FCMII - Programming Report 2

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, as well as some organization, specifically CHATGPT 4-o. This was to avoid repetitively coding the plots, and design the testing more efficiently.

1 Introduction

In this project we want to test different interpolation techniques and compare the accuracy with respect to work done; this includes global interpolation, piecewise interpolation and cubic splines. We want to compare two different approaches to Cubic splines, both leveraging structured sparse system solves.

2 Routines and Task 1

We will first give examples of the routines on the basic Runge function, as an initial validation (we can then compare these values to the experiments conducted in the notes); The Runge function is

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

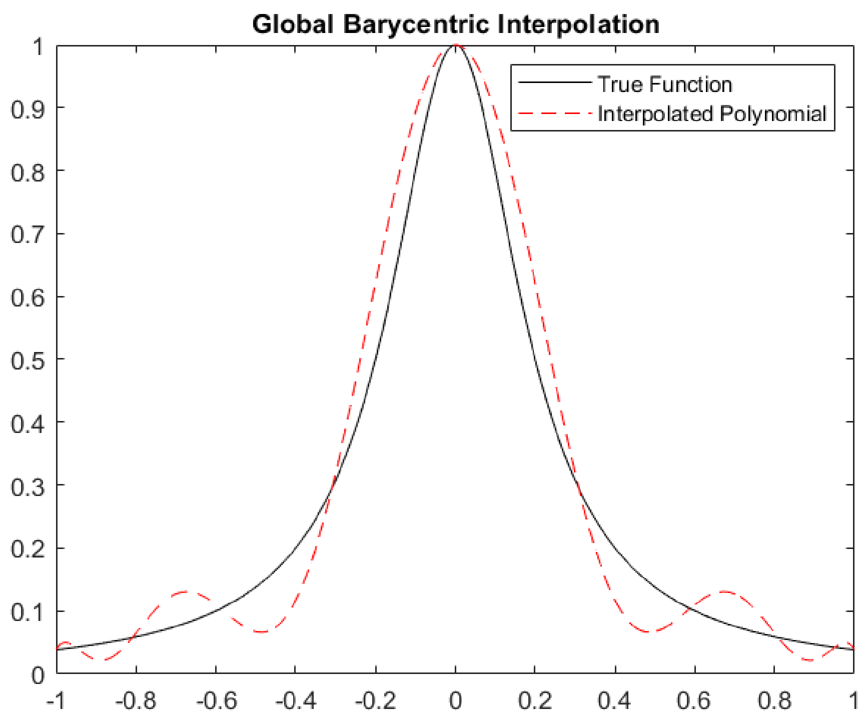
2.1 Global Interpolating Polynomial

We implement the global interpolating polynomial for an arbitrary function $f\,[a,b]$, using the two types of Chebyshev points (which gives us convergence as the number of points grow larger. We also write a routine to linearly transform any points from the interval $[a,b]$ to the interval $[c,d]$. We specifically use the Barycentric Form 1 (modified Lagrange) for the global interpolation. These methods were evaluated in the previous homework. Using the following parameters

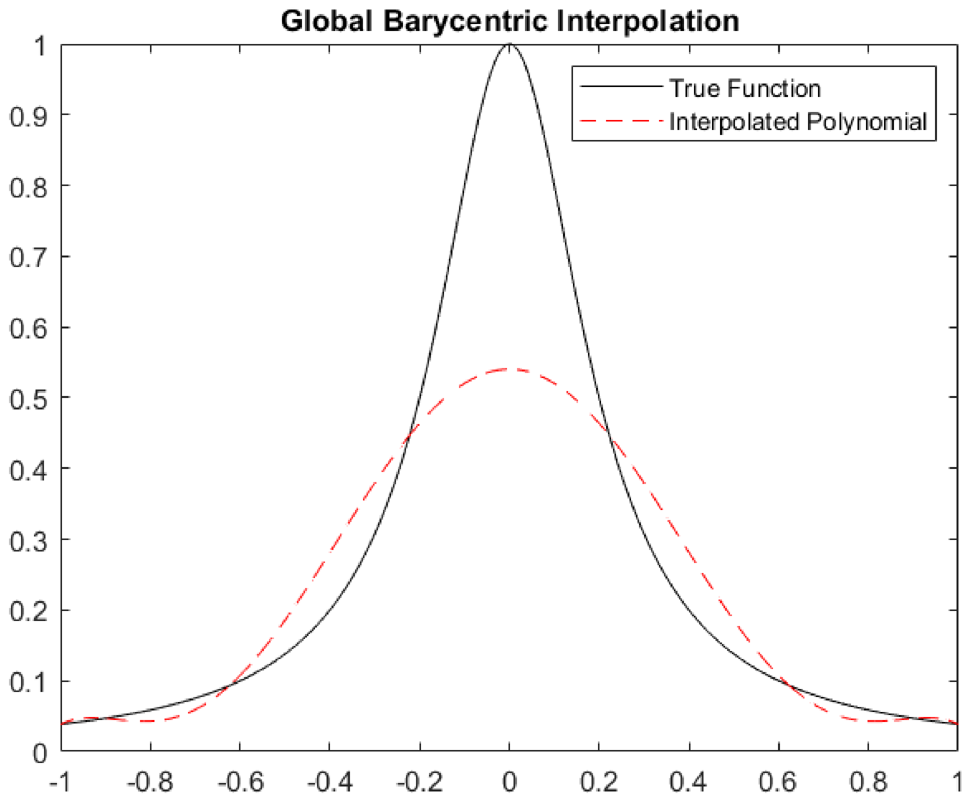
```
% Parameters
n = 10;
a = -1;
b = 1;
node_type = 'chebyshev2';

ncheck = 1000;
xcheck = linspace(a, b, ncheck)';
```

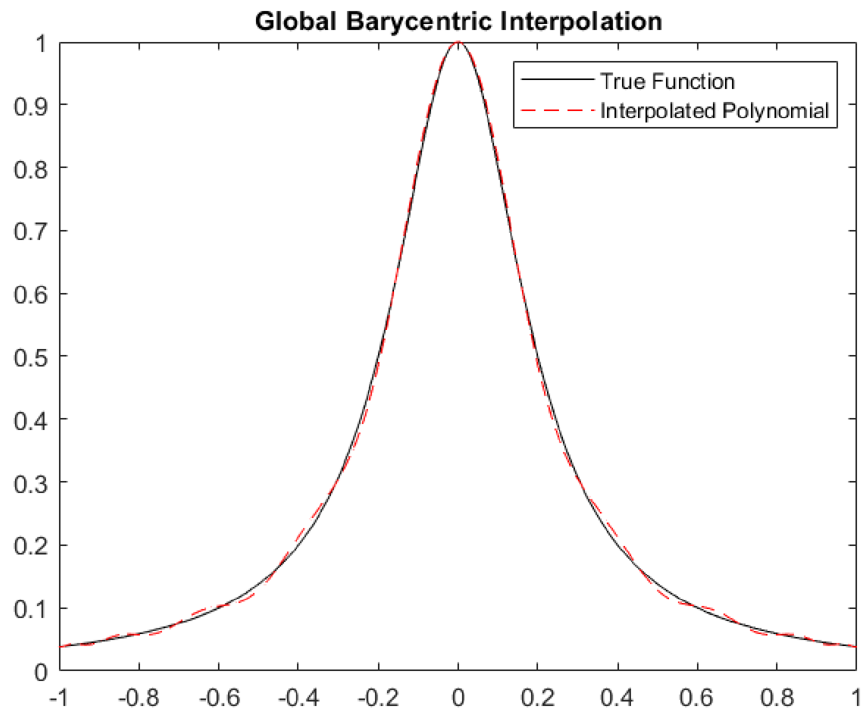
where xcheck is defined to evaluate the function and the polynomial for comparison. We get the following interpolation:



Note that we can reproduce the issues seen in the notes (e,g choosing an odd n, say n=7), we get:



which misses the extreme behavior in the midpoint of the interval. We can also visually observe convergence, by using a larger value of n. We set n=20 to get:



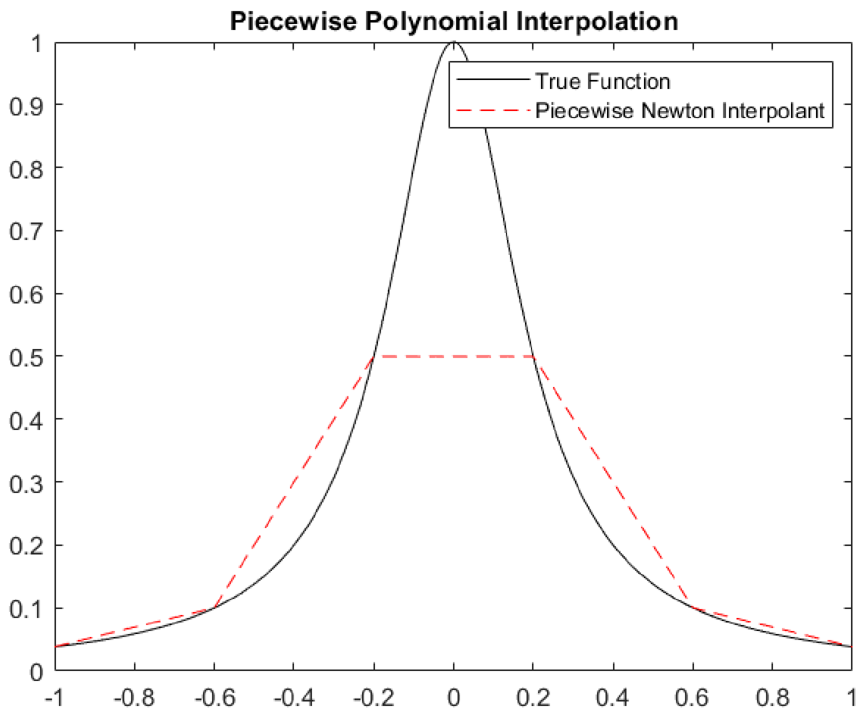
We will empirically test convergence more thoroughly in Task 1.

Piecewise Interpolating Polynomial

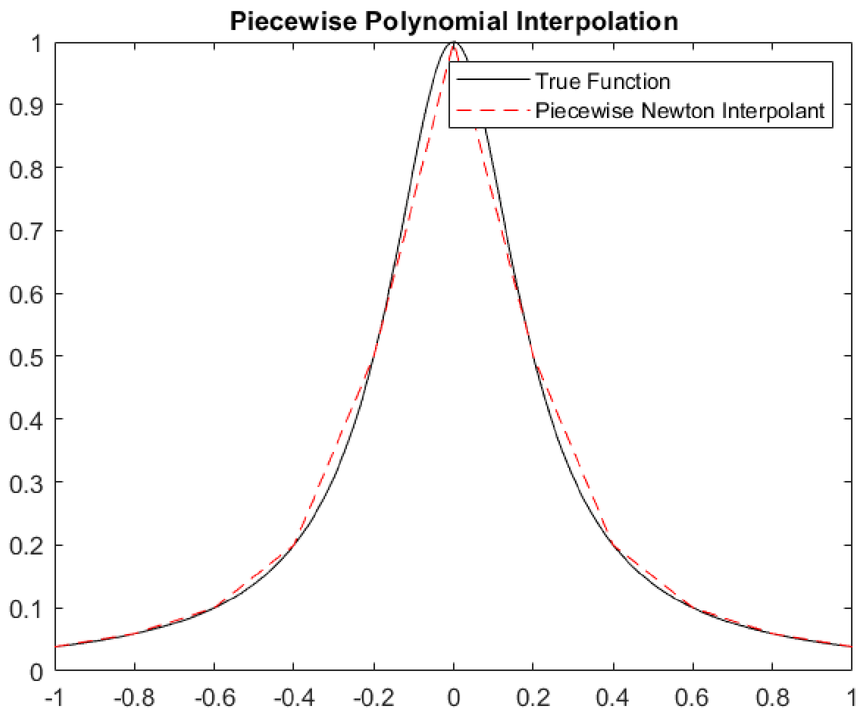
We set the following parameters for the piecewise interpolating polynomial:

```
% Parameters
num_intervals = 5;
a = -1;
b = 1;
global_mesh = linspace(a,b,num_intervals+1);
s = 1;                          % Degree of local Interpolant
node_type = 'chebyshev2';       % chebyshev2 or uniform
```

