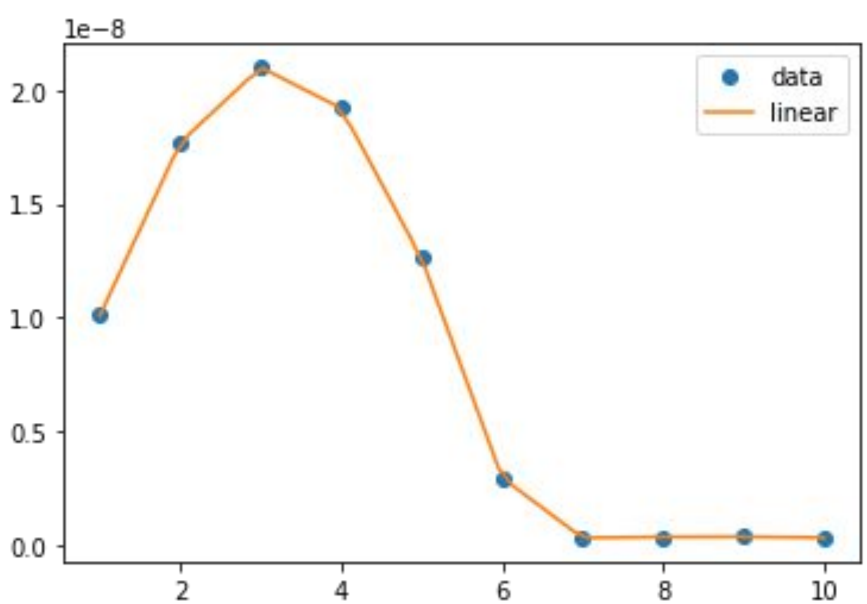
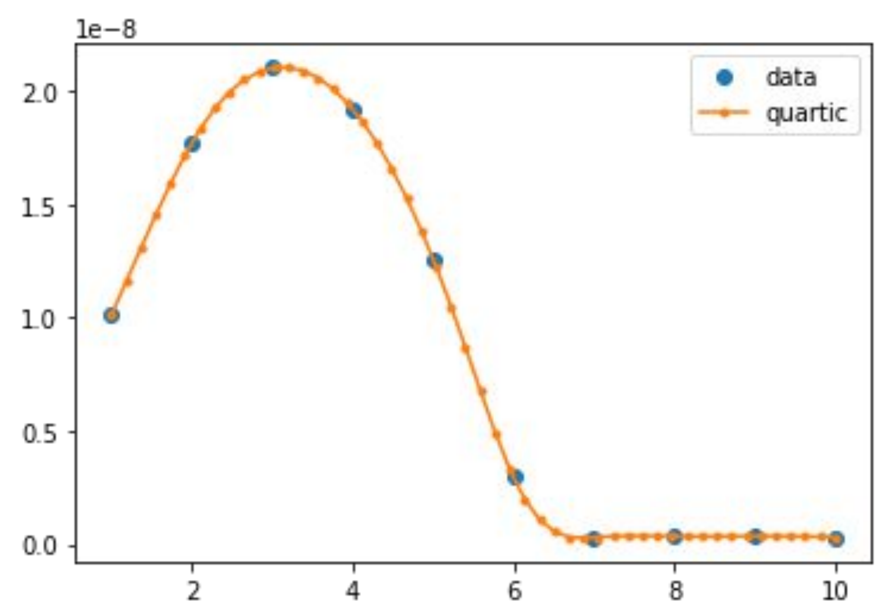
Purpose

Structural engineers, Civil engineers, Mechanical engineers, and many other professions use simulations to study the motion of a body when an external load is applied on the system. Software that uses this simulation discretizes their domain into discrete point called nodes; a differential equation is used to calculate the effect of the external load on the system, which is calculated at each node and generates a value that shows the net force of a field. Currently, modern element analysis methods, like FEA and CFD, use a linear interpolation to connect these nodes and draw a general trend line. This is inaccurate as the function is not differentiable, and a lower order polynomial gives less accurate results. In order to make these simulations more effective and accurate, we wanted to test the accuracy of other existing interpolation methods applied to simulation data and develop our own algorithm to interpolate data: a Quartic Piecewise Spline function.

Question

Our research question involved designing an algorithm and testing its effectiveness against the current interpolation method used by ANSYS and many other major Finite Element Method softwares—linear interpolation. We want to test the effectiveness of this new method of interpolating the data, and if it will produce an accurate solution when given a lower number of nodes.





Abstract

As a polynomial’s order increases, its corresponding interpolation method yields more precise data trends. ANSYS, a software that simulates the effects external forces pose on a geometry, uses linear interpolation to generate values. By using an interpolation method with a higher order polynomial, ANSYS can use less nodes to generate a predicted path, which will increase its runtime. Designing a piecewise quartic spline algorithm will cut down on the number of nodes that need to be interpolated, and provide a strong graphical representation of the trend within the dataset. We tested 3 different geometries—a cube, a square frame, and a pipe—under multiple stresses. These structures were tested under a moment force, linear force, and heat flux. The resultant data points were graphed and interpolated with our new methods. We calculated the error from the actual solution between our model and a linear model (ANSYS generated model). The goal of our research was to minimize the errors in engineering simulations.

Hypothesis

By using a stiffness matrix to store a series of systems of equations that restrain our spline to fit a data set, we can interpolate the function and find the coefficients to a set of equations that will generate a piecewise function quartic polynomial that is smooth and differentiable. With this new function, we can reduce the number of nodes needed to generate an accurate graph of predicted values.

Materials

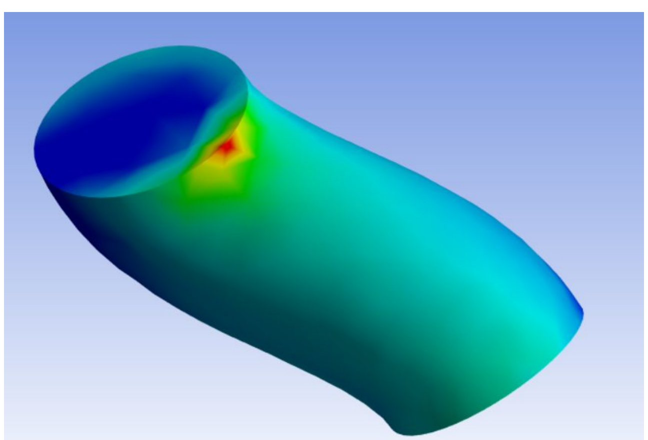
To conduct this experiment, we used ANSYS, a CFD and FEA application, to generate a set of nodes and calculate the new basis for our interpolation. For the code, we used Google Colab as it allows us to read the data from Google Spreadsheets and provides a simple and clean IDE.

Research and Background

To design our algorithm, we researched methods to store multiple systems of equations with an unknown multiplier. We also realized that in other published interpolation sets, the function is smooth and was described by a polynomial, so a restraint we included for our generated function was that it had to be differentiable. Linear algebra has many methods that can translate matrices, so we used an algorithm—a stiffness matrix—to store our matrix. Our dataset will include n nodes, each of which has 5 coefficients, to make a 5(n-1)x5(n-1) matrix. Each set of coefficients can be expressed as a set of vectors, which are all independent of each other, which means there will be only 1 solution to our equation. The solution it produces is a non-trivial solution, and will produce a smooth quartic piecewise spline interpolation.

Procedure (Overall)

1) First, we used ANSYS to generate a geometry, and apply an external load on it (shear forces and torque)
2) Next, we grab all of the results from the data (directional deformation, total stress, and total strain)
3) Then, we designed our own interpolation method* and used it to test the data with 5 and 10 nodes, and compared it to a graph of 100 and 200 nodes to find the accuracy
4) We compared this accuracy to the accuracy of a linear interpolation, and see which is more accurate

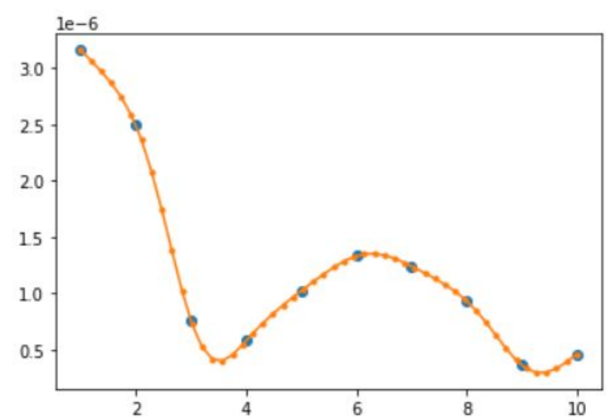


Time [s]	[B] Total Deform	[C] Equivalent E	[D] Equivalent E
1.00E-02	1.96E-08	7.53E-09	7.70E-07
2.00E-02	2.31E-08	5.25E-09	8.77E-07
3.00E-02	2.71E-08	7.14E-09	9.84E-07
4.00E-02	3.14E-08	6.18E-09	1.09E-06
5.00E-02	3.60E-08	6.21E-09	1.19E-06
6.00E-02	4.06E-08	4.13E-09	1.30E-06
7.00E-02	4.57E-08	4.14E-09	1.40E-06
8.00E-02	5.07E-08	6.54E-09	1.49E-06
9.00E-02	5.57E-08	9.96E-09	1.59E-06
0.1	6.07E-08	1.11E-08	1.68E-06
0.11	6.57E-08	9.58E-09	1.77E-06
0.12	7.05E-08	8.26E-09	1.85E-06
0.13	7.54E-08	7.34E-09	1.93E-06

Procedure (Interpolation Design*)

1) First, we created an array of all the stored nodes and the time intervals
2) Next, we created a nxn square matrix with each vector being independent from each other in order to generate a single solution
3) This matrix is multiplied by an (n-1)x1 matrix, where it stores the coefficients for the n-1 splines, and equals another (n-1)x1 matrix which holds the solutions to all of the systems of equations
4) From there, we invert the nxn matrix and multiply it by the solution matrix, which gives us the coefficient matrix
5) This matrix gives us the coefficients to our quartic spline polynomial

```
[ 1, 1, ..., 0,  0]
[16, ., ..., .,  .]
[ ., ., ..., .,  .]
[ ., ., ..., .,  .]
[ ., ., ..., .,  .]
[ 0, ., ..., .,  .]
[ 0, ., ..., ., -1]
```



Results

When quantifying the effectiveness of a quartic spline interpolation against a linear interpolation, we use a data set of 100 real nodes for models 102 and 201 and 500 real nodes for model 301 generated by the ANSYS simulation. From there, we find the residuals between these real nodes and the interpolation throughout the graph. A residual is the difference between the predicted against the actual value. The higher a residual is in magnitude, the higher the margin of error is between the interpolation and the actual graph. After calculating many dozens of datasets of both quartic and linear interpolation values, the quartic interpolation had a lower average residual when compared to the average of other interpolation methods including the linear interpolation. To be specific, all 5 interpolation methods from the scipy interpolate library - Linear, Cubic Spline, Lagrange, Pchipinterpolator, and Make_Interp_Spline - had lower accuracy than Quartic Spline.

