Minh-Thi Nguyen
MAT321: Numerical Methods
Professor Nicolas Boumal
Due: 7 December 2018

MAT-APC 321: Numerical Methods Homework 4: Tridiagonalization and Interpolation

1 The Householder algorithm for tridiagonalization

We use the Householder algorithm as a guide to construct our implementation of the tridiagonalization of the symmetric matrix $A \in R^{n\times n}$. We observe that the orthogonal matrices Q_k have the structure:

$$Q_k = \begin{bmatrix} I_k & 0 \\ 0 & H_k \end{bmatrix} \tag{1}$$

where H_k is an $(n-k)\times(n-k)$ matrix. Thus, we see that, from the lecture notes, with every $Q_k^T A^{(k-1)} Q_k$ multiplication, the lower $n-k+1$ elements of the kth column and the right $n-k+1$ elements of the kth row become 0. Furthermore, because of the I_k structure in the top left of Q_k, the $Q_k^T A^{(k-1)}$ only affects the structure of A in the $k+1$ to n rows and k to n columns due to the multiplication of the H_k matrix and the zeros in the matrix of $A^{(k-1)}$. Then, we also observe that multiplying that by Q_k changes all the columns of $A^{(k-1)}$ but only the lower $n-k$. Nevertheless, because A is symmetric, we observe that it only changes then the k to n columns. Furthermore, to save computational efficiency, we observe that for a reflector $H = I_k - 2uu^T$, we can exploit the structure of H with the matrix product $HM = M - 2u(u^T M)$. Thus, we implement our tridiagonalization algorithm shown below:

```
1  function[T] = tridiagonalization(A)
2
3  %find the size of A
4  n = size(A,1);
5
6  %loop over the iteration from Householder algorithm
7  for k = 1:(n-2)
8
9      %create vector x, extracted from A_k-1
10     x = A(k+1:n,k);
11
12     %Built the H matrix
13     v = sign(x(1))*norm(x,2)*eye(size(x)) + x;
14     %normalize
15     v = v/norm(v,2);
16
17     %computationally expensive matrix multiplication
18 %     H = eye(n-k) - 2*v*v';
19 %     Q = eye(n);
20 %     Q(k+1:n,k+1:n) = H;
```

```
%     B = Q'*B*Q;

%Q'A
A(k+1:n,k:n) = A(k+1:n,k:n) - 2* v*(v'*A(k+1:n,k:n));
A(k+2:n,k) = 0; %make exactly 0

%Q'AQ
A(k:n,k+1:n) = A(k:n,k+1:n) - 2 * (A(k:n,k+1:n)*v)*v';
A(k, k+2:n) = 0; %make exactly 0

end

%update final matrix after tridiagonalization
T = A;
end
```

We observe that our algorithm follows the following steps, with corresponding flop count:

Algorithm 1 Household Tridiagonalization

1: A
2: **for** $k = 1 : n-2$ **do** ▷ iterate $n-2$ times
3: $x \leftarrow A_{k+1:n,k}$
4: $v \leftarrow sign(x)\|x\|_2 e_1$
5: $v \leftarrow v/\|v\|$ ▷ flop count: $3(n-k)$
6: $A_{(k+1):n,k:n} \leftarrow A_{(k+1):n,k:n} - 2v(v^T A_{(k+1):n,k:n})$ ▷ flop count: $4(n-k)(n-k+1)$
7: $A_{(k+2:n,k)} = 0$ ▷ make exactly 0 to make exactly Tridiagonal
8: $A_{k:n,k+1:n} \leftarrow A_{k:n,k+1:n} - 2(A_{k:n,k+1:n}v)v^T$ ▷ flop count: $4(n-k)(n-k+1)$
9: $A_{(k+2:n,k)} = 0$ ▷ make exactly 0 to make exactly Tridiagonal
10: **end for** ▷ total flop count for each loop: $8(n-k)(n-k+1) + 3(n-k)$

In Step 6, the $v^T A_{(k+1):n,k:n}$ (vector-Matrix product) operation costs $2(n-k)(n-k+1)flops$ and $2v(v^T A_{(k+1):n,k:n})$ costs $(n-k)(n-k+1)flops$ and $A_{(k+1):n,k:n} - 2v(v^T A_{(k+1):n,k:n})$ costs $(n-k)(n-k+1)flops$, so the total flop count is $2(n-k)(n-k+1) + (n-k)(n-k+1) + (n-k)(n-k+1) = 4(n-k)(n-k+1)$. The reasoning is similar in step 8. Therefore, the total flop count for the entire algorithm is: $\sum_{k=1}^{n-2} 8(n-k)(n-k+1) + 3(n-k) \approx \sum_{k=1}^{n-2} 8(n-k)^2 + 3(n-k) = \sum_{n-k=2}^{n-1} 8(n-k)^2 + 3(n-k) = \sum_{i=2}^{n-1} 8i^2 + 3i = 8\sum_{i=2}^{n-1} i^2 + 3\sum_{i=2}^{n-1} i = 8(n*(n+1)*(2n+1)/6) + 3(n*(n+1)/2) \approx 8n^3/3 + 3n^2/2$. Therefore, our algorithm costs $O(n^3) \approx \boxed{8n^3/3}$ flops.

We test our algorithm with matrix A described by the problem set.

```
% create symmetric A matrix
A = magic(8);
A = A + A';

%find eigenvalues of A
e = sort(eig(A));

% tridiagonalize matrix A given algorithm
T = tridiagonalization(A);
```

```
10
11 % print T
12 T
13
14 %find eigenvalues of new tridiagonal matrix
15 e1 = sort(eig(T));
16
17 %print the error of eigenvalues to 16 digits
18 fprintf('%.16e\n',e1-e);
```

Which outputs:

```
    T =

  128.0000 -169.5759         0         0         0         0         0         0
 -169.5759  341.9085  192.0672         0         0         0         0         0
         0  192.0672  141.8625 -125.9129         0         0         0         0
         0         0 -125.9129  -91.7710   -0.0000         0         0         0
         0         0         0   -0.0000    0.0000   -0.0000         0         0
         0         0         0         0   -0.0000    0.0000   -0.0000         0
         0         0         0         0         0    0.0000    0.0000   -0.0000
         0         0         0         0         0         0   -0.0000    0.0000
```

The error values are (which are accurate to machine precision):

```
e1 - e =

-1.4210854715202004e-13
3.0938408587235525e-15
-1.7825459064483956e-15
3.6050233547428745e-15
9.6245711270561099e-15
1.2034375263556718e-14
-2.8421709430404007e-14
1.1368683772161603e-13
```

2 Evaluating Lagrange Interpolants

(a) We implement the function using the barycentric formula as shown below:

```
1 function[q] = interp_lagrange(x, f, t)
2
3 %x = interpolation points
4 %f = function values corresponding to interpolation points
5 %t = data points
6
7 n = length(x)-1; %total number of x
8 m = length(t); %total number of data points
```

```
L = ones(1,m) %Lagrange basis polynomials

q1 = zeros(1,m); %interpolation results numerator
q2 = zeros(1,m); %interpolation results denominator
q = zeros(1,m); %interpolation results final polynomial vector

dw = ones(1,n); %find the weighted values

%loop to evaluate weighted values
for j = 1:n
    for k = 1:n
        if j~=k
            dw(j) = dw(j)*1/((x(j)-x(k)));
        end
    end
end

%consider the case where t coincides with x
exact = zeros(size(t));

%loop to determine the final polynomial vector using barycentric form
for i=1:n

    diff = t-x(i);

    %find w_i/(x-x_i)
    L = dw(i)./diff;

    %summation of numerator and denominator terms
    q1 = q1+f(i)*L;
    q2 = q2 + L;

    %consider the case where t coincides with x(i)
    exact(diff==0) = 1;
end

%final result for function
q = q1./q2;

%consider case where t coincides with x
jj = find(exact);
q(jj) = q(exact(jj));

end
```

We notice that when the t_i's coincide with the x_k's (but not exactly equal), the quotient of the $w_k/(t_i - x_k)$ becomes very large, because the same inaccuracy appears in the numerator and denominator of the Barycentric form, the inaccuracies cancel out and the formula remains stable overall. Nevertheless, we modify the code to incorporate the case where if $t_i = x_k$ using Berrut and Trefethen. When $t_i = x_k$, the values of the interpolant will come out as not-a-number. Thus, we can resolve this issue by adding a new variable "exact" to the kernel of the code to nake the formula reliable.

(b) We copy in the function for Matlab's polynomial interpolation and regression:

```
function q = interp_lagrange_poly(x, f, t)
n = length(x)-1;
p = polyfit(x, f, n);
q = polyval(p, t);
end
```

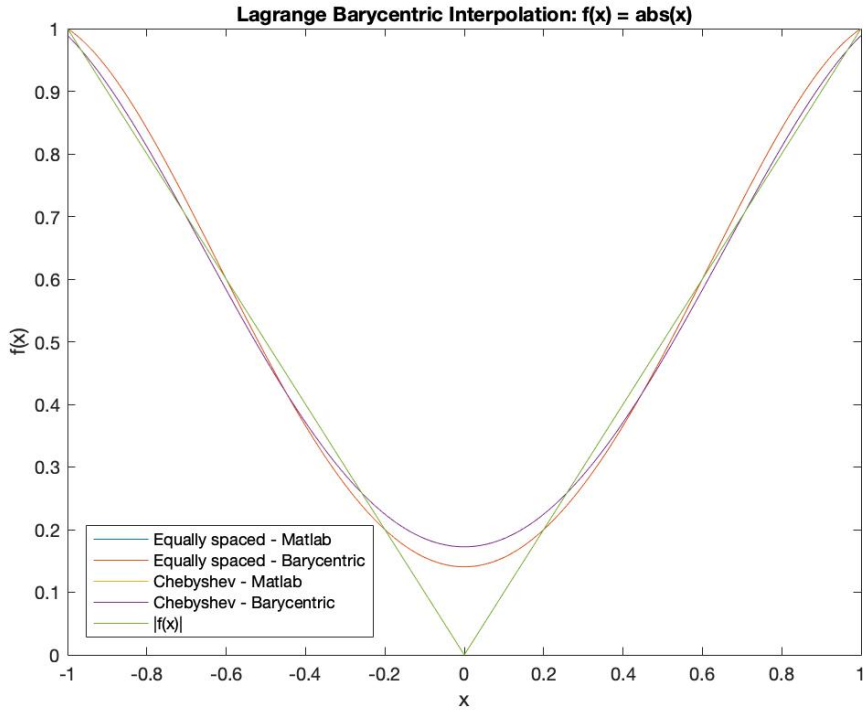
(c) We now test both methods to approximate $f(x)$ over the interval $[a, b] = [-1, 1]$ given the evaluation point distribution of t, for the given values of n.
Our implementation of the test is shown in the code below:

```
%set initial conditions for a, b, and t
a = -1;
b = 1;
t = linspace(a,b,501);

% establish the values of n to test
n_vector = [5,15,50,200];

%error vectors for each n
rel1_equi = zeros(1,4);
rel2_equi = zeros(1,4);
rel1_cheb = zeros(1,4);
rel2_cheb = zeros(1,4);

%establish function
f = @(x) abs(x);

%loop for each value of n in the n_vector
for i = 1:length(n_vector)

figure;
title('Lagrange Barycentric Interpolation: f(x) = abs(x)')
xlabel('x');
ylabel('f(x)');

%establish both equispace and chebyshev points
x_equi = linspace(a, b, n_vector(i)+1);
x_cheb = (a+b)/2 + (b-a)/2 * cos( (2*(0:n_vector(i)) + 1)*pi ./ (2*n_vector(i)+2) );

%matlab for equi
 plot(t,interp_lagrange_poly(x_equi, f(x_equi), t));
hold on;

%bary for equi
 plot(t,interp_lagrange(x_equi, f(x_equi), t));


 %matlab for cheb
 plot(t,interp_lagrange_poly(x_cheb, f(x_cheb), t));

 %bary for cheb
 plot(t,interp_lagrange(x_cheb, f(x_cheb), t));


