Program 2

David Miller
MAD5403: Foundations of Computational Math I

January 16, 2018

1 Executive Summary

Polynomial interpolation is a powerful method in which computed polynomials are used to approximate functions or sets of data. In this program we utilize Newton, Bernstein, piecewise interpolating, and interpolatory cubic spline polynomials. Accuracy, error, and complexity of each is discussed in detail.

2 Statement of Problem

We investigate four interpolatory methods:

1. Newton polynomial is an interpolation polynomial constructed given some mesh $\{(x_0, y_0), \ldots, (x_n, y_n)\}$ that uses divided difference values for its coefficients. A Newton polynomial is of the form:

$$\mathcal{N}(x) = \sum_{j=0}^{k} c_j \prod_{i=0}^{j-1} (x - x_i) \tag{1}$$

   where c_j is the divided difference value $[x_0, \ldots, c_j]$ and the x_i's are are mesh points.

2. Bernstein polynomial are interpolating polynomials constructed given some mesh $\{(x_0, y_0), \ldots, (x_n, y_n)\}$ defined on the unit interval $[0, 1]$. A Bernstein polynomial is of the form

$$B_n(x) = \sum_{k=0}^{n} f(\frac{k}{n}) \binom{n}{k} x^k (1-x)^{n-k}. \tag{2}$$

   However we can easily generalize this for some arbitrary interval $[a, b]$ by mapping the following

$$x \to \frac{x-a}{b-a}, \quad f(\frac{k}{n}) \to f(\frac{k}{n}(b-a) + a)$$

3. Piecewise polynomial is an interpolation polynomial where local polynomials are constructed on sub-intervals with the condition of continuity. More precisely we have a global interval $[a, b]$ with an interpolated global polynomial $p_{k,i}$, where k denotes the degree of each polynomial and i defines the subinterval defined on some sub-domain.

4. Cubic spline polynomials are piecewise polynomials with the added constraint on $p_{k,i}$ such that the values of their first $(k-1)$ derivatives also match at meshpoints x_i for $1 \leq i \leq n-1$. There are also two degrees of freedom in the construction of cubic splines where we must choose two boundary conditions so that we may solve for them.

These methods have their complexity and error analyzed and computed then compared to each other.

3 Description of Mathematics

3.1 Newton Method

The Newton interpolating polynomial is constructed via (1). Expanding out this out we get

$$\mathcal{N}(x) = [y_0] + [y_0, y_1](x - x_0) + \ldots + [y_0, \ldots, y_k](x - x_0)(x - x_1)\ldots(x - x_{k-1}).$$

If we investigate the form above, we can see Newton's method creates a Taylor polynomial based on the finite differences we compute, whereas Taylor's uses exact derivative values. Therefore we can view Newton's method as constructing a finite difference version of a Taylor series. Therefore as $k \to \infty$ we get that $\mathcal{N}(x)$ tends to the Taylor series. However Newton polynomials experience Runge phenomena around the boundaries since it is a single global interpolating polynomial. This is one thing piecewise fixes.

3.2 Bernstein Method

Bernstein polynomials provide numerically stable approximations to functions on the unit interval. In fact in can be shown that

$$\lim_{n\to\infty}(f) = f$$

on the interval [0,1] for some function f. The proof will be omitted for the sake of brevity. In fact it is uniformly convergent which implies

$$\lim_{n\to\infty} \sup\{|f(x) - B_n(f)(x)| : 0 \leq x \leq 1\} = 0.$$

This is of importance because this can be used to prove the Weierstrass approximation theorem which is a pivotal theorem to us in this class.

3.3 Piecewise Method

Given a global interval $[a, b]$ with mesh points $a = x_0 < x_1 < \ldots < x_n = b$, we can split this up into subintervals and apply Newton's method to those subintervals and enforcing the condition of continuity and boundary points. Therefore the problem can be described with the following rules

1. $[a, b] = \cup_s I_s$: union of disjoint subintervals (intersect only at subset of mesh points)

2. $g_k(x)$, on $I_s = [x_{i_s}, x_{i_{s+k}}]$ is in $\mathbb{P}_k$, where $g_k(x)$ is a piecewise polynomial
3. local interpolant $p_{k,i}(x_j) = f(x_j, \quad i_s \leq j \leq i_s + k)$
4. global interpolant $g_k(x_i) = f(x_i), \quad 0 \leq i \leq n$

From all this we form $p_{k,i}(x)$ where each is independent in construction and evaluation for all i.

3.4 Cubic Spline

3.4.1 Natural Boundary Condition

Given our form of cubic spline we must solve

$$\begin{pmatrix} \mu_1 & 2 & \lambda_1 & 0 & 0 & \dots \\ 0 & \mu_2 & 2 & \lambda_2 & 0 & \dots \\ & & \ddots & \ddots & \ddots & \\ 0 & \dots & & \mu_{n-1} & 2 & \lambda_{n-1} \end{pmatrix} \begin{pmatrix} s_0'' \\ s_1'' \\ \vdots \\ s_n'' \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{pmatrix}$$

to form $p_{k,i}$. However if we are given natural boundary conditions we have that

$$s_0'' = s_n'' = 0$$

and we can rewrite the linear system as

$$\begin{pmatrix} 2 & \lambda_1 & 0 & \dots & 0 \\ \mu_2 & 2 & \lambda_2 & 0 & \dots \\ \ddots & \ddots & \ddots & & \\ 0 & \dots & & \mu_{n-1} & 2 \end{pmatrix} \begin{pmatrix} s_1'' \\ s_2'' \\ \vdots \\ s_{n-1}'' \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{pmatrix}$$

This becomes trivial to solve using `solveTridig()` then computing $p_{k,i}$ using the results from solving the linear system.