	Model102 - B	Model102 - C	Model201 - H	Model201 - I	Model201 - J	Average
Linear	1.09	1.04	1.52	1.50	1.52	1.40
Cubic Spline	0.89	0.95	1.05	0.98	1.00	1.05
Lagrange	4.93	5.26	3.65	4.23	4.69	4.33
PchipInterpolato	0.87	0.87	1.33	1.34	0.98	1.09
Make_Interp_Sp	1.70	1.15	1.84	1.16	1.29	1.42
Quartic Spline	0.53	0.75	0.61	0.81	0.52	0.72

Results

Accuracy(Error)

	Model102 - B	Model102 - C	Model201 - H	Model201 - I	Model201 - J	Average
Linear	1.09	1.04	1.52	1.50	1.52	1.40
Cubic Spline	0.89	0.95	1.05	0.98	1.00	1.05
Lagrange	4.93	5.26	3.65	4.23	4.69	4.33
PchipInterpolato	0.87	0.87	1.33	1.34	0.98	1.09
Make_Interp_Sp	1.70	1.15	1.84	1.16	1.29	1.42
Quartic Spline	0.53	0.75	0.61	0.81	0.52	0.72

Time Analysis

Quartic Spline(s)

	Model102	Model201	Average
Quartic Spline	11.27	14.20	12.74

Ansys(100 nodes)(s)

	Model102	Model201	Average
100 nodes	73.90	78.98	76.44

```
 for i in range(len(y)-1):
#1st row
   for r in range(groupnum*5):
     row.append(0)
   row.append(y[i]**4)
   row.append(y[i]**3)
   row.append(y[i]**2)
   row.append(y[i])
   row.append(1)
   for f in range(((len(y)-2)-groupnum)*5):
     row.append(0)
   temparray.append([])
   count += 1
   for queue in range(len(row)):
     temparray[count-1].append(row[queue])
   row.clear()
   counter += 1
#2nd row same column
   for k in range((groupnum*5)):
     row.append(0)
   row.append((y[i+1])**4)
   row.append((y[i+1])**3)
   row.append((y[i+1])**2)
   row.append(y[i+1])
   row.append(1)
   for j in range(((len(y)-2)-groupnum)*5):
     row.append(0)
   temparray.append([])
   count += 1
   for i in range(len(row)):
     temparray[count-1].append(row[i])
   row.clear()
   groupnum += 1
```

```
[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[16, 8, 4, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 16, 8, 4, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 81, 27, 9, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 81, 27, 9, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 256, 64, 16, 4, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 256, 64, 16, 4, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 625, 125, 25, 5, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 625, 125, 25, 5, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1296, 216, 36, 6, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1296, 216, 36, 6, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2401, 343, 49, 7, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2401, 343, 49, 7, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4096, 512, 64, 8, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4096, 512, 64, 8, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6561, 729, 81, 9, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6561, 729, 81, 9, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10000, 1000, 100, 10, 1]
```

```
[32.0, 12.0, 4.0, 1.0, 0.0, -32.0, -12.0, -4.0, -1.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 108.0, 27.0, 6.0, 1.0, 0.0, -108.0, -27.0, -6.0, -1.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 256.0, 48.0, 8.0, 1.0, 0.0, -256.0, -48.0, -8.0, -1.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 500.0, 75.0, 10.0, 1.0, 0.0, -500.0, -75.0, -10.0, -1.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 864.0, 108.0, 12.0, 1.0, 0.0, -864.0, -108.0, -12.0, -1.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1372.0, 147.0, 14.0, 1.0, 0.0, -1372.0, -147.0, -14.0, -1.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2048.0, 192.0, 16.0, 1.0, 0.0, -2048.0, -192.0, -16.0, -1.0, -0.0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2916.0, 243.0, 18.0, 1.0, 0.0, -2916.0, -243.0, -18.0, -1.0, -0.0]
```

```
[48.0, 12.0, 2.0, 0.0, 0.0, -48.0, -12.0, -2.0, -0.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 108.0, 18.0, 2.0, 0.0, 0.0, -108.0, -18.0, -2.0, -0.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 192.0, 24.0, 2.0, 0.0, 0.0, -192.0, -24.0, -2.0, -0.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 300.0, 30.0, 2.0, 0.0, 0.0, -300.0, -30.0, -2.0, -0.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 432.0, 36.0, 2.0, 0.0, 0.0, -432.0, -36.0, -2.0, -0.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 588.0, 42.0, 2.0, 0.0, 0.0, -588.0, -42.0, -2.0, -0.0, -0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 768.0, 48.0, 2.0, 0.0, 0.0, -768.0, -48.0, -2.0, -0.0, -0.0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 972.0, 54.0, 2.0, 0.0, 0.0, -972.0, -54.0, -2.0, -0.0, -0.0]
```

```
[12.0, 6.0, 2.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1200.0, 60.0, 2.0, 0.0, 0.0]
```

```
[24.0, 6.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 48.0, 6.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 72.0, 6.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 96.0, 6.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 120.0, 6.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 144.0, 6.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 168.0, 6.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 192.0, 6.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 216.0, 6.0, 0.0, 0.0, 0.0]
```

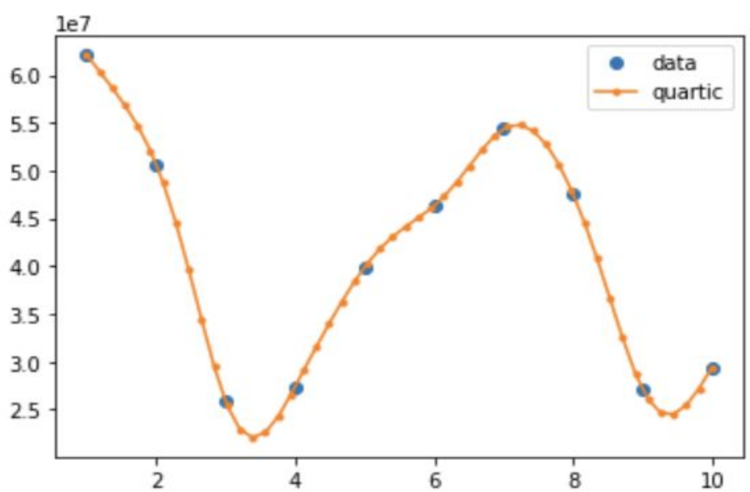
```
for z in range(2*(len(y)-2)):
  if derivdnum == y[-2]:
    groupnum = 0
    derivdnum = 1
    derivnum = 2
  for fqw in range(groupnum*5):
    row.append(0)
  b = derivnum+1
  if (b % 2)==0:
    b += 1
  row.append(float(diff(f1, x, derivnum).evalf(subs={x: y[derivdnum]})))
  row.append(float(diff(f2, x, derivnum).evalf(subs={x: y[derivdnum]})))
  row.append(float(diff(f3, x, derivnum).evalf(subs={x: y[derivdnum]})))
  row.append(float(diff(f4, x, derivnum).evalf(subs={x: y[derivdnum]})))
  row.append(float(diff(f5, x, derivnum).evalf(subs={x: y[derivdnum]})))
  row.append(-float((diff(f1, x, derivnum).evalf(subs={x: y[derivdnum]}))))
  row.append(-float((diff(f2, x, derivnum).evalf(subs={x: y[derivdnum]}))))
  row.append(-float((diff(f3, x, derivnum).evalf(subs={x: y[derivdnum]}))))
  row.append(-float((diff(f4, x, derivnum).evalf(subs={x: y[derivdnum]}))))
  row.append(-float((diff(f5, x, derivnum).evalf(subs={x: y[derivdnum]}))))
  for m in range(((len(y)-3)-groupnum)*5):
    row.append(0)
  temparray.append([])
  count += 1
  for i in range(len(row)):
    temparray[count-1].append(row[i])
  row.clear()
  derivdnum += 1
  counter += 1
  groupnum += 1
```

```
b = derivnum+1
if (b % 2)==0:
  b += 1
row.append(float(diff(f1, x, derivnum).evalf(subs={x: y[0]})))
row.append(float(diff(f2, x, derivnum).evalf(subs={x: y[0]})))
row.append(float(diff(f3, x, derivnum).evalf(subs={x: y[0]})))
row.append(float(diff(f4, x, derivnum).evalf(subs={x: y[0]})))
row.append(float(diff(f5, x, derivnum).evalf(subs={x: y[0]})))
for l in range((len(y)-2)*5):
  row.append(0)
temparray.append([])
count = count + 1
for i in range(len(row)):
  temparray[count-1].append(row[i])
row.clear()
counter += 1
for p in range((len(y)-2)*5):
  row.append(0)
row.append(float(diff(f1, x, derivnum).evalf(subs={x: y[len(y)-1]})))
row.append(float(diff(f2, x, derivnum).evalf(subs={x: y[len(y)-1]})))
row.append(float(diff(f3, x, derivnum).evalf(subs={x: y[len(y)-1]})))
row.append(float(diff(f4, x, derivnum).evalf(subs={x: y[len(y)-1]})))
row.append(float(diff(f5, x, derivnum).evalf(subs={x: y[len(y)-1]})))
# row.append(derivative(a1c, y[len(y)-1], 1.0,derivnum,args=(),order=b))
# row.append(derivative(b1c, y[len(y)-1], 1.0,derivnum,args=(),order=b))
# row.append(derivative(c1c, y[len(y)-1], 1.0,derivnum,args=(),order=b))
# row.append(derivative(d1c, y[len(y)-1], 1.0,derivnum,args=(),order=b))
# row.append(derivative(e1c, y[len(y)-1], 1.0,derivnum,args=(),order=b))
temparray.append([])
count = count + 1
for i in range(len(row)):
  temparray[count-1].append(row[i])
row.clear()
counter += 1
groupnum = 0
```

```
for s in range((len(y)-1)):
  b = derivnum+1
  if (b % 2)==0:
    b += 1
  if derivdnum == 0:
    for d in range(groupnum*5):
      row.append(0)
    row.append(float(diff(f1, x, derivnum).evalf(subs={x: y[0]})))
    row.append(float(diff(f2, x, derivnum).evalf(subs={x: y[0]})))
    row.append(float(diff(f3, x, derivnum).evalf(subs={x: y[0]})))
    row.append(float(diff(f4, x, derivnum).evalf(subs={x: y[0]})))
    row.append(float(diff(f5, x, derivnum).evalf(subs={x: y[0]})))
    for j in range(((len(y)-2)-groupnum)*5):
      row.append(0)
    groupnum += 1
  elif(groupnum == (len(y)-1)):
    break
  else:
    for d in range((groupnum)*5):
      row.append(0)
    row.append(float(diff(f1, x, derivnum).evalf(subs={x: y[derivdnum]})))
    row.append(float(diff(f2, x, derivnum).evalf(subs={x: y[derivdnum]})))
    row.append(float(diff(f3, x, derivnum).evalf(subs={x: y[derivdnum]})))
    row.append(float(diff(f4, x, derivnum).evalf(subs={x: y[derivdnum]})))
    row.append(float(diff(f5, x, derivnum).evalf(subs={x: y[derivdnum]})))
    groupnum += 1
    for j in range((((len(y)-1)-groupnum)*5)):
      row.append(0)
  temparray.append([])
  count += 1
  for i in range(len(row)):
    temparray[count-1].append(row[i])
  row.clear()
  derivdnum += 1
  counter += 1
```

