University of the Pacific

Numerical Analysis

Math 110

Math 110 Homework Project

Author:
Latimer D. Harris-Ward

Supervisor:
Dr. Aleksei Beltukov

August 28, 2022

CONTENTS

HOMEWORK 1

EXERCISE 1

(1) Derive the remainder of Lagrange interpolation for three nodes by solving an appropriate boundary value problem. Show that the formula is consistent with Taylor theory.

Solution: Before beginning with the following boundary value problem,

$$R^{(3)}(x) = f^{(3)}(x), \quad R(x_1) = R(x_2) = R(x_3) = 0, \tag{1}$$

I wanted to explore first how to interpolate three points through Lagrangian interpolation. However, I did not know where to begin. As such, instead of starting with linear algebra, I started with algebra. I attempted to use the general expression for a parabola, $y - h = a(x - h)$, but I quickly recognized that it wasn't feasible. I then looked up the Lagrange interpolant formula, but I was determined to derive the formula myself. After trying even calculus, that's when I used linear algebra to successfully derive the equation for a general quadratic interpolating three points. We begin with the space of polynomials of degree $n = 2$, $P_2 = \{ax^2 + bx + c \mid a, b, c \in \mathbb{R}\}$. Given three points $(x_1, f(x_1)), (x_2, f(x_2)), (x_3, f(x_3))$, the linear system that we would need to solve is

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix} \begin{bmatrix} c \\ b \\ a \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \end{bmatrix}.$$

As per our discussion in class, it would have been much more constructive to determine a suitable basis with which we can form our interpolating polynomial p, but I used Gaussian Elimination (lots of crying, but not really) in order to obtain the following expression for p,

$$p(x) = f(x_1)\frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} + f(x_2)\frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} + f(x_3)\frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)},$$

which is consistent with the Lagrange interpolation formula for three nodes. Now, we can begin to determine the formula of the remainder, given by $R(x) = f(x) - p(x)$. However, before setting the aforementioned expression in its form in equation (1), we need to first assume that $R, f \in C^3[x_1, x_3]$. Further, by definition of Lagrangian interpolation, the remainder evaluated at the nodes is identically zero. Now, to solve equation (1), we apply the Fundamental Theorem of Calculus three times, making sure that we take care to use appropriate dummy variables, and by doing so, we obtain the following expression,

$$R(x) = \int_{x_1}^{x} \left[\int_{x_1}^{t} \left[\int_{x_1}^{s} f^{(3)}(r)dr \right] ds \right] dt + R''(x_1)(x - x_1)^2 + R'(x_1)(x - x_1) + R(x_1).$$

At this point, because $R(x_1) = 0$, we can omit the last term. Then, to determine $R'(x_1)$ and $R''(x_1)$, we use the two remaining boundary conditions. In doing so, we find that our new expression becomes

$$\begin{aligned} R(x) &= \int_{x_1}^{x} \left[\int_{x_1}^{t} \left[\int_{x_1}^{s} f^{(3)}(r)dr \right] ds \right] dt \\ &- \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} \int_{x_1}^{x_2} \left[\int_{x_1}^{t} \left[\int_{x_1}^{s} f^{(3)}(r)dr \right] ds \right] dt \\ &- \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} \int_{x_1}^{x_3} \left[\int_{x_1}^{t} \left[\int_{x_1}^{s} f^{(3)}(r)dr \right] ds \right] dt. \end{aligned}$$

Now, by changing the order of integration for all three integrals, and then integrating, we obtain the following expression for the remainder:

$$\begin{aligned} R(x) &= \int_{x_1}^{x} \frac{(x-r)^2}{2} f^{(3)}(r)dr \\ &- \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} \int_{x_1}^{x_2} \frac{(x_2-r)^2}{2} f^{(3)}(r)dr \\ &- \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} \int_{x_1}^{x_3} \frac{(x_3-r)^2}{2} f^{(3)}(r)dr. \end{aligned}$$

Now, we write each of these integrals as one expression by introducing the Heaviside step function. The resulting equation is

$$\begin{aligned} R(x) &= \int_{x_1}^{x_3} \left[\begin{cases} \frac{(x-r)^2}{2}, & x_1 < r < x \\ 0 & , x < r < x_2 \\ 0 & , x_2 < r < x_3 \end{cases} \right] f^{(3)}(r)dr \\ &- \int_{x_1}^{x_3} \left[\begin{cases} \frac{(x-x_1)(x-x_3)(x_2-r)^2}{2(x_2-x_1)(x_2-x_3)}, & x_1 < r < x_2 \\ 0 & , x_2 < r < x_3 \end{cases} \right] f^{(3)}(r)dr \\ &- \int_{x_1}^{x_3} \frac{(x-x_1)(x-x_2)(x_3-r)^2}{2(x_3-x_1)(x_3-x_2)} f^{(3)}(r)dr. \end{aligned}$$

There are a few things to note. First and foremost, the x is set to be in the open interval (x_1, x_2). However, the conclusion of the computation does not change if x is between x_2 and x_3. Further, these three integrals can be combined into one integral, namely

$$R(x) = \int_{x_1}^{x_3} K(x,r) f^{(3)}(r)dr,$$

where the kernel, $K(x,r)$ is given by

$$K(x,r) = \begin{cases} \frac{(x-r)^2}{2} - \frac{(x-x_1)(x-x_3)(x_2-r)^2}{2(x_2-x_1)(x_2-x_3)} - \frac{(x-x_1)(x-x_2)(x_3-r)^2}{2(x_3-x_1)(x_3-x_2)}, & x_1 < r < x \\ -\frac{(x-x_1)(x-x_3)(x_2-r)^2}{2(x_2-x_1)(x_2-x_3)} - \frac{(x-x_1)(x-x_2)(x_3-r)^2}{2(x_3-x_1)(x_3-x_2)} & , x < r < x_2 \\ -\frac{(x-x_1)(x-x_2)(x_3-r)^2}{2(x_3-x_1)(x_3-x_2)} & , x_2 < r < x_3 \end{cases}.$$

By invoking the Generalized Mean Value Theorem, and evaluating the remaining integral, we obtain

$$\begin{aligned} R(x) &= \int_{x_1}^{x_3} K(x,r) f^{(3)}(r)dr \\ &= f^{(3)}(\xi) \int_{x_1}^{x_3} K(x,r)dr \\ &= \boxed{\frac{f^{(3)}(\xi)}{6}(x-x_1)(x-x_2)(x-x_3)}, \end{aligned}$$

which is consistent with the formula for the Lagrange remainder for three nodes, in the handout.

Plotting and Analyzing the Kernel

In order to further analyze what I just did, I plotted the Kernel. This was accomplished by first defining a function that took x, r, x_1, x_2, and x_3 as inputs. However, to avoid clutter, the following substitutions were made: $x_1 = a, x_2 = b, x_3 = c$. The code is presented below.

```
function K = quad_kern(x,r,a,b,c)

K = ( 0.5*(x-r).^2 - ((x-a)*(x-c)*(b-r).^2)/(2*(b-a)*(b-c)) - ...
    ((x-a)*(x-b)*(c-r).^2)/(2*(c-a)*(c-b)) ).*(a < r & r < x)...
    + ( - ((x-a)*(x-c)*(b-r).^2)/(2*(b-a)*(b-c)) - ...
        ((x-a)*(x-b)*(c-r).^2)/(2*(c-a)*(c-b)) ).*(x < r & r < b)...
    + ( - ((x-a)*(x-b)*(c-r).^2)/(2*(c-a)*(c-b)) ).*(b < r & r < c);

end
```

After defining the function, I wanted to select some good parameters to produce a decent depiction of the kernel, and I ultimately settled on $a = 2, b = 5, c = 7, x = 3$, with r being the dependent variable. Then, I plotted the kernel and its first and second derivatives. The resulting code and plots are below.

```
%% Exercise 1

```

```
r = linspace(0,10,1000);

% plotting the kernel

a = 2;
b = 5;
c = 7;
x = 3;

% standard kernel plot

figure
plot(r,quad_kern(x,r,a,b,c));
xlabel('$r$','interpreter','latex')
ylabel(sprintf('$K(%1.f,r)$',x),'interpreter','latex');
title('Kernel of Remainder: Quadratic ...
    Interpolation','Interpreter',"latex");
set(gca,'fontsize',20);

% plot of first derivative

r1 = r(:,1:end-1);

figure
plot(r1,diff(quad_kern(x,r,a,b,c)));
xlabel('$r$','interpreter','latex')
ylabel(sprintf('$K''(%1.f,r)$',x),'interpreter','latex');
title('First Derivative of the Kernel','Interpreter','latex');
set(gca,'fontsize',20);

% plot of the second derivative

r2 = r1(:,1:end-1);

figure
plot(r2,diff(diff(quad_kern(x,r,a,b,c))));
xlabel('$r$','interpreter','latex')
ylabel(sprintf('$K(3)'(%1.f,r)$',x),'interpreter','latex');
title('Second Derivative of the Kernel','Interpreter',"latex");
set(gca,'fontsize',20);
```

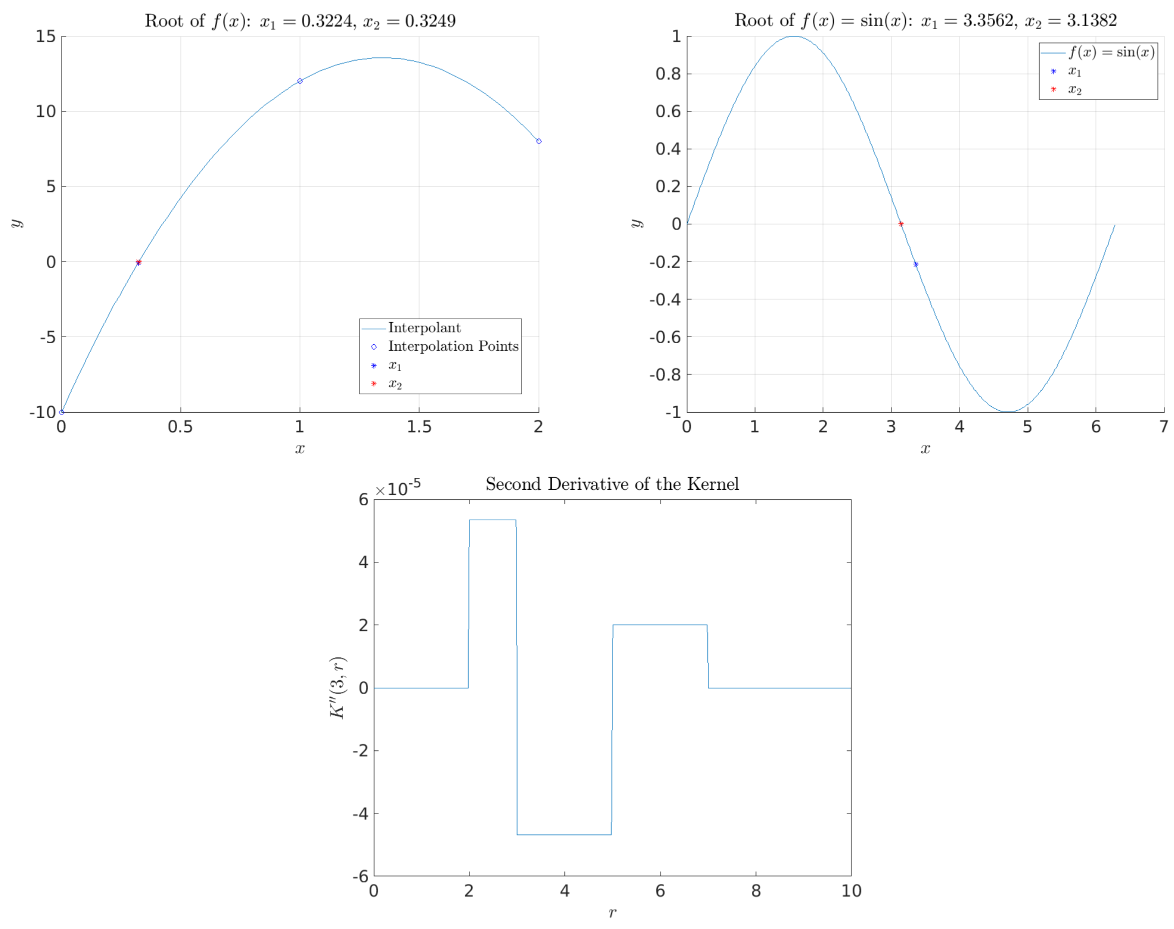


Figure 1: These plots are of the kernel, its first, and its second derivatives.

The plot of the kernel is consistent with what I would expect to happen because it is piecewise quadratic. I would expect that generally, for N nodes, the kernel would be a piecewise polynomial of degree $N - 1$, which is consistent with Lagrangian interpolation. Naturally, the first and second derivatives would be what we expect from such a kernel. Thus, the first derivative would be piecewise linear and the second derivative piecewise constant, which is pretty cool, and it's what we ultimately observe. I lastly wanted to observe what would happen as $x \to b$. The reason I chose this limit was because the other two limits $(x \to a, x \to c)$ are identical for this example. The resulting plots are below. These plots make sense because as $x \to b$, the third component of the kernel effectively vanishes, so there would only be two components to the piecewise kernel. If we let $x_3 \to x_2$, and then the resulting point approach x_1, we obtain

$$R(x) = \frac{f^{(3)}(\xi)}{6}(x - x_1)^3,$$

which is the formula for the Taylor remainder for quadratic Taylor interpolation.

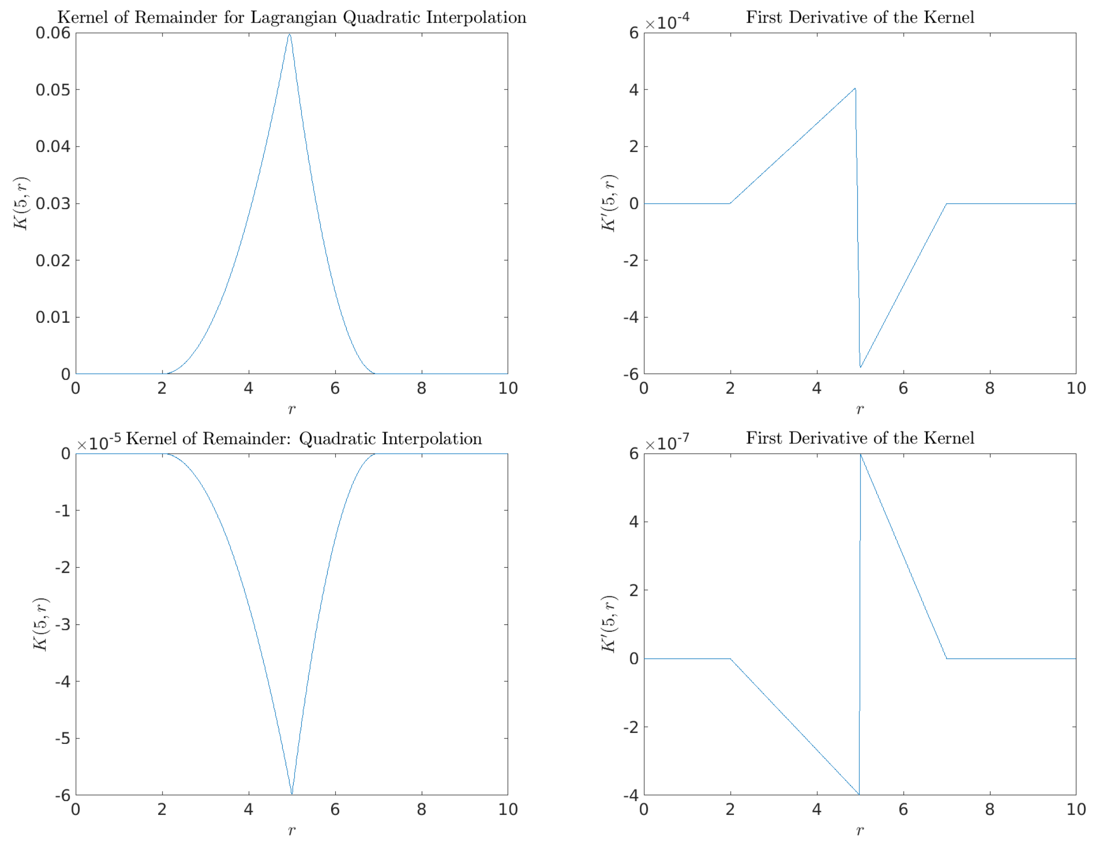


Figure 2: The upper plots show the limit $x \to b^-$ and the lower $x \to b^+$. We can clearly see that the limits from the left and the right don't agree.

Challenges with the Derivation

The derivation at first was not very difficult. Even determining the coefficients of the Remainder was not too difficult, only because I reduced the system to a 2×2 linear system and performed Gaussian Elimination. Difficulties arose when changing the order of integration for the triple integral. However, when in class we performed the change with two integrals at a time, then it was much easier to do. Now, I feel I can confidently interchange bounds for most integrals, not because of rote memorization, but because of understanding how to draw two dimensional regions, using logic in order to come to a sound conclusion. My last difficulty was determining the kernel. This was because I was unsure about whether I should segment the first integral into two piecewise functions or three. So, after long hours and lots of crying, I asked Justin and he said that he and Kelli had three conditions. Afterwards, I had a heading and I continued the derivation on my own, I encountered some user error concerning the integration of such a hefty kernel, and I was able to derive the remainder. However, why are there three inequalities? Is it appropriate to do so, and why did we do it? Well, because we have three distinct points, we have two separate intervals. And, our x could be in either interval, giving us a third interval. Thus, we need to separate our regions of integration based on whether our x was between the first two or the last two points. Therefore, we needed three inequalities with which we defined our kernel, and ultimately the integration bounds. Logically, we can induct that for four points, we would have four inequalities with which we would use to integrate.

EXERCISES 2 AND 3

(2) In MATLAB*, one can find Lagrange polynomials using the* **`polyfit`** *command. Write a script that plots Lagrange polynomials through N equidistant nodes $\{x_k = k/(N+1)\}, k = 1, \ldots, N$ for $f(x) = e^x$. What happens as N increases?*

(3) Repeat the previous exercise with MATLAB'S **`humps`** *function in place of the exponential. What are you observations?*

Solution: This exercise was very straightforward. For the exponential, as the number of nodes, and consequently the degree of the polynomial, increased, the fits became much better until they matched the function exactly. The humps function was approximated poorly by Lagrange interpolating polynomials. The interpolating polynomial for the exponential required fewer nodes and a lower degree than that of the humps function. This is definitely a result of the curvature[1] of the humps function, as compared to the monotonely increasing exponential. I will discuss the humps function more later. However, I also wanted to fix the degree of the polynomial and only vary the number of nodes; conversely, I wanted to fix the number of nodes and vary the degree of the interpolating polynomial. The code is presented at the end.

[1]To be precise, the curvature of a function is defined as such:

$$\kappa = \left|\frac{d\vec{T}}{ds}\right| = \frac{|\vec{T}'(t)|}{|r'(t)|},$$

where $\vec{T}'(t)$ is the unit tangent vector, $r'(t)$ is the tangent vector, and s is the arc length. If one had the desire, the curve can be parameterized, and its curvature can be calculated.

Fixing and Varying both the Degree and the Number of Nodes

(a) $f(x) = e^x$

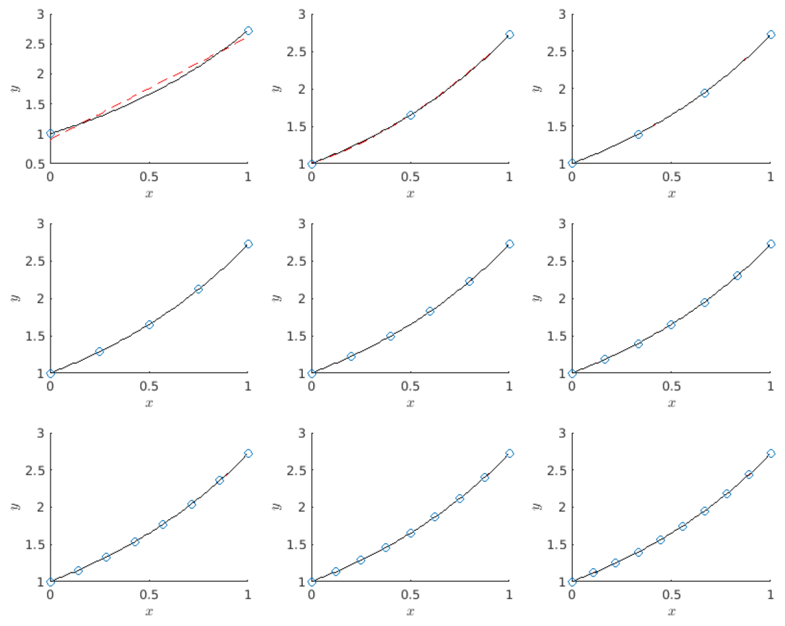


Figure 3: A plot of Lagrange interpolating polynomials approximating the exponential.

For the initial analysis, the degree of the polynomial was fixed to the fifth-degree. The resulting plots are below, with the number of nodes varying from $N = 2, \ldots, 4$. Further, in Figure 5, I fixed the number of nodes at 5 and decreased the degree of the polynomial to 2. Figure 4 conveys that as the number of nodes increases, the approximations of the target function become better, ultimately matching the function exactly. In Figure 5, by fixing the number of nodes at 5 and decreasing the degree to 2, we can clearly see that the interpolant is not in complete agreement with the parent function, which is what I expected. Thus, for a low degree polynomial we would need more nodes for a decent agreement with the parent function. This would logically imply that for a higher degree polynomial, we wouldn't need as many nodes. Of course, this choice ultimately depends on the function being approximated and/or the data being interpolated. We see examples of this for the humps function. The last plot on the second row of Figure 4 is the Lagrange polynomial interpolating the exponential with a degree of 4 because there are 5 nodes, which approximates the function very well.

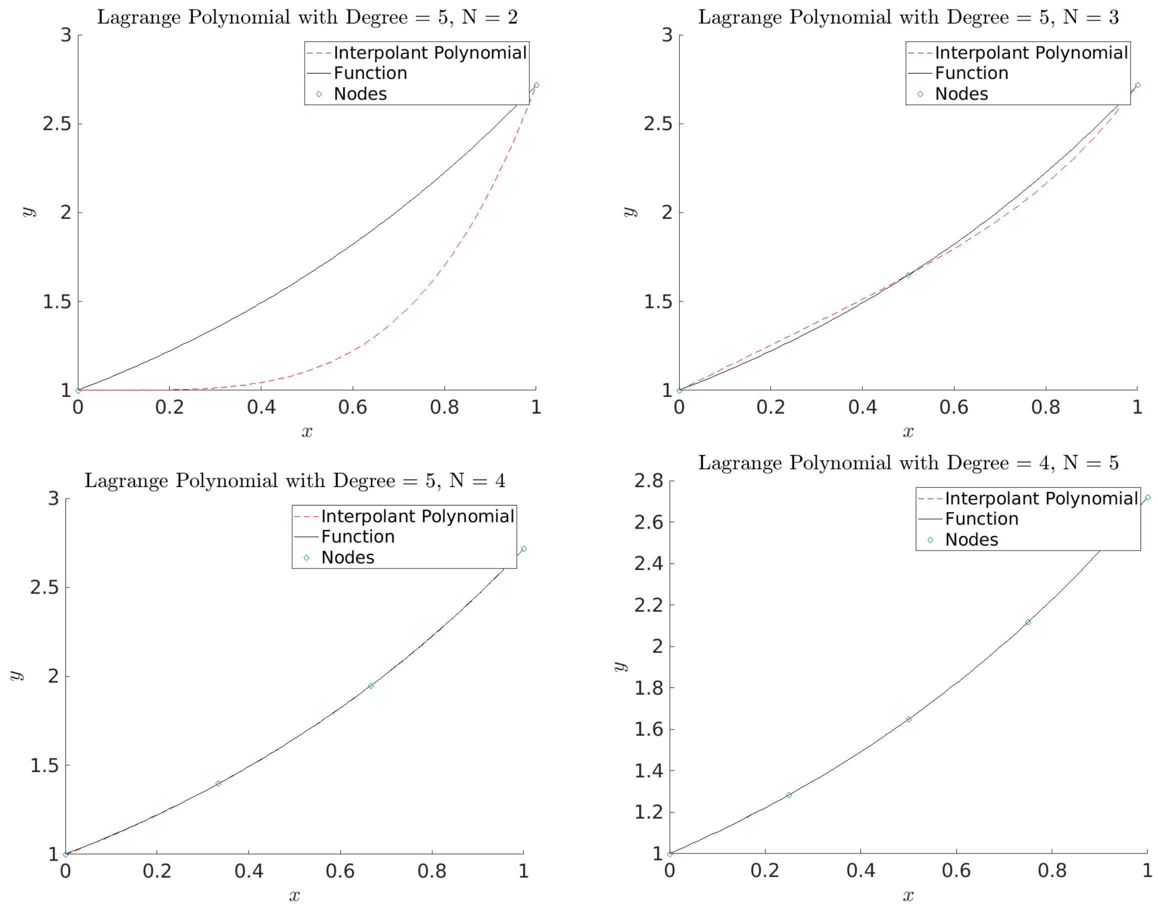


Figure 4: Graphs for interpolant functions approximating $f(x) = e^x$ where the degree is fixed, $d = 5$, and the nodes are varied from $N = 2, \ldots, 4$. The last plot is the Lagrange polynomial for 5 nodes.

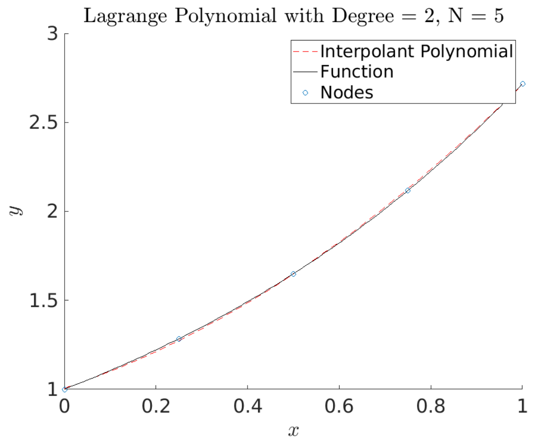


Figure 5: This plot is of $f(x) = e^x$ with 5 nodes and a degree of 2.

(b) $f(x) = \text{humps}(x)$

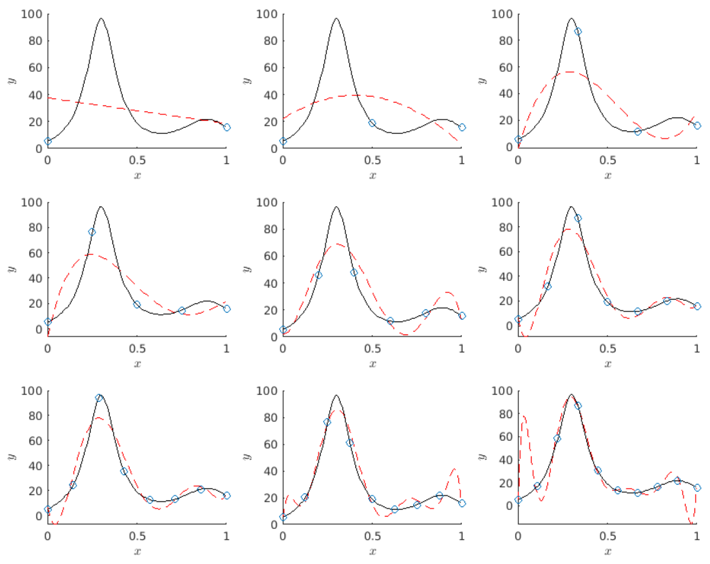


Figure 6: A plot of Lagrange interpolating polynomials approximating the MATLAB humps function.

For the humps function, things were quite different in terms of fixing either the nodes or the degree of the polynomial. By fixing the degree to be $d = 10$, and varying the nodes from $N = 10, \ldots, 80$, we observe that no matter how many nodes are used, the polynomial remains the same after a certain number of nodes are used, which is indeed interesting. Thus, for the case of the humps function, a 10 degree polynomial is not sufficient enough to approximate the function, even with 80 nodes. For a separate case, when the number of nodes is 10 and the degree is 20, the interpolant did not approximate the function very well. For the last case, the number of nodes was 100 and the degree of the polynomial was 15, and the interpolant approximated the function decently. However, I would really like to quantify the accuracy of the approximation. How good is "decently" good? Hopefully I will learn as I continue in this class.

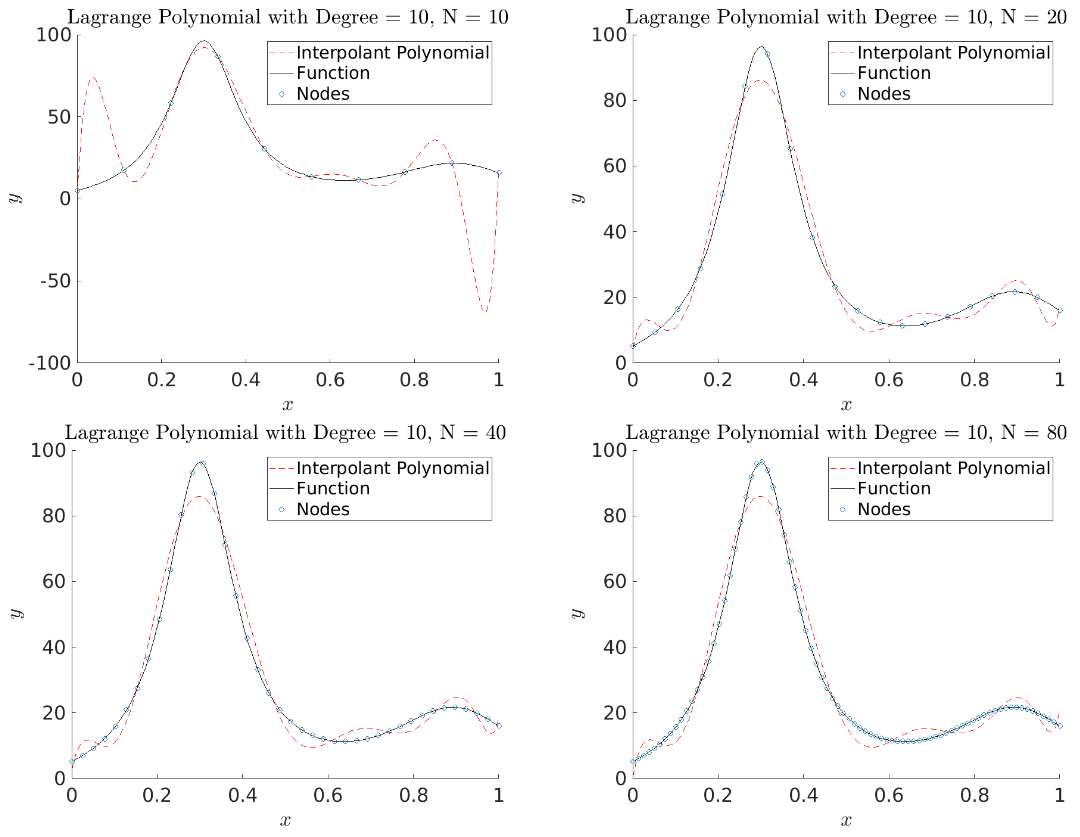


Figure 7: Here, the degree of the polynomial is fixed at 10 and the nodes change from $N = 10, \ldots, 80$.