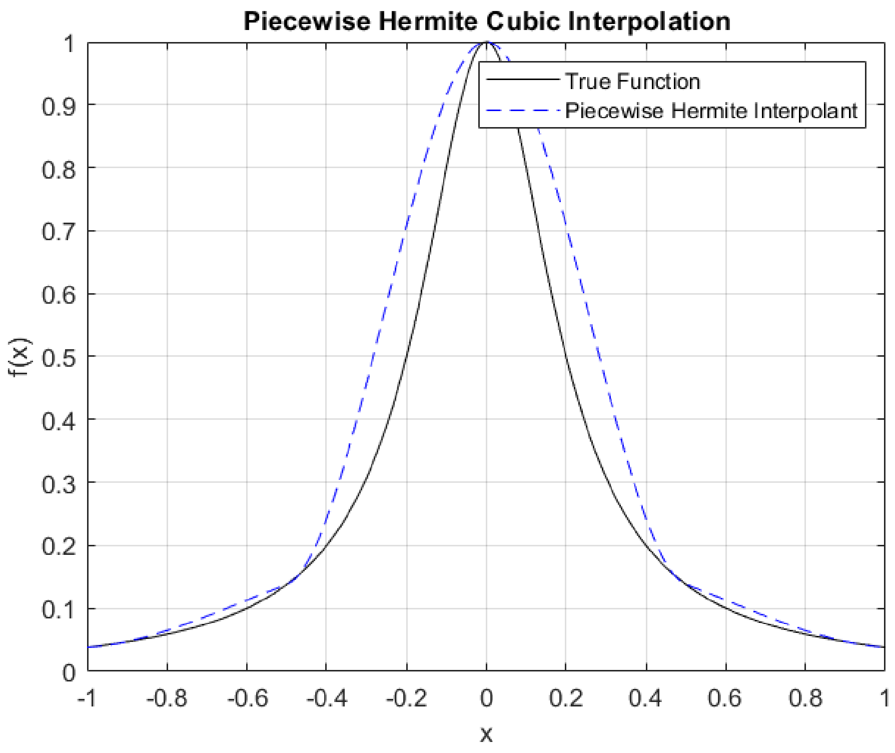
Note that the mesh can be chosen arbitrary here e.g. we support a non-uniform mesh. The convergence comes from the maximum length of the global mesh subinterval going down. We can choose s arbitrarily as well. We demonstrate our code using linear piecewise interpolation, and also demonstrate visual convergence:



We can then increase number of intervals:



We also include functionality for the piecewise cubic Hermite interpolation, where in each local interval, we interpolate $f(x_i), f(x_{i+1}), f'(x_i), f'(x_{i+1})$. The smoothness can be observed visually as well:



The piecewise code was implemented as follows:

1. We do reference interval implementation for the nodes into each local interval.
2. For each interval, we perform the Newton divided difference routine to obtain the local Newton interpolating form (i.e. parameterized by the local divided difference values and the local nodes). This is why we choose uniform or Chebyshev type 2 nodes; this gives us a closed mesh, and interpolation at the endpoints forces the piecewise polynomial to be C^1.
3. The local mesh and divided differences are stored in a matrix (1 row per interval). This allows us to evaluate a series of xcheck values, in $O(sn)$ space where s is the degree of the piecewise polynomial.
4. We first search the appropriate interval in $O(n)$ time to find the corresponding parameters. Then we evaluate the local polynomial using the modified Horner's method. Thus for evaluation, we get $O(n)$ time again.
5. The $O(n)$ space and time algorithm for the divided difference was used here, giving us the same complexity than for the piecewise, assuming s is a constant value (1,2, or 3).

The Hermite functionality was added by modifying the divided difference routine. We input a duplicated array function values and nodes, and whenever the divisor is 0, we replace the array with the appropriate value from the first derivatives array. This only affects the first iteration, and the other program proceeds as before.

We can also conclude the improvement in complexity for the piecewise. To the small degree on each piece, and the use of Newton, we were able to incorporate Hermite interpolation easily, while for Barycentric, the complexity is still $O(n^2)$.

Cubic Spline Interpolation (Spline 1)

We have the following parameters for the first Spline code:

```
% Replace with an arbitrary non-uniform mesh
a = -5;
b = 5;
num_intervals = 12;     % Keep even for Runge Function
global_mesh = linspace(a,b,num_intervals+1);
%global_mesh = [-5,-2.5,0,1,2,2.5,5]
bd_type = 'first';
% Define first derivative boundary conditions [s_0', s_n']
boundary_conditions = (1/26^2) * [25, -25];
```

We support the non-uniform mesh for the Spline 1 code, and I chose to use the s' parameterization. This is obtained from the Hermite interpolation using the cardinal basis within each subinterval:

$$p_i(x) = \psi_{L,i}(x)s_{i-1} + \psi_{R,i}(x)s_i + \Psi_{L,i}(x)s'_{i-1} + \Psi_{R,i}(x)s'_i$$

$$\psi_{L,i}(x) = \frac{(x-x_i)^2}{h_i^2}\left[1 + \frac{2}{h_i}(x - x_{i-1})\right]$$

$$\psi_{R,i}(x) = \frac{(x-x_{i-1})^2}{h_i^2}\left[1 - \frac{2}{h_i}(x - x_i)\right]$$

$$\Psi_{L,i}(x) = \frac{(x-x_i)^2}{h_i^2}(x - x_{i-1})$$

$$\Psi_{R,i}(x) = \frac{(x-x_{i-1})^2}{h_i^2}(x - x_i)$$

After we impose the relevant conditions to the second derivatives, we obtain the following system:

$$\begin{pmatrix} \lambda_1 & 2 & \mu_1 & 0 & \cdots & 0 & \\ 0 & \lambda_2 & 2 & \mu_2 & \cdots & \vdots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \\ \vdots & \cdots & 0 & \lambda_{n-2} & 2 & \mu_{n-2} & \\ 0 & \cdots & 0 & 0 & \lambda_{n-1} & 2 & \mu_{n-1} \end{pmatrix} \begin{pmatrix} s'_0 \\ s'_1 \\ \vdots \\ s'_{n-1} \\ s'_n \end{pmatrix} = \begin{pmatrix} g_1 \\ \vdots \\ g_{n-1} \end{pmatrix}$$

After we impose the first derivative Hermite conditions we get the following system (this is just specific values for the derivatives at the end-points). Imposing specific values on s'_0 and s'_n yields:

$$\begin{pmatrix} 2 & \mu_1 & 0 & \cdots & 0 \\ \lambda_2 & 2 & \mu_2 & \cdots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \cdots & \lambda_{n-2} & 2 & \mu_{n-2} \\ 0 & \cdots & 0 & \lambda_{n-1} & 2 \end{pmatrix} \begin{pmatrix} s'_1 \\ s'_2 \\ \vdots \\ s'_{n-2} \\ s'_{n-1} \end{pmatrix} = \begin{pmatrix} g_1 - \lambda_1 s'_0 \\ \vdots \\ g_{n-1} - \mu_{n-1}s'_n \end{pmatrix}$$

Note that the arrays λ, μ, g are defined in terms of the mesh points, the given f values (the divided differences) so they are known.

$$\lambda_i s'_{i-1} + 2s'_i + \mu_i s'_{i+1} = 3\left[\lambda_i f[x_{i-1}, x_i] + \mu_i f[x_i, x_{i+1}]\right] = g_i$$

$$\lambda_i = \frac{h_{i+1}}{h_i + h_{i+1}} < 1 \quad \text{and} \quad \mu_i = \frac{h_i}{h_i + h_{i+1}} < 1$$

The array of s' values are unknown and parameterize the piecewise spline characterization. Once this array is known, we are done, i.e. we need only solve the given system (note imposing the conditions only changes the first and last elements of the g RHS vector. To enforce $s''(x_0) = f''_0$ and $s''(x_n) = f''_n$ use the expression defined by $p''_1(x_0) = f''_0$ as the equation $i = 0$ and similarly for a derivative boundary condition at x_n. This yields $n+1$ equations in $n+1$ unknowns $s'_i,\ 0 \le i \le n$ as well. We can now derive this equations from the Hermite Cardinal Basis parameterization. We get the equation

$$p''_i(x) = \Psi''_{L,i}(x)f_{i-1} + \psi''_{R,i}(x)f_i + \Psi''_{L,i}(x)s'_{i-1} + \Psi''_{R,i}(x)s'_i$$

$$\Psi''_{L,i}(x) = \frac{4(x-x_i)}{h_i^2} + \frac{2(x-x_{i-1})}{h_i^2}$$

$$\Psi''_{R,i}(x) = \frac{2(x-x_i)}{h_i^2} + \frac{4(x-x_{i-1})}{h_i^2}$$

$$\psi''_{L,i}(x) = \frac{8(x-x_i)}{h_i^3} + \frac{4(x-x_{i-1})}{h_i^3} + \frac{2}{h_i^2}$$

$$\psi''_{R,i}(x) = -\frac{8(x-x_{i-1})}{h_i^3} - \frac{4(x-x_i)}{h_i^3} + \frac{2}{h_i^2}$$

with the values

$$\begin{aligned} \Psi''_{L,i}(x_i) &= \frac{2}{h_i} & \Psi''_{R,i}(x_i) &= \frac{4}{h_i} \\ \Psi''_{L,i+1}(x_i) &= -\frac{4}{h_{i+1}} & \Psi''_{R,i+1}(x_i) &= -\frac{2}{h_{i+1}} \\ \psi''_{L,i}(x_i) &= \frac{6}{h_i^2} & \psi''_{R,i}(x_i) &= -\frac{6}{h_i^2} \\ \psi''_{L,i+1}(x_i) &= -\frac{6}{h_{i+1}^2} & \psi''_{R,i+1}(x_i) &= \frac{6}{h_{i+1}^2} \end{aligned}$$

We then enforce the two second derivative Hermite (endpoint) conditions described above i.e.

$$p''_1(x_0) = f''_o$$
$$p''_n(x_n) = f''_n$$

to get the equations

$$\begin{aligned} p''_1(x_0) &= \left(\frac{-6}{h_1^2}\right) f_0 + \left(\frac{6}{h_1^2}\right) f_1 - \frac{4}{h_1} s'_0 - \frac{2}{h_1} s'_1 = f''_0 \\ p''_n(x_n) &= \left(\frac{6}{h_n^2}\right) f_{n-1} - \left(\frac{6}{h_n^2}\right) f_n + \frac{2}{h_n} s'_{n-1} + \frac{4}{h_n} s'_n = f''_n \end{aligned}$$

These two equations lead to the equations

$$\begin{aligned} 2s'_0 + s'_1 &= \frac{-h_1}{2} f''_o + 3f\left[x_0, x_1\right] = g_0 \\ s'_{n-1} + 2s'_n &= \frac{h_n}{2} f''_n + 3f\left[x_{n-1}, x_n\right] = g_1 \end{aligned}$$

The values of the g_o, g_1 are appended on the top and bottom of our general system, and the coefficients of the variables are appended on the top and bottom row. Note that this preserves the structure of the system, which we will discuss below.
Both the systems we get are of the form

$$\mathbf{Ax} = \mathbf{d}$$

where A is a strictly diagonally dominant tri diagonal (banded) matrix. This implies that we have a unique solution, and we can exploit the structure to obtain an efficient solving algorithm. We employ the *Thomas Algorithm*, which exploits forward elimination and backwards substitution for an $O(n)$ solving algorithm (similar to the lower and upper solves for lower-upper (LU) systems). Once we obtain $x = s'$, we use the local Hermite forms to evaluate just as we did for the piecewise polynomial (using the forms given above, parameterized by s and s'). These have straightforward, $O(n)$ complexity.

The spline code is implemented as follows:

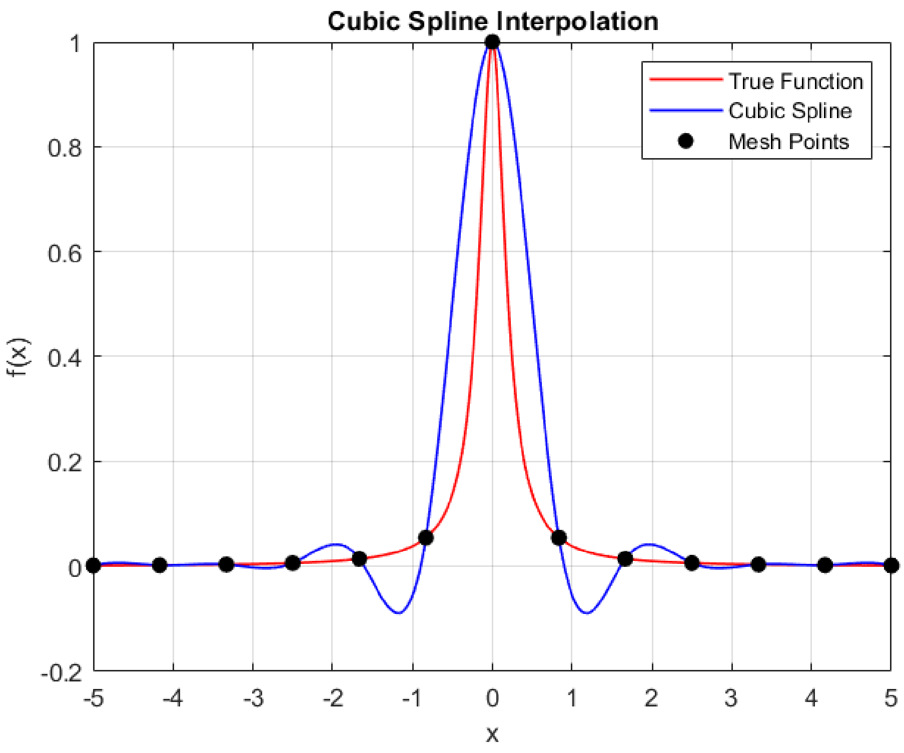
1. Once a mesh (may be non uniform) is defined, and we have function values, we can immediately parameterize the problem b computing the three diagonals of A (simple computations) and computing the right hand side g vectors in either case of Hermite Boundary conditions.