Conclusion

Since the difference between the actual graph of all the data points against the quartic interpolation produces a smaller residual when compared to the difference between the commonly used interpolation methods including the linear interpolation (Linear AVG: 1.4 Normalized Error vs Quartic AVG: 0.72 Normalized Error), the quartic spline interpolation works the best at predicting the values. When given less nodes, this interpolation method generates a more accurate graph, which can lead to a faster run time, less allocated memory to a set of solution, and many other benefits.



=

Next Step - Research & Application

Research: Even though accuracy increases as a polynomial's order increases, time to complete the interpolation method also increases. Therefore, we are planning to conduct a time-accuracy analysis to determine what polynomial degree is the most efficient for interpolation.

Research: In order to make our interpolation more accurate, we can transform this from an R^2 to an R^3 vector field. This would make our interpolation into a bi-quartic piecewise spline, adding a new dimension to our interpolation and yielding higher a higher degree of accuracy. Although this would require more processing power, it has been shown to have success with a cubic spline, so a quartic spline will increase the accuracy of interpolation to a higher power.

Next Step - Research & Application

Application: This can be applied to image processing when changing the size and resolution of images. During surgery, small nanobots enter the bloodstream and provide a low resolution video feedback to the surgeons. By applying this interpolation method on the HSV, RGB, and many other factors within the video, it can help increase the resolution of the image by providing “new” pixels that are smooth and coherent, helping guide the surgeon as it increases their vision with a better resolution video feed during surgery.

References Cited

Administrator. "Lecture Course Material." *Composites and Coatings Group*, 17 Apr. 2015, https://www.ccg.msm.cam.ac.uk/initiatives/femor/lecture-course-material.

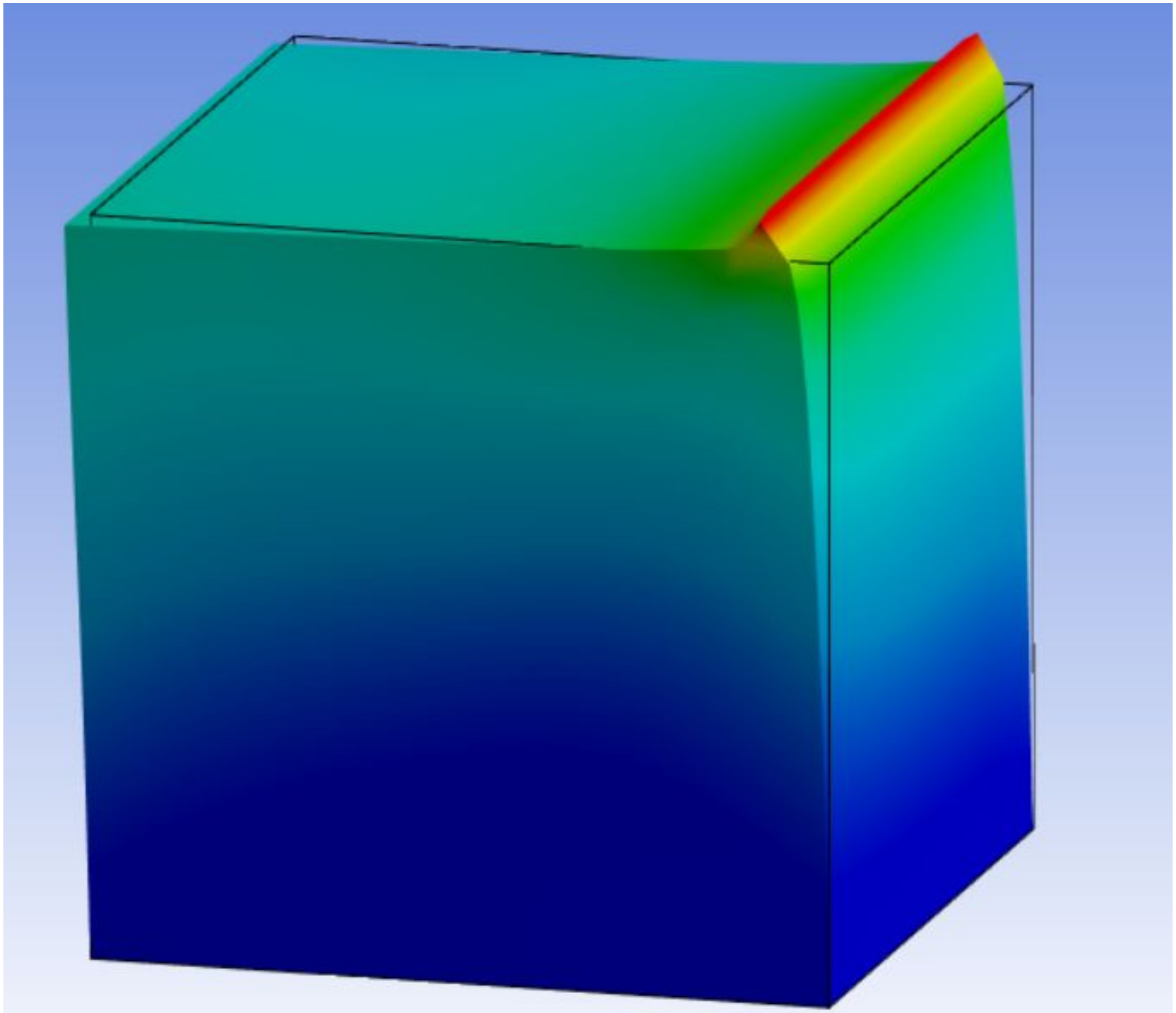
"Barycentric Coordinate System." *Wikipedia*, Wikimedia Foundation, 29 Jan. 2023, https://en.wikipedia.org/wiki/Barycentric_coordinate_system#Generalized_barycentric_coordinates.

Cubic Spline Interpolation — Python Numerical Methods. https://pythonnumericalmethods.berkeley.edu/notebooks/chapter17.03-Cubic-Spline-Interpolation.html.

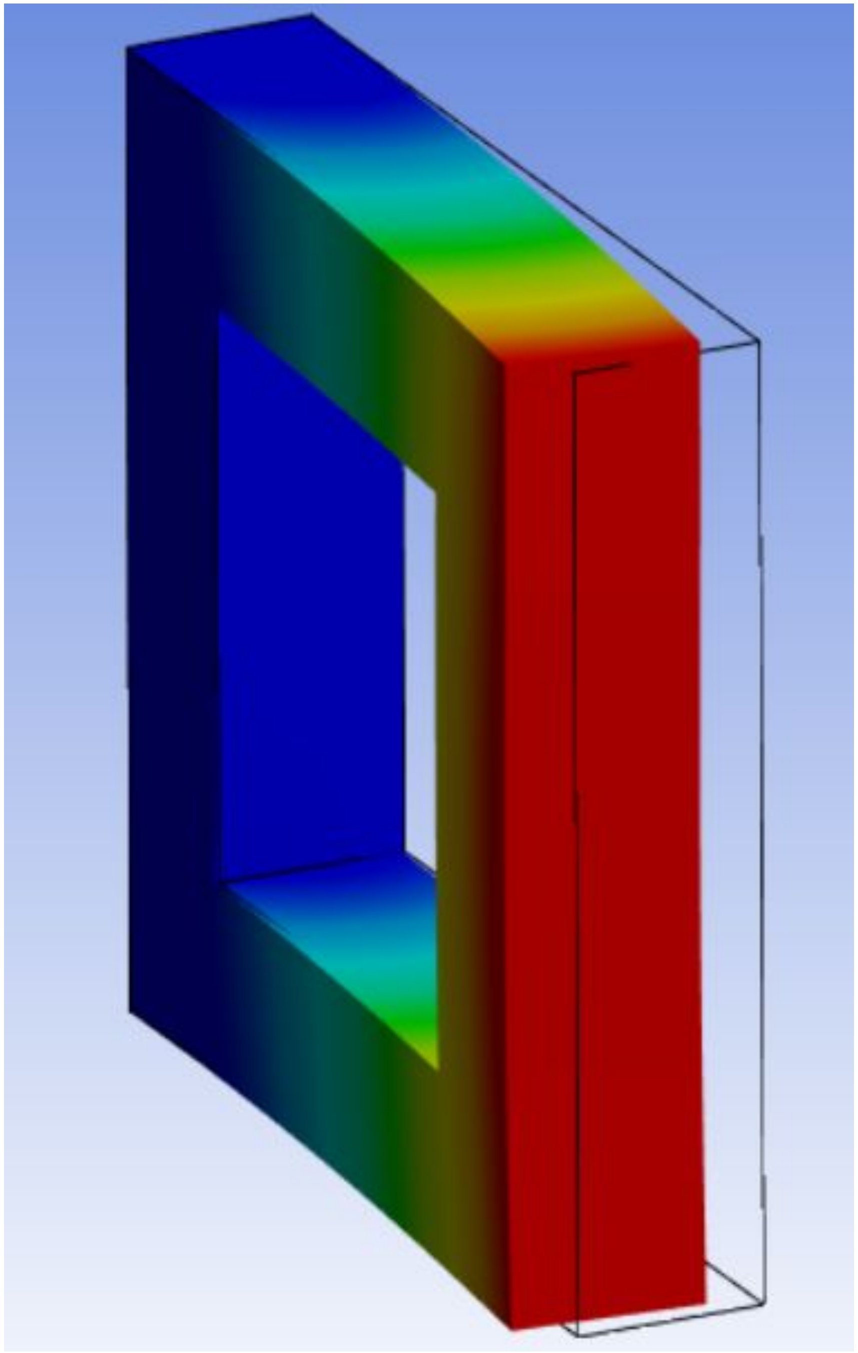
"How Spline Works." *How Spline Works-ArcGIS Pro | Documentation*, https://pro.arcgis.com/en/pro-app/latest/tool-reference/3d-analyst/how-spline-works.htm#:~:text=The%20Spline%20tool%20uses%20an,exactly%20through%20the%20input%20points.

