Interpolation and Polynomial Approximation

Abstract

This paper is about interpolation in general. We will discuss the different between each interpolation method. When to use one over another. What problem each interpolation method solves and of course, gain a better understanding of interpolation in general.

Introduction

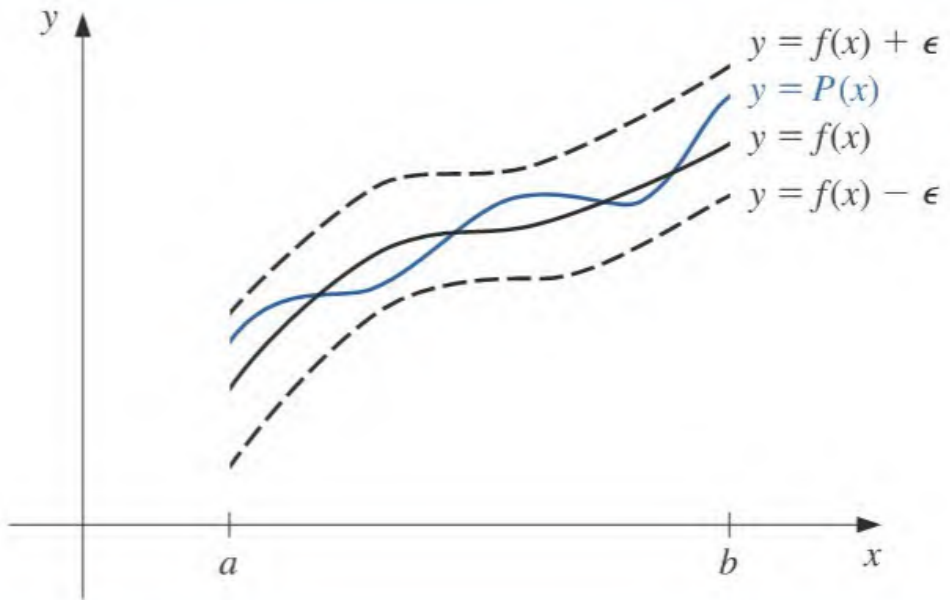
The most useful and well-known kind of functions mapping the set of real number onto itself is the polynomials. These polynomials uniformly approximate continuous functions. When given a function defined and continuous on a closed and bounded interval, there exist another polynomial that approximates the given function, or is considered to be "close" enough.

Weierstrass Approximation Theorem

Suppose that *f* is defined and continuous on the set closed set [a,b]. For any ε > 0, there exists a polynomial *P*(x), defined on a [a,b], with the property that

|*f*(x) - *P*(x)| < ε, for all *x* in [a,b].