```

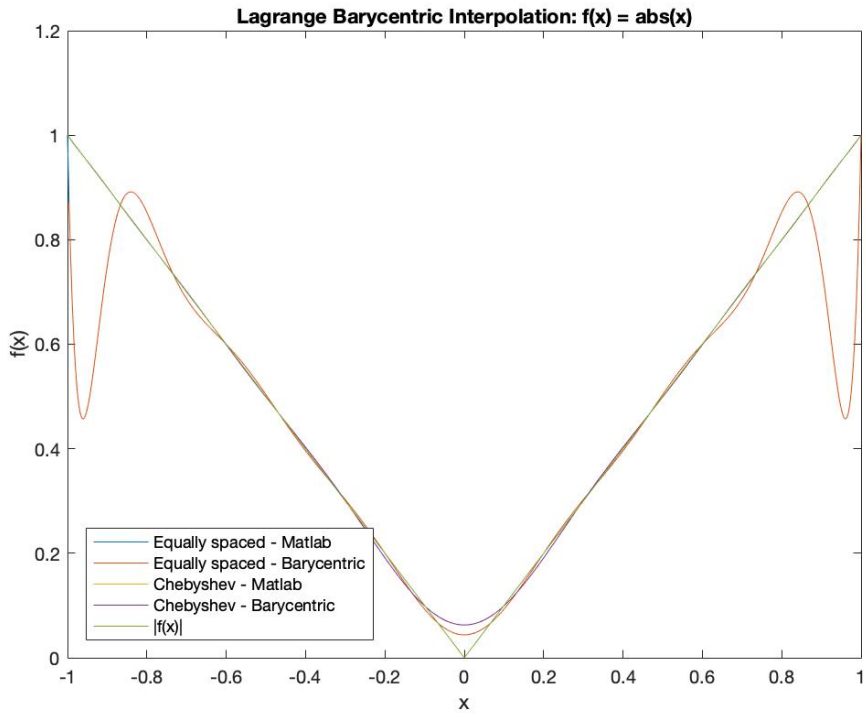
```
45  %theoretical
46  plot(t,f(t));
47
48  legend({'Equally spaced - Matlab','Equally spaced - Barycentric','Chebyshev - Matlab'
        ,'Chebyshev - Barycentric','|f(x)|'})
49  hold off;
50
51  %relative error values
52  rel1_equi(i) = max(abs(((interp_lagrange_poly(x_equi, f(x_equi), t)) - f(t))))/max(
        abs(f(t)));
53  rel2_equi(i) = max(abs(((interp_lagrange(x_equi, f(x_equi), t)) - f(t))))/max(abs(f(t
        )));
54
55  rel1_cheb(i) = max(abs(((interp_lagrange_poly(x_cheb, f(x_cheb), t)) - f(t))))/max(
        abs(f(t)));
56  rel2_cheb(i) = max(abs(((interp_lagrange(x_cheb, f(x_cheb), t)) - f(t))))/max(abs(f(t
        )));
57
58 end
59
60  %plot relative error
61 figure
62 semilogy(n_vector,rel1_equi);
63 hold on;
64 plot(n_vector,rel2_equi);
65 plot(n_vector,rel1_cheb);
66 plot(n_vector,rel2_cheb);
67 title('Lagrange Barycentric Interpolation: f(x) = abs(x) - Relative Error')
68 xlabel('n');
69 ylabel('error');
70 legend({'Equally spaced - Matlab','Equally spaced - Barycentric','Chebyshev - Matlab',
        'Chebyshev - Barycentric'});
71 hold off;
72
73
74 %plot norm as function of n
75 func1 = zeros(1,4);
76 func2 = zeros(1,4);
77
78 for i = 1:length(n_vector)
79     p1 = polyfit(x_equi, f(x_equi), n_vector(i));
80     func1(i) = norm(p1,inf);
81
82     p2 = polyfit(x_cheb, f(x_cheb), n_vector(i));
83     func2(i) = norm(p2,inf);
84 end
85 figure
86 semilogy(n_vector,func1);
87 hold on;
88 semilogy(n_vector,func2);
89
90 title('Polyfit norm')
91 xlabel('x');
92 ylabel('f(x)');
93 legend({'Equally spaced','Chebyshev'});
```

```
94
95 hold off;
```

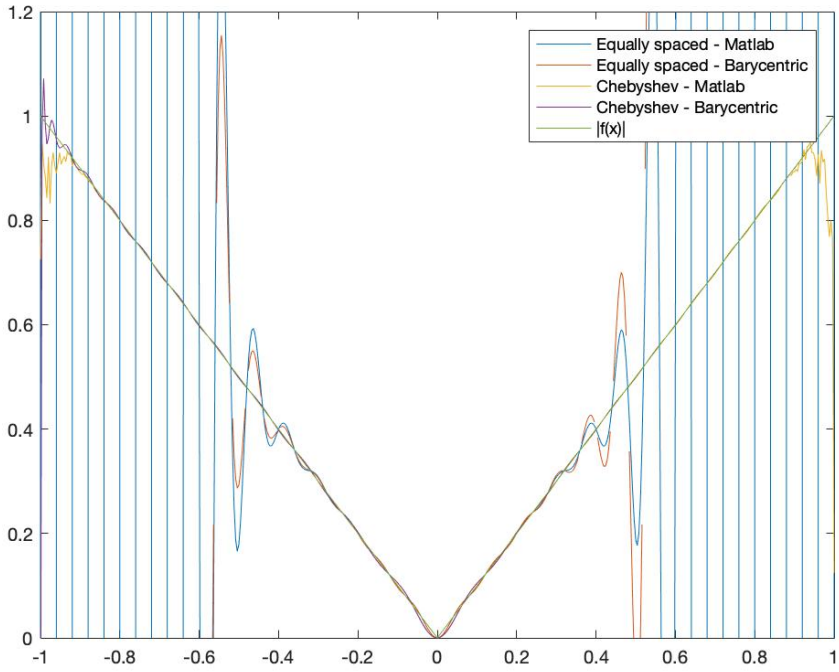
We obtain the following graphs: 1. n = 5:



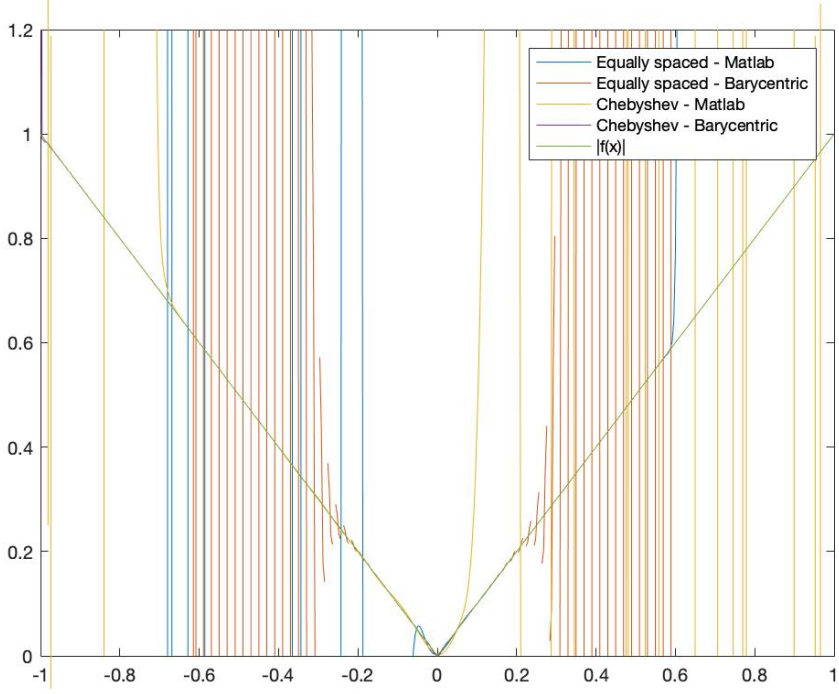
2. n = 15:



3. n = 50:



4. n = 200:

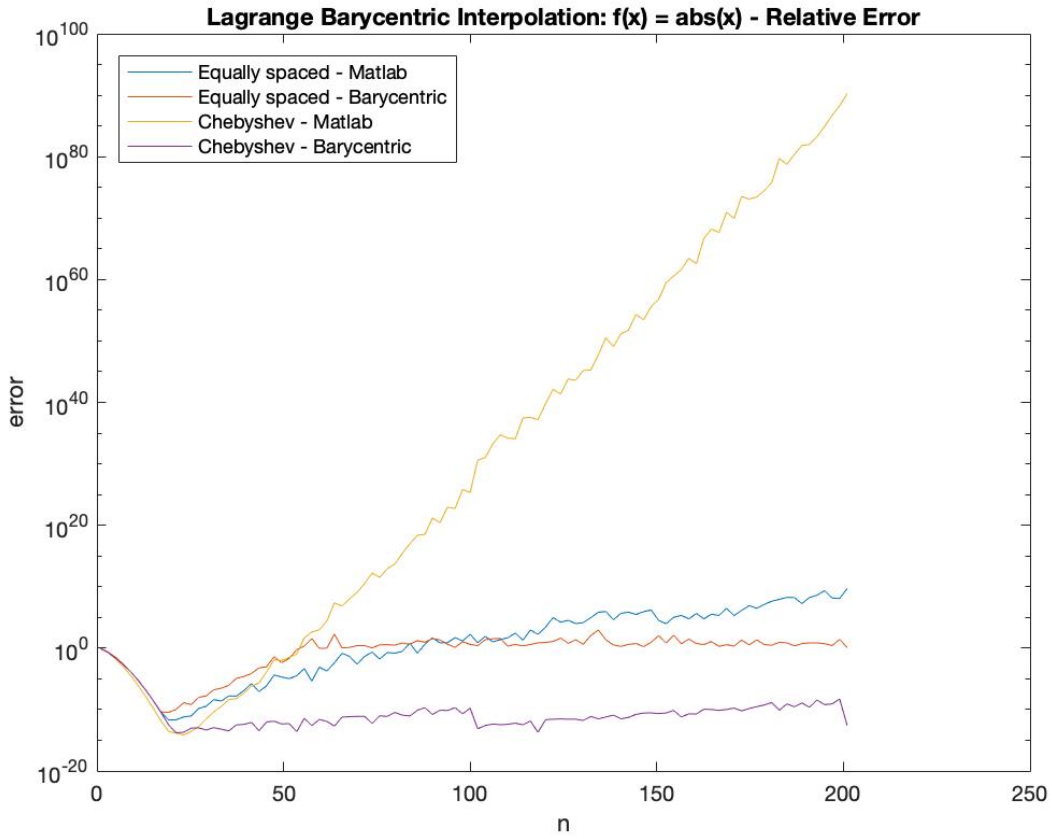


We obtain the relative errors from sampling a wide range of values for n, shown below:

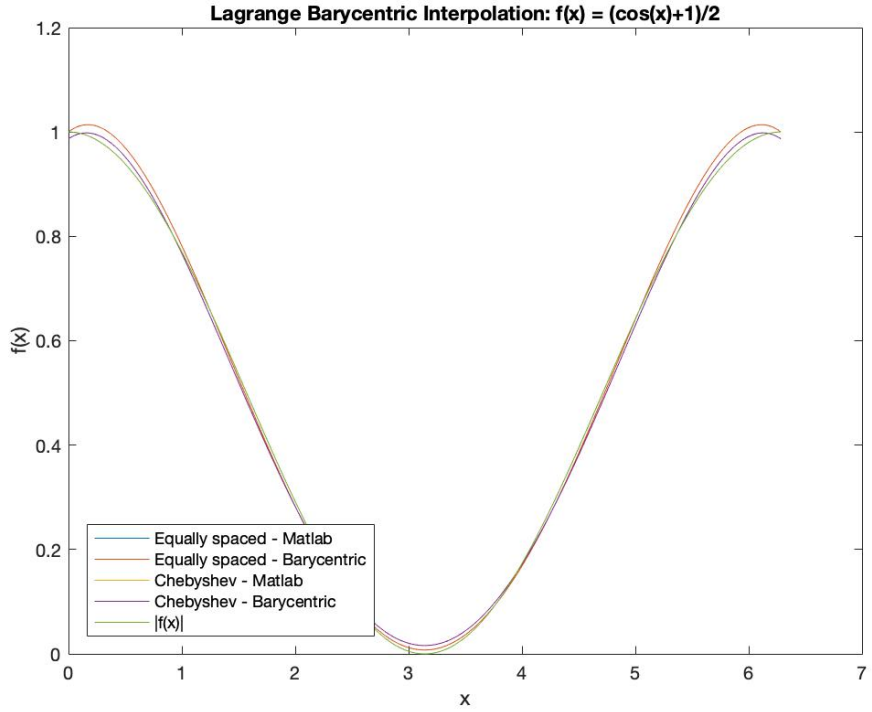
```
%set initial conditions for a, b, and t
a = -1;
b = 1;
t = linspace(a,b,501);
```

```
5
6  % establish the values of n to test
7  n_vector = linspace(1,201,100);
8
9  %error vectors for each n
10 rel1_equi = zeros(1,length(n_vector));
11 rel2_equi = zeros(1,length(n_vector));
12 rel1_cheb = zeros(1,length(n_vector));
13 rel2_cheb = zeros(1,length(n_vector));
14
15 %establish function
16 f = @(x) abs(x);
17
18 %loop for each value of n in the n_vector
19 for i = 1:length(n_vector)
20
21 %establish both equispace and chebyshev points
22 x_equi = linspace(a, b, n_vector(i)+1);
23 x_cheb = (a+b)/2 + (b-a)/2 * cos( (2*(0:n_vector(i)) + 1)*pi ./ (2*n_vector(i)+2) );
24
25  %relative error values
26  rel1_equi(i) = max(abs(((interp_lagrange_poly(x_equi, f(x_equi), t)) - f(t))))/max(
        abs(f(t)));
27  rel2_equi(i) = max(abs(((interp_lagrange(x_equi, f(x_equi), t)) - f(t))))/max(abs(f(t
        )));
28
29  rel1_cheb(i) = max(abs(((interp_lagrange_poly(x_cheb, f(x_cheb), t)) - f(t))))/max(
        abs(f(t)));
30  rel2_cheb(i) = max(abs(((interp_lagrange(x_cheb, f(x_cheb), t)) - f(t))))/max(abs(f(t
        )));
31
32 end
33
34  %plot relative error
35 figure
36 semilogy(n_vector,rel1_equi);
37 hold on;
38 plot(n_vector,rel2_equi);
39 plot(n_vector,rel1_cheb);
40 plot(n_vector,rel2_cheb);
41 title('Lagrange Barycentric Interpolation: f(x) = abs(x) - Relative Error')
42 xlabel('n');
43 ylabel('error');
44 legend({'Equally spaced - Matlab','Equally spaced - Barycentric','Chebyshev - Matlab',
       'Chebyshev - Barycentric'},'Location','northwest');
45 hold off;
```

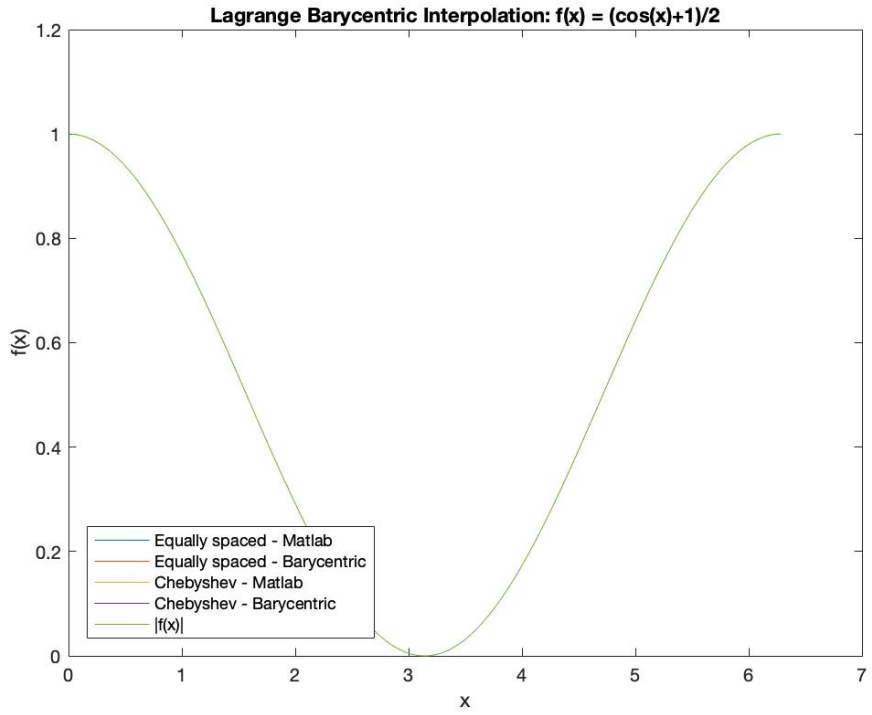
The relative errors are graphed below and we observe an increase in error as n becomes large, but the error is more constant and even decreases for Chebyshev Barycentric:



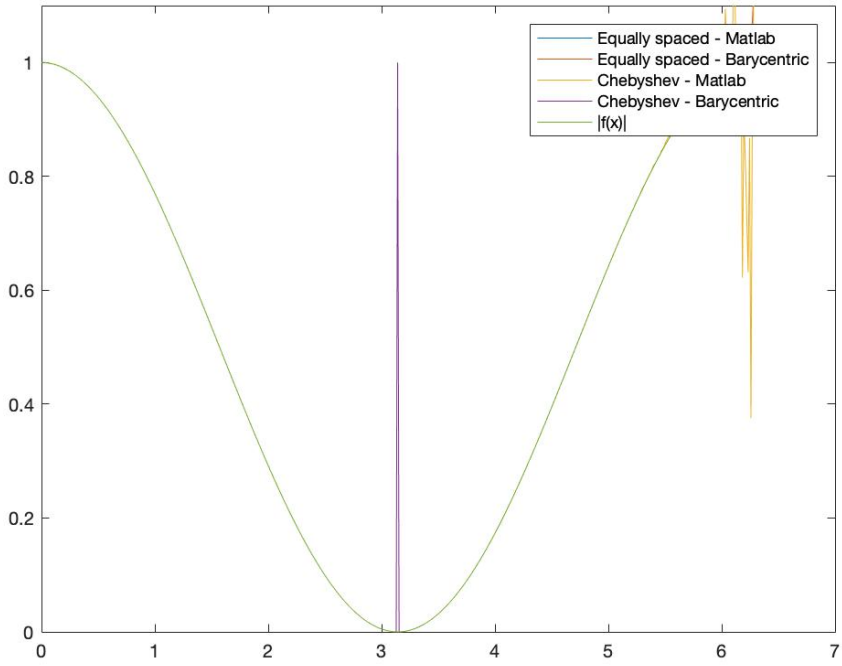
(d) We do the same for the function $f(x) = \frac{cos(x)+1}{2}$. The code is exactly identical, with only a change in the definition of the function. We obtain the graphs: 1. n = 5:


Lagrange Barycentric Interpolation: f(x) = (cos(x)+1)/2
f(x)
x
Equally spaced - Matlab
Equally spaced - Barycentric
Chebyshev - Matlab
Chebyshev - Barycentric
|f(x)|

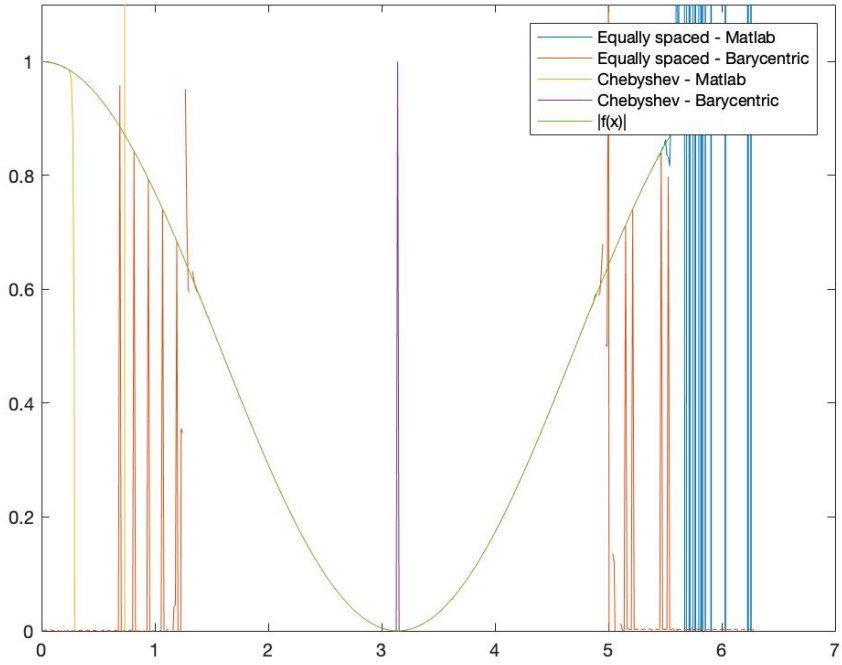
2. n = 15:


Lagrange Barycentric Interpolation: f(x) = (cos(x)+1)/2
f(x)
x
Equally spaced - Matlab
Equally spaced - Barycentric
Chebyshev - Matlab
Chebyshev - Barycentric
|f(x)|

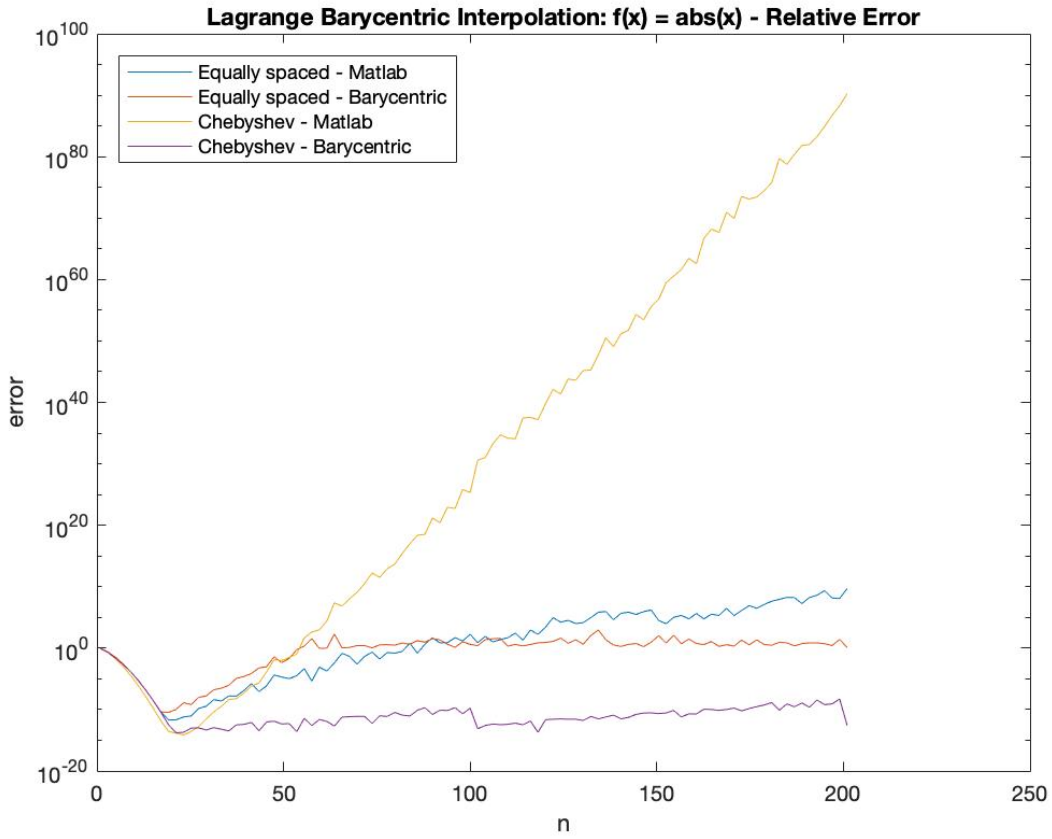
3. n = 50:



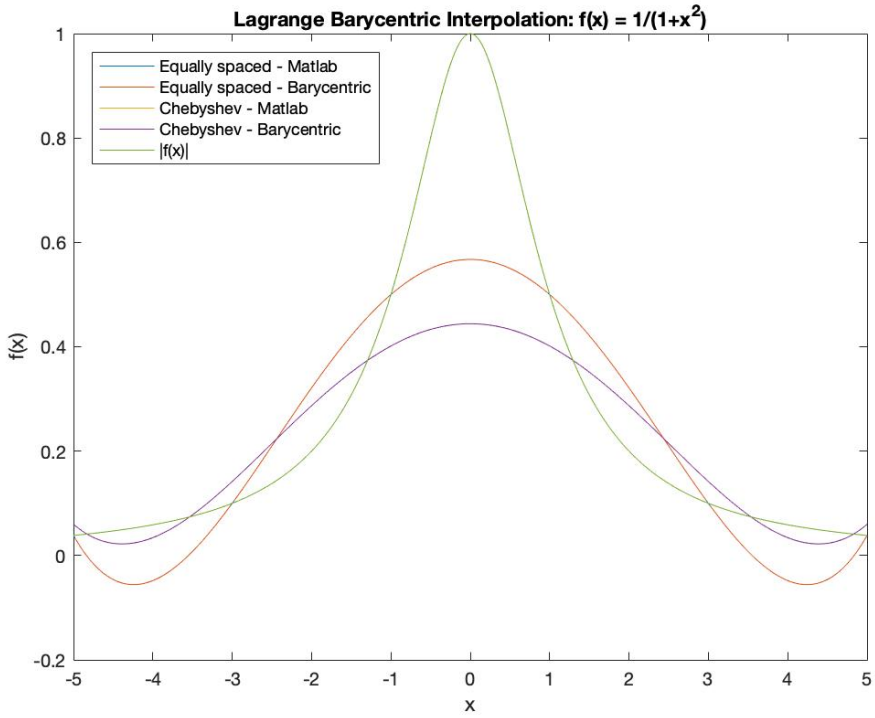
4. n = 200:



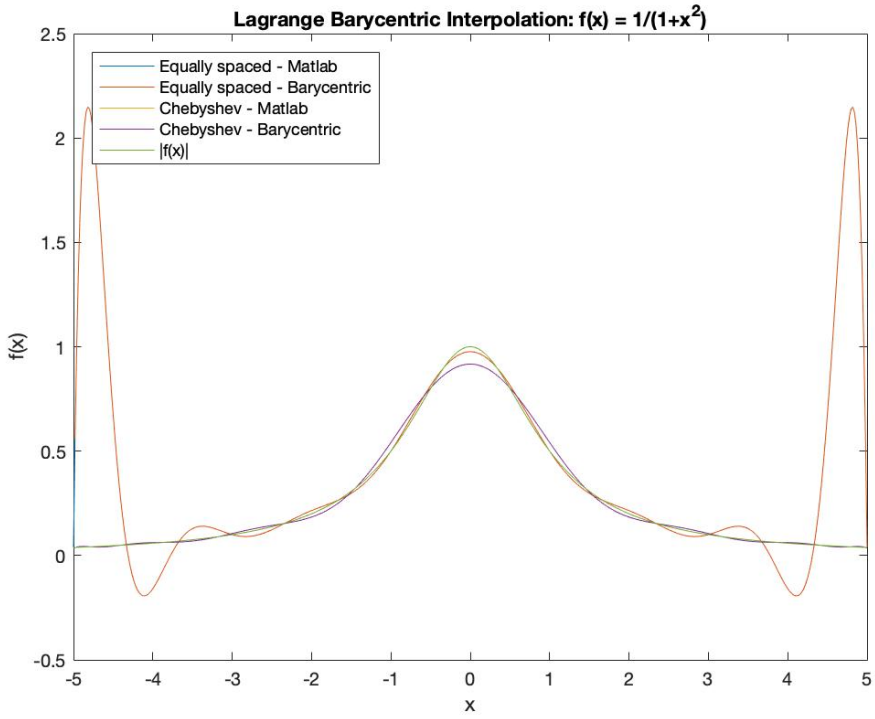
We obtain the relative errors from sampling a wide range of values for n, shown below:



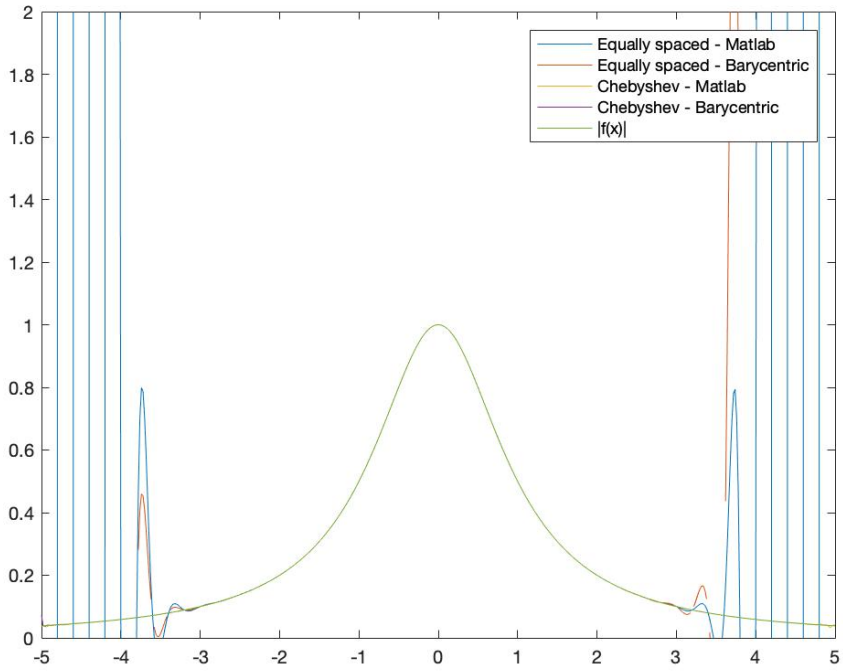
For the function $f(x) = \frac{1}{1+x^2}$, we obtain the graphs: 1. n = 5:


Lagrange Barycentric Interpolation: f(x) = 1/(1+x^2)
Equally spaced - Matlab
Equally spaced - Barycentric
Chebyshev - Matlab
Chebyshev - Barycentric
|f(x)|
f(x)
x

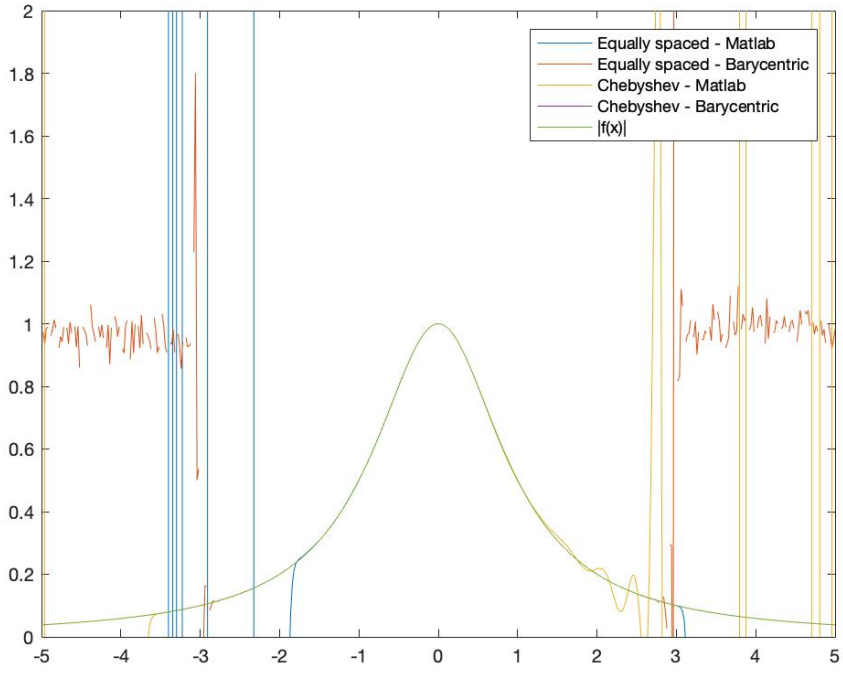
2. n = 15:


Lagrange Barycentric Interpolation: f(x) = 1/(1+x^2)
Equally spaced - Matlab
Equally spaced - Barycentric
Chebyshev - Matlab
Chebyshev - Barycentric
|f(x)|
f(x)
x

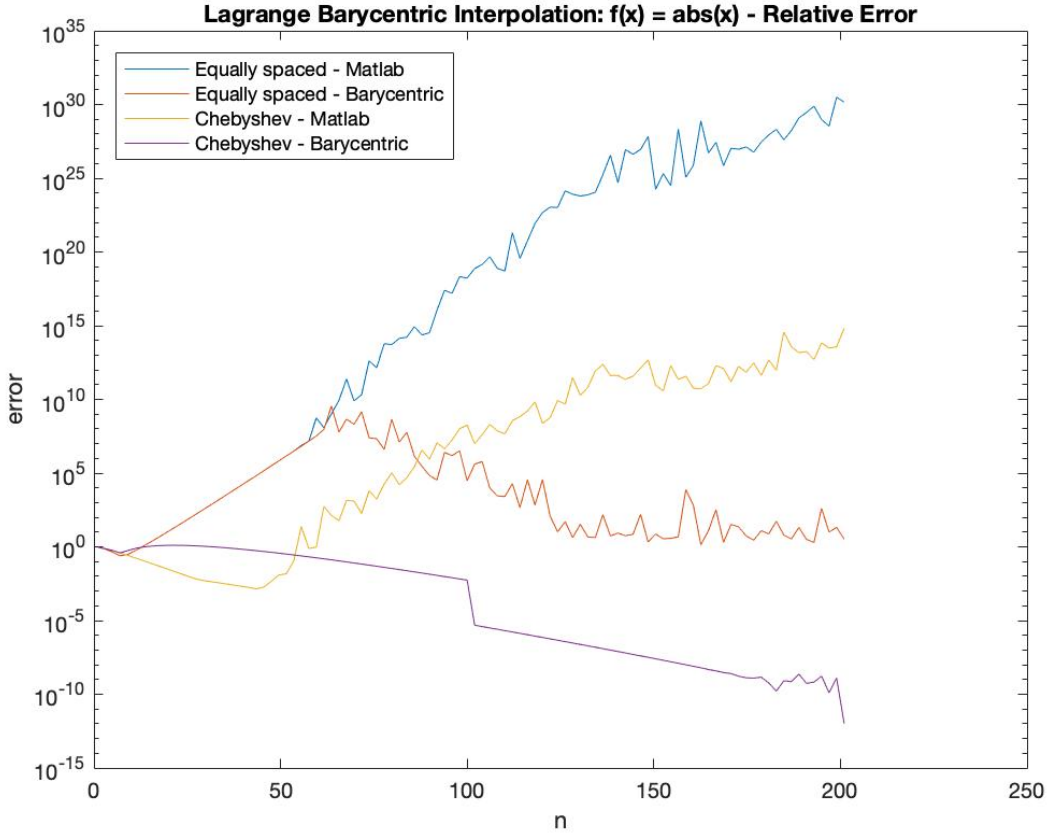
3. n = 50:



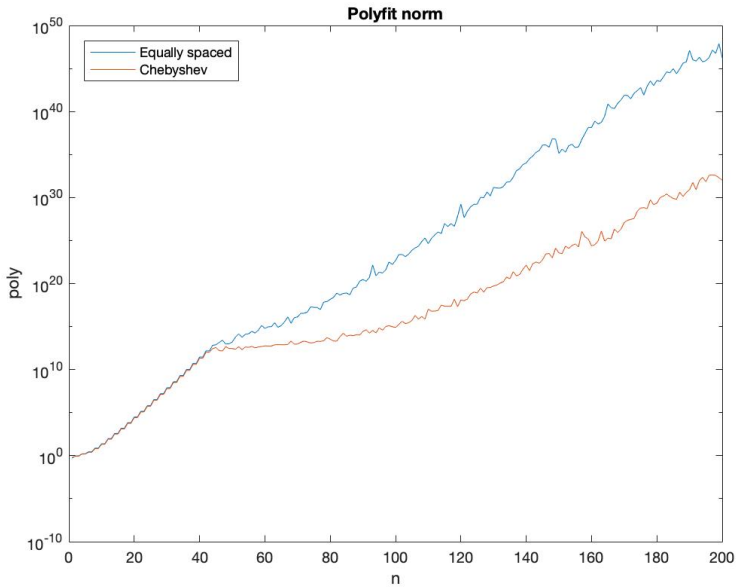
4. n = 200:



The relative errors are graphed below:



(e) We choose to plot the polyfit for the function $f(x) = |x|$; the graph of the norm of the polyfit as a function of n is shown below, seeing that the Chebyshev distribution of points and the equally spaced distribution diverge in the polynomial fitting norm with larger values of n:



Our code that was used to implement the graph is shown below:

```
%set initial conditions for a, b, and t
a = -1;
b = 1;
t = linspace(a,b,501);

% establish the values of n to test
n_vector = linspace(1,201,100);

%error vectors for each n
rel1_equi = zeros(1,length(n_vector));
rel2_equi = zeros(1,length(n_vector));
rel1_cheb = zeros(1,length(n_vector));
rel2_cheb = zeros(1,length(n_vector));

%establish function
f = @(x) abs(x);

%loop for each value of n in the n_vector
for i = 1:length(n_vector)

%establish both equispace and chebyshev points
x_equi = linspace(a, b, n_vector(i)+1);
x_cheb = (a+b)/2 + (b-a)/2 * cos( (2*(0:n_vector(i)) + 1)*pi ./ (2*n_vector(i)+2) );

 %relative error values
 rel1_equi(i) = max(abs(((interp_lagrange_poly(x_equi, f(x_equi), t)) - f(t))))/max(
     abs(f(t)));
 rel2_equi(i) = max(abs(((interp_lagrange(x_equi, f(x_equi), t)) - f(t))))/max(abs(f(t
     )));

 rel1_cheb(i) = max(abs(((interp_lagrange_poly(x_cheb, f(x_cheb), t)) - f(t))))/max(
     abs(f(t)));
 rel2_cheb(i) = max(abs(((interp_lagrange(x_cheb, f(x_cheb), t)) - f(t))))/max(abs(f(t
     )));

end

%plot norm as function of n
func1 = zeros(1,length(n_vector));
func2 = zeros(1,length(n_vector));

for i = 1:length(n_vector)
    p1 = polyfit(x_equi, f(x_equi), n_vector(i));
    func1(i) = norm(p1,inf);

    p2 = polyfit(x_cheb, f(x_cheb), n_vector(i));
    func2(i) = norm(p2,inf);
end
figure
semilogy(n_vector,func1);
hold on;
semilogy(n_vector,func2);

title('Polyfit norm')
```

```
xlabel('x');
ylabel('f(x)');
legend({'Equally spaced','Chebyshev'});

hold off;
```

3 Minimax Approximation

3.1 Suli and Mayers exercise 8.2

We let $p_n \in P_n$ be a minimax polynomial for $f(x)$ on $[-a,a]$. So, by the Oscillation Theorem, we know that there exists $n+2$ critical points $x_0, x_1, x_2, ..., x_{n+1}$ in the interval $[-a,a]$ such that: 1. $|f(x_i) - p_n(x_i)| = \|f - p_n\|_\infty$ for $i = 0, 1, 2, ..., n+1$ and 2. $f(x_i) - p_n(x_i) = (-1)(f(x_{i+1}) - p_n(x_{i+1}))$.
Thus, let $f(x_0) - p_n(x_0) = E$. Thus, we see that $|E| = \|f - p_n\|_\infty$. Hence, by the second property of the Oscillation Theorem, we see that we have:

$$(f(x_i) - p_n(x_i)) = (-1)f(x_{i-1}) - p_n(x_{i-1}) = (-1)^2(f(x_{i-2}) - p_n(x_{i-2})) = ... \tag{2}$$
$$= (-1)^i(f(x_0) - p_n(x_0) = (-1)^i E \tag{3}$$

We define another polynomial of degree n: $q_n(x) \in P_n$ such that $q_n(x) = p_n(-x) \rightarrow q_n(-x) = p_n(x)$. Thus, we can rewrite the equation (2) and (3) above as (for $i = 0, 1, 2, ..., n+1$):

$$(f(x_i) - p_n(x_i)) = (-1)^i E \rightarrow \tag{4}$$
$$(f(x_i) - q_n(-x_i)) = (-1)^i E \tag{5}$$

Because the function $f(x)$ is an even function, we know that $f(x) = f(-x)$. Thus, we have:

$$(f(x_i) - q_n(-x_i)) = (f(-x_i) - q_n(-x_i)) = (-1)^i E \tag{6}$$

where x_i are the critical points of p_n. Since x_i is in the interval $[-a,a]$, $-x_i$ must also be in the interval $[-a,a]$. We also observe that $\|f - q_n\|_\infty = |f(-x_0) - q_n(-x_0)| = |f(x_0) - p_n(x_0)| = |E|$. Thus, we see that $q_n(x)$ must also a minimax polynomial of $f(x)$ with the $n+2$ critical points $-x_i$. Thus, by the Uniqueness Theorem, we know that each function $f(x)$ can only admit a unique minimax polynomial. Thus, we conclude that $q_n(x) = p_n(-x) = p_n(x)$, and this minimax polynomial has critical values of both x_k and $-x_k$ in the interval for all k, with $n+2$ critical points total. Therefore, because $p_n(x) = p_n(-x)$, the polynomial $p_n(x)$, the minimax approximation polynomial of degree n, must be an even polynomial, as desired.

Thus, because $p_n(x)$ is an even polynomial, the minimax approximation of degree $2n+1$ must also satisfy $p_{2n+1}(x) = p_{2n+1}(-x)$. Thus, we conclude that in the polynomial $p_{2n+1}(x)$ must not have a x^{2n+1} term because we know that $p_{2n}(x) = p_{2n}(-x)$. Therefore, because $p_{2n+1}(x)$ does not have a x^{2n+1} term, its maximum order is $2n$. Thus, the minimax polynomial approximation of degree $2n+1$ is also the minimax polynomial approximation of degree $2n$. Thus, because for the minimax polynomial approximation $p_{2n+1}(x)$ $(= p_{2n}(x))$ has $2n+1+2 = 2n+3$ critical points in the interval $[-a, a]$ that we have shown earlier to have the property of having both x_k and $-x_k$ as critical values, we conclude that $p_2n(x)$ has $2n+3$ critical points with this property and one of the critical points is 0.