Pizzarelli, Marco. "Dataset of the Experimentally Measured Heat Transfer in the Throat Region of Liquid Rocket Engine Thrust Chambers." *Data in Brief*, U.S. National Library of Medicine, 29 May 2021, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8187832/.

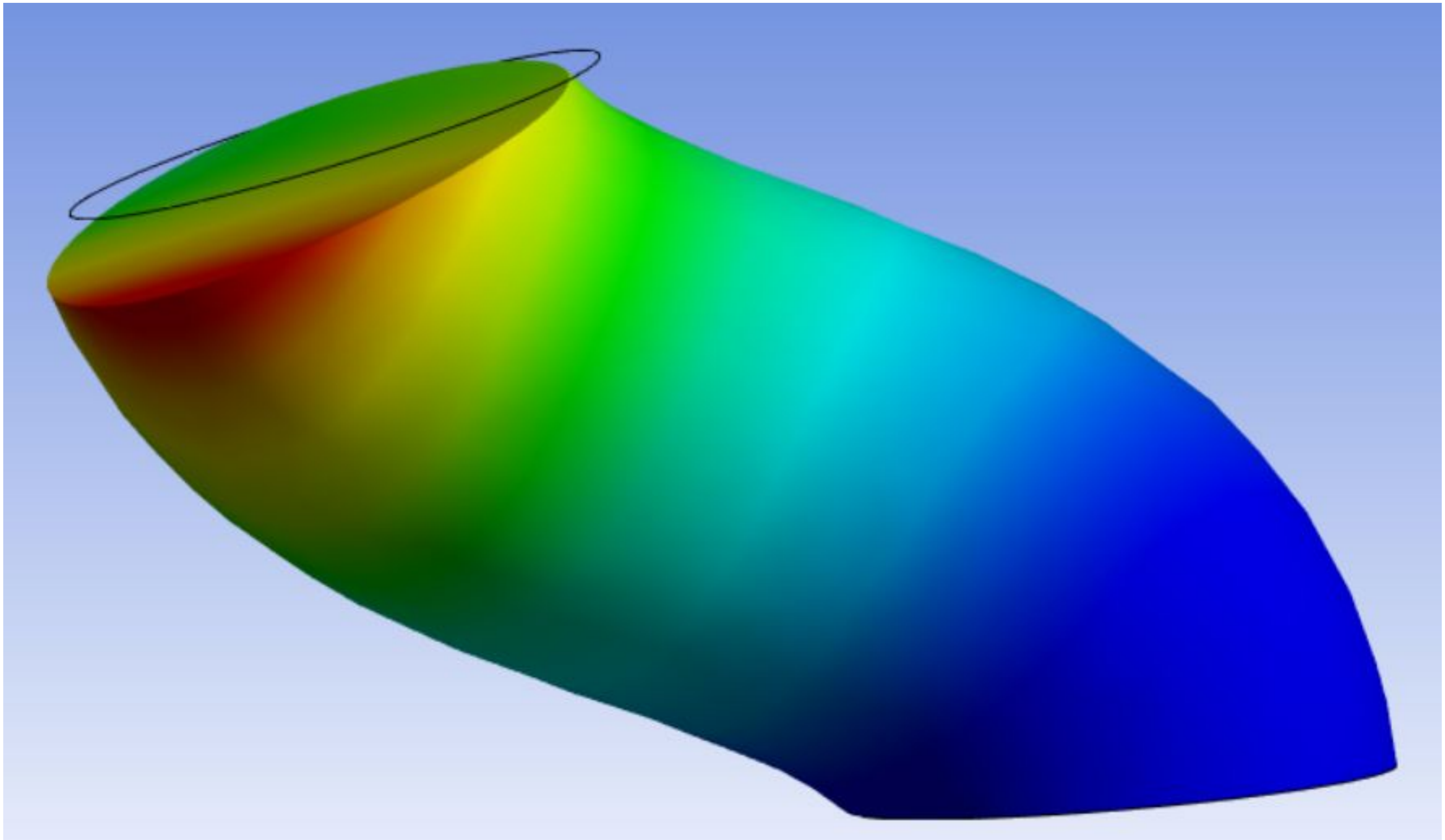
"Weighted Arithmetic Mean." *Wikipedia*, Wikimedia Foundation, 11 Jan. 2023, https://en.wikipedia.org/wiki/Weighted_arithmetic_mean.



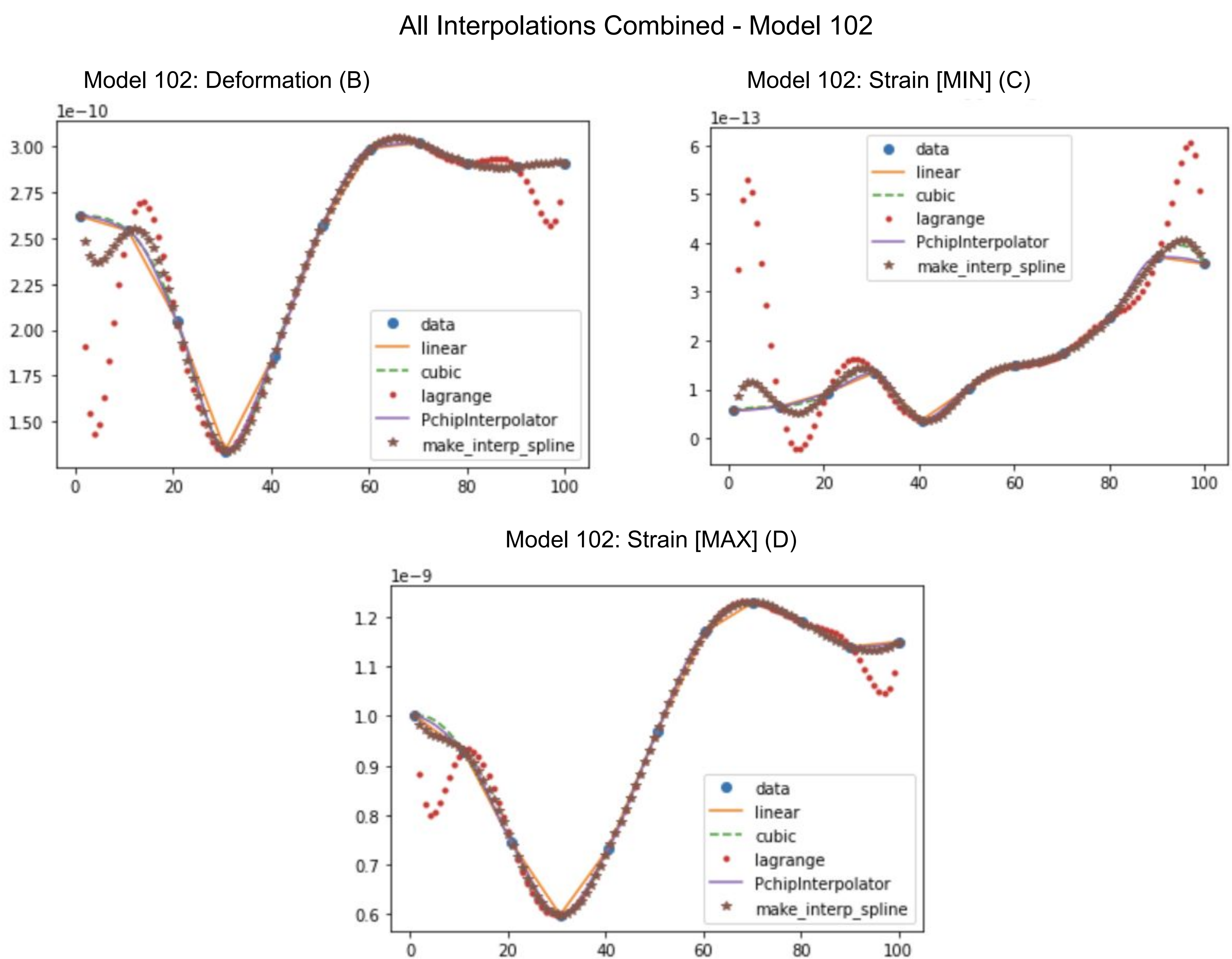
Model: 102
Multivariate Force Applied on a Cube



