FCMII - Programming Report 1

Note on Implementation

We implement our routines in septate MATLAB function.m files for easy organization. The test driver and also manual testing examples are done in live script file (.mlx). The vectorization provided by MATLAB is used as much as possible to avoid loops, since these operations are highly optimized in MATLAB. I used generative AI to streamline some plotting code and data visualization, specifically CHATGPT o3-mini-high. This was to avoid repetitively coding the plots. I avoided writing a routine to plot, since there are many different types of plots that were made for specific experiments. MATLAB automatically upcasts to double precision, so wherever single precision was necessary, we manually cast the inputs and the data to single precision before any processing.

1 Task 1

In this section we describe the implementation of our routines and their time and space complexities. We will empirically evaluate the code using simple interpolation problems that can be checked manually (by hand, for example).

1.1 Barycentric Form 1

We use the recursive code provided in the notes to compute the array of coefficients (labeled *gamma*) for the Barycentric 1 form. The algorithm reduces computation by reusing variables calculated previously to update the values. The **time complexity** of the naive method to evaluate the β's, which would be to compute the full product for each node is $O(n^2)$ specifically we get the number of computations:

$$((n-1)+n) * n + n = 2n^2.$$

For the recursive algorithm it can be shown that the **complexity** constant reduces from 2 to $\frac{3}{2}$. Since we only store the final array of the coefficients, the **space complexity** is $O(n)$. Our implementation of the Barycentric 1 evaluation simply computes the required sum using vectorization on MATLAB. We use the polynomial product routine wherever appropriate, since this is know to evaluate polynomials to a high degree of relative accuracy (Higham 2002 Accuracy and Stability of Numerical Algorithms, Second Edition). For example, $\omega_n(x)$ can be computed using the product routine. We also add functionality in this code to compute the condition numbers $\kappa(x, n, y)$ and $\kappa(x, n, 1)$, which simply have appropriate absolute values within the code. The Barycentric 1 routine computes the numerator for the general condition number. The error handling is similar to Barycentric 2 (i.e when $x \approx x_i$) and is discussed below.

We test the routine for simple monomial functions and also the function $f(x) = |x|$ for Barycentric 1. Here is what the output polynomial looks like for 5 uniform nodes in the interval $[-2, 2]$. This is what we expect to see. We also compute the Lebesgue constant which turns out to be approximately 31.0. We compute this by finding the maximum in the array of κ's with $y = 1$. This was tested for a thousand test points in the interval $[-2, 2]$. We use this approach for every subsequent analysis.

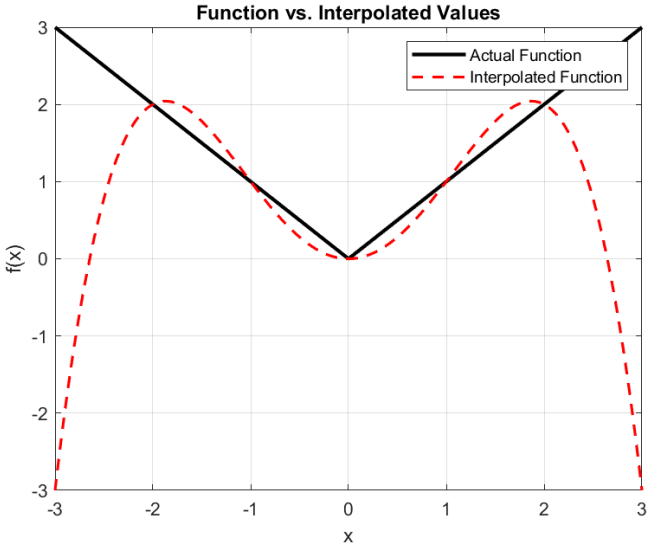


Figure 1: Barycentric 1 Interpolation

This validates our code for Barycentric 1, and the interpolating points are clear from the diagram.

1.2 Barycentric Form 2

The first routine to compute the function values and the array of betas, we assume 3 different type of possible nodes:

- Uniform Nodes
- Chebyshev 1st kind
- Chebyshev 2nd kind

For each type of node, we have reduced formulas to calculate the β's given the structure. They are given by the following formulas, respectively:

- $\beta_{i+1} = \beta_i \frac{n-i}{i+1}$ where $\beta_0 = 1$. This is easily verified from the relationship to γ_i.
- $\beta_j = (-1)^j \sin\left(\frac{(2j+1)\pi}{2n+2}\right), \quad 0 \leq j \leq n.$
- The β's for this is a simple array that has 0.5 in the end and the start and 1's in the middle with alternating signs.

For the last case we simply create the array with no computation ($O(1)$), and for the others we simply create an array of indices and apply the formulas directly. These computations are at most $O(n)$ in **space** and **time**.

The other major computation of this routine is producing the nodes for each kind. In MATLAB, these are $O(n)$, since we simply produce an array of indices and evaluate the respective functions at the indices. For uniform nodes, we simply use the *linspace* function in MATLAB. For the other two, we again simply apply the formulas, with at most $O(n)$ complexity.

Finally, we evaluate the function at the nodes. The complexity for this depends on the function we are evaluating and the form available.

We use the code provided from Berrut and Trefethen, Siam Review Vol. 46 No.3 in the notes to evaluate the Barycentric form of the polynomial at an array of values, and also use the streamlined code provided to produce the β array for Chebyshev points of the second kind. The **time complexity** of the evaluation is clearly $O(n)$ which is also obvious from the formula:

$$p_n(x) = \left[\sum_{j=0}^{n} \frac{\beta_j}{x - x_j}\right]^{-1} \sum_{i=0}^{n} \frac{y_i\,\beta_i}{x - x_i}$$

which is a form close to our implementation. Counting the operations we get

$$((n+1)(1+1+1)+n)+1+((n+1)(1+1)+n) = 7n+6 = O(n)$$

On MATLAB, we use arrays and element-wise operations to optimize the code to avoid loops as much as possible, which streamlines the computation. We take an input of the arrays *xvals*, *fvals*, *nodes* and *betas* and output the Lagrange (Barycentric 2) interpolating polynomial evaluated at *xvals*. The **space complexity** of the evaluation is also $O(n)$ even if we include the input and temporary space requirements.

Note that for the evaluation routines, we evaluate at an array of values, so we in fact get m times the relevant number of computations to get the full number of computations performed. This does not change the order since m is usually not a parameter, but some fixed value in a particular experiment/application. Also note that for both Barycentric routines, we do the following error handling:

To handle cases when $fl(x - x_i) < tol$ is too small or close to 0, we simply assign the given data i.e $f_i = y_i$ to $p_n(x)$ i.e. $p_n(x_i) = p_n(x)$. This avoids overflow (since in the computations we divide by this difference). However, this adds an extra layer of computation where we have to check for each value in x (the array of evaluation points) the distance to all the nodes and then find the minimum distance. If this distance is smaller than the tolerance, we do the manual assignment. I implemented this in MATLAB by assigning the values to p after the computation, and replacing the possibly erroneous values (or "NaN" values if the x value matches the value at a node). This simplifies the code, and the order is still $O(mn)$ for the operations since for each x, we calculate n distances, and then perform a sort to get the minimum value, and then a simple comparison. The sorting complexity depends on the algorithm used by MATLAB, but in any case, this does not make our routine considerably slower than before. In many cases, the evaluation points will not coincide with the nodes.

We now repeat the experiment in the paper by Berrut and Trefethen, to validate these routines for Barycentric 2 using Chebyshev points of the 2nd kind, for the function

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

Here are our results:

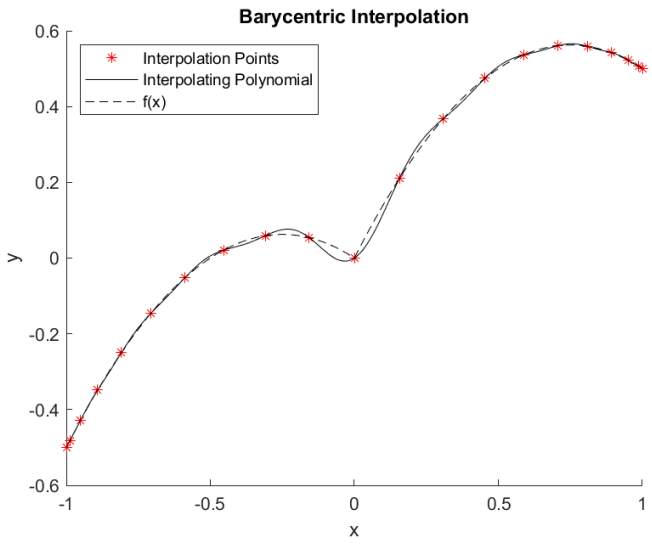


Figure 2: Berrut and Trefethen Experiment

In my testing, Chebyshev points of the first kind tested similarly. The unfirom points do interpolate, but Runge's phenomenon is quite apparent for this function: One less point to 'zoom' on the edges, we get the following result for uniform points This in particular motivates the use for Chebyshev nodes, and also gives insight into intervals of interest for testing.