2. The Thomas Solver is used to compute the solution s' vector efficiently.

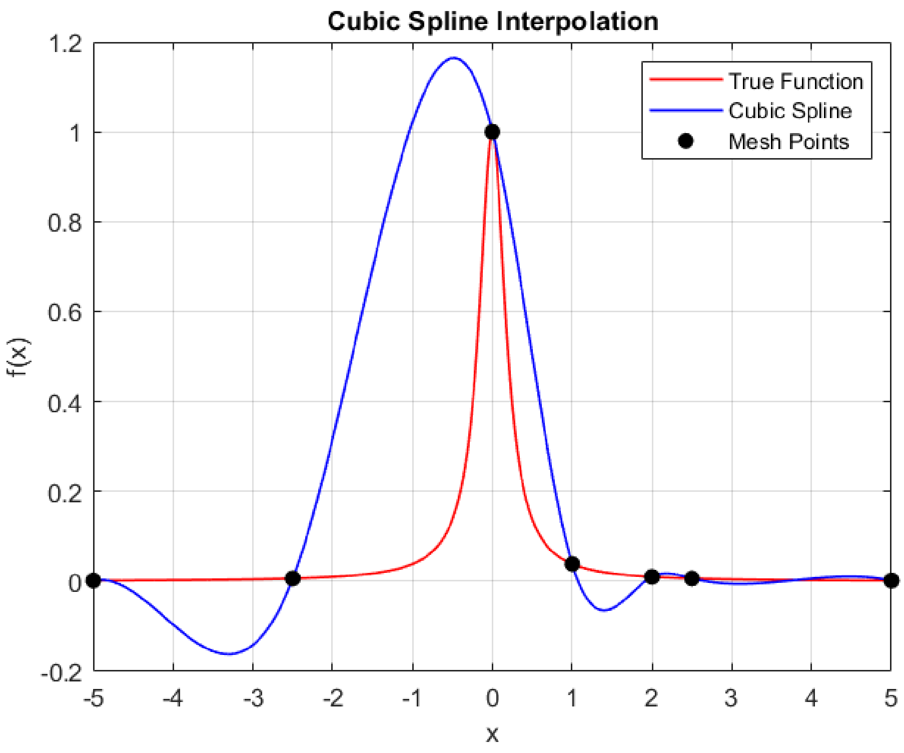
3. This gives us all local parameterizations of the spline.

4. Spline is evaluated on demand using the stored parameters, and an appropriate search for the relevant interval.

On the Runge function, we run the cubic spline (with the first derivative conditions) with 12 intervals.



We can also modify the code to include a non-uniform mesh.



Note the influence of the first derivative conditions.

B-Spline Interpolation (Spline 2)

For the B-splines, we assume a Uniform mesh (i.e. a constant h value), and implement the Bell splines basis parameterization. We also implement the two Hermite-type boundary conditions, each of which give us slightly different systems. The first is the first derivative conditions, $s'_0 = f'_0$ and $s'_n = f'_n$, which lead to the system in the next theorem.

The following theorem gives us the required parameterization for the B-splines.

Theorem 18.1 *Given, f'_0, f'_n, f_i and $x_0 < x_1 < \cdots < x_n$ for $0 \leq i \leq n$, the unique cubic spline, $s(x) \in \mathcal{B}$, such that*

$$s(x_i) = f_i \quad \text{for } 0 \leq i \leq n, \quad s'(x_0) = f'_0, \quad s'(x_n) = f'_n$$

is given by $s(x) = \sum_{i=-1}^{n+1} \alpha_i B_i(x)$ *where*

$$\begin{pmatrix} -\frac{3}{h} & 0 & \frac{3}{h} & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 4 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 4 & 1 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 4 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & \frac{-3}{h} & 0 & \frac{3}{h} \end{pmatrix} \begin{pmatrix} \alpha_{-1} \\ \alpha_0 \\ \vdots \\ \alpha_n \\ \alpha_{n+1} \end{pmatrix} = \begin{pmatrix} f'_0 \\ f_0 \\ \vdots \\ f_n \\ f'_n \end{pmatrix}$$

And we use the following definition of the B-splines for local evaluation. Assume uniform spacing $x_{i+1} - x_i = h$. The function $B_i(x)$ defined by

$$B_i(x) = \begin{cases} \frac{1}{h^3}(x - x_{i-2})^3, & \text{if } x_{i-2} \leq x \leq x_{i-1} \\ \frac{1}{h^3}\left(h^3 + 3h^2(x - x_{i-1}) + 3h(x - x_{i-1})^2 - 3(x - x_{i-1})^3\right), & \text{if } x_{i-1} \leq x \leq x_i \\ \frac{1}{h^3}\left(h^3 + 3h^2(x_{i+1} - x) + 3h(x_{i+1} - x)^2 - 3(x_{i+1} - x)^3\right), & \text{if } x_i \leq x \leq x_{i+1} \\ \frac{1}{h^3}(x_{i+2} - x)^3, & \text{if } x_{i+1} \leq x \leq x_{i+2} \\ 0, & \text{otherwise} \end{cases}$$

is a cubic B-spline (Prenter). x_i is the center reference point and the support is $[x_{i-2}, x_{i+2}]$ and includes 5 mesh points.

For the other Hermite conditions (second derivatives), we replace the appended f_0', f_n'' values on the known RHS f values vector with f_0'', f_n'' on the top and bottom, and the equations that we get come from the parameterization which gives us the equation

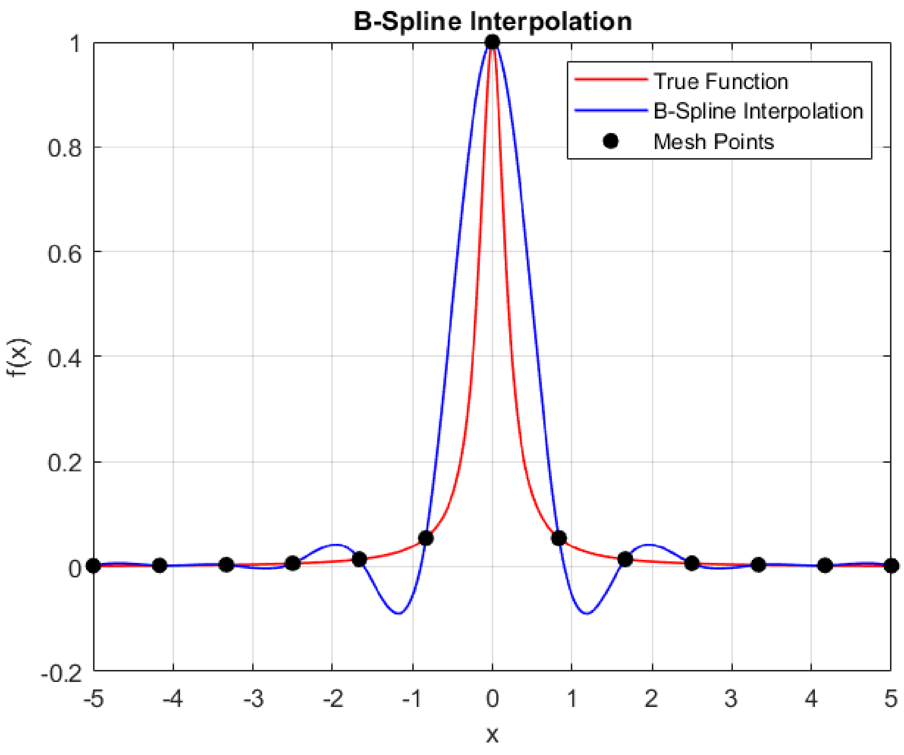
$$s''(x) = \sum_{i=-1}^{n+1} \alpha_i B_i''(x)$$

We then use the known values of the B_i'' at the end-points to get the equations with the coefficients for the required system. Only the first 3 basis vectors affect the value at the first point and the last 3 for the end point. The values appended now at the top left and the bottom right of the system become

$$\frac{6}{h^2}, \frac{-12}{h^2}, \frac{6}{h^2}$$

These systems are not tri diagonal, but some modifications to the earlier used Thomas Algorithm can be mode to still get $O(n)$ time. See code in the appendix for the details.

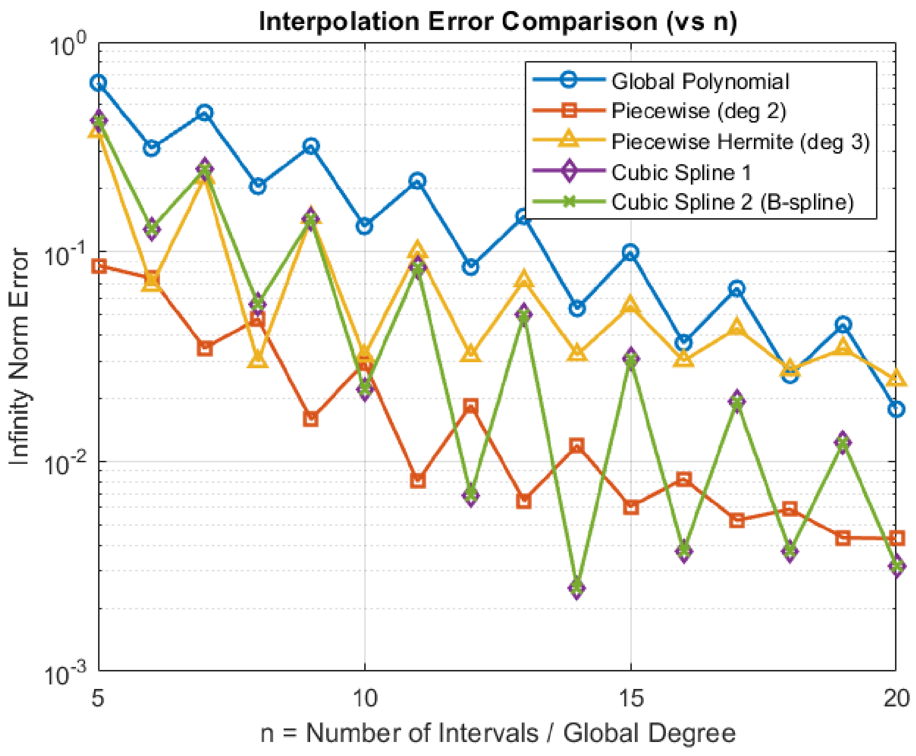
Note that the evaluation code was handled similarly to the piecewise, with an interval search proceeded by an evaluation. For a Cubic B-spline, also note that for a value between two mesh points (global), the only B-splines that contribute to the value are the 2 B-splines basis on either side of the point. If the point happens to match the global mesh, then we use the B-spline defined at that point and 1 on either side. This simplifies our computations. The reference spline code was also optimized for this usage. We also then find the appropriate coefficients from the array of alpha's that were solved before. We use the same parameters as the regular spline to be able to compare results, which ought to be the same.



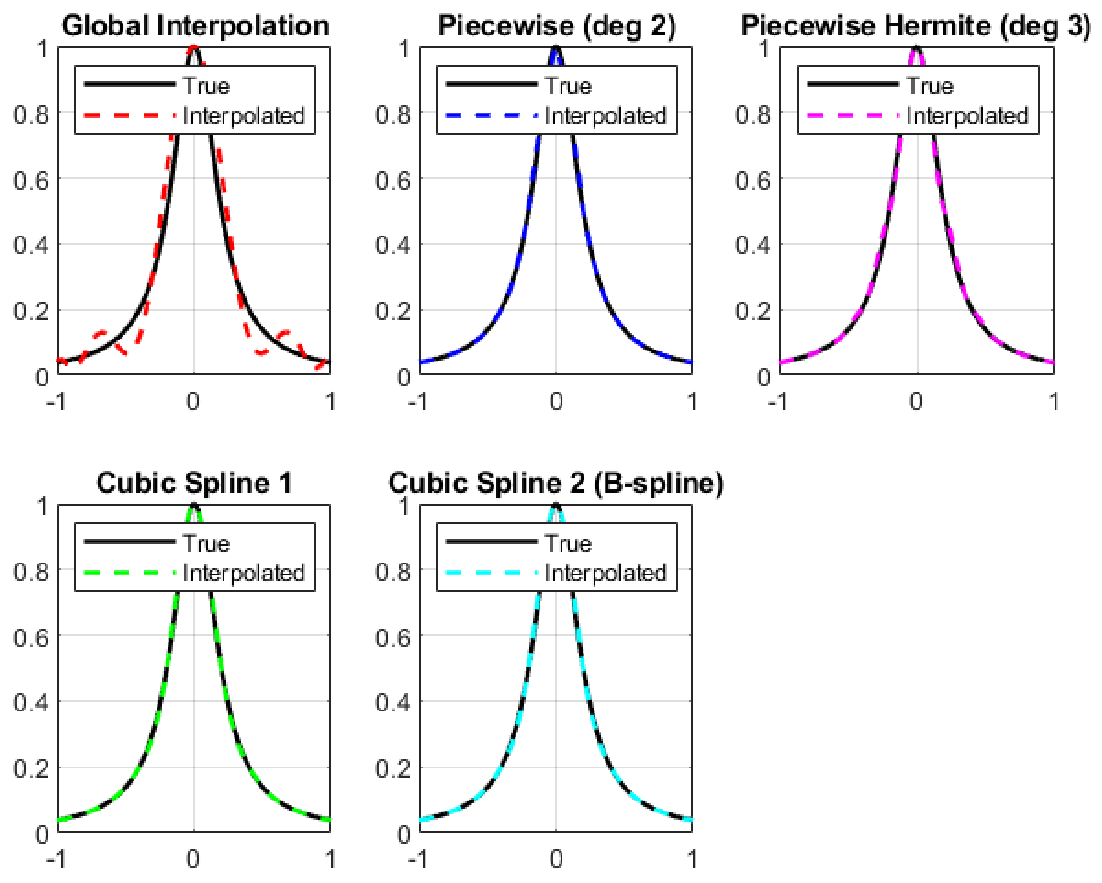
The other boundary condition was described above, and is implemented similarly. The code can be run for the other condition. Non-uniform functionality for the B-spline was not done.