3.2 Suli and Mayers exercise 8.8

We wish to find the minimax polynomial $p_n \in P_n$ on [-1,1] for the function:

$$f(x) = \sum_{k=0}^{n+1} a_k x^k = \sum_{k=0}^{n} a_k x^k + a_{n+1}x^{n+1} \tag{7}$$

Thus, we see that:

$$f(x) - p_n(x) = \sum_{k=0}^{n} a_k x^k + a_{n+1}x^{n+1} - p_n(x) = (a_{n+1})(x^{n+1} - \sum_{k=0}^{n}(-a_k)x^k - \frac{p_n(x)}{a_{n+1}}) \rightarrow \tag{8}$$

$$\frac{f(x) - p_n(x)}{a_{n+1}} = x^{n+1} - (\sum_{k=0}^{n}(-a_k)x^k + \frac{p_n(x)}{a_{n+1}}) \tag{9}$$

Thus, we see that $\frac{f(x)-p_n(x)}{a_{n+1}}$ is a monic polynomial of degree $n+1$. We see then that by Corollary 8.1, the polynomial $2^{-n}T_{n+1}$ has the smallest ∞-norm on the interval $[-1, 1]$. Thus, because we want to minimize $\|f(x)-p_n(x)\|_\infty$, we would like $x^{n+1} - (\sum_{k=0}^{n}(-a_k)x^k + \frac{p_n(x)}{a_{n+1}}) = 2^{-n}T_{n+1}(x)$. Thus, we see that $f(x) - p_n(x) = a_{n+1}2^{-n}T_{n+1}$, so we have $p_n(x) = \boxed{f(x) - a_{n+1}2^{-n}T_{n+1}(x)}$.

We know that $\|T_{n+1}/a_{n+1}\|_\infty = 1/|a_{n+1}|$. Thus,$\|f - p_n\|_\infty$ over the interval $[-1, 1] = \|a_{n+1}2^{-n}T_{n+1}\|_\infty = \|a_{n+1}^2 2^{-n}(T_{n+1}/a_{n+1})\|_\infty = a_{n+1}^2 2^{-n}\|(T_{n+1}/a_{n+1})\|_\infty = a_{n+1}^2 2^{-n}(1/|a_{n+1}|)$ $= \boxed{|a_{n+1}|2^{-n}}$

4 A peek at approximation for periodic functions

1. We are given the matrix A defined as (with n given as odd):

$$A = \begin{bmatrix} b_0(x_0) & b_1(x_0) & ... & b_{n-1}(x_0) \\ b_0(x_1) & b_1(x_1) & ... & b_{n-1}(x_1) \\ b_0(x_2) & b_1(x_2) & ... & b_{n-1}(x_2) \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ b_0(x_{n-1}) & b_1(x_{n-1}) & ... & b_{n-1}(x_{n-1}) \end{bmatrix} \tag{10}$$

with:

$$b_0(x) = 1 \tag{11}$$
$$b_1(x) = sin(x) \tag{12}$$
$$b_2(x) = cos(x) \tag{13}$$
$$b_3(x) = sin(2x) \tag{14}$$
$$b_4(x) = cos(2x) \tag{15}$$
$$b_5(x) = sin(3x) \tag{16}$$
$$b_6(x) = cos(3x) \tag{17}$$
$$. \tag{18}$$
$$. \tag{19}$$
$$. \tag{20}$$
$$\tag{21}$$

Thus, we see that A^T is:

$$A^T = \begin{bmatrix} b_0(x_0) & b_0(x_1) & ... & b_0(x_{n-1}) \\ b_1(x_0) & b_1(x_1) & ... & b_1(x_{n-1}) \\ b_2(x_0) & b_2(x_1) & ... & b_2(x_{n-1}) \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ b_{n-1}(x_0) & b_{n-1}(x_1) & ... & b_{n-1}(x_{n-1}) \end{bmatrix} \tag{22}$$

So, $A^T A$ becomes:

$$B = A^T A = \begin{bmatrix} b_0(x_0) & b_0(x_1) & ... & b_0(x_{n-1}) \\ b_1(x_0) & b_1(x_1) & ... & b_1(x_{n-1}) \\ b_2(x_0) & b_2(x_1) & ... & b_2(x_{n-1}) \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ b_{n-1}(x_0) & b_{n-1}(x_1) & ... & b_{n-1}(x_{n-1}) \end{bmatrix} \begin{bmatrix} b_0(x_0) & b_1(x_0) & ... & b_{n-1}(x_0) \\ b_0(x_1) & b_1(x_1) & ... & b_{n-1}(x_1) \\ b_0(x_2) & b_1(x_2) & ... & b_{n-1}(x_2) \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ b_0(x_{n-1}) & b_1(x_{n-1}) & ... & b_{n-1}(x_{n-1}) \end{bmatrix} \tag{23}$$

Thus, we observe that for indices i, j:

$$B_{ij} = B_{ji} = b_{j-1}(x_0)b_{i-1}(x_0) + b_{j-1}(x_1)b_{i-1}(x_1) + ... + b_{j-1}(x_{n-1})b_{i-1}(x_{n-1}) \tag{24}$$

We have the following cases:

Case 1: i is even, j is even:
We observe that for this case, $b_{j-1}(x) = sin((\frac{j}{2})x)$ and $b_{i-1}(x) = sin((\frac{i}{2})x)$. Because the x_k for $k = 0, 1, 2, ..., n-1$ is evenly distributed along the interval [0,2π], we see that we can represent

$x_k = k\frac{(2\pi)}{n}$. Thus, $b_{j-1}(x_k) = sin(\frac{jk\pi}{n})$ and $b_{i-1}(x) = sin(\frac{ik\pi}{n})$. We see that, thus:

$$B_{ij} = b_{j-1}(x_0)b_{i-1}(x_0) + b_{j-1}(x_1)b_{i-1}(x_1) + ... + b_{j-1}(x_{n-1})b_{i-1}(x_{n-1}) = \tag{25}$$

$$\sum_{k=0}^{n-1} sin(\frac{jk\pi}{n})sin(\frac{ik\pi}{n}) \rightarrow \tag{26}$$

$$2B_{ij} = \sum_{k=0}^{n-1} sin(\frac{jk\pi}{n})sin(\frac{ik\pi}{n}) + \sum_{k=0}^{n-1} sin(\frac{j(n-k)\pi}{n})sin(\frac{i(n-k)\pi}{n}) \tag{27}$$

We see that:

$$sin(\frac{j(n-k)\pi}{n}) = -sin(\frac{j(n-k)\pi}{n} - \pi) = -sin(\pi(j-1) - \frac{jk\pi}{n}) \tag{28}$$

Since j is even, $j-1$ is odd, so $sin(\frac{j(n-k)\pi}{n}) = -sin(\pi(j-1) - \frac{jk\pi}{n}) = -cos(\frac{jk\pi}{n})$. Thus, we have:

$$2B_{ij} = \sum_{k=0}^{n-1} sin(\frac{jk\pi}{n})sin(\frac{ik\pi}{n}) + \sum_{k=0}^{n-1} cos(\frac{jk\pi}{n})cos(\frac{ik\pi}{n}) = \tag{29}$$

$$\sum_{k=0}^{n-1} cos(\frac{jk\pi}{n} - \frac{ik\pi}{n}) = \sum_{k=0}^{n-1} cos(\frac{k\pi}{n}(j-i)) = \sum_{k=0}^{n-1} cos(\frac{2k\pi}{n}\frac{(j-i)}{2}) \tag{30}$$

Thus, because j and i are both within the interval $[0, n-1]$ and are both even, so $j-i$ is also even, we see that $\frac{(j-i)}{2}$ will always be less than n, so $2B_{ij}$ can only, by the identities provided in the homework question, have values of either 0 when $i \neq j$ and either n when $i = j$. Thus, we have $B_{ij} = B_{ji} = 0$ when $i \neq j$ and $n/2$ when $i = j$ when both i and j are even indices.

Case 2: i is even, j is odd: (Because B is symmetric, this case can be extended to when j is even, i is odd by symmetry.
We observe that for this case, $b_{j-1}(x) = cos((\frac{j-1}{2})x)$ and $b_{i-1}(x) = sin((\frac{i}{2})x)$. Similarly to the first case, we see that we can represent:

$$2B_{ij} = \sum_{k=0}^{n-1} cos(\frac{(j-1)k\pi}{n})sin(\frac{ik\pi}{n}) + \sum_{k=0}^{n-1} cos(\frac{(j-1)(n-k)\pi}{n})sin(\frac{i(n-k)\pi}{n}) \tag{31}$$

Because i is even and j is odd, we can conclude that: $cos(\frac{(j-1)(n-k)\pi}{n}) = cos(\frac{(j-1)k\pi}{n})$ and $sin(\frac{i(n-k)\pi}{n}) = -sin(\frac{ik\pi}{n})$. Thus, we have:

$$2B_{ij} = \sum_{k=0}^{n-1} cos(\frac{(j-1)k\pi}{n})(sin(\frac{ik\pi}{n}) - sin(\frac{ik\pi}{n})) = 0 \tag{32}$$

Thus, in this case, $B_{ij} = 0$ for all i and j when they diffeent parity, which garuantees that they are distinct and $i \neq j$.

Case 3: Both i and j are odd. Thus, we have $b_{j-1}(x) = cos((\frac{j-1}{2})x)$ and $b_{i-1}(x) = cos((\frac{i-1}{2})x)$. Thus, we observe that:

$$2B_{ij} = \sum_{k=0}^{n-1} 2cos(\frac{(j-1)k\pi}{n})cos(\frac{(i-1)k\pi}{n}) = \sum_{k=0}^{n-1} cos(\frac{2k\pi}{n}(i+j-2)/2) + \sum_{k=0}^{n-1} cos(\frac{2k\pi}{n}(i-j)/2) \tag{33}$$

Thus, we observe that $2B_{ij} = 2n$ when $i = j = 1$ because we must have $i + j - 2$ be equivalent to 0 (mod n) and we see that it cannot be n; $2B_{ij} = n$ when $i = j \neq 1$, and $B_{ij} = 0$ when $i \neq j$.

Thus, we see that from our three cases, the matrix $B = A^T A$ is symmetric and has the properties: $B_{ij} = B_{ji} = 0$ when $i \neq j$, $B_{ij} = n/2$ when $i = j \neq 1$, and $B_{ij} = n$ when $i = j = 1$. Therefore, we see that B must be a diagonal matrix. Thus, $D = A^T A$, where we have:

$$D = \begin{bmatrix} n & 0 & ... & 0 \\ 0 & n/2 & ... & 0 \\ 0 & 0 & ... & 0 \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ 0 & 0 & ... & n/2 \end{bmatrix} \tag{34}$$

2. We know that the singular values of a matrix A are the square roots of the eigenvalues of $A^T A$. Because $D = A^T A$ is a diagonal matrix, its eigenvalues are just the elements in the diagonal which are n and $n/2$. Therefore, the singular values of A are just $\boxed{\sqrt{n}}$ and $\boxed{\sqrt{n/2}}$.

3. We know that the condition number $\kappa(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)}$. We know that $\sigma_{max}(A) = \sqrt{n}$ and $\sigma_{min}(A) = \sqrt{n/2}$; therefore, the condition number is $\kappa(A) = \frac{\sqrt{n}}{\sqrt{n/2}} = \boxed{\sqrt{2}}$, as desired.

