obtain the coefficient c_k from previous calculations. Thus, in designing a computer program to find more of these orthogonal polynomials, we can store the previous calculations in an array and make our program twice as fast.

From previous work in deriving orthogonal polynomials, we know that the k^{th} polynomial has exactly k roots on the interval of integration, in this case, [0,1]. These roots have important properties in defining quadrature points. More explicitly, in Gaussian quadrature, using the roots of the k^{th} orthogonal polynomial results in a quadrature formula of degree $2k-1$, meaning the formula can exactly evaluate the integral of any polynomial with up to degree $2k-1$ on the interval [0,1] with weight function $\sqrt{x}$. This result has very powerful application in integral approximation and uncertainty quantification.

We find the roots of these polynomials using Newton's method. Below is the MATLAB code for newton's method and corresponding table of roots. x_i refers to the i^{th} root of the polynomial.

Polynomial	x_1	x_2	x_3	x_4	x_5
ϕ_0					
ϕ_1	.6				
ϕ_2	.2899	.8212			
ϕ_3	.1647	.5499	.9008		
ϕ_4	.1051	.3762	.6989	.9373	
ϕ_5	.0727	.2695	.5331	.7869	.9569

Table 1: *Roots and quadrature points of first 6 orthogonal polynomials for the given inner product.*

```
function [x,niter] = Newton_method(f,df,x0,Tol, MaxIter)
niter=0;
while 1
    x = x0 - f(x0)/df(x0);
    niter=niter+1;
    if niter>MaxIter
        disp('Maximum number of iterations reached')
        break
    end
    if abs(f(x))<Tol
        break
    end
    x0=x;
end
```

To find the weights for each quadrature point, we can apply Lagrange polynomial interpolation. That is, we can construct Lagrange polynomials using the roots of the orthogonal polynomial and then integrate them with respect to the weight function on [0,1]. Once the weights are calculated, we can easily use Gaussian quadrature to approximate given integral quantities.

Below are the weights in each quadrature formula and the corresponding MATLAB code to calculate them. Here, we use symbolic computations to determine the weights w_i.

Polynomial	w_1	w_2	w_3	w_4	w_5
ϕ_0					
ϕ_1	.6667				
ϕ_2	.2775	.3891			
ϕ_3	.1258	.3076	.2333		
ϕ_4	.0657	.1961	.2523	.1524	
ϕ_5	.0382	.1256	.1987	.1976	.1057

Table 2: *Weights of corresponding quadrature points in Gaussian quadrature for given inner product.*

```
% Lagrange polynomial
syms x;
roots_phi_1 = 3/5;
roots_phi_2 = [.2899 .8212];
roots_phi_3 = [.1647,.5499,.9008];
roots_phi_4 = [.1051 .3762 .6989 .9373];
roots_phi_5 = [.0727 .2695 .5331 .7869 .9569];
roots = {roots_phi_1, roots_phi_2, roots_phi_3, roots_phi_4, roots_phi_5};

for i=1:length(roots)
    root = roots{i};
    weight = zeros(1, i); % weights for phi_i
    for j=1:length(root)
        lagrange = 1;
        for k=1:length(root)
            if (k ~= j)
                lagrange = lagrange * (x-root(k))/(root(j)-root(k));
            end
        end
        w = int(lagrange*sqrt(x),x,0,1);
        weight(j) = w;
    end
    disp(weight)
end
```

4 Tri-Diagonal Matrix Eigenvalue Problems

We now consider eigenvalue problems for tri-diagonal matrices. Tri-diagonal matrices are very special mathematical structures, due to their sparseness, and are important objects in several mathematical contexts. Tri-diagonal matrices are common in polynomial spline interpolation and in solving differential equations using finite difference methods (centered difference formulas). In addition, tri-diagonal matrices can be used to model linear chains of particle interactions. There is a significant body of research in understanding tri-diagonal matrices and how to exploit their structure to yield faster computations.

In this section, our goal is to develop more efficient algorithms to find the eigenvalues of tri-diagonal matrices. We consider three different algorithm improvements. First, we implement a faster matrix multiplication for tri-diagonal matrices and then insert this subroutine into the power method. Then, we implement the Thomas method subroutine for the inverse power method. The Thomas method is far more efficient than traditional linear solvers, requiring only $O(n)$ operations, as opposed to the $O(n^3)$ needed for standard linear solving algorithms. Lastly, we examine Cuppen's divide and conquer algorithm, which can solve for all eigenvalues and eigenvectors of a tri-diagonal symmetric matrix. We derive the basic steps that form Cuppen's method and implement the algorithm in MATLAB.