3 Compiled Results

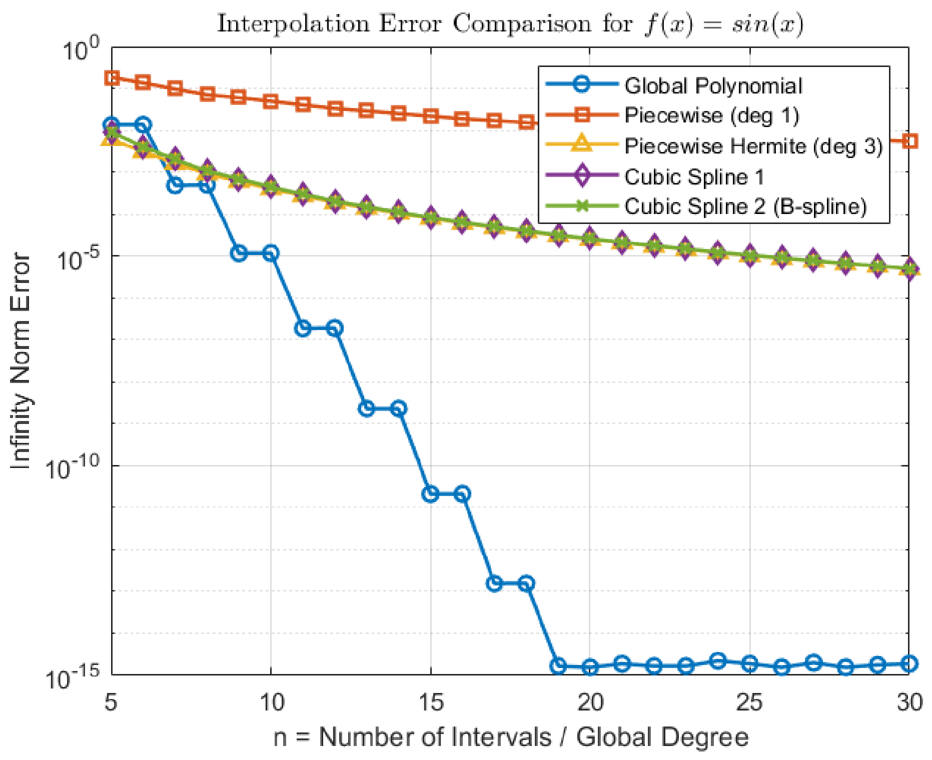
We first summarize the results for the Runge function:



The sawtooth behavior is due to the parity of the nodes, as discussed above. This empirically validates too the conceptual equality of the Spline 1 and 2 methods. We can hypothesis convergence from this graph and also conclude that the piecewise and spline methods work better than the global interpolation, for much less work (the same value of n actually means more work for the global). The local Hermite also performs well, but the splines eventually perform better. The graphs of the interpolations can also be visualized in the "test" routine if needed. It displays as follows:

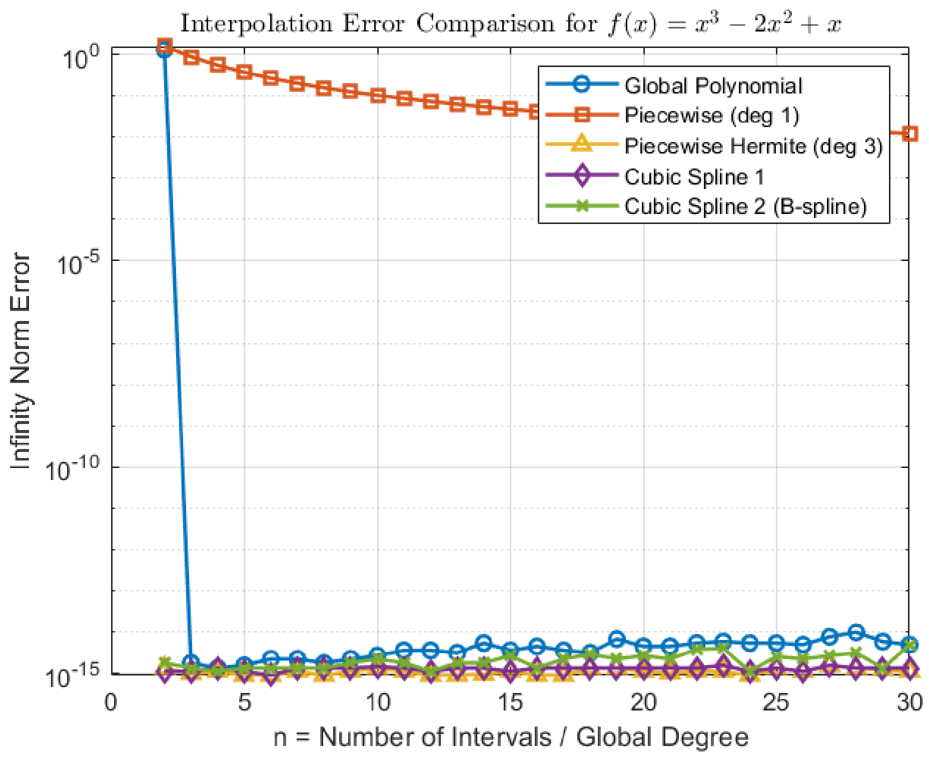


In Matlab, they can be enlarged for better visualization.
We now summarize results for $f(x) = sin(x)$ on $[-\pi, \pi]$.



Here the global outperforms the other methods, but for a lot more work. This is because of the local oscillations and the very high degree of the global polynomial (so a lot higher complexity). I also validated the second boundary conditions in this example by using $-sin(x)$ to give me values for the second derivatives on the end points. I also set the degree of the piecewise to 1.

Lets now test a cubic polynomial. This will be a good test for validation. I start from a smaller n (2) and go to 30, and set the piecewise to degree 1 for comparison. The interval is set to $[-1, 2]$. See *test* .mlx file for more details on all the parameters:



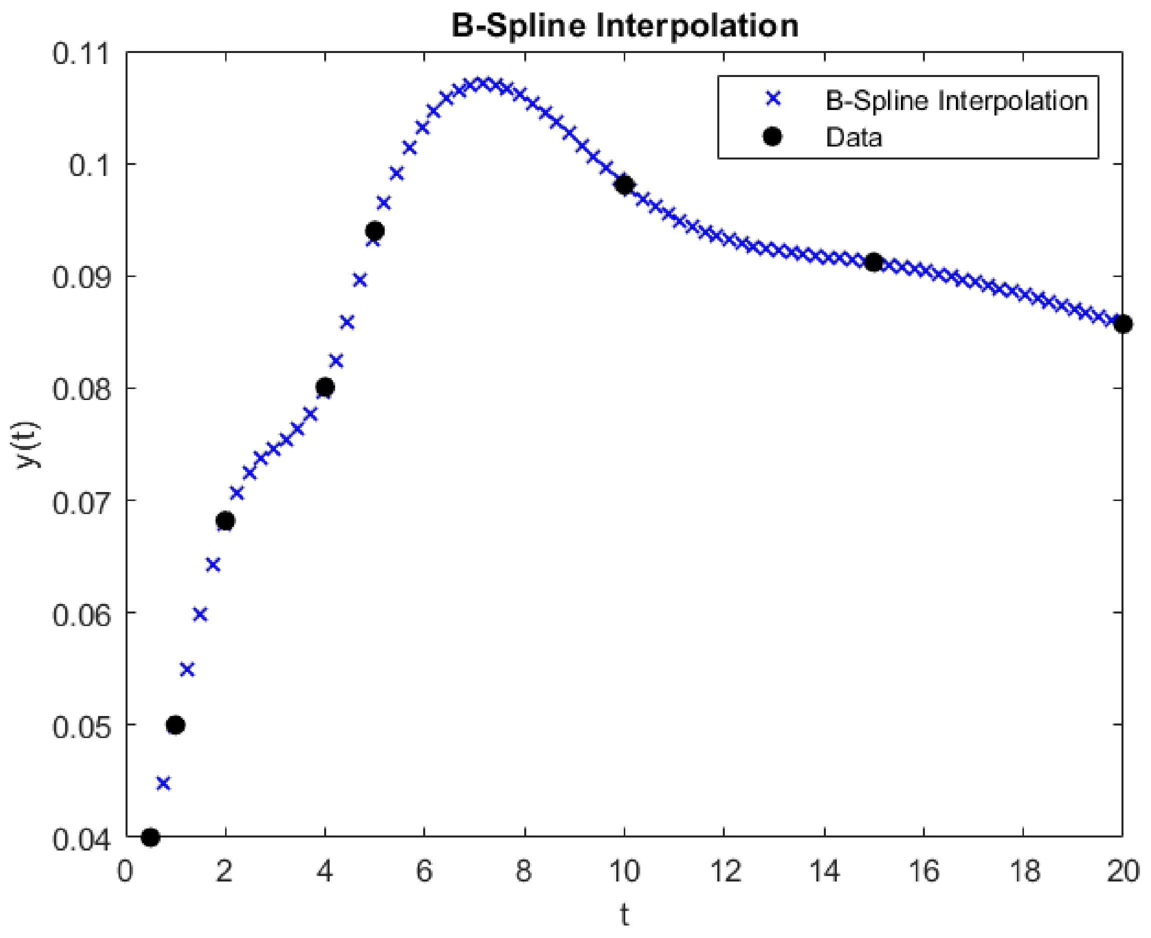
The routines perform as we expect them to (besides linear, due to the curvature of the cubic, so the convergence is very very slow). We can also see that the error is immediately to the order of double precision 10^{-16} which is what we except to see. We can also now try a higher degree polynomials, and see similar results, although for higher values of n, the global polynomial performs much better, although for a lot more work. In these experiments, we also tested the agreement between the two spline codes. Please see the testing driver files.

4 Task 2

We are given the data $y(t)$ in the form of discrete values of (t_i, y_i), for $0 \leq i \leq n$ rather than as a continuous function. Specifically, consider the following data (t_i, y_i):

t_i	0.5	1.0	2.0	4.0	5.0	10.0	15.0	20.0
$y(t_i)$	0.04	0.05	0.0682	0.0801	0.0940	0.0981	0.0912	0.0857

where $n = 7$. For the data given, we first compute the Cubic Spline interpolation, for 80 points with uniform spacing of $\Delta t = 0.5$ in the given data interval $[0.5, 20]$. Note the actual mesh is non-uniform. We can visualize this in the following plot:



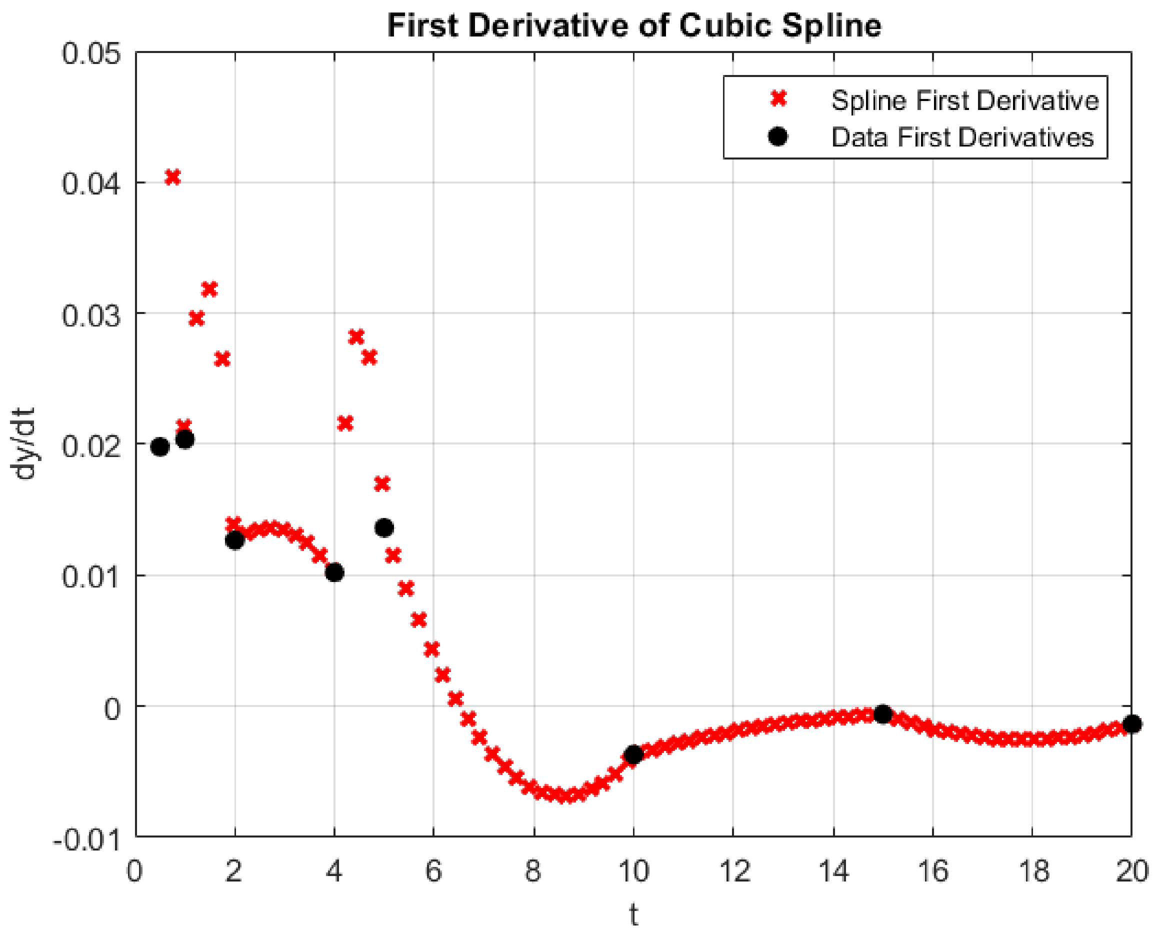
Note that the tabulated values can be obtained from the live .mlx script in the code section labeled as Task 2.