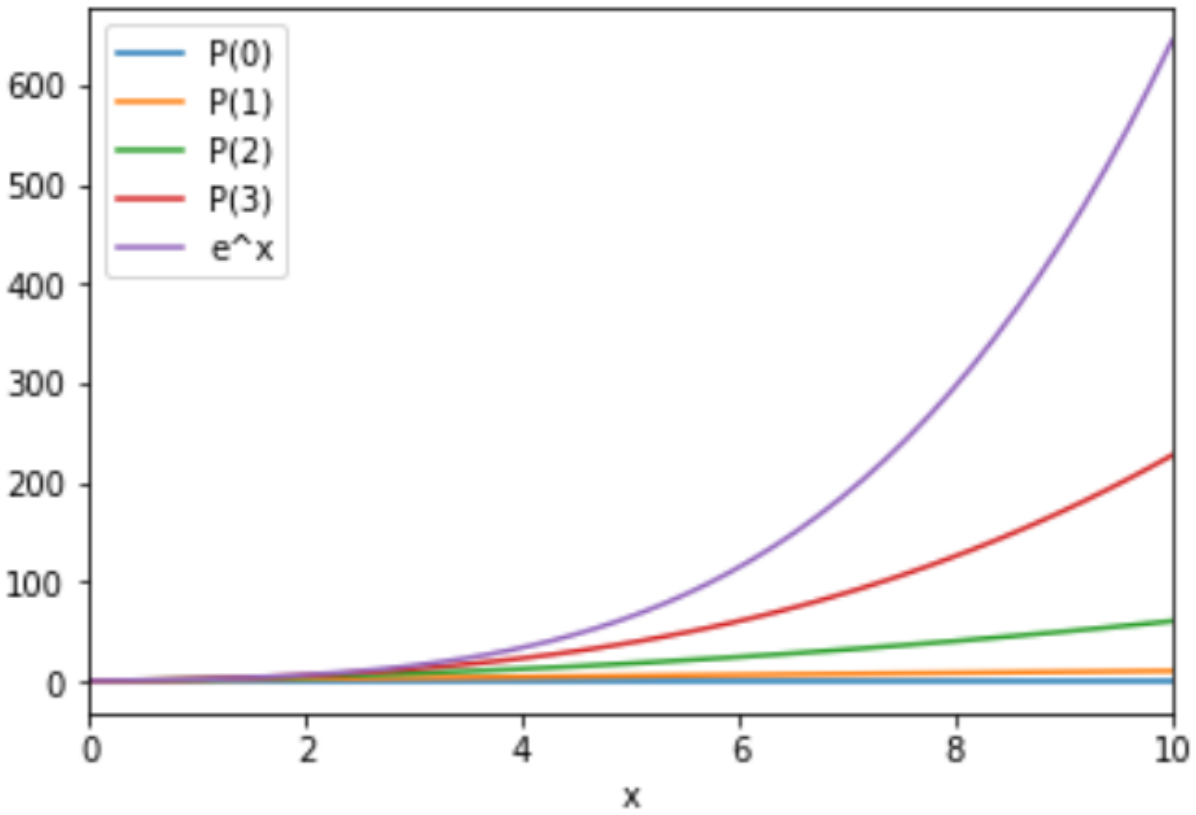
A simple explanation for the theorem above is that there exist a polynomial *P*(x) "close" to the given function *f* (x). This function *P*(x) is an approximation.



Taylor Series

Interpolation is a way of approximating the true underlying function *f*(x) with some interpolating function *P*(x), but Taylors expansion is also an approximation method. Why use Interpolation then? both sound similar. The main difference is that a Taylor expansion approximates a function around *a point*; in addition, they concentrate their accuracy near that point. This means that as you moved farer and farer from the specific point it becomes less accurate. A good approximating polynomial needs to be relatively accurate over the entire interval.

```
#parameter: x returns f(x)
def TaylorExp(x, n):
    y = 0
    for i in range(n):
        numerator = x**i
        denominator = math.factorial(i)
        y += numerator/denominator
    return y
```



The line label as $\mathbf{e^x}$ was given a degree size of **100** compare to the other lines in the graph, $\mathbf{e^x}$ is the more accurate. That's the point, better approximations are obtained for *f*(x) if higher degree Taylor polynomials are used, however, this isn't true for all functions if the polynomial is of a lower degree. In conclusion, Taylor series is used for the approximation at a single number but loses its accuracy the farer we move from that point.

Lagrange Interpolation

Lagrange Linear Interpolation

Lets say we take a sample of just two data points. The two points are given by the coordinates (x_0,y_0) and (x_1,y_1), the linear interpolation of these two points is a straight line between these points.