3.4.2 Second Hermite Boundary Condition

Consider the second hermite boundary conditions $f''(a) = s_0''$ and $f''(b) = s_n''$ on our global interval $[a, b]$. Taking this into account we can rewrite the system
With substitution we get the two equations

$$d_1 - \mu_1 f''(a) = 2s_1'' + \lambda_1 s_2''$$
$$d_{n-1} - \lambda_{n-1} f''(b) = \mu_{n-1} s_{n-2}'' + 2s_{n-1}''$$

This allows us to have a square matrix which we can pass into the `solveTridig()` and we just have to do four simple overwrites to our data

$$\text{Update array element: } s''_0 \leftarrow f''(a)$$
$$\text{Update array element: } s''_n \leftarrow f''(b)$$
$$\text{Update array element: } d_1 \leftarrow d_1 - \mu_1 f''(a)$$
$$\text{Update array element: } d_{n-1} \leftarrow d_{n-1} - \lambda_{n-1} f''(b)$$

This gives us sufficient information to solve when given second Hermite boundary conditions.

4 Description of the Algorithm and Implementation

The Divided Difference algorithm has time complexity $\mathcal{O}(n^2)$, where n is the degree, while having space complexity $\mathcal{O}(n)$ to store the values of the divided differences. This is because we have a nested for loop that iterates up to $n-1$ and stores a value every time in the nested for loop.

Input: x-Mesh X, y-Mesh Y, results R, degree
Output: None, we store the results in the array passed in.

Algorithm 1 Divided Difference

```
for i = 0 : 1 : degree do
    R_i] ← Y_i
end for
level ← 1
for i = 0 : 1 : degree − 1 do
    for j = degree : −1 : 1 do
        R_j ← (R_j − R_{j−1})/(X_j − X_{j−level})
    end for
    level ← level + 1
end for
```

The Newton Method algorithm has time complexity $\mathcal{O}(n^2)$, where n is the degree, while having space complexity $\mathcal{O}(n)$ to store values of divided difference values when `dividedDifference()` is called and to store m results, where m is the amount of points we pass in to evaluate at. If we take into account all the values we are evaluating at then the time complexity of Newton method becomes $\mathcal{O}(mn^2)$.

Input: points P, numPoints, x-Mesh X, y-Mesh Y, degree, results R
Output: None, we store the results in the array passed in.

Algorithm 2 Newton Method

```
1: newtonMethod(xMesh, yMesh, C, degree)
2: for i = 0 : 1 : numPoints do
3:     value ← C_0
4:     for j = 1 : 1 : degree do
5:         product ← 1
6:         for k = 0 : 1 : j − 1 do
7:             product ← prodcut(P_i − X_k)
8:         end for
9:         value ← C_j ∗ product
10:    end for
11:    R_i ← value
12: end for
```

The Bernstein Method algorithm has time complexity $\mathcal{O}(n)$, where n is the degree, while having space complexity $\mathcal{O}(n)$ since we only have to pass yMesh values and store one temporary variable for the result value. However since the `nChooseK()` function takes linear time with respect to n we have that Bernstein has time complexity $\mathcal{O}(n^2)$. However we can sacrifice some space complexity and compute all values of `nChoosek()` once and store them. Then our Bernstein algorithm will have time complexity $\mathcal{O}(n)$ still with linear storage.

Input: point p, y-Mesh Y, degree, leftBound, rightBound
Output: Approximation of function we are interpolating

Algorithm 3 Bernstein Method

```
1: result ← 0
2: x ← (p − leftBound)/(rightBound − leftBound)
3: for i = 0 : 1 : degree do
4:     result ← Y_i ∗ (n choose i) ∗ f^i ∗ (1 − f)^(n−i)
5: end for
6: return result
```

The Piecewise Method algorithm has time complexity $\mathcal{O}(nlogn)+\mathcal{O}(n)+\mathcal{O}(d^2) = \mathcal{O}(nlogn)$, where n is number of mesh points and d is the degree of the interpolating polynomials. The first bound is from heap sort and the third comes from calling `newtonMethod()` with mesh size d. The third is not so obvious, but it comes from searching for the appropriate sub-interval and constructing it. When we search for the appropriate sub-interval we are iterating over the mesh points but with jump size d since we want to jump to the next sub-interval, and then building it requires $\mathcal{O}(d)$ time, and therefore $\mathcal{O}(\frac{n}{d} * d) = \mathcal{O}(n)$. However since we evaluate at m points, the complexity for this becomes $\mathcal{O}(mnlogn)$. The space complexity of Piecewise is $\mathcal{O}(n)$ since we must store results, mesh points and their values, and points to evaluate at.

The Piecewise Method should have the mesh points sorted so that searching for the ap-

propriate sub interval can be done. If the mesh points were not sorted then we would not be able to effectively calculate sub-intervals. To sort the mesh points, and their respective values, I implemented heap sort which is well known to be $\mathcal{O}(nlogn)$ in time and $\mathcal{O}(1)$ in memory.

Input: points P, numPoints, x-Mesh X, y-Mesh Y, meshSize, degree, results R
Output: None, we store the results in the array passed in

Algorithm 4 Piecewise Method

```
1: subintervals ← (meshSize − 1)/degree
2: heapsort(X, Y, meshSize)
3: for i = 0 : 1 : numPoints − 1 do
4:     for j = 0 : degree : meshSize − 1 do
5:         if P_i ≥ X_j && P_i ≤ X_{j+degre} then
6:             for k = 0 : 1 : degree do
7:                 subX_k ← X_{j+k}
8:                 subY_k ← Y_{j+k}
9:             end for
10:            break
11:        end if
12:    end for
13:    newtonMethod(P_i, 1, subX, subY, degree, value)
14:    R_i ← value
15: end for
```

The Solve Tridiagonal algorithm, sometimes called Thomson's algorithm, has time complexity $\mathcal{O}(n)$, where n is one less than the mesh size, and storage complexity $\mathcal{O}(n)$ since we have to store n results.

Input: vector a, vector b, vector c, vector d, results R, degree
Output: Output: None, we store the results in the array R

Algorithm 5 Solve Tridiagonal

```
1: n ← meshSize − 1
2: for i = 2 : 1 : n − 2 do
3:     c_i ← c_i/(b_i − a_i ∗ c_{i−1})
4: end for
5: for i = 2 : 1 : n − 1 do
6:     d_i ← (d_i − a_i ∗ d_{i−1})/(b_i − a_i ∗ c_{i−1})
7: end for
8: Rn − 1 ← d_{n−1}
9: for i = n − 2 : −1 : 1 do
10:    R_i ← d_i − c_i ∗ R_{i+1}
11: end for
```

Just like the Piecewise Method, we are bounded above by the time complexity of our sorting algorithm. The time complexity of Cubic spline is $\mathcal{O}(nlogn)$ and taking into account evaluating at m points the time complexity becomes $\mathcal{O}(mnlogn)$. We have $\mu, \lambda, d, M, \gamma, \tilde{\gamma}$, and results array that store values in $\mathcal{O}(n)$ and therefore Cubic spline has $\mathcal{O}(n)$ storage complexity.