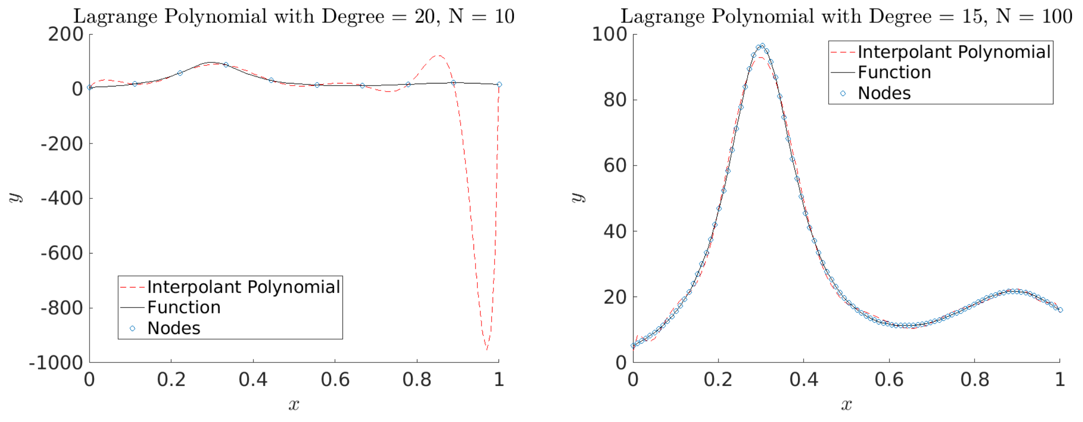


Figure 8: On the left is the interpolant evaluated using 10 nodes and a degree of 20. The right is the interpolant using 100 nodes and a degree of 15.

```
%% Exercise 2

N = 5; % number of nodes and degree of interpolating polynomial
D = N-1; % degree of interpolant polynomial

x = linspace(0,1,N); % interval from 0 to 1 of equidistant nodes
y = exp(x); % function in question
p = polyfit(x,y,D); % coefficients for interpolant polynomial of degree D
xx = linspace(0,1,100); % finer grid for standard plot
yy = exp(xx); % function evaluated on finer grid

% Graphing

figure
hold on
plot(xx,polyval(p,xx),'r--'); % plot of the lagrange interpolating ...
    polynomial
plot(xx,yy,'k'); % plot of the function
plot(x,y,'o'); % plot of nodes
xlabel('$x$','Interpreter','latex'); % x-axis
ylabel('$y$','Interpreter','latex'); % y-axis
legend('Interpolant Polynomial','Function','Nodes');
title(sprintf('Lagrange Polynomial with Degree = %1.f, N = ...
    %1.f',D,N),'interpreter','latex');
set(gca,'fontsize',23);

for k = 1:9
    subplot(3,3,k);
    hold on

    u = linspace(0,1,k+1); % interval from 0 to 1 of equidistant nodes
    v = exp(u); % function in question
    pp = polyfit(x,y,k); % coefficients for interpolant polynomial of ...
        degree D
    uu = linspace(0,1,100); % finer grid for standard plot
    vv = exp(xx); % function evaluated on finer grid
    plot(uu,polyval(pp,uu),'r--'); % plot of the lagrange interpolating ...
        polynomial
    plot(uu,vv,'k'); % plot of the function
    plot(u,v,'o'); % plot of nodes
    xlabel('$x$','Interpreter','latex'); % x-axis
    ylabel('$y$','Interpreter','latex'); % y-axis
end

%% Exercise 3

N = 11; % number of nodes and degree of interpolating polynomial
D = N-1; % degree of interpolant polynomial
```

```

x = linspace(0,1,N); % interval form 0 to 1 of equidistant nodes
y = humps(x); % function in question
p = polyfit(x,y,D); % coefficients for interpolant polynomial of degree D
xx = linspace(0,1,100); % finer grid for standard plot
yy = humps(xx); % function evaluated on finer grid

% Graph

figure
hold on
plot(xx,polyval(p,xx),'r—'); % plot of the lagrange interpolating ...
    polynomial
plot(xx,yy,'k'); % plot of the function
plot(x,y,'o'); % plot of nodes
xlabel('$x$','Interpreter','latex'); % x—axis
ylabel('$y$','Interpreter','latex'); % y—axis
ylim([min(y),max(y)]);
legend('Interpolant Polynomial','Function','Nodes');
title(sprintf('Lagrange Polynomial with Degree = %1.f, N = ...
    %1.f',D,N),'interpreter','latex');
set(gca,'fontsize',23);

for k = 1:9
    subplot(3,3,k);
    hold on

    u = linspace(0,1,k+1); % interval from 0 to 1 of equidistant nodes
    v = humps(u); % function in question
    pp = polyfit(x,y,k); % coefficients for interpolant polynomial of ...
        degree D
    uu = linspace(0,1,100); % finer grid for standard plot
    vv = humps(xx); % function evaluated on finer grid
    plot(uu,polyval(pp,uu),'r—'); % plot of the lagrange interpolating ...
        polynomial
    plot(uu,vv,'k'); % plot of the function
    plot(u,v,'o'); % plot of nodes
    xlabel('$x$','Interpreter','latex'); % x—axis
    ylabel('$y$','Interpreter','latex'); % y—axis
end
```

HOMEWORK 2

DERIVING AND CODING ADAPTIVE QUADRATURE: SIMPSON'S RULE

Implement adaptive Simpson's rule and test it by integrating **`humps`** *on* $[0, 8]$ *with tolerance ranging from* 10^{-1} *to* 10^{-10}*. Present your results in tabular form where for each tolerance level you compute the approximate value of the integral, the absolute error, and the number of evaluations. Additionally, make a log-log plot of the number of functional evaluations (cost) against the tolerance, similar to Figure 11. What can you say about the efficiency of the adaptive Simpson's rule?*

Answer: To start, we begin with the derivation of Simpson's rule. Let $f \in C^3[a, b]$, since we are using three nodes for interpolation. So, our interpolation of f is given by

$$f(x) = \sum_{k=1}^{3} L_k(x) f(x_k) + \frac{f^{(4)}(\xi)}{6}(x - x_1)(x - x_2)(x - x_3),$$

where $L_k(x)$ are the Lagrange polynomials

$$L_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)}, \qquad L_2(x) = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)}, \qquad L_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}.$$

Integrating both sides over the interval gives us

$$\int_{x_1}^{x_3} f(x)\ dx = \int_{x_1}^{x_3} \sum_{k=1}^{3} L_k(x) f(x_k) dx + \int_{x_1}^{x_3} \frac{f^{(4)}(\xi(x))}{6}(x - x_1)(x - x_2)(x - x_3) dx.$$

To avoid clutter, we will make the following substitutions: $x_1 = a, x_2 = c$, and $x_3 = b$, defining c as the midpoint of a and b. Applying the property of linearity to the first integral and applying the additive property of integration to the second integral, we obtain

$$\begin{aligned}\int_a^b f(x)\ dx = &\sum_{k=1}^{3} \int_a^b L_k(x) f(x_k) dx \\ &+ \frac{f^{(4)}(\xi_1)}{6} \int_a^c (x - a)(x - c)(x - b) dx \\ &+ \frac{f^{(4)}(\xi_2)}{6} \int_c^b (x - a)(x - c)(x - b) dx.\end{aligned}$$

After computing the integrals and simplifying each expression, we obtain

$$\int_a^b f(x)\ dx = \frac{(b - a)}{6}\big(f(a) + 4f(c) + f(b)\big) + \frac{(b - a)^4}{384}\big(f^{(4)}(\xi_1) - f^{(4)}(\xi_2)\big)$$

By applying first the Fundamental Theorem of Calculus and then the Mean Value Theorem to the error, we get

$$\int_a^b f(x)\ dx = \frac{(b-a)}{6}\big(f(a)+4f(c)+f(b)\big) + \frac{f^{(4)}(\eta)}{384}(b-a)^4(\xi_1-\xi_2).$$

Although this expression for the error is true[2], we apply Peano's Kernel Theorem to obtain a not only more desirable form, but a form that will be useful for coding our adaptive quadrature function. By doing so, we obtain

$$\int_a^b f(x)\ dx = \frac{(b-a)}{6}\big(f(a)+4f(c)+f(b)\big) - \frac{f^{(4)}(\xi)}{2880}(b-a)^5 \tag{2}$$

$$= \frac{h}{3}\big(f(a)+4f(c)+f(b)\big) - \frac{f^{(4)}(\xi)}{90}h^5, \tag{3}$$

where $b-a=2h$, and equation (3) is the expression found in most numerical analysis textbooks. To code our adaptive quadrature rule for Simpson's Rule, we use equation (2). Equation (2) was derived using three nodes, with two equispaced subdivisions. When applying Simpson's rule to five nodes, with four equispaced subdivisions, the expression we obtain, in addition to equation (2), is

$$\int_a^b f(x)\ dx = \frac{(b-a)}{6}\big(f(a)+4f(c)+f(b)\big) - \frac{f^{(4)}(\xi_1)}{2880}(b-a)^5 \tag{4}$$

$$\int_a^b f(x)\ dx = \frac{(b-a)}{12}\big(f(a)+4\big(f(d)+f(e)\big)+2f(c)+f(b)\big) - \frac{f^{(4)}(\xi_2)}{46080}(b-a)^5, \tag{5}$$

where d is the midpoint of a and c and e is the midpoint of c and b:

$$c = \frac{a+b}{2}, \qquad d = \frac{3a+b}{4}, \qquad e = \frac{a+3b}{4}.$$

Let's abbreviate the quadrature rules in equations (4) and (5) as S_1 and S_2, respectively. By equating equations (4) and (5), assuming that $f^{(4)}(\xi_1) = f^{(4)}(\xi_2)$, and solving for the error, we get

$$\frac{f^{(4)}(\xi_1)}{2880}(b-a)^5 = -\frac{16}{15}(S_2-S_1). \tag{6}$$

Substituting the above expression into either equation (4) or (5) gives us

$$\int_a^b f(x)\ dx = \frac{1}{15}(16S_2-S_1). \tag{7}$$

[2]This expression is true because as the step size decreases, and because by the Extreme Value Theorem, $(\xi_1-\xi_2)\le 2h$ where $h=(b-a)/2$, as h decreases the difference $(\xi_1-\xi_2)$ becomes an additional factor.

Now, we can use equations (6) and (7) to construct an adaptive quadrature routine in the same manner as the adaptive trapezoid routine in the hand out with a few modifications, provided below.

```
function [Q,evals] = a_simpson(f,a,b,tol)

    % Adaptive Simpson's Rule Overview
    %
    % This function implements Simpson's rule using adaptive quadrature to
    % numerically compute integrals. The function takes a parent function
    % f, the endpoints, a & b, and the desired tolerance, tol, as inputs.

% Code Body

fa = f(a);        % the function evaluated at left endpoint
fc = f((a+b)/2); % function evaluated at midpoint of a & b
fb = f(b);        % function evaluated at right endpoint
evals = 3;         % there are already three function evaluations
Q = (fa + 4*fc + fb)*(b-a)/6; % Simpson's rule applied once
Q = quadstep(a,b,fa,fb,Q);

    function q = quadstep(a,b,fa,fb,S1)
        h = b-a;
        d = a + 0.25*h;
        m = a + 0.5*h;
        e = a + 0.75*h;
        fd = f(d);
        fm = f(m);
        fe = f(e);
        evals = evals + 2;
        Sleft = (fa + 4*fd + fm)*h/12;
        Sright = (fm + 4*fe + fb)*h/12;
        S2 = Sleft + Sright;
        E = 16*abs(S2 - S1)/15;
        if (b - a)*E < tol
            q = (16*S2 - S1)/15;
        else
            q = quadstep(a,m,fa,fm,Sleft) + quadstep(m,b,fm,fb,Sright);
        end


    end


end
```

As we can see, we include the midpoint and the midpoints of both subintervals. Then, we have the function evaluated at those points and we compute Simpson's rule for each subinterval, taking their superposition afterwards. Then, we implement a conditional statement to see if the quadrature rule meets the desired tolerance. If not, the process repeats. The table is below. For the function evaluations, when implementing Simpson's rule initially, there are only three—namely evaluating at the endpoints and at the midpoint. When applying Simpson's rule twice, as in the case of equation (5) there are two additional function evaluations. Thus, we have two added to the original function evaluations.

Tolerance	Approximate Value of Integral	Absolute Error	Functional Evaluations
10^{-1}	-5.498138637457739	0.004761858389888	37
10^{-2}	-5.490011872602810	4.223958931675043e-05	53
10^{-3}	-5.458028545549226	4.223958931675043e-05	81
10^{-4}	-5.457644392070641	4.223958931675043e-05	133
10^{-5}	-5.457635305163176	8.071421130277183e-07	157
10^{-6}	-5.457630560827074	8.071421130277183e-07	273
10^{-7}	-5.457631121001405	2.017872115326706e-08	369
10^{-8}	-5.457631135561272	5.667475970009642e-10	577
10^{-9}	-5.457631133678461	5.667475970009642e-10	785
10^{-10}	-5.457631133619131	1.680930949987669e-11	1173

Table 1: This table includes the tolerance, the approximation of the integral, the absolute error, and the number of functional evaluations.

Some things to note is that from 10^{-2} to 10^{-4} the error doesn't change even though we see differences in the approximations. This also occurs for other values in the table. I'm honestly not sure why this occurred. It may be due to the difference being so small that not even formatting with `long` works, but the differences in looking at the approximate values of the integral aren't so small so as to maintain the same value for different tolerances. However, here is the log-log plot of the function counts against the tolerance, and the code preceding it.

```
%% Homework 2

% log—log plot

x = [1/10 1/100 1/1000 1/10000 1/100000 1/1000000 ...
    1/10000000 1/100000000 1/1000000000 1/10000000000];
y = [37 53 81 133 157 273 369 577 785 1173];

logx = log(x);
logy = log(y);
[p,S] = polyfit(logx,logy,1);
pp = polyval(p,logx);

```

```
figure
hold on
plot(logx,logy,'bo');
plot(logx,pp,'b-');
xlabel('$log(tol)$','interpreter','latex');
ylabel('$log(evals)$','interpreter','latex');
title(['Log of Function' ...
    ' Evaluations vs. Log of Tolerance'],'interpreter','latex');
legend('Data','Fit','interpreter','latex');
set(gca,'fontsize',20);
```

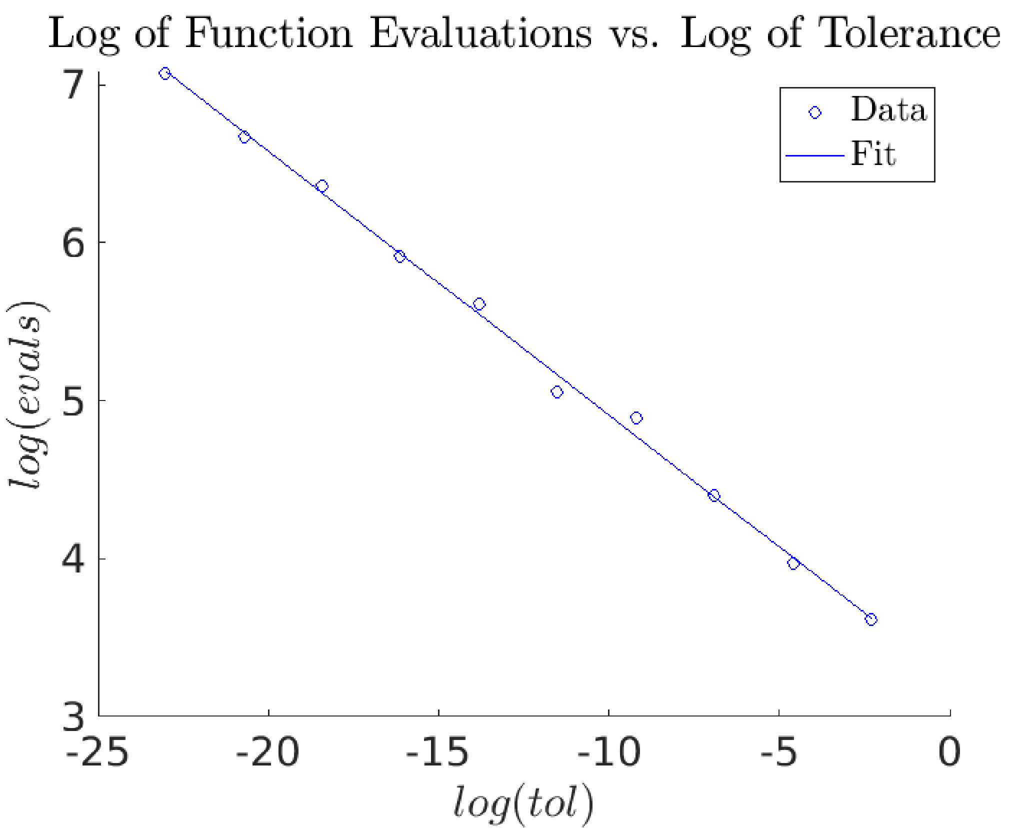


Figure 9: Plot of the $\log(evals)$ against $\log(tol)$. The slope of the line is

The slope of the fit is -0.1669. Clearly, this method is quite efficient, especially compared to adaptive trapezoid rule. For a tolerance of 10^{-6}, adaptive trapezoid rule required 607 functional evaluations whereas adaptive Simpson's rule requires only 273 functional evaluations, which is two times less than that of adaptive trapezoid rule, which is pretty significant.

```
function A = mid_quad(a,b,N,f)
    %
    % Function Overview
    %
    % This function implements the midpoint rule to compute integrals
    % numerically. The function takes the endpoints (a,b), the number of
    % subdivisions (N), and the function in question (f) as inputs and
    % returns the "area under the curve," or integral, A. The key ...
       assumption
    % one must make to implement the midpoint rule is that the data is
    % piecewise constant. As such, the data used for quadrature is ...
       interpolated using constant
    % segments whose value is the value of the function evaluated at
    % exactly the midpoint of each pair of data points (hence the midpoint
    % rule), assuming that each interval is exactly the same width, and ...
       further, symmetric.
    %
    % Limits to the Method Below and Ways to Improve
    %
    % The method below uses a "for" loop, which restricts the number of
    % subdivisions that can be used to approximate the integral, even
    % though the function is only several lines of code. Improvement ...
       can be
    % achieved, I think with the use of more linear algebra to implement
    % the use of matrix multiplication, which is well suited for MATLAB.
    % Other ways to improve this code is to implement an input for
    % tolerance, so as to quantify how accurate the code is and so it can
    % be compared with other quadrature methods.
    %
    % Updates
    %
    % The code was re-written withough using "for" loops. A comparison was
    % made using y = sin(x), a = 0, b = pi, and N = 10000. For this ...
       test run,
    % the updated code was over 100 times faster than the original code


% Original Code

% tic
% x = linspace(a,b,N); % forming the domain of integration
%
% Δ_x = (b-a)/N; % creating the step-size
%
% y = @(x) f(x); % the function in question
%
% A = zeros(N); % array for size preallocation
%
```

```
% for n = 1:N-1
%     A(n) = y((x(n) + x(n+1))/2).*Δ_x; % midpoint rule
%     A = sum(A); % computing the riemann sum
% end
% toc

% Updated Code

tic
x = linspace(a,b,N); % forming domain of integration

Δ_x = (b-a)/N; % step-size

y = @(x) f(x); % defining the function

A = zeros(N); % array for preallocation

xx = (x(1:end-1) + x(2:end))/2; % array of midpoints

yy = f(xx); % function evaluated at midpoints

A = sum(yy.*Δ_x); % integral
toc

end
```

```
function A = trap_quad(a,b,N,f)
    %
    % Function Overview
    %
    % This code implements the trapezoid rule for numerical integration.
    % The function takes the endpoints (a,b), the number of
    % subdivisions (N), and the function in question (f) as inputs and
    % returns the "area under the curve," or integral, A. The key
    % assumption that must be made in order to use the trapezoid rule is
    % that the data is piecewise linear. Thus, the function data is
    % interpolated using linear approximations, and the areas of the
    % trapezoids are thus added together for the integral. For this code
    % the, size of the subinterval is assumed to be the same for each
    % subinterval.
    %
    % Limits to the Original Method and Ways to Improve
    %
    % As with the code for the midpoint rule, a "for" loop is used, which
    % restricts the number of rectangles to approximate the integral of the
    % target function. Further, there should also be an input for tolerance
    % to determine and compare the effectiveness of the trapezoid rule.
    %
    % Updates
    %
    % The updated code contains no "for" loops. A comparison was
    % made using y = sin(x), a = 0, b = pi, and N = 10000. For this ...
       test run,
    % the updated code was over 150 times faster than the original code.

% Original Code

% tic
% x = linspace(a,b,N); % creating domain with constant step-size
%
% Δ_x = (b-a)/N; % creating step-size
%
% y = @(x) f(x); % function in question
%
% A = zeros(N); % array for preallocation
%
% for n = 1:N-1
%     A(n) = ((y(x(n)) + y(x(n+1)))/2).*Δ_x;
%     A = sum(A);
% end
% toc

% Updated Code

tic
x = linspace(a,b,N); % creating domain with constant step-size

Δ_x = (b-a)/N; % creating step-size

y = @(x) f(x); % function in question

A = zeros(N); % array for preallocation

yy = f(x); % function

yy = (yy(1:end-1) + yy(2:end))/2;

A = sum(yy.*Δ_x); % area of trapezoids summed together
toc

end
```

EXAM 1

EXERCISE 1

(10 points) Consider the problem of solving $f(x) = 0$. Suppose that it is known that (i) the equation has a unique positive root x_*, and (ii) $f(0) = -10, f(1) = 12$, and $f(2) = 8$. Find an estimate for x_* and call it x_1. Next, propose a computational scheme for refining x_1 into a better estimate x_2. Extra points if you try the scheme on an equation of your choice.

Solution: In order to determine x_*, I first used Lagrange interpolation, which for three points is of the form

$$f(x) \approx \sum_{j=0}^{2} f(x_j) \prod_{\substack{i=0 \\ i \neq j}}^{2} \frac{x - x_i}{x_j - x_i}.$$

I must note that by using Lagrange interpolation for three nodes, I am assuming the data to be quadratic, which means $f(x) \in C^2[a, b]$. To find an estimate x_1 of the root x_*, I implemented my own code utilizing Newton's method, which is used to approximate the root(s) of a function. Given a closed interval $[a, b]$, the use of Newton's method demands that the function be at least once differentiable, or $f(x) \in C^1[a, b]$, because the method involves the use of the function's first order Taylor polynomial. Generally, for a function $f(x) \in C^1[a, b]$ with a root $x_* \in [a, b]$, we can choose some $x_0 \in [a, b]$ as our center of expansion for our first order Taylor polynomial. As such, we have

$$T_1(x) = f(x_0) + f'(x_0)(x - x_0).$$

By setting $T_1 = 0$, we obtain the root to our tangent line, which serves as an approximation to the root of our function f:

$$\begin{aligned} 0 = f(x_0) + f'(x_0)(x_1 - x_0) &\implies x_1 - x_0 = -\frac{f(x_0)}{f'(x_0)} \\ &\implies x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \end{aligned}$$

By performing this a second time using x_1 as the center of expansion for our Taylor polynomial, we find that the better estimate of the root x_2 is

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}.$$

If this is computed many more times, then we are able to find a recursive algorithm with which we can compute the root(s) of functions:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

The function is presented below.

```
function [x1,x2] = newt_method(f,dfdx,x0,N)

x = [x0 zeros(1,N-1)]; % vector of length N

for n = 1:N-1
    x(n+1) = x(n) - f(x(n))/dfdx(x(n));
end

x1 = x(2); % approximation of x*
x2 = x(3); % better estimate of x*

end
```

This function was used to compute of the interpolated quadratic and the function $y = \sin(x)$. The resulting code and plots are below.

```
%% Exercise 1

% Function evaluations

u = [0,1,2];
v = [-10,12,8];

% Lagrange interpolation

xx = linspace(0,2);
p = polyfit(u,v,2);
pp = polyval(p,xx);

% Functions

f = @(x) p(1)*x.^2 + p(2)*x + p(3); % interpolated quadratic
dfdx = @(x) 2*p(1)*x + p(2);

f1 = @(x) sin(x); % my function
df1dx = @(x) cos(x);

% Roots

[x1,x2] = newt_method(f,dfdx,0.25,5);
root1 = fzero(f,0.25);

[xx1,xx2] = newt_method(f1,df1dx,3*pi/4,5);
root2 = fzero(f1,3*pi/4);

```

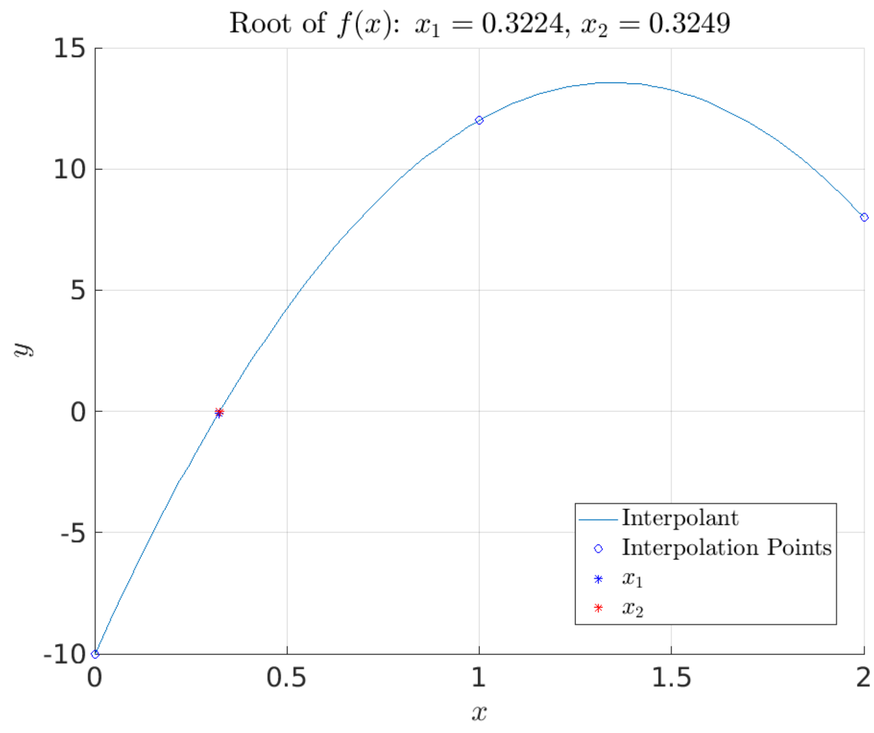
```
% Graph of interpolant

figure
hold on
grid on
plot(xx,pp);
plot(u,v,'bo');
plot(x1,f(x1),'b*',x2,f(x2),'r*');
xlabel('$x$','interpreter','latex');
ylabel('$y$','interpreter','latex');
legend('Interpolant','Interpolation ...
    Points','$x_{1}$','$x_{2}$','interpreter','latex');
title(sprintf('Root of $f(x)$: $x_{1} = %1.4f$, $x_{2} = ...
    %1.4f$',[x1,x2]),'interpreter','latex');
set(gca,'fontsize',20);

% Graph of my function

figure
hold on
grid on
plot(0:0.01:2*pi,f1(0:0.01:2*pi));
plot(xx1,f1(xx1),'b*',xx2,f1(xx2),'r*');
xlabel('$x$','interpreter','latex');
ylabel('$y$','interpreter','latex');
legend('$f(x) = \sin(x)$','$x_{1}$','$x_{2}$','interpreter','latex');
title(sprintf('Root of $f(x) =$ sin$(x)$: $x_{1} = %1.4f$, $x_{2} = ...
    %1.4f$',[xx1,xx2]),'interpreter','latex');
set(gca,'fontsize',20);
```

As we can see in the figures below, our approximations of the root for the data provided are

$$\boxed{x_1 = 0.3224 \quad \text{and} \quad x_2 = 0.3249},$$

and the approximations of the root for the sin function are

$$\boxed{x_1 = 3.3562 \quad \text{and} \quad x_2 = 3.1382}.$$



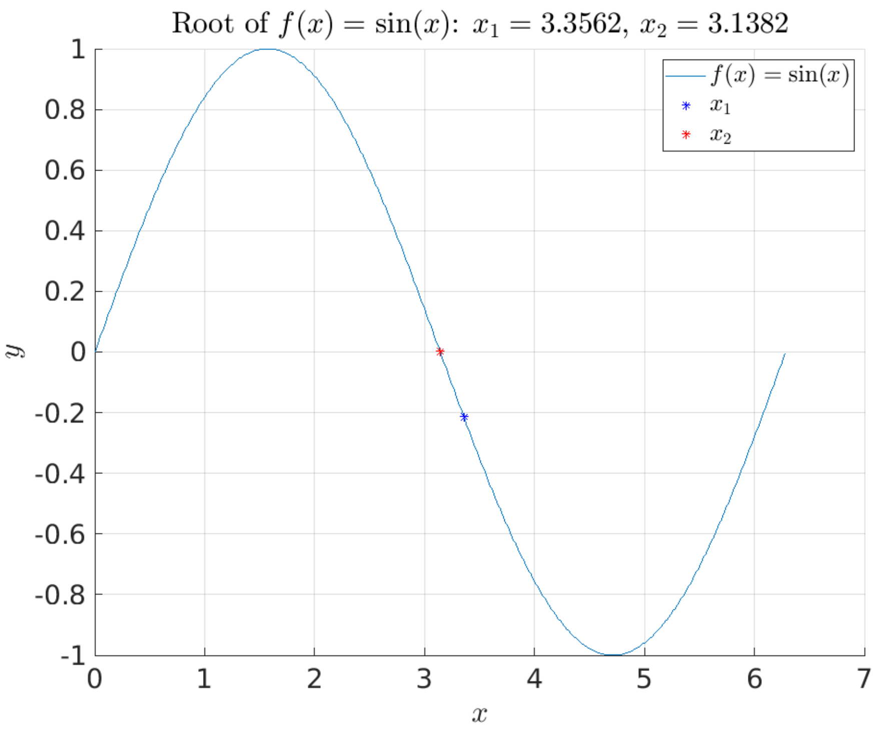


Figure 10: The first figure is a plot of the interpolated function and it's root estimates. The second plot is of the sin function and it's root estimates near π.

EXERCISE 2

(10 points) Let

$$E(f) = \int_0^1 x^2(1-x)^2 f^{(3)}(x)dx$$

You can think of E as an error functional for some quadrature rule. if $a \le |f^{(3)}(x)| \le \pi$ for all $x \in [0,1]$, what are the corresponding bounds for $|E(f)|$ Explain you answer.

Solution: We begin with the following:

$$\begin{aligned}|E(f)| &= \left|\int_0^1 x^2(1-x)^2 f^{(3)}(x)dx\right| \\ &\le \int_0^1 \left|x^2(1-x)^2\right|\left|f^{(3)}(x)\right|dx = \int_0^1 x^2(1-x)^2\left|f^{(3)}(x)\right|dx \\ &\le \int_0^1 x^2(1-x)^2\pi dx = \frac{\pi}{30}\end{aligned}$$

Firstly, the inequalities are true for multiple reasons. We can use logic to determine why the first inequality is true. If the integrand changes sign and is integrated, then the integral will be positive in some regions and negative in other regions. After taking the sum of the the positive and negative areas, the absolute value of that number will be less than if the absolute value of the integrand was integrated. This is so because in the latter case, the absolute value is being applied only to the integrand, and the integrand will always be positive. Thus, integrating an always positive function and then taking its absolute value would mean that the areas are all positive, giving a larger number than if there were positive and negative areas. Further, the second equality is true because $x^2(1-x)^2$ doesn't change sign, as demonstrated by the plot.

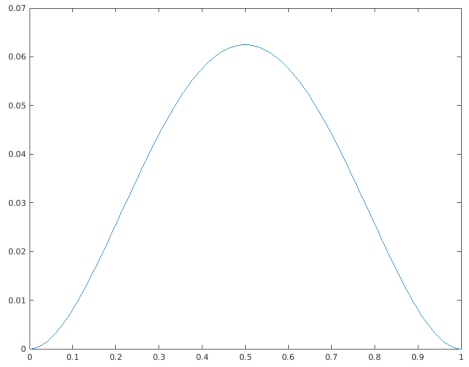


Figure 11: The plot of $x^2(1-x)^2$.

Because this function doesn't change sign, we can remove the absolute value. Next, the second equality is true due to the given bounds and the Extreme Value Theorem:

$$\inf_{x\in[0,1]} f^{(3)}(x) = 1 \leq \left|f^{(3)}(x)\right| \leq \pi = \sup_{x\in[0,1]} f^{(3)}(x).$$

So, the bound of our error functional is

$$\begin{aligned}\left|E(f)\right| \leq \frac{\pi}{30} &\implies E(f) \leq \frac{\pi}{30} \quad \text{and} \quad -E(f) \leq \frac{\pi}{30}\\ &\implies \boxed{-\frac{\pi}{30} \leq E(f) \leq \frac{\pi}{30}}.\end{aligned}$$

EXERCISE 3

(15 Points) Let $Q_N : f \mapsto (b-a)\sum_{n=0}^{N-1} w_n f(x_n)$ denote the Newton-Cotes quadrature with equispaced nodes $x_n = a + \frac{b-a}{N-1}n$. The coefficients w_n are usually referred to as the weights of the quadrature and are some numbers that depend on N. For instance, for the left-end point rule $w_1 = 1$; for the trapezoid rule $w_1 = w2 = \frac{1}{2}$; for the Simpson rule $w_1 = w_3 = \frac{1}{6}$ $w_2 = \frac{2}{3}$; and so on. Notice that for $N = 1, 2, 3$ the weights are all positive. The question is: do Newton-Cotes weights remain positive for higher N? Investigate the question computationally. If the weights stop being positive for some N, find that N and list the weights. If they always remain positive, try to find an explanation.

Solution: In order to compute the weights w_n, which depend on the number of nodes N, I created a function in MATLAB. Let's look at some theory first. For the quadrature rule $Q_N : f \mapsto (b-a)\sum_{n=0}^{N-1} w_n f(x_n)$ with nodes $x_n = a + \frac{b-a}{N-1}n$, we can use the error functional to give us an idea what our code should look like. I used the cases $N = 2$ and $N = 3$ to determine a way to construct a linear system, which depends on N, so as to determine the weights. For the $N = 2$ case, we have two unknown weights and we know what our nodes are. Thus, we need two equations. These equations were constructed using the error functional. We define our error functional as

$$E_N(f) = I(f) - Q_N(f),$$

where $I(f) = \int_a^b f(x)dx$. Because we have two nodes, we can interpolate linearly, and we would like for our error functional to at least annihilate the monomial $f(x) = 1$ and the first degree monomial $f(x) = x$. Thus, we have

$$\begin{aligned}E_2(1) &= I(1) - Q_2(1) = 0\\ E_2(x) &= I(x) - Q_2(x) = 0.\end{aligned}$$

By including all the details, we obtain the following matrix-vector system:

$$\begin{bmatrix} 1 & 1 \\ x_0 & x_1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \frac{1}{(b-a)} \begin{bmatrix} I(1) \\ I(x) \end{bmatrix}$$

and for $N = 3$, where we want our error functional to annihilate at least quadratics, we have

$$\begin{bmatrix} 1 & 1 & 1 \\ x_0 & x_1 & x_2 \\ x_0^2 & x_1^2 & x_2^2 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \frac{1}{(b-a)} \begin{bmatrix} I(1) \\ I(x) \\ I(x^2) \end{bmatrix}$$

The matrix contains the function values evaluated at the nodes and the right-hand side contains the integral of the functions. This was the foundation of the function I wrote, which is below.

```matlab
function weights = newt_cotes_weights(a,b,N)

    % Newton-Cotes Weight Determination
    %
    % This code determines the weights for Newton-Cotes methods for any
    % number of nodes N. This also takes in the endpoints a and b as
    % inputs. The number of nodes must be no less than one. Further, having
    % zero as an endpoint provides a poor result from integrating the
    % interpolant, hence the error message for zero being an endpoint. This
    % code also works for any two endpoints.

if N < 1
    error('The number of nodes N must at least be 1.')
end

if a == 0
    error('Zero is bad. Pick another endpoint.')
end

A = ones(N,N);
q = ones(N,1);
x = ones(1,N);

for k = 0:N-1
    x = linspace(a,b,N);
    y = @(x) x.^k;
    q(k+1,:) = quadgk(y,a,b)/(b-a);
    A(k+1,:) = y(x);
end

weights = A\q;

end
```

The code above begins with an error message if the number of nodes is less than zero. Then, another error message follows if one of the nodes is zero, due to a poor quadrature approximation. Aside from these two issues, the function takes any two endpoints as inputs and an N value that is not to large, due to matrix conditioning (which could be addressed using `pinv()`). Then, the function determines the weights by solving the appropriate matrix-vector system. To test this code, I tried the cases $N = 1, 2, 3, 4$. These are the outputs.

```
% N = 1

>> newt_cotes_weights(1,3,1)

ans =

     1

% N = 2

>> newt_cotes_weights(-1,3,2)

ans =

    0.5000
    0.5000

% N = 3

>> newt_cotes_weights(-10,100,3)

ans =

    0.1667
    0.6667
    0.1667

% N = 4

>> newt_cotes_weights(-10,2,4)

ans =

    0.1250
    0.3750
    0.3750
    0.1250
```

The code gives us exactly what we would expect for $N = 1, 2, 3, 4$. So to determine if Newton-Cotes methods have negative values, I just incremented N until the code produced negative nodes. This occurred at $N = 9$. The output is below.

```
% N = 9

>> newt_cotes_weights(-1,2,9)

ans =

    0.0349
    0.2077
   -0.0327
    0.3702
   -0.1601
    0.3702
   -0.0327
    0.2077
    0.0349
```

EXERCISE 4

(15 points) Consider the following quadrature rule as an approximation to $\int_{-1}^{1} f(x)dx$:

$$Q(f) = f\big(-1/\sqrt{3}\big) + f\big(1/\sqrt{3}\big).$$

Use Peano's Kernel Theorem to derive an expression for the error of this rule. If you can, write the final answer in the form $Cf^{(k)}(\xi)$—the Cauchy form of the remainder.

