Interpolation Analysis, Program 2

Colton Piper

October 23th, 2017

1 Executive Summary

This report focuses on three different interpolation techniques and the Bernstein polynomials which does converge to the function as the degree gets higher. The first one is constructing a global polynomial of degree n to interpolate the $n + 1$ mesh or points given. The next are the mesh getting split up into sub intervals and having lower degree polynomials interpolate the mesh in the sub intervals and the mesh on the boundaries of the sub intervals. The last one is cubic splines which is a piece-wise function, but different from the last because we construct them to be $C^{(2)}$ continuous even at the boundaries of the sub intervals. We then analyzed these analytically and experimentally for two different functions. We saw that

2 Statement of the Problem

The first problem in this program is implementing all for the codes to get the polynomials or piece-wise polynomials. The Newton Form algorithm has two functions, one which calculates the divided differences that are required for the polynomial construction and the other which evaluates the the function at a set of points passed into the it. The piece-wise polynomial is similar and also uses both functions of the newton form because the polynomial in the sub interval is still going to have to interpolate the mesh in that interval. The spline we had to implement needed to accept both the Hermite boundary conditions and the natural conditions.

After creating the code we need to analyze the time complexities and space complexities of each of our algorithms. Then discuss the efficiency of each interpolation and discuss how having higher degree polynomials can make the approximation worse. Now each code should also be able to be implemented in time double or single precision and with this we will need to derive some error bounds for our two functions given and see if the code and implementation of the code using those functions matches the error bounds. This will be for both piece-wise interpolation polynomials, i.e. the regular piece-wise and the splines.

The two functions given are

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

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

on the interval $[-1, 1]$. With these and a few other functions we will experiment how the error changes with different amounts of mesh for a function. In the end we will be showing how the experiment compares to the theory we have discussed.

3 Description of the Mathematics

3.1 Newton Interpolation Form

The newton for of the interpolating polynomial is similar to the the Taylor Expansion of a function. The divided difference of recursion formula is as follows

$$y[x_0, x_1, \cdots, x_n] = \frac{y[x_1, \cdots, x_n] - y[x_0, x_1, \cdots, x_{n-1}]}{x_n - x_0}.$$

With these we get the interpolating polynomial of

$$y(x) = y[x_0] + y[x_0, x_1](x - x_0) + \cdots + y[x_0, \cdots, x_n](x - x_0) \cdots (x - x_{n-1}).$$

It is similar to the Taylor expansion because the divided difference in a sense is finding slopes of slopes. They could be taken as approximations of derivatives. Like $y[x_0, x_1, x_2]$ would be an approximation of the second derivative. Thus the interpolating polynomial

looks like the Taylor expansion where $y[x_0, \cdots, x_n] \approx f^{(n)}(x)$.

3.2 Bernstein Polynomial

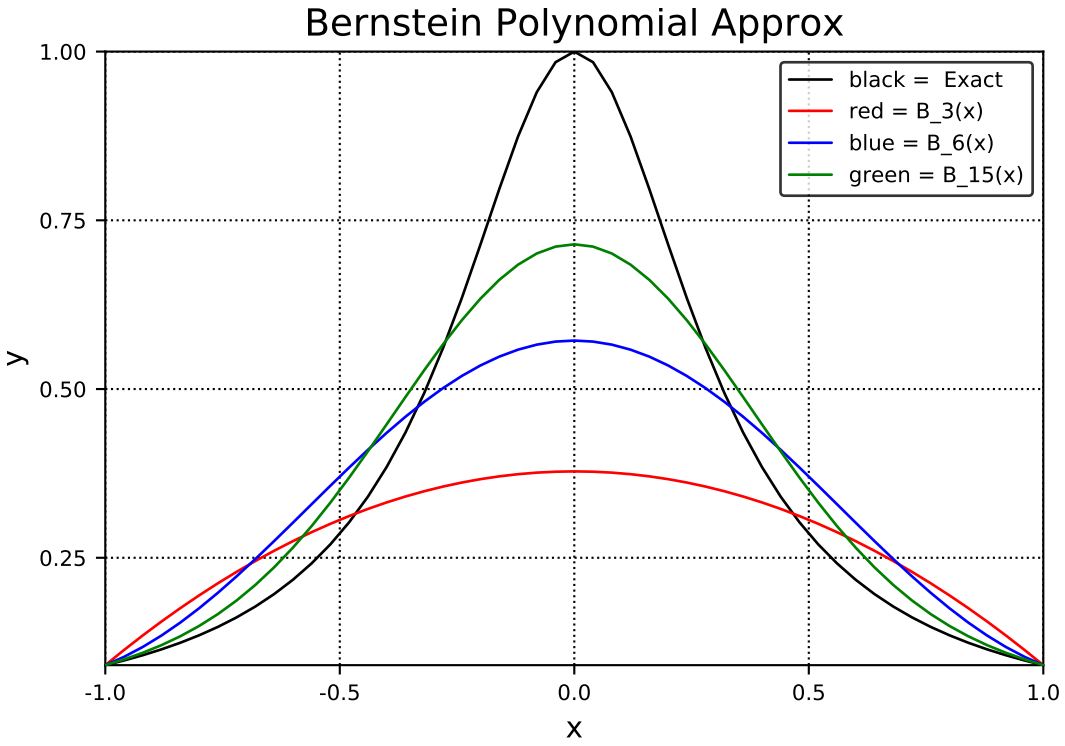
The Bernstein Polynomial is the following function.