If we include a lot more points, the interpolation begins to become much smoother, but here we are concerned with the discrete case. On the same points, we are also interested in estimating the values of the functions $D(t)$ and $f(t)$, where

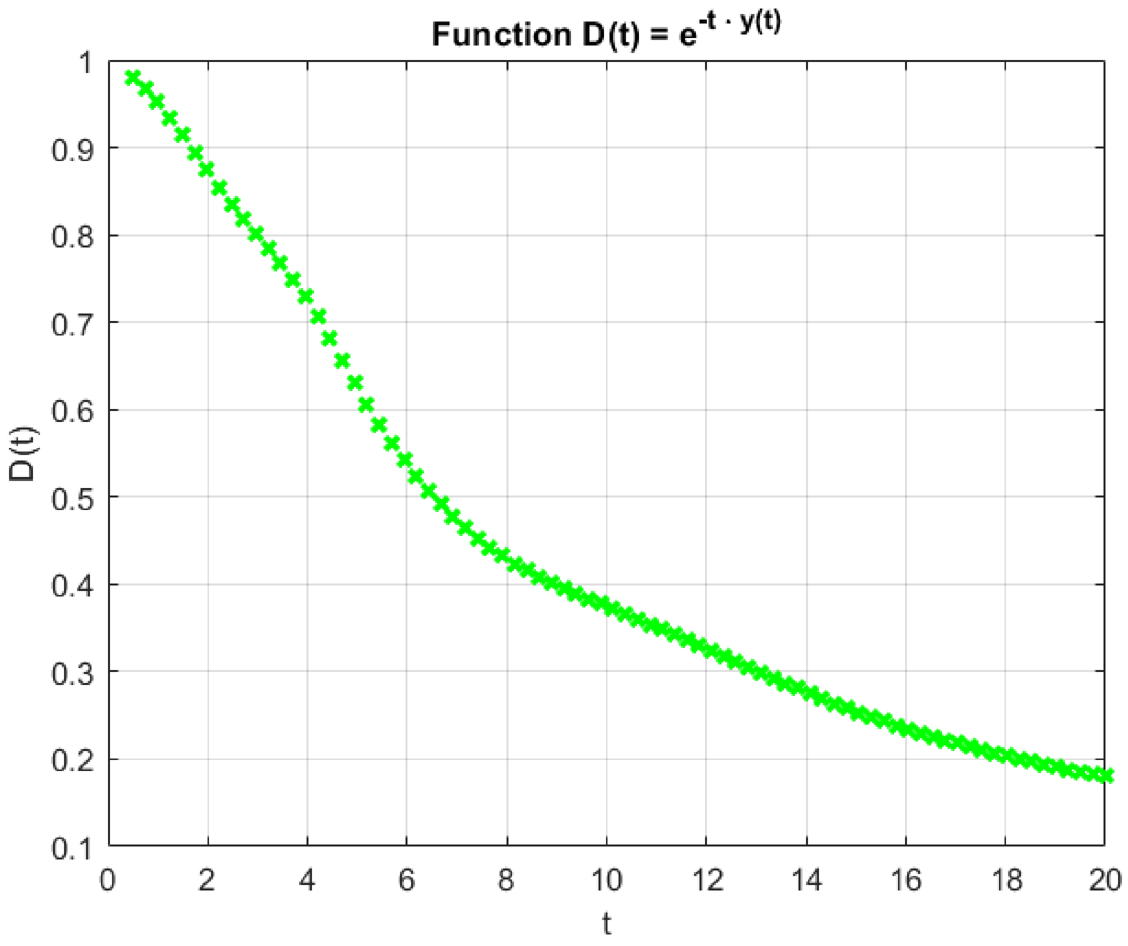
$$D(t) = e^{-ty(t)} = e^{-\int_0^t f(\tau)\,d\tau}$$

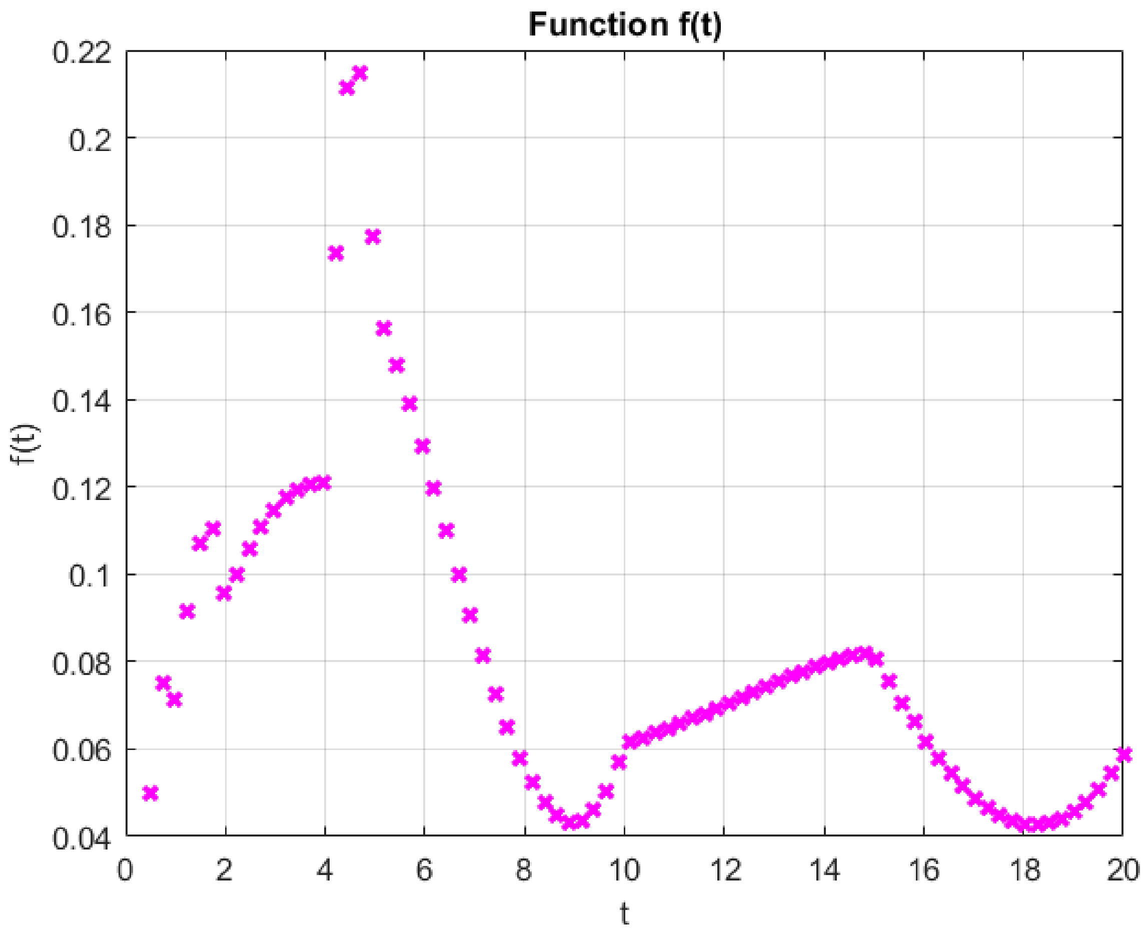
$$f(t) = y(t) + ty'(t)$$

To find the value for $y'(t)$, we note that we have the cubic spline parameterized in terms of the Hermite basis functions, so to get the first derivatives, we simply write another routine to evaluate the first parameterization of the first derivative., using the same xcheck array:



We can now evaluate the functions given to us. Note that there appear to be discontinuities in the spline derivative values, but this is only due to the coarse mesh used to find the values. The spline is indeed at least C^1. It should also be C^2, but this property seems to be violated at the closer mesh points (perhaps an issue with the numerical computation of s' near the nodes). I was unable to explain this issue, since we expect the first derivative to be continuously differentiable for the spline. Nevertheless, we get the following functions:


Function D(t) = e^{-t · y(t)}
D(t)
t
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10
12
14
16
18
20



Function f(t)
f(t)
t
0.22
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0
2
4
6
8
10
12
14
16
18
20

For this small discrete data, the issue is finding the appropriate parameter d for the piecewise polynomial interpolation. Since the number of points are small, the most obvious choice is $d = 1$ i.e. find a linear interpolating polynomial. It is not possible to do quadratic interpolation in each interval (since we require three points) but we can do $d = 3$ i.e. cubic interpolants, but this limits us to exactly two intervals. The results are as follows:

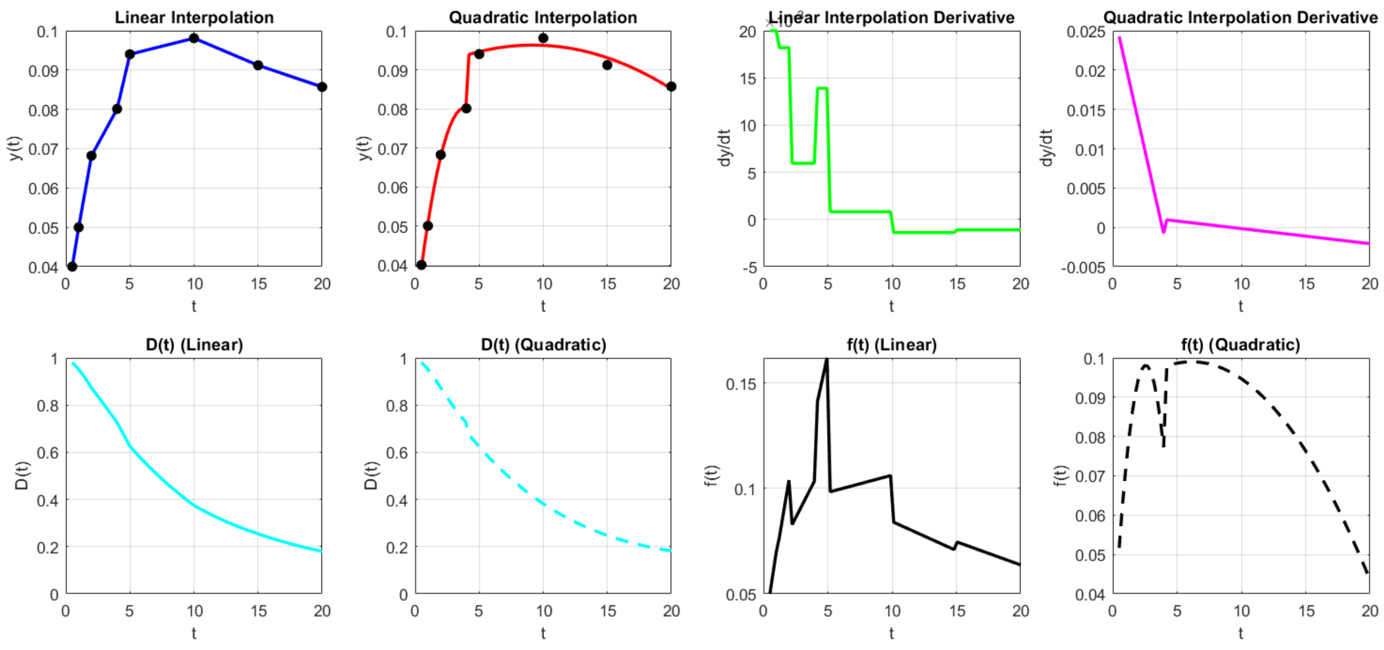


Figure 1: Comparison of Piecewise Interpolations

We can see that the function D is estimated well for both approaches, but the derivatives are not (this makes sense, since the linear functions have constant derivatives). As a result the function f, which uses the derivative is not estimated well. However, for quadratic, we also lose a lot of local behavior, since we are restricted to 2 sub-intervals. This is much more obvious in f, but is also an issue with estimating D. The spline achieves the estimation much better, at least within each subinterval. The nonuniform mesh also exacerbates the issues with both approaches, but less so with the spline.