Solution: To begin, we use the error function to determine for which monomials our quadrature rule is exact:

$$E(f) = \int_{-1}^{1} f(x)dx - f\big(-1/\sqrt{3}\big) - f\big(1/\sqrt{3}\big). \tag{8}$$

By applying our error functional to monomials of increasing degree, we find that

$$\begin{aligned}
E(1) &= \int_{-1}^{1} 1dx - 1 - 1 = 0,\\
E(x) &= \int_{-1}^{1} xdx + \frac{1}{\sqrt{3}} - \frac{1}{\sqrt{3}} = 0,\\
E(x^2) &= \int_{-1}^{1} x^2dx - \left(-\frac{1}{\sqrt{3}}\right)^2 - \left(\frac{1}{\sqrt{3}}\right)^2 = 0,\\
E(x^3) &= \int_{-1}^{1} x^3dx - \left(-\frac{1}{\sqrt{3}}\right)^3 - \left(\frac{1}{\sqrt{3}}\right)^3 = 0.
\end{aligned}$$

Our error functional annihilates up to cubic monomials. For quartic monomials, the error we obtain is

$$E(x^4) = \int_{-1}^{1} x^4 dx - \Big(-\frac{1}{\sqrt{3}}\Big)^4 - \Big(\frac{1}{\sqrt{3}}\Big)^4 = \frac{8}{45}.$$

Thus, by application of Peano's Kernel theorem, because our error functional is linear and annihilates monomials (and thus polynomials) of degree 3, we have

$$E(f) = \int_{-1}^{1} f^{(4)} K(t) dt,$$

where

$$K(t) = \frac{1}{3!} E_x((x-t)_+^3).$$

Now, to determine an explicit form for the kernel $K(t)$, we substitute $(x-t)_+^3$ into equation (8). By doing so, we obtain

$$E_x((x-t)_+^3) = \int_{-1}^{1} (x-t)_+^3 dx - \Big(-\frac{1}{\sqrt{3}} - t\Big)_+^3 - \Big(\frac{1}{\sqrt{3}} - t\Big)_+^3$$

To compute the integral of the truncated polynomial, we know that as long as $x < t$, the integral will return zero by definition of the truncated polynomial. So, our lower bound is t instead of -1. By performing the integration, we obtain

$$K(t) = \frac{(1-t)^4}{4} - \Big(-\frac{1}{\sqrt{3}} - t\Big)_+^3 - \Big(\frac{1}{\sqrt{3}} - t\Big)_+^3,$$

which is our kernel. So, the error becomes

$$E(f) = \frac{1}{6}\int_{-1}^{1} f^{(4)}(t)\Big[\frac{(1-t)^4}{4} - \Big(-\frac{1}{\sqrt{3}} - t\Big)_+^3 - \Big(\frac{1}{\sqrt{3}} - t\Big)_+^3\Big] dt.$$

Now, we apply the GMVT. However, we must confirm that the kernel doesn't change sign. This is confirmed with the following MATLAB code and plot.

```
%% Exercise 4

t = linspace(-1,1);
K = @(t) (((1-t).^4)/4 - ((-1/sqrt(3) - t).^3).*(-1 ≤ t ≤ -1/sqrt(3))...
    - ((1/sqrt(3) - t).^3).*(-1 ≤ t ≤ 1/sqrt(3)))/6;

plot(t,K(t));
```

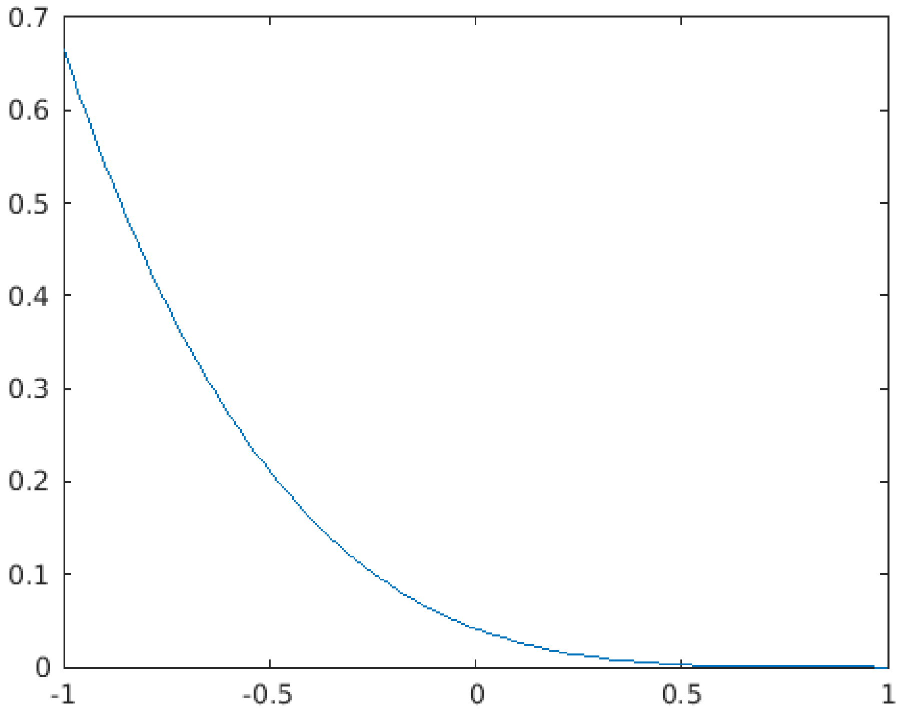


Figure 12: A plot of the Kernel. It indeed does not change sign over the interval $[-1,1]$.

Because the Kernel doesn't change sign, we can apply the GMVT and evaluate the integral:

$$\begin{aligned}
E(f) &= \frac{1}{6}\int_{-1}^{1} f^{(4)}(t)\left[\frac{(1-t)^4}{4} - \left(-\frac{1}{\sqrt{3}}-t\right)_+^3 - \left(\frac{1}{\sqrt{3}}-t\right)_+^3\right]dt \\
&= \frac{f^{(4)}(\xi)}{6}\int_{-1}^{1}\left[\frac{(1-t)^4}{4} - \left(-\frac{1}{\sqrt{3}}-t\right)_+^3 - \left(\frac{1}{\sqrt{3}}-t\right)_+^3\right]dt \quad \text{by the GMVT} \\
&= \frac{f^{(4)}(\xi)}{6}\left[\int_{-1}^{1}\frac{(1-t)^4}{4}dt - \int_{-1}^{-\frac{1}{\sqrt{3}}}\left(-\frac{1}{\sqrt{3}}-t\right)^3 dt - \int_{-1}^{\frac{1}{\sqrt{3}}}\left(\frac{1}{\sqrt{3}}-t\right)^3 dt\right] \\
&= \boxed{\frac{f^{(4)}(\xi)}{135}, \quad \text{where} \quad -1 \le \xi \le 1}.
\end{aligned}$$

EXERCISE 5

(50 points) Let $Q : f \mapsto \sum_{n=0}^{3} w_n f(x_n)$ be the quadrature rule with four evenly spaced nodes. That is, the nodes $x_0, \dots, x_3$ partition the interval $[a, b]$ into three equal subintervals. We will use the symbol Q_h to denote the composite quadrature rule with subdivision size h.

(1) Compute the weights $w_0, \dots, w_3$; you can use `Maple` or some other computer algebra system (CAS). Explain how the weights are computed and, if you use CAS, attach appropriate code.

Solution: I utilized Lagrange interpolation in order to determine the weights of the quadrature rule Q and validated with MATLAB, as demonstrated in Exercise 3. First, in using Lagrange interpolation, I determined that

$$f(x) \approx \sum_{j=0}^{3} L_j(x) f(x_j), \tag{9}$$

where

$$L_j(x) = \prod_{\substack{i=0 \\ i \neq j}}^{3} \frac{x - x_i}{x_j - x_i}.$$

Then, to obtain the quadrature rule, we integrate both sides of equation (9), where our weights are given by

$$w_j = \int_{x_0}^{x_3} L_j(x)\, dx,$$

where $a = x_0$ and $b = x_3$. As a demonstration, the integral of $L_0(x)$ is

$$\begin{aligned} w_0 &= \int_{x_0}^{x_3} \frac{(x - x_1)(x - x_2)(x - x_3)}{(x_0 - x_1)(x_0 - x_2)(x_0 - x_3)}\, dx \\ &= -\frac{(x_0 - x_3)(3x_0^2 + 2x_0(x_3 - 2(x_1 + x_2)) + x_3^2 - 2x_3(x_1 + x_2) + 6x_1x_2)}{12(x_0 - x_1)(x_0 - x_2)} \\ &= \frac{x_3 - x_0}{8}, \end{aligned}$$

where $x_1 = \frac{2x_0 + x_3}{3}$ and $x_2 = \frac{2x_3 + x_0}{3}$. Thus the weight $w_0 = 1/8$, which is confirmed by my function used in Exercise 3. The rest of the weights are computed in a similar matter. The weights are then

$$w_0 = w_3 = \frac{1}{8}, \quad w_1 = w_2 = \frac{3}{8}.$$

Further, by applying the same analysis as done in Exercise (4), the error for Simpson's 3/8 rule is

$$E(f) = -\frac{(x_3 - x_0)^5}{6480} f^{(4)}(\xi).$$

Thus, the overall quadrature rule Q is given by, after making the substituting $x_3 - x_0 = 3h$, is

$$\boxed{Q = \frac{3h}{8}\big(f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)\big) - \frac{3h^5}{80} f^{(4)}(\xi)}. \tag{10}$$

(2) Derive an expression for the error of composite quadrature $E_h(f) = \int_a^b f(x)\,dx - Q_h(f)$ involving an appropriate derivative of f. present a clear exposition of the derivation.

Solution: The formula for composite Simpson's 3/8 rule Q_h, determined by brute force, is

$$Q_h = \frac{3h}{8}\left(f(x_0) + 3\sum_{i=1}^{n/3}\big[f(x_{3i-2}) + f(x_{3i-1})\big] + 2\sum_{k=1}^{n/3-1} f(x_{3k}) + f(x_n)\right),$$

where the number of nodes N must follow $N = 3m + 4$, where $m \in Z^*$; the number of subdivisions n is given by $n = N - 1$; and $h = (x_n - x_0)/n$. In order to obtain the error for composite Simpson's 3/8 rule, we take the sum all the errors for each implementation of the quadrature rule. Thus, we have

$$E_h = \sum_{i=1}^{n/3} -\frac{3h^5}{80} f^{(4)}(\eta_i), \quad x_{3i-3} < \eta_i < x_{3i}.$$

We can factor out the terms $-3h^5/80$, and by invoking the Intermediate Value Theorem, we obtain

$$\begin{aligned}
E_h(f) &= \sum_{i=1}^{n/3} -\frac{3h^5}{80} f^{(4)}(\eta_i) \\
&= -\frac{3h^5}{80}\sum_{i=1}^{n/3} f^{(4)}(\eta_i) \\
&= -\frac{3h^5}{80}\cdot\frac{n}{3}\cdot\frac{3}{n}\sum_{i=1}^{n/3} f^{(4)}(\eta_i) \\
&= -\frac{3h^5}{80}\cdot\frac{n}{3} f^{(4)}(\xi) \quad \text{by IVT} \\
&= \boxed{\frac{-h^4(x_n - x_0)}{80} f^{(4)}(\xi)}.
\end{aligned}$$

(3) Confirm your expression for E_h numerically by testing it on random polynomials of sufficiently high degree. Present code and plots.

(4) Implement an adaptive quadrature routine based on Q. Preface the code with explanation of error control.

Solution: For error control, I applied Simpson's 3/8 rule thrice, once to four nodes and twice for seven nodes. For the first application of Simpson's 3/8 rule to four nodes, we obtain equation (10). By applying the rule twice, we first apply it to the first four nodes and then again to the last four nodes, meaning that the midpoint is used twice. So, we obtain

$$\int_a^b f(x)\,dx = S_1 - \frac{(x_3-x_0)^5}{6480} f^{(4)}(\xi_1) \tag{11}$$

$$\int_a^b f(x)\,dx = S_2 - \frac{(x_3-x_0)^5}{207360} f^{(4)}(\xi_2). \tag{12}$$

By equating equations (11) and (12), assuming that $f^{(4)}(\xi_1) = f^{(4)}(\xi_2)$[3], and solving for the error, we get

$$\frac{f^{(4)}(\xi)}{6480}(b-a)^5 = -\frac{32}{31}(S_2 - S_1). \tag{13}$$

Substituting the above expression into either equation (11) or (12) gives us

$$\int_a^b f(x)\,dx = \frac{1}{31}(32S_2 - S_1). \tag{14}$$

The code for the adaptive routine is below.

```
function [Q,evals,E] = a_simp_38(f,a,b,tol)

    % Adaptive Simpson's Rule Overview
    %
    % This function implements Simpson's 3/8 rule using adaptive ...
        quadrature to
    % numerically compute integrals. The function takes a parent function
    % f, the endpoints, x0 & x3, and the desired tolerance, tol, as inputs.
    % Because equal subintervals are desired to be used, we use the
    % following expression: xi = x0 + i(x3 - x0)/3, where i = 0,1,2,3. This
    % gives us the intermediate x values.

% Code Body

```

[3]This can be done because as the subdivisions become smaller and smaller, $\xi_1 \to \xi_2$.

```
fa = f(a);        % function evaluated at left endpoint
fc = f((2*a + b)/3); % function evaluated at intermediate points
fd = f((2*b + a)/3);
fb = f(b); % function evaluated at right endpoint
evals = 4;        % there are already three function evaluations
Q = (fa + 3*fc + 3*fd + fb)*(b-a)/8; % Simpson's 3/8 rule applied once
Q = quadstep(a,b,fa,fb,Q);

    function q = quadstep(a,b,fa,fb,S1)
        h = b-a;
        e = (5*a+b)/6;
        cc = (2*a+b)/3;
        m = (b+a)/2;
        dd = (2*b+a)/3;
        g = (5*b+a)/6;
        fe = f(e);
        fcc = f(cc);
        fm = f(m);
        fdd = f(dd);
        fg = f(g);
        evals = evals + 3;
        Sl = (fa + 3*fe + 3*fcc + fm)*h/16;
        Sr = (fm + 3*fdd + 3*fg + fb)*h/16;
        S2 = Sl + Sr;
        E = 32*abs(S2 - S1)/31;
        if h*E < tol
            q = (32*S2 - S1)/31;
        else
            q = quadstep(a,m,fa,fm,Sl) + quadstep(m,b,fm,fb,Sr);
        end

    end

end
```

The plot of the log of the evaluations against the log of the tolerance is on the following page.

(6) Compute the "error curve" by plotting the number of functional evaluations against tolerance in log-log coordinates. If the points cluster around a line (which they should for a simple function like humps) do a linear fit and find the exponent p in the power law # evaluations $\sim C\ tol^{-p}$.

Solution: Here is the code for the error curve using tolerances from 10^{-1} to 10^{-10}.

```
%% Exercise 5 Part 6

x = [1e-1 1e-2 1e-3 1e-4 1e-5 1e-6 1e-7 1e-8 1e-9 1e-10];
y = [43 61 115 157 223 331 463 769 955 1591];
```

```

logx = log(x);
logy = log(y);
[p,S] = polyfit(logx,logy,1);
pp = polyval(p,logx);

figure
hold on
plot(logx,logy,'bo');
plot(logx,pp,'b-');
xlabel('$\log(tol)$','interpreter','latex');
ylabel('$\log(evals)$','interpreter','latex');
title(['Log of Function' ...
    ' Evaluations vs. Log of Tolerance'],'interpreter','latex');
legend('Data','Fit','interpreter','latex');
set(gca,'fontsize',20);
```

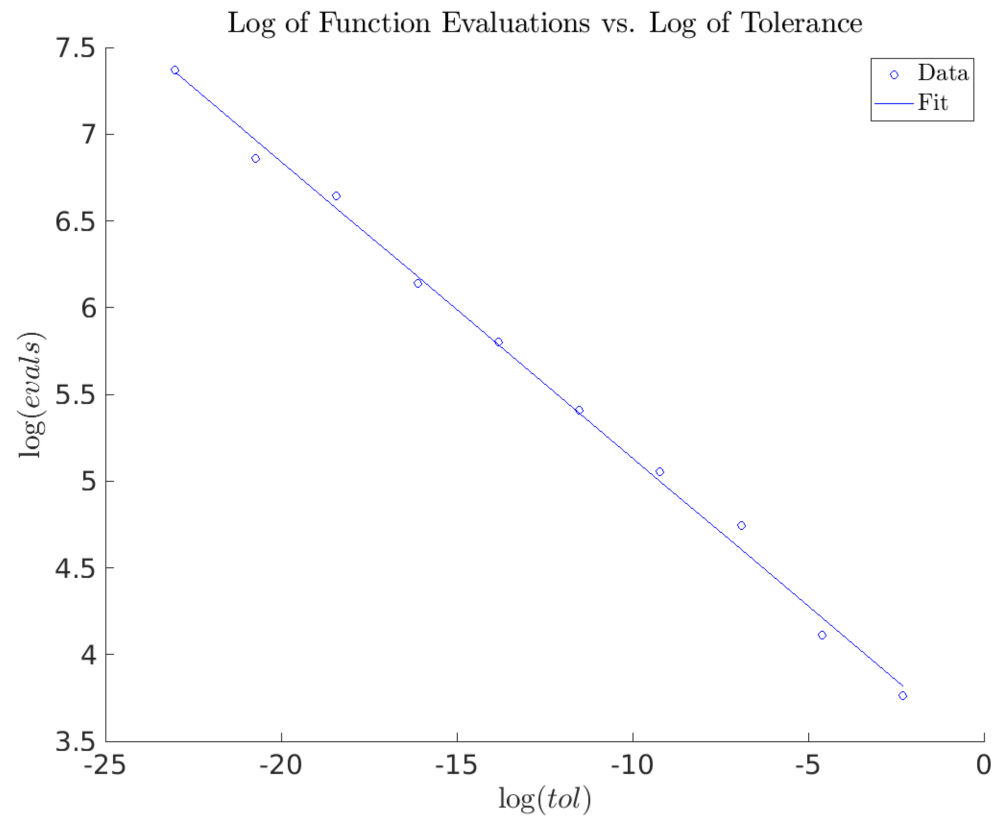


Figure 13: The log-log plot of the functional evaluations against the tolerance.

From the code, the output of the fit is $-p = -0.1708$. Thus, our power law is `# evaluations` $\sim C$ `tol`$^{-0.1708}$.

REVISIONS AND REFLECTIONS

(5.3) Confirm your expression for $E_h(f)$ numerically by testing it on random polynomials of sufficiently high degree. Present code and plots.

Solution: For this exercise, I coded my own composite Simpson's 3/8 rule, in addition to composite Simpson's 1/3 rule, and also improving upon my Trapezoid rule and Midpoint rule functions. At first, when plotting the log-log plot of the error against the step size, the only function that returned the expected value was my Trapezoid Rule function, provided below in addition to the code used to plot the log-log plots, and its respective plot.

```
function Q = trap_quad(f,a,b,n)
% TRAP_QUAD This code implements the trapezoid rule for numerical ...
    integration.
%       Q = TRAP_QUAD(f,a,b,n) computes the approximation of the integral
%       of f by the use of the trapezoid rule. The inputs are the function
%       f, the endpoints a and b, and the number of subdivisions. The
%       function uses linear interpolants for numerical integration.
%
%
%
%       Limits to the Original Method and Ways to Improve
%
%       As with the code for the midpoint rule, a "for" loop is used, which
%       restricts the number of rectangles to approximate the integral ...
    of the
%       target function. Further, there should also be an input for ...
    tolerance
%       to determine and compare the effectiveness of the trapezoid rule.
%
%       Updates
%
%       The updated code contains no "for" loops. A comparison was
%       made using y = sin(x), a = 0, b = pi, and N = 10000. For this ...
    test run,
%       the updated code was over 150 times faster than the original code.
%
%       Original Code
%
%       tic
%       x = linspace(a,b,N); % creating domain with constant step-size
%
%       Δ_x = (b-a)/N; % creating step-size
%
%       y = @(x) f(x); % function in question
%
%       A = zeros(N); % array for pre-allocation
%
```

```
%        for n = 1:N-1
%            A(n) = ((y(x(n)) + y(x(n+1)))/2).*Δ_x;
%            A = sum(A);
%        end
%        toc


% Updated Code

%tic
x = linspace(a,b,n+1);

h = x(2)-x(1); % creating step-size

y = @(x) f(x); % function in question

yy = f(x); % function

yy = (yy(1:end-1) + yy(2:end))/2;

Q = sum(yy.*h); % area of trapezoids summed together
%toc

end
```

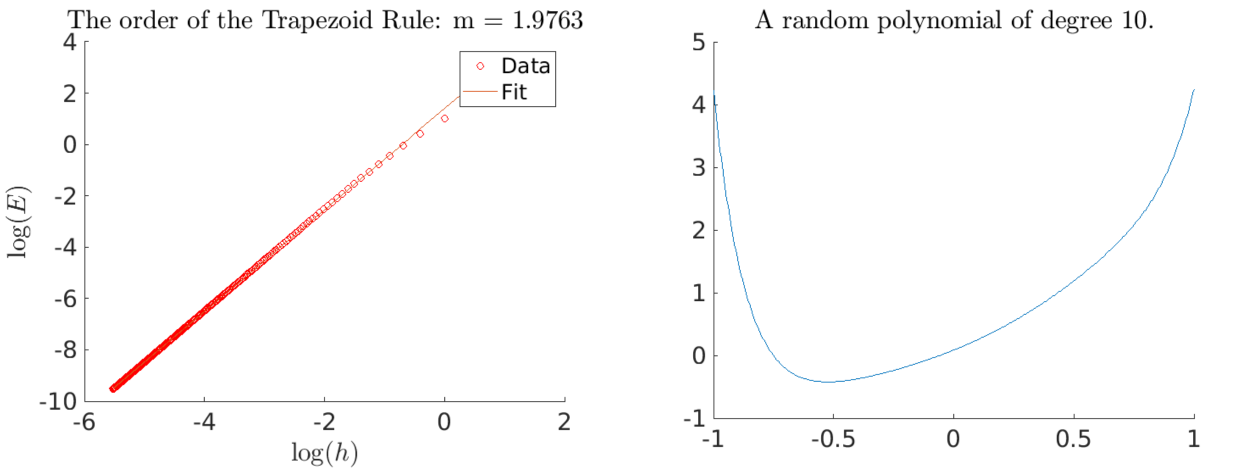


Figure 14: The left figure is the log-log plot and the right is the random tenth degree polynomial.

As expected, the slope of the log-log plot of the error against the step-size is 2, suggesting that the order of the error of the Trapezoid is $\mathcal{O}(h^2)$. However, for my Midpoint rule, Composite Simpson's rule, and Composite Simpson's 3/8 rule functions, the slope would always return an order of $\mathcal{O}(h)$, which is contrary to what we expect for the aforementioned rules. There were a few possibilities to consider why this discrepancy was present: my code determining the error and plotting the log of the error against the log of the step-size was incorrect or my functions themselves were incorrect. However, because my code determining the error of the Trapezoid rule and plotting its log against the log of the step-size returned the correct value for the slope, then it was my functions that were wrong. And thus begins the debugging. For the Midpoint rule, I noticed that what I had defined as the number of nodes was the number of subdivisions. This was a problem because I used `linspace`. The reason why this was an issue is because `linspace` takes the endpoints and the number of nodes as the inputs, not the number of subdivisions. So, in correcting this error, I updated my Midpoint rule function and it also returned the expected value for the order of the error, which is also $\mathcal{O}(h^2)$. The Midpoint rule function and the plots are presented below.

```
function Q = mid_quad(f,a,b,n)
% MID_QUAD This function implements the midpoint rule to compute integrals
% numerically.
%       A = MID_QUAD(f,a,b,n) computes the approximation of the ...
    integral of
%       f by using the midpoint rule. The inputs to the function are the
%       function handle f, the endpoints a and b, and the number of
%       subdivisions n. The function implements quadrature using piecewise
%       constant functions for numerical integration.
%
%       Limits to the Method Below and Ways to Improve
%
%       The "Original Code" below uses a "for" loop, which restricts ...
    the number of
%       subdivisions that can be used to approximate the integral, even
%       though the function is only several lines of code. Improvement ...
    can be
%       achieved, I think with the use of more linear algebra to implement
%       the use of matrix multiplication, which is well suited for MATLAB.
%       Other ways to improve this code is to implement an input for
%       tolerance, so as to quantify how accurate the code is and so it can
%       be compared with other quadrature methods.
%
%       Updates
%
%       The code was re-written without using "for" loops. A comparison was
%       made using y = sin(x), a = 0, b = pi, and N = 10000. For this ...
    test run,
%       the updated code was over 100 times faster than the original code.
%
%       Original Code
%
%       tic
%       x = linspace(a,b,n); % forming the domain of integration
%
%       Δx = (b-a)/n; % creating the step-size
```

```
%
%        y = @(x) f(x); % the function in question
%
%        Q = zeros(N); % array for size pre-allocation
%
%        for n = 1:N-1
%            Q(n) = y((x(n) + x(n+1))/2).*Δx; % midpoint rule
%            Q = sum(Q); % computing the riemann sum
%        end
%        toc


% Updated Code

%tic
x = linspace(a,b,n+1); % forming domain of integration

h = x(2)-x(1); % step-size

y = @(x) f(x); % defining the function

xx = (x(1:end-1) + x(2:end))/2; % array of midpoints

yy = y(xx); % function evaluated at midpoints

Q = sum(yy.*h); % integral
%toc

end
```

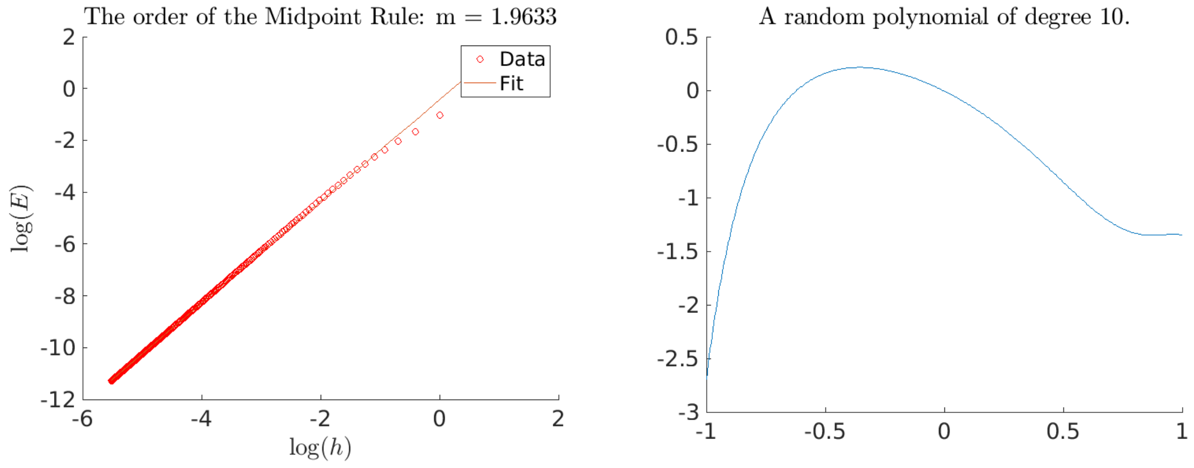


Figure 15: The left figure is the log-log plot and the right is the random tenth degree polynomial.

For my Composite Simpson's 1/3 and 3/8 rules, the issue was much more subtle, which required some brute force. At first, I experimented with my Simpson's 1/3 rule function. I was convinced that what I used for my step-size was incorrect, but it was how I determined the function values, which was incorrect. The following formula was used as a basis for my Simpson's 1/3 rule code,

$$Q_h = \frac{3h}{8}\left(f(x_0) + 3\sum_{i=1}^{n/3}\left[f(x_{3i-2}) + f(x_{3i-1})\right] + 2\sum_{k=1}^{n/3-1} f(x_{3k}) + f(x_n)\right). \tag{15}$$

Equation (15) includes nodes with values ranging from $i = 0, ..., n$. Thus, there are $n+1$ nodes present in our formula. However, in MATLAB, the index begins not at 0, like in Python or other programs, but at 1. When I figured this out that I needed to increment the indices by 1, my code finally worked and I was able to prove that for both Composite Simpson's 1/3 and 3/8 rules that the order of both rules are $\mathcal{O}(h^4)$. Here are the respective codes and plots for the 1/3 and 3/8 rules.

```
function Q = comp_simp(f,a,b,n)
% COMP_SIMP Composite Simpson's 1/3 rule for numerical integration.
%       Q = COMP_SIMP(f,a,b,n) computes an approximation of the ...
    integral of
%       f, given endpoints a and b, with n subdivisions, which must be a
%       multiple of 2. Simspson's 1/3 rule utilizes quadratic interpolants
%       for numerical integration.
%
%       Warning messages result if the number of subdivisions is not a
%       multiple of 2.
%
%       Example: approximating the integral of x.^3 on the interval [0,1]
%       with 8 subintervals
%
%          % Approximate the integral of x.^3 given the data:
%
%            f = @(x) x.^3;
%            a = 0;
%            b = 1;
%            Q = comp_simp(f,a,b,n);
%
%       Class support for inputs f,a,b,n:
%
%          function_handle
%          float: double, single


if rem(n,2) ≠ 0
    error('The value for the number of subintervals n should be even.')
end

h = (b-a)/n;
```

```
x = a:h:b;
y1 = zeros(size(x));
y2 = y1;

for j = 1:((n/2)-1)
    y1(j) = 2*f(x(2*j+1));
end

for k = 1:(n/2)
    y2(k) = 4*f(x(2*k));
end

Q = h*(f(a) + f(b) + sum(y1) + sum(y2))/3;

end
```

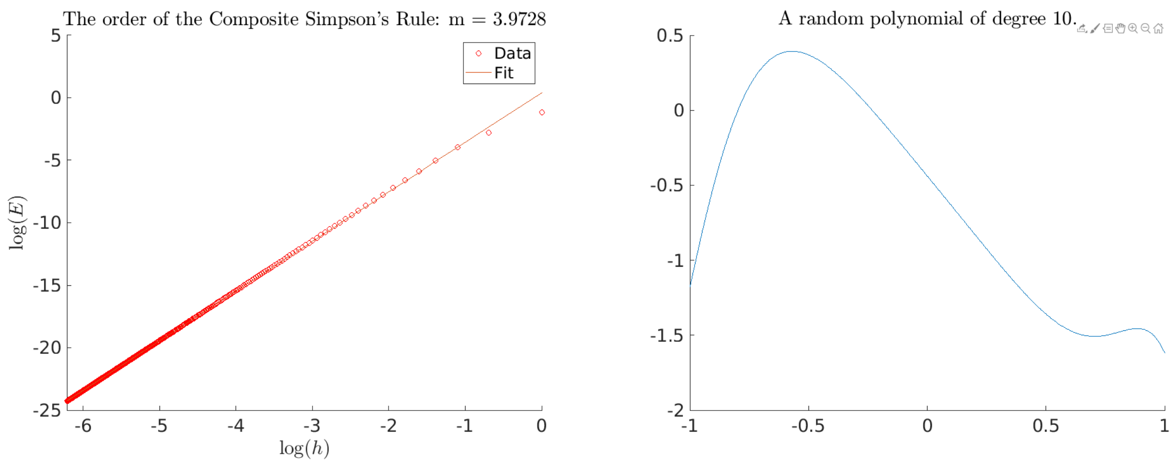
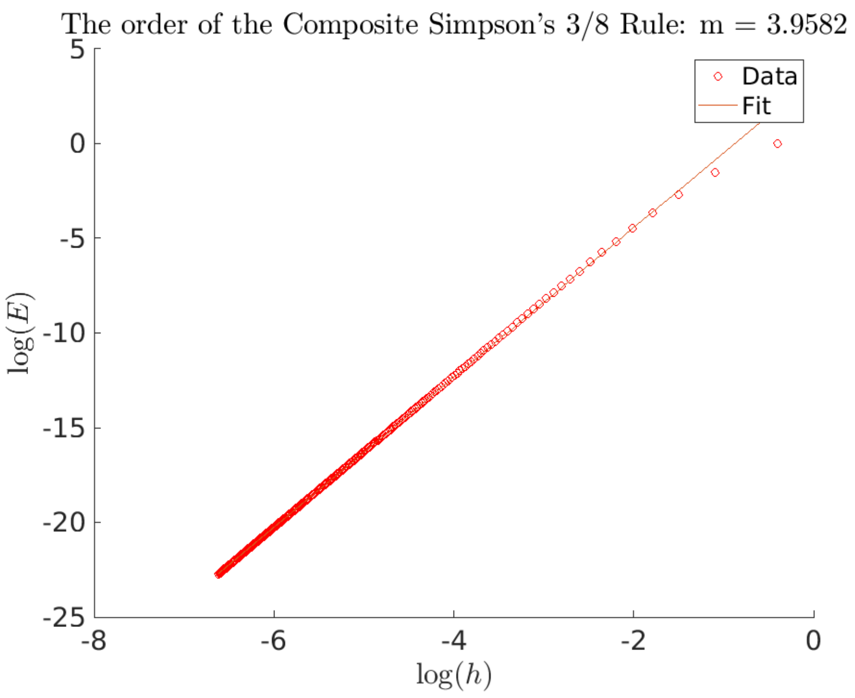


Figure 16: The left figure is the log-log plot and the right is the random tenth degree polynomial.

```
function Q = comp_simp38(f,a,b,n)
% COMP_SIMP38 Composite Simpson's 3/8 rule for numerical integration.
%       Q = COMP_SIMP38(f,a,b,n) computes an approximation of the ...
    integral of
%       f, given endpoints a and b, with n subdivisions, which must be a
%       multiple of 3. Simspson's 3/8 rule utilizes cubic interpolants
%       for numerical integration.
%
%       Warning messages result if the number of subdivisions is not a
%       multiple of 3.
%
%       Example: approximating the integral of x.^3 on the interval [0,1]
%       with 8 subintervals
%
%          % Approximate the integral of x.^3 given the data:
%
%            f = @(x) x.^3;
%            a = 0;
%            b = 1;
%            Q = comp_simp38(f,a,b,n);
%
%       Class support for inputs f,a,b,n:
%
%          function_handle
%          float: double, single

if rem(n,3)≠0
   error('The value n should be a multiple of 3.');
end

h = (b-a)/n;
x = a:h:b;
y1 = zeros(size(x));
y2 = y1;
y3 = y2;

for j = 1:(n/3)
    y1(j) = 3*f(x(3*j-1));
    y2(j) = 3*f(x(3*j));
end

for k = 1:(n-3)/3
    y3(k) = 2*f(x(3*k+1));
end

Q = (f(a) + f(b) + sum(y1) + sum(y2) + sum(y3))*3*h/8;

end
```



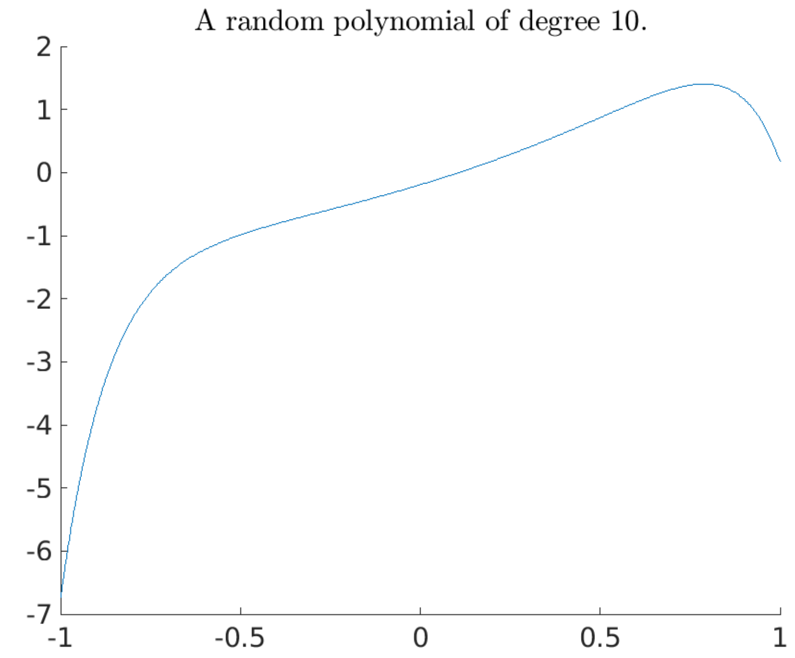


Figure 17: The left figure is the log-log plot and the right is the random tenth degree polynomial.