Improvement 1

For the first improvement, we recode the matrix-vector multiplication so that we are only performing multiplications that always result in nonzero products. In particular, on each iteration, we only perform the three multiplications which occur with the matrix's tri-diagonal entries. This modification helps reduce the time complexity of the matrix multiplication step in power method from $O(n^2)$ to $O(n)$. Furthermore, instead of storing the entire matrix, we create three arrays which store its three diagonals. This change in data structures eliminates sparse entries from memory. As a result, the program has linear space complexity, rather than the typical quadratic space complexity of matrix storage.

Below is the MATLAB code for our implementation of matrix vector multiplication along with code for the power method.

```
function y = tri_diag_mult_array(c,a,b,x)
%     c = diag(A,1);
%     a = diag(A,-1);
%     b = diag(A);
    n = length(x);
    y = zeros(n,1);
    y(1) = b(1)*x(1) + c(1)*x(2);
    for i=2:(n-1)
        y(i) = a(i-1)*x(i-1) + b(i)*x(i) + c(i)*x(i+1);
    end
    y(n) = a(n-1)*x(n-1) + b(n)*x(n);

function [mu, x] = tri_diag_power(c,a,b, x, tol, maxIter)
    iter = 1;
    [~, p] = max(x);
    x = x/x(p);
    while iter < maxIter
        y = tri_diag_mult_array(c,a,b,x);
        mu = y(p);
        [~, p] = max(x);
        if y(p) == 0
            error("eigenvalue was zero. Restart with new x")
        end
        err = norm(x - y/y(p));
        x = y/y(p);
        if err < tol
            disp(mu)
            disp(iter)
            break
        end
        iter = iter + 1;
    end
```

To measure the speed our new implementation, we run several numerical experiments comparing the speed of the original power method implementation to the new implementation. More precisely, we generate random matrices of varying sizes and calculate the convergence time for both implementations to find the matrix's largest eigenvalue. We use the built-in "tic toc" functionality in MATLAB to measure the speed of execution. Below are the results and code.

Matrix Size	Original Power Method Time	New Power Method Time
40	.000165	.000186
80	.000243	.000181
160	.000501	.000284
320	.001808	.000848
640	.001766	.000403
1280	.029229	.002765
2560	.314050	.006445

Table 3: *Result of numerical experiments using improved matrix multiplication subroutine.*

```
%numerical experiments
tol = 1e-5;
maxIter = 1000;

for k=1:8
    A = triDiag(10*2^(k));
    x0 = zeros(10*2^(k),1);
    x0(1)=1;
    disp("time 1")
    tic
    [mu1, x1] = power_method(A, x0, tol, maxIter);
    toc

    c = diag(A,1);
    a = diag(A,-1);
    b = diag(A);

    disp("time 2")
    tic
    [mu2, x2] = tri_diag_power(c,a,b,x0,tol,maxIter);
    toc
end
```

We can see that for small matrices, the difference in execution times between the two implementations is of trivial significance. However, as the matrix size increases, the new implementation quickly dominates the old domination in time. In particular, for square matrices with dimension $n = 2560$, the new implementation is over 48 times faster. For matrices with sizes in the thousands and millions, the new implementation will be crucial in obtaining quick solutions for the largest eigenvalue.

Improvement 2

For the next improvement, we consider the inverse power method, designed to find the smallest eigenvalue of some matrix. Typically, solving a matrix has $O(n^3)$ time complexity using Gauss-Jordan elimination. Later developments have improved the runtime of matrix solvers, but even the best-known solution procedures, optimized CW-like algorithms, have $O(n^{2.373})$ time complexity. Tri-diagonal matrices, on the other hand, can be solved in linear time using the Thomas algorithm. The Thomas algorithm exploits the fact that on the k^{th} step of row reduction, there is only one operation required to zero out the all entries in the k^{th} column. Using this observation, one can derive a recurrence relation to quickly compute each entry used in back substitution, resulting in far more efficient computation than traditional matrix solving tools.

In general, the Thomas algorithm is not stable. If the matrix is diagonally dominant or SPD, the Thomas method will achieve a convergent solution; otherwise, the algorithm may diverge. An alternative to the Thomas algorithm is the Cholesky decomposition, which factorizes any symmetric tri-diagonal matrix into some lower triangular matrix, L, and its conjugate transpose, L^*. The Cholesky decomposition is commonly used in distributed computing as its computations can be parallelized. However, the method only works for SPD matrices and the non-parallelized version has time complexity $O(\frac{n^3}{3})$, so we omit its implementation here.