5 Conclusions

We can conclude that the piecewise interpolation achieves much higher degrees of accuracy than global interpolation for the same amount of work (time complexity). The splines are even further improvement, leveraging piecewise interpolation, but also providing required smoothness (C^2 for cubic) at the nodes of the mesh. This was validated on various types of functions with varying parameters.

6 Appendix

Global Interpolation

Live Script:

```
% Function
k=25;
f = @(x) 1 ./ (1+k*x.^2);        % Runge Function

% Parameters
n = 10;
a = -1;
b = 1;
node_type = 'chebyshev2';
```

```
ncheck = 1000;
xcheck = linspace(a, b, ncheck)';

% This in addition returns kappa_numer and lebesgue_vals (remove ~)
[pvals, ftruevals, ~, ~] = global_p(f, xcheck, a, b, n, node_type);

% Visualization
plot(xcheck, ftruevals, 'k-', xcheck, pvals, 'r--');
legend('True Function', 'Interpolated Polynomial');
title('Global Barycentric Interpolation');
```

Barycentric Evaluation Routine:

```
function [pvals, ftruevals, lebesgue_vals, kappa_numer] = global_p(f, xcheck, a, b, n, node_type)
%GLOBAL_P Barycentric global interpolation over [a,b]
%
%   [pvals, ftruevals, lebesgue_vals, kappa_numer] = global_p(f, xcheck, a, b, n, node_type)
%
%   Inputs:
%       f         - Function handle to interpolate
%       xcheck    - Column vector of points to evaluate the interpolation at
%       a, b      - Interval [a, b]
%       n         - Degree of interpolating polynomial
%       node_type - 'chebyshev1' or 'chebyshev2'
%
%   Outputs:
%       pvals         - Interpolated polynomial values at xcheck
%       ftruevals     - True function values at xcheck
%       lebesgue_vals - Lebesgue function values at xcheck
%       kappa_numer   - Barycentric numerator sum at xcheck (Lebesgue kernel)

    % Generate interpolation nodes
    switch lower(node_type)
        case 'chebyshev1'
            xvals = chebyshev1(a, b, n);
        case 'chebyshev2'
            xvals = chebyshev2(a, b, n);
        otherwise
            error('Invalid node type. Use "chebyshev1" or "chebyshev2".');
    end

    % Sort and evaluate function at interpolation nodes
    xvals = sort(xvals);
    yvals = f(xvals);

    % True function values at evaluation points
    ftruevals = f(xcheck);

    % Barycentric weights
    gammainvs = bary1coeffs(xvals, n);

    % Preallocate output arrays
    kappa_numer = zeros(size(xcheck));
    lebesgue_vals = zeros(size(xcheck));

    % Evaluate barycentric interpolant
    [pvals, kappa_numer, lebesgue_vals] = bary1eval( ...
        xvals, yvals, gammainvs, n, xcheck, length(xcheck), ...
        kappa_numer, lebesgue_vals ...
    );
end
```

Codes provided by Dr. Gallivan:

```
function [gammainvs] = bary1coeffs(xvals,n)

%-------------------------------------------------------
%
% This code computes the inverses of the coefficients
% for the barycentric form 1 (modified Lagrange) of
% the interpolating polynomial on the mesh given
% in the n+1 by 1 column vector xvals.
% They are returned in the n+1 by 1 column vector gammainvs.
% n is the degree of the interpolating polynomial to be formed.
%
%-------------------------------------------------------


  np1=n+1;
gammainvs = zeros(np1,1);


%-------------------------------------------------------
% initialize to the linear coefficients
%-------------------------------------------------------

gammainvs(1) = (xvals(1) - xvals(2));  %   (x_0 - x_1)
gammainvs(2) = (xvals(2) - xvals(1));  %   (x_1 - x_0)
d=1;  % degree for whom the parameters are in gammainvs

%-------------------------------------------------------
% from degree 2 to n
% add effect of x_d, i.e., xvals(d+1)
% to gammainvs
%
%-------------------------------------------------------
for d=2:n
  p=1.0;
  for i= 0:d-1;
      t=(xvals(i+1) - xvals(d+1));
      gammainvs(i+1)=gammainvs(i+1)*t;  % m_i^d = t x m_i^{n-1}
      p=-p*t;
  end
  gammainvs(d+1)= p;
end

return

function [pvals,kappa_numer,lebesgue_vals] = bary1eval(xvals,yvals,gammainvs,n,
            xcheck,ncheck,kappa_numer,lebesgue_vals)

%-------------------------------------------------------
%
% This code evaluates the interpolating polynomial p(x)
% of degree d=n in barycentric form 1 (modified Lagrange)
% for the data pairs in (xvals(i),yvals(i)) specifed by
% the barycentric form 1 coefficients in gammainvs
% computed by bary1coeffs.m.
% xvals, yvals and gammainvs are all assumed n+1 by 1
% column vectors.
%
% The output is an ncheck by 1 column vector pvals
% containing pvals(i)=p(xcheck(i))
%
% The numerator of kappa(x,n,f) is returned
% in kappa_numer(1:ncheck)
%
% kappa(x,n,1) values are returned in lebesgue_vals(1:ncheck)
%
%-------------------------------------------------------


%-------------------------------------------------------
%
% EVALUATION ON A FINE GRID OF POINTS
%
% protection made if x = x_i for some i
% see Berrut and Trefethen for brief discussion
% exploits 0/0 = NAN but does not stop
%
%-------------------------------------------------------

omega = ones(ncheck,1);
exact = zeros(ncheck,1);
onevals = omega;
np1=n+1;

for i=1:np1
  xdiff=(xcheck - onevals*xvals(i));
  exact(xdiff==0) = i;
  omega = omega.*xdiff;
end

%
% The numerator of kappa(x,n,f) is computed and returned
% in kappa_numer(1:ncheck)
% and pvals the polynomial using the original data.



pvals = zeros(size(xcheck));
kappa_numer = zeros(size(xcheck));
lebesgue_vals= zeros(size(xcheck));

for i=1:np1
  xdiff=(xcheck - onevals*xvals(i));
  pvals = pvals + ((onevals*(yvals(i)/gammainvs(i))).*(omega./xdiff));
  kappa_numer = kappa_numer + (abs(onevals*(yvals(i)/gammainvs(i))).*abs(omega./xdiff));
  lebesgue_vals = lebesgue_vals  + (abs(onevals*(1.0/gammainvs(i))).*abs(omega./xdiff));
end

jfixes = find(exact);
pvals(jfixes)=yvals(exact(jfixes));
kappa_numer(jfixes)=abs(yvals(exact(jfixes)));
lebesgue_vals(jfixes)=abs(yvals(exact(jfixes)));

pvals = pvals';
kappa_numer = kappa_numer';
lebesgue_vals = lebesgue_vals';
return
```

Chebyshev Nodes:

```
function [x] = chebyshev1(a,b,n)

x = zeros(1,n+1);
for j = 0:n
```

```
    x(j+1) = cos( (2.0*j + 1.0)*pi / (2.0*n + 2) );
end

x = lineartransform(x,-1,1,a,b);
end

function [x] = chebyshev2(a,b,n)
x = zeros(1,n+1);
for j = 0:n
    x(j+1) = cos((j * pi) / n);
end

x = lineartransform(x,-1,1,a,b);
end
```

Linear transform:

```
function y = lineartransform(x,a,b,c,d)
%LINEARTRANSFORM Linearly transform into the interval c,d from the interval
% a,b
%   This routine takes in the array of x values in the interval [a,b] and
%   applies a linear transform to all the values into the interval [c,d]
if a == b
        error('Original interval [a, b] must have a != b');
end
y = (x - a)*(d - c)/ (b - a) + c;
end
```

Piecewise Interpolation

Live Script:

```
% Parameters
num_intervals = 4;
a = -1;
b = 1;
global_mesh = linspace(a,b,num_intervals+1);
s = 1;                          % Degree of local Interpolant
node_type = 'chebyshev2';       % chebyshev2 or uniform

f_prime = @(x) (-25*x) ./ ((1+k*x.^2).^2) ;
hermite_flag = true;

ncheck = 1000;
xcheck = linspace(a, b, ncheck)';

[divdiffs, local_meshes] = local_divdiffs(f, global_mesh, s, node_type, false, f_prime);
pvals = piecewise_p(xcheck, global_mesh, divdiffs, local_meshes, num_intervals, false);

ftrue = f(xcheck);

[divhermite, localhermite] = local_divdiffs(f, global_mesh, 3,'uniform', hermite_flag,f_prime);
pvals_hermite =  piecewise_p(xcheck, global_mesh, divhermite, localhermite, num_intervals,hermite_flag)


% Visualization
plot(xcheck, ftrue, 'k-', xcheck, pvals, 'r--');
legend('True Function', 'Piecewise Newton Interpolant');
title('Piecewise Polynomial Interpolation');

% Visualization for Hermite Interpolation
figure;
```

```
plot(xcheck, ftrue, 'k-', xcheck, pvals_hermite, 'b--');
legend('True Function', 'Piecewise Hermite Interpolant');
title('Piecewise Hermite Cubic Interpolation');
xlabel('x');
ylabel('f(x)');
grid on;
```

Piecewise Evaluation Routine:

```
function [pvals] = piecewise_p(xcheck, global_mesh, divdiffs, local_meshes,
                                num_intervals, hermite_flag)
%PIECEWISE_P Evaluates piecewise Newton or Hermite interpolant at specified xcheck points
%
%   [pvals, ftrue_vals] = piecewise_p(f, xcheck, global_mesh, divdiffs,
%                          local_meshes, num_intervals, hermite_flag)
%
%   Inputs:
%       f             - Function handle to compute true values
%       xcheck        - Column vector of evaluation points
%       global_mesh   - Vector of interval endpoints (length = num_intervals + 1)
%       divdiffs      - Matrix of divided differences (size depends on interpolation type)
%       local_meshes  - Matrix of local interpolation nodes (size depends on interpolation type)
%       num_intervals - Number of subintervals
%       hermite_flag  - Boolean: true = Hermite interpolation, false = Newton
%
%   Outputs:
%       pvals         - Column vector of interpolated values at each xcheck point
%       ftrue_vals    - True function values at each xcheck point


    pvals = zeros(size(xcheck));

    for k = 1:length(xcheck)
        x = xcheck(k);

        % Find which interval x belongs to
        if x == global_mesh(end)
            i = num_intervals;
        else
            i = find(global_mesh <= x, 1, 'last');
            if i >= length(global_mesh)
                i = num_intervals;
            end
        end

        % Select corresponding local mesh and divided differences
        local_x = local_meshes(i, :);
        local_coeffs = divdiffs(i, :);

        % If Hermite, duplicate local_x as [a, a, b, b]
        if hermite_flag
            local_x = reshape([local_x; local_x], 1, []);
        end

        % Evaluate using nddeval
        pvals(k) = nddeval(local_coeffs, local_x, x);
    end
end
```

Local Handling Routine:

```
function [divdiffs, local_meshes] = local_divdiffs(f, global_mesh, s, node_type, hermite_flag, fprime)
```

```
%LOCAL_DIVDIFFS Computes local Newton or Hermite divided differences for piecewise interpolation
%
%   [divdiffs, local_meshes] = local_divdiffs(f, global_mesh, s, node_type, hermite_flag, fprime)
%
%   Inputs:
%       f            - Function handle to interpolate
%       global_mesh  - Vector defining subinterval endpoints (length = num_intervals + 1)
%       s            - Degree of local Newton interpolants (ignored for Hermite, always cubic)
%       node_type    - 'chebyshev2' or 'uniform'
%       hermite_flag - Boolean: true = Hermite interpolation, false = standard Newton
%       fprime       - Function handle for f'(x), only used if hermite_flag = true
%
%   Outputs:
%       divdiffs     - Matrix, each row is local divided differences
%       local_meshes - Matrix, each row is the local interpolation nodes

    num_intervals = length(global_mesh) - 1;

    if hermite_flag
        % Hermite cubic: always 4 divided differences from 2 nodes
        divdiffs = zeros(num_intervals, 4);
        local_meshes = zeros(num_intervals, 2);  % Only 2 actual nodes (a and b)
    else
        % Newton: s+1 nodes and divided differences
        divdiffs = zeros(num_intervals, s+1);
        local_meshes = zeros(num_intervals, s+1);
    end

    for i = 1:num_intervals
        a = global_mesh(i);
        b = global_mesh(i+1);

        if hermite_flag
            % Use just the endpoints for Hermite interpolation
            local_nodes = [a, b];
            fvals_local = f(local_nodes);
            fprime_local = fprime(local_nodes);
            n = 4;  % Hermite cubic (with node duplication)

            div_row = nddcoeffs(local_nodes, fvals_local, fprime_local, n, true);
        else
            % Choose local interpolation nodes based on node_type
            switch lower(node_type)
                case 'chebyshev2'
                    local_nodes = chebyshev2(a, b, s);
                case 'uniform'
                    local_nodes = linspace(a, b, s+1);
                otherwise
                    error('Unsupported node type. Use "chebyshev2" or "uniform".');
            end

            local_nodes = sort(local_nodes); % Ensure ascending order
            fvals_local = f(local_nodes);
            n = s + 1;

            div_row = nddcoeffs(local_nodes, fvals_local, [], n, false);
        end

        divdiffs(i, :) = div_row;
        local_meshes(i, :) = local_nodes;
    end
```

```
end
```