Thus, the data is consistent with the derivation in 5.2. Here is the code for Exercise 5 part 3.

```matlab
%% Exercise 5 Part 3

n = 500;
a = -1;
b = 1;
E = zeros(size(1:n));

c = randn(1,11);
f = @(x) polyval(c,x);
I = integral(f,a,b);
hh = zeros(size(1:n));
x = linspace(-1,1,n);

for k = 1:n
    h = (b-a)/(k); % multiply k by 2 for comp_simp and by 3 for comp_simp38
    %Q = comp_simp38(f,a,b,3*k);
    %Q = trap_quad(f,a,b,k);
    %Q = mid_quad(f,a,b,k);
    %Q = comp_simp(f,a,b,2*k);
    E(k) = abs(I-Q);
    hh(k) = h;
end

u = log(hh);
v = log(E);
p = polyfit(u,v,1);
txt1 = 'Midpoint Rule';
txt2 = 'Trapezoid Rule';
txt3 = 'Composite Simpson''s Rule';
txt4 = 'Composite Simpson''s 3/8 Rule';

figure
hold on
plot(u,v,'ro');
plot(u,polyval(p,u));
xlabel('$\log(h)$','interpreter','latex');
ylabel('$\log(E)$','interpreter','latex');
legend('Data','Fit');
title(sprintf('The order of the %s: m = ...
    %1.4f',txt4,p(1)),'interpreter','latex');
set(gca,'fontsize',20);

figure
hold on
plot(x,f(x))
title('A random polynomial of degree 10.','interpreter','latex');
set(gca,'fontsize',20);
```

(5.5) Test your adaptive quadrature on 5000 random polynomials of degree 10. Compute the failure rate and compare it with that of quad.

solution: I performed two different tests: the first test includes varying the tolerances from $1e-3$ to $2e-7$ and the second test includes fixing the tolerance at $1e-15$. The results are below with the code.

```
%% Exercise 5 part 5

n = 5000;
a = -1;
b = 1;

fail_asimp38 = 0;
fail_quad = 0;

for j = 1:n
    tol = 1/(1000*n);
    %tol = 1e-15;
    c = randn(1,11);
    I = polyval(polyint(c),b) - polyval(polyint(c),a);
    f = @(x) polyval(c,x);
    Q1 = a_simp38(f,a,b,tol);
    Q2 = quad(f,a,b,tol);
    E1 = abs(I-Q1) > tol;
    E2 = abs(I-Q2) > tol;
    if E1 > tol
        fail_asimp38 = fail_asimp38 + 1;
    end
    if E2 > tol
        fail_quad = fail_quad + 1;
    end
end

rate1 = fail_asimp38/5000;
rate2 = fail_quad/5000;

% Varying Tolerance

>> rate1

rate1 =

    0.3450

>> rate2

rate2 =

    0.0346

% Fixed Tolerance

>> rate1

rate1 =

    0.9982

>> rate2

rate2 =

    0.0034
```

Considering the varying tolerances, my adaptive routine fails ten times more than quad's routine. Considering the fixed tolerance, my adaptive routine failed almost one hundred percent of the time, or about three hundred times more fails than quad.

HOMEWORK 3

EXERCISE 1

(1) Derive the formula for the closed Newton-Cotes rule with four nodes and explain why it is called the Simpson's 3/8 rule. Use CAS if you can.

Solution: The derivation was included in the first exam. Here are the results:

$$Q = \frac{(x_3 - x_0)}{8}\big(f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)\big) - \frac{(x_3 - x_0)^5}{6480} f^{(4)}(\xi).$$

By making the substitution $3h = x_3 - x_0$, we obtain

$$\boxed{Q = \frac{3h}{8}\big(f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)\big) - \frac{3h^5}{80} f^{(4)}(\xi)}.$$

So, the rule is referred to as Simpson's 3/8 rule because of the factor in front due to the aforementioned substitution.

EXERCISE 2

(2) Use Peano Kernel Theorem to find the error term for the Simpson's Rule. Be sure to provide a clean derivation of the Peano kernel (it will be easier if you use CAS).

Solution: To begin the derivation, we must determine the highest order polynomial that Simpson's rule terminates. This can be done with the following error functional,

$$E(f) = \int_0^1 f(x)\, dx - \frac{1}{6}\big(f(x_0) + 4f(x_1) + f(x_2)\big). \tag{16}$$

Ideally, our error functional should return zero up to some degree. The greatest degree monomial (and hence polynomial) that is annihilated is the third degree, as demonstrated below.

$$\begin{aligned} E(x^3) &= \int_0^1 x^3\, dx - \frac{1}{6}\big(f(x_0) + 4f(x_1) + f(x_2)\big) \\ &= \frac{1}{4} - \frac{1}{6}\left(0 + \frac{1}{2} + 1\right) = 0. \end{aligned}$$

So, our Peano Kernel is given by

$$K(t) = \frac{1}{6} E_x\big((x - t)_+^3\big), \tag{17}$$

and

$$E(f) = \frac{1}{6}\int_{x_0}^{x_2} K(t) f^{(4)}(t)\ dt. \tag{18}$$

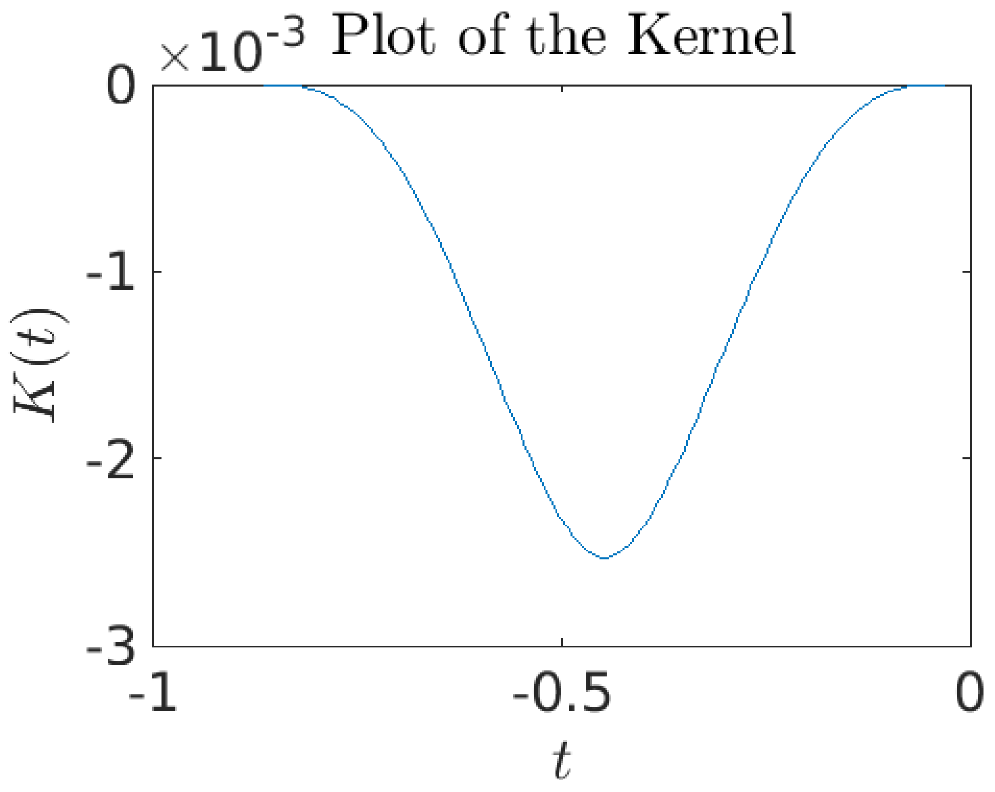


Figure 18: The plot of the kernel with random endpoints.

Now, we apply Peano's Kernel Theorem to equation (17), of course with the bounds of integration being x_0, x_2. In doing so, we have

$$\begin{aligned}
K(t) &= \int_{x_0}^{x_2} (x-t)_+^3\ dt - \frac{(x_2-x_0)}{6}\left((x_0-t)_+^3 + 4\left(\frac{x_0+x_2}{2} - t\right)_+^3 + (x_2-t)_+^3\right) \\
&= \int_{t}^{x_2} (x-t)^3\ dt - \frac{(x_2-x_0)}{6}\left(4\left(\frac{x_0+x_2}{2} - t\right)_+^3 + (x_2-t)_+^3\right) \\
&= \frac{(x_2-t)^4}{4} - \frac{(x_2-x_0)}{6}\left(4\left(\frac{x_0+x_2}{2} - t\right)_+^3 + (x_2-t)_+^3\right).
\end{aligned}$$

In the second step, the truncated polynomial $(x_0-t)_+^3$ vanishes by definition of the truncated polynomial. To express the Kernel in a nicer form, we distribute $(x_2-x_0)/6$ and we use the definition of the truncated polynomial for the remaining terms. Tus, we have

$$K(t) = \begin{cases} \frac{(a-t)^3(a+2b-3t)}{12}, & a \le t \le \frac{a+b}{2} \\ \frac{(b-t)^3(2a+b-3t)}{12} & \frac{a+b}{2} \le t \le b. \end{cases}$$

Because the Kernel does not change sign, as per Figure 18, we can rewrite equation (3) as

$$E(f) = \frac{f^{(4)}(\xi)}{6} \int_{x_0}^{x_2} K(t)\, dt.$$

Integrating the kernel gives us

$$\boxed{E(f) = -\frac{(x_2 - x_0)^5}{2880} f^{(4)}(\xi) = -\frac{h^5}{90} f^{(4)}(\xi), \quad x_2 - x_0 = 2h.}$$

EXERCISE 3

(3) Consider the following quadrature rule:

$$\int_{-1}^{+1} f(x)\, dx \approx f\Big(-\frac{1}{\sqrt{3}}\Big) + f\Big(\frac{1}{\sqrt{3}}\Big).$$

Find the order of accuracy and the error term.

This exercise was completed on the previous exam and by applying the above analysis to Gaussian quadrature provides the error term

$$\boxed{E(f) = \frac{f^{(4)}(\xi)}{135}, \quad -1 < \xi < 1.}$$

The order of accuracy is 3 because Gaussian quadrature annihilates monomials up to the third degree.

EXERCISE 4

(4) Let

$$D(f) = \frac{f(b) - f(a)}{b - a}$$

be an approximation to $f'(b)$. Find the error of this approximation.

Solution: To begin, we interpolate f with two nodes a, b using Lagrange interpolation and the corresponding error. So, we have

$$f(x) = f(a)\frac{(x-b)}{(a-b)} + f(b)\frac{(x-a)}{(b-a)} + \frac{f''(\xi)}{2}(x-a)(x-b),$$

where if we replace $a = b - h$, we have

$$f(x) = -f(b-h)\frac{(x-b)}{h} + f(b)\frac{(x-b+h)}{h} + \frac{f''(\xi)}{2}(x-b+h)(x-b). \tag{19}$$

Now, we take the derivative of equation (19) to obtain

$$f'(x) = \frac{f(b)-f(b-h)}{h} + \frac{(2x-2b+h)}{2}f''(\xi(x)) + \frac{(x-b)(x-b+h)}{2}\frac{d}{dx}[f''(\xi(x))] \tag{20}$$

By evaluating equation (20) at $x = b$, we obtain the approximation of the derivative with its error:

$$\boxed{f'(b) = \frac{f(b)-f(b-h)}{h} + \frac{h}{2}f''(\xi) = \frac{f(b)-f(a)}{b-a} + \frac{(b-a)}{2}f''(\xi).}$$

EXERCISE 5

(5) Repeat the previous exercise but regard D as the approximation to $f'(c)$ where $c = \frac{a+b}{2}$.

Solution: We begin in the same manner as the last exercise using three nodes for our Lagrange interpolation. Thus, we have

$$f(x) = f(a)\frac{(x-c)(x-b)}{(a-c)(a-b)} + f(c)\frac{(x-a)(x-b)}{(c-a)(c-b)} + f(b)\frac{(x-a)(x-c)}{(b-a)(b-c)} + \frac{f(3)(\xi)}{6}(x-a)(x-c)(x-b).$$

By taking the first derivative and making the substitution $c = \frac{a+b}{2}$, we get

$$\begin{aligned} f'(x) = {} & f(a)\frac{(4x-3b-a)}{(a-b)^2} - 4f\left(\frac{a+b}{2}\right)\frac{(2x-a-b)}{(a-b)^2} + f(b)\frac{(4x-3a-b)}{(a-b)^2} \\ & + \frac{1}{2}[a^2+4ab-6ax+b^2-6bx+6x^2]\frac{f(3)(\xi(x))}{6} - \frac{(a-x)(b-x)(a+b-2x)}{12}\frac{d}{dx}[f(3)(\xi(x))] \end{aligned}$$

Now, by evaluating $f'(x)$ at $x = \frac{a+b}{2}$,some terms cancel and we obtain

$$f'\left(\frac{a+b}{2}\right) = \frac{f(b)-f(a)}{b-a} - \frac{(b-a)^2}{24}f^{(3)}(\xi).$$

Because $c = \frac{a+b}{2}$ and $b - a = 2h$, we can make the following substitutions $b = c + h$ and $a = c - h$. By making these substitutions, we obtain

$$\boxed{f'(c) = \frac{f(c+h)-f(c-h)}{2h} - \frac{h^2}{6}f^{(3)}(\xi).}$$

EXERCISE 6

(6) Let $f \in C^2([a,b])$. Use interpolation to construct a rule for finding the second derivative $f''(b)$ from the function values at five equispaced nodes (CAS is recommended[4]). What is the order of accuracy of the rule? What is the error term?

Solution: To begin, we interpolate using Lagrange interpolation, as before, using five nodes. I performed this computation twice: the first by treating b as the endpoint of the interval and the second time by treating b as the midpoint of the interval. For the first computation, I used WolframAlpha and lots of crying to compute the second derivative of the following interpolation:

$$f(x) = \sum_{k=0}^{4} f(x_k) \prod_{\substack{i=0 \\ i\neq j}}^{4} \frac{x - x_i}{x_j - x_i} + \frac{f^{(5)}(\xi)}{120} \prod_{k=0}^{4} (x - x_k),$$

where $\{x_k\}_{k=0}^4 = \{a, c, d, e, b\}$. By differentiating the above function twice, making appropriate substitutions using $4h = b - a$, and by evaluating the resulting function at $x = b$, we obtain the following expression for the approximation of the second derivative:

$$\begin{aligned} f''(b) = {} & \frac{35f(b) - 104f(b-h) + 114f(b-2h) - 56f(b-3h) + 11f(b-4h)}{12h^2} \\ & + \frac{5h^3}{6} f^{(5)}(\xi) + \frac{2h^4}{5} \frac{d}{dx}[f^{(5)}(\xi)]. \end{aligned}$$

To check my work, I used Maple and obtained the exact same expression as above. So, the order of accuracy of this rule is $\mathcal{O}(h^3)$. The reason why $\mathcal{O}(h^3)$ is the order instead of $\mathcal{O}(h^4)$, if I was determining error, I would use the greatest value of the error so that the error is not underestimated. Then, in a separate computation, I wanted to treat b as the midpoint. I performed this process again and what I obtained is

$$f''(b) = \frac{-f(b-2h) + 16f(b-h) - 30f(b) + 16f(b+h) - f(b+2h)}{12h^2} + \frac{h^4}{15} \frac{d}{dx}[f^{(5)}(\xi)].$$

This rule is the five point central difference approximation to the second derivative, with a better order of accuracy of $\mathcal{O}(h^4)$. This is most likely due to the symmetry concerning the nodes. With respect to the error, after discussing with Justin, I was unaware that I can use Peano's Kernel Theorem concerning derivatives. It never occurred to me that I could compute the error of quadrature rules as well as derivative rules, and other rules where the functional is linear. So I utilized Peano's Kernel Theorem in the same manner as I did in exercise 2. First, I determined the highest order monomial that is annihilated using the rule with the following error functional:

[4]Oh yeah it is! Quite the opposite of "quick maths" indeed. I purchased a Maple license, which was after I did the computation.

$$\begin{aligned} E_1(f) &= D_x^2(f)\big|_{x=b} - \tilde{D}_x^2(f)\big|_{x=b} \\ &= f''(b) - \frac{35f(b) - 104f(b-h) + 114f(b-2h) - 56f(b-3h) + 11f(b-4h)}{12h^2}. \end{aligned}$$

I found that the highest degree monomial terminated by the error functional is x^4. Then, I utilized Peano's Kernel Theorem. Using Maple, concerning expression from the first derivation, the kernel that I computed is

$$\frac{1}{4!}E_1\big((x-t)_+^4\big) = K(t) = \frac{1}{4!}\Bigg[12(b-t)^2 \\ - \frac{1}{12h^2}\big(35(b-t)^4 - 104(b-h-t)_+^4 + 114(b-2h-t)_+^4 - 56(b-3h-t)_+^4\big)\Bigg].$$

As we can see the term involving the first value of the interval $b - 4h = a$ vanishes by definition of the truncated polynomial. Then, by integrating the kernel, and applying the GMVT with the assumption that the Kernel doesn't change sign, the expression for the error that I obtained was exactly the same as the *first* term present in the expression. Thus, the approximation to the second derivative evaluated at $x = b$ is

$$\boxed{f''(b) = \frac{35f(b) - 104f(b-h) + 114f(b-2h) - 56f(b-3h) + 11f(b-4h)}{12h^2} + \frac{5h^3}{6}f^{(5)}(\xi)},$$

with order of $\mathcal{O}(h^3)$ and *not* involving the other derivative. For the second derivation, the error functional I used and the Kernel I derived were

$$\begin{aligned} E_2(f) &= D_x^2(f)\big|_{x=b} - \tilde{D}_x^2(f)\big|_{x=b} \\ &= f''(b) - \frac{-f(b-2h) + 16f(b-h) - 30f(b) + 16f(b+h) - f(b+2h)}{12h^2} \end{aligned}$$

and

$$\frac{1}{5!}E_2\big((x-t)_+^5\big) = K(t) = \frac{1}{5!}\Bigg[20(b-t)_+^3 \\ - \frac{1}{12h^2}\big(16(b-h-t)_+^5 - 30(b-t)_+^5 + 16(b+h-t)_+^5 - (b+2h-t)^5\big)\Bigg],$$

respectively, because the highest degree monomial that is terminated is x^5. Thus, the overall expression of the second derivative evaluated at $x = b$ is

$$\boxed{f''(b) = \frac{-f(b-2h) + 16f(b-h) - 30f(b) + 16f(b+h) - f(b+2h)}{12h^2} + \frac{h^4}{90}f^{(6)}(\xi)}.$$

The highest order derivative in each case is exactly what was expected. To confirm my expressions, I used the error functional on the monomial with a degree of one more than the monomial that was annihilated. For the first case, x^4 was annihilated. So E_1 applied to $f(x) = x^5$ returns an error of $100h^2$. By looking at the error term $5h^3 f^{(5)}(\xi)/6$, the fifth derivative of f is a constant, so we obtain exactly $100h^2$ as expected. For the second case, the E_2 for $f(x) = x^6$ returns $8h^4$. Again, in looking at $h^4 f^{(6)}(\xi)/90$, differentiating the function six times and simplifying, we obtain $8h^2$, thus verifying the derivation for the error terms.

I think that it's pretty cool that symmetric nodes, at least for this case, improve the accuracy of the method. I'm not sure if this is generally true, but it is an interesting consequence to observe.

EXERCISE 7

(7) Derive the error term for composite Simpson's rule. Illustrate it with figures similar to 6 and 7.

Solution: The composite Simpson's rule is given by

$$Q_h = \frac{h}{3}\left(f(x_0) + 2\sum_{k=1}^{n/2-1} f(x_{2k}) + 4\sum_{k=1}^{n/2} f(x_{2k-1}) + f(x_n)\right),$$

where the number of nodes N must follow $N = 2m + 3$, where $m \in Z^*$; the number of subdivisions n is given by $n = N - 1$; and $h = (x_n - x_0)/n$. In order to obtain the error for composite Simpson's 1/3 rule, we take the sum of all the errors for each implementation of the quadrature rule. Thus, we have

$$E_h = \sum_{i=1}^{n/2} -\frac{h^5}{90} f^{(4)}(\eta_i), \quad x_{2i-2} < \eta_i < x_{2i}.$$

We can factor out the terms $-h^5/90$, and by invoking the Intermediate Value Theorem, we obtain

$$\begin{aligned}
E_h(f) &= \sum_{i=1}^{n/2} -\frac{h^5}{90} f^{(4)}(\eta_i) \\
&= -\frac{h^5}{90} \sum_{i=1}^{n/2} f^{(4)}(\eta_i) \\
&= -\frac{h^5}{90} \cdot \frac{n}{2} \cdot \frac{2}{n} \sum_{i=1}^{n/2} f^{(4)}(\eta_i) \\
&= -\frac{h^5}{90} \cdot \frac{n}{2} f^{(4)}(\xi) \quad \text{by IVT} \\
&= \boxed{\frac{-h^4(x_n - x_0)}{180} f^{(4)}(\xi)}.
\end{aligned}$$

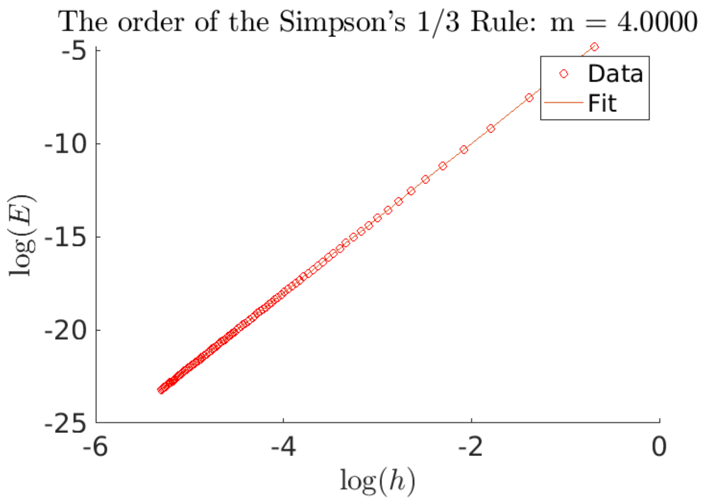
To test my derivation, we apply composite Simpson's rule to $\int_0^1 x^4 \, dx$ and plot the log of the error against the log of the step-size. In doing so, we should expect a slope of 4 and a y-intercept of $\ln\left(\frac{2}{15}\right) = -2.0149\ldots$. This can be determined because the fourth derivative of x^4 is a constant, so the absolute error of composite Simpson's rule is

$$E = \frac{2h^4}{15}.$$

By taking the natural log of both sides, we get

$$\ln E = 4 \ln h + \ln \frac{2}{15}.$$

So, I plotted the log of the error against the log of the step-size and the code is below along with the plots. What I obtained was a slope of exactly 4 and a y-intercept of $\ln\left(\frac{2}{15}\right)$. Then, I applied composite Simpson's rule to a random tenth-degree polynomial.



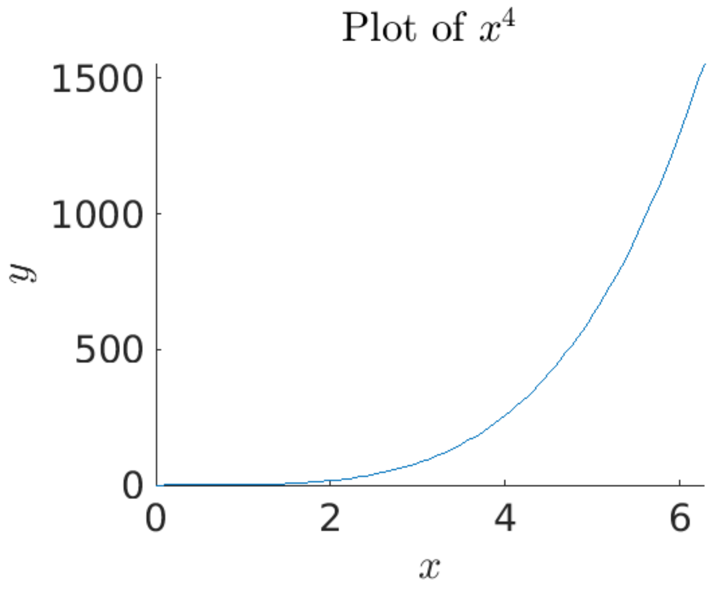


Figure 19: Plots for the order of the composite error and a plot of the function for $\int_0^1 x^4\,dx$. The slope of the line is indeed 4 and the y-intercept is $\ln\left(2/15\right)$.

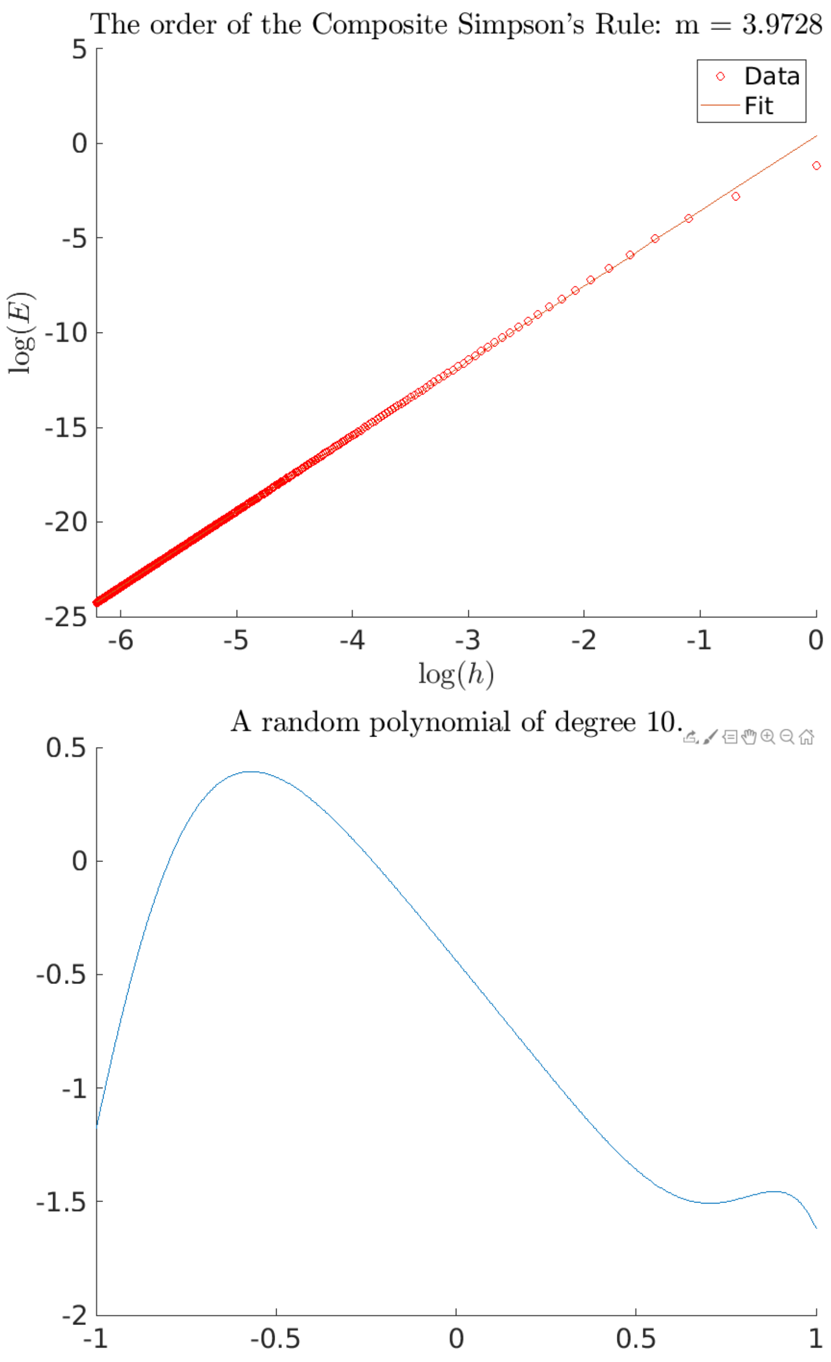


Figure 20: Plot of the order of the composite error and a plot of the random tenth degree polynomial. The slope is 3.9728.

EXERCISES 8 AND 10

(8) Confirm numerically that Equation (32) applies to

$$\int_0^{2\pi} \frac{dx}{2+|\sin(x)|} \tag{21}$$

even though the integrand is not differentiable. How would you explain that?

(10) Investigate (numerically) the validity of Equation (32) for the following integral:

$$\int_0^{2\pi} \frac{dx}{2+\sin(x)}. \tag{22}$$

How fast does the error of the composite trapezoid rule seem to decrease for smooth periodic functions?

Solution: Here is the log-log plot in addition to the plot of the function using the code from the previous exercise.

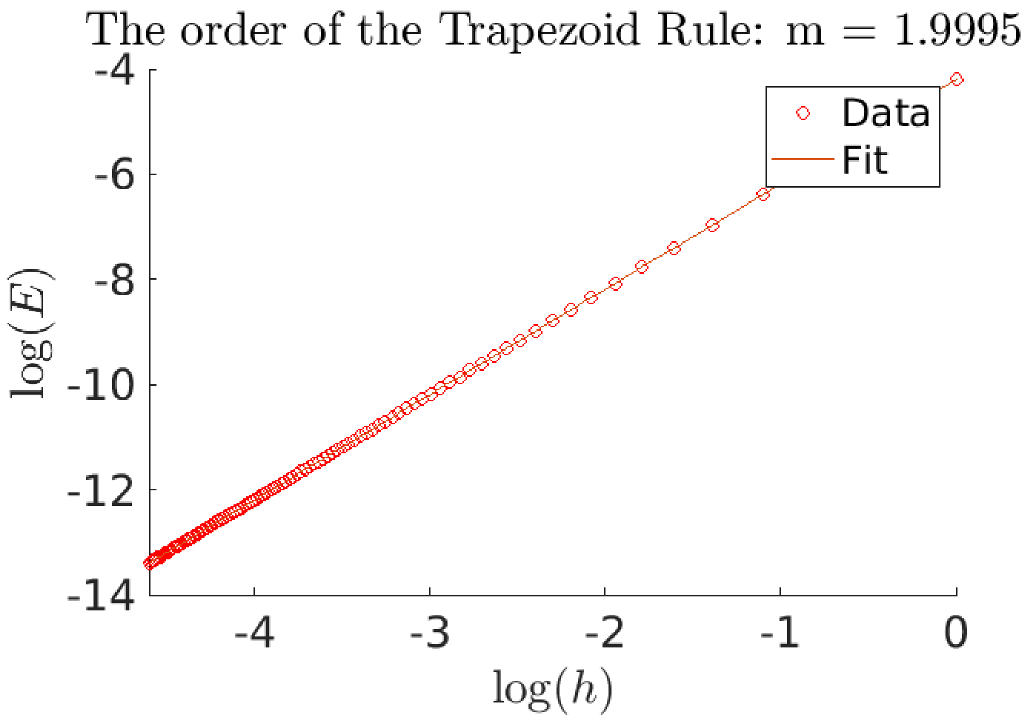


Figure 21: Plot of the order of error for the trapezoidal rule used for equation (6).

