AE370: Homework 2

[redacted]

1 Problem 1

[redacted]

1.1 Part A

Using Matlab, the **A** matrix was found for both the monomial and Lagrange basis for $n = 3, 6, 12, 24, 48, 96$. For the monomial basis, a custom script was written that built up the individual elements of the matrix by following the appropriate pattern. The matrix for the Lagrange basis was simply the identity matrix so the "eye" function was used. Each of these matrices was an $(n+1)$ by $(n+1)$ matrix. The code to develop each set of condition numbers is shown in 1.1. The condition numbers for the successive "n's" for the monomial basis are shown in 1. The condition numbers for the successive "n's" for the Lagrange basis are shown in 2.

$$\text{cond}(\mathbf{A}_{\text{monomial}}) \text{ for n=3,...,96} = \begin{bmatrix} 9.87 \times 10^2 \\ 36.06 \times 10^5 \\ 67.81 \times 10^9 \\ 27.22 \times 10^{18} \\ 11.72 \times 10^{19} \\ 39.01 \times 10^{20} \end{bmatrix} \tag{1}$$

$$\text{cond}(\mathbf{A}_{\text{Lagrange}}) \text{ for n=3,...,96} = \begin{bmatrix} 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \end{bmatrix} \tag{2}$$

```
%% P1a
clc
clear
% set up the n's we want and the chosen range
n = [3 6 12 24 48 96];
range = [0,1];
```

```
% Call the function to find out the condition numbers for a
    monomial basis
linConNum = vpa(linConNums(n,range))
% Call the function to find out the condition numbers for the
    lagrange
% basis
lagConNum = vpa(lagConNums(n,range))

% Problem 1a
function [linConNum] = linConNums(nvec,range)
% linConNums finds the condition numbers of monomial basis
    matrix
k = 1;
    % iterates through all the n values, creates a equally
        space set of
    % points, fills the A matrix out, then evaluates condition
        number
    while k < length(nvec)+1
        n = nvec(k);
        int = linspace(range(1),range(2),n+1);
        A = zeros(length(int),length(int));
        i = 1;
        j = 1;
        while i < size(int,2) + 1
            while j < length(int) + 1
                A(i,j) = int(i)^(j-1);
                j = j+1;
            end
            i = i+1;
            j = 1;
        end
        linConNum(k) = cond(A);
        k = k+1;
    end
end

% p1a
function [lagConNum] = lagConNums(nvec,range)
% lagConNums finds the condition numbers of lagrangian basis
k = 1;
    % simply iterates through, but it is all 1
    while k < length(nvec)+1
        n = nvec(k);
        int = linspace(range(1),range(2),n+1);
        A = eye(length(int),length(int));
```

```
        lagConNum(k) = cond(A);
        k = k+1;
    end
end
```

1.2 Part B

This problem required us to plot the six basis functions for both the monomial and Lagrange basis on a set of 1000 points from 0 to 1. The code to perform this is shown in 1.2. The graph of the monomial functions evaluated across all 1000 points from 0 to 1 is shown in 1. The graph of the Lagrange functions evaluate across all 1000 points from 0 to 1 is shown in 2.