4. We observe that A is nonsingular due to the basis of the periodic polynomial basis, so A must be invertible. We know that $D = A^T A$; since D is a diagonal matrix, it must be invertible, so we have:

$$D^{-1} = (A^T A)^{-1} = A^{-1}(A^T)^{-1} \rightarrow \tag{35}$$

$$A^{-1} = A^{-1}(A^T)^{-1}(A^T) = \boxed{D^{-1}A^T} \tag{36}$$

We want $D^{-1}D = I$, so we have the following expression for the inverse of the diagonal matrix:

$$D^{-1} = \begin{bmatrix} 1/n & 0 & ... & 0 \\ 0 & 2/n & ... & 0 \\ 0 & 0 & ... & 0 \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ 0 & 0 & ... & 2/n \end{bmatrix} \tag{37}$$

5. We wish to solve for the vector c given the y vector to be the values of f at the interpolation points. So, we have the linear system $Ac = y$. Thus, we have:

$$Ac = y \rightarrow A^{-1}Ac = c = A^{-1}y = D^{-1}A^T y \tag{38}$$

So, we have the expression for the conditions for interpolation:

$$c = D^{-1}A^T y \tag{39}$$

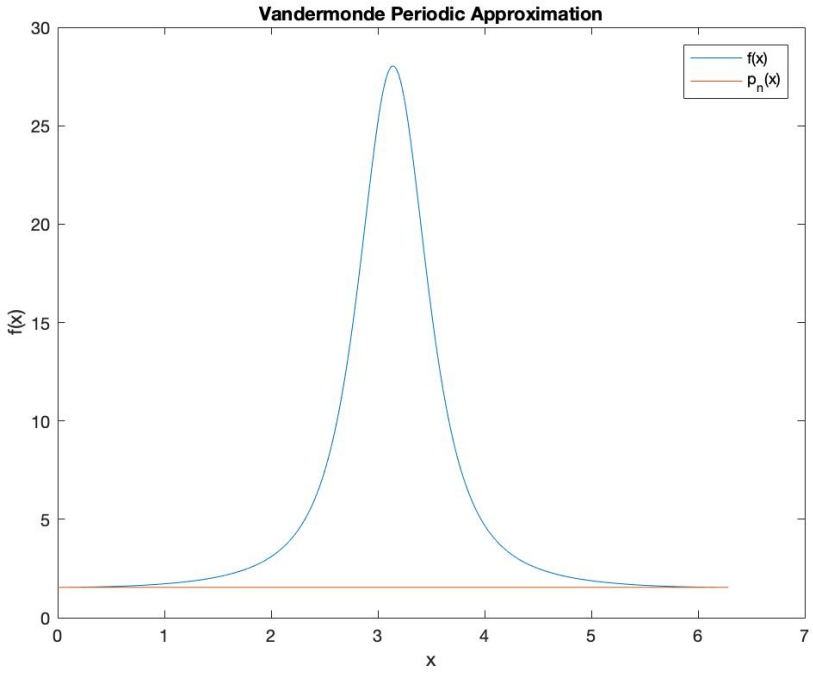
We see that the matrix multiplication of $D^{-1}A^T$ requires only n^2 flops, for there are only operations along the diagonals of A^T, as $A_{ii} \leftarrow A_{ii}/D_{ii}$. Furthermore, because y is a $n \times 1$ vector, we observe that the matrix-vector multiplication $D^{-1}A^T y$ requires n^2 flops. Thus, solving the Vandermonde system only requires a total of $O(n) = 2n^2 = O(n^2)$ flops.

6. We implement the algorithm shown below for the function $f(x)$ that was given:

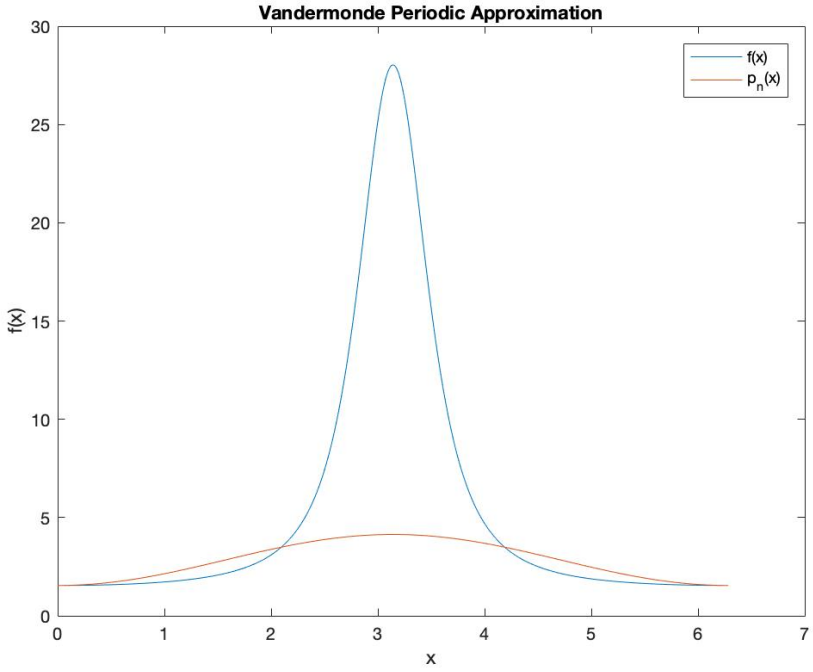
```
%Implement Vandermonde Algorithm
function[] = vandermonde(n)

%establish x points
x = linspace(0, 2*pi, n+1);
x = x(1:end-1);
x = x';

%create function f
f = @(x) exp(1./(1.3 + cos(x));

%y vector
y = f(x);

%create vandermonde matrix
A = zeros(n,n);

%loop to create A
for i = 0:n-1
    if (mod(i,2)==0)
        A(:,i+1) = cos((i/2)*x);
    else
        A(:,i+1) = sin(((i+1)/2)*x);
    end
end

%c = D^(-1)A^Ty
%compute D^(-1)A^T
A = A';
A(1,:) = A(1,:)/n;
for k = 2:n
        A(k,:) = A(k,:) * (2/n);
end

%compute c
c = A*y;

%fine grid to value function f and pn
n1 = 1001;
t = linspace(0, 2*pi, n1);
t = t';

%find approximation
p = zeros(n1,1);

%create polynomial
for j = 0:n-1
    if (mod(j,2)==0)
        p = p + c(j+1)*cos((j/2)*t);
```

```
    else
        p = p + c(j+1)*sin(((j+1)/2)*t);
    end
end

%plot
figure
plot(t,f(t));
hold on;
plot(t,p);

title('Vandermonde Periodic Approximation');
xlabel('x');
ylabel('f(x)');
legend('f(x)', 'p_n(x)');
hold off;

%abolute error
err = abs(f(t) - p);
fprintf('%.16e\n',max(err))
end
```

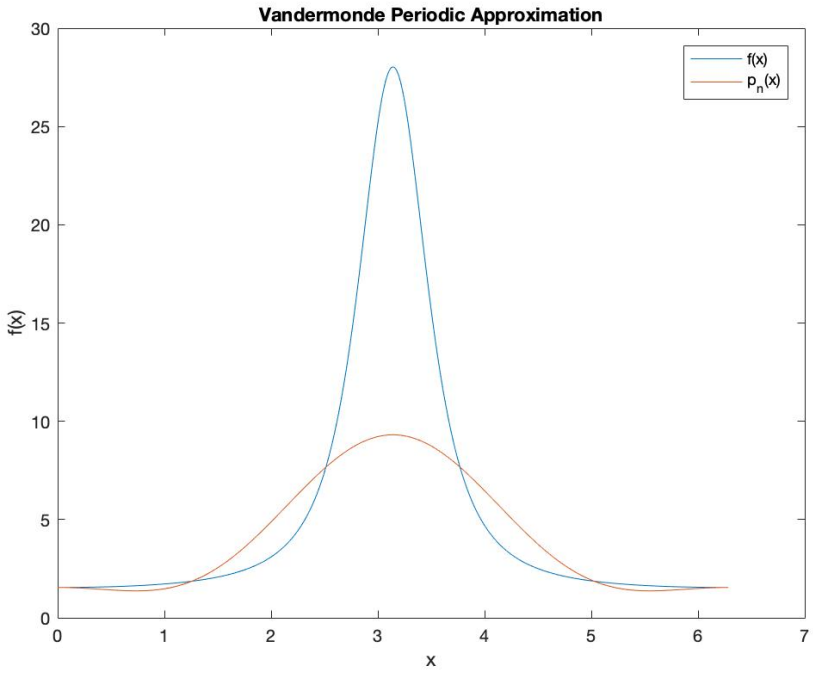
For values of n within values given, we obtain the following graphs:
1. n = 1:



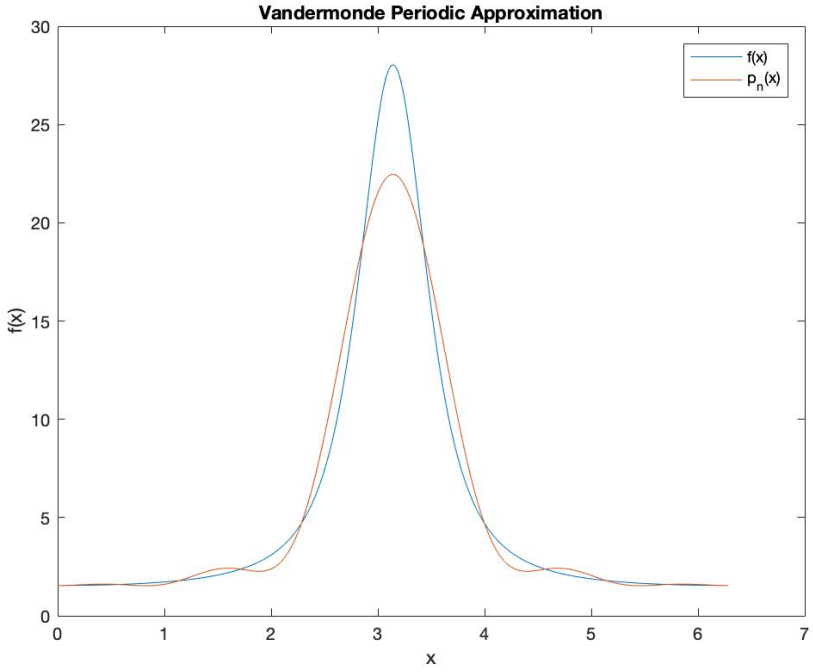
2. n = 3:


Vandermonde Periodic Approximation
f(x)
$p_n(x)$
f(x)
x
0
5
10
15
20
25
30
0
1
2
3
4
5
6
7