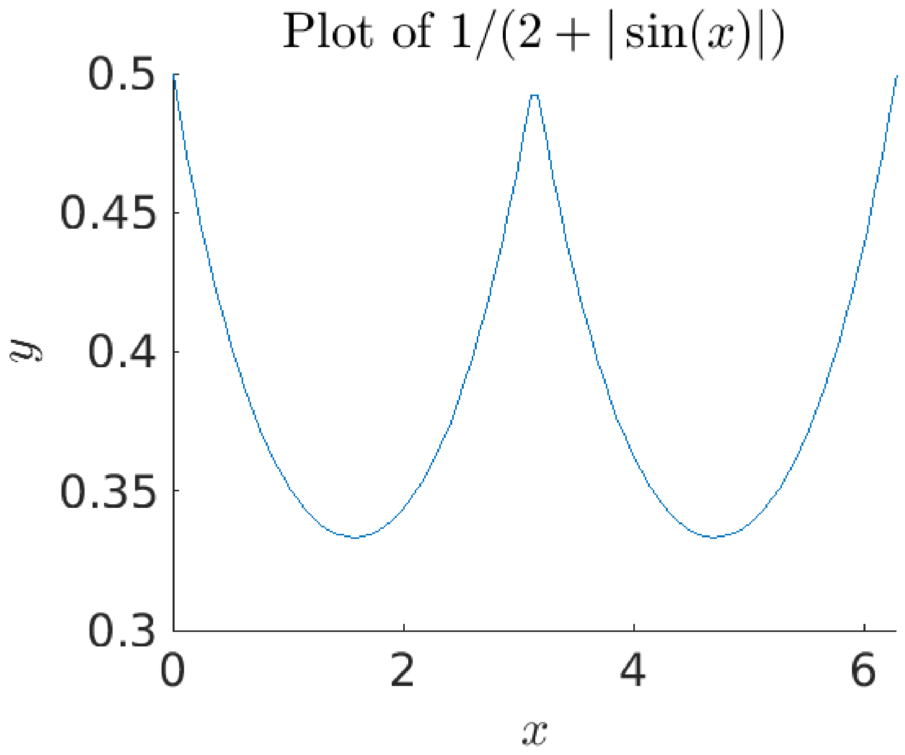


Figure 22: The plot of the function that was used to produce the previous plot.

We can clearly see that the order is $\mathcal{O}(h^2)$ for the trapezoidal rule. I think that this occurs because of the presence of the absolute value. I feel this is best discussed in the context of exercise 10. Once the absolute value is removed, the convergence is extremely fast. We can see this in the figure below.

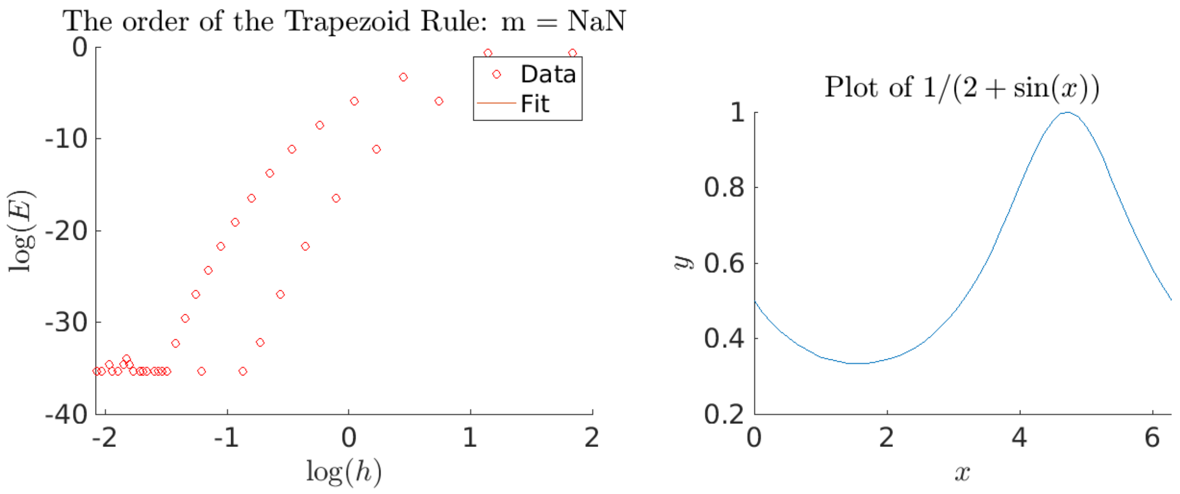


Figure 23: The log-log plot of the error against the step-size of the trapezoidal rule applied to $\int_0^{2\pi} \frac{1}{2+\sin(x)}\, dx$, and the plot of the function.

The above plots were generated with only 50 subdivisions, even though the log-log plot shows the clustering around `eps('double')` at between 25 to 30 nodes. This conveys that the trapezoidal rule converges extremely fast for smooth periodic functions. In re-deriving the trapezoidal rule for a smooth periodic functions, I found that

$$T_n = \frac{2\pi}{n}\left(f(0) + f(2\pi) + 2\sum_{i=1}^{n-1} f\left(\frac{2\pi i}{n}\right)\right).$$

As we can see, the function is evaluated at points on the unit circle, which, I'm sure, improves the computation and the rate of convergence. After doing some research, convergence is actually exponential (and involves a scary formula called the Euler-Maclaurin formula). The error for the trapezoidal rule is $\mathcal{O}(e^{-an})$, where a depends on the periodic function and n is the number of subintervals and naturally involves the step-size h. Equation (17) involves a function that's periodic and not differentiable. So, in light of the discussion that was had this past Thursday, I wanted to see if different functions with or without these properties obeyed Equation (32). I tried some functions that were only non-differentiable, functions that were only differentiable, functions that were only non-periodic, functions that were only periodic, and permutations of these. Also, the quadrature was computed over an interval including the non-differentiable part of the function. We see the results from equation (17) and equation (18) (periodic and non-differentiable & only periodic, respectively). We also know that differentiable, non-periodic functions obey Equation (32). However, interestingly enough non-periodic, non-differentiable functions are quite compelling. The types of non-differentiabilities that I observed were non-differentiability due to absolute values, fractional powers, and infinite discontinuities. Clearly, for the infinite discontinuities, like $\tan(x)$ or $\sec(x)$, bad things happen. Considering the absolute value discontinuities of the kind $f(x) = |x|^k$ and raising to any power $p \geq 1$, the trapezoidal rule obeys Equation (32). However, considering cases where $k < 1$, the order of the rule definitely depends on the value of k. This is best conveyed by expressing the functions as $f(x) = x^{1/k}$, where $k > 0$, but keeping the absolute value removes complex numbers. For example, when $k = 2$, the slope of the log-log fit of the error against the step-size returns $m \approx 1.5$. The same for these cases: $k = 3$ returns $m \approx 1.33$, $k = 4$ returns $m \approx 1.25$, $k = 5$ returns $m \approx 1.20$ and so on. I'm not entirely sure why this is the case, but it is interesting nonetheless. This occurs, though, with periodic functions as well. My plan is to code a scheme that plots the slope as a function of k to see the dependence. Until then, I'm going to finish exercise 9 and work on the next homework.

```
%% Exercise 8 and 10

n = 100;
a = 0;
b = 1;
E = zeros(size(1:n));

```

```
f = @(x) x.^4;
I = integral(f,a,b);
hh = zeros(size(1:n));
x = linspace(0,2*pi,n);

for k = 1:n
    h = (b-a)/(2*k); % multiply k by 2 for comp_simp and by 3 for ...
        comp_simp38
    Q = comp_simp(f,a,b,2*k);
    E(k) = abs(I-Q);
    hh(k) = h;
end

u = log(hh);
v = log(E);
p = polyfit(u,v,1);
txt = 'Simpson''s 1/3 Rule';

figure
hold on
plot(u,v,'ro');
plot(u,polyval(p,u));
xlabel('$\log(h)$','interpreter','latex');
ylabel('$\log(E)$','interpreter','latex');
legend('Data','Fit');
title(sprintf('The order of the %s: m = ...
    %1.4f',txt,p(1)),'interpreter','latex');
set(gca,'fontsize',20);

figure
hold on
plot(x,f(x));
title('Plot of $x^2$','interpreter','latex');
xlabel('$x$','interpreter','latex');
ylabel('$y$','interpreter','latex');
set(gca,'fontsize',20);
```

EXERCISE 9

(9) Perform several numerical experiments where you compare the accuracy of the composite trapezoid and composite midpoint rules. Is it fair to say that the midpoint rule tends to be twice as accurate as trapezoid?

Solution: This is the code that was used to run the numerical experiments.

```
%% Exercise 9

n = 10000;
a = 0;
b = 1;

fail_trap = 0;
fail_mid = 0;
k = abs(randn);

tic
for j = 1:n
    tol = 10^(-5*k);
    c = randn(1,6);
    I = polyval(polyint(c),b) - polyval(polyint(c),a);
    f = @(x) polyval(c,x);
    Q1 = comp_trap(f,a,b,j);
    Q2 = comp_mid(f,a,b,j);
    E1 = abs(I-Q1);
    E2 = abs(I-Q2);
    if E1 > tol
        fail_trap = fail_trap + 1;
    end
    if E2 > tol
        fail_mid = fail_mid + 1;
    end
end
toc

rate1 = fail_trap/n;
rate2 = fail_mid/n;
ratio = rate1/rate2;
```

The numerical experiment consisted of seeing how many times each method failed to reach a certain tolerance. Then, I took a ratio to see how many more times midpoint rule was more accurate than trapezoidal rule. I tested them on random polynomials of degree 5 with random tolerances. Ultimately, it would be accurate to say the midpoint rule is between 1.5-2 times as accurate.

HOMEWORK 4

EXERCISE 1

(1) Think of vectors x and y in $\mathbb{R}^2$ as, simply, points in the plane. Given a norm $\|\cdot\|$ on $\mathbb{R}^2$, we can define distance from x to y as $\|x-y\|$. A circle of radius r centered at the origin can ten be defined as a set of points equidistant from the origin, which is to say all vectors of norm one. Plot the unit circles corresponding to the p-norms with $p = 1, 2, 4, \infty$.

Solution: I wanted to create a function in MATLAB that would plot the unit circle for any p-norm. To begin, we start with $x \in \mathbb{R}^2$. By definition, in R^2, the definition of the 1-sphere is $S^1 = \{x \in \mathbb{R}^2 : \|x\| = 1\}$. So, in order that we obtain unit spheres for a given p-norm, we must satisfy the condition that $\|x\|_p = 1$. Let's take a set of vectors $s = \{x_1,\ x_2, \ldots, x_n\}$, where $x_i \in \mathbb{R}^2$ with components $a_i \in \mathbb{R}$. Further, let's scale each one by its respective p-norm $\|x_i\|_p$. In doing so, we obtain the new set of scaled vectors $\tilde{s} = \{u_1,\ u_2, \ldots, u_n\}$, where $u_i = \frac{x_i}{\|x_i\|_p}$ with scalars $b_i \in \mathbb{R}$. For the set $\tilde{s}$, if we plot the set of all y components against the set of all x components, we will obtain the unit sphere for any p-norm. For any vector u_i, we can check to see if it's p-norm returns a value of 1.

$$\|u_i\|_p = \left(\sum_{k=1}^{n} |b_k|^p\right)^{1/p} \tag{23}$$

$$= \left(\left|\frac{a_1}{(\sum_{k=1}^n |a_k|^p)^{1/p}}\right|^p + \left|\frac{a_2}{(\sum_{k=1}^n |a_k|^p)^{1/p}}\right|^p + \cdots + \left|\frac{a_n}{(\sum_{k=1}^n |a_k|^p)^{1/p}}\right|^p\right)^{1/p} \tag{24}$$

$$= \left(\frac{1}{|(\sum_{k=1}^n |a_k|^p)^{1/p}|^p}\left(|a_1|^p + |a_2|^p + \cdots + |a_n|^p\right)\right)^{1/p} \tag{25}$$

$$= \left(\frac{\sum_{k=1}^n |a_k|^p}{\sum_{k=1}^n |a_k|^p}\right) = 1. \tag{26}$$

There are a few things to note. We can firstly rewrite equation (24) by placing absolute value signs around the numerator and denominator separately. Then, we can factor out those terms, leading to equation (25). Then, because the norm of a vector is always positive, we can replace the absolute value with parentheses, and the p terms simplify, thus leading to rewriting the sum of the a_i as a summation and ultimately equation (26), leading us to the desired result. This was the basis of my code, which is just a few lines long. The code that generated the plots and the plots will be below the function code. The plots make sense because as $p \to \infty$, you are scaling the vector by larger and larger values until you scale the vector by the largest component. Thus, plotting the y components versus the x components would naturally portray how a rhombus becomes a square as the p increases.

```matlab
function n = norm_plot(x,p)

if p < 1
    error("The value 'p' must be greater than or equal to 1.")
end

n = zeros(size(x));

for c = 1:length(x)
    n(:,c) = x(:,c)/norm(x(:,c),p);
end

end
```

```matlab
%% Exercise 1

n = 500;

x = randn(2,n);

plt1 = norm_plot(x,1);
plt2 = norm_plot(x,2);
plt3 = norm_plot(x,4);
plt4 = norm_plot(x,inf);
plt = {plt1 plt2 plt3 plt4};
txt1 = '1-Norm';
txt2 = '2-Norm';
txt3 = '3-Norm';
txt4 = 'Infinity-Norm';
txt = {txt1 txt2 txt3 txt4};

for k = 1:4
    subplot(2,2,k);
    plot(plt{k}(1,:),plt{k}(2,:),'b.');
    title(txt{k});
    xlim([-2 2]);
    ylim([-2 2]);
    axis equal
end
```

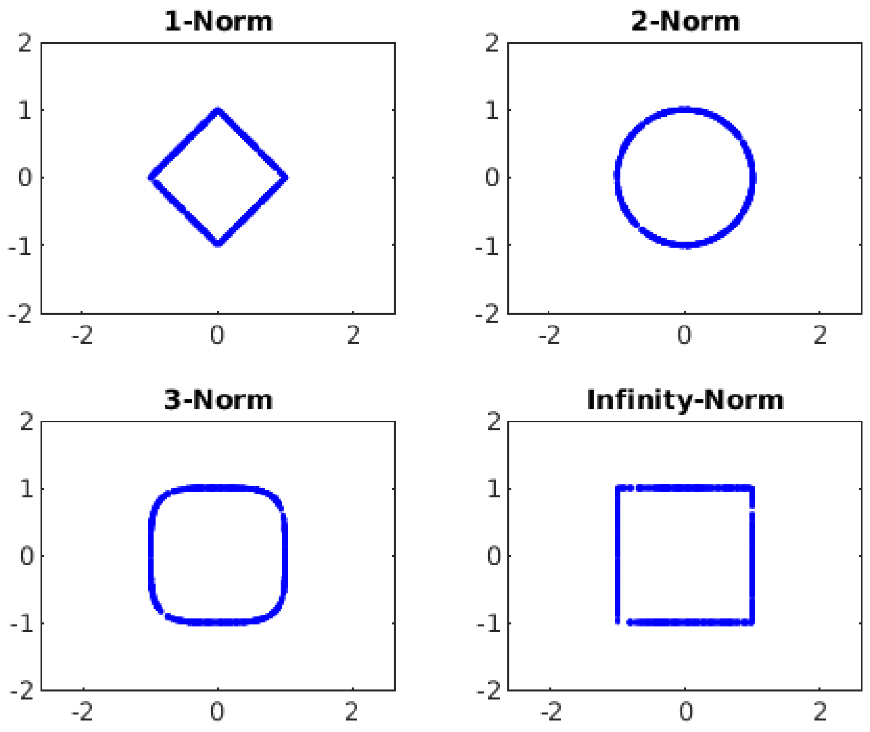


Figure 24: Plot of the p-norms for $p = 1, 2, 3, \infty$.

EXERCISE 2

(2) Define the inner product on $\mathbb{R}^2$ by the formula:

$$\langle x, y\rangle = x_1y_1 + 4x_2y_2.$$

Here we use the convention of denoting the n-th component of a vector with a subscript. Find a two-by-two matrix A such that:

$$\langle x, y\rangle = x^TAy.$$

Solution: Sorry for the lack of exposition. I'll write more soon. Here is the solution, though. Define:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

$$\langle x, y\rangle = x^TAy = ax_1y_1 + bx_1y_2 + cx_2y_1 + dx_2y_2 = x_1y_1 + 4x_2y_2$$
$$\boxed{a = 1, \quad b = c = 0, \quad d = 4}.$$

EXERCISE 3

(3) Let

$$A = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}.$$

For $x, y \in \mathbb{R}^2$, suppose we define

$$[x, y] = x^T Ay.$$

Is [x,y] an inner product on $\mathbb{R}^2$? If you answered "No," explain what breaks down. If you answered "Yes," either provide a formal argument, if you can, or generate convincing evidence in MATLAB.

Solution: The operation $[x, y]$ is an inner product.

Proof. Let us see if $[x, y]$ satisfies

1. Symmetry
2. Bilinearity
3. Positive Definiteness.

By applying the operations $[x, y]$ and $[y, x]$, we find that

$$\begin{aligned} [x, y] = x^T Ay &= (2a_1 - a_2)b_1 + (-a_1 + 2a_2)b_2 \\ &= 2a_1b_1 - a_1b_2 - a_2b_2 + 2a_2b_2 \\ &= (2b_1 - b_2)a_1 + (-b_1 + 2b_2)a_2 \\ &= y^T Ax = [y, x]. \end{aligned}$$

Clearly, the operation $[\cdot, \cdot]$ is symmetric. Next, we apply the operation to a linear combination. Thus

$$\begin{aligned} [c_1x_1 + c_2x_2, y] &= 2a_1b_1c_1 - a_1b_2c_1 - a_2b_2c_1 + 2a_2b_2c_1 + 2a_3b_1c_2 - a_3b_2c_2 - a_4b_1c_2 + 2a_4b_2c_2 \\ &= c_1(2a_1b_1 - a_1b_2 - a_2b_2 + 2a_2b_2) + c_2(2a_3b_1 - a_3b_2 - a_4b_1 + 2a_4b_2) \\ &= c_1[x_1, y] + c_2[x_2, y], \end{aligned}$$

where

$$x_1 = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}, \quad x_2 = \begin{bmatrix} a_3 \\ a_4 \end{bmatrix}, \quad y = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

Lastly, to prove positive definiteness, we check if $u^T Au \geq 0$ by completing the square. By expanding the expression, we obtain

$$\begin{aligned} u^T Au &= 2u_1^2 - 2u_1u_2 + 2u_2^2 \\ &= 2\left(u_1 - \frac{u_2}{2}\right)^2 + \frac{3u_2^2}{2} \geq 0. \end{aligned}$$

It is clear that this value is always greater than zero. The only time this expression is zero is when $u_1 - u_2/2 = 0$ where $u_1 = u_2 = 0$. Thus, A is positive. So, the operation $[\cdot, \cdot]$ is indeed an inner product. □

EXERCISE 4

(4) Consider quadratics on the interval $[0,1]$ *with the* L^2 *inner product:*

$$\langle f,g\rangle = \int_0^1 f(x)g(x)dx.$$

Find an orthogonal basis of this space. Validate your result with an orthogonal expansion of $q = x^2+x+1$.

Solution: To begin, I start with the basis $p = \{1, x, x^2\} = \{v_1, v_2, v_3\}$. Now, I perform the Gram-Schmidt Orthogonalization process to orthogonalize the vectors. So,

$$\begin{aligned} u_1 &= \frac{v_1}{\sqrt{\langle v_1, v_1\rangle}} = 1, \\ u_2 &= v_2 - \frac{\langle v_2, u_1\rangle}{\langle u_1, u_1\rangle}u_1 = x - \frac{1}{2}, \\ u_3 &= v_3 - \frac{\langle v_3, u_1\rangle}{\langle u_1, u_1\rangle}u_1 - \frac{\langle v_3, u_2\rangle}{\langle u_2, u_2\rangle}u_2 = x^2 - x + \frac{1}{6}. \end{aligned}$$

Because computing the inner product of each vector with the others in the set $\tilde{p} = \{u_1, u_2, u_3\}$ returns zero, the vectors are orthogonal. To check, we see if we can write

$$q = \frac{\langle q, u_1\rangle}{\langle u_1, u_1\rangle}u_1 + \frac{\langle q, u_2\rangle}{\langle u_2, u_2\rangle}u_2 + \frac{\langle q, u_3\rangle}{\langle u_3, u_3\rangle}u_3.$$

So,

$$\begin{aligned} \langle q, u_1\rangle &= \int_0^1 (x^2+x+1)1\ dx = \frac{11}{6}, \\ \langle q, u_2\rangle &= \int_0^1 (x^2+x+1)\left(x - \frac{1}{2}\right)\ dx = \frac{1}{6}, \\ \langle q, u_3\rangle &= \int_0^1, (x^2+x+1)\left(x^2 - x + \frac{1}{6}\right)\ dx = \frac{1}{180}, \end{aligned}$$

and

$$\begin{aligned} \langle u_1, u_1\rangle &= \int_0^1 1^2\ dx = 1, \\ \langle u_2, u_2\rangle &= \int_0^1 \left(x - \frac{1}{2}\right)^2\ dx = \frac{1}{12}, \\ \langle u_3, u_3\rangle &= \int_0^1 \left(x^2 - x + \frac{1}{6}\right)^2\ dx = \frac{1}{180}. \end{aligned}$$

Ultimately,

$$\begin{aligned}\frac{\langle q,u_1\rangle}{\langle u_1,u_1\rangle}u_1+\frac{\langle q,u_2\rangle}{\langle u_2,u_2\rangle}u_2+\frac{\langle q,u_3\rangle}{\langle u_3,u_3\rangle}u_3 &= \frac{11}{6}+\frac{(1/6)}{(1/12)}\left(x-\frac{1}{2}\right)+\frac{(1/180)}{(1/180)}\left(x^2-x+\frac{1}{6}\right)\\ &= \frac{11}{6}+2x-1+x^2-x+\frac{1}{6}\\ &= x^2+x+1=q,\end{aligned}$$

as expected.

Exercise 5

(5) Consider Lagrange interpolation on the interval $[-1,1]$. *Suppose we want to minimize the remainder by choosing nodes so as to minimize the norm of the nodal polynomial* p. *From the handout you know that minimization of the* $\infty-$ *norm of* p *leads to the conclusion that* p *must be a (monic) Chebyshev polynomial of the first kind. Show that minimization of the 2-norm leads to monic Legendre polynomials.*

Solution: To minimize the 2-norm of the nodal polynomial, I utilized the code in the handout, with some modifications, and confirmed that the monic polynomials produced from minimizing the 2-norm are Legendre polynomials. I did this using my own function, which generates monic Legendre polynomials. Below is the function and below the function is the code.

```
function [pp,r] = legendre_n(x,n)
P = legendre(n,x);
y = P(1,:);
p = polyfit(x,y,n);
r = roots(p);
pp = poly(r);
end
```

```
%% Exercise 5

N = 2;

x = linspace(-1,1,1024);

r0 = -1 + 2*(1:N)/(N+1);
r = fminsearch(@two_norm,r0);
p = poly(r);
```

```

figure
plt = plot(x,abs(polyval(p,x)), 'b-',r,0,'ro',r0,0,'bo');

[p1,r1] = legendre_n(x,N); % my function

function f = two_norm(r)
    pp = poly(r);
    f = sqrt(integral(@(x)polyval(pp,x).^2,-1,1));
end

% N = 2

p =

    1.0000    0.0000   -0.3333

p1 =

    1.0000         0   -0.3333

% N = 3

p =

    1.0000    0.0000   -0.6000   -0.0000

p1 =

    1.0000    0.0000   -0.6000    0.0000

% N = 4

p =

    1.0000   -0.0001   -0.8572    0.0000    0.0857

p1 =

    1.0000   -0.0000   -0.8571    0.0000    0.0857
```

EXERCISE 6

Which monic polynomials have the smallest 1-norm on $[-1,1]$? Compute the first three numerically and try to identify them using **`Maple`**. *50 bonus points for a rigorous proof.*

Solution: Using and modifying the above code, we obtain:

```
%% Exercise 6

r0 = -1 + 2*(1:N)/(N+1);
r = fminsearch(@one_norm,r0);
pp = poly(r);

figure
plt1 = plot(x,abs(polyval(pp,x)), 'b-',r,0,'ro',r0,0,'bo');

function f = one_norm(r)
    pp = poly(r);
    f = integral(@(x) abs(polyval(pp,x)),-1,1);
end

% N = 0

p =

     1

% N = 1

p =

     1     0

% N = 2

p =

    1.0000    0.0001   -0.2502

% N = 3

p =

    1.0000    0.0010   -0.4998   -0.0005

% N = 4

p =

    1.0000    0.0002   -0.7501   -0.0001    0.0625
```

We can see that the first five monic polynomials are

$$1,\ x,\ x^2-\frac{1}{4},\ x^3-\frac{1}{2}x,\ x^4-\frac{3}{4}x^2+\frac{1}{16}.$$

In Maple, I was able to determine that these polynomials are Jacobi polynomials with $a = b = \frac{1}{2}$. I used trial and error to determine a and b. I was able to discern that a must be equal to b.

Exercise 7

(7) Define a weighted L^2-inner product on $[-1, 1]$ as

$$\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)(1-x^2)dx.$$

Find the first ten orthogonal polynomials corresponding to that inner product. Let p_9 be the last of these—polynomial of degree nine. Find its roots and use them as nodes to interpolate

$$y = \frac{1}{1+25x^2}$$

Compare the result with interpolation of the same function on equispaced nodes. What are your observations?

Solution: Let's use the code from the handout, using the weighted inner product.

```
> restart;
> w := -x^2 + 1
> L2ip := (f, g) -> int(g*f*w, x = -1 .. 1);
> L2norm := h -> sqrt(L2ip(h, h));
> V := [seq(x^(k - 1), k = 1 .. 10)];
> W[1] := V[1]/L2norm(V[1]);
> for n from 2 to 10 do
>    z := V[n] - add(L2ip(V[n], W[j])*W[j], j = 1 .. n - 1);
>    W[n] := z/L2norm(z);
> end do
```

The output is

$$\begin{aligned}
W_1 &= \frac{\sqrt{3}}{2},\\
W_2 &= \frac{\sqrt{15}}{2}x,\\
W_3 &= \frac{5(x^2-\frac{1}{5})\sqrt{42}}{8}\\
W_4 &= \frac{21(x^3-\frac{3}{7}x)\sqrt{10}}{8}\\
W_5 &= \frac{21(x^4-\frac{2}{3}x^2+\frac{1}{21})\sqrt{165}}{16}\\
W_6 &= \frac{33(x^5-\frac{10}{11}x^3+\frac{5}{33}x)\sqrt{273}}{16}\\
W_7 &= \frac{429(x^6-\frac{15}{13}x^4+\frac{45}{143}x^2-\frac{5}{429})\sqrt{105}}{64}\\
W_8 &= \frac{2145(x^7-\frac{7}{5}x^5+\frac{7}{13}x^3-\frac{7}{143}x)\sqrt{17}}{64}\\
W_9 &= \frac{7293(x^8-\frac{28}{14}x^6+\frac{14}{17}x^4-\frac{28}{221}x^2+\frac{7}{2431})\sqrt{95}}{256}\\
W_{10} &= \frac{4199(x^9-\frac{36}{19}x^7+\frac{378}{323}x^5-\frac{84}{323}x^3+\frac{63}{4199}x)\sqrt{1155}}{256}
\end{aligned}$$

I was able to compute the roots with the MATLAB code below. Below are also the plots.

```
%% Exercise 7

x = linspace(−1,1,100);
xx = linspace(−1,1,8);

c = 4199*sqrt(1155)*[1 0 −36/19 0 378/323 0 −84/323 0 63/4199]/256;
root_s = roots(c)';

f = @(x) polyval(c,x);

y = @(x) 1./(1+25*x.^2);

p = polyfit(root_s,y(root_s),7);
pp = polyval(p,x);

p1 = polyfit(xx,y(xx),7);
pp1 = polyval(p1,x);
```

```


figure
hold on
plot(x,f(x));
plot(root_s,zeros(size(root_s)),'ro');
title('Plot of the Polynomial.')
set(gca,'fontsize',20);

figure
hold on
plot(x,y(x));
plot(x,pp,'k-');
plot(x,pp1,'r-');
legend('$\frac{1}{1+25x^{2}}$','Interpolant: Roots','Interpolants: ...
    Nodes','interpreter','latex');
set(gca,'fontsize',20);
```

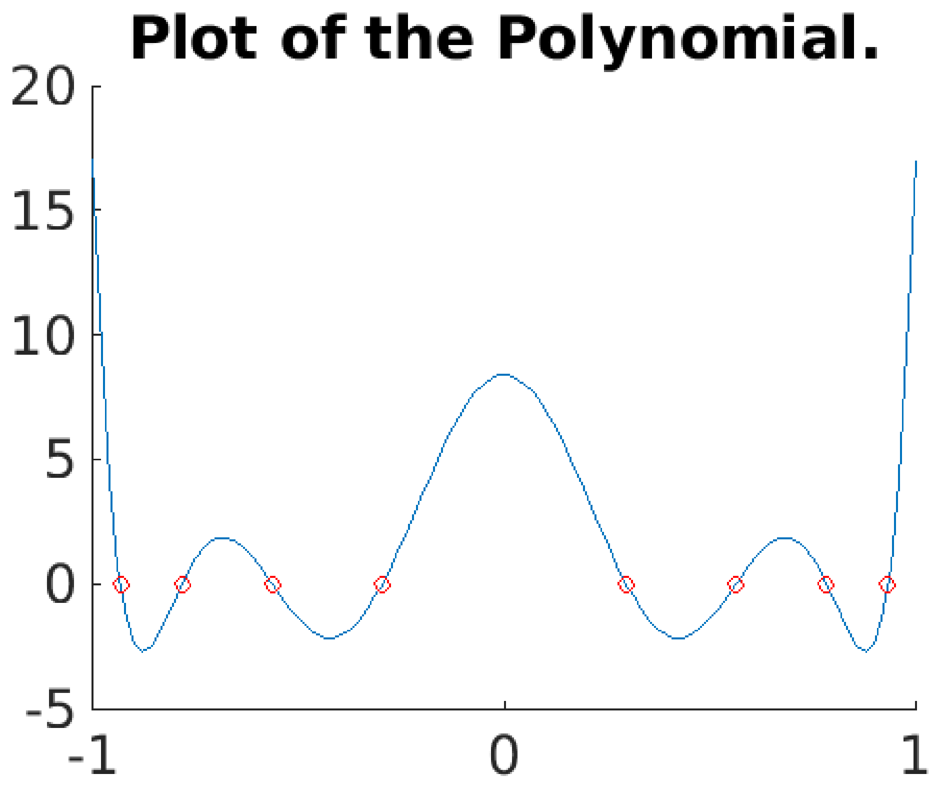


Figure 25: Plot of the ninth degree polynomial.

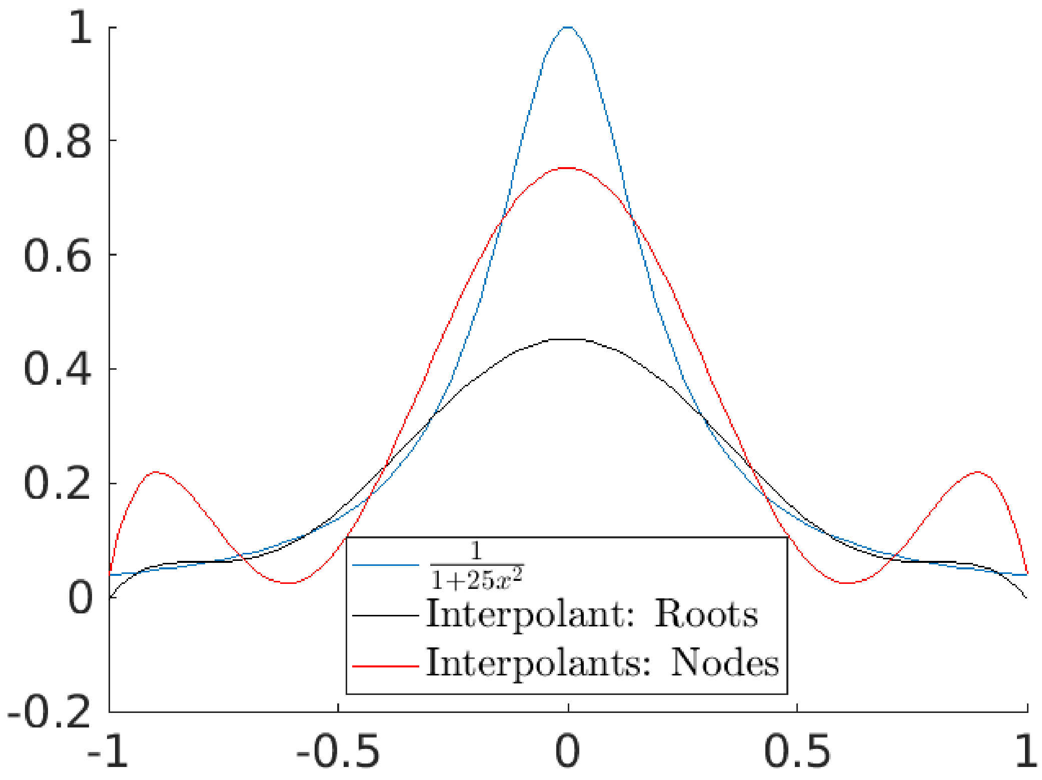


Figure 26: Plot of the interpolation with the interpolants.

We can see that the plot of the interpolant with the roots interpolates better around the endpoints, leaving little room for Runge's Phenomenon. The equispaced interpolant, on the other hand has quite a bit of Runge's Phenomenon. I suspect that for higher degree polynomials, using their roots to interpolate would provide a fantastic approximation.

HOMEWORK 5

GAUSSIAN QUADRATURE IN MATLAB

This was a fun exercise. I don't really have much to say. I will, however, begin with the definition of Gaussian Quadrature.

