Computational Project 1 Group 2 Report

Prepared by: Zhenyuan Ni, Jade Chung, John Lain, Hank McKinley, & Charlie White

Abstract

In this report, we will analyze the stability of three different methods of interpolating data: trigonometric interpolation, cubic spline, and Lagrange polynomial interpolation. We will compare the results of tests on both discrete data points and those generated from a function. When testing with discrete data, we found that the Lagrange polynomial interpolation will suffer Runge's phenomenon, which cannot be avoided with Chebyshev nodes due to our use of fixed data points for comparison. Cubic splines and trigonometric interpolations performed admirably in multiple test cases.

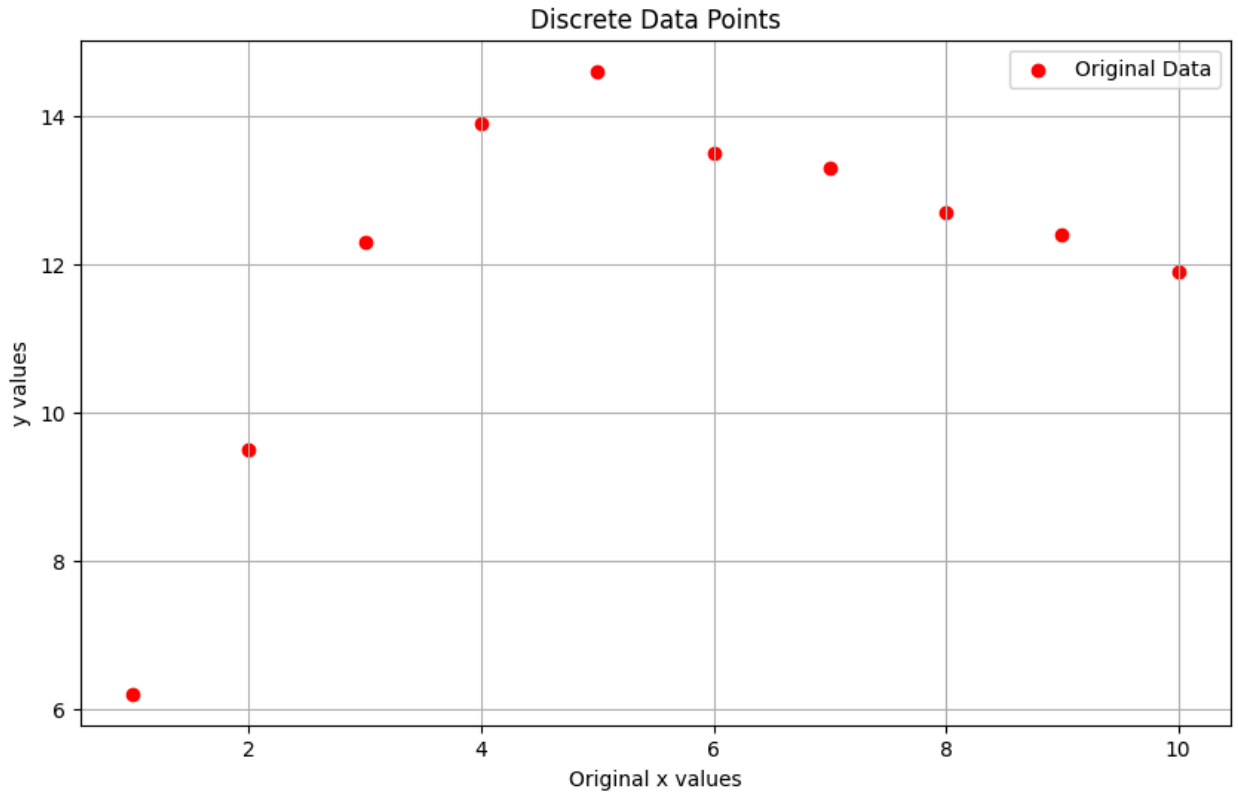
Introduction

In this project, we chose to focus our efforts on Topic 2: interpolation study on cubic splines. All of us were curious about the stability and effectiveness of different interpolation methods. In particular, we wanted to test Lagrange polynomial interpolation, trigonometric interpolation, and cubic spline interpolation. All three of these techniques are useful and allow for the estimation of data, but which one of these methods is the best? To find out, we will be scrutinizing each method and its performance on both discrete data points and function-generated data and then comparing the methods' performance with absolute error and RMSE. We will first take a look at the data we aim to interpolate. Then, we will dive into each interpolation method. Finally, we will compare the performance of each method to decide which method is best.