Input: points P, numPoints, x-Mesh X, y-Mesh Y, meshSize, results R
Output: **Output:** None, we store the results in the array passed in

Algorithm 6 Cubic Spline

$n \leftarrow meshSize - 1$
heapsort(X, Y, $meshSize$)
for $i = 1 : 1 : n - 1$ **do**
    $\mu_i \leftarrow (X_i - X_{i-1})/(X_{i+1} - X_{i-1})$
    $\lambda_i = (X_{i+1} - X_i)/(X_{i+1} - X_{i-1})$
    $d_i \leftarrow 6 * (((Y_{i+1} - Y_i)/(X_{i+1} - X_i)) - ((Y_i - Y_{i-1})/(X_i - X_{i-1})))/(X_{i+1} - X_{i-1}))$
end for
if $secondHermite$ **then**
    $M_0 \leftarrow leftSecondDerivative$
    $M_n \leftarrow rightSecondDerivative$
    $d_1 \leftarrow d_1 - \mu_1 * leftSecondDerivative$
    $d_{n-1} = d_{n-1} - \lambda_{n-1} * rightSecondDerivate$
end if
solveTridig($\mu, 2, \lambda, d, M, meshSize$)
for $i = 1 : 1 : n$ **do**
    $\gamma_{i-1} \leftarrow (Y_i - Y_{i-1})/(X_i - X_{i-1}) - (M_i - M_{i-1}) * ((X_i - X_{i-1})/6)$
    $\tilde{\gamma}_{i-1} \leftarrow Y_{i-1} - M_{i-1} * (X_i - X_{i-1})^2/2$
end for
for $i = 0 : 1 : numPoints - 1$ **do**
    for $j = 0 : 1 : meshSize - 2$ **do**
        if $P_i \geq X_j$ && $P_i \leq X_{j+1}$ **then**
            $R_i \leftarrow M_j * (X_{j+1} - P_i)^3/(6 * (X_{j+1} - X_j)) + M_{j+1} * (P_i - X_j)^3/(6 * (X_{j+1} - X_j)) + \gamma_j * (P_i - X_j) + \tilde{\gamma}_j$
        end if
        break
    end for
end for

5 Description of the Experiment Design and Results

For this program,all the user needs to specify is the left boundary, right boundary, function to interpolate/approximate,and what boundary conditions to impose on the cubic spline algorithm. To impose mesh size or polynomial degree restrictions, we hard code it in the main function since it is faster and easier than asking the user all of these things via command line. Further details about the code will be discussed during the live code demonstration.

If we look at the figures, we see many things we expect to see:

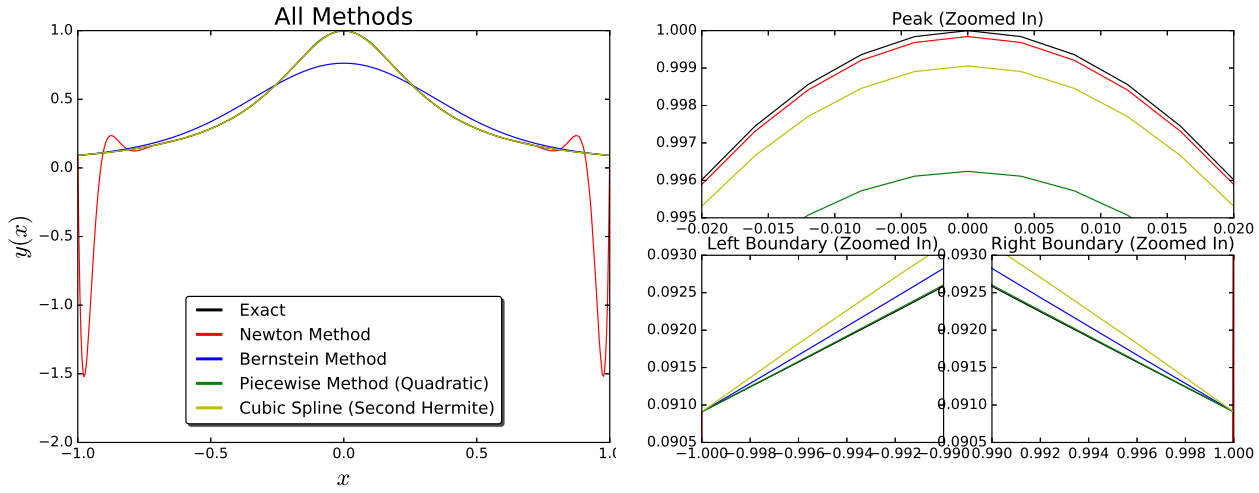
- Newton's Method is exhibiting Runge's Phenomena near the boundaries whereas using the Newton Method to interpolate subintervals in Piecewise does not exhibit this.
- Bernstein polynomials converge to the exact solution as we increase the degree of the polynomial. This is confirmed in figure 15.
- Figure 14(b) depicts the conditioning of the problem. Essentially we have a smaller conditioning number when $x < 0$ so our error will not be as high when $x > 0$.
- All interpolatory methods (everything except Bernstein's Method) that are fed order meshed points have zero error at the mesh points. Passing in a non order set will create errors at the mesh points. However Newton's method will start to incur errors near the boundaries due to Runge's Phenomena and will incur even greater error in its derivatives.
- The convergence rate of Bernstein is $\sqrt{n}$, where n is the amount of mesh points.

6 Conclusions

The interpolatory and approximation methods provide accurate results. Newton's method experiences Runge's Phenomena, Bernstein's polynomial converges to what we want as we increases the degree, Piecewise uses Newton's on subintervals which get rid of Runge's phenomena, and Cubic splines provide the best approximations that also give us differentiability. The interpolatory methods incur error on the mesh points if the mesh is not sorted when passed. Due to limited amount of time, a rigorous analysis is omitted and numerical evidence is instead presented in Section 7.