Newton Interpolation Routines:

```
function [divdiffs] = nddcoeffs(nodes, fvals, f_prime_vals, n, hermite_flag)
%NDDCOEFFS Compute divided difference table
%   The function takes as input function y values 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.
%   If hermite_flag is true, the function uses Hermite interpolation by incorporating
%   first derivative values into the divided difference computation.

    if hermite_flag
        % **Hermite Interpolation: Duplicate nodes and function values**
        nodes = reshape([nodes; nodes], 1, []);
        fvals = reshape([fvals; fvals], 1, []);

        % Initialize divided differences array
        divdiffs = fvals;

        % **First iteration: Use f' values for repeated nodes**
        i = 1;
        divdiffs(i+1:end) = divdiffs(i+1:end) - divdiffs(i:end-1);
        D = nodes(i+1:end) - nodes(1:end-i);

        % Replace zero denominators with derivative values
        zero_idx = find(D == 0);
        if ~isempty(zero_idx)
            divdiffs(zero_idx + i) = f_prime_vals((zero_idx + 1) / 2);
            D(zero_idx) = 1;  % Prevent division by zero (not needed but ensures safety)
        end

        divdiffs(i+1:end) = divdiffs(i+1:end) ./ D;

        % **Subsequent iterations: Standard divided difference**
        for i = 2:n-1
            divdiffs(i+1:end) = divdiffs(i+1:end) - divdiffs(i:end-1);
            D = nodes(i+1:end) - nodes(1:end-i);
            divdiffs(i+1:end) = divdiffs(i+1:end) ./ D;
        end

    else
        % **Standard Newton Interpolation**
        divdiffs = fvals;
        for i = 1:n-1
            divdiffs(i+1:end) = divdiffs(i+1:end) - divdiffs(i:end-1);
            D = nodes(i+1:end) - nodes(1:end-i);
            divdiffs(i+1:end) = divdiffs(i+1:end) ./ D;
        end
    end
end

function [p] = nddeval(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
```

Cubic Spline Interpolation (Spline 1)

Live Script:

```
% Replace with an arbitrary non-uniform mesh
a = -5;
b = 5;
num_intervals = 12;     % Keep even for Runge Function
global_mesh = linspace(a,b,num_intervals+1);
%global_mesh = [-5,-2.5,0,1,2,2.5,5]
bd_type = 'first';
% Define first derivative boundary conditions [s_0', s_n']
boundary_conditions = (1/26^2) * [25, -25];

fvals = f(global_mesh);

% Compute the spline first derivative solution
s_prime = cubic_spline_first_derivatives(global_mesh, fvals, boundary_conditions, bd_type);

% Define xcheck points for evaluation
xcheck = linspace(-5, 5, 1000);

% Evaluate the cubic spline
spline_vals = cubic_spline(global_mesh, fvals, s_prime, xcheck);

true_vals = f(xcheck);

% Plot results
figure;
plot(xcheck, true_vals, 'r-', 'LineWidth', 1); hold on;
plot(xcheck, spline_vals, 'b-', 'LineWidth', 1);
plot(global_mesh, f(global_mesh), 'ko', 'MarkerFaceColor', 'k'); % Mesh points
legend('True Function', 'Cubic Spline', 'Mesh Points');
xlabel('x');
ylabel('f(x)');
title('Cubic Spline Interpolation');
grid on;
hold off;
```

Spline Evaluation:

```
function [spline_vals] = cubic_spline(global_mesh, fvals, s_prime, xcheck)
% CUBIC_SPLINE Evaluates the cubic spline at given xcheck points.
%
% Inputs:
%   global_mesh - (1 x (n+1)) array of sorted mesh points.
%   f - Function handle for the original function.
%   s_prime - (1 x (n+1)) array of computed first derivatives at mesh points.
%   xcheck - (1 x m) array of x-values to evaluate the spline.
%   num_intervals - Number of intervals used in the spline (n).
%
% Outputs:
%   spline_vals - (1 x m) array of spline values at xcheck.

    % Compute h array (interval widths)
    h = diff(global_mesh);

    % Initialize output arrays
    spline_vals = zeros(size(xcheck));

    % Loop through each xcheck value
    for j = 1:length(xcheck)
        x = xcheck(j);

        % Find the relevant interval i where x belongs to [x_{i-1}, x_i]
        i = find(global_mesh(1:end-1) <= x & x <= global_mesh(2:end), 1, 'last');

        if isempty(i)
            error('xcheck value out of global mesh range.');
        end

        % Interval endpoints
        x_left = global_mesh(i);
        x_right = global_mesh(i+1);
        h_i = h(i);

        % Function values and derivatives
        f_left = fvals(i);
        f_right = fvals(i+1);
        s_left = s_prime(i);
        s_right = s_prime(i+1);

        % Compute Hermite basis functions
        psi_L = ((x - x_right).^2 / h_i^2) .* (1 + 2 * (x - x_left) / h_i);
        psi_R = ((x - x_left).^2 / h_i^2) .* (1 - 2 * (x - x_right) / h_i);
        Psi_L = ((x - x_right).^2 / h_i^2) .* (x - x_left);
        Psi_R = ((x - x_left).^2 / h_i^2) .* (x - x_right);

        % Compute spline value using the Hermite basis
        spline_vals(j) = psi_L * f_left + psi_R * f_right + Psi_L * s_left + Psi_R * s_right;
    end
end
```

Parameter Routine (s'):

```
function s_prime = cubic_spline_first_derivatives(global_mesh, fvals, boundary_conditions, bc_type)
% CUBIC_SPLINE_FIRST_DERIVATIVES computes first derivative parameters for a cubic spline.
%
% Inputs:
%   global_mesh - (1 x (n+1)) array of mesh points.
%   f - Function handle for the given function.
%   boundary_conditions - (1x2) array specifying either:
%       * [s_0', s_n'] for 'first' derivative boundary conditions
%       * [f_0'', f_n''] for 'second' derivative boundary conditions
%   bc_type - String ('first' or 'second') indicating which boundary condition to use.
%
% Output:
%   s_prime - (1 x (n+1)) array of computed first derivatives for the cubic spline.

    % Number of points
    np1 = length(global_mesh); % np1 = n+1
    n = np1 - 1; % Number of intervals

    % Compute h array
    h = diff(global_mesh);

    % Compute lambda and mu
    lambda = h(2:end) ./ (h(1:end-1) + h(2:end));
    mu = h(1:end-1) ./ (h(1:end-1) + h(2:end));
    % Compute first-order divided differences
    div_diff = diff(fvals) ./ h; % f[x_i, x_{i+1}]

    % Compute g array
    g = 3 * (lambda .* div_diff(1:end-1) + mu .* div_diff(2:end));

    % System Setup Based on Boundary Condition Type
    if strcmp(bc_type, 'first')
        % **First Derivative Boundary Conditions**
        s_0_prime = boundary_conditions(1);
        s_n_prime = boundary_conditions(2);

        % Modify g for first derivative case
        g(1) = g(1) - lambda(1) * s_0_prime;
        g(end) = g(end) - mu(end) * s_n_prime;

        % Define system diagonals
        main_diag = 2 * ones(n-1, 1);
        sub_diag = lambda(2:end); % Lower diagonal (size n-2)
        super_diag = mu(1:end-1); % Upper diagonal (size n-2)

        % Solve the system using the Thomas algorithm
        s_inner = thomas_solver(sub_diag, main_diag, super_diag, g); % Size (n-1)

        % Append boundary conditions
        s_prime = [s_0_prime; s_inner; s_n_prime]; % Final array of size (n+1)

    elseif strcmp(bc_type, 'second')
        % **Second Derivative Boundary Conditions**
        f_0_double_prime = boundary_conditions(1);
        f_n_double_prime = boundary_conditions(2);

        % Ensure g_0 and g_n are **scalars**
        g_0 = (-h(1) / 2) * f_0_double_prime + 3 * div_diff(1);
        g_n = (h(end) / 2) * f_n_double_prime + 3 * div_diff(end);

        % Ensure g is a **column vector**
        g = [g_0; g(:); g_n]; % Fix concatenation issue

        % Define system diagonals
        main_diag = 2 * ones(n+1, 1);
        sub_diag = [lambda(:); 1]; % Append 1 to the end
        super_diag = [1; mu(:)];   % Append 1 to the beginning

        % Solve the system using the Thomas algorithm
        s_prime = thomas_solver(sub_diag, main_diag, super_diag, g); % Final array of size (n+1)

    else
        error('Invalid boundary condition type. Use "first" or "second".');
    end

end
```

Tri-diagonal Solver (Thomas Algorithm):

```
function x = thomas_solver(a, b, c, d)
%THOMAS_SOLVER Solves a tridiagonal system A * x = d using the Thomas algorithm
%
%   x = thomas_solver(a, b, c, d)
%
%   Inputs:
%       a - (n-1)x1 subdiagonal vector (a(2) to a(n))
%       b - (n)x1 main diagonal vector
%       c - (n-1)x1 superdiagonal vector (c(1) to c(n-1))
%       d - (n)x1 right-hand side vector
%
%   Output:
%       x - (n)x1 solution vector

    n = length(b);

    % Ensure column vectors
    a = a(:);
    b = b(:);
    c = c(:);
    d = d(:);

    % Forward elimination
    for i = 2:n
        w = a(i-1) / b(i-1);
        b(i) = b(i) - w * c(i-1);
        d(i) = d(i) - w * d(i-1);
    end

    % Back substitution
    x = zeros(n, 1);
    x(n) = d(n) / b(n);
    for i = n-1:-1:1
        x(i) = (d(i) - c(i) * x(i+1)) / b(i);
    end
end
```

B-Splines Interpolation (Spline 2)

Live Script:

```
% Parameters
num_intervals = 12;
a = -5;
b = 5;
global_mesh = linspace(a,b, num_intervals + 1); % n+1 points
bd_type = 'first';

h = 1.0 *((b - a) / num_intervals); % Compute h from interval length
xcheck = linspace(a, b, 1000);
% Compute B-spline coefficients (alpha) using B_spline_alpha
boundary_conditions = (1/26^2) * [25, -25];

fvals = f(global_mesh);
alpha = B_spline_alpha(global_mesh, fvals, boundary_conditions, h, bd_type);

spline_vals = B_spline(xcheck, alpha, global_mesh);

true_vals = f(xcheck);

% Plot results
figure;
plot(xcheck, true_vals, 'r-', 'LineWidth', 1); hold on;
plot(xcheck, spline_vals, 'b-', 'LineWidth', 1);
plot(global_mesh, f(global_mesh), 'ko', 'MarkerFaceColor', 'k'); % Mesh points
legend('True Function', 'B-Spline Interpolation', 'Mesh Points');
xlabel('x');
```

```
ylabel('f(x)');
title('B-Spline Interpolation');
grid on;
hold off;
```

B-spline Evaluation

```
function [spline_vals] = B_spline(xcheck, alpha, global_mesh)
% B_SPLINE Evaluates the cubic B-spline at given xcheck points.
%
% Inputs:
%   xcheck - (1 x m) array of x-values to evaluate the spline.
%   alpha - (1 x (n+3)) array of B-spline coefficients (includes alpha_{-1} and alpha_{n+1}).
%   global_mesh - (1 x (n+1)) array of sorted mesh points.
%   f - Function handle for the original function.
%
% Outputs:
%   spline_vals - (1 x m) array of spline values at xcheck.
%   true_vals - (1 x m) array of true function values f(xcheck).

    % Number of intervals
    np1 = length(global_mesh); % np1 = n+1
    n = np1 - 1; % Number of intervals
    h = global_mesh(2) - global_mesh(1); % Uniform spacing

    % Extend the global mesh to include x_{-1} and x_{n+1}
    extended_mesh = [global_mesh(1) - h, global_mesh, global_mesh(end) + h];

    % Initialize output arrays
    spline_vals = zeros(size(xcheck));
    % Loop through each xcheck value
    for j = 1:length(xcheck)
        x = xcheck(j);

        % Check if xcheck exactly matches a global mesh point
        idx_exact = find(global_mesh == x, 1);

        if ~isempty(idx_exact)
            % Special case: xcheck is exactly at a mesh point
            i = idx_exact + 1; % Adjust for extended mesh indexing

            % Compute three contributing B-splines
            B_left = bell_spline(x, extended_mesh(i-1), h); % Left B-spline
            B_center = bell_spline(x, extended_mesh(i), h); % Center B-spline
            B_right = bell_spline(x, extended_mesh(i+1), h); % Right B-spline

            % Compute the weighted sum of B-splines
            spline_vals(j) = alpha(i-1) * B_left + alpha(i) * B_center + alpha(i+1) * B_right;

        else
            % General case: xcheck lies in an interval between mesh points
            i = find(extended_mesh(1:end-1) <= x & x <= extended_mesh(2:end), 1, 'last');

            if isempty(i)
                error('xcheck value out of global mesh range.');
            end

            % Compute the four contributing B-splines
            B_left_2 = bell_spline(x, extended_mesh(i-1), h); % Second left
            B_left_1 = bell_spline(x, extended_mesh(i), h);   % First left
            B_right_1 = bell_spline(x, extended_mesh(i+1), h); % First right
            B_right_2 = bell_spline(x, extended_mesh(i+2), h); % Second right
            % Compute the weighted sum of B-splines using alpha coefficients
            spline_vals(j) = alpha(i-1) * B_left_2 + alpha(i) * B_left_1 + ...
                             alpha(i+1) * B_right_1 + alpha(i+2) * B_right_2;
        end
    end
end
```