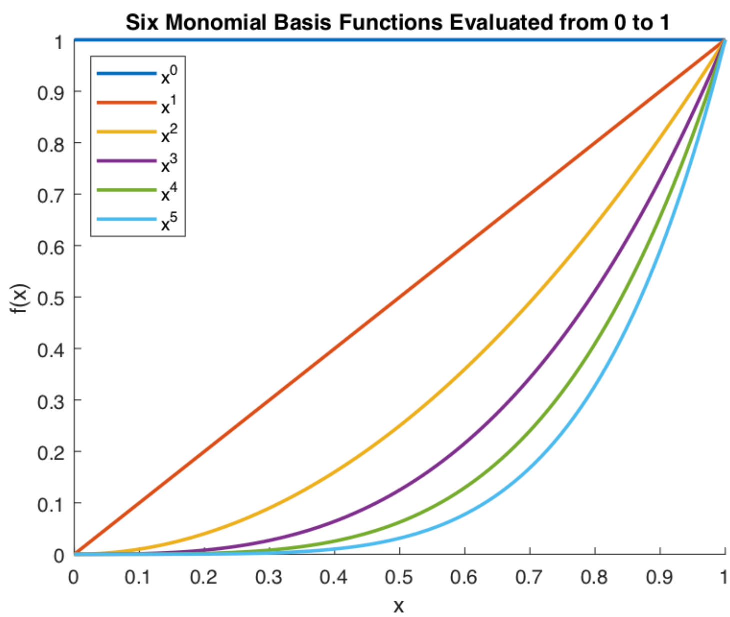

Figure 1: Monomial basis functions

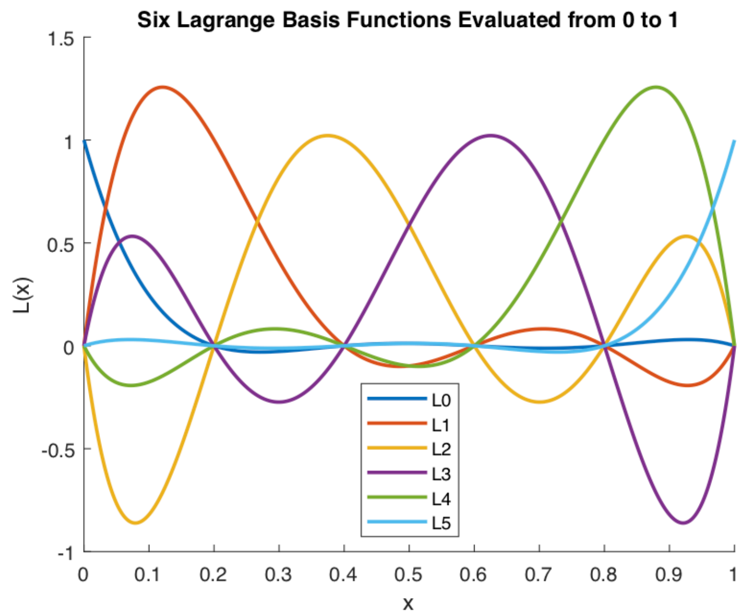

Figure 2: Lagrangian basis functions

In order to minimize condition number, one wants to choose basis that are as linearly independent as possible. The monomial basis functions are very similar shapes. This means that it makes it difficult to make good approximations to functions based off linear combinations of these functions. Each of the Lagrange basis functions are very different from each other. This should allow them to approximate functions much more affectively.

```
%% P1b1
clear
clc
% set up the givens
n = 5;
range = [0,1];
fineness = 100;
% get the data for the plots
monoData = monoPlot2(n,range,fineness);
% create the plot range
plotrange = linspace(range(1),range(2),fineness);
% plot the data
figure(); hold on
for i = 1:n+1
    plot(plotrange,monoData(i,:),'linewidth',1.75);
end
hold off
title('Six Monomial Basis Functions Evaluated from 0 to 1')
xlabel('x')
ylabel('f(x)')
```

```
legend('x^0','x^1','x^2','x^3','x^4','x^5','location','northwest');

%% P1b2
clear
clc
% given
n = 5;
% define some range
range = [0,1];
% number of points to plot on
fineness = 1000;
% call the lagrange function to find this stuff
lagData = lagPlot3(n,range,fineness);
% create the points that everything is being evaluated at
plotrange = linspace(range(1),range(2),fineness);
% plot
figure(); hold on
for i = 1:n+1
    plot(plotrange,lagData(i,:),'linewidth',1.75);
end
hold off
title('Six Lagrange Basis Functions Evaluated from 0 to 1')
xlabel('x')
ylabel('L(x)')
legend('L0','L1','L2','L3','L4','L5','location','northeastoutside');

% p1b
function [monoPlotData] = monoPlot2(n,range,m)
% creates the data for the monomial basis
    range = linspace(range(1),range(2),m);
    for i=1:n+1
        for j=1:length(range)
            monoPlotData(i,j) = range(j)^(i-1);
        end
    end
end

% p1b
function [lagPlotData] = lagPlot3(n,range,m)
    % this function creates the data for the lagrange basis functions. It
    % creates 'n+1' functions. The interval points are defined by the
```

```
    % 'range' and 'n+1'. The fineness is determind by 'm' allows the basis
    % functions to be graphed along
    % the range in a nice manner. 'n' is an integer, 'rang'e is the
    % bounding points in a array, and 'm' is the number of points to plot
    % on through the 'range'

    % set up the interval, n+1 evenly spaced points
    int = linspace(range(1),range(2),n+1);
    % set up 'm' # of points to evaluate at
    range = linspace(range(1),range(2),m);
    for i=1:length(int)
        f = @(x) 1;
        % iterate through the 6 points
        for k=1:length(int)
            % if j ==  i, then num and den are just one again
            if i~=k
                f = @(x) f(x)*(x-int(k))/(int(i)-int(k));
            end
        end
        % once the function is made, iterate through the 1000 points and
        % store them
        for j=1:length(range)
%                 lagPlotData(i,j) = subs(num,x,range(j))/subs(den,x,range(j));
            lagPlotData(i,j) = f(range(j));
        end
    end
end
```