$$B_n(x) = \sum_{k=0}^{n} f(\frac{k}{n}) \frac{n!}{k!(n-k)!} x^k (1-x)^{n-k} \quad \text{for } x \in [0,1]$$

Now to use this this function for any interval we have to map the domain to $[0,1]$. Thus if the domain is $[a,b]$ we can map the domain to $[0,1]$ using $\frac{x-a}{b-a}$ where $x \in [a,b]$. Thus we can see that

$$\begin{aligned}
\frac{x-a}{b-a} &= \frac{k}{n} \\
x - a &= \frac{k}{n}(b-a) \\
x &= \frac{k}{n}(b-a) + a \quad \text{for } x \in [a,b] \\
B(x) &= \sum_{k=0}^{n} f(\frac{k}{n}(b-a)+a) \frac{n!}{k!(n-k)!} \left(\frac{x-a}{b-a}\right)^k \left(1 - (\frac{x-a}{b-a})\right)^{n-k} \quad \text{for } x \in [a,b].
\end{aligned}$$

Thus now we have the Bernstein Polynomial for any domain on the real number line. This does converge to the function as $n \to \infty$, but slowly. Here are the Bernstein polynomials with $n = 3, 6, 15$ approximating the function $1/(1+10x^2)$. We can see that it is converging



3.3 Cubic Splines

Cubic splines are smooth piece-wise polynomials that are $C^{(2)}$ globally through the interval of interest. Something that the piece-wise interpolating polynomials lacked even with Hermite conditions it only got it to $C^{(1)}$ globally. We use the constraints that the splines must be continuous on both the first and second derivatives to help us set up a matrix we can solve. First we start with

$$s''_{i-1}(x) = s''_{i-1}\frac{(x_i - x)}{hi} + s''_i\frac{(x - x_{i-1})}{h_i}$$

where $h_i = x_i - x_{i-1}$, which is the Lagrange form for the second derivative. Then integrating it twice we get.

$$s_{i-1}(x) = s''_{i-1}\frac{(x_i - x)^3}{6hi} + s''_i\frac{(x - x_{i-1})^3}{6h_i} + C_{i-1}(x - x_{i-1}) + B_{i-1}$$

Then with using the continuity of the first derivative and some manipulation we get that.

$$a_i s''_{i-1} + 2s''_i + b_i s''_{i+1} = d_i$$

where $a_i = \frac{h_i}{h_i+h_{i+1}}$, $b_i = \frac{h_{i+1}}{h_i+h_{i+1}}$, and $d_i = 6f[x_{i-1}, x_i, x_{i+1}]$. Thus we have the matrix that

$$\begin{bmatrix} a_1 & 2 & b_1 & 0 & 0 & \cdots & 0 \\ 0 & a_2 & 2 & b_2 & 0 & \cdots & 0 \\ \vdots & & \cdots & & \cdots & & \vdots \\ 0 & \cdots 0 & 0 & 0 & a_{n-1} & 2 & b_{n-1} \end{bmatrix} \begin{bmatrix} s''_0 \\ \vdots \\ \vdots \\ s''_n \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ \vdots \\ d_{n-1} \end{bmatrix}.$$