Below is the code for the Thomas algorithm and the inverse power method. Note that here, we consider the symmetric power method for faster run times.

```
function x = thomas(A,b)
    n = length(b);
    x = zeros(n,1);
    c_ = zeros(n-1,1);
    d_ = zeros(n,1);

    % modifying coefficient matrix values
    c_(1) = A(1,2)/A(1,1);
    d_(1) = b(1)/A(1,1);

    for i=2:(n-1)
        c_(i) = A(i,i+1)/(A(i,i) - A(i,i-1)*c_(i-1));
        d_(i) = (b(i) - A(i,i-1)*d_(i-1))/(A(i,i) - A(i,i-1)*c_(i-1));
    end

    d_(n) = (b(n) - A(n,n-1)*d_(n-1))/(A(n,n) - A(n,n-1)*c_(n-1));

    %back substitution
    x(n) = d_(n);
    for i=(n-1):-1:1
        x(i) = d_(i) - c_(i)*x(i+1);
    end

function [mu, x] = tri_diag_inv_pow(A, x, tol, maxIter)
    iter = 1;
    [~, p] = max(x);
    x = x/x(p);
    while iter < maxIter
        % y = A*x; old
        y = thomas(A,x);
        mu = y(p);
        [~, p] = max(x);
        if y(p) == 0
            error("eigenvalue was zero. Restart with new x")
        end
        err = norm(x - y/y(p));
        x = y/y(p);
        if err < tol
            % disp(iter)
            mu = 1/mu;
            break
        end
        iter = iter + 1;
        if iter == maxIter
            mu = 1/mu;
            break
        end
    end
```

We again conduct several numerical experiments to test the effectiveness of our new implementation. Here, we use the same MATLAB script as in the previous subsection, but we replace the power method implementations with the inverse power method implementations. Below are the results of our experiments.

Matrix Size	Original Inverse Power Method Time	New Inverse Power Method Time
40	.000582	.000085
80	.000987	.000063
160	.008292	.000265
320	.027342	.000427
640	.044377	.000431
1280	.189745	.000502
2560	1.8666872	.001493

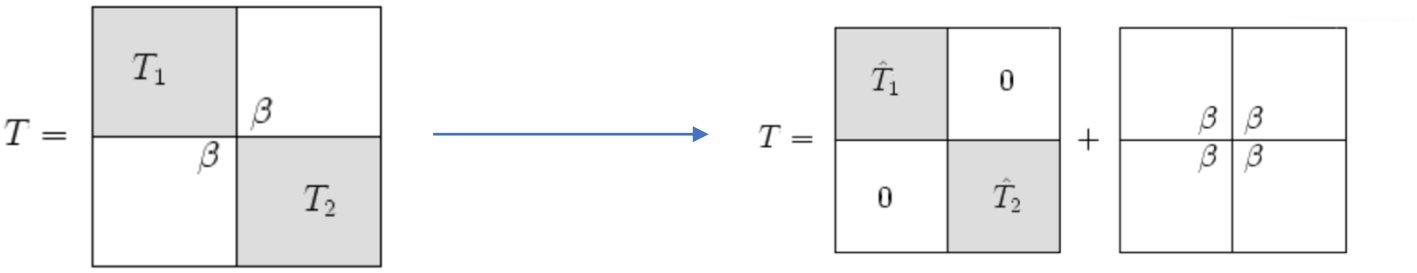
Table 4: *Results of numerical experiments after implementing Thomas algorithm for simplification.*

Unlike in the previous subsection, the improvement of using the new implementation with Thomas algorithm is readily apparent here. For each tested matrix size, using Thomas' method is overwhelmingly faster than the original Matrix solver. The last row of the above table especially highlights this stark difference in run time. For the tested matrix with 2560 rows, the Thomas algorithm results in the inverse power method converging 1,250 times faster than the old implementation. This improvement has major ramifications for higher dimension systems.

Improvement 3

We now examine Cuppen's algorithm for eigenvalue problems of real tri-diagonal symmetric matrices. Unlike the methods discussed in the previous two subsections, Cuppen's algorithm can compute all the eigenvalues and eigenvectors of a tri-diagonal symmetric matrix. The computing power of this algorithm is far greater than the previous two methods and offers several advantages than more standard deflation methods.