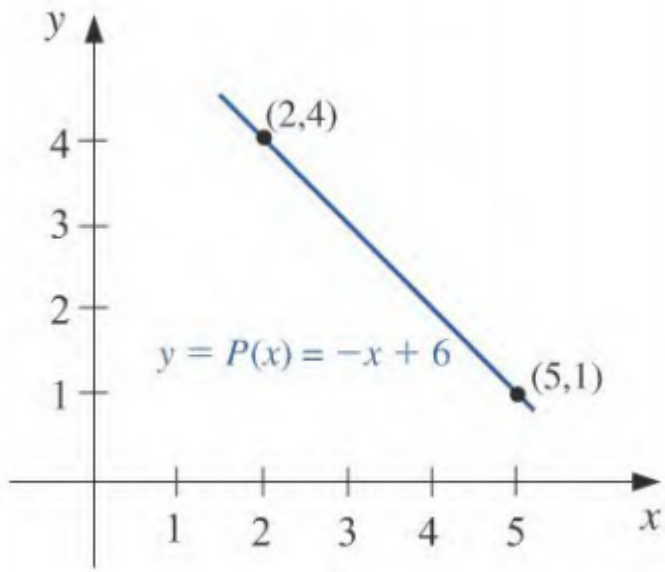
Define the functions

$$L_0(x) = \frac{x - x_1}{x_0 - x_1} \quad \text{and} \quad L_1(x) = \frac{x - x_0}{x_1 - x_0}.$$

The linear **Lagrange interpolating polynomial** through (x_0, y_0) and (x_1, y_1) is

$$P(x) = L_0(x)f(x_0) + L_1(x)f(x_1) = \frac{x - x_1}{x_0 - x_1}f(x_0) + \frac{x - x_0}{x_1 - x_0}f(x_1).$$

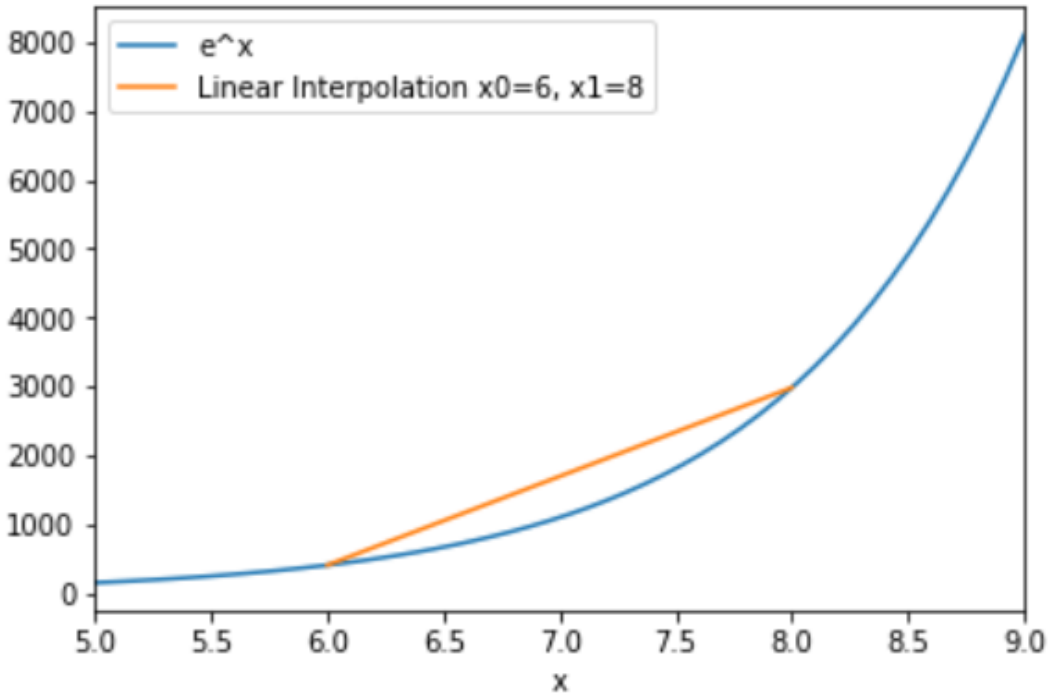
Linear interpolation is one of the more simpler methods, but obviously isn't very accurate.



Let's use the function $\mathbf{e^x}$ as an example. Lets pretend we only have two data points (6, 403.4287) and (8, 2980.9579). We can use linear interpolation to approximate the *f*(x) where x =7.