We need two more constraints to be able to solve for this linear system. That is with the Boundary conditions. We have either the natural $s''_0 = s''_n = 0$, the first Hermite $s'_0 = f'(a)$, $s'_n = f'(b)$, and the second Hermite $s''_0 = f''(a)$, $s''_n = f''(b)$. With the natural the boundary conditions the matrix becomes

$$\begin{bmatrix} 2 & b_1 & 0 & 0 & \cdots & 0 \\ a_2 & 2 & b_2 & 0 & \cdots & 0 \\ \vdots & & \cdots & & \cdots & \vdots \\ 0 & \cdots 0 & 0 & 0 & a_{n-1} & 2 \end{bmatrix} \begin{bmatrix} s''_1 \\ \vdots \\ \vdots \\ s''_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ \vdots \\ d_{n-1} \end{bmatrix}$$

and with second Hermite boundary conditions the matrix becomes

$$\begin{bmatrix} 2 & b_1 & 0 & 0 & \cdots & 0 \\ a_2 & 2 & b_2 & 0 & \cdots & 0 \\ \vdots & & \cdots & & \cdots & \vdots \\ 0 & \cdots 0 & 0 & 0 & a_{n-1} & 2 \end{bmatrix} \begin{bmatrix} s''_1 \\ \vdots \\ \vdots \\ s''_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 - a_1 f''(a) \\ \vdots \\ \vdots \\ d_{n-1} - b_{n-1} f''(b) \end{bmatrix}.$$

Now with first Hermite conditions we have to look $s'_0(a)$ and $s'_{n-1}(b)$. We can see that

$$\begin{aligned} s'_{i-1}(x) &= s'_{i-1}(\frac{h_i}{6} - \frac{(x_i - x)^2}{2h_i}) + s''_i(\frac{(x - x_{i-1})^2}{2h_i} - \frac{h_i}{6}) + \frac{f_i - f_{i-1}}{h_i} \\ s'_0(a) &= s''_0(\frac{h_1}{6} - \frac{h_1}{2}) + s''_1(-\frac{h_1}{6}) + \frac{f_1 - f_0}{h_1} = f'(a) \\ s''_0\frac{h_1}{3} + s''_1\frac{h_1}{6} &= \frac{f_1 - f_0}{h_1} - f'(a) \\ 2s''_0 + 1s''_1 &= \frac{6}{h_1}\left(\frac{f_1 - f_0}{h_1} - f'(a)\right). \end{aligned}$$

Then we get

$$1s''_{n-1} + 2s''_n = \frac{6}{h_n}\left(f'(b) - \frac{f_n - f_{n-1}}{h_n}\right)$$

similarly. Thus the matrix becomes

$$\begin{bmatrix} 2 & 1 & 0 & & \cdots & & 0 \\ a_1 & 2 & b_1 & 0 & 0 & \cdots & 0 \\ 0 & a_2 & 2 & b_2 & 0 & \cdots & 0 \\ \vdots & & \cdots & & \cdots & & \vdots \\ 0 & \cdots 0 & 0 & 0 & a_{n-1} & 2 & b_{n-1} \\ 0 & \cdots 0 & \cdots & 0 & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} s''_0 \\ \vdots \\ \vdots \\ s''_n \end{bmatrix} = \begin{bmatrix} d_0 \\ \vdots \\ \vdots \\ d_n \end{bmatrix}.$$

All three matrices with the boundary conditions can be solved using Thomas's algorithm for tridiagonal matrices. Then with s''_i we can solve for C_i and B_i. Thus we have our splines.