7 Figures

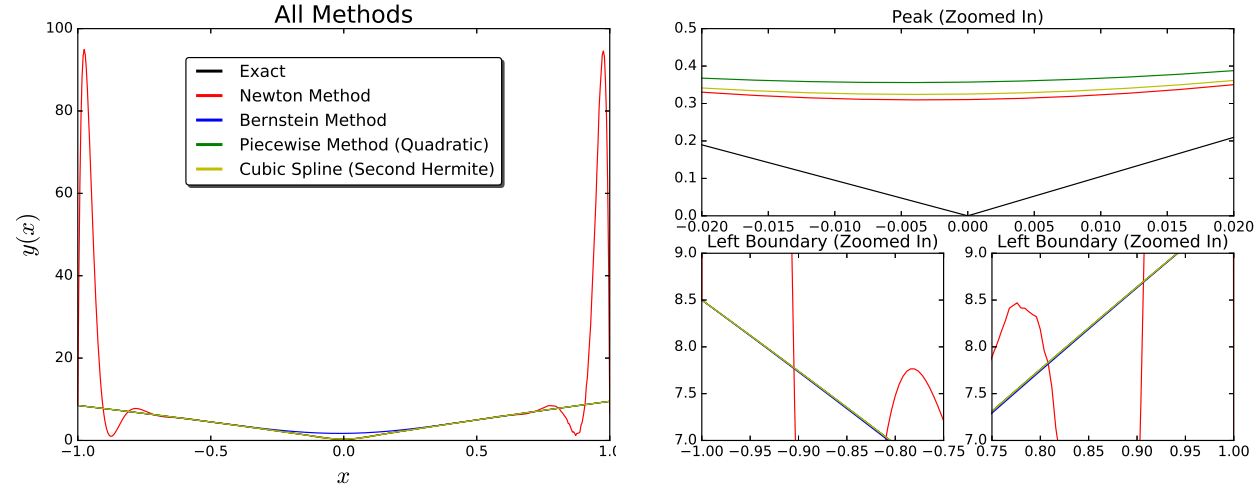
7.1 Hermite Boundary Conditions



(a) Global view of interpolating methods.

(b) Local view of interpolating methods.

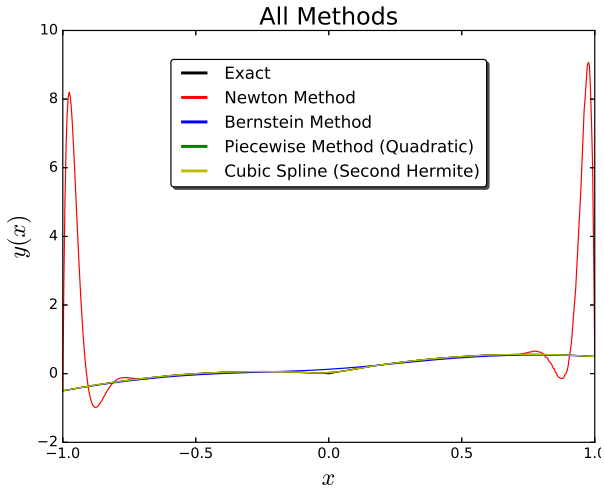
Figure 1: Interpolation for $f(x) = 1/(1 + 10x^2)$ with 22 uniform mesh points on [-1,1].



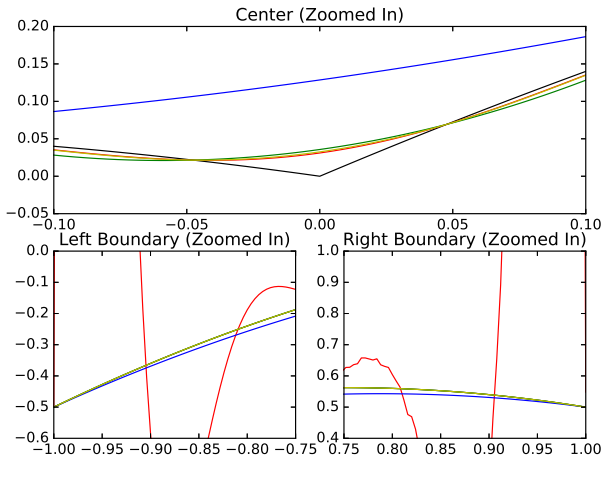
(a) Global view of interpolating methods.

(b) Local view of interpolating methods.

Figure 2: Interpolation for $f(x) = |10x| + x/2 - x^2$ with 22 uniform mesh points on [-1,1].

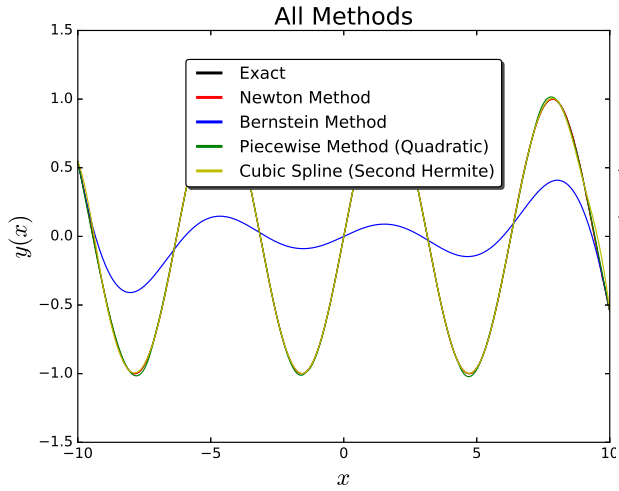


(a) Global view of interpolating methods.