> **Definition:** For the following linear functional I defined on the space $R([-1,1])^5$,
>
> $$I(f) = \int_{-1}^{1} f(x)\, dx,$$
>
> *Gaussian quadrature* Q_N is a linear functional $Q_N(f) = \sum_{n=1}^{N} w_n f(x_n)$ that is exact on all monomials $x^k \in R([-1,1])$, and hence polynomials in the space, of degree $2N-1$ or less, where $N \in \mathbb{Z}^*$ and w_n and x_n denote the weights and nodes, respectively. That is,
>
> $$Q(x^k) = I(x^k).$$

To code Gaussian quadrature, I utilized the code that we used in class in addition to code I wrote for Newton's method, with some good advice from Justin and the help of Katie's notes from class. The code for the function is below along with a test for the function.

```
function Q = quad_gauss(f,n,tol)

x0 = cos(pi*(0.75 + (0:n-1))/(n + 0.5));

for k = 1:1000
    [p,dp] = legendrep(n,x0);
    x0 = x0 - (p./dp).*(x0.^2 - 1);
    if norm(p,inf) < tol
        break
    end
end
w = 2./(dp.^2).*(1 - x0.^2);

Q = w*f(x0)';
end

function [p,dp] = legendrep(n,x)

if n == 0
```

^{5}This denotes Riemann-integrable functions. Not all functions are Riemann-integrable. An example would be $f(x) = 1/x$ on the interval $[0,b]$. Other types of integrals include Lebesgue integrals, Daniell integrals, and many more in the integral zoo.

```
    p = ones(size(x));
    dp = zeros(size(x));
elseif n == 1
    p = x;
    dp = ones(size(x));
else
    p1 = x;
    p2 = ones(size(x));
    for m = 2:n
        p = ((2*m - 1)*(x.*p1) - (m-1)*p2)/m;
        p2 = p1;
        p1 = p;
    end
    dp = n.*(x.*p - p2);
end
end
```

```
%% Exercise 1

omega = 1000000;

f = @(s) omega*cos(omega*s)*0.5;
I = sin(omega);
tol = 1e-3;

Q_gauss = quad_gauss(f,1000000,tol);
err_gauss = abs(I - Q_gauss);
Q_quadgk = quadgk(f,-1,1);
err_quadgk = abs(I - Q_quadgk);
Q_integral = integral(f,-1,1);
err_integral = abs(I - Q_integral);

%% Frequency = 1,000,000

>> Q_gauss

Q_gauss =

   -0.3500

>> err_gauss

err_gauss =

   1.2023e-08

>> Q_integral

```

```
Q_integral =

  -2.2927e+03

>> err_integral

err_integral =

   2.2923e+03

>> Q_quadgk

Q_quadgk =

  -1.0551e+04

>> err_quadgk

err_quadgk =

   1.0551e+04
```

Firstly, the function `quad_gauss` takes the function f, the number of weights and nodes n, and the desired tolerance `tol` as inputs, outputting the Gaussian quadrature. The most challenging aspect of the code was definitely coding Newton's method. This difficulty arose because in the `legendrep` function, the derivative in the code `dp` is not actually the derivative. Let us define the output $\tilde{p}'(x) =$ `dp`. The derivative is then actually $p'(x) = \tilde{p}'(x)/(x^2-1)$, whereas the output is $\tilde{p}'(x) = p'(x)(x^2-1)$. So, in the code, I struggled with this concerning `x0` but I was able to eventually figure out how to code this. I also struggled with the weights. In class, we learned that the weights for Gaussian quadrature is given by

$$w_k = \frac{2}{(1-x_k^2)[P_n'(x_k)]^2}.$$

I confused the x^2-1 term with the term in the formula for the weights, but I was able to work this out. In the above code, using 1,000,000 Gaussian nodes and a tolerance of 10^{-3}, both MATLAB's `quadgk` and global adaptive quadrature method `integral` both utterly fail compared to Gaussian quadrature. However, with just a frequency of 100,000, `integral` performs pretty well, with an error of $\mathcal{O}(10^{-5})$. However, Gaussian quadrature has an error of $\mathcal{O}(10^{-10})$.

EXERCISE 2: PROVING THE RECURRENCE RELATION

To begin, we start with

$$(1-2xt+t^2)\sum_{n=1}^{\infty} P_n(x)nt^{n-1} = (x-t)\sum_{n=0}^{\infty} P_n(x)t^n,$$

which is derived from the Generating function for Legendre polynomials

$$g(x,t) = \frac{1}{\sqrt{1-2xt+t^2}}$$

We prove the recurrence relation for Legendre polynomials by equating the terms for t, t^2, and t^3. This is done by expanding the sums and collecting the like terms. What we obtain are the following expressions:

$$\begin{aligned} t:&\ 2P_2 - 2xP_1 = xP_1 - P_0, \\ t^2:&\ 3P_3 - 4xP_2 + P_1 = xP_2 - P_1, \\ t^3:&\ 4P_4 - 6xP_3 + 2P_2 = xP_3 - P_2. \end{aligned}$$

By solving for the highest order Legendre polynomial, we have

$$\begin{aligned} P_2 &= \frac{3xP_1 - P_0}{2}, \\ P_3 &= \frac{5xP_2 - 2P_1}{3}, \\ P_4 &= \frac{7xP_3 - 3P_2}{4}, \\ &\vdots \\ &\boxed{P_n = \frac{(2n-1)xP_{n-1} - (n-1)P_{n-2}}{n}}. \end{aligned}$$

We can see just from the first three terms, we can derive the recurrence relation for Legendre polynomials.

EXAM 2

EXERCISE 1

(25 points) Your main objective in this problem is to write a version of ***InvSqrt*** *based on Newton's method. The routine should take a positive number x and return its inverse square root computed to four digits of accuracy. Bear in mind that the main objective is the speed of computation.*

I suggest that you organize your solution as follows:

1. *Explain the algorithm verbally. You do not need to explain how Newton's method is derived, unless you really want to. However, you do need to explain the rationale behind your code: What is being iterated? How is the starting value computed? How does the code determine when to stop? And so on. As always, suppress the algebraic and Calculus details and focus on the main ideas behind the computation.*

2. *Test your algorithm on a large array of random numbers which you can generate using the* ***randn*** *command in* MATLAB*. The tolerance is low—only four digits—yet it must be met.*

3. *Time your routine and compare its performance with the naïve implementation of the inverse square root given above.*

4. *Conclude with a discussion of your results. Is your routine fast and reliable? If not, what can be done to improve it?*

Solution: Below is my `InvSqrt` function.

```
function y = InvSqrt(x)
    tol = 1e-9;
    y = newt_boi(x,tol);
    y = abs(y);
end

function xx = newt_boi(x,tol)
    xx = 1.5 - 0.5*x;
    while any(xx.^-2 - x > tol)
        xx = 1.5*xx - 0.5*x.*xx.^3;
    end
end
```

```
%% Exercise 1 Exam 2
```

```
clc
clear

N = 1e3;
m = 100;
t = zeros(size(1,N));
tt = t;
err = zeros(m,N);
tol = 1e-4;

for k = 1:N
    x = abs(randn(m,1));
    tic
    y = 1./sqrt(x);
    t(k) = toc;
    tic
    yy = InvSqrt(x);
    tt(k) = toc;
    err(:,k) = abs(y - yy)';
end

fails = 0;

for j = 1:N
    if err(j) > tol
        fails = fails + 1;
    end
end

fail_rate = fails/N * 100;

mean_t = mean(t);
std_t = std(t);

mean_tt = mean(tt);
std_tt = std(tt);

mean_err = mean(err);
std_err = std(err);

vals = [y yy];
```

Given a closed interval $[a, b]$, the use of Newton's method demands that the function be at least once differentiable, or $f(x) \in C^1[a, b]$, because the method involves the use of the function's first order Taylor polynomial. Generally, for a function $f(x) \in C^1[a, b]$ with a root $x_* \in [a, b]$, we can choose some $x_0 \in [a, b]$ as our center of expansion for our first order Taylor polynomial. As such, we have $T_1(x) = f(x_0) + f'(x_0)(x - x_0)$. By setting $T_1 = 0$, we obtain the root to our tangent line, which serves as the next "guess," and thus the next center of expansion, towards the approximation of the root of our function. This process can be implemented iteratively so as to potentially obtain the best approximation of the root.

Sometimes, the scheme doesn't converge, but when it does, it does so quadratically, as will be demonstrated later with a quasi proof.

My function began as something quite odious and heinous to all MATLAB-ians: a haven of for-loops. However, after refining my code, I ended up with the algorithm above. My function is a mapping $I : \mathbb{R}^n \mapsto \mathbb{R}^n$, which takes a vector as the input and outputs the vector whose components are the inverse square root. Within the function is my very own `newt_boi`, an implementation of Newton's method, which also takes the vector x as its input, in addition to the tolerance `tol`. Before it's final form, it also took the function $f(y) = 1/y^2 - x$ and its derivative with respect to y as inputs. However, I realized I could take advantage of MATLAB's vectorization. The `newt_boi` function begins with the guess, which was determined by linearizing f using a first order Taylor expansion at $y = 1$[6] and finding its root, given by $3/2 - x/2$. Then, I implemented a while-loop such that Newton's method iterates while the function evaluated at any element in the vector is greater than the tolerance, which is `1e-9`. The reason why the tolerance is 10^{-5} times less than the requirement is because even with a tol $= 10^{-8}$, the number of fails would be one to two, giving a non-zero failure-rate. So, the tight tolerance was a necessity to meet the tolerance. Once `newt_boi` finishes iterating, `InvSqrt` returns the vector of inverse square roots. I tested my function on a $10,000 \times 1$ vector of random numbers, and I performed this experiment 1000 times. Here are the results.

```
fail_rate =

     0

% [Naive_Method My_Method]
ans =

    0.6677    0.6677
    1.6941    1.6941
    1.1180    1.1180
    2.5645    2.5645
    1.0778    1.0778

% Mean of naive method
mean_t =

   3.5973e-05

% Mean of my method
mean_tt =

    0.0068

% Ratio
```

[6]Courtesy of MIT: web.mit.edu/10.001/Web/Course_Notes/NLAE/node7.html

```
>> mean_tt/mean_t

ans =

  190.1844
```

Firstly, the failure rate is zero, which is calculated by using the absolute value of the difference between mine and the naive approach, and the first five inverse square roots match with four digits of accuracy, and even more if the `format long` is used. The average time used by the naïve method is on the order of 10^{-5} seconds, whereas my method has a speed on the order of 10^{-3}, which is 200 times slower. My routine is definitely reliable, but it is quite a bit slower than the naïve method. I did as much optimization as I could, removing the necessity of for-loops and rewriting the code so as to optimize MATLAB's vectorization. So, there are two options that I could perform: remove the tightness of the tolerance and/or improve the initial guess. By doing either or both of these, my code would significantly improve. However, due to the inaccessibility to bit arithmetic, I don't think that I will be able to improve the code.

On the quadratic convergence of Newton's method, I performed some tests on a few functions. However, let me quickly define the quadratic convergence. Let $e_n = |x_n - x_*|$ be the error of the n-th approximation to the root. Newton's method converges in this manner: $e_{n+1} \sim C \cdot e_n^2$. Here are the results.

```
%% Exercise 1 Exam 2 "Proof" of quadratic convergence

y = @(s) cos(s) - s.^3;
dy = @(s) -sin(s) - 3*s.^2;
y1 = @(s) cos(2*s) - sin(2*s)./2 + s.^4;
dy1 = @(s) -2*sin(2*s) - cos(2*s) + 4*s.^3;

x = newt_method(y,dy,0.5,eps('double'));
yy = y(x);
err = abs(x - fzero(y,0.5))';
x1 = newt_method(y1,dy1,0.4,eps('double'));
err1 = abs(x1 - fzero(y1,0.4))';
yy1 = y1(x1)';

errs = [err(1:end-1) err1];

>> errs

errs =

    0.246667603995658 (1e-1)   0.025845035585202 (1e-2)
    0.044198660635192 (1e-2)   0.001151011157064 (1e-3)
    0.001789785107202 (1e-3)   0.000002701365516 (1e-6)
    0.000003102196650 (1e-6)   0.000000000015002 (1e-11)
    0.000000000009342 (1e-12)                  0
```

Although these are only two experiments, we can observe the quadratic decay.

Significantly Speeding Up the Code

Before submitting the exam, my code was slower than the naive method by about 200 times. However, after submitting the exam, and discussing with Justin, I modified my code slightly, and now my code is only slower than the naive method by only about 12 times, which is about 17 times faster than before! Here is that code and its output.

```
function y = InvSqrt(x)
    tol = 1e-2;
    y = newt_boi(x,tol);
    y = abs(y);
end

function xx = newt_boi(x,tol)
    xx = sqrt(x);
    while any(xx.^2 - 1./x > tol)
        xx = 0.5*xx + 0.5*(1./x).*(1./xx);
    end
end

mean_t =

     3.248099999999998e-05


mean_tt =

     3.954160000000009e-04

ratio =

  12.173763123056590
```

The difference between the last code and this code is instead of using $f(y) = 1/y^2 - x$, I used $f(y) = y^2 - 1/x$. This allowed me to optimize several aspects of my code. First, I decreased my tolerance to `1e-2` so that my code knows when to stop. This was fantastic because with the new equation, the new initial guess is $\sqrt{x}$. Then, instead of raising the input to negative one using the dot, I performed division using the dot. I initially didn't think that I could improve my code, but by recasting the equation into a different form, I increased the speed of my code by 17 times. So, my code is still reliable and much faster than before, but the naive case still wins.

EXERCISE 2

(20 points) Compute the following integral to single precision (6 - 7 digits):

$$\int_{-1}^{1} P_{100}(x)\cos(500x)\ dx.$$

Here $P_{100}(x)$ is the classical Legendre polynomial.

Solution: To compute this integral, I utilized my own `LegendreP` function in MATLAB and my Gaussian quadrature routine. I implemented the function in this code below and compared this with the output of MATLAB's global quadrature routine `integral`, in addition to Maple's output. The code and the Maple output are below, followed by my `LegendreP` and Gaussian quadrature functions below it, respectively.

```
%% Exercise 2 Exam 2

y = @(x) LegendreP(100,x);
f = @(x) y(x).*cos(500*x);

t = ones(size(1,10000));
tt = t;

for k = 1:10000
    tic
    I = integral(f,-1,1);
    t(k) = toc;
    tic
    Q = quad_gauss(f,1000,1e-4);
    tt(k) = toc;
end

mean_t = mean(t);
std_t = std(t);

mean_tt = mean(tt);
std_tt = std(tt);

% Output
>> Q

Q =

   0.003761482443423

>> I

I =

   0.003761482443423

>> [mean_t std_t]

ans =

   0.002494121600000   0.000915889183096

>> [mean_tt std_tt]

ans =

   0.003529128700000   0.000320615065988
```

```
> y := LegendreP(100, x);
                          y := LegendreP(100, x)
> g := cos(500*x);
                             g := cos(500 x)
> evalf(int(y*g, x = -1 .. 1));
                         0.003761482443422672782 4
```

```
function [p,dp] = LegendreP(n,x)

if n == 0
    p = ones(size(x));
    dp = zeros(size(x));
elseif n == 1
    p = x;
    dp = ones(size(x));
else
    p1 = x;
    p2 = ones(size(x));
    for m = 2:n
        p = ((2*m - 1)*(x.*p1) - (m-1)*p2)/m;
        p2 = p1;
        p1 = p;
    end
    dp = n.*(x.*p - p2);
end
end
```

```
function Q = quad_gauss(f,n,tol)

x0 = cos(pi*(0.75 + (0:n-1))/(n + 0.5));

for k = 1:1000
    [p,dp] = legendrep(n,x0);
    x0 = x0 - (p./dp).*(x0.^2 - 1);
    if norm(p,inf) < tol
        break
    end
end
w = 2./(dp.^2).*(1 - x0.^2);

Q = w*f(x0)';
end

function [p,dp] = legendrep(n,x)

if n == 0
    p = ones(size(x));
    dp = zeros(size(x));
elseif n == 1
    p = x;
    dp = ones(size(x));
else
    p1 = x;
    p2 = ones(size(x));
    for m = 2:n
        p = ((2*m - 1)*(x.*p1) - (m-1)*p2)/m;
        p2 = p1;
        p1 = p;
    end
    dp = n.*(x.*p - p2);
end
end
```

As we can see, the computation using Gaussian quadrature, with 1000 nodes and a tolerance of 10^{-4}, is exactly the same as the output used by MATLAB's integral function. Further, both are accurate to 14 digits after the decimal, twice the required amount. In performing 10,000 experiments, the average time taken by both methods are on the order of 10^{-3} seconds with standard deviations on the order of 10^{-4}. Thus, either method is suitable for this specific computation. When increasing the frequency and degree of the harmonic and polynomial, respectively, MATLAB's `integral` function outperforms my quadrature routine. Thus, the adaptive method is superior.

EXERCISE 3

(20 Points) Let $\{P_n\}_{n=0}^{\infty}$ be the sequence of classical Legendre polynomials. Consider the problem of approximating an exponential with a linear combination of the first four Legendre polynomials: $e^x \sim \sum_{n=0}^{3} c_n P_n$. Define the error of approximation to be:

$$E = \int_{-1}^{1} \left(e^x - \sum_{n=0}^{3} c_n P_n \right)^2 dx.$$

Find the coefficients c_n that minimize E. Generalize your result.

Solution: To begin, notice that we can define the space in which we are working by $L^2[-1,1]$, outfitted with the standard L^2-inner product, and consequently the L^2-norm. As such, we can define our error as

$$E = \int_{-1}^{1} \left(e^x - \sum_{n=0}^{3} c_n P_n \right)^2 dx = \left\langle e^x - \sum_{n=0}^{3} c_n P_n, e^x - \sum_{n=0}^{3} c_n P_n \right\rangle.$$

With this definition, we see that we are minimizing the square of the L^2-norm of our integrand. Thus, minimizing this error can easily be generalized, and this is what I will use for the derivation. Due to the bilinearity of the operation, we obtain

$$E = \langle e^x, e^x \rangle - 2 \sum_{n=0}^{N} c_n \langle e^x, P_n \rangle + \sum_{k=0}^{N} \sum_{n=0}^{N} c_k^2 \langle P_k, P_n \rangle,$$

assuming that $c_n = c_k$. So, to minimize E, which is a function of the c_n, we set the first derivative equal to zero,

$$\frac{\partial E}{\partial c_k} = -2 \sum_{n=0}^{N} \langle e^x, P_n \rangle + 2 \sum_{k=0}^{N} \sum_{n=0}^{N} c_k \langle P_k, P_n \rangle = 0.$$

This system can thus be written as a matrix vector system with the Gram matrix of Legendre polynomials, $Gc = b$, where $G = 2 \sum_{k=0}^{N} \sum_{n=0}^{N} c_k \langle P_k, P_n \rangle$, $c = c_k$, and $b = -2 \sum_{n=0}^{N} \langle e^x, P_n \rangle$. So, to solve for our coefficients c_n we solve the problem $c = G^{-1} b$. By solving for c, we find that

$$c_n = \frac{\langle e^x, P_n \rangle}{\langle P_n, P_n \rangle},$$

which is the n-th coefficient of the n-th term in the Fourier, or orthogonal expansion of e^x using the Legendre polynomials as the basis of expansion. This is the basis of my code. Below is my function to determine the coefficients and code that compares my Maple output of the coefficients with those determined by MATLAB.

```matlab
function [c,gram] = coeffs(N)
    gram = zeros(N,N);
    for ii = 1:N
        for jj = 1:N
            gram(ii,jj) = integral(@(x) ...
                LegendreP(ii−1,x).*LegendreP(jj−1,x),−1,1);
        end
    end
    b = zeros(1,N);
    for k = 1:N
        b(k) = integral(@(x) exp(x).*LegendreP(k−1,x),−1,1);
    end
    c = pinv(gram)*b';
    %c = gram\b';
end
```

```matlab
%% Exercise 3 Exam 2
% Solution

p0 = @(x) LegendreP(0,x);
p1 = @(x) LegendreP(1,x);
p2 = @(x) LegendreP(2,x);
p3 = @(x) LegendreP(3,x);

% Computed from Maple
c = [(exp(1)^2 − 1)/(2*exp(1)) 3/exp(1) (5*exp(1)^2 − 35)/(2*exp(1))...
    (−35*exp(1)^2 + 259)/(2*exp(1))];

% Computed from my function
[d,G] = coeffs(4);

x = linspace(−1,1,20);
f = @(x) c(1)*p0(x) + c(2)*p1(x) + c(3)*p2(x) + c(4)*p3(x);
g = @(x) d(1)*p0(x) + d(2)*p1(x) + d(3)*p2(x) + d(4)*p3(x);

figure
hold on
plot(x,exp(x),'b−');
plot(x,f(x),'ro');
plot(x,g(x),'k*');
legend('$y=e^{x}$','Maple Interpolant','My ...
    Interpolant','interpreter','latex');
xlabel('$x$','interpreter','latex');
ylabel('$y$','interpreter','latex');
title('Interpolating the Exponential');

>> coeffs(4)

```

```
ans =

   1.175201193643802
   1.103638323514327
   0.357814350647372
   0.070455633668489
```

```
> Digits := 20:
> N := 7;
> C := [seq(c[k], k = 0 .. N)];
      C := [c[0], c[1], c[2], c[3], c[4], c[5], c[6], c[7]]
> for n from 0 to N do
    z[n] := expand(LegendreP(n, x));
  end do;
> f := 0;

> for m to N + 1 do
    f := -C[m]*z[m - 1] + f;
  end do;

> f := (exp(x) + f)^2;

> E := integrate(f, x = -1 .. 1);

> for p to N + 1 do
    d[p] := diff(E, C[p]);
    soln[p] := evalf(solve(d[p] = 0, C[p]));
  end do;

> soln[1];
                     1.1752011936438014569
> soln[2];
                     1.1036383235143269648
> soln[3];
                     0.3578143506473724605
> soln[4];
                      0.070455633668489027
```

We can see that my output in MATLAB is exactly the same as my Maple algorithm of setting each partial derivative equal to zero. Below is a plot of the exponential and the interpolation using the first four Legendre polynomials.

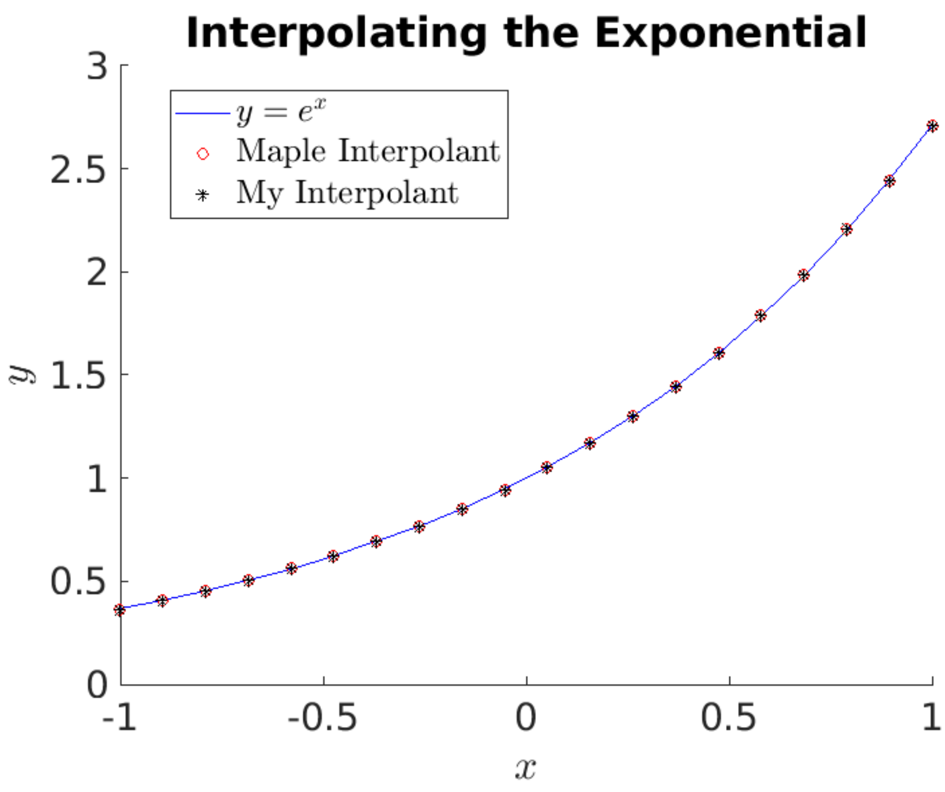


Figure 27: A plot of the exponential, my MATLAB interpolant, and my Maple interpolant.

EXERCISE 4

The setting for this problem is $L^2[1,\infty)$*—the space of functions on* $[1,\infty)$ *with the standard* L^2 *inner product:*

$$\langle f, g\rangle = \int_1^\infty f(x)g(x)dx.$$

Since $\langle x^k, x^k\rangle = \int_1^\infty x^{2k}dx = \infty$ *for all* $k \geq 0$*, the space* $L^2[1,\infty)$ *does not contain polynomials. Therefore, in order to construct an orthonormal basis using Gram-Schmidt process, we must start with functions other than monomials. One possible choice is negative powers* $\{x^{-n-1}\}_{n=0}^\infty$*. Let* $\{f_n\}_{n=0}^\infty$ *be the functions resulting from applying the Gram-Schmidt process to* $\{x^{-n-1}\}_{n=0}^\infty$*.*

(a) *(10 points) Show that* $f_n = \sqrt{2n+1}x^{-n-1}p_n$ *where* p_n *is a monic polynomial of degree* n *with integer coefficients. If you cannot find a rigorous proof, demonstrate that by computing* $p_0, p_1, \dots, p_{10}$*. Extra credit for finding a closed form expression for* p_n

(b) *(10 points) Let*

$$g(x) = \begin{cases} 1, & 1 \leq x \leq 2, \\ 0, & \text{otherwise.} \end{cases}$$

Compute the orthogonal expansion $h_N = \sum_{n=0}^N \langle g, f_n\rangle f_n$ *for* N *as large as possible. Plot* h_N *and* g *on the same axes and comment on the result.*

(c) *(10 points) Let* $q_n = (2n+1)^{-1/2}x^{-1}f_n(x^{-1})$*. Convince yourself that* q_n *is a polynomial of degree* n *(which should be restricted to the interval* $[0,1]$*). Does* $\{q_n\}_{n=0}^\infty$ *satisfy a three-term recurrence of the form* $xq_n = a_nq_{n+1} + b_nq_n + a_{n-1}q_{n-1}$*?*

Solution: To begin, I was not entirely sure how I would give a proof aside from performing Gram-Schmidt. I implemented Gram-Schmidt in Maple for the first 11 polynomials, from $n = 0, \dots, 10$. My maple code is below and the output below it.

```
> restart;
> with(orthopoly);
> with(PolynomialTools);
> with(combinat);
> N := 11;
> L2ip := (f, g) -> int(g*f, x = 1 .. infinity);
> L2norm := h -> sqrt(L2ip(h, h));
> V := [seq(x^(-k - 1), k = 0 .. N)];
> W[1] := V[1]/L2norm(V[1]);
> for n from 2 to N do
    z[n] := simplify(V[n] - add(L2ip(V[n], W[j])*W[j], j = 1 .. n - 1));
    W[n] := z[n]/L2norm(z[n]);
  end do;
```

$$\begin{aligned}
f_0 &= \frac{1}{x}\\
f_1 &= \frac{(2-x)\sqrt{3}}{x^2}\\
f_2 &= \frac{(x^2-6x+6)\sqrt{5}}{x^3}\\
f_3 &= \frac{(-x^3+12x^2-30x+20)\sqrt{7}}{x^4}\\
f_4 &= \frac{(x^4-20x^3+90x^2-140x+70)\sqrt{9}}{x^5}\\
f_5 &= \frac{(-x^5+30x^4-210x^3+560x^2-630x+252)\sqrt{11}}{x^6}
\end{aligned}$$

We can't really draw any conclusions in going from $n = 0$ to $n = 1$, but we can clearly see that, for example, when $n = 4$, we find that

$$f_4 = \sqrt{2 \cdot 4 + 1}x^{-4-1}p_4$$

where

$$p_n = \sum_{k=0}^{n} \frac{\Gamma(n+1)(2-x)^{n-2k}(x-1)^k(-1)^k}{\Gamma(n-2k+1)\Gamma(k+1)^2}.$$

Thus, in all it's glory, we have

$$f_n = \sqrt{2n+1}\; x^{-n-1} \sum_{k=0}^{n} \frac{\Gamma(n+1)(2-x)^{n-2k}(x-1)^k(-1)^k}{\Gamma(n-2k+1)\Gamma(k+1)^2}.$$

The derivation of p_n was quite the journey. I was able to discern a pattern with the expressions $(2-x)$ and $(x-1)$. Here is that pattern for the first five polynomials.

$$\begin{aligned}
p_0 &= (2-x)^0\\
p_1 &= (2-x)^1\\
p_2 &= (2-x)^2 - 2(x-1)\\
p_3 &= (2-x)^3 - 6(x-1)(2-x)\\
p_4 &= (2-x)^4 - 12(x-1)(2-x)^2
\end{aligned}$$

So immediately, my mind went to a modified binomial theorem. It was easy to see that the polynomials alternated, hence the $(-1)^k$ term. However, the most difficult part was determining the coefficients of the choose function, and what was particularly disorienting was the fact that there was a second choose function. Then, using Maple, I converted the choose functions into the gamma notation presented above. I spent a lot of time deriving the above function, but not in vain. I figured that if I could determine the function explicitly, the computation of the orthogonal expansion and plotting q would be simple, which was exactly the case. Below is the Maple code for the above computations in addition to the code for computing the coefficients for part (c) and the corresponding polynomials q.

```
> p := (x, n) -> sum(GAMMA(n + 1)*(2 - x)^(n - 2*k)*
(x - 1)^k*(-1)^k/(GAMMA(n - 2*k + 1)*GAMMA(k + 1)^2), k = 0 .. n);

> f := (x, n) -> sqrt(2*n + 1)*x^(-n - 1)*p(x, n);

> for m from 0 to 15 do
    q[m] := simplify(f(1/x, m)/(sqrt(2*m + 1)*x));
  end do

> for l to 14 do
    a[l] := IP(x*q[l], q[l + 1])/IP(q[l + 1], q[l + 1]);
    b[l] := IP(x*q[l], q[l])/IP(q[l], q[l]);
    c[l] := IP(x*q[l], q[l - 1])/IP(q[l - 1], q[l - 1]);
    {a[l], b[l], c[l]};
  end do
```

The first five polynomials q_n are

$$\begin{aligned} q_0 &= 1, \\ q_1 &= 2x - 1, \\ q_2 &= 6x^2 - 6x + 1, \\ q_3 &= 20x^3 - 30x^2 + 12x - 1, \\ q_4 &= 70x^4 - 140x^3 + 90x^2 - 20x + 1. \end{aligned}$$

The sequences for a_n, b_n, and c_n are given by

$$\begin{aligned} a_n &= \left\{\frac{1}{3}, \frac{3}{10}, \frac{2}{7}, \frac{5}{18}, \frac{3}{11}, \frac{7}{26}, \frac{4}{15}, \frac{9}{34}, \frac{5}{19}, \dots\right\} \\ b_n &= \left\{\frac{1}{2}\right\} \\ c_n &= \left\{\frac{1}{6}, \frac{1}{5}, \frac{3}{14}, \frac{2}{9}, \frac{5}{22}, \frac{3}{13}, \frac{7}{30}, \frac{4}{17}, \frac{9}{38}, \dots\right\}, \end{aligned}$$

where

$$a_n = \frac{n+1}{4n+2},$$
$$c_n = \frac{n}{4n+2} \neq a_{n-1}.$$

Clearly, given the sequence for c_n, these polynomials don't obey $xq_n = a_n q_{n+1} + b_n q_n + a_{n-1} q_{n-1}$. However, to verify, I computed the coefficients for $n = 1$ and $n = 2$ as though the polynomials obeyed the rule, and I confirmed that they don't. However, they do obey the rule $xq_n = a_n q_{n+1} + b_n q_n + c_n q_{n-1}$, which is true for any set of orthogonal polynomials. Below are the plot and the code for the plots of the q_n.