4 Description of the Algorithms and Implementation

4.1 Newton Interpolation Form

The complexities of each algorithm are quadratic at most as we will see and are very precise when it comes to indexing in arrays used to store data. The first algorithm is newtons method and more precisely the divided difference function and the newton form evaluation function. The time complexity of the construction of the newton form is of quadratic order, more precisely $O(3n^2)$, with a storage of only $O(n)$ space. The divided difference uses the two arrays for the x and y mesh and an array for the divided differences themselves. First the code initializes the array with all of the y mesh. Then it will go through and calculate all of the first order divided differences. While computing these it will replace the no longer needed $y[x_i]$, for example $y[x_0, x_1]$ will replace $y[x_1]$. Now while doing this we will need to two temporary variables that hold $y[x_i]$ for one more iteration because we will need it to compute the next divided difference. Thus in our example $y[x_1]$ will get held for one more iteration so $y[x_1, x_2]$ can be computed. Then after that temporary variable will be changed to hold $y[x_2]$. Thus we will compute the whole divided difference

but we will be only storing $n+1$ of them at a time.

The evaluation of the newton form is quite simple. The newton form is

$$y(x) = y[x_0] + y[x_0, x_1](x - x_0) + \cdots + y[x_0, \cdots, x_n](x - x_0) \cdots (x - x_{n-1}).$$

Thus the evaluation just iterates through each term starting with the constant. Then we can easily have a variable called w that updates each iteration by multiplying itself by $(x - x_i)$ and having the evaluation adding to itself $d_i * w$ where d_i is the divided difference. Thus the order of time is linear only being $O(5n)$ with a space complexity of $O(1)$ because all we use is the variables w and eval. Thus Newton's Form as a whole has a time complexity of about $O(3n^2)$ and a linear space order $O(n)$.

4.2 Bernstein Polynomial

Berntein's polynomial is nice the fact that both the time and space complexities are linear if you code it with efficiency in mind. Bernstein's polynomial is

$$B(x) = \sum_{k=0}^{n} f(\frac{k}{n}) \frac{n!}{k!(n-k)!} x^k (1-x)^{n-k}$$

on the interval $[0, 1]$. To get linear time complexity we sacrifice constant space complexity. We need to create two arrays, a factorial array and an array holding the power values up to $(1-x)^n$ for $1 \leq k \leq n$. With those two arrays and a variable that updates x^k each iteration we do not have to compute factorials within the for loop with iterates the evaluation of the polynomial at a point. Thus we get a time complexity of $O(14n)$ and a space of $O(2n)$.

4.3 Piece-wise Interpolating Polynomial

Next is the regular not globally differentiable piece-wise polynomials. The implementation of this is an interesting case as the time complexity of the sort function of the coder's choice is essentially the bound for it. I used a bubble sort which has a quadratic worst case complexity and a linear complexity when it is sorted. The piece-wise algorithm determines what sub interval the point being evaluated lies and then uses the newton form

to evaluate the point at the desired mesh. The search is linear time complexity. So we have three complexities to worry about in this. The bubble sort of $O(\frac{9}{2}n^2)$ flops, the search of $O(n)$ flops, and the newton evaluation of $O(3d^2)$ flops where d is the degree of the piece-wise polynomial. We only need to pass in the mesh of the sub interval which is why $3d^2$ will be less than n^2 in most cases unless the mesh is already sorted. So in most cases the bubble sort bounds the time complexity. Thus a better sort like Merge Sort or Heap Sort with complexities of $n \log n$ flops may be better as large unsorted data sets will take much longer with the bubble sort. Storage complexity is linear as the arrays passed into the function are the meshes which have n size.

4.4 Cubic Splines

Spline complexities are similar to the piece-wise in the sense that it is bounded by the sorting algorithm in use by the coder. Thus in this case the time complexity is bounded by $O(\frac{9}{2}n^2)$ again. Now thanks to the Thomas algorithm for solving tridiagonal matrices, we can solve the linear system in $O(n)$ time instead of $O(n^3)$ using the Gaussian elimination. The code works in five steps. The first is sorting the data, second is constructing the matrix in the linear system, then using solve the linear system and lastly calculate our other two constants and then evaluating. Now in there we need to find in what sub interval the point being evaluated lives. Now if the mesh is already sorted the time complexity will be around $O(63n)$ as there are a couple non nested for loops that have many operations in them. The storage is linear as we need to create six different arrays inside the function to store all the constants needed.