Model: 201
Torque Applied on a Frame

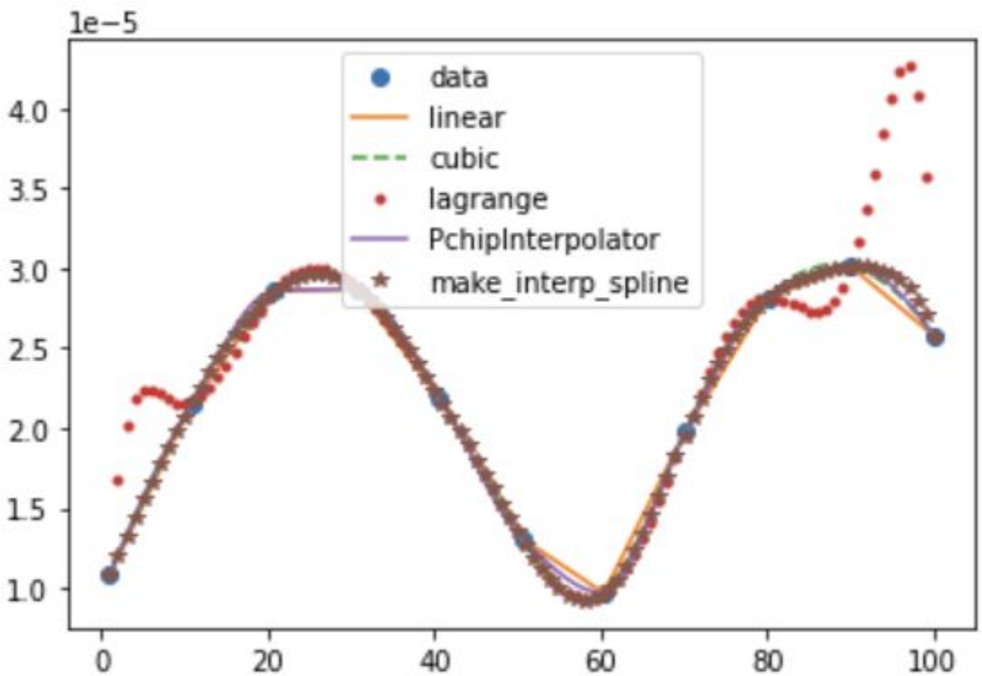


Model: 301
Multivariate Force Applied on a Pipe

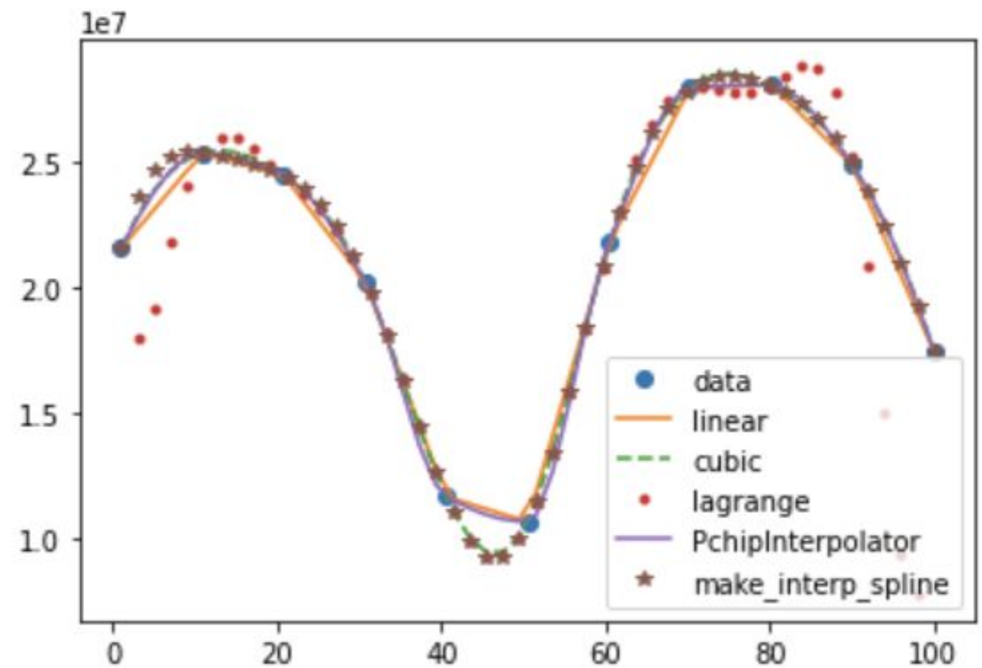

All Interpolations Combined - Model 102
Model 102: Deformation (B)
1e−10
3.00
2.75
2.50
2.25
2.00
1.75
1.50
0
20
40
60
80
100
data
linear
cubic
lagrange
PchipInterpolator
make_interp_spline
Model 102: Strain [MIN] (C)
1e−13
6
5
4
3
2
1
0
0
20
40
60
80
100
data
linear
cubic
lagrange
PchipInterpolator
make_interp_spline
Model 102: Strain [MAX] (D)
1e−9
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0
20
40
60
80
100
data
linear
cubic
lagrange
PchipInterpolator
make_interp_spline

All Interpolations Combined - Model 201

Model 201: Deformation (B)

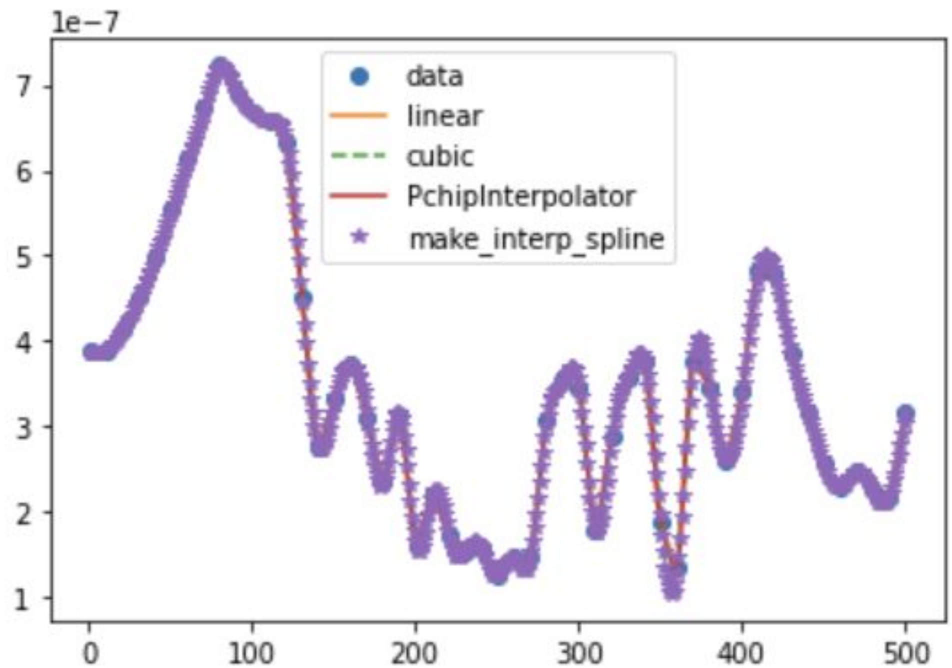


Model 201: Stress [Y] (I)

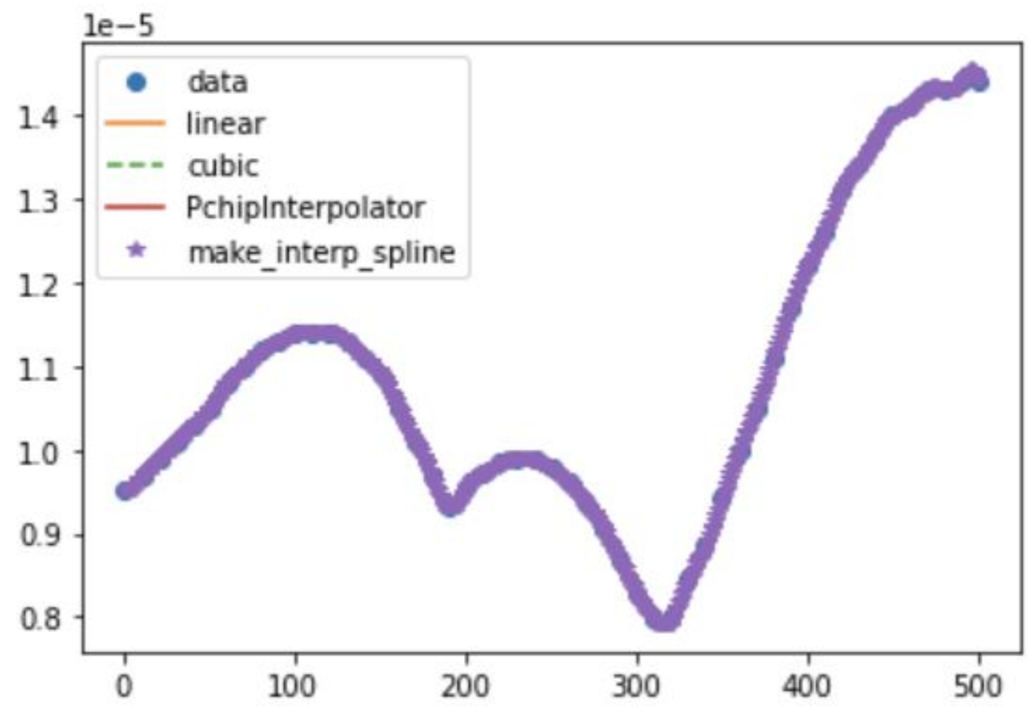


All Interpolations Combined - Model 301

Model 301: Strain [MIN] (C)

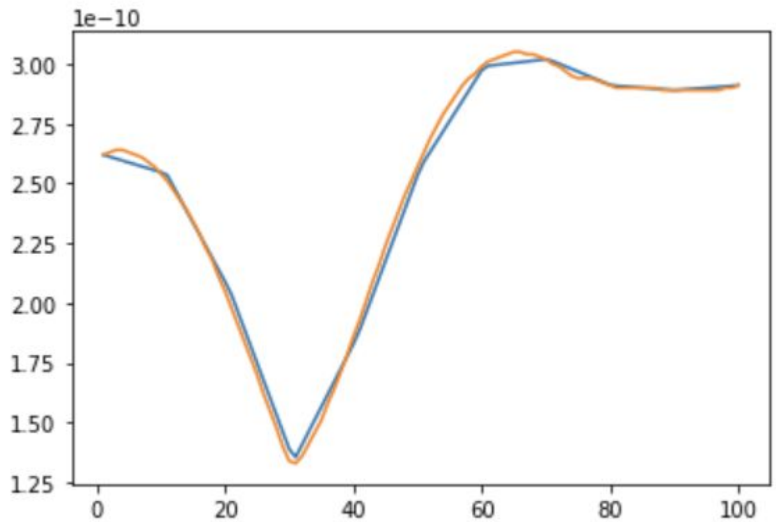


Model 301: Strain [MAX] (D)