Data

In this section, we will present the different data sets we will attempt to interpolate. We chose a combination of arbitrary data and data pulled from functions in order to compare both real-world situations and situations where we have a more concrete idea of the true solution.

For our discrete data, we chose the following points relatively arbitrarily:



For functions, we chose to use the normal distribution, a trigonometric function, and the Runge Function as follows:

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

Lagrange Interpolation

Lagrange interpolation is a technique that utilizes Lagrange polynomials constructed based on known data to create a polynomial function that contains all the data points. This polynomial can then be used to interpolate unknown values within the data range. For a first-order polynomial, the Lagrange interpolation formula is:

$$f(x) = \frac{(x-x_1)}{(x_0-x_1)} \times y_0 + \frac{(x-x_0)}{(x_1-x_0)} \times y_1$$

Generalizing to the n-th order, given n + 1 nodes, the Lagrange interpolation formula becomes:

$$f(x) = \frac{(x-x_1)(x-x_2)...(x-x_n)}{(x_0-x_1)(x_0-x_2)...(x_0-x_n)} \times y_0 + \frac{(x-x_0)(x-x_2)...(x-x_n)}{(x_1-x_0)(x_1-x_2)...(x_1-x_n)} \times y_1 + ... + \frac{(x-x_0)(x-x_1)...(x-x_n-1)}{(x_n-x_0)(x_n-x_1)...(x_n-x_n-1)} \times y_n$$

Source

The resulting polynomial can also be found as a solution to a system of linear equations, which has the form of Ax = b, where A is the Vandermonde matrix, and b is the y data. We can easily solve this system, as it is known that the Vandermonde matrix is invertible under our conditions. The solution returns the coefficients of the unique nth-degree polynomial, so we know it is the same solution as the Lagrange interpolation.

Vandermonde Matrix, which in our case has n = m = (number of points - 1):

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^n \end{bmatrix}$$

Wikipedia

After incorporating this method into Python, here is the code we used:

```
def interpolation(data):
    '''
    Polynomial interpolation using Vandermonde matrix
    Takes data in the form of x in the first column and y in the second
    Returns the coefficients for the interpolating polynomial
    '''
    n = data.shape[0]
    A = np.zeros((n,n))
    A[:, 0] = 1
    for i in range(1, n):
        A[:, i] = data[:,0]**i
    c = np.linalg.solve(A, data[:,1])
    return c
```

Trigonometric Interpolation

Trigonometric interpolation is a technique that uses combinations of the sine and cosine functions to estimate values within a set of data with equally spaced nodes. This method is generally regarded as effective for periodic and/or cyclical data because it can usually model wave-like data very accurately. The generalized trigonometric polynomial of degree *n* is:

$$p(x) = \frac{a_0}{2} + \sum_{k=1}^{n} a_k \cos(k\pi x) + b_k \sin(k\pi x)$$

for real constants a_k, b_k.

Source

As displayed by the above formula, trigonometric interpolation utilizes Fourier transforms and Inverse Fourier Transforms to find the interpolating coefficients that are used to interpolate the data. The code for our project used the Fast Fourier Transform, which is an algorithm for solving the Discrete Fourier transform in a quicker way than the traditional method. The Discrete Fourier transform and the Inverse Discrete Fourier Transform can be represented as matrices, which can be used to solve interpolation problems.

Here is the code we used to implement trigonometric interpolation in Python:

```
def dftinterp(inter, x, n, p):
    c, d = inter
    t = c + (d - c) * np.linspace(0, n - 1, n) / n
    tp = c + (d - c) * np.linspace(0, p - 1, p) / p

    y = np.fft.fft(x)
    yp = np.zeros(p, dtype=complex)
    yp[:n//2 + 1] = y[:n//2 + 1]
    yp[p - n//2 + 1:] = y[n//2 + 1:]

    xp = np.real(np.fft.ifft(yp)) * (p / n)

    #plt.plot(t, x, 'o', tp, xp)
    #plt.show()
    return((tp, xp))
```

We got this code by using the provided MATLAB code in the textbook and asking ChatGPT to translate it. This did implement some errors that we had to then fix.