Cuppen's method belongs to a family of algorithms called divide-and-conquer algorithms. The idea of these algorithms is to recursively divide a given problem into smaller, more manageable problems and then combine the solutions to these problems to form the general solution. Here, Cuppen's method exploits the fact that any symmetric tri-diagonal matrix, T, is essentially block diagonal and can be decomposed into block matrices with a rank-one update. Moreover, each of these blocks can be diagonalized, and after some straightforward calculations, it can be shown that T is similar to a diagonal matrix plus a rank-one update. We can observe this derivation below (information obtained from LAPACK documentation, images obtained from Wikipedia page).



Here, $\widehat{T_1}$ and $\widehat{T_2}$ are identical to T_1 and T_2, except we subtract β from the bottom diagonal element of T_1 and the top diagonal element of T_2. Now, let $\hat{T}_i = Q_i D_i Q_i^T$ be the diagonalization of $\widehat{T}_\iota, i = 1, 2$. Then, by matrix multiplication, we obtain that

$$T = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \left(\begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix} + \beta z z^T \right) \begin{bmatrix} Q_1^T & \\ & Q_2^T \end{bmatrix}$$

Where z^T is a row vector containing the last row of Q_1 and the first row of Q_2. Clearly, T is similar to the middle matrix given in the above expression, since the Q_i matrices are orthogonal, which implies that Q^T is the inverse of Q (here, Q is the left matrix on the RHS, and Q^T is the right matrix on the RHS). Note that each $\widehat{T}_i$ is real and symmetric, so D_i is a diagonal matrix. Hence, we have that T is similar to a diagonal plus rank-one update matrix, as claimed. In particular, this implies that these two matrices have the same eigenvalues.

To obtain the eigenvalues of the diagonal plus rank-one update matrix, we can calculate the roots of the following secular equation

$$f(\lambda) = 1 + \beta \sum_{i=1}^{n} \frac{z_i^2}{d_i - \lambda}$$

We do not prove the validity of the secular equation, although the proof is not hard to follow. We also remark that there are special cases to consider when calculating the eigenvalues. More explicitly, the secular equation assumes that all the d_i values are distinct and that all the z_i values are nonzero. However, in either case, we can still find an eigenpair and reduce the problem to find the remaining eigenpairs using the secular equation.

Overall, because the secular equation is a rational function, we can find each root in m flops, implying that the process of finding each eigenvalue is $O(m^2)$. Thus, applying master's theorem to analyze recursive time complexity, we have that

$$T(m) = 2 * T\left(\frac{m}{2}\right) + O(m^2) \xrightarrow{master's\ theorem} T(n) = O(n^2)$$

Indeed, we have that the time complexity of Cuppen's algorithm is quadratic, which is respectable compared to other eigenvalue algorithms.

We now implement Cuppen's divide-and-conquer algorithm in MATLAB. We remark that, for stability purposes, we use standard QR factorization to find the eigenvalues and eigenvectors of each diagonal plus rank-one matrix. Implementing a roots finder for the secular equation is certainly not difficult, but it may result in unstable solutions for this eigenproblem. In any case, we are more interested in the divide-and-conquer aspect of Cuppen's algorithm than the method to find the roots.

```
function [lambda,Q] = cuppen(T)
    %base case
    if all(size(T)== [1,1])
        lambda = T;
        Q = 1;
        return
    else
        m = floor(length(T)/2);
        n = length(T);
        beta = T(m+1,m);

        T1 = T(1:m,1:m);
```

```
    T1(m,m) = T1(m,m) - abs(beta);

    T2 = T(m+1:n,m+1:n);
    T2(1,1) = T2(1,1) - abs(beta);

    %recursive step of algorithm
    [lambda1, Q1] = cuppen(T1);
    [lambda2, Q2] = cuppen(T2);

    D = diag([diag(lambda1);diag(lambda2)]);
    z = transpose([Q1(length(Q1),:), Q2(1,:)]);

    % diagonal plus rank-one update eigenvalues
    [Q_, lambda] = eig(D + beta*z*transpose(z));
    % [Q_, lambda] = secular(D,z); using secular method

    q_12 = zeros(n);
    q_12(1:m,1:m) = Q1;
    q_12(m+1:n,m+1:n) = Q2;

    Q = q_12*Q_; % eigenvector correction to produce orthogonal eigenvectors
    return
end
```

5 Analysis of Generalized Euler's Method

For our final problem, we consider generalized Euler's method to solve nonhomogeneous differential equations of the form:

$$\overrightarrow{y'}(t) = A\vec{y}(t) + f(\vec{y}(t))$$

Although the nonhomogeneous solutions have an analytic derivation, in practice, they involve complex integrals and can be difficult to evaluate. To avoid such computation, we can implement generalized Euler's method to solve a discretized version of the original problem. This discretization approach is very common for highly complex functions *f* and can be useful in solving the given class of differential equations.