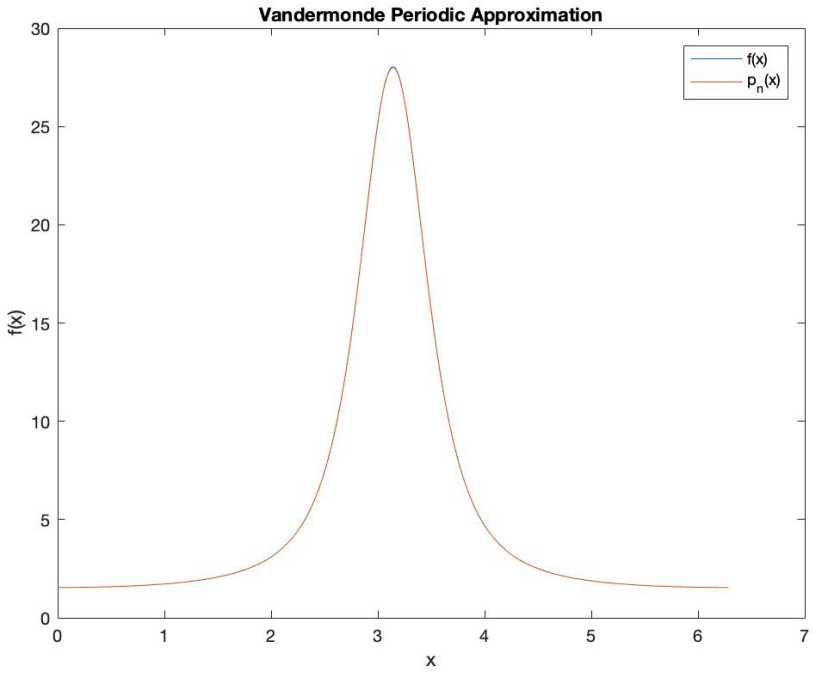
3. n = 5:


Vandermonde Periodic Approximation
f(x)
$p_n(x)$
f(x)
x
0
5
10
15
20
25
30
0
1
2
3
4
5
6
7

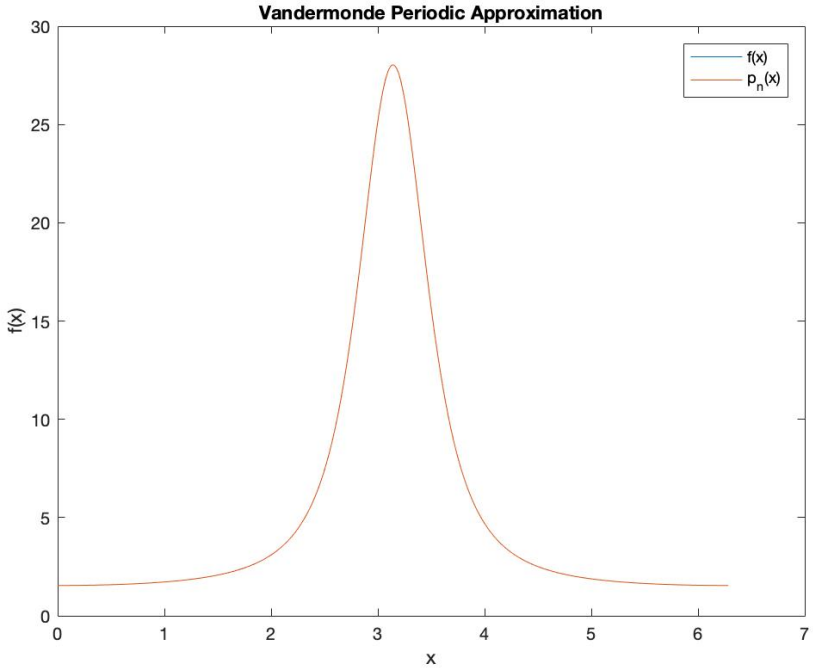
4. n = 11:


Vandermonde Periodic Approximation
f(x)
$p_n(x)$
f(x)
x
0
5
10
15
20
25
30
0
1
2
3
4
5
6
7

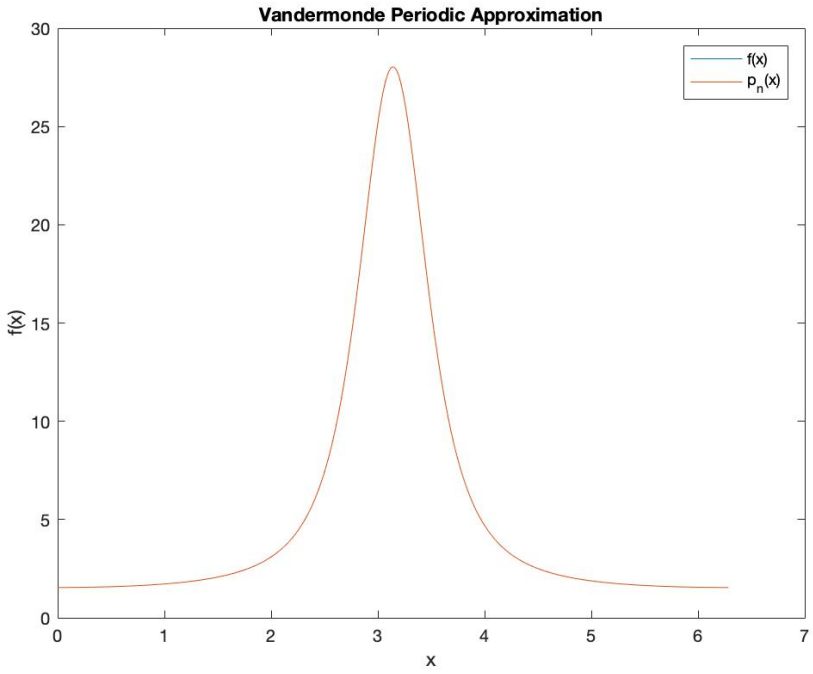
5. n = 31:


Vandermonde Periodic Approximation
f(x)
$p_n(x)$
f(x)
x
0
5
10
15
20
25
30
0
1
2
3
4
5
6
7

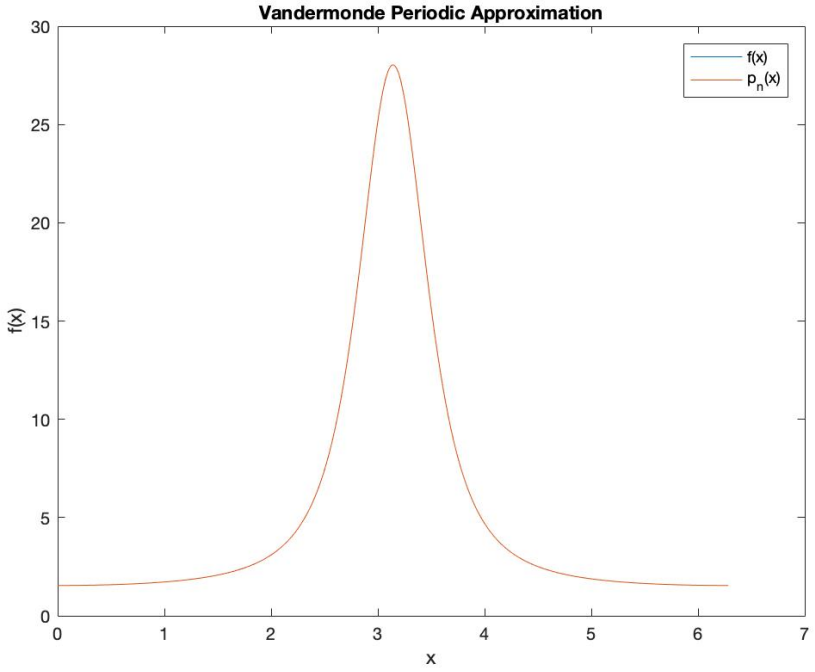
6. n = 51:


Vandermonde Periodic Approximation
f(x)
$p_n(x)$
f(x)
x
0
5
10
15
20
25
30
0
1
2
3
4
5
6
7

7. n = 101:


Vandermonde Periodic Approximation
f(x)
$p_n(x)$
f(x)
x
0
5
10
15
20
25
30
0
1
2
3
4
5
6
7

8. n = 201:



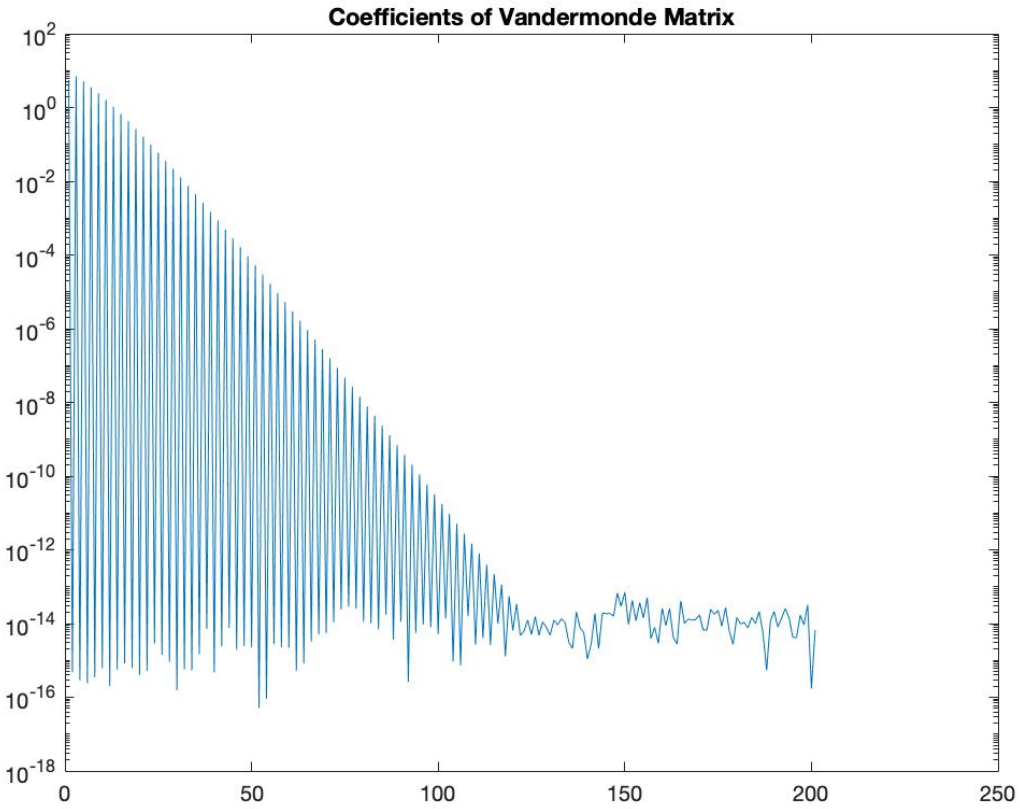
We test to see the absolute errors for each value of n as well:

```
%test vandermonde

n_sample = [1 3 5 11 31 51 101 201];

m = size(n_sample);

for i = 1:m(2)
    vandermonde(n_sample(i));
end
```

Our output for the maximum absolute error for each value of n tested is shown below:

```
n = 1: 2.6486997660605297e+01
n = 3: 2.3892710029217277e+01
n = 5: 1.8720743493066397e+01
n = 11: 5.5713490050254464e+00
n = 31:3.5866346287996009e-02
n = 51: 1.3502330439862931e-04
n = 101: 4.0596859207653324e-11
n = 201: 6.8567374000849668e-13
```

7. For $n = 201$, we plot the absolute values of the coefficients c on a log-scale.



We observe that the absolute value of the coefficients oscillate because the cosine function is better at approximating the function because they are both even functions and at larger indices have significantly smaller value. As n increases, the coefficients corresponding to b_n decreases to nearly 0.