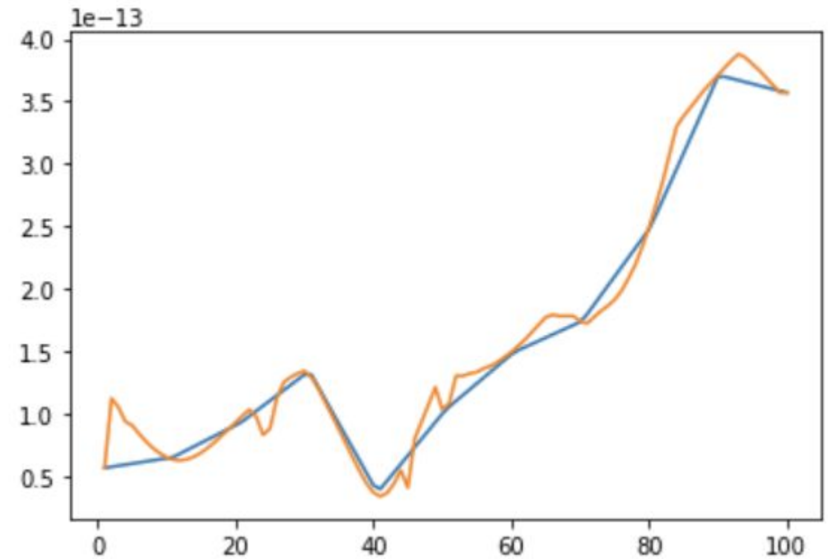


Linear Interpolation- Model 102

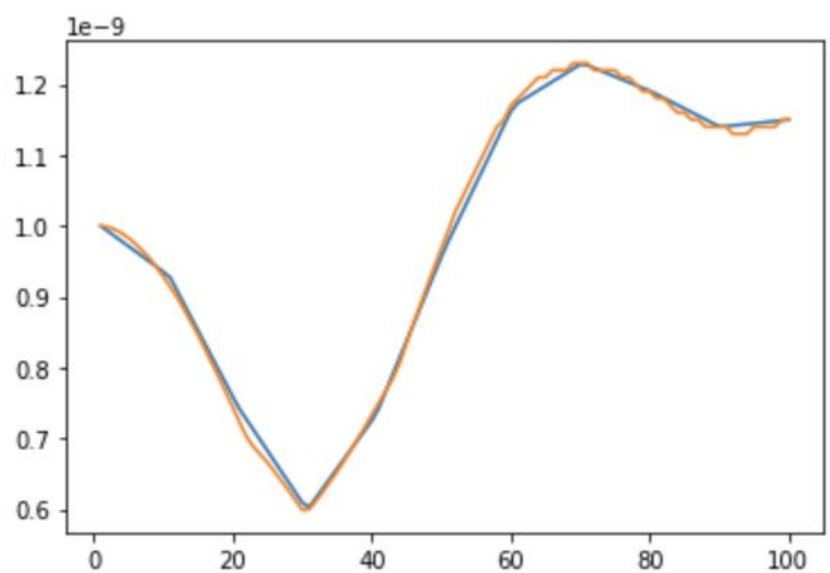
Model 102: Deformation (B)



Model 102: Strain [MIN] (C)

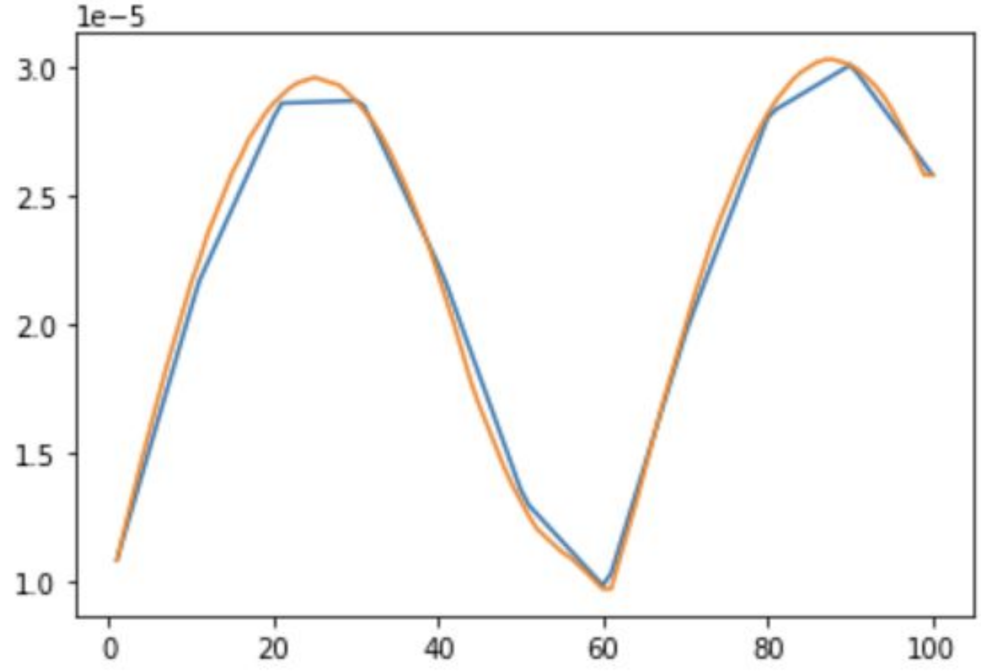


Model 102: Strain [MAX] (D)

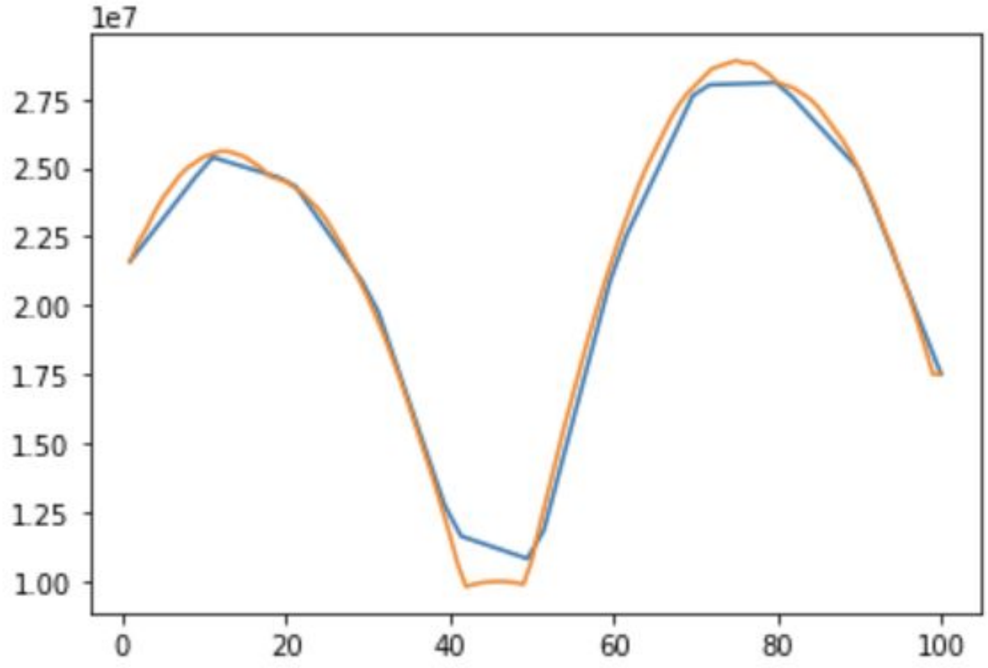


Linear Interpolations - Model 201

Model 201: Deformation (B)

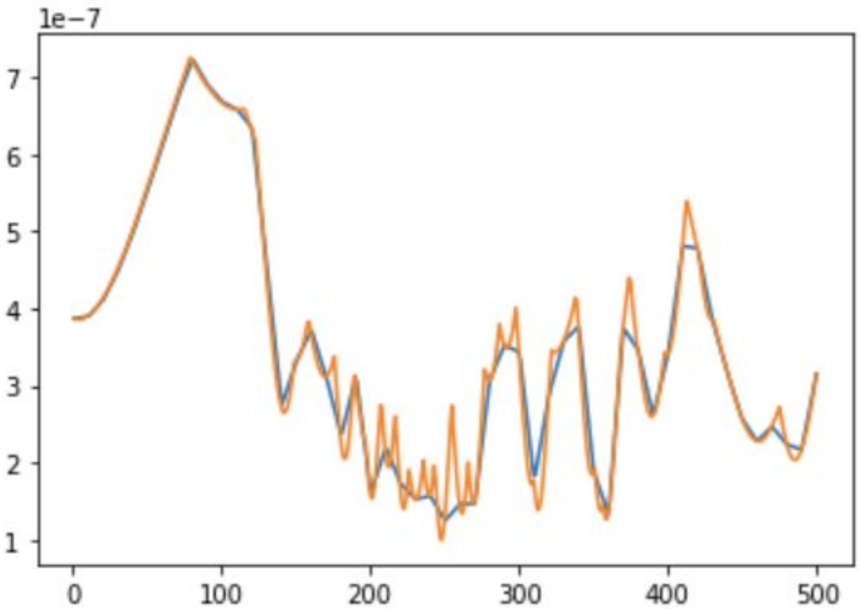


Model 201: Stress [Y] (I)