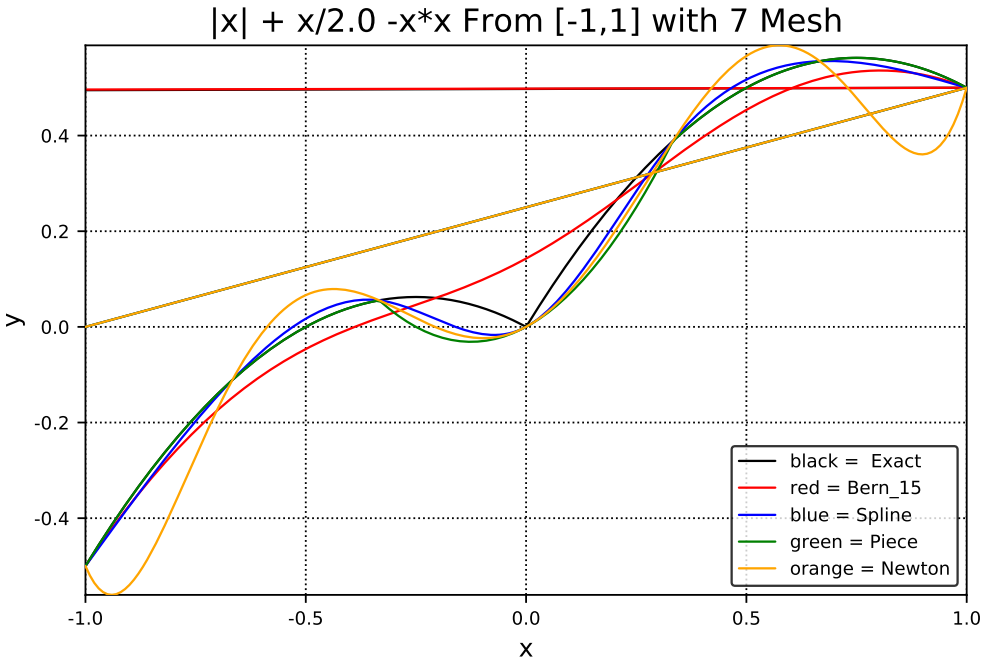
5 Description of the Experimental Design and Results

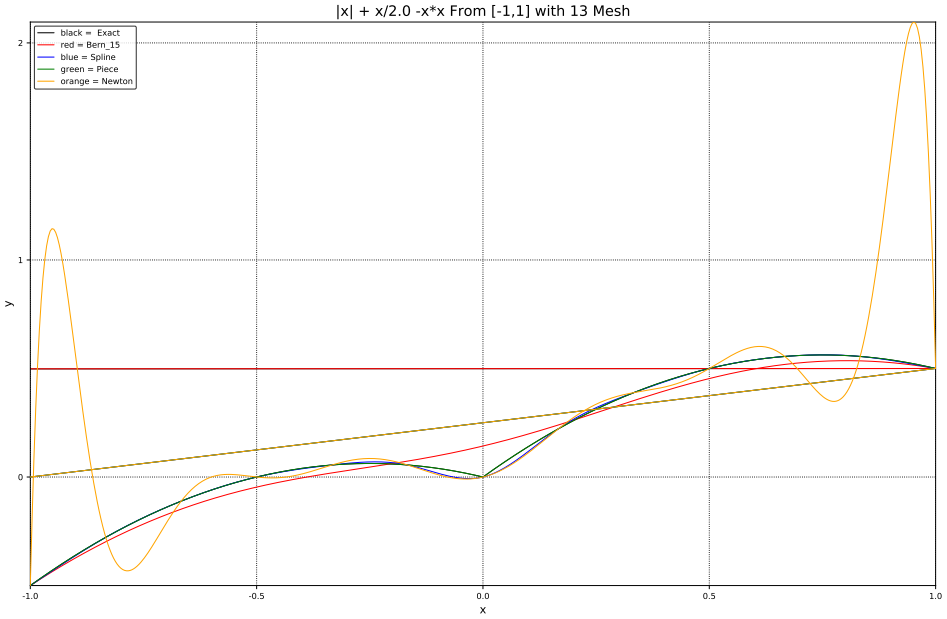
The Experimental design of this was to test it on the two functions