1.3 Part C

Condition is dependant on the basis that are used, so choosing different points should not help how bad the monomial basis is. As done at the beginning of class, the matrices

$$\mathbf{A}_1 = \begin{bmatrix} 1 & 1 \\ 0 & \epsilon \end{bmatrix} \quad \text{and} \quad \mathbf{A}_2 = \begin{bmatrix} 1 & \epsilon \\ 0 & 1 \end{bmatrix}$$

were considered when looking at conditioned problems. When ϵ was very small, $\mathbf{A}_1$ had two basis vectors that were very similar. This caused the inability for it to cover a space effectively. One must know the value of ϵ to very accurate decimal places to the point that it is nearly useless. On the other hand, $\mathbf{A}_2$ could cover a space much better with its columns and it can do this across a range of values for ϵ. In the case of using the monomial basis, it is similar to using $\mathbf{A}_1$ to solve problems. The points we would have to choose to make it an effective basis would be such a small set, it is nearly useless. Chebyshev points wouldn't make this problem any better because it is an issue related to the basis functions themselves.

2 Problem 2

2.1 Part A

The function

$$f(x) = \frac{1}{1+x^2}$$

was interpolated in equispaced points using the Lagrange basis for $n = 5, 10, 15, 20$ where n is the order of the polynomial interpolant. The results of this process are plotted in 3. The end behavior of the polynomials is extreme, and the graph has been y-axis limited for ease of comparison to 2.1. The code for this is attached in 2.2.

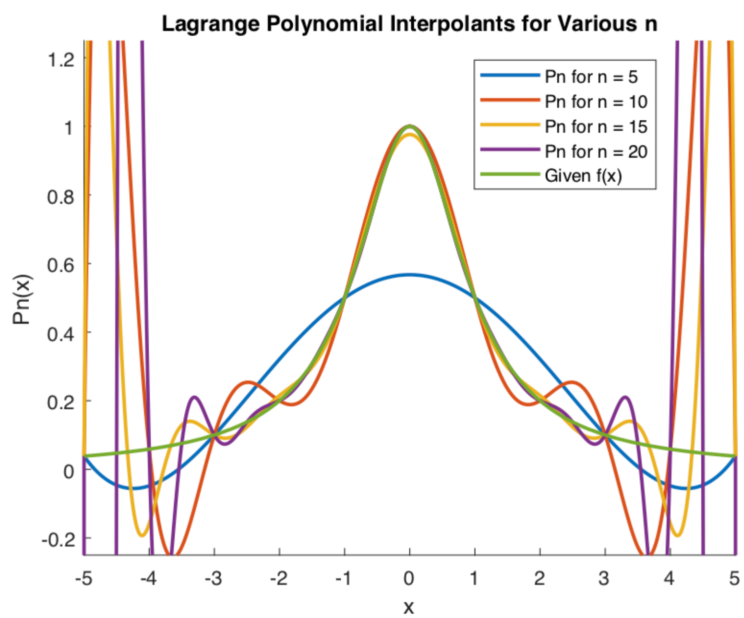

Figure 3: Polynomial interpolants in equispaced points of degree n against 2.1

2.2 Part B

2.1 was again interpolated, but in Chebyshev points using the Lagrange basis for $n = 5, 10, 15, 20$ where n is the order of the polynomial interpolant. The results of this process are plotted in 4. The end behavior of the polynomials is much more controlled. The code for this is attached in 2.2.