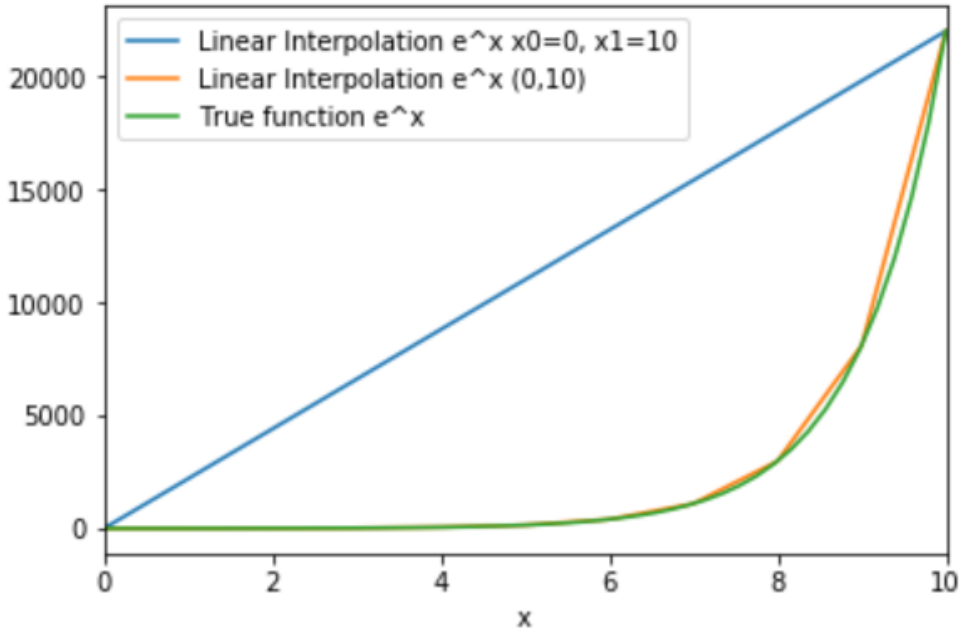
```
def linearInterpolation(x, x0, x1,function):
    lagrange0 = (x-x1)/(x0-x1)
    lagrange1 = (x-x0)/(x1-x0)
    return lagrange0*function(x0) + lagrange1*function(x1)
```

	Approximation e^7	Absolute Error	Relative Error
Linear Interpolation e^x	1692.19339	595.560232	0.543081



In the graph above the blue line represents the true function $\mathbf{e^x}$. Using only two data points we denote the function we want to interpolate by g. We would approximate the function *f*(x) by *g*(x). Where *g*(x) lies between the function *f*(a) and *f*(b). The absolute error and relative error is high when we which to approximate *f*(7) using *g*(7), however; what it lacks in accuracy it makes up for in simplicity. Lets see what happens if the two data points we have are far come each other.

	e^7 Approximation	Absolute Error	Relative Error
Approximation e^7 x0=0, x1=10	15418.826056	14322.192898	13.060149
Approximation e^7 x0=6, x1=8	1692.193390	595.560232	0.543081



The graph above shows a linear interpolation where (0,*f*(0)) and (10,*f*(10)) are known for the blue line. As you can imagine two data points that are far from each other won't produce accurate results with linear interpolation. For the best accuracy possible, it helpful to have two data points relativity close to one another. Looking at the orange line it looks like if we took more and more samples we'll get a more accurate line. The orange line uses the x values from the set of real number {0,1,2,3,4,5,6,7,8,9,10}, where the value y = *f*(x) for each x.

Lagrange Interpolation Polynomial

If $x_0, x_1, \ldots, x_n$ are $n+1$ distinct numbers and f is a function whose values are given at these numbers, then a unique polynomial $P(x)$ of degree at most n exists with

$$f(x_k) = P(x_k), \quad \text{for each } k = 0, 1, \ldots, n.$$

This polynomial is given by

$$P(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x) = \sum_{k=0}^{n} f(x_k)L_{n,k}(x), \tag{3.1}$$

where, for each $k = 0, 1, \ldots, n$,

$$L_{n,k}(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)} \tag{3.2}$$

$$= \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{(x-x_i)}{(x_k-x_i)}.$$