Parameter Routine (α):

```
function alpha = B_spline_alpha(global_mesh, fvals, boundary_conditions, h, bc_type)
% B_SPLINE_ALPHA Computes the B-spline alpha parameters by solving a linear system.
%
% Inputs:
%   global_mesh - (1 x (n+1)) array of sorted mesh points.
%   f - Function handle for function evaluation.
%   boundary_conditions - (1x2) array specifying:
%       * [f_0', f_n'] for 'first' derivative boundary conditions
%       * [f_0'', f_n''] for 'second' derivative boundary conditions
%   h - Uniform interval length.
%   bc_type - String ('first' or 'second') indicating boundary condition type.
%
% Output:
%   alpha - (1 x (n+3)) array of computed B-spline parameters.

    % Number of intervals
    np1 = length(global_mesh); % np1 = n+1
    n = np1 - 1; % Number of intervals
    N = n + 3; % Total system size

    % Construct b vector (same for both cases)
    b = [boundary_conditions(1); fvals(:); boundary_conditions(2)]; % Ensure column vector

    % Construct A matrix
    A = zeros(N, N);

    % **Handle Boundary Condition Cases**
    if strcmp(bc_type, 'first')
        % First row: [-3/h, 0, 3/h, 0, ..., 0]
        A(1,1) = -3/h;
        A(1,3) = 3/h;

        % Last row: [0, ..., -3/h, 0, 3/h]
        A(N,N-2) = -3/h;
        A(N,N)   = 3/h;

    elseif strcmp(bc_type, 'second')
        % First row: [6/h^2, -12/h^2, 6/h^2, 0, ..., 0]
        A(1,1) = 6/h^2;
        A(1,2) = -12/h^2;
        A(1,3) = 6/h^2;

        % Last row: [0, ..., 6/h^2, -12/h^2, 6/h^2]
        A(N,N-2) = 6/h^2;
        A(N,N-1) = -12/h^2;
        A(N,N)   = 6/h^2;

    else
        error('Invalid boundary condition type. Use "first" or "second".');
    end
    % **Fill tridiagonal structure (rows 2 to N-1) - Unchanged**
    for i = 2:N-1
        A(i,i-1) = 1; % Subdiagonal
        A(i,i)   = 4; % Main diagonal
        A(i,i+1) = 1; % Superdiagonal
    end

    alpha = A \ b;

end
```

Single Bell Spline:

```
function B_val = bell_spline(xcheck, x_ref, h)
% BELL_SPLINE Evaluates the B-spline centered at x_ref at a given xcheck.
%
% Inputs:
%   xcheck - The point where we evaluate the B-spline.
%   x_ref - The reference center point x_i.
%   h - Uniform interval spacing.
%
% Output:
%   B_val - The computed B-spline value at xcheck.

    % Compute the five reference points
    x_m2 = x_ref - 2*h; % x_{i-2}
    x_m1 = x_ref - h;   % x_{i-1}
    x_0  = x_ref;       % x_i (center)
    x_1  = x_ref + h;   % x_{i+1}
    x_2  = x_ref + 2*h; % x_{i+2}

    % Initialize B-spline value
    B_val = 0;

    % Evaluate the B-spline based on the interval conditions
    if x_m2 <= xcheck && xcheck < x_m1
        B_val = (1/h^3) * (xcheck - x_m2)^3;

    elseif x_m1 <= xcheck && xcheck < x_0
        B_val = (1/h^3) * (h^3 + 3*h^2*(xcheck - x_m1) + 3*h*(xcheck - x_m1)^2 - 3*(xcheck - x_m1)^3);

    elseif x_0 <= xcheck && xcheck < x_1
        B_val = (1/h^3) * (h^3 + 3*h^2*(x_1 - xcheck) + 3*h*(x_1 - xcheck)^2 - 3*(x_1 - xcheck)^3);

    elseif x_1 <= xcheck && xcheck <= x_2
        B_val = (1/h^3) * (x_2 - xcheck)^3;
    end

end
```

Task 2

Live Script:

```
% Time values (t_i)
t = [0.5, 1.0, 2.0, 4.0, 5.0, 10.0, 15.0, 20.0];

% Corresponding y values (y(t_i))
y = [0.04, 0.05, 0.0682, 0.0801, 0.0940, 0.0981, 0.0912, 0.0857];

a = 0.5;
b = 20;
```

```
numintervals = 7;
bd_type = 'second';
boundary_conditions = [0, 0];    % Natural Boundary Conditions

s_prime = cubic_spline_first_derivatives(t, y, boundary_conditions, bd_type);

% Define xcheck points for evaluation
tstep = 80 % (del(t) = 0.5 from 0.5 to 40)
xcheck = linspace(a, b, 80);

% Evaluate the cubic spline
spline_vals = cubic_spline(t, y, s_prime, xcheck);

spline_deriv_vals = cubic_spline_derivative(t, y, s_prime, xcheck);

% ---- PLOT 1: Cubic Spline Interpolation ----
figure;
plot(xcheck, spline_vals, 'bx', 'LineWidth', 1); hold on;
plot(t, y, 'ko', 'MarkerFaceColor', 'k'); % Mesh points
legend('Cubic Spline Interpolation', 'Data Points');
xlabel('t');
ylabel('y(t)');
title('Cubic Spline Interpolation');
grid on;
hold off;

% ---- PLOT 2: First Derivative of the Spline ----
figure;
plot(xcheck, spline_deriv_vals, 'rx', 'LineWidth', 2); hold on;
plot(t, s_prime, 'ko', 'MarkerFaceColor', 'k'); % First derivative at known points
legend('Spline First Derivative', 'Data First Derivatives');
xlabel('t');
ylabel('dy/dt');
title('First Derivative of Cubic Spline');
grid on;
hold off;


% Compute functions D(t) and f(t)
D_t = exp(-xcheck .* spline_vals);
f_t = spline_vals + xcheck .* spline_deriv_vals;

% ---- PLOT 3: Function D(t) = exp(-t * y(t)) ----
figure;
plot(xcheck, D_t, 'gx', 'LineWidth', 2);
xlabel('t');
ylabel('D(t)');
title('Function D(t) = e^{-t \cdot y(t)}');
grid on;

% ---- PLOT 4: Function f(t) = y(t) + t * y\'(t) ----
figure;
plot(xcheck, f_t, 'mx', 'LineWidth', 2);
xlabel('t');
ylabel('f(t)');
title('Function f(t)');
grid on;

Manually doing g_d interpolation for this task to see details
y_linear = zeros(size(xcheck));
y_linear_deriv = zeros(size(xcheck));
```

```
% Compute linear interpolants
for i = 1:length(t)-1
    idx = (xcheck >= t(i)) & (xcheck <= t(i+1));
    slope = (y(i+1) - y(i)) / (t(i+1) - t(i));
    y_linear(idx) = y(i) + slope * (xcheck(idx) - t(i));
    y_linear_deriv(idx) = slope;  % Constant derivative for linear function
end

% first 4 points for first quadratic
T1 = t(1:4)';
Y1 = y(1:4)';
A1 = [T1.^2, T1, ones(size(T1))] \ Y1

% last 4 points for second quadratic
T2 = t(5:8)';
Y2 = y(5:8)';
A2 = [T2.^2, T2, ones(size(T2))] \ Y2;

y_quad = zeros(size(xcheck));
y_quad_deriv = zeros(size(xcheck));
for i = 1:length(xcheck)
    if xcheck(i) <= t(4)
        y_quad(i) = A1(1)*xcheck(i)^2 + A1(2)*xcheck(i) + A1(3);
        y_quad_deriv(i) = 2*A1(1)*xcheck(i) + A1(2);
    else
        y_quad(i) = A2(1)*xcheck(i)^2 + A2(2)*xcheck(i) + A2(3);
        y_quad_deriv(i) = 2*A2(1)*xcheck(i) + A2(2);
    end
end
D_linear = exp(-xcheck .* y_linear);
f_linear = y_linear + xcheck.*y_linear_deriv;
D_quad = exp(-xcheck .* y_quad);
f_quad = y_quad +xcheck .* y_quad_deriv;

figure;
% ---- Row 1: Interpolated Function ----
subplot(2,4,1);
plot(xcheck, y_linear, 'b-', 'LineWidth', 2); hold on;
plot(t, y, 'ko', 'MarkerFaceColor', 'k');
xlabel('t'); ylabel('y(t)');
title('Linear Interpolation');
grid on;

subplot(2,4,2);
plot(xcheck, y_quad, 'r-', 'LineWidth', 2); hold on;
plot(t, y, 'ko', 'MarkerFaceColor', 'k');
xlabel('t'); ylabel('y(t)');
title('Quadratic Interpolation');
grid on;

% ---- Row 2: First Derivative ----
subplot(2,4,3);
plot(xcheck, y_linear_deriv, 'g-', 'LineWidth', 2);
xlabel('t'); ylabel("dy/dt");
title('Linear Interpolation Derivative');
grid on;

subplot(2,4,4);
plot(xcheck, y_quad_deriv, 'm-', 'LineWidth', 2);
xlabel('t'); ylabel("dy/dt");
title('Quadratic Interpolation Derivative');
```

```
grid on;

% ---- Row 3: Function D(t) = e^{-t * y(t)} ----
subplot(2,4,5);
plot(xcheck, D_linear, 'c-', 'LineWidth', 2);
xlabel('t'); ylabel('D(t)');
title('D(t) (Linear)');
grid on;

subplot(2,4,6);
plot(xcheck, D_quad, 'c--', 'LineWidth', 2);
xlabel('t'); ylabel('D(t)');
title('D(t) (Quadratic)');
grid on;

% ---- Row 4: Function f(t) = y(t) + t * y'(t) ----
subplot(2,4,7);
plot(xcheck, f_linear, 'k-', 'LineWidth', 2);
xlabel('t'); ylabel('f(t)');
title('f(t) (Linear)');
grid on;

subplot(2,4,8);
plot(xcheck, f_quad, 'k--', 'LineWidth', 2);
xlabel('t'); ylabel('f(t)');
title('f(t) (Quadratic)');
grid on;

% Adjust spacing
sgtitle('Comparison of Linear vs Quadratic Piecewise Interpolation');
```

Splines Derivatives (Hermite Form):

```
function [spline_deriv_vals] = cubic_spline_derivative(global_mesh, fvals, s_prime, xcheck)
% CUBIC_SPLINE_DERIVATIVE Evaluates the derivative of the cubic spline at given xcheck points.
%
% Inputs:
%   global_mesh - (1 x (n+1)) array of sorted mesh points.
%   fvals - (1 x (n+1)) array of function values at mesh points.
%   s_prime - (1 x (n+1)) array of computed first derivatives at mesh points.
%   xcheck - (1 x m) array of x-values to evaluate the spline derivative.
%
% Outputs:
%   spline_deriv_vals - (1 x m) array of spline derivative values at xcheck.

    % Compute h array (interval widths)
    h = diff(global_mesh);

    % Initialize output arrays
    spline_deriv_vals = zeros(size(xcheck));

    % Loop through each xcheck value
    for j = 1:length(xcheck)
        x = xcheck(j);

        % Find the relevant interval i where x belongs to [x_{i-1}, x_i]
        i = find(global_mesh(1:end-1) <= x & x <= global_mesh(2:end), 1, 'last');

        if isempty(i)
            error('xcheck value out of global mesh range.');
        end
        % Interval endpoints
        x_left = global_mesh(i);
        x_right = global_mesh(i+1);
        h_i = h(i);

        % Function values and derivatives
        f_left = fvals(i);
        f_right = fvals(i+1);
        s_left = s_prime(i);
        s_right = s_prime(i+1);

        % Compute derivatives of Hermite basis functions
        psi_L_deriv = (2 * (x - x_right) / h_i^2) .* (1 + 2 * (x - x_left) / h_i)
        + ((x - x_right).^2 / h_i^2) * (2 / h_i);
        psi_R_deriv = (2 * (x - x_left) / h_i^2) .* (1 - 2 * (x - x_right) / h_i)
        - ((x - x_left).^2 / h_i^2) * (2 / h_i);
        Psi_L_deriv = ((x - x_right).^2 / h_i^2) + (2 * (x - x_right) / h_i^2) .* (x - x_left);
        Psi_R_deriv = ((x - x_left).^2 / h_i^2) - (2 * (x - x_left) / h_i^2) .* (x - x_right);

        % Compute the spline derivative value using the derivative basis functions
        spline_deriv_vals(j) = psi_L_deriv * f_left
                +psi_R_deriv * f_right + Psi_L_deriv * s_left + Psi_R_deriv * s_right;
    end
end
```