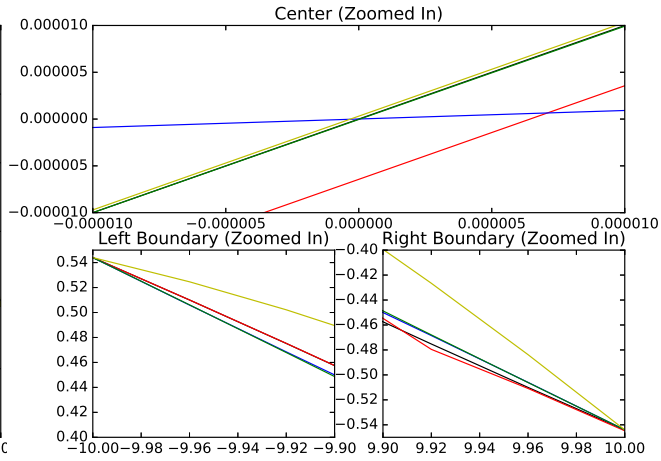


(b) Local view of interpolating methods.

Figure 3: Interpolation for $f(x) = |x| + x/2 - x^2$ with 22 uniform mesh points on [-1,1].

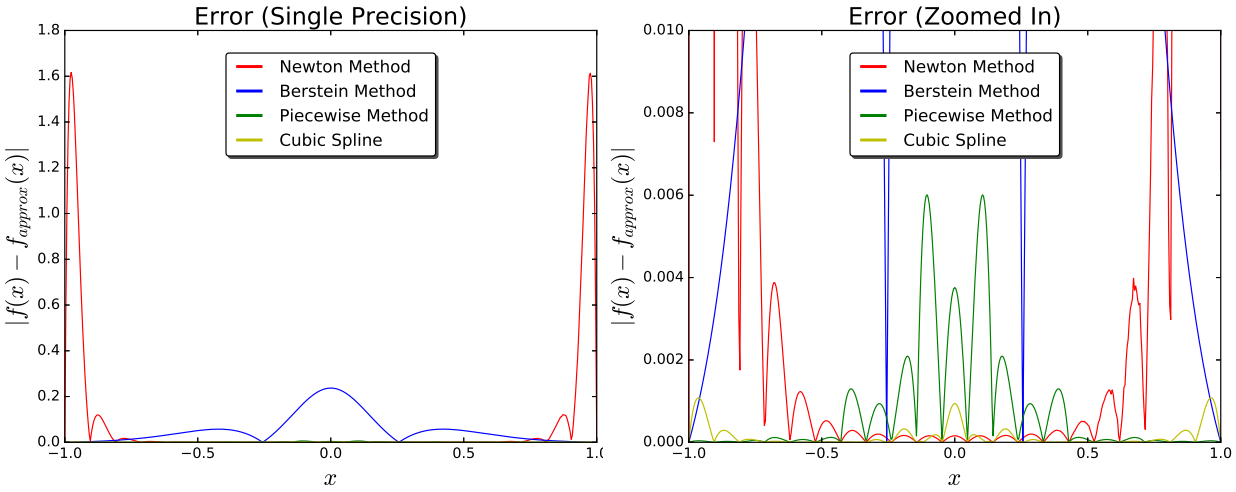


(a) Global view of interpolating methods.



(b) Local view of interpolating methods.

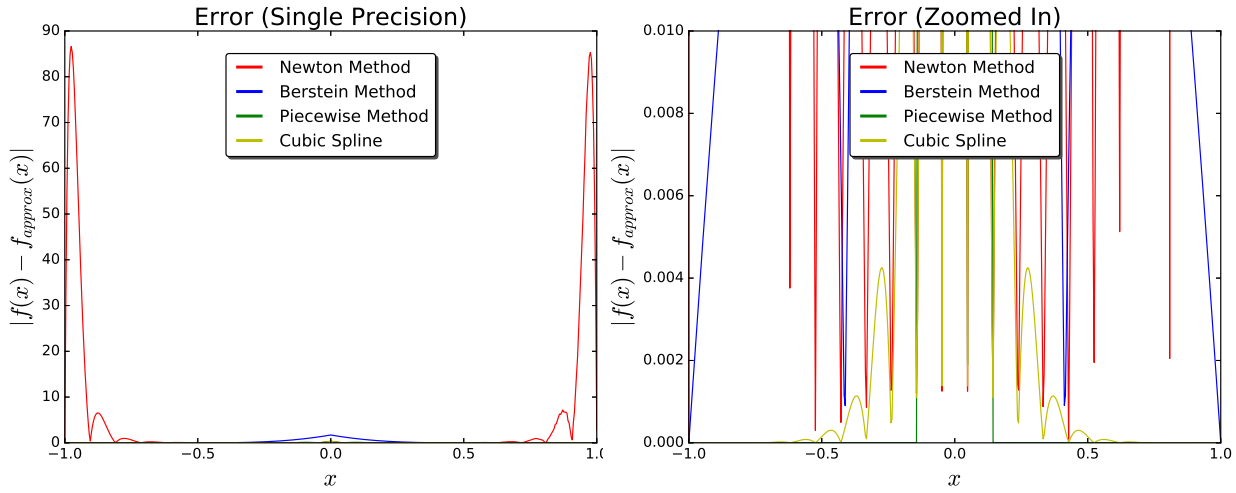
Figure 4: Interpolation for $f(x) = sin(x)$ with 22 uniform mesh points on [-10,10].



(a) Global view of error.

(b) Local view of error.

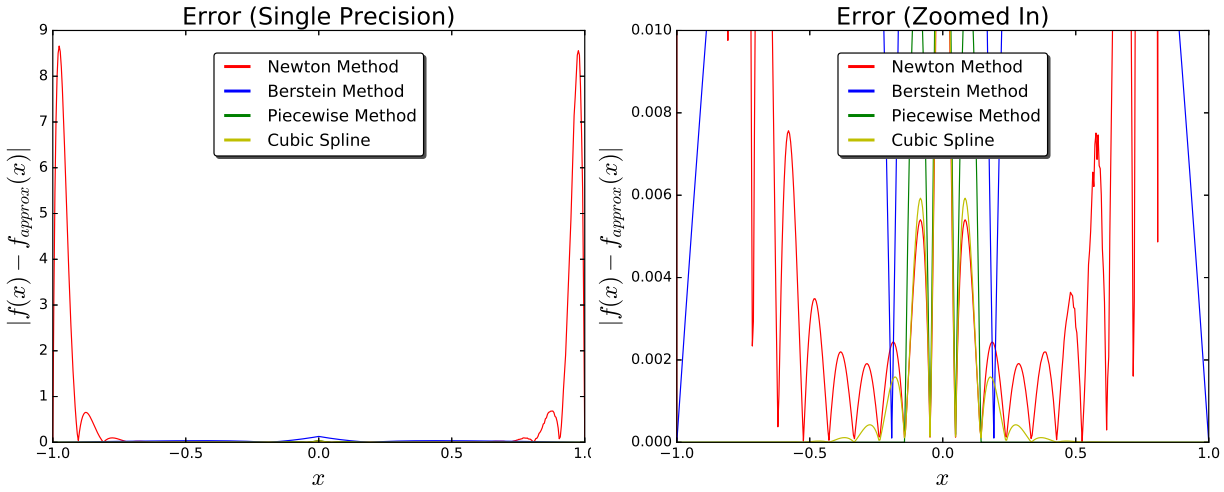
Figure 5: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = 1/(1 + 10x^2)$ (double precision evaluation).