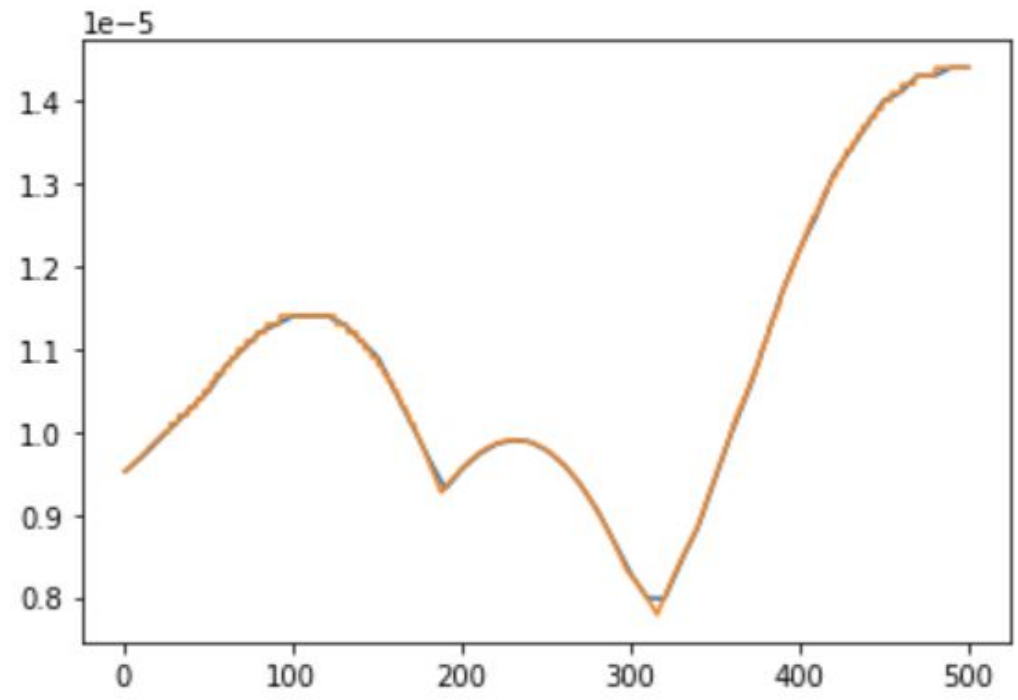


Linear Interpolations - Model 301

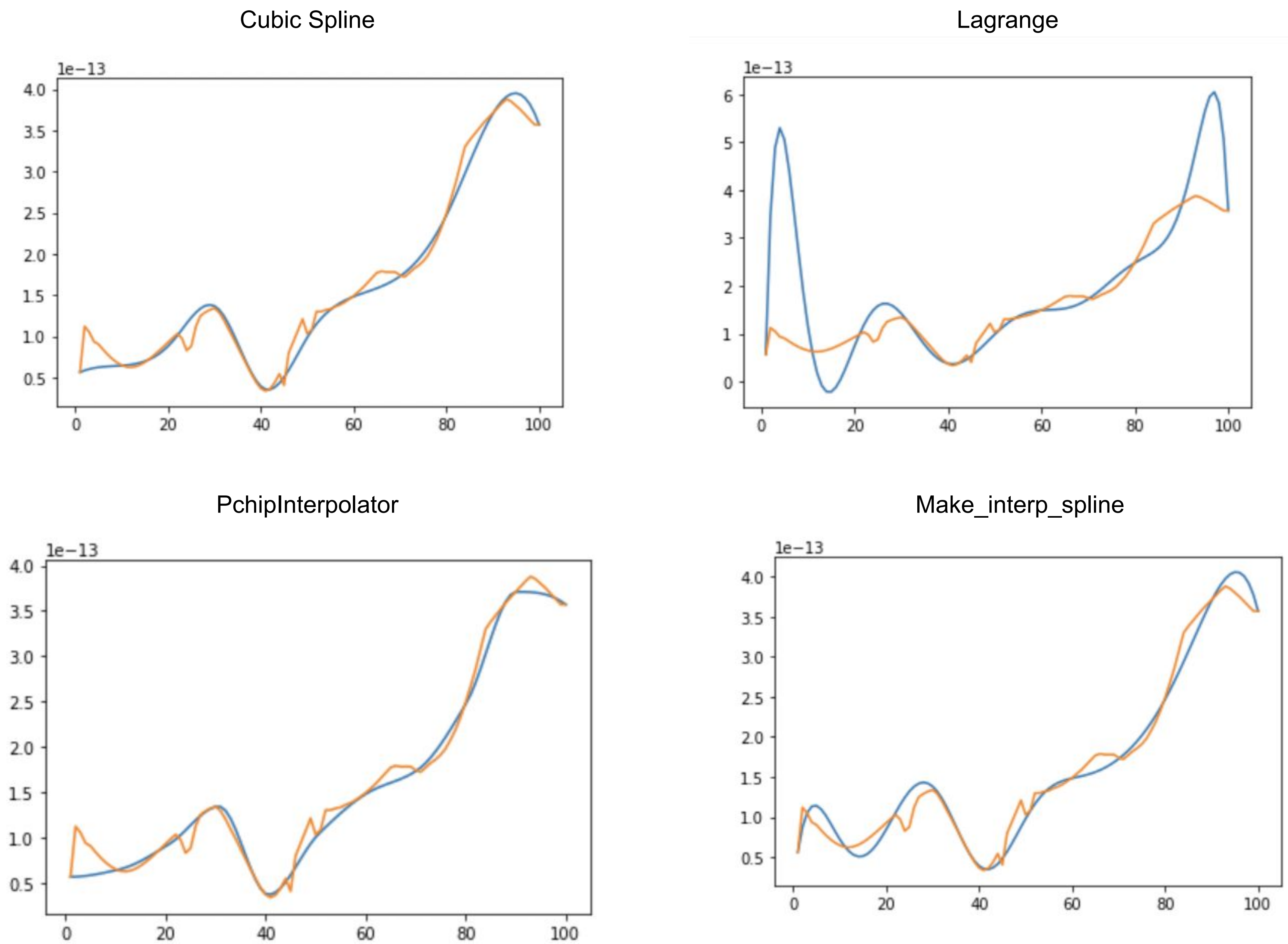
Model 301: Strain [MIN] (C)



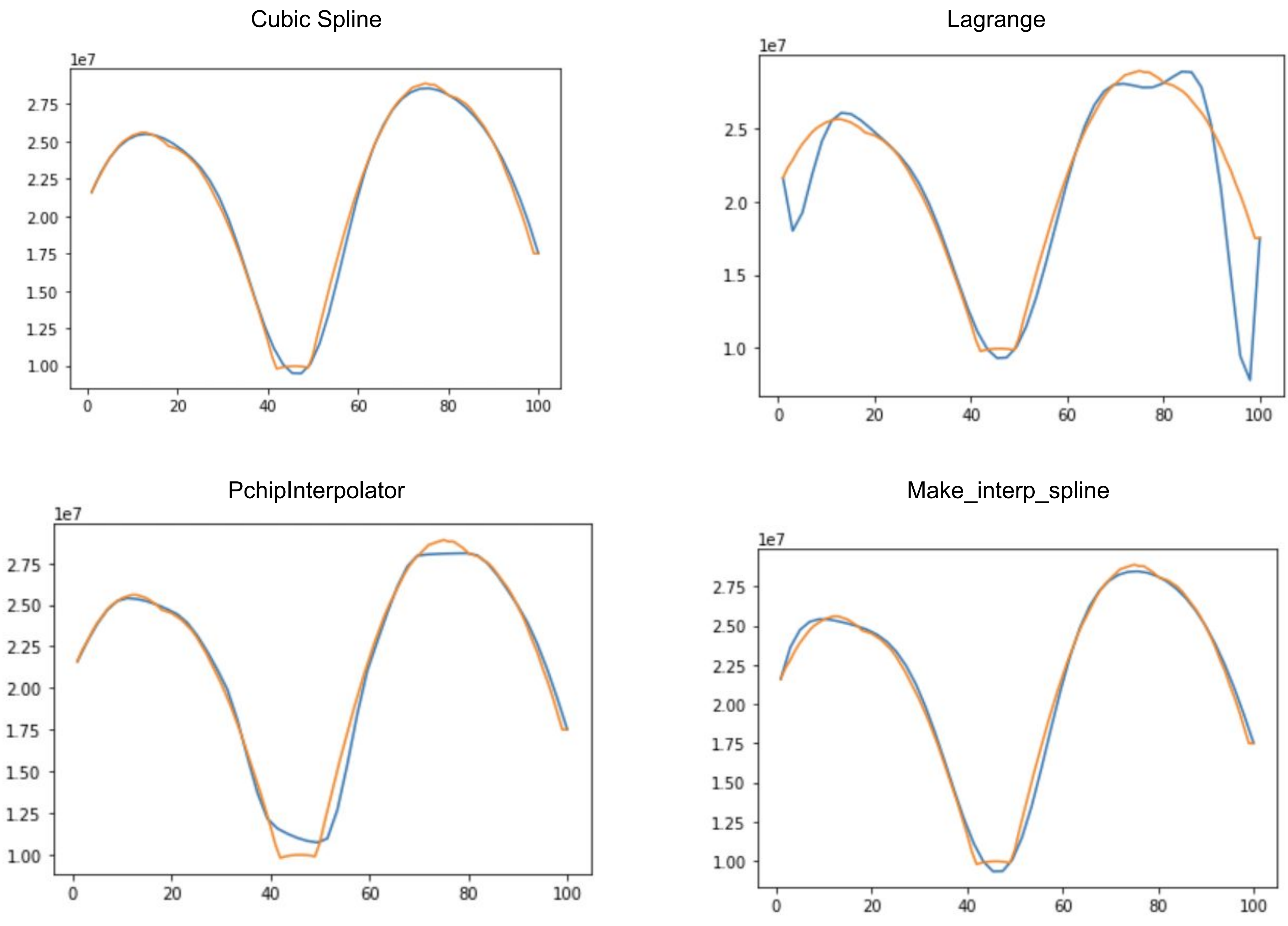
Model 301: Strain [MAX] (D)



Interpolations- Model 102Strain [MIN] (C)

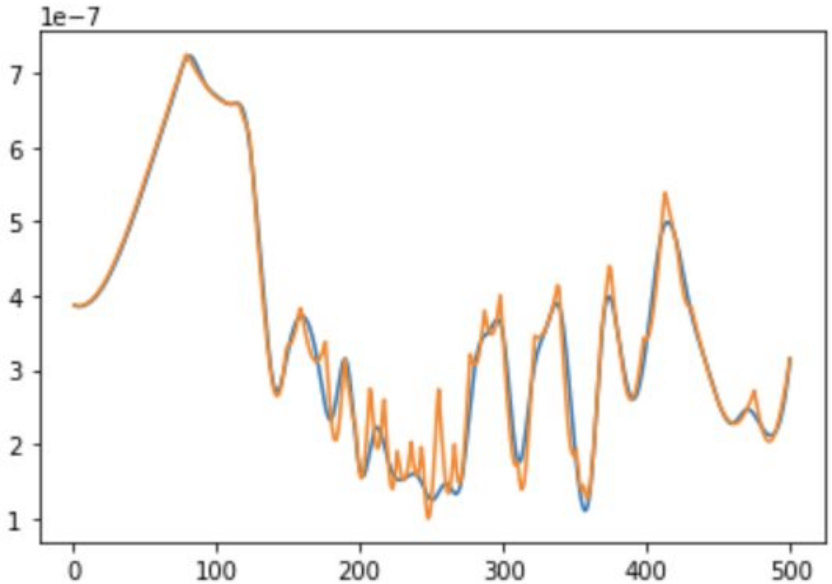


Interpolations- Model 201 Stress [Y] (I)