To assess the quality of generalized Euler's method in producing reliable solutions, we first evaluate the method's order of accuracy. Then, we consider the stability of this method on a given test problem, using the matrix exponential to better characterize stability. Both these analyses follow standard approaches used in numerical analysis.

Order of Accuracy

Although generalized Euler's method works for systems of differential equations, for simplicity, we solve this method using component wise calculation. This approach will help simplify notation and ease of explanation.

We define

$$e_i \coloneqq y(t_{i+1}) - \left(e^{Ah}y(t_i) + he^{hA}f\big(y(t_i)\big)\right)$$

to be the local error of generalized Euler's method with h has the step size. Here, we are taking the difference between the exact solution and the numerical solution. Taylor expanding $y(t_{i+1})$ around t_i with $t_{i+1} = t_i + h$, we obtain

$$y(t_{i+1}) = y(t_i + h) = y(t_i) + y'(t_i)h + \frac{y''(t_i)h^2}{2} + O(h^3)$$

Since $y'(t_i) = Ay(t_i) + f\big(y(t_i)\big)$, we have that

$$y''(t_i) = Ay'(t_i) + f_y\big(y(t_i)\big)\, y'(t_i) = \Big(A + f_y\big(y(t_i)\big)\Big)\Big(Ay(t_i) + f\big(y(t_i)\big)\Big)$$

Hence,

$$y(t_{i+1}) = y(t_i + h) = y(t_i) + \Big(Ay(t_i) + f\big(y(t_i)\big)\Big)\, h + \Big(A + f_y\big(y(t_i)\big)\Big)\Big(Ay(t_i) + f\big(y(t_i)\big)\Big)\frac{h^2}{2} + O(h^3)$$

Plugging this into local error, we obtain

$$e_i = (t_i + h) = y(t_i) + \Big(Ay(t_i) + f\big(y(t_i)\big)\Big)\, h + \Big(A + f_y\big(y(t_i)\big)\Big)\Big(Ay(t_i) + f\big(y(t_i)\big)\Big)\frac{h^2}{2} + O(h^3) - \Big(e^{Ah}y(t_i) + he^{Ah}f\big(y(t_i)\big)\Big)$$

$$e_i = y(t_i)[1 - e^{hA}] + \Big(Ay(t_i) + [1 - e^{hA}]f\big(y(t_i)\big)\Big)\, h + \Big(A + f_y\big(y(t_i)\big)\Big)\Big(Ay(t_i) + f\big(y(t_i)\big)\Big)\frac{h^2}{2} + O(h^3)$$

We know that Taylor expansion of e^x is $1 + x + \frac{x^2}{2} + \frac{x^3}{3} + \cdots$, and thus,

$$e_i = y(t_i)\left[1 - \left(1 + Ah + \frac{Ah^2}{2} + O(h^3)\right)\right] + \left(Ay(t_i) + \left[\left(1 + Ah + \frac{Ah^2}{2} + O(h^3)\right)\right] f\big(y(t_i)\big)\right) h + \Big(A + f_y\big(y(t_i)\big)\Big)\Big(Ay(t_i) + f\big(y(t_i)\big)\Big)\frac{h^2}{2} + O(h^3)$$

$$e_i = y(t_i)\left[-Ah - \frac{Ah^2}{2} + O(h^3)\right] + \left(Ay(t_i) + \left[-Ah - \frac{Ah^2}{2} + O(h^3)\right] f\big(y(t_i)\big)\right) h + \Big(A + f_y\big(y(t_i)\big)\Big)\Big(Ay(t_i) + f\big(y(t_i)\big)\Big)\frac{h^2}{2} + O(h^3)$$

Simplifying and expanding, we obtain

$$e_i = -y(t_i)Ah - y(t_i)\frac{Ah^2}{2} + Ahy(t_i) + -Ah^2 f\big(y(t_i)\big) - \frac{Ah^3}{2} f\big(y(t_i)\big) + O(h^2)$$

$$\boldsymbol{e_i = -y(t_i)\frac{Ah^2}{2} - Ah^2 f\big(y(t_i)\big) - \frac{Ah^3}{2} f\big(y(t_i)\big) + O(h^2) = O(h^2)}$$

Hence, generalized Euler's has quadratic local error. Consequently, applying standard arguments to determine the global error, we have generalized Euler's method is $O(h)$, or first order. This result matches the first order nature of original Euler's method for solving ODEs.