(a) Global view of error.

(b) Local view of error.

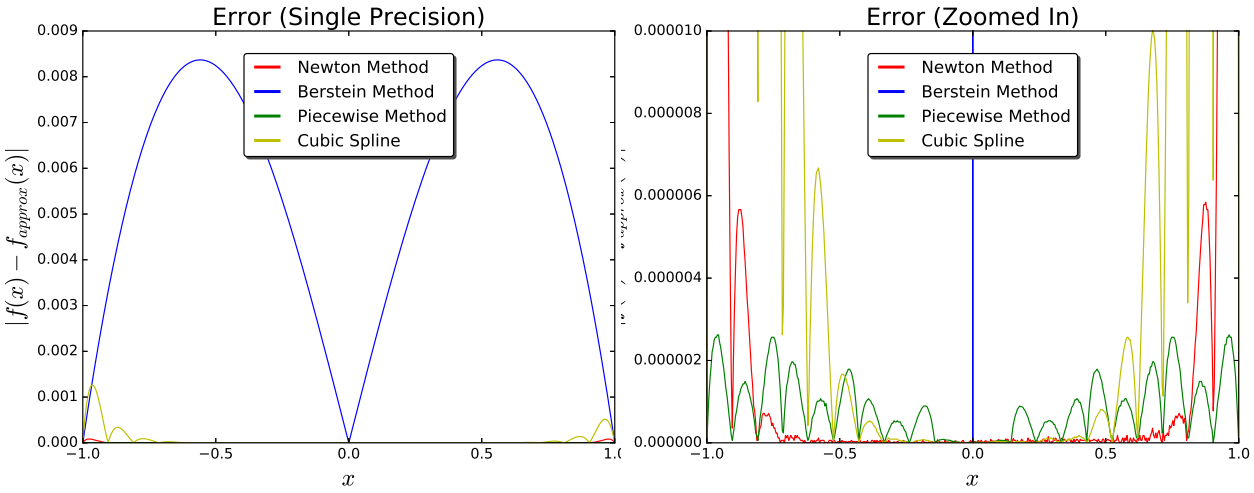
Figure 6: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = |10x| + x/2 - x^2$ (double precision evaluation).



(a) Global view of error.

(b) Local view of error.

Figure 7: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = |x| + x/2 - x^2$ (double precision evaluation).

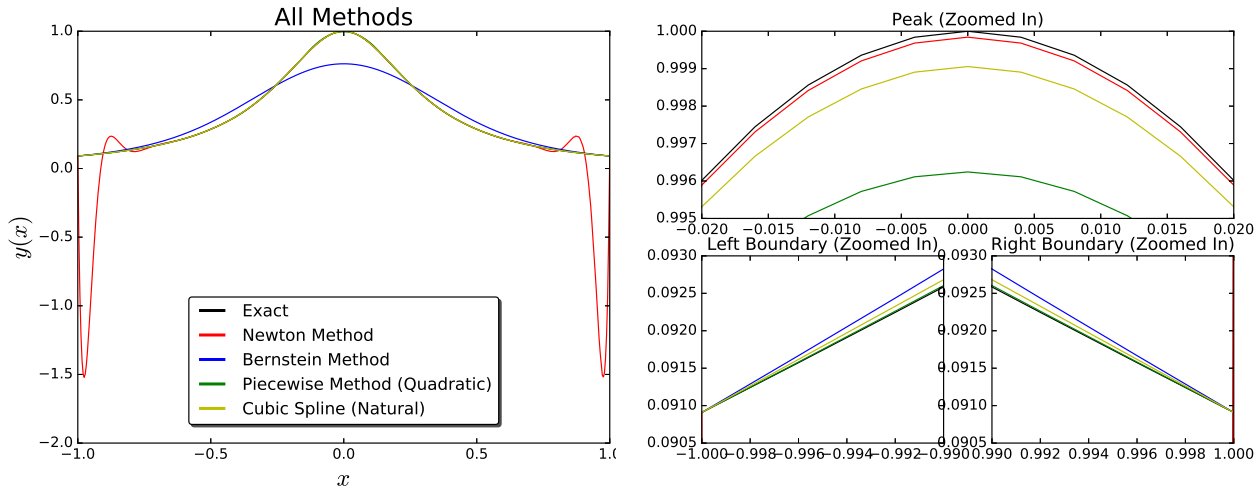


(a) Global view of error.

(b) Local view of error.