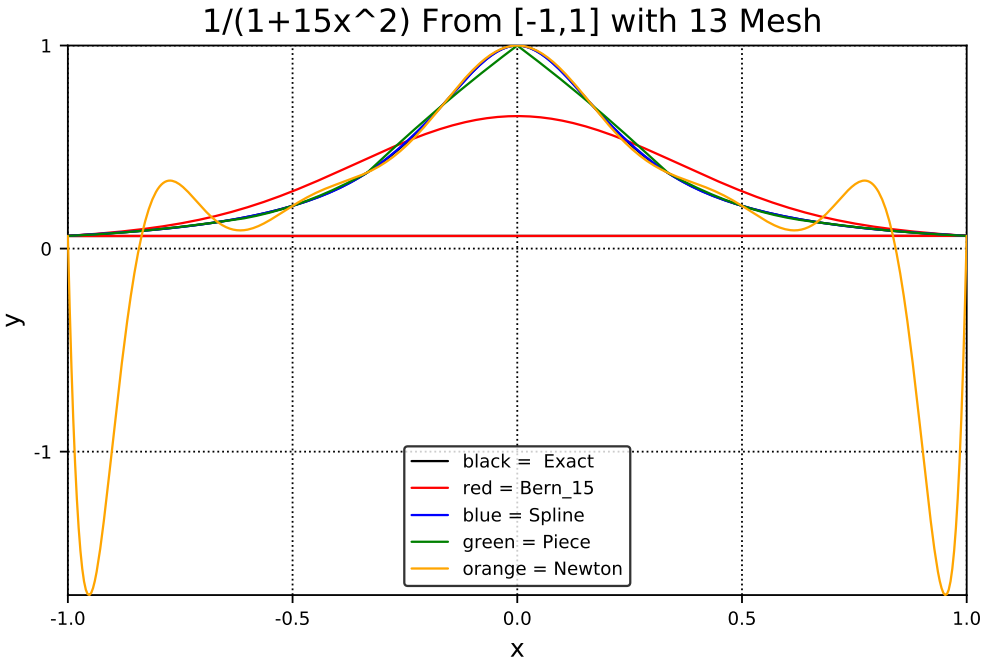
$$f(x) = |\alpha x| + \frac{x}{2} - x^2 \tag{1}$$

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

The first two graphs are using all four polynomials approximating (1) where $\alpha = 1$. The top graph is having a mesh of seven points while the bottom is using 13 points as the mesh. Ignore a red line at $y = 0$ and the linear gold line. Not sure why those show up, but I cannot seem to get rid of them. We can That all four approximations approximate the function decently with 7 mesh. But we can see that as your mesh gets larger the piece wise approximations get better while the global polynomial starts getting large oscillations near the end points of the interval. That phenomena is called Runge's phenomena. Then the third case is with function (2) with $\alpha = 15$ shows Runge's phenomena happening with a 12 degree polynomial.