■

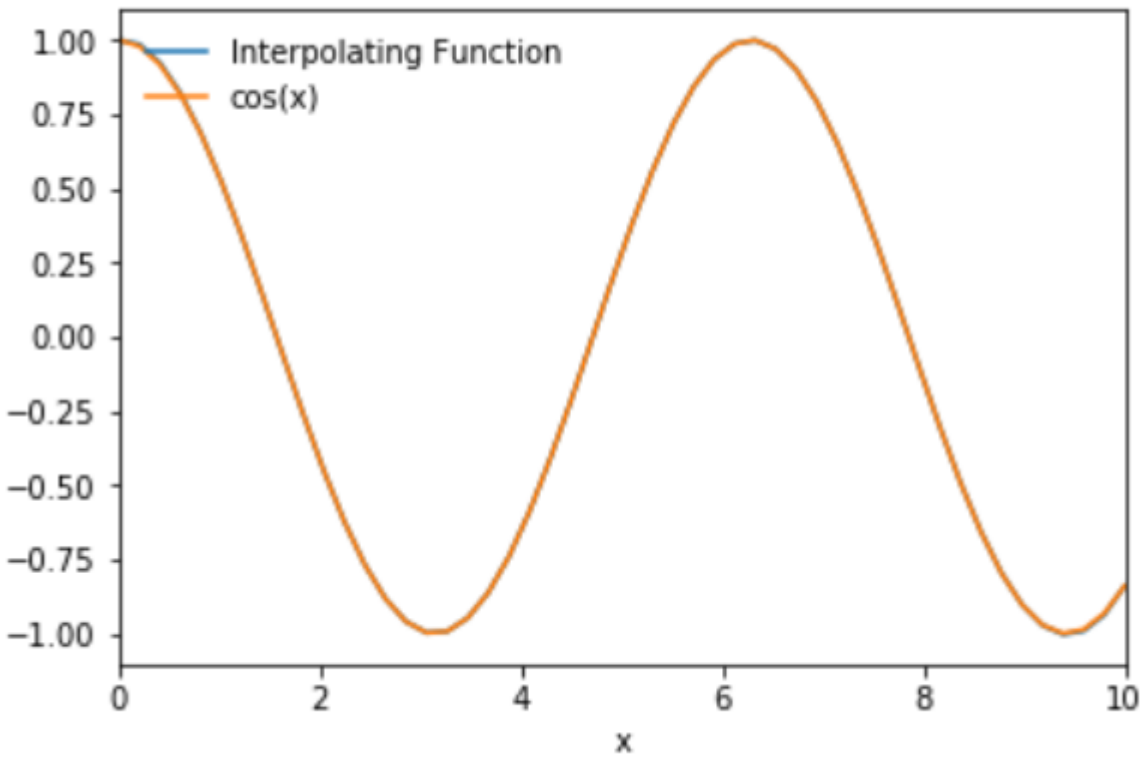
We are generalizing the concept of linear interpolation. Which brings me back to my previous point, the more samples taken from the data the more accurate the approximation. Here is the python implementation of Lagrange Interpolation Polynomial.

```
def lagrangeInterpolatingPolynomial(x, dataPoints, function):
    approximatedValue = 0
    for i in dataPoints:
        lagrangeValue = 1
        for j in dataPoints:
            if (i != j):
                numerator = (x - j)
                denominator = (i - j)
                lagrangeValue *= (numerator/denominator)
        approximatedValue += (function(i)*lagrangeValue)
    return approximatedValue
```

Lets take a look at the function cos(x), for this example the known data points are for each element x in the set {0,1,2,3,4,5,6,7,8,9,10} where y = *f*(x). I decided to approximate each value over a specified interval which is (0,10).

```
dataPoints = [0,1,2,3,4,5,6,7,8,9,10]
x = linspace(0,10)
y = getAllLagrangeInterpolatingPolynomial(x, dataPoints, math.cos)
```

Here are the results.



Based on the graph alone, the approximation of the function cos(x) seems pretty close.