Stability

We now evaluate the stability of Euler's method using the following test problem:

$$\vec{x}'(t) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}\vec{x}(t) + \begin{bmatrix} 0 \\ \sin\big(x_1(t)\big) \end{bmatrix};\ \vec{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}$$

To perform the stability analysis, we first linearize the sine term in the above equation. Then, we evaluate the matrix exponential for the above *2 x 2* matrix.

Based on previous results in matrix exponentiation, we know that

$$e^{Ah} = Se^{\Lambda h}S^{-1} \quad (1)$$

Where $A = S\Lambda S^{-1}$ is the diagonalization of matrix A. Hence, to find the matrix exponential, we diagonalize A and then evaluate the RHS of the above equation. For the eigenvalues of A, we obtain

$$\det(A - \lambda I) = 0 \xrightarrow{substitution} \begin{vmatrix} -\lambda & 1 \\ -1 & -\lambda \end{vmatrix} = 0 \xrightarrow{simplification} \lambda^2 + 1 = 0 \xrightarrow{simplification} \lambda = \pm i$$

Finding the eigenvectors of eigenvalue $\lambda_1 = i$,

$$(A - \lambda_1 I)\vec{v} = \begin{bmatrix} -i & 1 \\ -1 & -i \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \xrightarrow{letting\ v_1=1} v_2 = i;\ \overrightarrow{v_1} = \begin{bmatrix} 1 \\ i \end{bmatrix}$$

Similarly, for $\lambda_2 = i$,

$$(A - \lambda_1 I)\vec{v} = \begin{bmatrix} i & 1 \\ -1 & i \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \xrightarrow{letting\ v_1=1} v_2 = -i;\ \overrightarrow{v_2} = \begin{bmatrix} 1 \\ -i \end{bmatrix}$$

Note that these eigenvectors are orthogonal. We now normalize the above eigenvectors so that we achieve an orthogonal matrix in our diagonalization. Consequently,

$$\frac{\overrightarrow{v_1}}{\|\overrightarrow{v_1}\|} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{bmatrix};\frac{\overrightarrow{v_2}}{\|\overrightarrow{v_2}\|} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-i}{\sqrt{2}} \end{bmatrix};$$

We can now easily verify that the matrix of eigenvectors S, is orthogonal, which implies the inverse of S is its complex conjugate transpose. Hence, we have that

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = A = S\Lambda S^{-1} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} & \frac{-i}{\sqrt{2}} \end{bmatrix}\begin{bmatrix} i & 0 \\ 0 & -i \end{bmatrix}\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-i}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{i}{\sqrt{2}} \end{bmatrix}$$

Hence, plugging into equation (1), we have

$$e^{Ah} = Se^{\Lambda h}S^{-1} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} & \frac{-i}{\sqrt{2}} \end{bmatrix}\begin{bmatrix} e^{ih} & 0 \\ 0 & e^{-ih} \end{bmatrix}\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-i}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{i}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{e^{ih}}{2} + \frac{e^{-ih}}{2} & \frac{e^{ih}}{2i} - \frac{e^{-ih}}{2i} \\ \frac{-e^{ih}}{2i} + \frac{e^{-ih}}{2i} & \frac{e^{ih}}{2} + \frac{e^{-ih}}{2} \end{bmatrix} = \begin{bmatrix} \cos(h) & \sin(h) \\ -\sin(h) & \cos(h) \end{bmatrix}$$

Plugging the matrix exponential into the generalized Euler equation yields

$$\vec{x}_{i+1} = \begin{bmatrix} \cos(h) & \sin(h) \\ -\sin(h) & \cos(h) \end{bmatrix}\vec{x}_i + h\begin{bmatrix} \cos(h) & \sin(h) \\ -\sin(h) & \cos(h) \end{bmatrix}\begin{bmatrix} 0 \\ x_{i,1} \end{bmatrix} \quad (2)$$