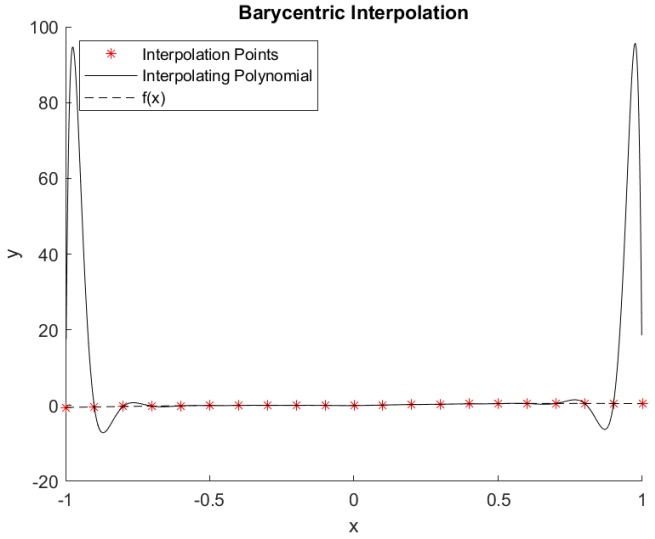


Figure 3: Runge's Phenomenon

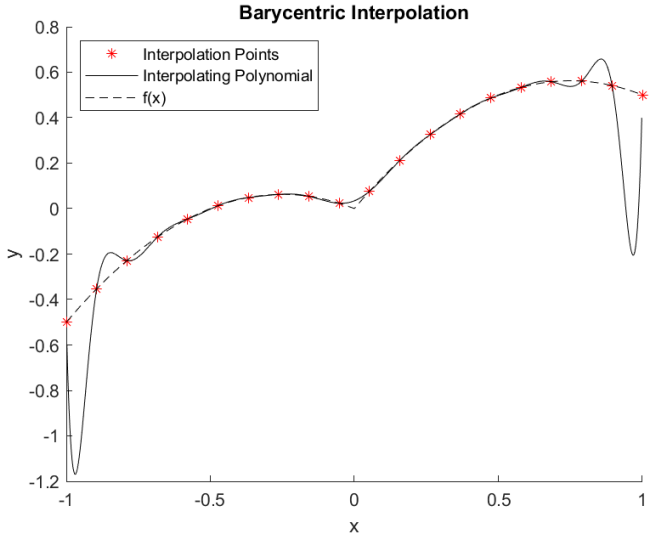


Figure 4: Closer look

1.3 Newton Divided Differences

For this routine, we only compute the "left" column of the divided difference table, in a single work vector/array, allowing us to have a **space complexity** of $O(n)$. This however means that we do not store the rest of the divided difference table, and lose the ability to interpolate a (contiguous) subset of the original data points quickly. Surprisingly, we can still add an additional point to interpolate in $O(n)$ time and space. The algorithm is implemented fairly succinctly in MATLAB due to vectorization, allowing us to access and update segments quickly. The algorithm is explained easily with an example:

$$\begin{array}{llll} y_0 & \to y_0 & \to y_0 & \to y_0 \\ y_1 & \to y[x_0, x_1] & \to y[x_0, x_1] & \to y[x_0, x_1] \\ y_2 & \to y[x_1, x_2] & \to y[x_0, x_1, x_2] & \to y[x_0, x_1, x_2] \\ y_3 & \to y[x_2, x_3] & \to y[x_1, x_2, x_3] & \to y[x_0, x_1, x_2, x_3] \end{array}$$

We can use a "top down" approach. We start with a vector of all function values, y. At every step of the divided difference, we subtract two of the previous divided differences and divide by the differences of the appropriate x values. On the i-th step, we need so subtract from the array $y[i+1, end]$ the array that is shifted one up i.e. $y[i, end-1]$. Once this is done we simply do element-wise division from a "difference" array D, which is constructed as follows: at every step, we start on the bottom and "move up" i steps successively to get the

difference e.g. on the array.

$$x_0$$
$$x_1$$
$$x_2$$
$$x_3$$

we get the differences $(x_3 - x_2), (x_2 - x_1), (x_1 - x_0)$ on step 1. On the next step, we skip two values each time, still reading successively from the bottom, and terminating when we cannot skip 2 steps (i in general) to the top to a value, to get the differences $(x_3 - x_1), (x_2 - x_0)$. This means we subtract from the array $x[i+1, end]$ (reading from the bottom) the array $x[1, end - i]$ (reading from the top) to get D. Note that the size of the array D and the part of the work vector that needs to be updated in the i-th step match. We continue until $i = n - 1$. This makes obvious too that we get $O(n^2)$ **computations** for the divided difference table.

To evaluate the interpolating polynomial in the Newton form, we use a form of the Horner's Method. This too is explained easily with an example. For a polynomial in Newton basis form, we can rewrite it as follows (this is in degree 4 as well):

$$p_4(x) \;=\; \alpha_0 + \alpha_1 \,(x{-}x_0) + \alpha_2 \,(x{-}x_0)(x{-}x_1) + \alpha_3 \,(x{-}x_0)(x{-}x_1)(x{-}x_2) + \alpha_4 \,(x{-}x_0)(x{-}x_1)(x{-}x_2)(x{-}x_3).$$

$$p_4(x) \;=\; \Big(\Big(\Big(\alpha_4 \,(x - x_3) + \alpha_3\Big)(x - x_2) + \alpha_2\Big)(x - x_1) + \alpha_1\Big)(x - x_0) + \alpha_0.$$

which motivates an algorithm to evaluate the polynomial inside out. We compute a difference, followed by a multiplication, followed by an addition, recursively. This gives us $3n$ computations, for order $O(n)$ for **evaluation** of the Newton divided difference. Note that the divided difference code gives us the array α and the array x is given. We also adapt Horner's method to evaluate an array of input x values.

We can run one of the experiments in the later tasks using the three different types of nodes to get the following results (we use the interval $[-10, 15]$ and use 4 points to interpolate the function $f(x) = (x-2)^9$):

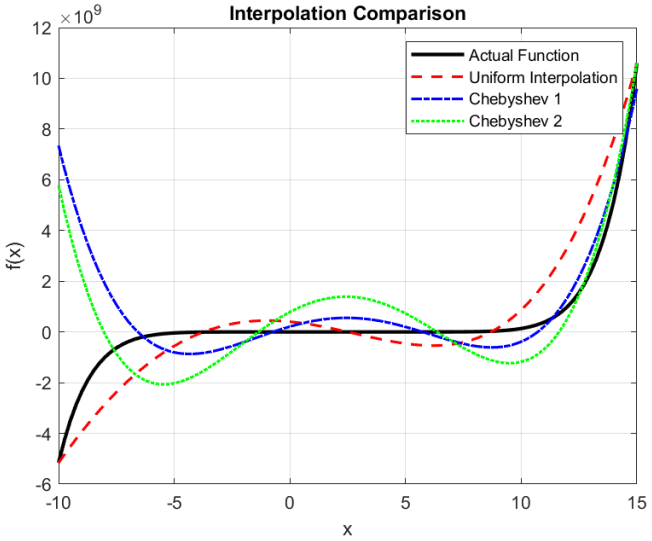


Figure 5: Divided Differences Example

When we increase the number of points to m=9, we expect convergence to the actual polynomial. Indeed, this is what we observe:

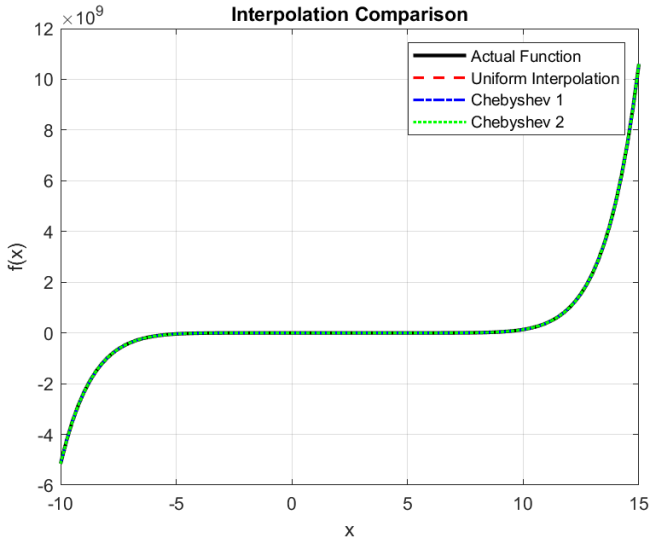


Figure 6: Divided Difference with points=degree

Note that this indirectly also tests the code for producing nodes, the product form of a polynomial, as well as the Horner's Method routine. Other smaller routines are tested initially in the testing driver live script on MATLAB (interpolation.mlx).