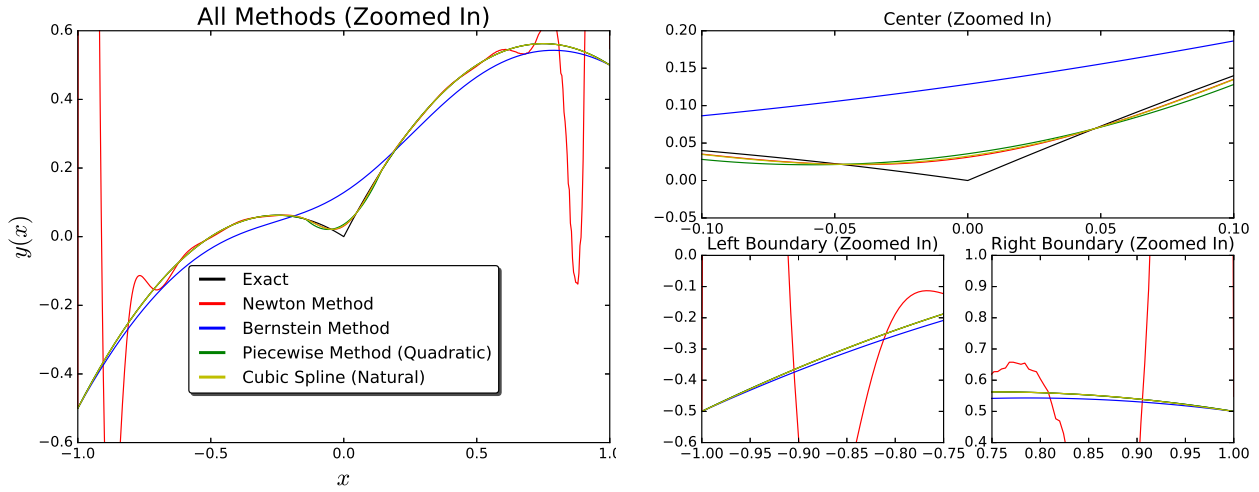
Figure 8: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = sin(x)$ (double precision evaluation).

7.2 Natural Boundary Conditions



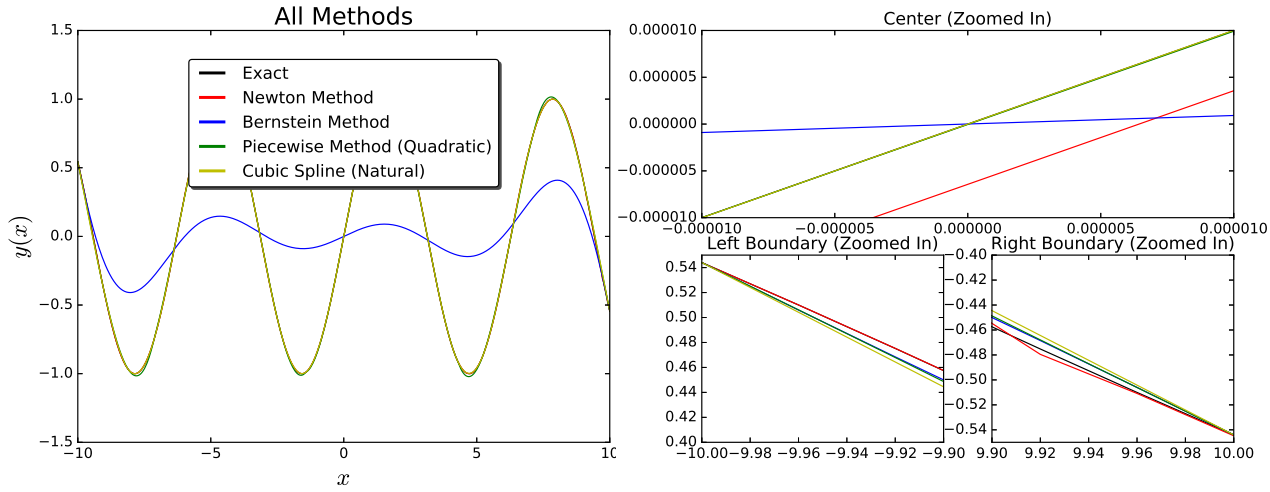
(a) Global view of error. (b) Local view of error.

Figure 9: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = 1/(1 + 10x^2)$ (double precision evaluation).



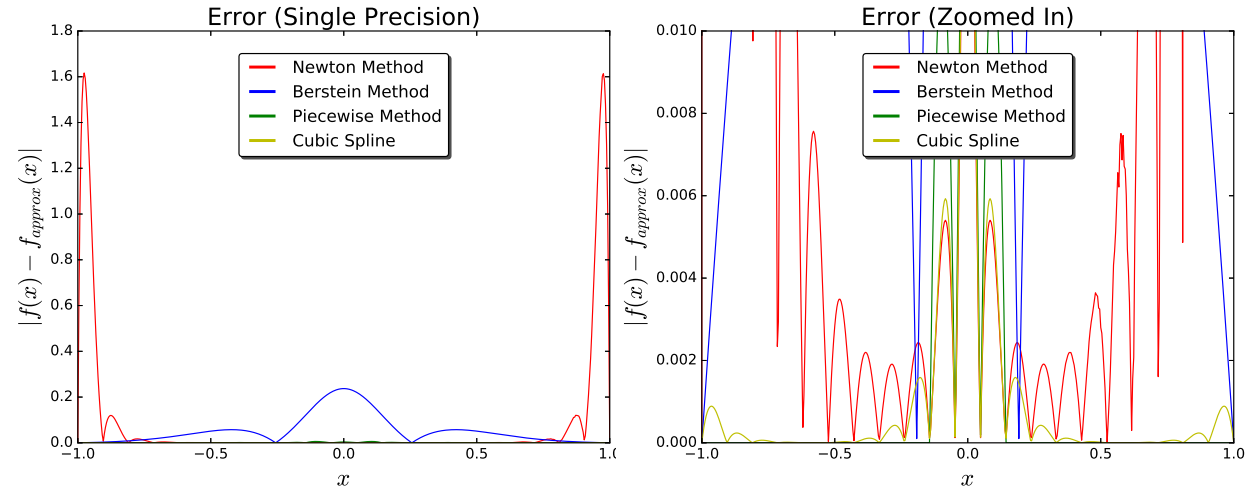
(a) Global view of error. (b) Local view of error.

Figure 10: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = |x| + x/2 - x^2$ (double precision evaluation).