Where x_1 is the linearization of the sine term. To place the above equation into a more appropriate form for analysis, we want to find a matrix B such that we can place Euler's equation into the form

$$\vec{x}_{i+1} = G\vec{x}_i$$

Such that we can determine the stability criterion via the eigenvalues of G. To make this change, we make the following observation:

$$\begin{bmatrix} 0 \\ x_{i,1} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x_{i,1} \\ x_{i,2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix} \vec{x}_i$$

Substituting this into equation (2) yields

$$\vec{x}_{i+1} = \begin{bmatrix} \cos(h) & \sin(h) \\ -\sin(h) & \cos(h) \end{bmatrix} \vec{x}_i + h \begin{bmatrix} \cos(h) & \sin(h) \\ -\sin(h) & \cos(h) \end{bmatrix} \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix} \vec{x}_i = G\vec{x}_i$$

$$G = \begin{bmatrix} \cos(h) & \sin(h) \\ -\sin(h) & \cos(h) \end{bmatrix} + h \begin{bmatrix} \cos(h) & \sin(h) \\ -\sin(h) & \cos(h) \end{bmatrix} \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} \cos(h) & \sin(h) \\ -\sin(h) & \cos(h) \end{bmatrix} + h \begin{bmatrix} -\sin(h) & 0 \\ -\cos(h) & 0 \end{bmatrix}$$

Hence,

$$G = \begin{bmatrix} \cos(h) - h\sin(h) & \sin(h) \\ -\sin(h) - h\cos(h) & \cos(h) \end{bmatrix}$$

We know that the method is stable for choices of h such that the eigenvalues of G have modulus less than or equal to 1. After some simplifications, the characteristic polynomial for G is

$$\lambda^2 + (h\sin(h) - 2\cos(h))\lambda + 1$$

Finding the roots of this equation is equivalent to finding the eigenvalues of G. However, this task can be difficult to complete analytically. Instead, we consider a variety of possible step sizes and plot the behavior of the eigenvalues for each step size. This approach will give us an understanding of suitable choices of h and a threshold for which the method becomes unstable.

Below is the MATLAB code used to test different h values. For accuracy purposes, we do not consider significantly large step sizes, since the errors at each computation could result in inaccurate solutions.

```
%project 4 stability
b = @(h)(h*sin(h)-2*cos(h));
root1 = @(h)((-(b(h))+sqrt((b(h)^2)-4))/2);
root2 = @(h)((-(b(h))-sqrt((b(h)^2)-4))/2);

h = linspace(0,10,10000);
pos_roots = zeros(length(h),1);
neg_roots = zeros(length(h),1);

for step=1:length(h)
    r1 = root1(h(step));
    r2 = root2(h(step));

    pos_roots(step) = norm(r1);
    neg_roots(step) = norm(r2);
end

hold on
% viscircles([0,0],1);
plot(pos_roots)
plot(neg_roots)
```

We now plot the norms of each eigenvalue pair for each h value, shown below.