1.4 Product form of a Polynomial

Whenever we have a polynomial available in the product form i.e.

$$p_n(x) = \alpha \prod_{i=1}^{n} (x - \rho_i)$$

we use the streamlined code provided in the notes on the routines in the homework to evaluate. This is implemented as recursive multiplication of the differences of x from the nodes, and a final multiplication with the scaler parameter. We modify it to return values at an array of input values, and again use MATLAB functionality to streamline (element-wise multiplication in this case). The **time complexity** for this is $O(mn)$ where m is the number of values where we require the evaluation. For a single value, we get a total of $n-1+1+n = 2n$ computations. For m such values, we then get $2mn$ computations. The **space complexity** is clearly $O(n)$ for the input arrays and auxiliary and output memory.

Note that this routine was used extensively to evaluate products where possible, and this is indicated in the code.

1.5 Ordering

We implement three sorting algorithms for three different types of orderings:

- For ascending and descending orders, we use the standard *sort* function in MATLAB. The functions uses the quick sort algorithm which has **time complexity** $O(n \log n)$.
- For Leja ordering, we implement the routine by computing the products for successive choice of nodes to each of the points, and then choosing the index with the maximum product of the absolute differences. The algorithm gives us $O(n^2)$ **time complexity**, since for each point we have roughly $O(n)$ computations.

See the code appendix for details of the implementation for Leja ordering. The **space complexity** for the sorting routines is $O(n)$, since we work on the given array, or make an auxillary array of products for each outer iteration for Leja ordering. There are related points called fast Leja points that have similar desirable properties and are faster to compute, given in the paper by Reichel (Fast Leja Points).

2 Task 2

Task 2 focuses on the following function of interest:

$$f(x) = (x-2)^9 = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512$$

where the coefficients are available by the binomial theorem. In general we get

$$f(x;\rho,d) = (x-\rho)^d$$

Where we have a repeated root *rho* of multiplicity d.

We first find the points that this function interpolates, in an interval of interest (i.e. one that contains the root). We choose the interval $[a,b] = [-10,15]$ which is symmetric interval around 2. We also set the the number of points we want to use to interpolate. Since we have a degree $m = 9$ polynomial, we have $m+1 = 10$ parameters in the interpolating polynomial. We thus use 10 points of each kind (uniform, Chebyshev 1, Chebyshev 2) and evaluate the function in the chosen interval to get the 'apriori' data (x_i, f_i) for $0 \leq i \leq m$. Note also that we have the result that the pointwise interpolation error (for a C^{n+1} function) is given by

$$E_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}\,\omega_{n+1}(x)$$

For our degree d C^∞ function this is in fact, quite simple to compute:

$$E_n(x) = 0$$

for $n \geq d$ and for $n = d-1$ we get

$$E_n(x) = w_d(x)$$

This means one test of accuracy is that we can choose $m = 9$ as our parameter and get no error for our interpolating polynomial (in the interval).

In the testing for this task, it appeared more appropriate to plot the absolute error rather than the relative error, since our computations are so direct, and since the root is a very small number ($\rho = 2$) i.e. close to 0. Even on this problem, we can begin to distinguish between the accuracy of the Barycentric 1 (modified Lagrange) and the Newton divided differences:

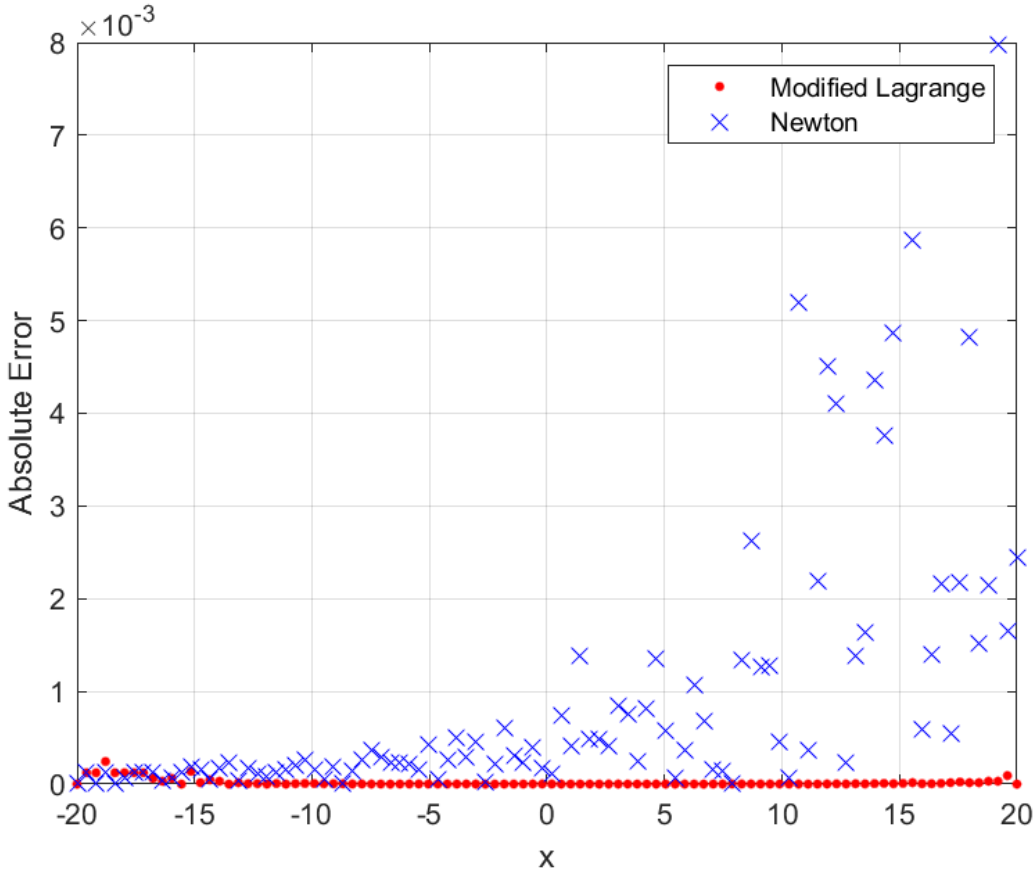


Figure 7: Absolute error in computed $p_n(x)$ for 10 equally spaced points x_i in increasing order

Another behavior we begin to notice is that that we interpolate using lesser points, the error grows quite large at the root or close to the root (this is also what makes it difficult to plot the relative errors - there is a

big outlier around the root). This plot was ran for double precision accuracy for both the "true" evaluation and the interpolating evaluation. To see more comprehensive results, we switch our codes to single precision for the interpolation routines. In MATLAB, we do explicit casting for this. We get the following absolute error plot:

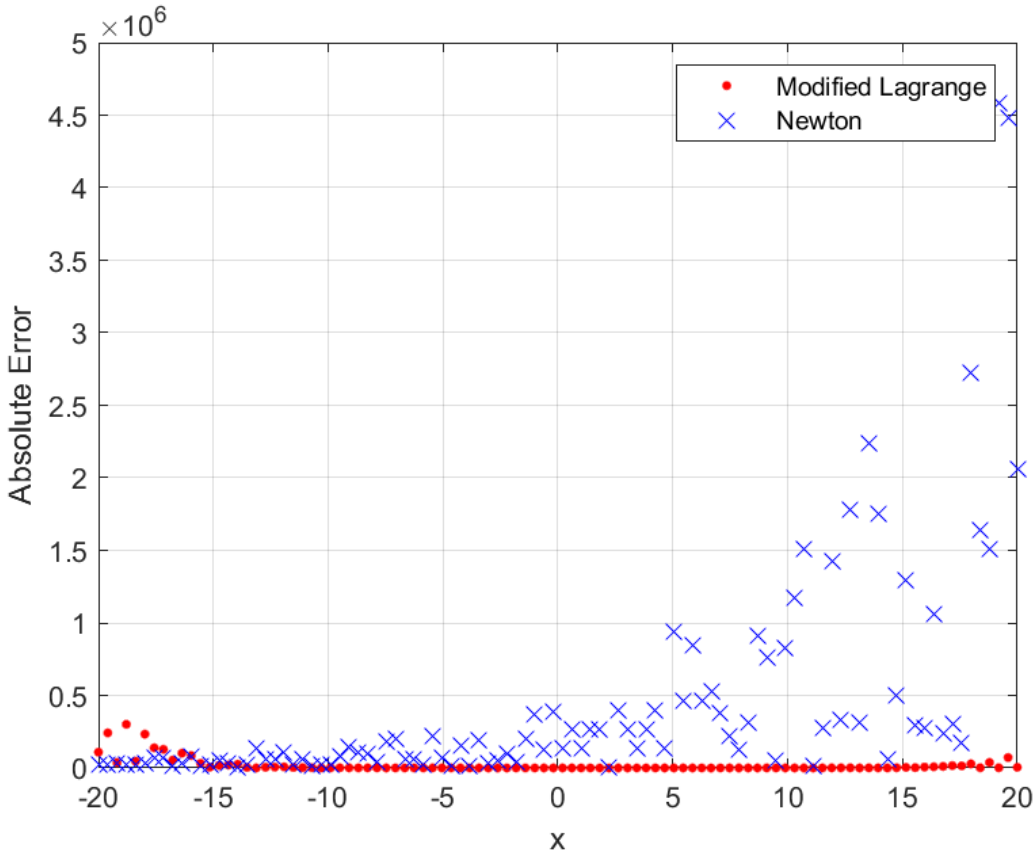


Figure 8: Single precision for routines

The Lebesgue constant for these experiments was approximately $\Lambda_n = 17.6433$. Now we add all three routines for uniform points:

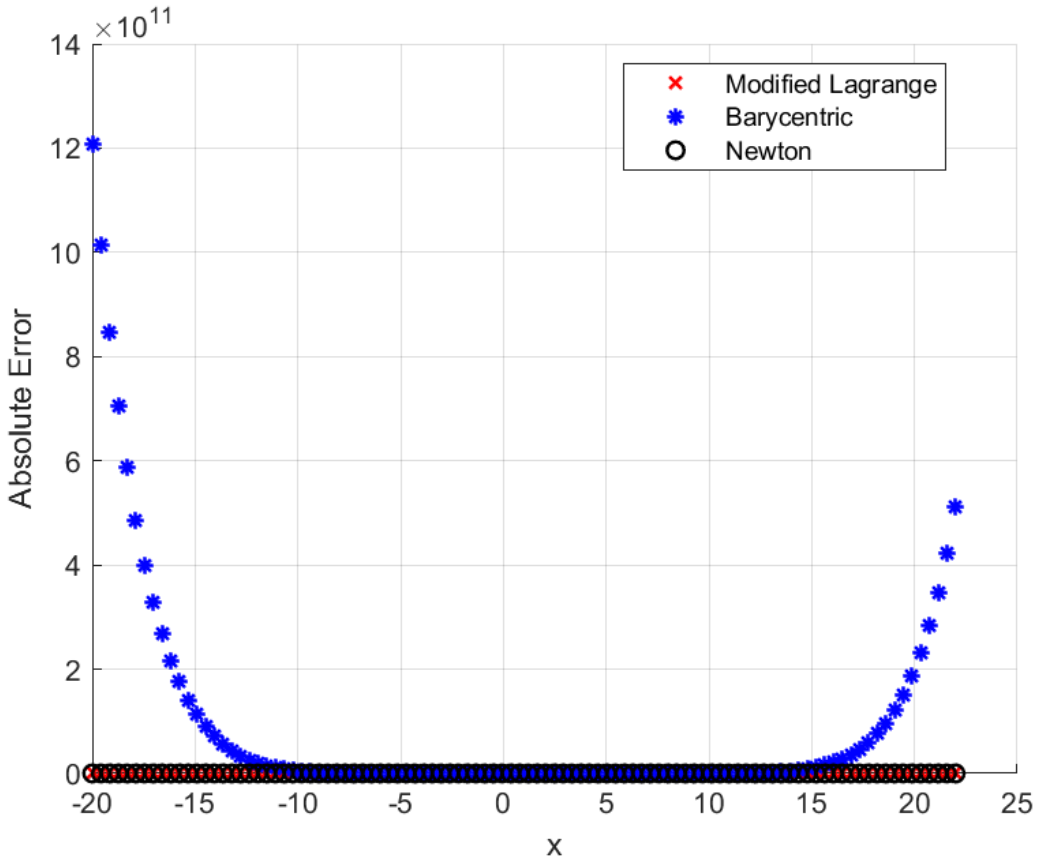
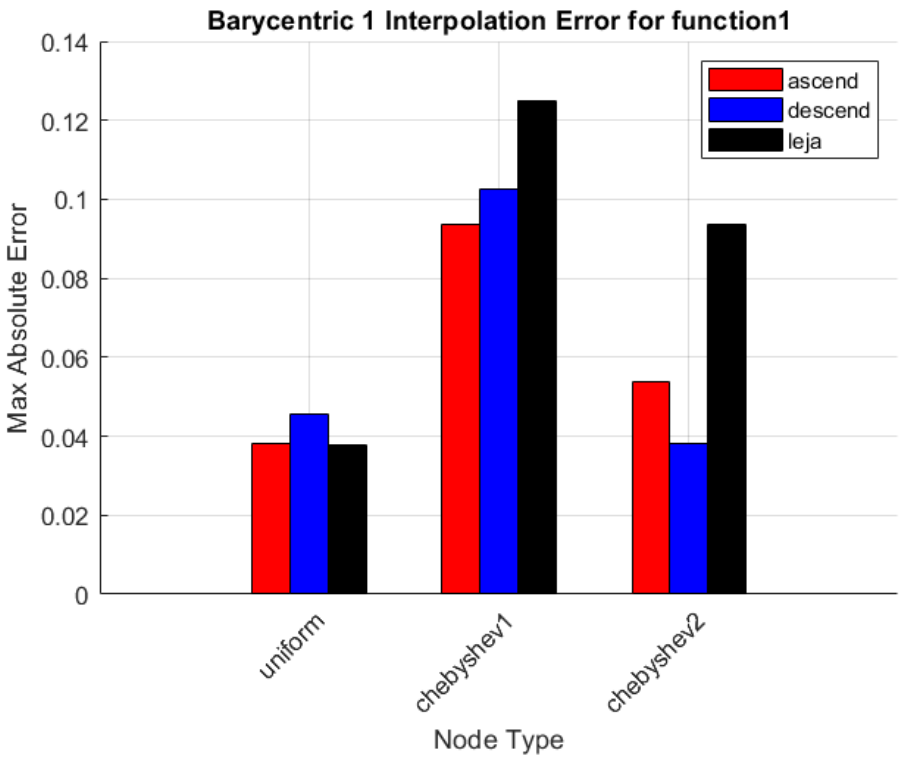