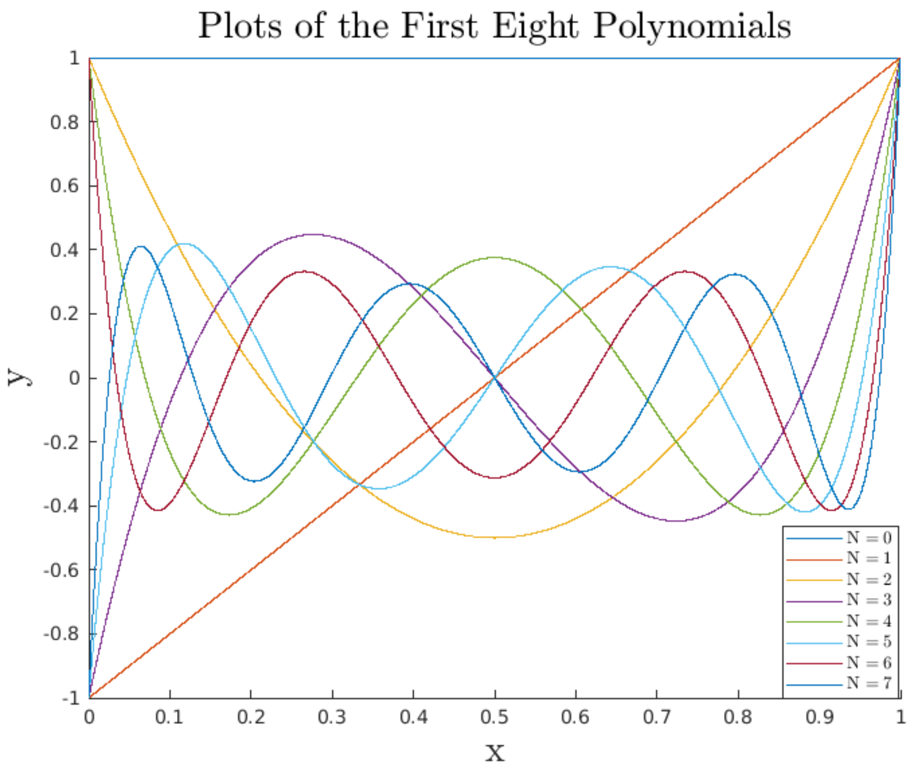


Figure 28: Plot of $q_n = (2n+1)^{-1/2} x^{-1} f_n(x^{-1})$ for $n = 0..7$.

```
function f = ex2_fun(x,n)
    pn = zeros(size(x));
    for k = 0:n
        pn = pn + ...
            ((gamma(n+1))/((gamma(k+1)^2).*gamma(n-2*k+1))).*((2-x).^(n ...
            - 2*k)).*((x-1).^(k)).*(-1)^k;
    end
    f = sqrt(2*n + 1).*(x.^(-n-1)).*pn;
end
```

```
%% Exercise 4 Exam 2 Part 3

N = 7;
x = linspace(0,1,10000);
u = 1./x;
q = zeros(N+1,10000);

for k = 0:N
    q(k+1,:) = (1/sqrt(2*k+1)).*u.*ex2_fun(u,k);
    hold on
    plot(x,q(k+1,:));
end
```

We can see that the polynomials are indeed restricted on [0,1]. Lastly, below is the code and plot for the orthogonal expansion of g.

```
%% Exercise 4 Exam 2 Part 2

N = 1000;
n = 49;
x = linspace(1,2,N);

g = @(x) 1;
h = zeros(size(x));

for k = 0:n
    h = h + integral(@(x) ex2_fun(x,k).*g(x),1,2).*ex2_fun(x,k);
end

figure
hold on
plot(x,h);
plot(x,ones(size(x)));
legend('$h_{49}$','y = g(x)','interpreter','latex');
xlabel('$x$','interpreter','latex');
ylabel('$y$','interpreter','latex');
title('Interpolation of $y = g(x)$','interpreter','latex');
set(gca,'fontsize',20);
```

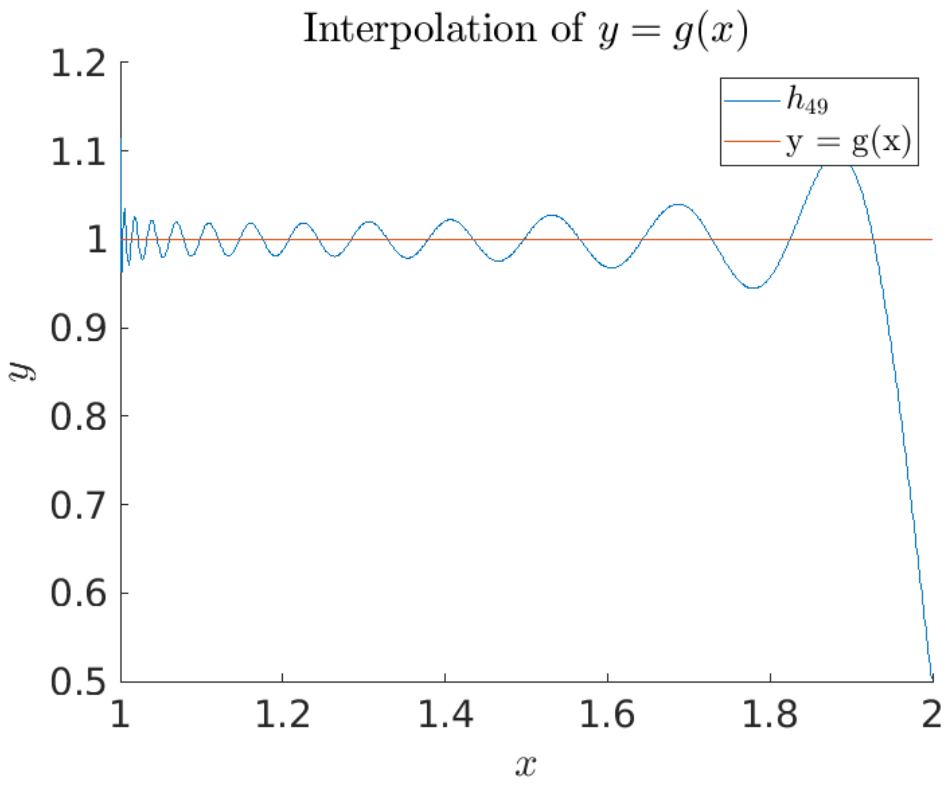


Figure 29: The plot of the orthogonal expansion of g using 49 polynomials f_n.

The highest degree of f_n I could use was $N = 49$. In doing so, the orthogonal expansion of g by the f_n results in Runge's phenomenon at the ends of the intervals. This is expected due to the use of equispaced nodes and the use of polynomials for interpolation.

Some Comments: The Mystery Polynomials

After doing some thinking and some research, I discovered that the mystery polynomials are indeed shifted Legendre polynomials, where

$$p_n = \widetilde{P}_n(x) = P_n(2x-1).$$

As such, we can modify the original three term recurrence relation for Legendre polynomials by replacing x with $2x-1$ to obtain

$$\widetilde{P}_n(x) = P_n(2x-1) = \frac{(2n-1)(2x-1)\widetilde{P}_{n-1} - (n-1)\widetilde{P}_{n-2}}{n}.$$

Although my mess of a sum is indeed an explicit formula for these polynomials, the above representation is much more compact and pleasing to the eye.

HOMEWORK 6

EXERCISE 1

1.) Consider the problem of minimizing

$$u = -c\left(\frac{1}{2}\sum_i \ln\left(1 - x_i^2\right) + \sum_{i>j} \ln(x_i - x_j)\right), \quad c = \frac{1}{2\pi N^2}.$$

This expression differs from Equation (14) only by a factor of $\frac{1}{2}$ in front of the first sum: the fixed charges at the endpoints have half the magnitude of the charges in the interior.

(a) Compute the first seven nodal polynomials corresponding to the minima of potential energy. Plot the polynomials and their roots as in Figure 13. Also produce a plot similar to Figure 15 showing that the roots are interlaced.

(b) Find the three-term recurrence relation satisfied by the nodal polynomials. These polynomials are Jacobi polynomials $P_n^{(a,b)}$—find a and b.

(c) Compute 500 nodes and plot their cumulative distribution function. Compare that with the inverse sine law.

Solution: To compute the nodes that minimize u, I first derived the gradient and the Hessian for the i-th node, given below.

$$\text{Gradient:} \quad \frac{\partial u}{\partial x_i} = -c\left(\frac{-x_i}{1 - x_i^2} + \sum_{i \neq j}^{N} \frac{1}{x_i - x_j}\right),$$

$$\text{Hessian:} \quad \frac{\partial^2 u}{\partial x_i \partial x_j} = -c\begin{cases} \dfrac{-1}{(x_i - x_j)^2}, & i \neq j \\ \dfrac{(1 + x_i^2)}{(1 - x_i^2)^2} - \displaystyle\sum_{i \neq j} \frac{1}{(x_i - x_j)^2}, & i = j \end{cases}$$

To understand why we need the gradient and the Hessian, let me first discuss Newton's method—first for a function $f : \mathbb{R} \to \mathbb{R}$ and then for $f : \mathbb{R}^n \to \mathbb{R}$.

Newton's Method in 1-D

Let's define $x_* \in (a, b)$ as the root of f such that $f(x_*) = 0$. If f maps the open interval $(a, b) \subset \mathbb{R}$ into $\mathbb{R}$, and f is differentiable at some $x_0 \in (a, b)$, then the approximation x_1 of the root of f is given by

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

This is Newton's method for the 1-dimensional case. The use of such a method necessitates the use of the derivative of f at x_0. However, the classical definition of the derivative,

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h},$$

which is a number, does not generalize well for higher dimensional cases. So, I will explain a different way to conceptualize the derivative such that it generalizes well for higher dimensional cases[7]. We can rewrite the above expression as

$$f(x+h) - f(x) = f'(x)h + R(h),$$

where we can express the sum on the left-hand-side as the sum of the linear transformation $f'(x)$ that takes $h \in \mathbb{R}$ to $f'(x)h$ and the remainder or error $R(h)$, which is small such that

$$\lim_{h\to 0} \frac{R(h)}{h} = 0.$$

Thus, $f'(x)$ is the linear transformation that satisfies

$$\lim_{h\to 0} \frac{f(x+h) - f(x) - f'(x)h}{h} = 0.$$

This definition is much stronger because the derivative can be generalized to higher dimensions. Further, it makes sense because every $h \in \mathbb{R}$ has an associated linear operator on $\mathbb{R}$, i.e. scaling by that h, and conversely, every linear operator from $\mathbb{R}$ to $\mathbb{R}$ is scalar multiplication.

Definition. For a function $f : E \to \mathbb{R}^m$, where $E \subset \mathbb{R}^n$ and E contains a neighborhood of x, f is *differentiable* at x if there is an $m \times n$ matrix A such that

$$\lim_{h\to 0} \frac{|f(x+h) - f(x) - Ah|}{|h|} = 0,$$

where $h \in \mathbb{R}^n$. This is called the *total derivative* or the *differential* of f at x.

Remarks. Although I will not prove it here, $A = f'(x)$ is unique and can generally be expressed as the Jacobian Matrix[8]. When $n = 1$, A is given by the Jacobian as a single column vector, and when $m = 1$, like in the above exercise, A is given by the gradient, which is a row vector.

[7]Courtesy of Walter Rudin's *Principles of Mathematical Analysis, 3ed* and James Munkres' *Analysis on Manifolds.*

[8]The total derivative is not the Jacobian and can only be expressed as such when all partial derivatives of f at some point x exist and are continuous around x. So, the total derivative of f at some x is a linear map describing the best linear approximation of f at x where the remainder, or error, is small and zero at best. To connect this idea to Numerical Analysis and the Calculus that I took long ago in a galaxy far, far away, the total derivative is used in the generalization of the best first-order approximation, or linear interpolant, of a mapping at an element in its domain of definition.

Multi-Dimensional Newton's Method

In light of the previous discussion, let us first define $u : E \to \mathbb{R}$ where $E \subset \mathbb{R}^n$, and let $x_* \in E$ be the vector of values that minimize u. This means that $u'(x_*) = 0$. Here, $u'(x) = \nabla u(x) = g(x)$. Let $x_1 \in E$ be the vector of approximations to the problem $g(x) = 0$ obtained from $x_0 \in E$, the vector of guesses used to initialize Newton's method. So Newton's method is given by

$$\begin{aligned} x_1 &= x_0 - g'(x_0)g(x_0) \\ &= x_0 - H^{-1}g(x_0). \end{aligned}$$

where $g(x_0)$ is the differential of u at x_0 (the gradient), and $H^{-1} = \nabla^T(\nabla u(x_0))$, which is the Hessian of u evaluated at x_0. This is why the gradient and the Hessian matrices are required for multi-dimensional Newton's method, and hence the linear transformation definition of the derivative.

Back to the Exercise at Hand

So, after determining the gradient and the Hessian using Maple, I coded Newton's method in MATLAB. The first code snippet below contains Newton's method.

```
function x = optimize(p,tol)
    MaxIter = 25;
    opts.SYM = true;
    opts.POSDEF = true;
    x = p(:);
    o = ones(size(x));
    N = length(x);
    idx = (1:(N+1):N^2);
    c = 1/(2*pi*N^2);

    for k = 1:MaxIter

        % forming the gradient
        H = -1./(x*o' - o*x');
        y = 1./(1-x.^2);
        H(idx) = x.*y;

        g = sum(H,2); % gradient of u

        % forming the Hessian matrix
        H = -H.^2;
        H(idx) = 2*H(idx);
        H(idx) = y - sum(H,2);

        % solving linear system and
        y = linsolve(H,g,opts);
        x = x - y;

        fprintf('iter = %f; norm(y) = %f; cond(H) = ...
            %f\n',[k,norm(y,'inf'),cond(H)])

        % stopping point
        if norm(y,'inf') < tol
            break;
        end
    end
end
```

This first block of code serves as the set up for Newton's method. The code takes the initial guess and the tolerance as inputs and outputs the refined approximations for the roots of u. The plots of the first six nodal polynomials are below, in addition to the plot of the interlaced roots.

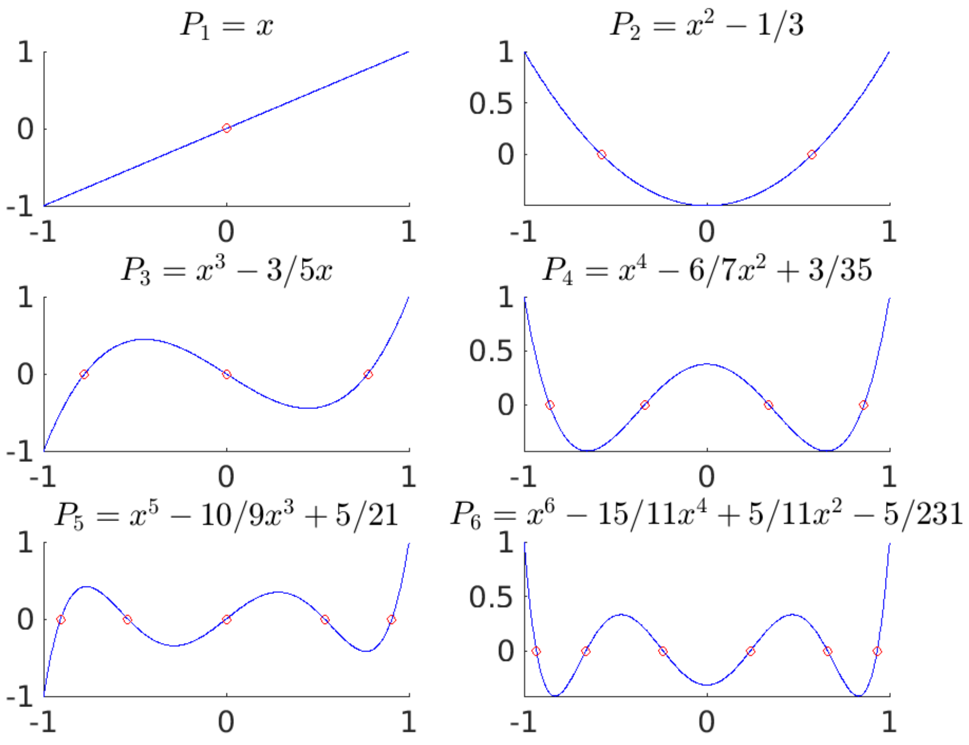


Figure 30: Plot of the first six polynomials formed from the roots of u, which are Legendre polynomials.

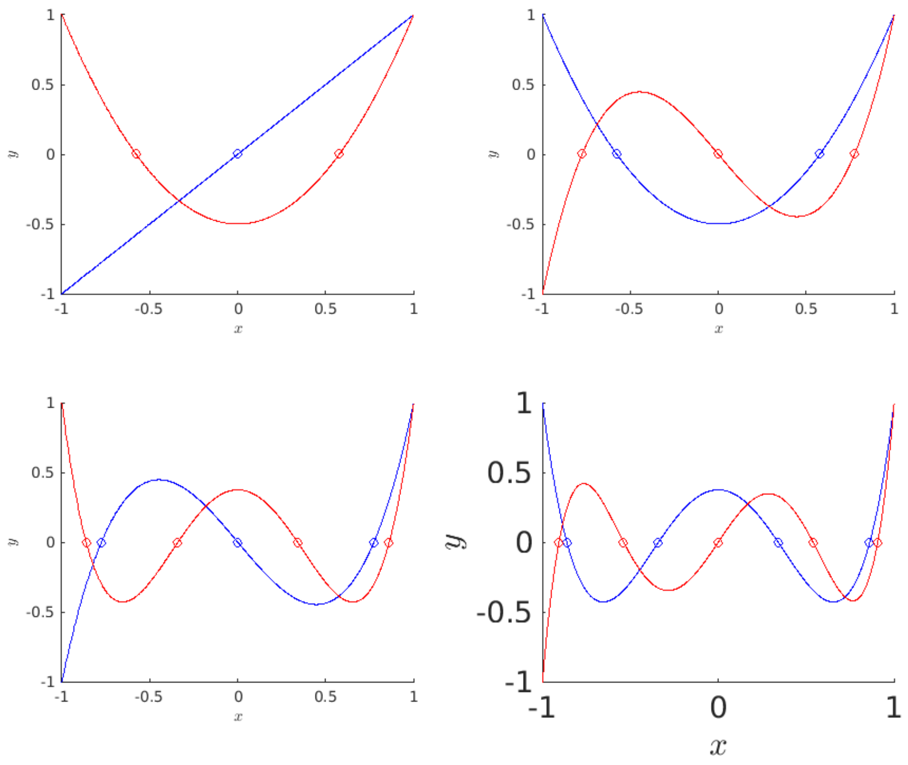


Figure 31: The interlaced nodes.

Figure 1 shows that the roots of u are Legendre polynomials, and Figure 2 shows that they are interlaced. As such, $a = b = 0$ and the recurrence relation is

$$P_n = \frac{(2n-1)xP_{n-1} - (n-1)P_{n-2}}{n}.$$

Here is the code that generated the above plots.

```
    N = 1000;
    p = initial_guess(N); % hopefully better guess
    x = optimize(p,eps('double'));
    u = energy(x);
    xx = linspace(-1,1,length(x));
    xx1 = linspace(0,1,length(x));
    poly(x)
    txt1 = '$P_{1} = x$';
    txt2 = '$P_{2} = x^2 - 1/3$';
    txt3 = '$P_{3} = x^3 - 3/5x$';
    txt4 = '$P_{4} = x^4 - 6/7x^2 + 3/35$';
    txt5 = '$P_{5} = x^5 -10/9x^3 + 5/21$';
    txt6 = '$P_{6} = x^6 - 15/11x^4 + 5/11x^2 - 5/231$';
    txt = {txt1 txt2 txt3 txt4 txt5 txt6};

    % plots of the first 6 polynomials
    figure
    for k = 1:6
        subplot(3,2,k)
        hold on
        plot(xx,LegendreP(k,xx),'b-');
        plot(optimize(initial_guess(k),eps('double')),zeros(1,k),'ro');
        title(txt{k},'interpreter','latex');
    end

    % plots of the interlaced roots
    figure
    for j = 1:4
        subplot(2,2,j)
        hold on
        plot(xx,LegendreP(j,xx),'b-',...
        optimize(initial_guess(j),eps('double')),zeros(1,j),'bo');
        plot(xx,LegendreP(j+1,xx),'r-',...
        optimize(initial_guess(j+1),eps('double')),zeros(1,j+1),'ro');
    end
```

Lastly, I wanted to plot not only the CDF, but also the distribution of the nodes. These are below.

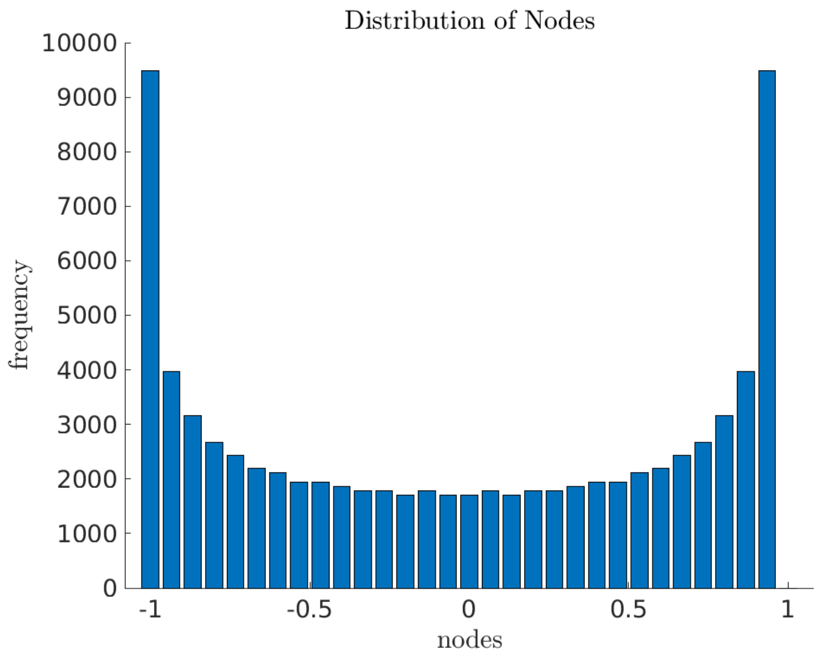


Figure 32: Plot of the distribution of the nodes.

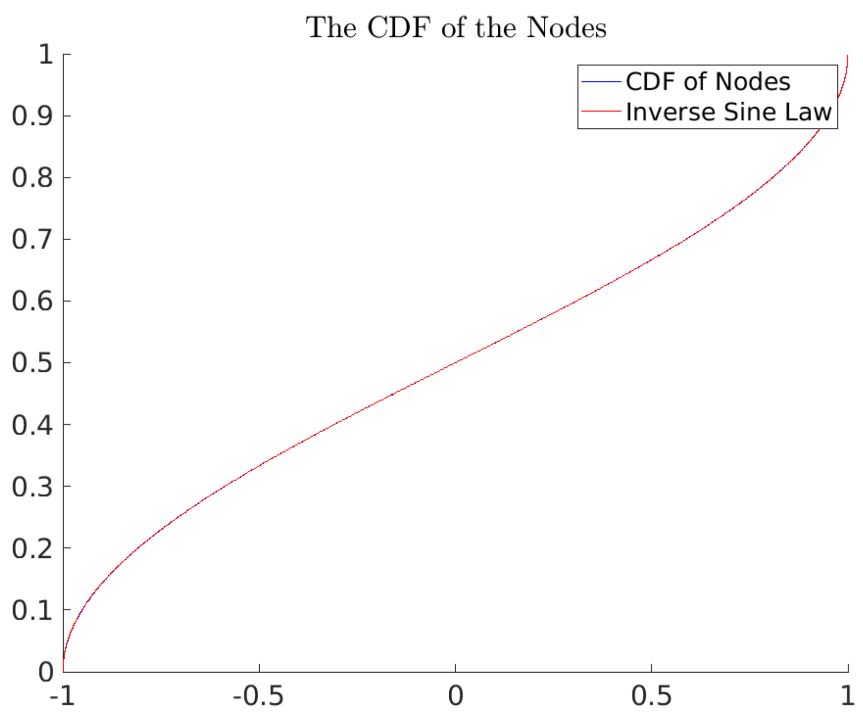


Figure 33: Plot of the CDF.

We can see that the distribution of the nodes is concentrated around the endpoints and almost exactly matches the inverse sine law. For 5000 roots, the energy is $u = 0.055023335729060$ which looks very similar to $\frac{\ln(2)}{4\pi}$. Below are the rest of the functions used to compute the initial guess, the energy, and the histogram of the nodes.

```
function x = initial_guess(N,MaxIter)
    x = 0;
    for n = 1:N-1
        x = .5*([-1;x] + [x;1]);
        x = optimize(x,eps('double'),MaxIter);
    end
end

function u = energy(p)
    x = p(:);
    N = length(x);
    c = 1/(2*pi*N^2);
    o = ones(N,1);
    xx = abs(x*o' - o*x');
    xx(1:N+1:N^2) = 1;
    u = -c*(0.5*sum(log(1-x.^2)) + 0.5*sum(log(xx(:))));
```

```matlab
end

function plt = my_hist(x)
    n = length(x);
    xx = linspace(-1,1,sqrt(n));
    dx = x(2) - x(1);
    yy = histc(x,xx);
    yy = yy/(numel(x)*dx);
    plt = bar(xx,yy);
end
```

EXERCISE 2

2.) Minimize the energy of "real" three-dimensional charges:

$$u = -c\left(\sum_i (1-x_i)^{-1} + \sum_i (x_i+1)^{-1} + \sum_{i>j}(x_i-x_j)^{-1}\right).$$

The constant c should be set to $\frac{1}{4\pi N^2}$. Follow the outline of the previous exercise. What can you say about the minimal energy as $N \to \infty$?

Solution: To begin, I rewrote u in the following manner,

$$u = -c\left(\sum_i \frac{2}{1-x_i^2} + \sum_{i>j}\frac{1}{x_i-x_j}\right),$$

and the gradient and Hessian are given by

$$\text{Gradient:}\quad \frac{\partial u}{\partial x_i} = \frac{4x_i}{(1-x_i^2)^2} + \sum_{j=1}\begin{cases}\dfrac{1}{(x_i-x_j)^2}, & i<j\\[2ex] \dfrac{-1}{(x_i-x_j)^2}, & i>j,\end{cases},$$

$$\text{Hessian:}\quad \frac{\partial^2 u}{\partial x_i \partial x_j} = \begin{cases}\dfrac{-2}{(x_i-x_j)^3}, & i\neq j\\[3ex] \dfrac{16x_i^2}{(1-x_i^2)^3} + \dfrac{4}{(1-x_i^2)^2} + \displaystyle\sum_{k=1}\begin{cases}\dfrac{-2}{(x_i-x_k)^3}, & i<k\\[2ex] \dfrac{2}{(x_i-x_k)^3}, & i>k\end{cases}, & i=j.\end{cases}$$

of course scaling both by $-c$. The code for Newton's method is below.

```
function x = optimize(p,tol,MaxIter)
    opts.SYM = true;
    opts.POSDEF = true;
    x = p(:);
    o = ones(size(x));
    N = length(x);
    idx = (1:(N+1):N^2);

    for k = 1:MaxIter

        % forming the gradient
        xx = x*o'-o*x';
        H = xx.^-2;
        H = -tril(H,-1) + triu(H,1);
        y = 4./(1-x.^2).^2;
        H(idx) = x.*y;

        g = sum(H,2);

        % Forming the Hessian
        H = 2*H./xx;
        H(idx) = -y.^2.*x.^2.*(1-x.^2);
        H(idx) = y - sum(H,2);

        % solving linear system and
        y = linsolve(H,g,opts);
        x = x - y;

        %fprintf('iter = %f; norm(y) = %f; cond(H) = ...
            %f\n',[k,norm(y,'inf'),cond(H)])

        % stopping point
        if norm(y,'inf') < tol
            break;
        end
    end
end
```

The thing to note about the code is that to accomplish the gradient, the `tril` and `triu` functions were used to return the lower and upper triangular portions of H. Otherwise, the Hessian was simple to implement. Below are the plots and the code that generated them.

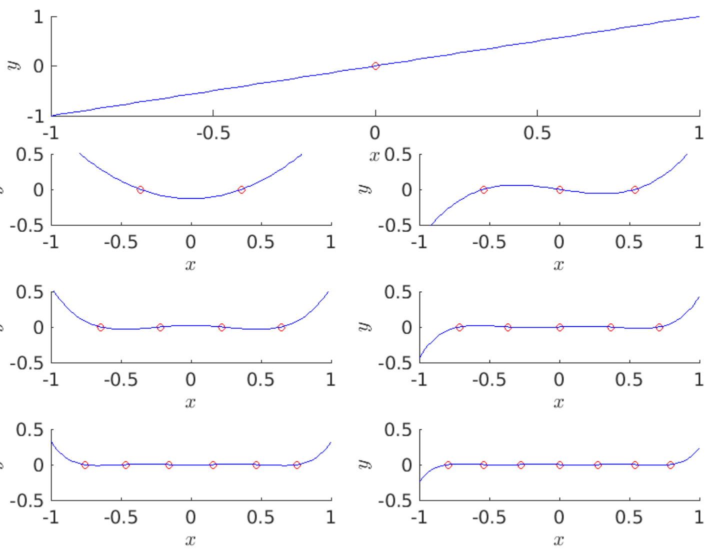


Figure 34: Plot of the first seven polynomials.

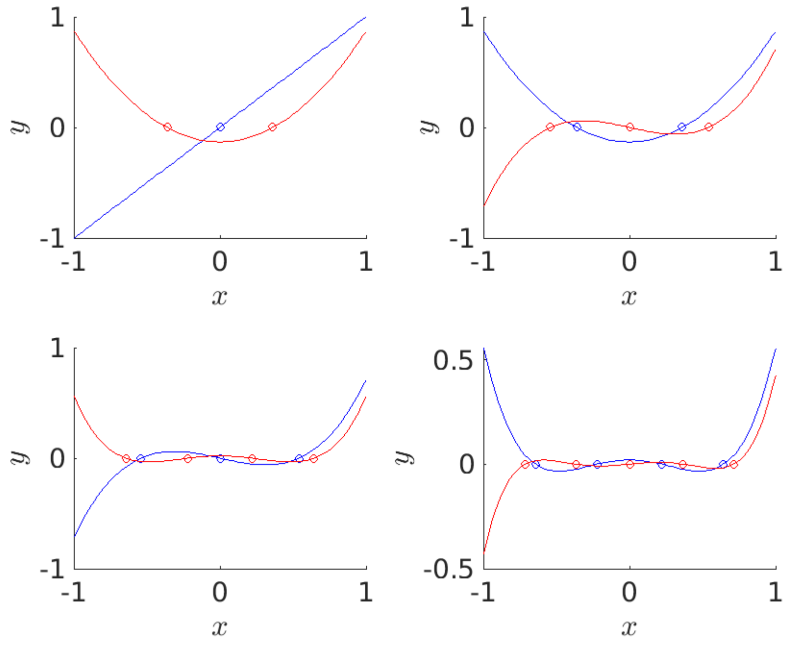


Figure 35: Plot of the interlaced nodes.

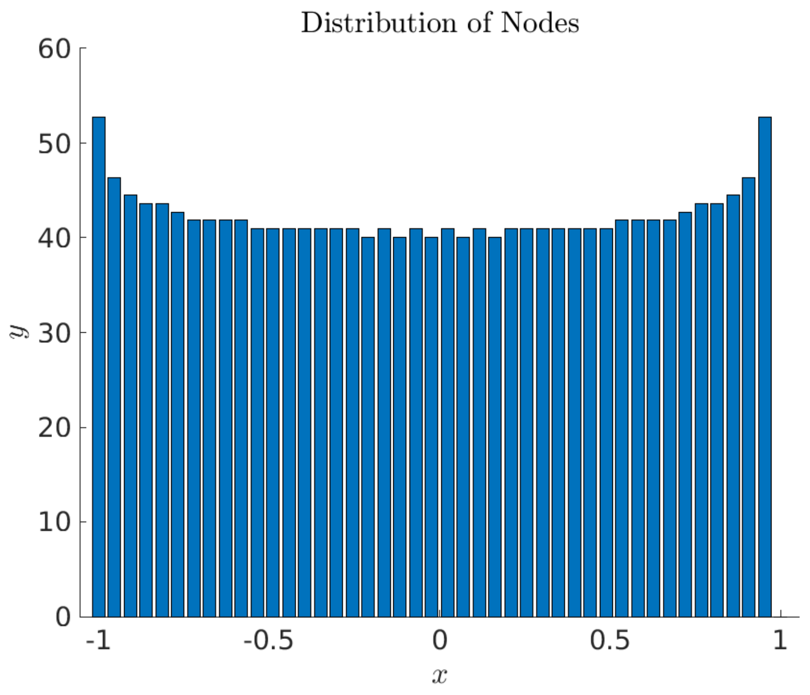


Figure 36: Distribution of the nodes

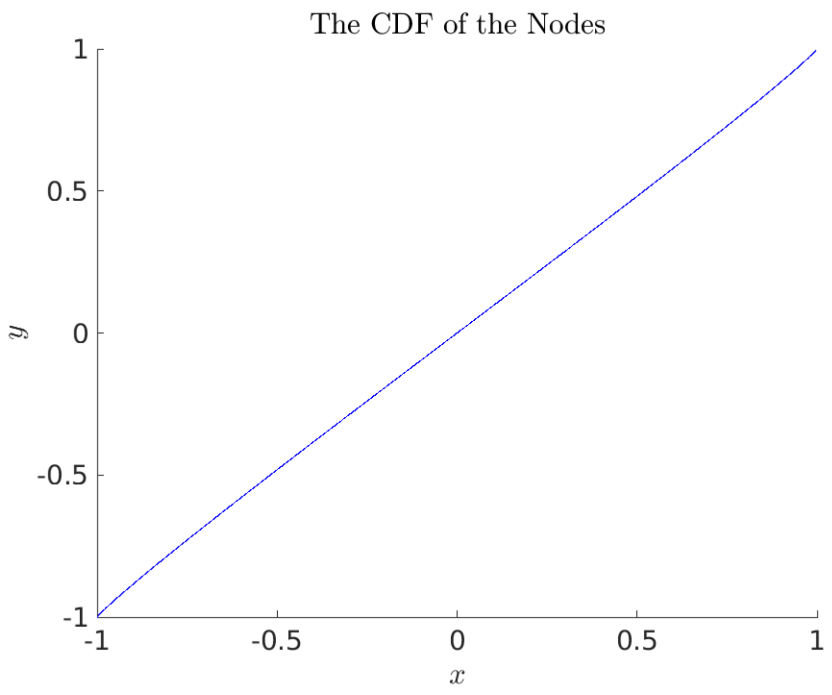


Figure 37: Plot of the CDF

```
N = 2000;
p = initial_guess(N,2); % hopefully better guess
x = optimize(p,eps('double'),20);
u = energy(x)
%xx = linspace(-1,1,N);

% plots of the first 7 polynomials
figure
subplot(4,2,[1 2])
r = optimize(initial_guess(0,2),eps("double"),20);
rr = linspace(-1,1);
p = poly(r);
hold on
plot(optimize(initial_guess(0,2),eps('double'),10),0,'ro');
plot(rr,polyval(p,rr),'b-');
xlabel('$x$','interpreter','latex');
ylabel('$y$','interpreter','latex');
set(gca,'fontsize',15);
for k = 0:5
    r = optimize(initial_guess(k+2,2),eps("double"),20);
    rr = linspace(-1,1);
    p = poly(r);
    subplot(4,2,k+3)
    hold on
    plot(optimize(initial_guess(k+2,2),eps('double'),20),0,'ro');
    plot(rr,polyval(p,rr),'b-');
    xlabel('$x$','interpreter','latex');
    ylabel('$y$','interpreter','latex');
    ylim([-0.5 0.5]);
    set(gca,'fontsize',15);
end

% plots of the interlaced roots
figure
title('Interlaced Nodes');
for j = 1:4
    r = optimize(initial_guess(j,2),eps("double"),10);
    r1 = optimize(initial_guess(j+1,2),eps("double"),10);
    rr = linspace(-1,1);
    p = poly(r);
    p1 = poly(r1);
    subplot(2,2,j)
    hold on
    plot(rr,polyval(p,rr),'b-',...
        optimize(initial_guess(j,2),eps('double'),10),0,'bo');
    plot(rr,polyval(p1,rr),'r-',...
        optimize(initial_guess(j+1,2),eps('double'),10),0,'ro');
    xlabel('$x$','interpreter','latex');
    ylabel('$y$','interpreter','latex');
    set(gca,'fontsize',20);
end

% distribution of the nodes
figure
hold on
my_hist(x);
xlabel('$x$','interpreter','latex');
ylabel('$y$','interpreter','latex');
title('Distribution of Nodes','interpreter','latex');
set(gca,'fontsize',20);

% plot of CDF
figure
hold on
plot(x,xx,'b-');
xlabel('$x$','interpreter','latex');
ylabel('$y$','interpreter','latex');
title('The CDF of the Nodes','interpreter','latex');
set(gca,'fontsize',20);
```

The first few polynomials are the following,

$$\begin{aligned} p_1 &= x \\ p_2 &= x^2 - \frac{366}{2801} \\ p_3 &= x^3 - \frac{356}{1223}x \\ p_4 &= x^4 - \frac{746}{1613}x^2 + \frac{89}{4457} \\ p_5 &= x^5 - \frac{968}{1515}x^3 + \frac{471}{7000}x, \end{aligned}$$

and their roots appear to be evenly distributed according to the histogram and the CDF, which does not obey the inverse sine law. I'm not sure what these polynomials are, and in Maple, I tried to see if I could determine a and b for the Jacobi polynomials, but I could not. The fact that the roots are interlaced suggests that the polynomials may or may not belong to the family of the classical orthogonal polynomials. As for the minimum energy, this is what I observed:

$$\begin{aligned} N &= 1000 \qquad u \approx 0.27 \\ N &= 2000 \qquad u \approx 0.28 \\ N &= 5000 \qquad u \approx 0.32. \end{aligned}$$

It appears that the minimal potential energy approaches some value, but I'm not entirely sure what that value is. The code for the plots are the same as those as above.