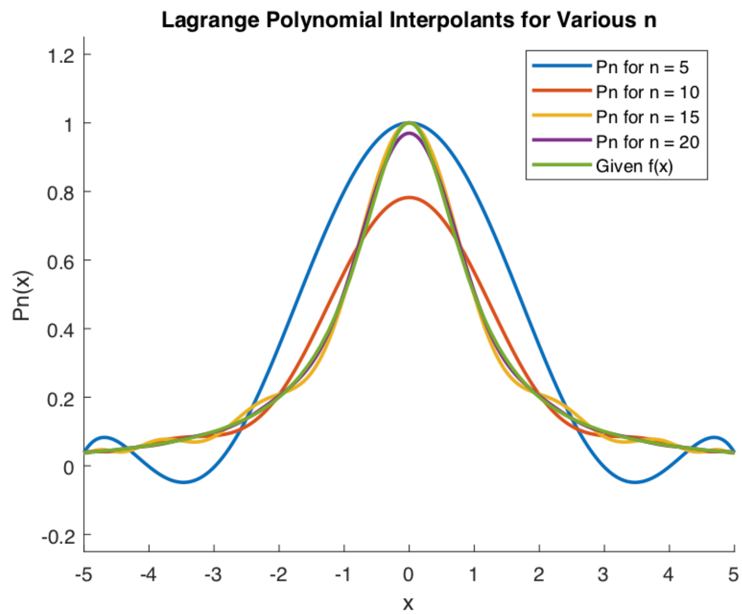

Figure 4: Polynomial interpolants in Chebyshev points of degree n against 2.1

```
%% P2a
clear
clc
% given function
f = @(x) 1/(1+x.^2);
% define range
range = [-5,5];
% define the vector of the # of points we want
n = [5 10 15 20];
% define some 'fineness' of the mesh we are plotting over and
    make the mesh
fine = 1000;
evalrange = linspace(range(1),range(2),fine);
%call the function to get the data
pdata = linPolyMaker1(n,range,fine,f)
% plot the data
hold on
for i = 1:length(n)
    plot(evalrange,pdata(i,:),'linewidth',1.75);
end
%plot the given func
fplot(f,range,'linewidth',1.75)
ylim([-.25,1.25]);
hold off
title('Lagrange Polynomial Interpolants for Various n')
```

```matlab
xlabel('x')
ylabel('Pn(x)')
legend('Pn for n = 5','Pn for n = 10','Pn for n = 15','Pn for n
    = 20','Given f(x)','location','northeastoutside');

% p2a
function [polyPlotData] = linPolyMaker1(n,range,m,f)
    % this function creates the polynomial interpolant and
       evaluates it for
    % the points on the range

    % set up 'm' # of points to evaluate the interpolant at
    evalrange = linspace(range(1),range(2),m);
    % set up the parent for loop to iterate through the
       different n's
    for i = 1:length(n)
    % create the interpolation points and the polynomial
       interpolant
    % function
    int = linspace(range(1),range(2),n(i)+1);
    Pn = @(x) 0;
    % for loop that builds the interpolant
        for j=1:length(int)
            % create a scalar for the basis
            d = f(int(j));
            % create the unit lagrange basis
            L = @(x) 1;
            % iterate through the 6 points
            for k=1:length(int)
                % if j ==  i, then num and den are just one
                   again
                if j~=k
                % build the lagrange basis function
                L = @(x) L(x)*(x-int(k))/(int(j)-int(k));
                end
            end
            % append the lagrange basis function to the total
               interpolant
            Pn = @(x) Pn(x) + d*L(x);
        end
        for d=1:length(evalrange)
            polyPlotData(i,d) = Pn(evalrange(d));
        end
    end
end
```

```
%% P2b
clear
clc

% given function
f = @(x) 1/(1+x.^2);
% define range
range = [-5,5];
% define the vector of the # of points we want
n = [5 10 15 20];
% define some 'fineness' of the mesh we are plotting over
fine = 1000;
% create the mesh
evalrange = linspace(range(1),range(2),fine);
% get the data
pdata = chebPolyMaker1(n,range,fine,f);
% plot the data
hold on
for i = 1:length(n)
    plot(evalrange,pdata(i,:),'linewidth',1.75);
end
ylim([-.25,1.25]);
fplot(f,range,'linewidth',1.75)
hold off
title('Lagrange Polynomial Interpolants for Various n')
xlabel('x')
ylabel('Pn(x)')
legend('Pn for n = 5','Pn for n = 10','Pn for n = 15','Pn for n
    = 20','Given f(x)','location','northeastoutside');

function [polyPlotData] = chebPolyMaker1(n,range,m,f)
    % this function creates the polynomial interpolant and
       evaluates it for
    % the points on the range

    % set up 'm' # of points to evaluate the interpolant at
    evalrange = linspace(range(1),range(2),m);
    % set up the parent for loop to iterate through the
       different n's
    for i = 1:length(n)
    % create the interpolation points and the polynomial
       interpolant
    % function
```

```
    int = chebSpace(range,n(i)+1);
    Pn = @(x) 0;
    % for loop that builds the interpolant
        for j=1:length(int)
            % create a scalar for the basis
            d = f(int(j));
            % create the unit lagrange basis
            L = @(x) 1;
            % iterate through the 6 points
            for k=1:length(int)
                % if j ==  i, then num and den are just one again
                if j~=k
                % build the lagrange basis function
                L = @(x) L(x)*(x-int(k))/(int(j)-int(k));
                end
            end
            % append the lagrange basis function to the total interpolant
            Pn = @(x) Pn(x) + d*L(x);
        end
        for d=1:length(evalrange)
            polyPlotData(i,d) = Pn(evalrange(d));
        end
    end
end

function [chebPoints] = chebSpace(range,n)
    for i=1:n+1
        chebPoints(i) = -(max(range)-min(range))*cos((i-1)*pi/n)/2;
    end
end
```