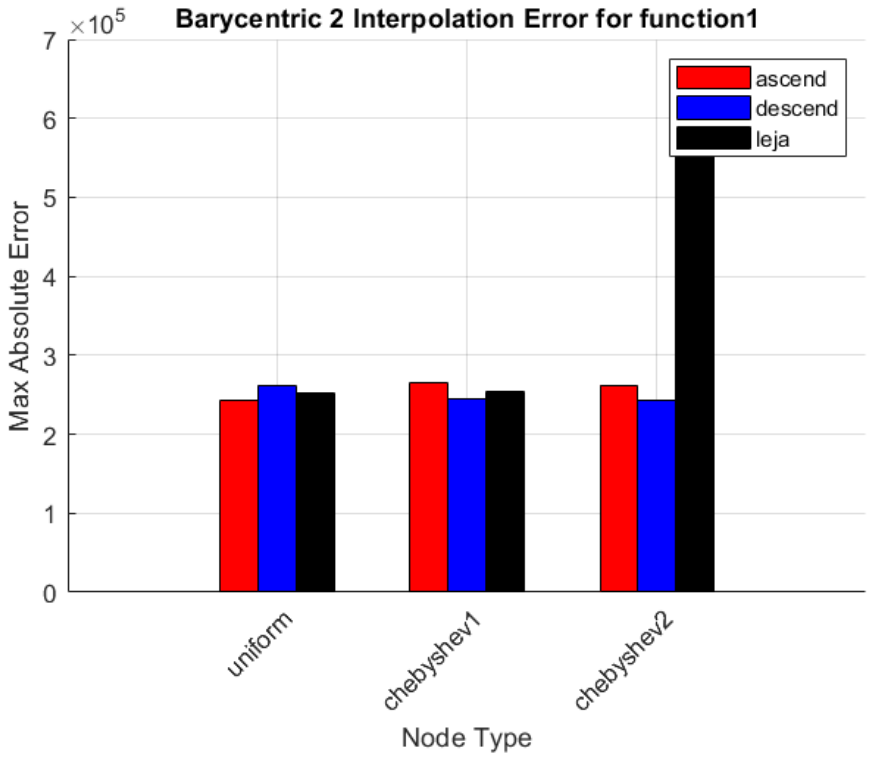
Figure 9: All Routines

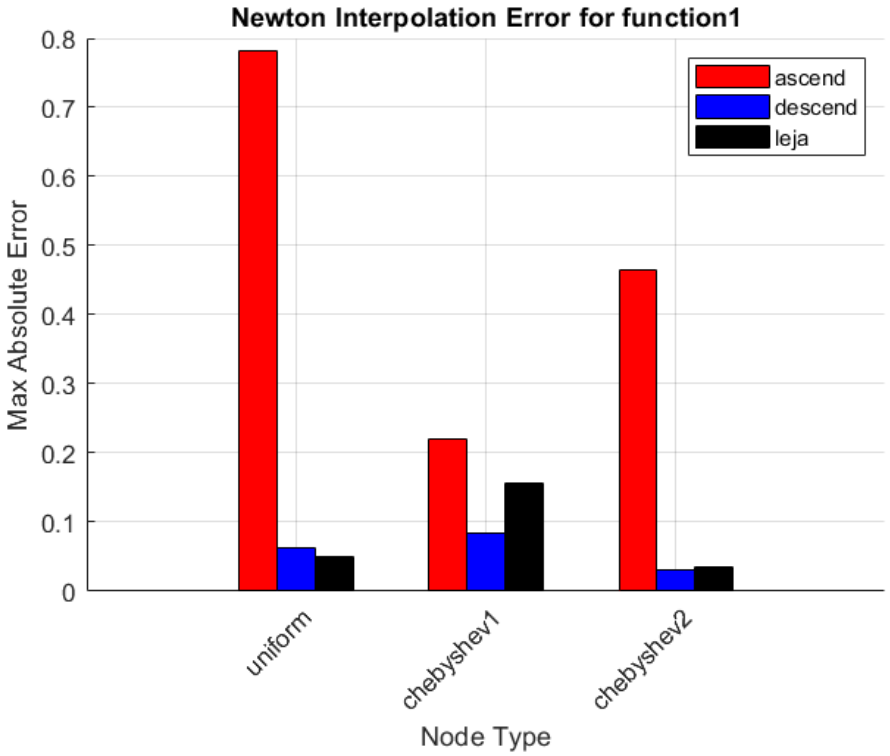
Note the similarity to the results in Higham, even though a different function was used. For descending order, we get the same results for this function and uniform points. Now that we have visualized the expected results, we can summarize our results across experiments for each node type, and for each ordering, per method. I chose to present the Lebesgue constant and the maximum of the condition numbers H_n, as well as the infinity-norms of the errors for each experiments (the absolute errors):

Node Type	Lebesgue Constant	H_n
uniform	17.6433	242461.0000
chebyshev1	60.2688	245241.3125
chebyshev2	19.0000	242744.9062

Table 1: Conditioning Metrics for function 1


Barycentric 1 Interpolation Error for function1
Max Absolute Error
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
ascend
descend
leja
uniform
chebyshev1
chebyshev2
Node Type


Barycentric 2 Interpolation Error for function1
×10^5
Max Absolute Error
7
6
5
4
3
2
1
0
ascend
descend
leja
uniform
chebyshev1
chebyshev2
Node Type



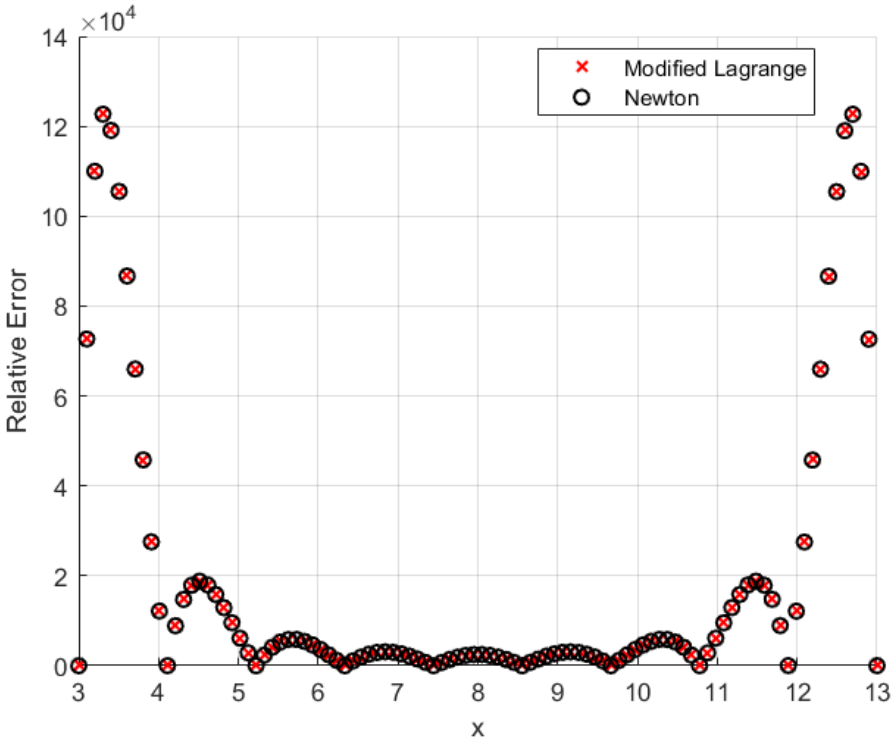
We can now compare the methods overall by looking at the axes (y-axis). As expected, Barycentric 1 performs the best, and Barycentric 2 is very ill-conditioned. For the max norm, barycentric 1 diverges on the ends of the interval, raising the max norm. For Newton, the choice of nodes reduces the error significantly. Changing the ordering from ascending also has improves performance.

3 Task 3 and 4

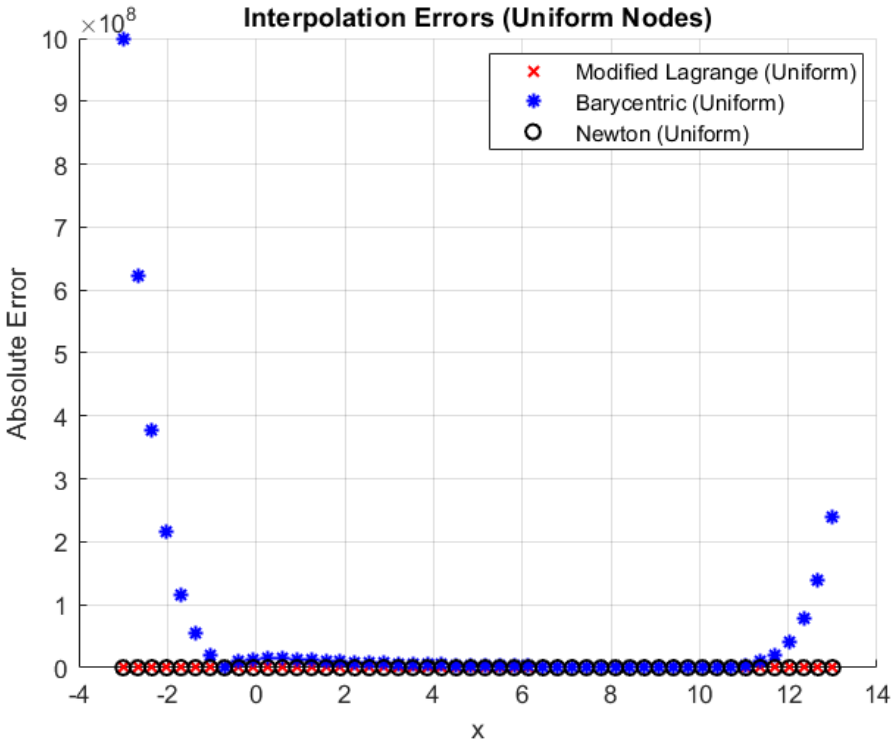
For this task, we show results for the function:

$$f(x) = \prod_{i=1}^{d} (x - i)$$

which has roots at the first d integers. We first look at the relative errors of just the Modified Lagrange and Newton:



They perform similarly, and we can see Runge's phenomenon, where the relative error is small close the nodes as well. When we add barycentric, we get the same phenomenon as the previous function on the relative errors:



Node Type	Lebesgue Constant	H_n
uniform	17.6433	997920000.0000
chebyshev1	60.2688	1001734144.0000
chebyshev2	19.0000	998176960.0000

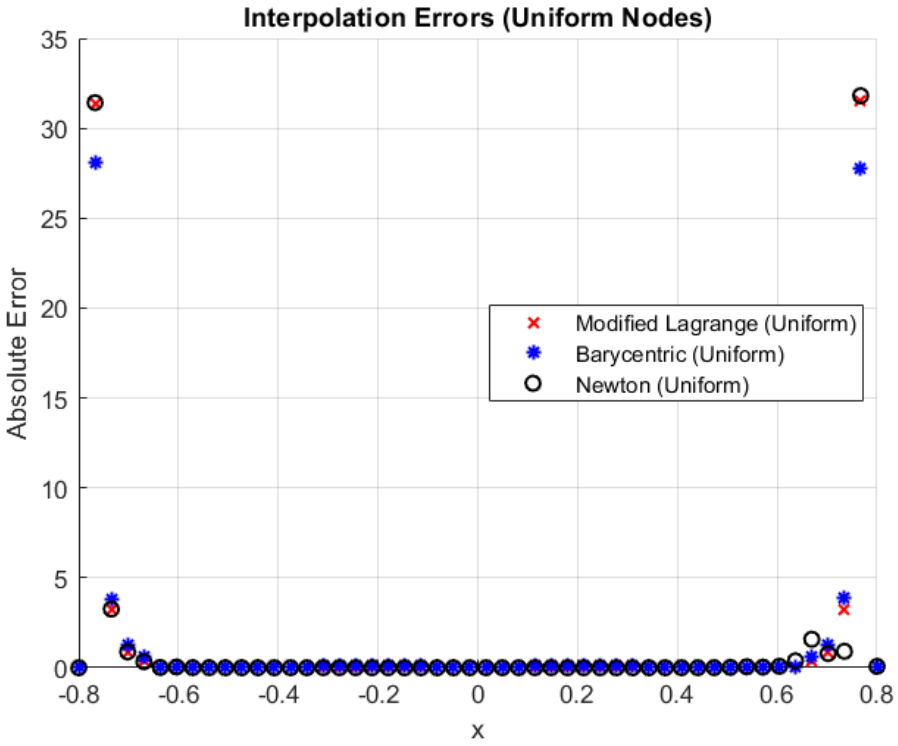
Table 2: Conditioning Metrics for function2

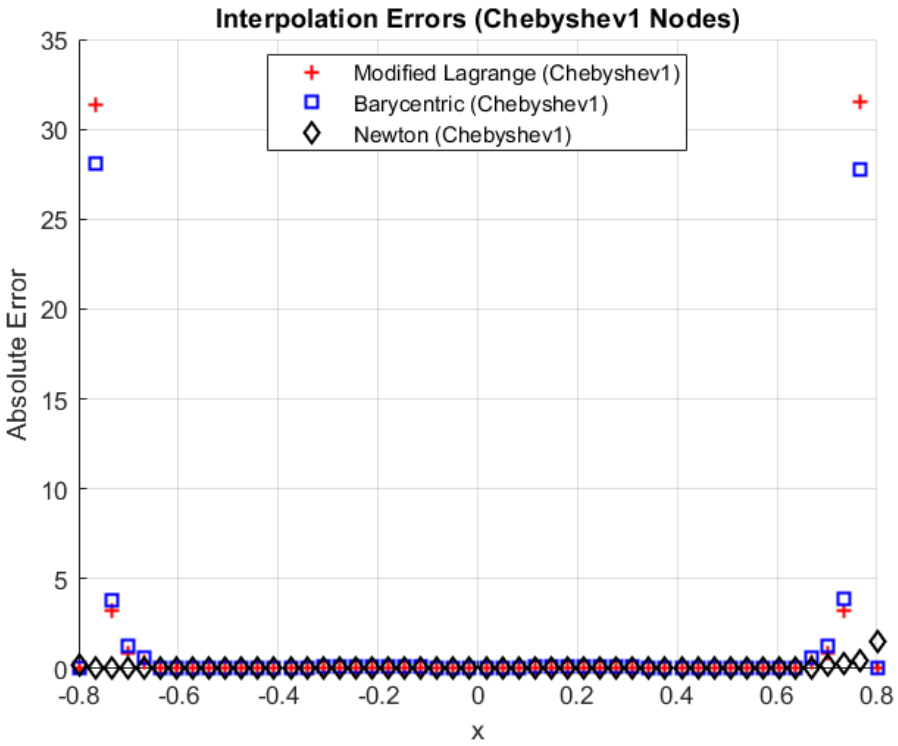
4 Task 5

In this task we interpolate the Runge function, namely:

$$f(x) = \frac{1}{1 + 25x^2}$$

We try to replicate the experiment in Higham's paper in this task, by using first 30 uniform nodes in the interval $[-1, 1]$, and then using Chebyshev nodes (while we give summarized results for all node types and orderings).


Interpolation Errors (Uniform Nodes)
Absolute Error
x
Modified Lagrange (Uniform)
Barycentric (Uniform)
Newton (Uniform)
35
30
25
20
15
10
5
0
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8

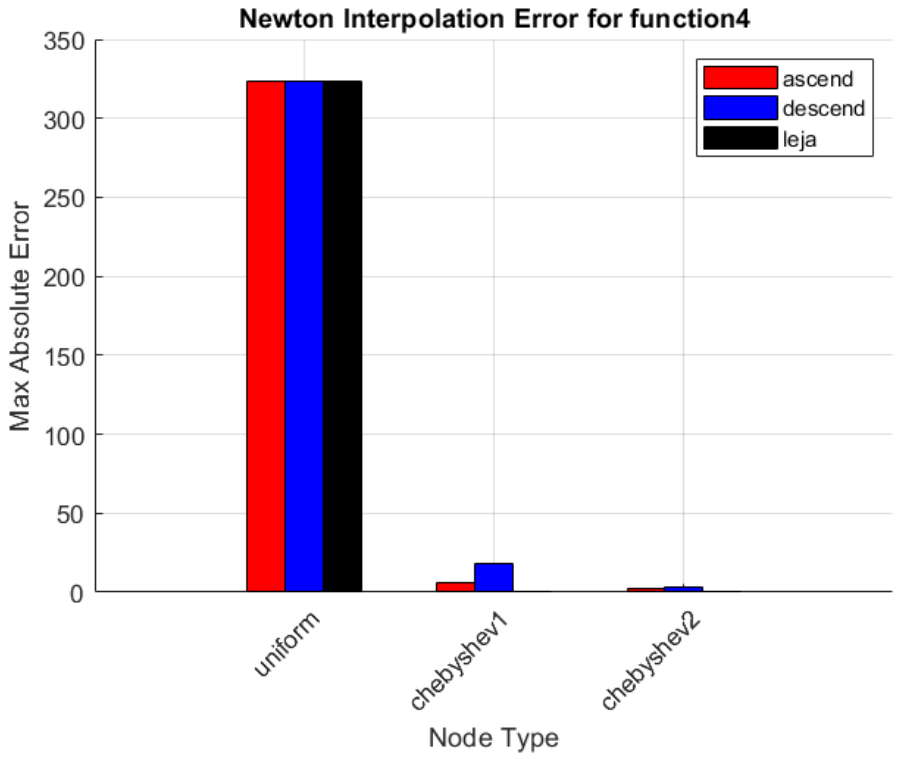

Interpolation Errors (Chebyshev1 Nodes)
Absolute Error
x
Modified Lagrange (Chebyshev1)
Barycentric (Chebyshev1)
Newton (Chebyshev1)
35
30
25
20
15
10
5
0
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8