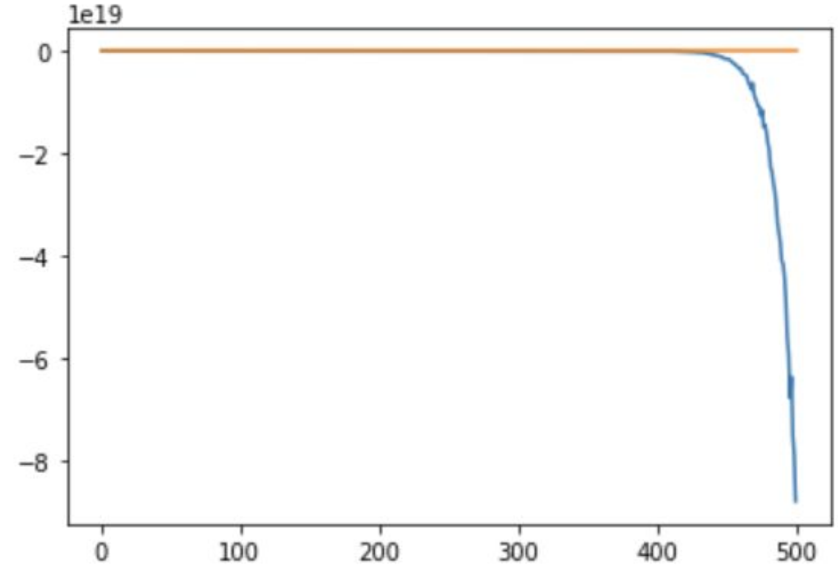


Interpolations- Model 301 Strain [MIN] (C)

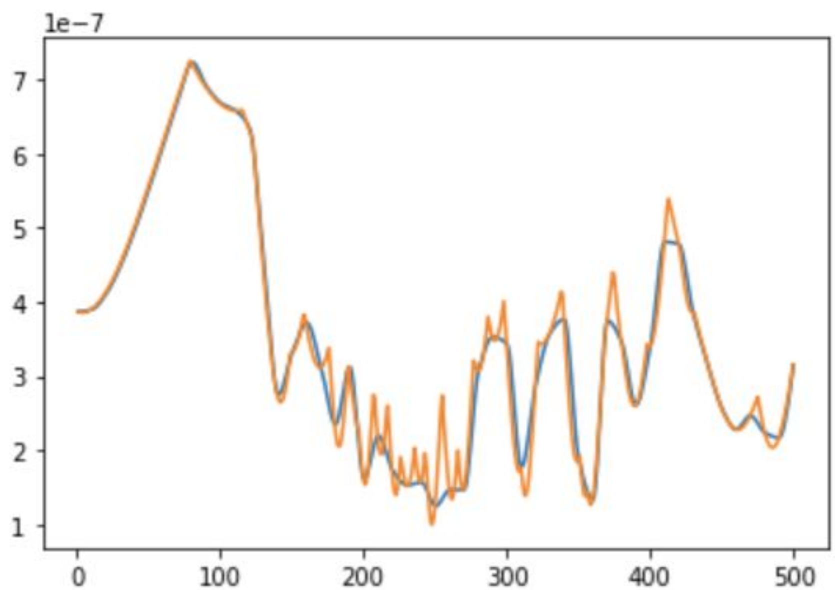
Cubic Spline



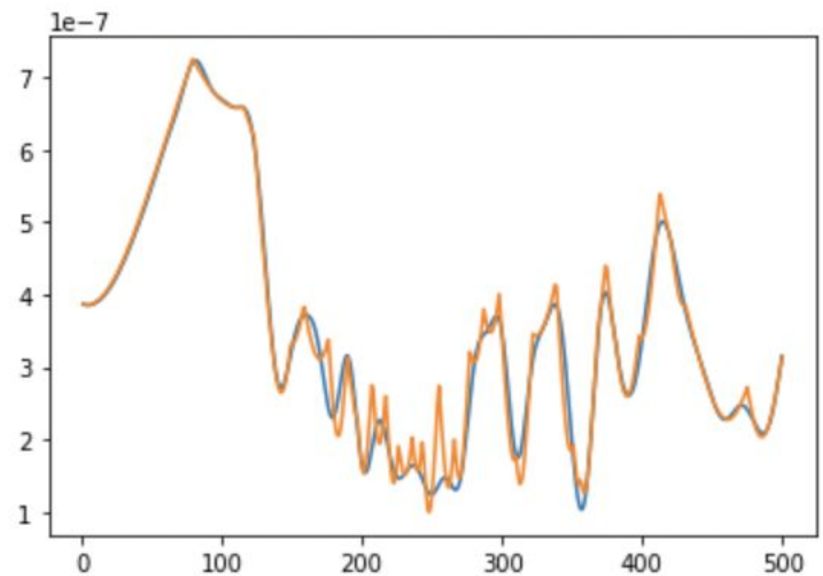
Lagrange



PchipInterpolator

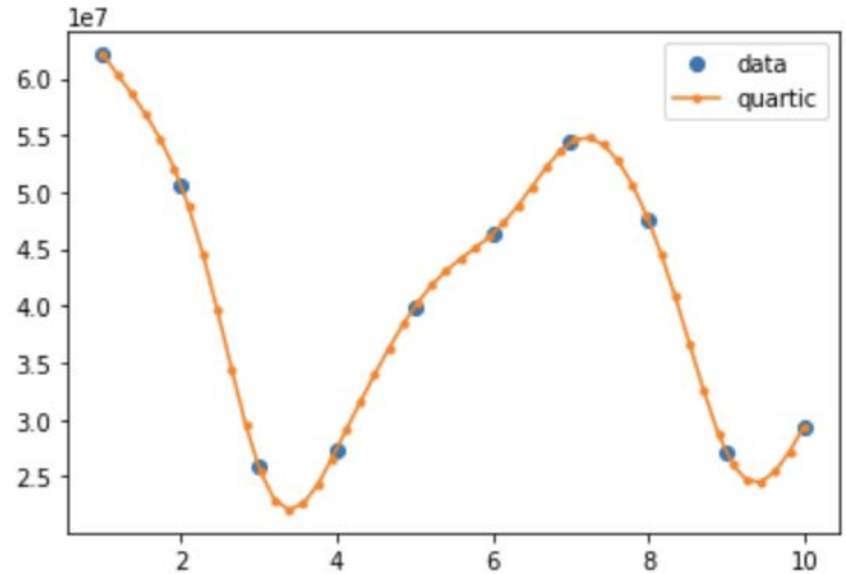


Make_interp_spline

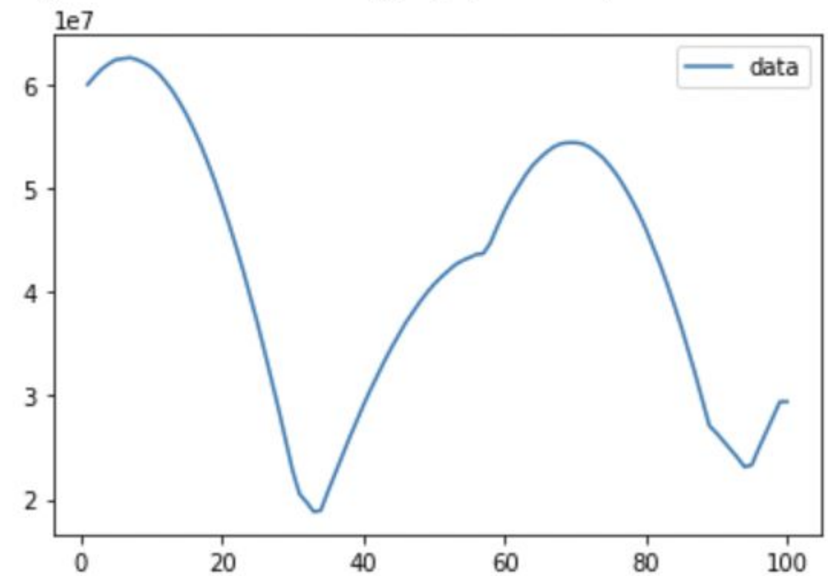


Quartic Spline

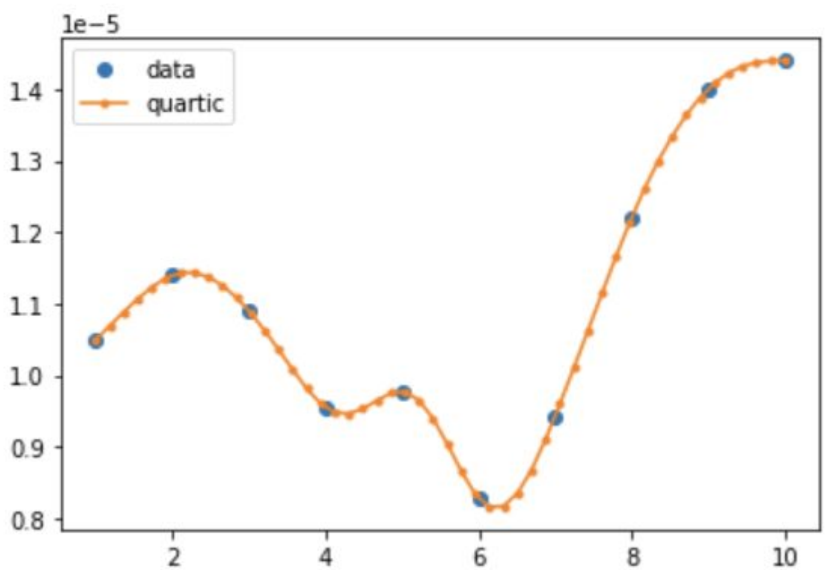
Model 201 - Quartic Spline Stress Probe (H)



Model 201 - Actual Graph (100 nodes)



Model 301 - Elastic Strain(D)



Model 302 - Actual Graph (500 nodes)