EXERCISE 3

3.) Replace the interval [-1,1] with an arc of the unit circle. That is, consider the problem of minimizing

$$u = -c\left\{\sum_{i=1}^{N}\ln|p_0-p_i| + \sum_{i=1}^{N}\ln|p_{N+1}-p_i| + \sum_{\substack{i=1\\i>j}}^{N}\ln|p_i-p_j|\right\}$$

where $p_i = (cos(t_i), sin(t_i))$ with $t_0 = 0$ and $t_{N+1} = \pi$, and $c = \frac{1}{2\pi N^2}$.

(a) *Compute the first seven nodal polynomials. Do they satisfy a three-term recurrence relation? Are the roots interlaced?*

(a) *Solve the problem for as high an N as possible. Compute the cumulative distribution: does it bear any similarity to the inverse sine law? Also, compute the minimal electrostatic energy u_N^*. Does it seem to approach a limit as $N \to \infty$.*

Solution: Again, I rewrote the u as

$$u = -c\left\{2N\ln(2) + \sum_{i=1}^{N}\ln(\sin(t_i)) + \sum_{\substack{i=1\\i>j}}^{N}\ln\left[\sin\left(\frac{t_i-t_j}{2}\right)\right]\right\}.$$

The above expression was accomplished by taking the norm of the arguments of the natural logarithms and after simplifying using trigonometric identities and using the properties of logarithms, we obtain the above result. The gradient and Hessian are given by

$$\text{Gradient}\quad \frac{\partial u}{\partial t_i} = -c\left(\cot(t_i) + \frac{1}{2}\sum_{\substack{i=1\\i\neq j}}^{N}\cot\left(\frac{t_i-t_j}{2}\right)\right),$$

$$\text{Hessian}\quad \frac{\partial^2 u}{\partial t_i\partial t_j} = -c\begin{cases}\frac{1}{4}+\frac{1}{4}\cot^2\left(\frac{t_i-t_j}{2}\right), & i\neq j\\ -\csc^2(t_i) - \frac{1}{4}\sum_{\substack{i=1\\i\neq j}}^{N}\cot^2\left(\frac{t_i-t_j}{2}\right), & i=j\end{cases}.$$

Here's Newton's method for this system. And below are the plots for this system.

```
function t = optimize(p,tol,MaxIter)
    opts.SYM = true;
    opts.POSDEF = true;
    t = p(:);
    o = ones(size(t));
    N = length(t);
    idx = (1:(N+1):N^2);

    for k = 1:MaxIter

        % forming the gradient
        tt = 0.5*(t*o'-o*t');
        H = 0.5*cot(tt);
        H(idx) = cot(t);
        g = -sum(H,2);

        % forming the Hessian
        H = H.^2;
        H(idx) = 0;
        HH = 0.25 + H;
        HH(idx) = -csc(t).^2 - sum(H,2);
        HH = -HH;

        % solving linear system and
        y = linsolve(HH,g,opts);
        t = t - y;

        %fprintf('iter = %f; norm(y) = %f; cond(H) = ...
            %f\n',[k,norm(y,'inf'),cond(H)])

        stopping point
        if norm(y,'inf') < tol
            break;
        end
    end
end
```

The first several polynomials are:

$$p_1 = x - \frac{\pi}{2}$$
$$p_2 = x^2 - \pi x + 2.0886$$
$$p_3 = x^3 - 4.7124x^2 + 6.5262x + 2.4998$$
$$p_4 = x^4 - 2\pi x^3 + 13.3883x^2 - 11.0545x + 2.7903$$

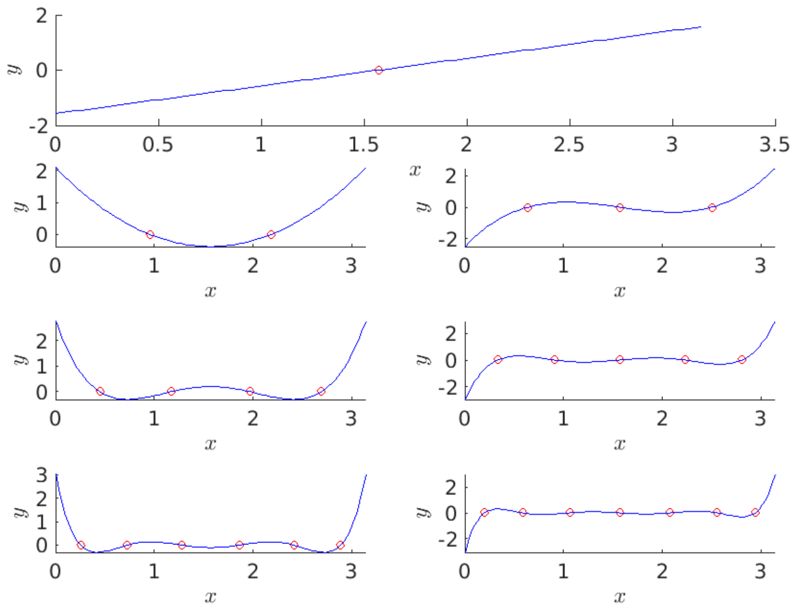


Figure 38: Plot of the first seven nodal polynomials.

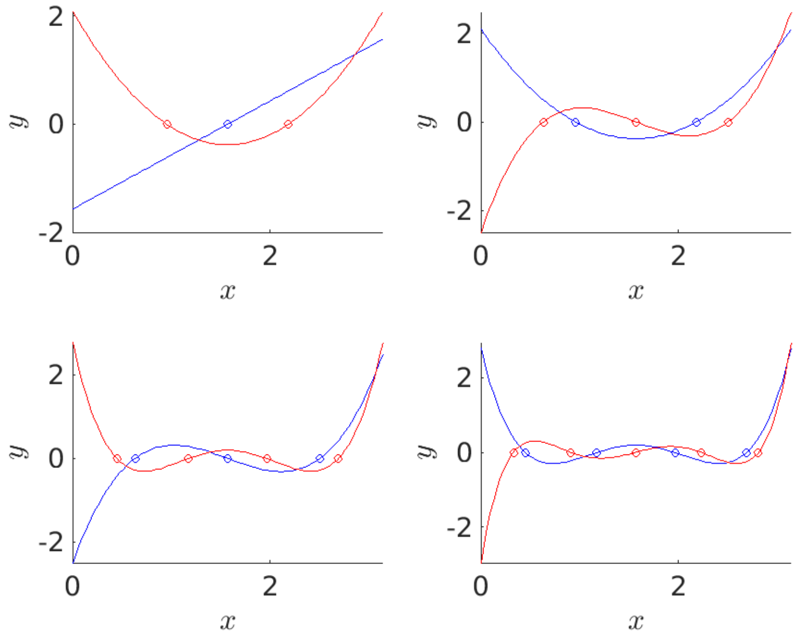


Figure 39: Plot of the interlaced roots.

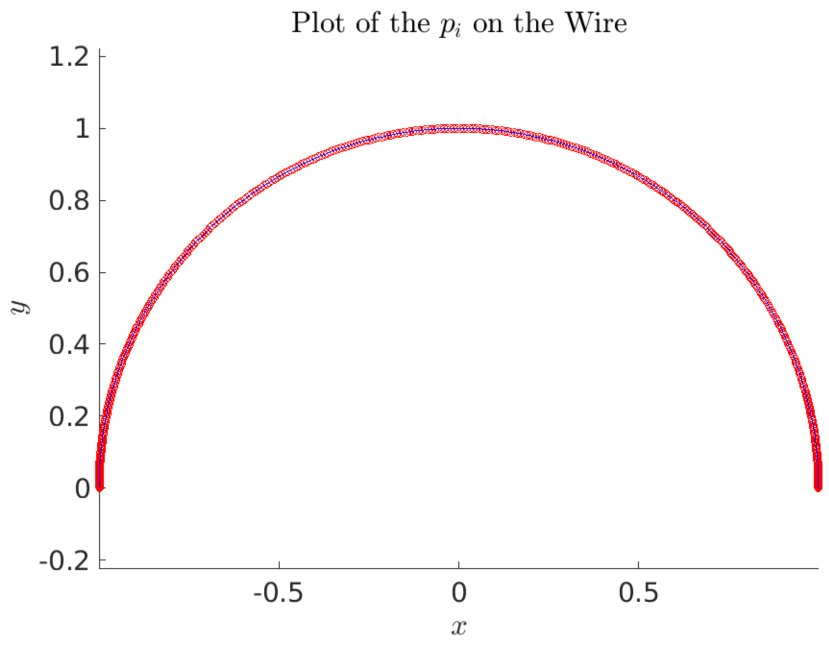


Figure 40: Plot of the p_i on the wire.

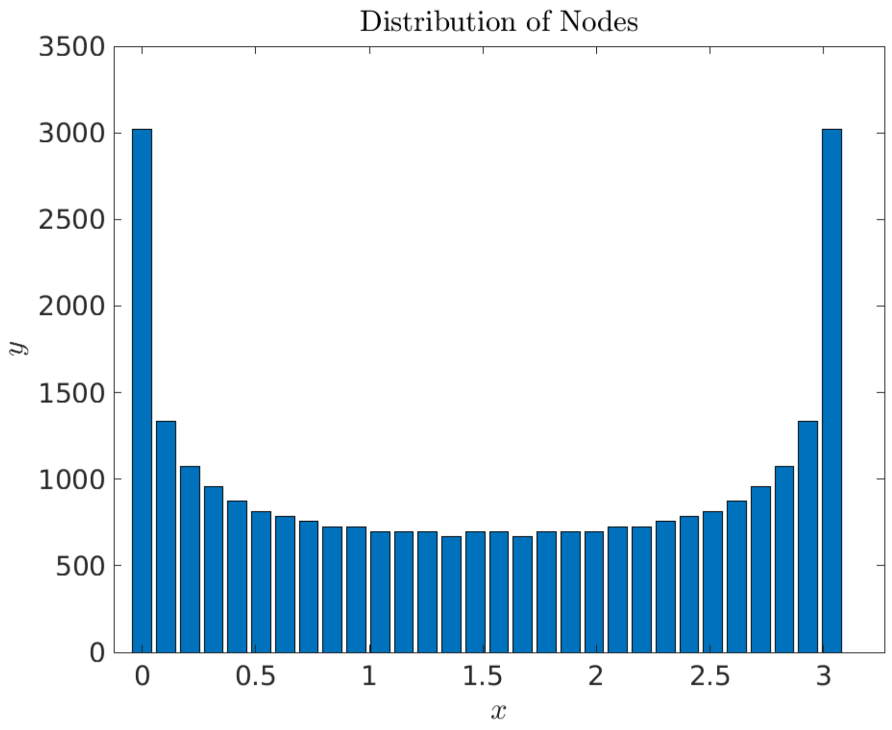


Figure 41: plot of the distribution of the nodes.

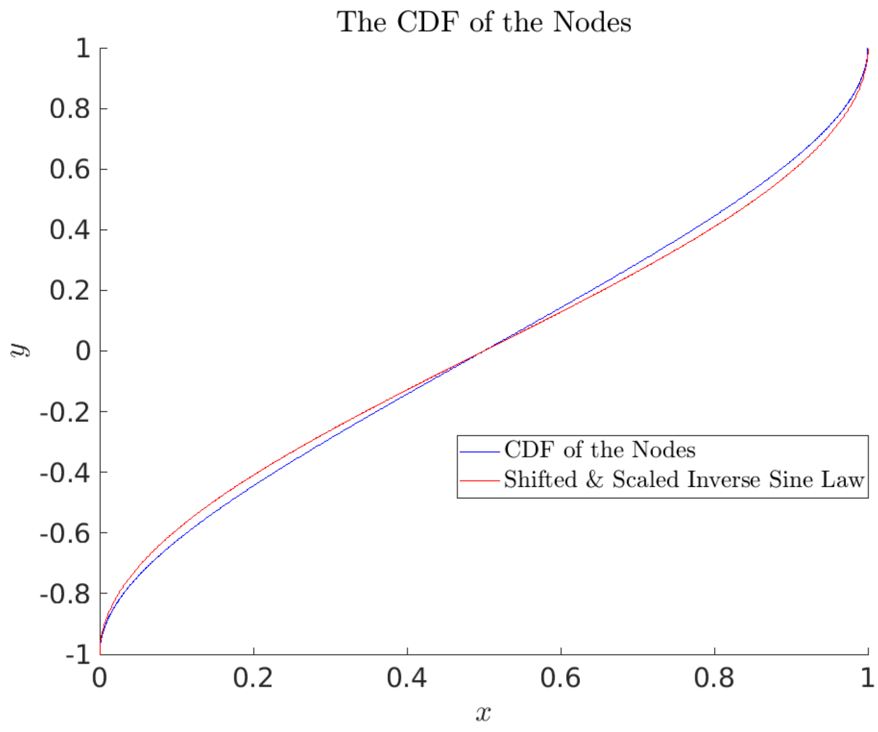


Figure 42: Plot of the CDF and a shifted and scaled inverse sine law.

What we can see from the plots is that the nodes definitely obey the inverse sine law, and in this case, it is shifted and scaled. Further, the nodes are indeed interlaced which suggests that the polynomials may be part of the classical orthogonal polynomials. I'm not sure if they satisfy a three-term recurrence relation. As for the energy, this is what I observed that

$$\begin{aligned} N &= 2000, \qquad u = 0.0823 \\ N &= 3000, \qquad u = 0.0824 \\ N &= 4000, \qquad u = 0.0825, \end{aligned}$$

which suggests that the minimal potential energy $u_N^* \to 0.083$ as $N \to \infty$. I'm not sure what conclusions I can draw. I did try a few things, though. The first thing I did was I tried writing u in terms of a Riemann sum so that I might express u in integral form, but the integrals that I obtained were singular. Even though I was able to form the Riemann sum, bad things happened, so this was as far as I was able to get. The code for the energy is below and the plot of the half circle are also below.

```
function u = energy(t)
    % output of method 1 = output of method 2

    % method 1
    N = length(t);
    s1 = log(sin(t));
    tt = t(:);
    c = 1/(2*pi*N^2);
    o = ones(N,1);
    s2 = 0.5*abs(tt*o' - o*tt');
    s2 = tril(s2);
    s2 = log(sin(s2(s2>0)));
    u = -c*(2*N*log(2) + sum(s1) + sum(s2));

    % method 2
%     p0 = [cos(0);sin(0)];
%     pN1 = [cos(pi);sin(pi)];
%     p = [cos(t);sin(t)];
%     s1 = log(vecnorm(p0 - p));
%     s2 = log(vecnorm(pN1 - p));
%     tt = t(:);
%     N = length(t);
%     o = ones(N,1);
%     c = 1/(2*pi*N^2);
%     s3 = 0.5*abs(tt*o' - o*tt');
%     s3 = tril(s3);
%     s3 = (log(sin(s3(s3>0))));
%     u = -c*(N*log(2) + sum(s1) + sum(s2) + sum(s3));
end
```

```
N = 100;
x = linspace(-1,1,N);
xx = linspace(1,1,N);
g = initial_guess(N,eps('double'),1); % much better guess
t = optimize(g,eps('double'),25);
u = energy(t);
pl = [cos(t) sin(t)]';

% plot of the wire with the nodes
figure
hold on
plot(pl(1,:),pl(2,:),'ro');
plot(pl(1,:),pl(2,:),'b-')
xlabel('$x$','interpreter','latex');
ylabel('$y$','interpreter','latex');
set(gca,'fontsize',20);
```

```
title('Plot of the $p_{i}$ on the Wire','interpreter','latex');
axis equal
```

Difficulties

The primary difficulties I experienced with this homework was for the determination of the polynomials in exercises 2 and 3. For exercise 1, I was able to formulate a logical way in which I could discern the polynomials. I first tried $a \neq b \neq 0$ in Maple to see if the polynomials produced from the roots were consistent with the Maple output. I quickly recognized that it was necessary that $a = b$. Then, I started from $a = b = 3$, and as I decremented, I finally reached $a = b = 0$, which produces Legendre polynomials. For exercises 2 and 3, however, this was not the case. I tried the same process, even modifying the argument of P to be $P_n^{(a,b)}(2x - \pi)$ for exercise 3, and different arguments also for exercise 2, but I could not determine what the polynomials were. In terms of the minimum energy, I figured I could use Riemann sums. This is what I obtained for exercise 3:

$$u^* = \lim_{N\to\infty} -\frac{\ln(2)}{\pi N} - \frac{1}{2\pi N}\sum_{i=1}^{N} \ln(\sin(\pi i/N))\frac{1}{N} - \frac{1}{4\pi}\sum_{i=2}^{N}\sum_{j=1}^{i-1} \ln(\sin((i-j)\pi/2N))\frac{1}{N^2}.$$

For the second sum, we quickly run into problems because for each $i = j$, we obtain singularities, which are indeed dangerous. If I were to leave the second in the way in which it was presented, I wouldn't have known how to represent it as a Riemann sum. Even with the expressions with the energy for the in exercises 1 and 2 gave me singular results.

Questions

1.) How should I think about the exercises above concerning the limiting behavior? Is rewriting the energy in terms of Riemann sums and integrals the best way to approach the problem at hand or should I use a combination of analytical methods and numerical methods, perhaps even literature?

2.) How would I determine the polynomials? How could I devise a logical way to deduce the identity of the polynomials in exercises 2 and 3 besides just trial and error, or is trial and error the way to go?

3.) Clearly there are values for the minimal energy. How can I express the energies in a way so as to avoid singularities in the Riemann sums?

4.) Is there a way to express the sum where $i > j$ in terms of one index or, is there a better way to rewrite the sum that makes it easy to discern a Riemann sum?

HOMEWORK 7

EXERCISE 1

(1) Prove that the trapezoid method (58) converges.

Theorem The *trapezoidal rule* is convergent.

Proof. Let us define the following problem

$$\frac{dx}{dt} = f(t, x), \quad x(0) = x_0, \quad 0 \le t \le T,$$

and let us define $y_n = y(t_n)$ to be the sequence of values produced by the trapezoidal rule where each time step t_n is $t_n = nh$, where $n = 0, \dots, N$. Further, we choose N such that the step-size h is given by $h = T/N$. Lastly, we assume that $x \in C^3[0, T]$. The solution to the ODE can be expressed as

$$x = x_0 + \int_0^t f(\tau, x(\tau))d\tau.$$

First, for a general subinterval $[t_n, t_{n+1}] \subset [0, T]$, we can express the solution to the ODE evaluated at t_n using the trapezoidal method with its appropriate error, given by

$$\begin{aligned} x(t_{n+1}) &= x(t_n) + \int_{t_n}^{t_{n+1}} f(\tau, x(\tau))d\tau && (27) \\ &= x(t_n) + \frac{h}{2}\big(f(t_n, x(t_n)) + f(t_{n+1}, x(t_{n+1}))\big) - \frac{x^{(3)}(\xi_n)h^3}{12}, \quad t_n < \xi_n < t_{n+1} && (28) \end{aligned}$$

where we will henceforth use $x_n = x(t_n)$. Likewise, the sequence y_n is generated from

$$y_{n+1} = y_n + \frac{h}{2}\big(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\big). \qquad (29)$$

By subtracting equation (29) from equation (28), we obtain the

$$\begin{aligned} e_{n+1} = e_n + \frac{h}{2}\big\{&[f(t_n, x_n) - f(t_n, y_n)] \\ &+ [f(t_{n+1}, x_{n+1}) - f(t_{n+1}, y_{n+1})]\big\} - \frac{x^{(3)}(\xi_n)h^3}{12} \end{aligned} \qquad (30)$$

where $e_{n+1} = x_{n+1} - y_{n+1}$ and $e_n = x_n - y_n$, which we define as *local errors*, and the *global error* we will define as

$$E = \|e\|_\infty = \sup_{1<n<N} |e_n|.$$

Next, because of our definition of the local and global errors, we take the absolute values of both sides of equation (30), we impose the triangle inequality, and we assume Lipshitz continuity of f in the second variable so as to obtain

$$|e_{n+1}| \leq |e_n| + \frac{Lh}{2}\{|e_n| + |e_{n+1}|\} + \frac{ch^3}{12},$$

where L is the Lipshitz constant and because of the EVT, we have an upper bound c such that $|x^{(3)}(\xi_n)| < c$. By collecting terms, we obtain

$$|e_{n+1}| \leq \left(\frac{1+\frac{1}{2}Lh}{1-\frac{1}{2}Lh}\right)|e_n| + \left(\frac{c}{1-\frac{1}{2}Lh}\right)\frac{h^3}{12}$$

By recursively solving for the e_{n+1} starting at $n = 0$, and thus by induction on n, we find that

$$|e_n| \leq \frac{h^3}{12}\left(\frac{c}{1-\frac{1}{2}Lh}\right)\sum_{k=0}^{n-1}\left(\frac{1+\frac{1}{2}Lh}{1-\frac{1}{2}Lh}\right)^k = \frac{ch^2}{12L}\left[\left(\frac{1+\frac{1}{2}Lh}{1-\frac{1}{2}Lh}\right)^n - 1\right],$$

using the geometric sum formula. Because we defined the global error as the sup-norm of all the errors, then naturally our error is given by

$$E \leq \frac{ch^2}{12L}\left[\left(\frac{1+\frac{1}{2}Lh}{1-\frac{1}{2}Lh}\right)^N - 1\right] = \frac{ch^2}{12L}\left[\left(1+\frac{Lh}{1-\frac{1}{2}Lh}\right)^{T/h} - 1\right].$$

By taking the limit as $h \to 0^+$ of the expression in the brackets, we have $\lim_{h\to 0^+}(1+\frac{Lh}{1-\frac{1}{2}Lh})^{T/h} = e^{LT}$ by using the natural logarithm and L'Hôspital's rule. Lastly, we take the limit as $h \to 0^+$ over all to get

$$\lim_{h\to 0^+} E \leq \frac{c}{12L}\left(e^{LT} - 1\right)\lim_{h\to 0^+} h^2 = 0.$$

Because the global error goes to zero as the step-size goes to zero, the trapezoidal rule is convergent. □

EXERCISE 2

(1) Consider the following two-step implicit scheme:

$$y_{n+2} - 3y_{n+1} + 2y_n = h\left(\frac{13}{12}f_{n+2} - \frac{5}{3}f_{n+1} - \frac{5}{12}f_n\right)$$

Here $f_k = f(t_k, y_k)$.

(a) Find the order of accuracy.

(b) Test the stability of the scheme by applying it to the trivial IVP:

$$\frac{dx}{dt} = 0, \quad x(0) = 1,$$

on $[0, 20]$ with $h = 0.1$, $h = 0.05$, and $h = 0.025$; for each step size initialize the method using exact values $y_0 = y_1 = 1$. Is the method stable?

Solution: To determine the order of the method, we utilize the following equation

$$\sum_{m=0}^{N} a_m x(t+mh) - h\sum_{m=0}^{N} b_m x'(t+mh) = \mathcal{O}(h^{k+1}),$$

where the order of the method is given by k and by convention, $a_N = 1$. We can make the following approximations: $y_{n+m} = x(t+mh)$ and $f(t_{n+m}, y_{n+m}) = x'(t+mh)$ for the implicit scheme. Thus, the scheme can be rewritten as

$$x(t+2h) - 3x(t+h) + 2(t) - h\left(\frac{13}{12}x'(t+2h) - \frac{5}{3}x'(t+h) - \frac{5}{12}x'(t)\right) = \mathcal{O}(h^{k+7}).$$

By expanding the left-hand side using Taylor theory at $h = 0$, we obtain

$$\begin{aligned}
&x(t) + 2x'(t)h + 2x''(t)h^2 + \frac{4x'''(t)}{3}h^3 - \\
&3x(t) - 3x'(t)h - \frac{3x''(t)}{2}h^2 - \frac{x'''(t)}{2}h^3 + \\
&2x(t) - \frac{13x'(t)}{12}h - \frac{13x''(t)}{6}h^2 - \frac{13x'''(t)}{6}h^3 - \\
&\frac{5x'(t)}{3}h + \frac{5x''(t)}{3}h^2 + \frac{5x'''(t)}{6}h^3 + \frac{5x'(t)}{12}h + \mathcal{O}(h^4),
\end{aligned}$$

and collecting like terms with respect to h gives us

$$-\frac{1}{2}x'''(t)h^3 + \mathcal{O}(h^4).$$

Thus, the order of the method is 2. In MATLAB, I wrote the following script in order to utilize the method with the above ODE and for the IVP $y' = -y$ with $y(0) = 1$.

```
function Hw_7_Ex_2
    clear;
    close all;
    clc;

    f = @(t,s) 0.*t.*s;
    lambda = -1;
    f1 = @(t,s) lambda*s;
    h = [0.1 0.05 0.025];

    figure
    for k = 1:3
        hold on
        [y1,t1] = ode_solver(f1,0,20,h(k),1,exp(lambda*h(k)));
        plot(t1,exp(lambda*t1),'b-');
        plot(t1,y1,'r-.');
        ylim([-2.5 2.5]);
        xlim([0 2.5]);
    end

    [y,t] = ode_solver(f,0,20,h(2),1,1);

    figure
    hold on
    plot(t,ones(size(t)),'b-')
    plot(t,y,'r-.')

end

function [y,t] = ode_solver(rhs,t0,tf,h,y0,y1)
    t = t0:h:tf;
    y = zeros(size(t));
    y(1) = y0;
    y(2) = y1;

    for k = 1:length(t)-2
        ynew = y(k+1) + rhs(t(k+1),y(k+1))*h;
        for j = 1:100
            ynew = 3*y(k+1) - 2*y(k)...
            + h*(13*rhs(t(k+2),ynew)/12 - 5*rhs(t(k+1),y(k+1))/3 - ...
                5*rhs(t(k),y(k))/12);
        end
        y(k+2) = 3*y(k+1) - 2*y(k)...
            + h*(13*rhs(t(k+2),ynew)/12 - 5*rhs(t(k+1),y(k+1))/3 - ...
                5*rhs(t(k),y(k))/12);
    end
end
```

In the code above, because we initialize the process with $y_0 = y_1 = 1$, I used Newton's Method to determine y_{n+2}, with the starting guess being one iteration of Euler's method. For the above ODE, however, when plotting the solution of the IVP against the method, agreement is achieved for all h. So, numerically, this ODE is not a very good ODE to test whether or not the method is stable, hence the second IVP mentioned. For this example, very clearly we can see that the method is unstable. As we decrease the step-size, the approximation becomes further away from the true solution.

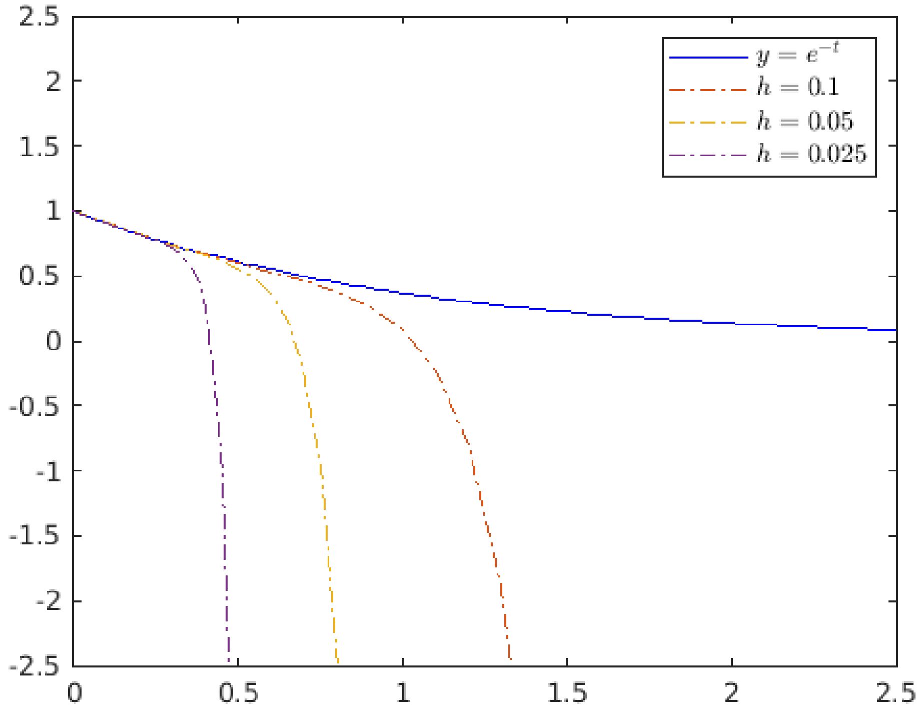


Figure 43: A plot of the solution to the IVP $y' = y,\ y(0) = 1$ against the two step implicit method. Clearly the method diverges from the true solution as step-size decreases, which means it's an unstable method.

EXERCISE 3

(3) Find the orders of the Adams-Bashford methods for $N = 2, 3, 4, 5$.

Solution: In the same manner as that in Exercise (2), I computed the orders. A table of results is given below.

$$\begin{aligned}
N = 2, &\quad \frac{5}{12}x^{(3)}(t)h^3 + \mathcal{O}(h^4), \quad \text{Order} = 2\\
N = 3, &\quad \frac{3}{8}x^{(4)}(t)h^4 + \mathcal{O}(h^5), \quad \text{Order} = 3\\
N = 4, &\quad \frac{251}{720}x^{(5)}(t)h^5 + \mathcal{O}(h^6), \quad \text{Order} = 4\\
N = 5, &\quad \frac{95}{288}x^{(6)}h^6 + \mathcal{O}(h^7), \quad \text{Order} = 5.
\end{aligned}$$

Generally, we can say that for an N-step Adams-Bashford method, the order is N.

EXERCISE 4

Find an implicit three-step method of order six (it is unique). Is it stable?

Solution To begin, we use

$$\sum_{m=0}^{3} a_m x(t + mh) - h \sum_{m=0}^{3} b_m x'(t + mh) = \mathcal{O}(h^{k+1}),$$

where $a_3 = 1$ by convention. After expanding each term into a Taylor expansion at $h = 0$ and collecting the terms according to h, we have

$$x(t)(a_0 + a_1 + a_2 + 1) + x'(t)(a_1 + 2a_2 - b_0 - b_1 - b_2 - b_3 + 3)h+$$
$$\frac{1}{2}x^{(2)}(t)(a_1 + 4a_2 - 2b_1 - 4b_2 - 6b_3 + 9)h^2 + \frac{1}{6}x^{(3)}(t)(a_1 + 8a_2 - 3b_1 - 12b_2 - 27b_3 + 27)h^3$$
$$\frac{1}{24}x^{(4)}(t)(a_1 + 16a_2 - 4b_1 - 32b_2 - 108b_3 + 81)h^4 + \frac{1}{120}x^{(5)}(t)(a_1 + 32a_2 - 5b_1 - 80b_2 - 405b_3 + 243)h^5$$
$$\frac{1}{720}x^{(6)}(t)(a_1 + 64a_2 - 6b_1 - 192b_2 - 61458_3 + 729)h^2 + \mathcal{O}(h^7).$$

Because we want an order of six, we need to set the expressions in the parentheses equal to zero. In doing so, we obtain the following linear system:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & -1 & -1 & -1 & -1 \\ 0 & 1 & 4 & 0 & -2 & -4 & -6 \\ 0 & 1 & 8 & 0 & -3 & -12 & -27 \\ 0 & 1 & 16 & 0 & -4 & -32 & -108 \\ 0 & 1 & 32 & 0 & -5 & -80 & -405 \\ 0 & 1 & 64 & 0 & -6 & -192 & -1458 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -3 \\ -9 \\ -27 \\ -81 \\ -243 \\ -729 \end{bmatrix}$$

In solving the linear system, we find that the method is

$$y_{n+3} + \frac{27}{11} y_{n+2} - \frac{27}{11} y_{n+1} - y_n = h\left(\frac{3}{11} f_{n+3} + \frac{27}{11} f_{n+2} + \frac{27}{11} f_{n+1} + \frac{3}{11} f_n \right).$$

The method is not stable. To check, we find the roots of characteristic equation, given by

$$\sum_{m=0}^{3} a_m \lambda^m = \lambda^3 + \frac{27}{11}\lambda^2 - \frac{27}{11}\lambda - 1 = 0,$$

and see if the root condition holds, as described in this[9] paper. The roots are

$$\lambda = 1, -\frac{19 + 4\sqrt{15}}{12}, -\frac{19 - 4\sqrt{15}}{11},$$

for which only $\lambda = 1$ doesn't fail. So, the root condition fails overall, meaning that the method is not stable.

[9] `https://www3.nd.edu/~zxu2/acms40390F13/Lec-5.10.pdf`