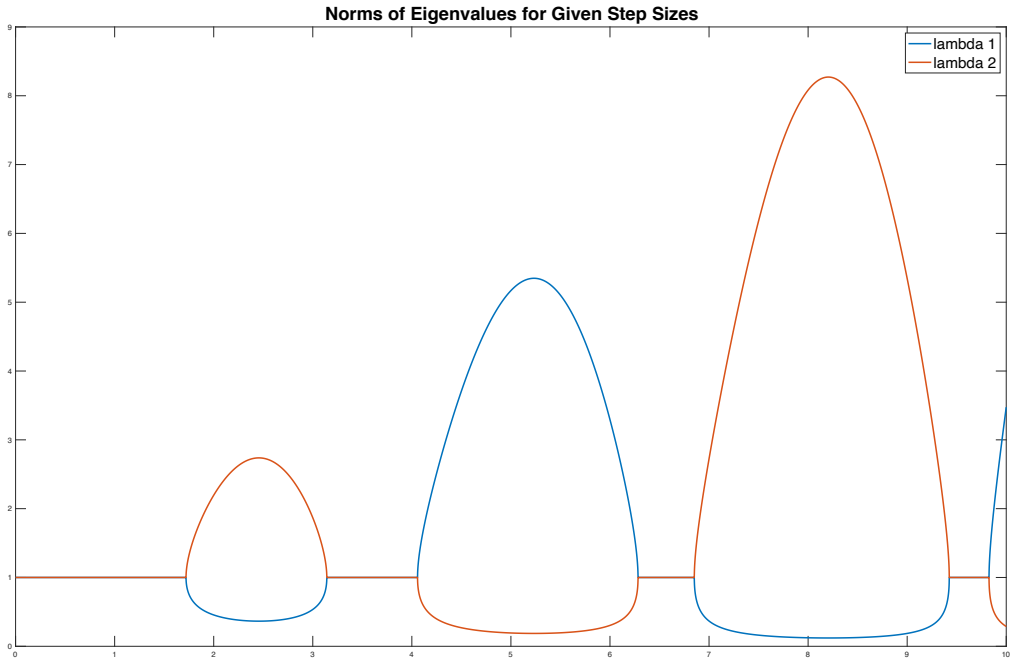


Figure 6: *Plot of norms of eigenvalues of matrix G for different step sizes h.*

Notice that the only time when both roots have norm less than or equal to one is exactly when both roots have unit modulus. This result, in fact, matches our expectation given the characteristic polynomial. In particular, since the constant term in the characteristic polynomial is 1, we have that the product of the roots is equal to 1, and thus,

$$\lambda_1 * \lambda_2 = 1 \xrightarrow{modulus\ operations} |\lambda_1 * \lambda_2| = |1| = 1 \xrightarrow{modulus\ operation} |\lambda_1| * |\lambda_2| = 1$$

Hence, we have that the product of the moduli of the two roots is equal to 1. Since we require that each modulus itself is also less than or equal to one, we have that the modulus of both roots must equal exactly 1 in order for the method to be stable. Equivalently, the method is only stable if both eigenvalues have modulus 1.

From the above plot, we can also see interesting behaviors in the interactions of the norms of the two roots. Namely, when the modulus of each root is not equal to one, their plots combined from cone-like shapes that increase in size as the step size increases. We can plot the norms of the two roots in the $\lambda_1 - \lambda_2$ plane to see how the interact with one another more directly.

If we define the square in the lower left corner of the above plot to be the region of stability, we again see that the only pair of eigenvalues in this region is *(1,1).* Otherwise, as the modulus of one eigenvalue decreases, the modulus of the other increases.

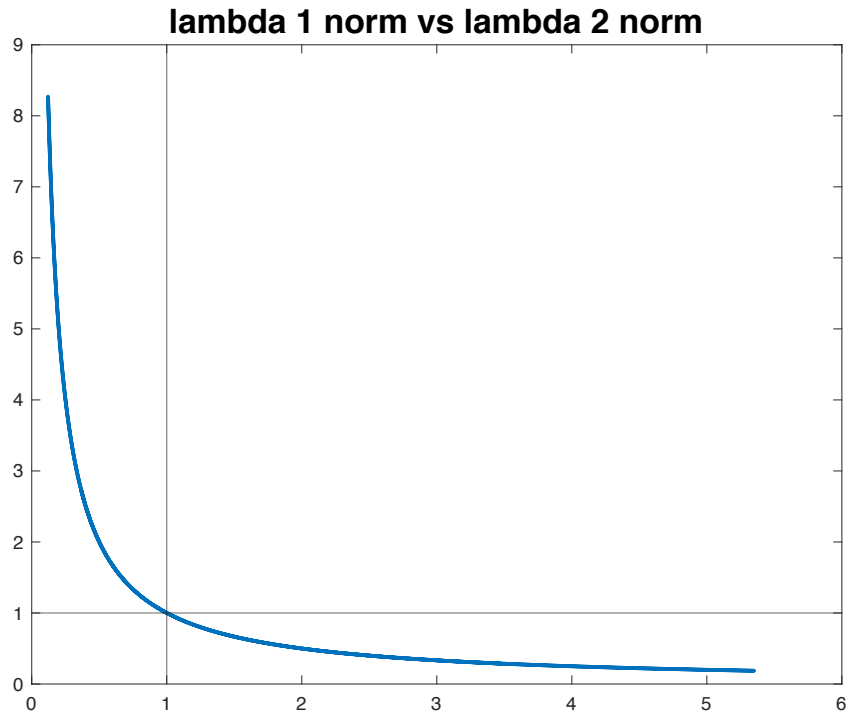


Figure 7: *Plot of norms of eigenvalues on* $\lambda_1 - \lambda_2$ *plane.*

Thus, overall, we characterize generalized Euler's method to be *weakly stable* for this problem, i.e., the error is bounded and constant over time.

Acknowledgements

I would like to thank Professor Xiantao Li for all his contributions to the making of this paper.