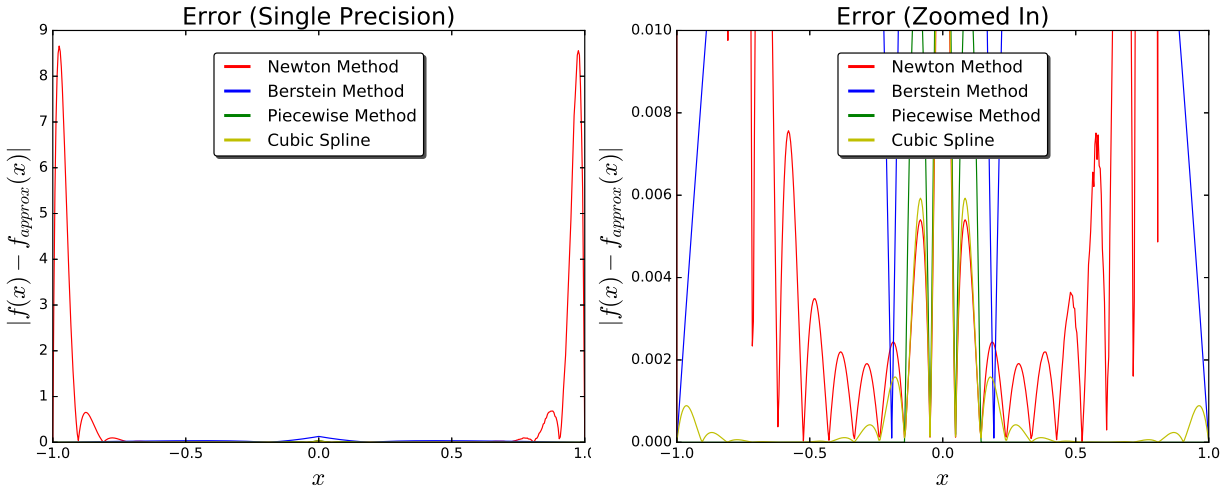
(a) Global view of error. (b) Local view of error.

Figure 11: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = sin(x)$ (double precision evaluation).



(a) Global view of error. (b) Local view of error.

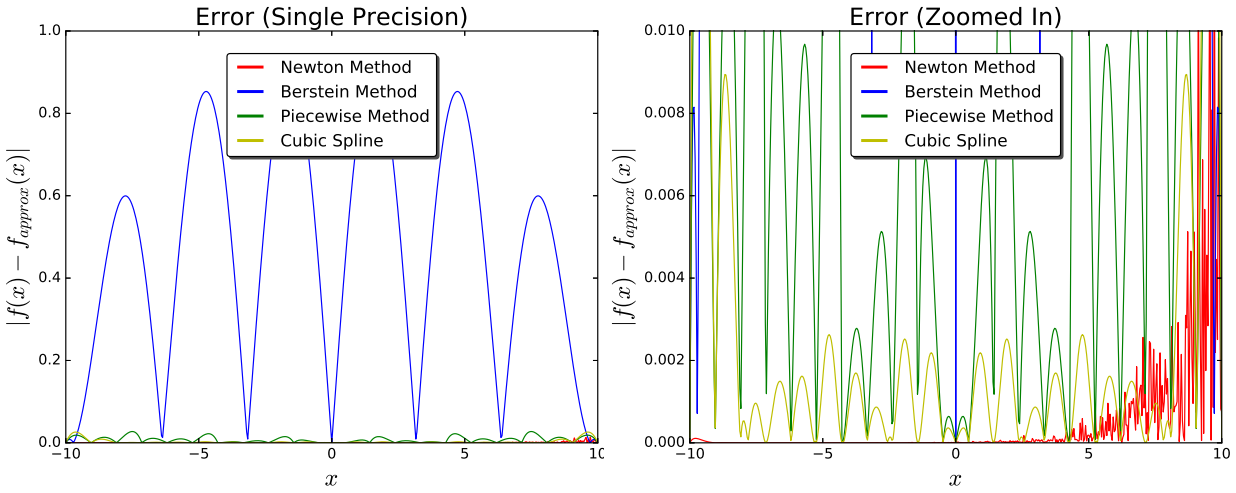
Figure 12: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = 1/(1 + 10x^2)$ (double precision evaluation).



(a) Global view of error.

(b) Local view of error.

Figure 13: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = |x| + x/2 - x^2$ (double precision evaluation).

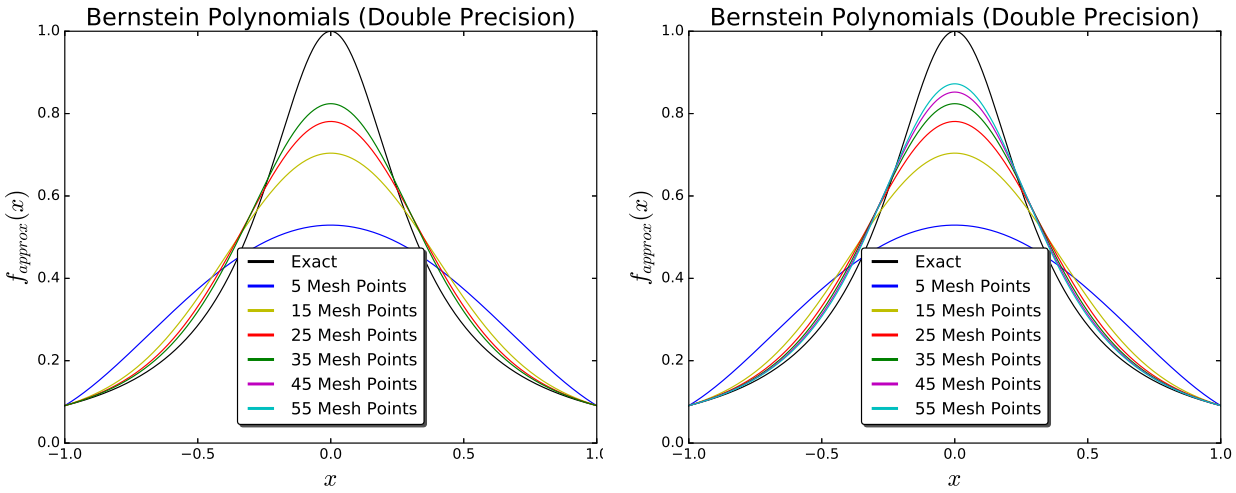


(a) Global view of error.

(b) Local view of error.

Figure 14: Error for interpolating methods $f_{approx}(x)$ vs exact $f(x) = sin(x)$ (double precision evaluation).

7.3 Other Figures



(a) Single Precision.

(b) Double precision.

Figure 15: Bernstein polynomials converge to our function but can only do as well as the precision set.