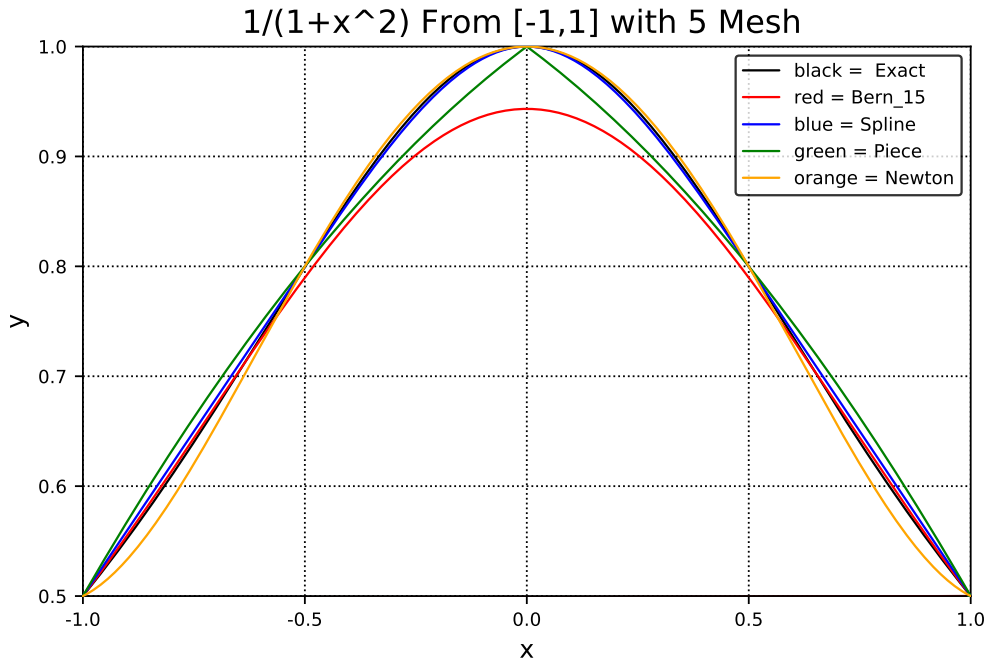


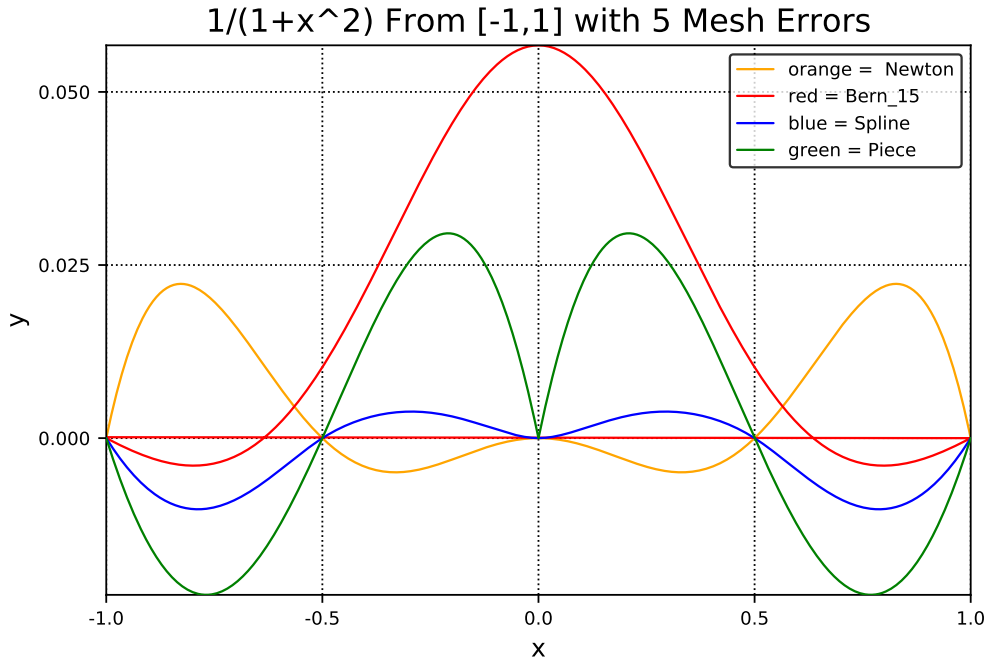




The next graphs are for the function (2) again with $\alpha = 1$. The first graph are the polynomials versus the exact function. Then the second one is the error plots. Thus it seems like Bernstein's Polynomial is good only the fact that it does converge. But evaluating huge degree polynomials can take much longer to evaluate than the other polynomials.

The splines here look to be the best approximation, but all three interpolating polynomials are decent approximations here. This shows that at low degrees the global polynomial can be a good approximation.





6 Conclusions

From the results of the of the of the experiment we saw that for small number meshes the polynomials were all decent approximations for the limited amount of information given about the function. But as you increase the mesh, we saw that the splines and piece-wise seem to be the best approximations. The newton form falls short with bigger mesh as Runge's phenomena pops up with any function. Then Bernstein's polynomial, which does converge as you increase the mesh, converges very slowly and you need a uniform mesh for it to work. Thus I do not foresee it being used in practice as it is slow to converge and you need a specific set of mesh for it to work and a lot of them.

Now we also saw that each of these polynomials functions can be constructed and evaluated in decent time and storage complexities. The Newton can be evaluated in $O(3n^2)$ flops with $O(n)$ space. Then Bernstein can be evaluated $O(14n)$ flops with $O(2n)$ space. Piece-wise has a sorting algorithm which bumps the time complexity up to $O(9/2 * n^2)$ with storage of $O(2d)$. Lastly the Splines are $O(9/2 * n^2)$ for time complexity with $O(6n)$ storage.