2.3 Part C

The error bound discussed in class is

$$\max_{a\leq x\leq b}\left|f(x)-\sum_{i=0}^{n}c_i b_i(x)\right| \leq \left[\max_{a\leq r\leq b}\frac{f^{(n+1)}(r)}{(n+1)!}\right]\left[\max_{a\leq r\leq b}\prod_{j=0}^{n}(x-x_j)\right] \tag{3}$$

The only part of this inequality relevant to the discussion of how the chosen interpolation points achieves convergence is the third part involving the product operator. This part's value is dependant on the points that we choose to interpolate from. The second part is related to the function and is not dependant on the points we choose. In order to minimize the error between the function and our approximation on the left side, we must minimize the value of the product between the chosen points and interpolation points.

It is useful to choose an example range and interpolate using equispaced and Chebyshev points to see the benefits that the Chebyshev distribution brings. The range that will be inspected will be $x = [-3, 3]$ with $n = 10$. This will generate 11 points, as shown in 2.3.

$$x_{Eq} = \begin{bmatrix} -3.00 & -2.40 & -1.80 & -1.20 & -0.60 & 0.00 & 0.60 & 1.20 & 1.80 & 2.40 & 3.00 \end{bmatrix}$$

$$x_{Ch} = \begin{bmatrix} -3.00 & -2.85 & -2.43 & -1.76 & -0.93 & 0.00 & 0.93 & 1.76 & 2.43 & 2.85 & 3.00 \end{bmatrix}$$

I created a Matlab function that automated the process of finding a maximum across 100 points in the chosen range. This code can be seen in 2.3. After evaluating the product for 100 points in $[-3, 3]$, the two values below were found

$$\max_{\text{Equispaced}} = 1508.1$$

$$\max_{\text{Chebyshev}} = 341.6$$

Adding more points within the range to evaluate at did not yield much difference between successive maximums so 100 points was deemed satisfactory for this example. What this shows is that the choice of Chebyshev points can minimize the maximum error between function and polynomial interpolant by nearly five times regardless of the function being interpolated for $n = 5$. This is a HUGE change. When taken to the extreme ($n > 100$), the error grows much faster for the equispaced than the Chebyshev points. I would say that the reason Chebyshev can achieve convergence along the function is that the error simply grows much slower due to the choice of points being more clustered near the endpoints.

```matlab
%% 2c
clear
clc
n = 10;
range = [-3,3];
fineness = 100;
xeq = linspace(range(1),range(2),n+1);
xcheb = chebSpace([range(1),range(2)],n);
maxeq = prodDiff(range,xeq,fineness)
maxcheb = prodDiff(range,xcheb,fineness)

function [chebPoints] = chebSpace(range,n)
    for i=1:n+1
        chebPoints(i) = -(max(range)-min(range))*cos((i-1)*pi/n
           )/2;
    end
end

function [maximum] = prodDiff(range,points,fine)
range = linspace(range(1),range(2),fine);
for i=1:length(range)
```

```
    prodDiffMat(i) = 1;
    for j=1:length(points)
        prodDiffMat(i) = prodDiffMat(i)*(range(i)-points(j));
    end
end
maximum = max(prodDiffMat);
end
```