Cubic Splines

Splines are collections of low-order polynomials glued together. They are useful as an interpolation technique because they ensure continuity and smoothness at the data points. In this project, we will look at only natural cubic splines solutions to our interpolation problems. These

splines also ensure that there is zero curvature at the endpoints. That means our splines (**S**) will take on:

- Form
  - S has the form

$$\begin{aligned} S_1(x) &= y_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 && \text{on } [x_1, x_2] \\ S_2(x) &= y_2 + b_2(x - x_2) + c_2(x - x_2)^2 + d_2(x - x_2)^3 && \text{on } [x_2, x_3] \\ &\vdots \\ S_{n-1}(x) &= y_{n-1} + b_{n-1}(x - x_{n-1}) + c_{n-1}(x - x_{n-1})^2 + d_{n-1}(x - x_{n-1})^3 && \text{on } [x_{n-1}, x_n] \end{aligned}$$

- Property
  1. $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$ for $i = 1, \ldots, n-1$.
  2. $S'_{i-1}(x_i) = S'_i(x_i)$ for $i = 2, \ldots, n-1$.
  3. $S''_{i-1}(x_i) = S''_i(x_i)$ for $i = 2, \ldots, n-1$

Since we are using natural cubic splines, our splines will have the additional property:

- Property 4a
  - $S''_1(x_1) = 0$ and $S''_{n-1}(x_n) = 0$.

Source

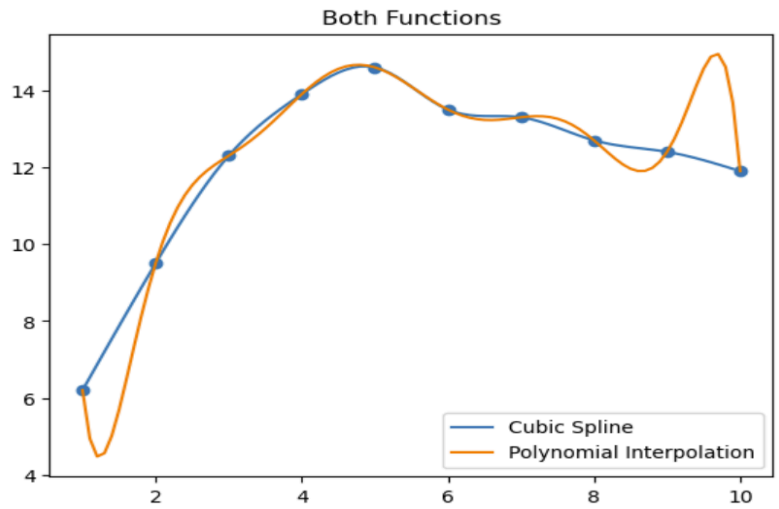
This problem can be reformulated into a system of linear equations, which can be solved to find the c coefficients. This system, when formulated as a matrix, is strictly diagonally dominant and square, so a unique solution can be found through a solver (we use the numpy.linalg.solve function). The c coefficients can then be used to solve for the other b and d coefficients of the polynomials.

Here is the code that we used to implement natural cubic splines:

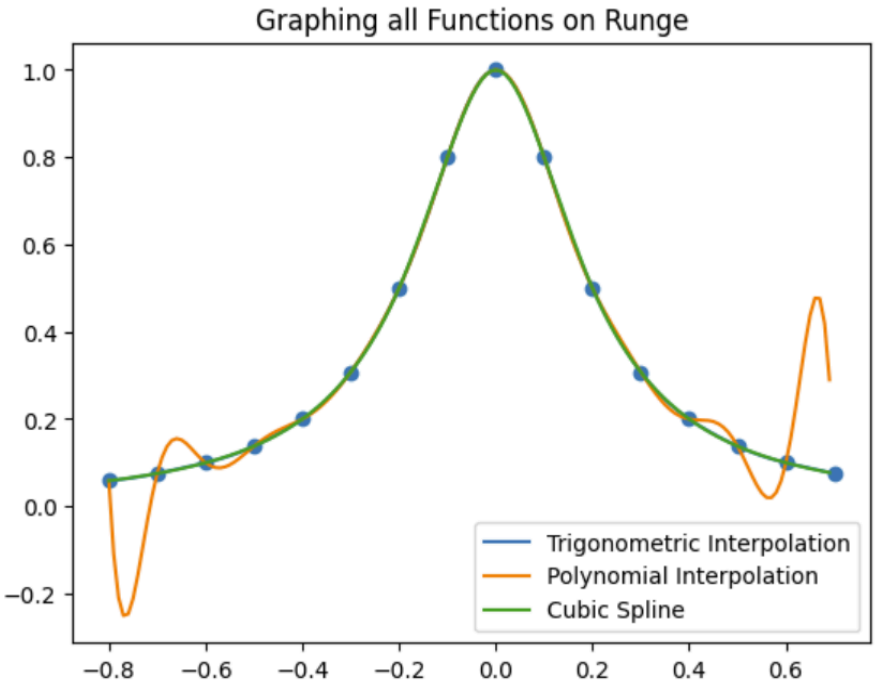
```
def cubicspline(data):
    """
    Creates a set of cubic splines from data
    Returns a set of functions, which allows evaluation of splines
    """
    n = data.shape[0]

    dx = data[1:, 0] -  data[:-1, 0]
    dy = data[1:, 1] -  data[:-1, 1]

    #Creating the diagonal columns of the SDD matrix we want to solve
    diagonal = np.ones(n)
    diagonal[1:n-1] = 2*dx[:n-2] + 2*dx[1:]
    diagonalu = np.zeros(n - 1)
    diagonalu[1:] = dx[1:]
    diagonall = np.zeros(n - 1)
    diagonall[:n - 2] = dx[:n-2]

    #Creating matrices with diagonal columns of the SDD matrix we want to solve
    U = np.diag(diagonalu, 1)
    A = np.diag(diagonal)
    L = np.diag(diagonall, -1)

    #Adding up the matricies to create the final SDD matrix which we can solve to find the c coefficients
    A = A + U + L

    #Creating the constants vector which is on the rhs of the equation Ac = b
    bconstants = np.zeros(n)
    bconstants[1:n-1] = 3*(dy[1:]/dx[1:] - dy[:n-2]/dx[:n-2])

    #Solving for c, b, and d coefficients
    c = np.linalg.solve(A, bconstants)
    d = (c[1:] - c[:n-1])/(3*dx)
    b = dy/dx - dx/3*(2*c[:n-1] + c[1:])

    spline_func = [0]*(n-1)
    for i in range(n - 1):
        spline_func[i] = lambda x, i=i: data[i,1] + b[i]*(x - data[i,0]) + c[i]*(x - data[i,0])**2 + d[i]*(x - data[i,0])**3
    #spline_func is a list which stores the ith cubic polynomial at index i-1, calling spline_func[i - 1](x) evaluates the ith polynomial at x

    return spline_func
```

Observation of Results

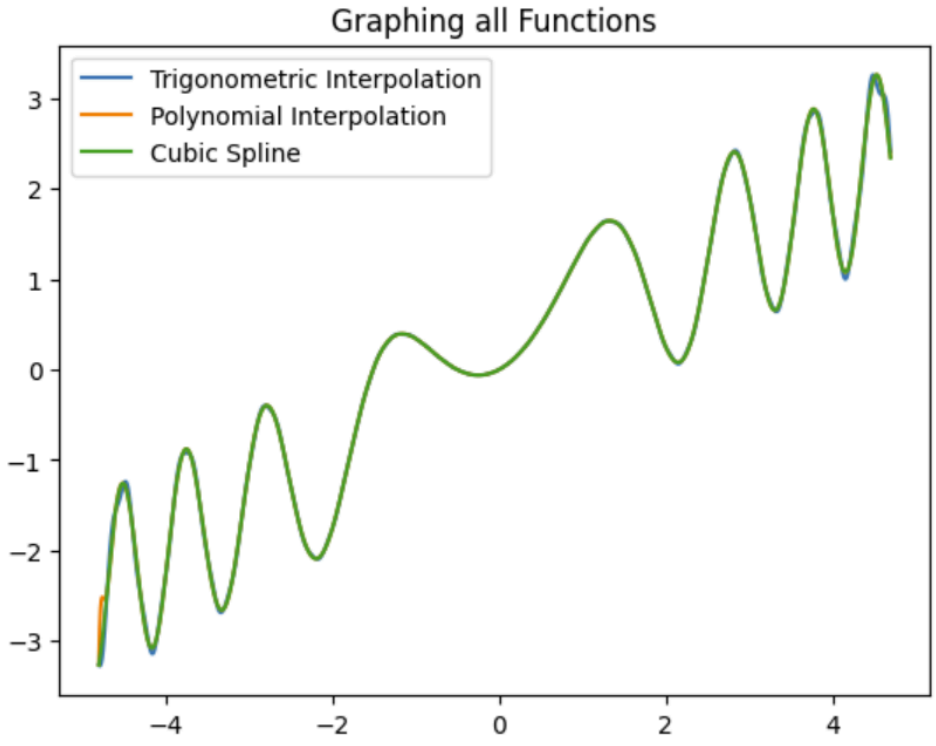
Discrete Data Points:



Runge Function:



Trigonometric Function:



Performance Analysis

In all the cases above, we can see the oscillation behavior of polynomial interpolation near the edge points. This is the main reason we test both the centered interval and the whole domain of the chosen node.

Now, we can take a look at how our three methods performed compared to each other in a specific example:

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

$f(x) = \frac{1}{1+25x^2}$	**Integral of Absolute Error**	**Root Mean Square Error**
Cubic Spline	0.0008016675976101905	2.3574770665952617e-09
Polynomial Interpolation	0.057889896198926774	1.055085e-03
Trigonometric Interpolation	0.001195	1.802198e-09

This table is the error term that includes the edge part (where Runge's phenomenon dominates the polynomial interpolation). In the table, we can see the cubic spline and trigonometric interpolation have much less absolute error and root mean square error, which means it has higher accuracy for the approximation. The next table is the error term only with the middle ⅓ of the x domain to avoid the influence of Runge's phenomenon on the results.

$f(x) = \frac{1}{1+25x^2}$ (Centered)	**Integral of Absolute Error**	**Root Mean Square Error**
Cubic Spline	0.000781	9.244686e-09
Polynomial Interpolation	0.001614	1.333961e-06
Trigonometric Interpolation	0.000636	2.292486e-08

In this table, we can see the difference between the three methods becomes really low, which means the polynomial interpolation also has good stability and accuracy on the center part. Since the polynomial uses only one expression to express the function, it is efficient for understanding and expressing. The lower degree polynomial used for cubic splines can be an advantage in a situation that needs to predict large amounts of data points which saves computation time.

In the example of standard normal distribution on [-5,5], the error terms look like the form below:

Normal distribution	**Integral of Absolute Error**	**Root Mean Square Error**
Cubic Spline	1.552733e-06	1.276943e-17
Polynomial Interpolation	9.012694e-08	4.291340e-14

Normal distribution (centered)	**Integral of Absolute Error**	**Root Mean Square Error**
Cubic Spline	1.550495e-06	6.039644e-17
Polynomial Interpolation	1.115425e-09	8.366606e-18

The error analysis for a periodic example:

$$f(x) = sin(x^2) + 1/2x$$

$f(x) = sin(x^2) + 1/2x$	**Integral of Absolute Error**	**Root Mean Square Error**
Cubic Spline	0.001866	4.168908e-08
Polynomial Interpolation	0.033635	8.173909e-03
Trigonometric Interpolation	0.198579	4.830536e-06

$f(x) = sin(x^2) + 1/2x$ (centered)	**Integral of Absolute Error**	**Root Mean Square Error**
Cubic Spline	0.001107	5.364809e-07
Polynomial Interpolation	0.000013	2.952655e-11
Trigonometric Interpolation	0.125271	1.772843e-07

From all the above examples, we can find out that the cubic spline has overall good performance at the whole domain, polynomial interpolation works pretty well only at the center part, and the trigonometric interpolation seems quite unstable. A deeper analysis might need to be done after systematically learning trigonometric interpolation later this quarter. For the pros/cons of different methods, as mentioned above, the polynomial interpolation has pros in

understanding and using, and the cubic spline has pros in the easier calculation, especially processing large amounts of data points.

Conclusion

In this report, we analyzed the error generated when implementing various interpolation methods. We chose to measure the error in multiple ways, including the sum of absolute error and the root mean squared error, at both the whole chosen domain and the center part of the domain. This gives us an understanding of how the error is behaving over the entire interval as well as some sense of the average at any given point. This is useful because some interpolation methods exhibit good approximation in certain parts of the interval while generating massive errors in others. Of the three methods we analyzed we found that generally cubic splines and trigonometric interpolation outperformed polynomial interpolations, especially in cases where we are unable to alter the data to fit Chebyshev node spacing. What's more, the cubic spline is more stable compared to trigonometric interpolation.