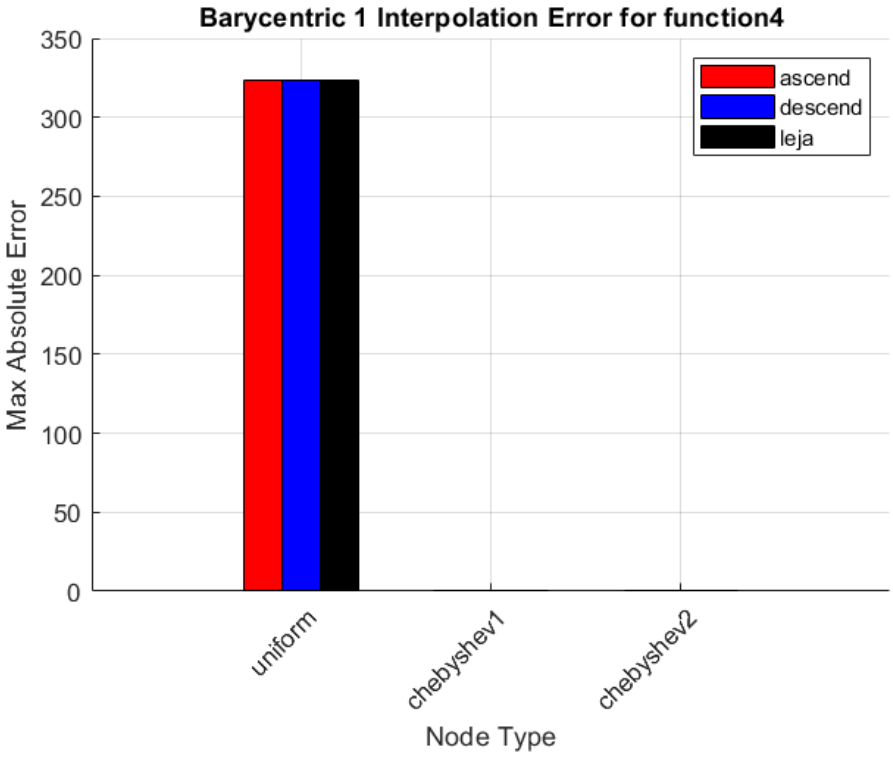
We can see that the results of the experiments are similar. We get the following Lebesgue and other constants:

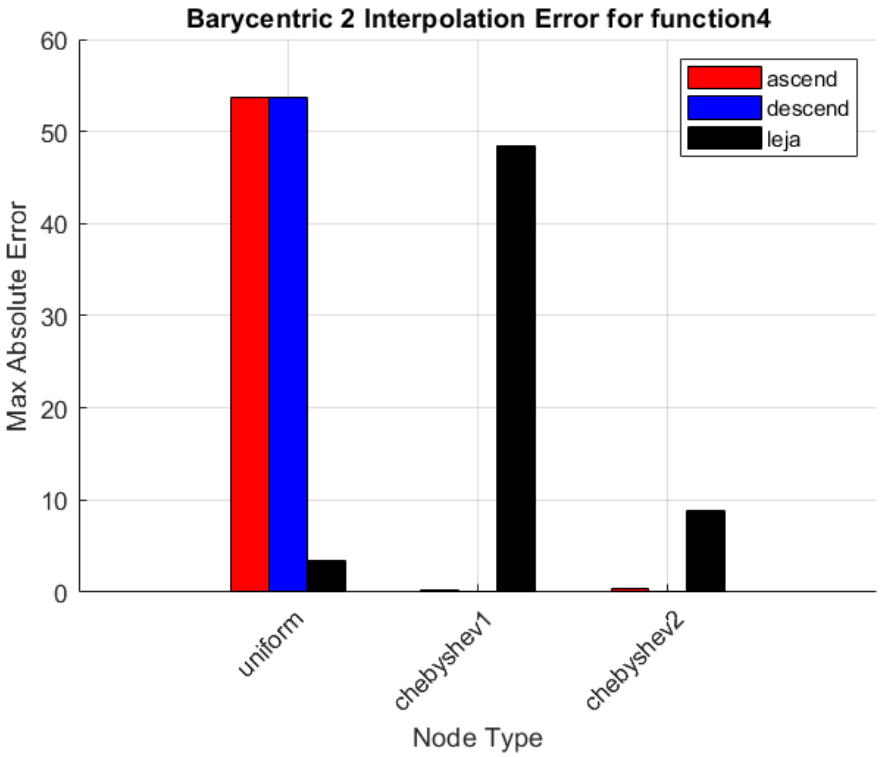
Node Type	Lebesgue Constant	H_n
uniform	3325811.5000	53.7101
chebyshev1	493.0692	0.0034
chebyshev2	59.0000	0.0024

Table 3: Conditioning Metrics for function 4

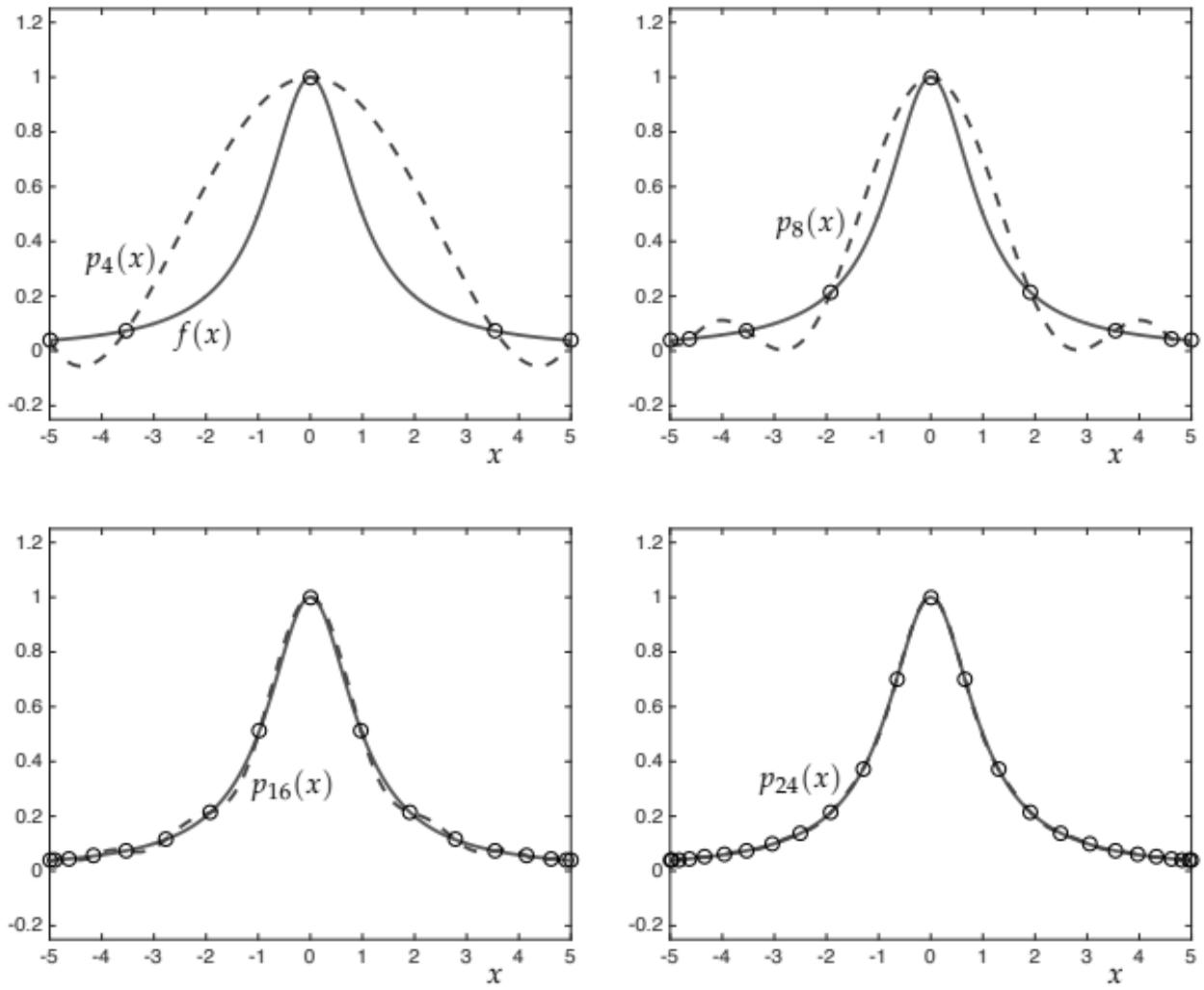
This corroborates the error results summarized in the following tables:




Barycentric 1 Interpolation Error for function4
Max Absolute Error
350
300
250
200
150
100
50
0
ascend
descend
leja
uniform
chebyshev1
chebyshev2
Node Type


Barycentric 2 Interpolation Error for function4
Max Absolute Error
60
50
40
30
20
10
0
ascend
descend
leja
uniform
chebyshev1
chebyshev2
Node Type

We can see that using different node types has a significant impact on the error - but Barycentric 2 is still the choice of interpolation for this function as seen from the axes, specially when either Chebyshev kind nodes are used (Leja ordering in this case causes issues). The ordering has minimal impact on the maximum error, since (as we saw in the Higham experiment), this only changes which side of the interval the error becomes large. The relative error plots look similar., and we get **convergence for Chebyshev nodes for the Runge's function**:



5 Appendix

5.1 Lagrange Barycentric 1

```
function [gamma,fvals] = barycentric1(func, n, nodes)
%BARYCENTRIC1 Computes the array of Barycentric coefficients gammas and
%the function values at the nodes
%   This routine takes in as parameters a function, the number of nodes n,
%   and the array of nodes and computes the coefficients gammas required
%   for interpolation in Barycentric form 1, as well as the function values
%   at the nodes
fvals = func(nodes);

if length(nodes) < n
    error('nodes must have at least n elements.');
end

if n == 1
```

```
    m = 1;  % Only one node: trivial case.
    return;
end

m = zeros(1, n);

m(1) = nodes(1) - nodes(2);
m(2) = nodes(2) - nodes(1);

for k = 2:(n-1)
    p = 1;
    for i = 0:(k-1)
        t = nodes(i+1) - nodes(k+1);
        m(i+1) = t * m(i+1);
        p = -t * p;
    end
    m(k+1) = p;
end

gamma = 1 ./ m;
end
```

5.2 Barycentric 1 Evaluation

```
function p = bary1Eval(xvals,fvals,nodes,gammas)
%BARY1EVAL Summary of this function goes here
%   Detailed explanation goes here

omega = polyProduct(1,nodes,xvals);

diffMat = xvals.' - nodes;
numVec = fvals .* gammas;
termMat = numVec ./ diffMat;

p = sum(termMat, 2);
p = p' .* omega;

tol = 1e-12;

dists = abs(xvals - nodes');
[minDist, minDistIdx] = min(dists, [], 2);
closeMask = (minDist < tol);
p(closeMask) = fvals(minDistIdx(closeMask));
end
```

5.3 Lagrange Barycentric 2

```
function [beta,fvals] = barycentric2(func,n,flag)
%BARYCENTRIC2 Summary of this function goes here
%   Detailed explanation goes here

if nargin < 2
    flag = 'uniform';
end

switch lower(flag)
    case 'uniform'
        [nodes, beta] = uniform(n);
    case 'cheby1'
        [nodes, beta] = chebyshev1(n);
```

```
    case 'cheby2'
        [nodes, beta] = chebyshev2(n);
    otherwise
        error('Unknown ordering flag: %s. Use "uniform", "cheby1", or "cheby2".', flag);
end

fvals = func(nodes);


end

function [x, betas] = chebyshev1(n)

j = 0:n;
x = cos( (2*j + 1)*pi / (2*n + 2) );
betas = ((-1).^j) .* sin((2*j + 1)*pi / (2*n + 2));

end

function [x, betas] = chebyshev2(n)

j = 0:n;
    x = cos((j * pi) / n);

deltas = ones(1, n+1);
deltas(1)   = 1/2;
deltas(end) = 1/2;

j = 0:n;
betas = ((-1).^j) .* deltas;

end
```

5.4 Barycentric 2 Evaluation

```
function p = bary2Eval(xvals,fvals,nodes,betas)
%BARY2EVAL Summary of this function goes here
%   Detailed explanation goes here

n = size(xvals);
numer = zeros(size(xvals));
denom = zeros(size(xvals));
for j = 1:n+1
    xdiff = xvals - nodes(j);
    temp = betas(j) ./ xdiff;
    numer = numer + (temp*fvals(j));
    denom = denom + temp;
end

p = numer ./ denom;

dists = abs(xvals - nodes');
[minDist, minDistIdx] = min(dists, [], 2);
closeMask = (minDist < tol);
p(closeMask) = fvals(minDistIdx(closeMask));
end
```

5.5 Newton's Divided Differences

```
function [dd,fvals] = newtondd(func,xvals)
```

```
%NEWTONDD Compute divided difference table
%   The function takes as input a function parameter and x values and
%   outputs the top row/left column of the divided differences table for
%   interpolation assuming the x values are indexed from top to bottom in the array

n = length(xvals);
fvals = func(xvals);

dd = fvals;
for i = 1:n-1
    dd(i+1:end) = dd(i+1:end) - dd(i:end-1);
    D = xvals(i+1:end) - xvals(1:end-i);
    dd(i+1:end) = dd(i+1:end) ./ D;
end

end
```

5.6 Horner's Method

```
function p = horner(alphas,xnodes,xvals)
%HORNER Evaluate the polynomial in the Newton Basis form
%   This functions takes in as input the alphas and xnodes that define a
%   polynomial in the newton basis and evaluates it at the points in xvals
%   and returns the matrix/vector of evaluations

p = zeros(size(xvals));
p(:) = alphas(end);
for i = length(alphas)-1 : -1: 1
    p = (p .* (xvals - xnodes(i))) + alphas(i);
end
end
```

5.7 Product form of a Polynomial

```
function p = polyProduct(alpha,nodes,xvals)
%POLYPRODUCT Polynomial Evaluation for the product form
%   This function takes in paramaters defining a polynomial in product
%   form, alpha and the nodes and evaluates it on the array xvals, returning an array of evaluations

p = zeros(size(xvals));
p(:) = alpha;
for i = 1:length(nodes)
    p = p .* (xvals - nodes(i));
end
```

5.8 Ordering

```
function x_ordered = order(x,flag)
%ORDER Compute ordered array for the given ordering type
%   This function takes an as input an unordered array and a flag
%   indicating the type of ordering, and returns the ordered array (Leja,
%   descending, or ascending)

if nargin < 2
    flag = 'ascend';
end

switch lower(flag)
    case 'ascend'
        x_ordered = sort(x, 'ascend');
```

```
    case 'descend'
        x_ordered = sort(x, 'descend');
    case 'leja'
        x_ordered = lejaOrder(x);
    otherwise
        error('Unknown ordering flag: %s. Use "ascend", "descend", or "leja".', flag);
end
end

function leja = lejaOrder(x)
n = length(x);
leja = zeros(size(x));

[~, idx] = max(abs(x));
leja(1) = x(idx);
x(idx) = [];

% Iteratively select the next node
for k = 2:n
    products = zeros(size(x));
    for j = 1:length(x)
        products(j) = prod(abs(x(j) - leja(1:k-1)));
    end
    [~, maxIdx] = max(products);
    leja(k) = x(maxIdx);
    x(maxIdx) = [];
end
end
```

5.9 Functions of Interest

```
function fvals = function1(xvals)
%FUNCTION1 Outputs the values of function 1 using the product method
%   This routine takes in the parameters
%   p and d to evaluate (x-p)**d for all xvals
d = 9;
p = 2;
nodes = zeros(1, d);
nodes(:) = p;
fvals = polyProduct(1,nodes,xvals);
end

function fvals = function2(xvals)
%FUNCTION2 Outputs the values of function 2 using the product method
%   This routine takes in the parameters
%   d to evaluate \prod(1,d) (x-i) for all xvals

d = 10;
nodes = 1:d;
fvals = polyProduct(1,nodes,xvals);
end

function fvals = function3(xvals,nodes)
%FUNCTION3 Outputs the values of function 3
%   This function is the last lagrange basis for a set of nodes (parameter.
nodes = nodes(1:end-1);

fvals = polyProduct(1,nodes,xvals);

div = polyProduct(1,nodes,nodes(end));
```

```
fvals = fvals / div;
end

function fvals = function4(xvals)
%FUNCTION4 Outputs the values of function 4
%   This routine takes in the xvals array and outputs fvals = 1/(1+25*x**2)
fvals = (xvals).^2;
fvals = fvals *25;
fvals = fvals +1;
fvals = 1 ./ fvals;
end
```