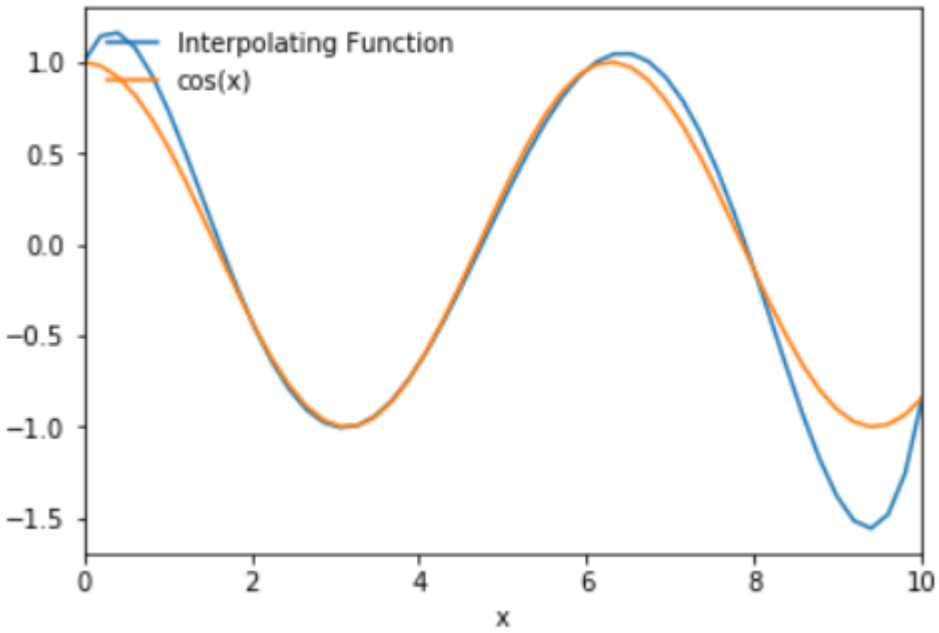
	cos(x) Approximation	Absolute Error	Relative Error
x = 0.0	1.000000	0.000000	0.000000
x = 0.20408163265306123	0.985640	0.006393	0.006528
x = 0.40816326530612246	0.924067	0.006215	0.006772
x = 0.6122448979591837	0.822225	0.003865	0.004723
x = 0.8163265306122449	0.686387	0.001485	0.002168
x = 1.0204081632653061	0.522899	0.000119	0.000228
x = 1.2244897959183674	0.338575	0.000850	0.002506
x = 1.4285714285714286	0.140819	0.000927	0.006539
x = 1.6326530612244898	-0.062463	0.000646	-0.010451
x = 1.836734693877551	-0.263079	0.000264	-0.001004
x = 2.0408163265306123	-0.452851	0.000053	-0.000118
x = 2.2448979591836737	-0.623965	0.000231	-0.000370
x = 2.4489795918367347	-0.769315	0.000265	-0.000344
x = 2.6530612244897958	-0.882827	0.000196	-0.000222
x = 2.857142857142857	-0.959736	0.000081	-0.000084
x = 3.0612244897959187	-0.996803	0.000030	-0.000030
x = 3.2653061224489797	-0.992459	0.000102	-0.000103
x = 3.4693877551020407	-0.946876	0.000121	-0.000128
x = 3.673469387755102	-0.861950	0.000093	-0.000108
x = 3.8775510204081636	-0.741225	0.000037	-0.000050
x = 4.081632653061225	-0.589732	0.000023	-0.000039
x = 4.285714285714286	-0.413779	0.000067	-0.000163
x = 4.4897959183673475	-0.220678	0.000082	-0.000371
x = 4.6938775510204085	-0.018445	0.000065	-0.003510
x = 4.8979591836734695	0.184531	0.000024	0.000133
x = 5.1020408163265305	0.379842	0.000024	0.000064
x = 5.3061224489795915	0.559395	0.000064	0.000114
x = 5.510204081632653	0.715752	0.000080	0.000111

The full chart is available in the source code provided, based on the chart the error is kept to minimal thanks to the many samples taken. Now lets see what happens if we take remove some samples. To see how the interpolating function behaves, change the array called data points.

```
dataPoints = [0,1,2,3,4,5,6,7,8,9,10]
```

Lets try taking out all the odd x values.

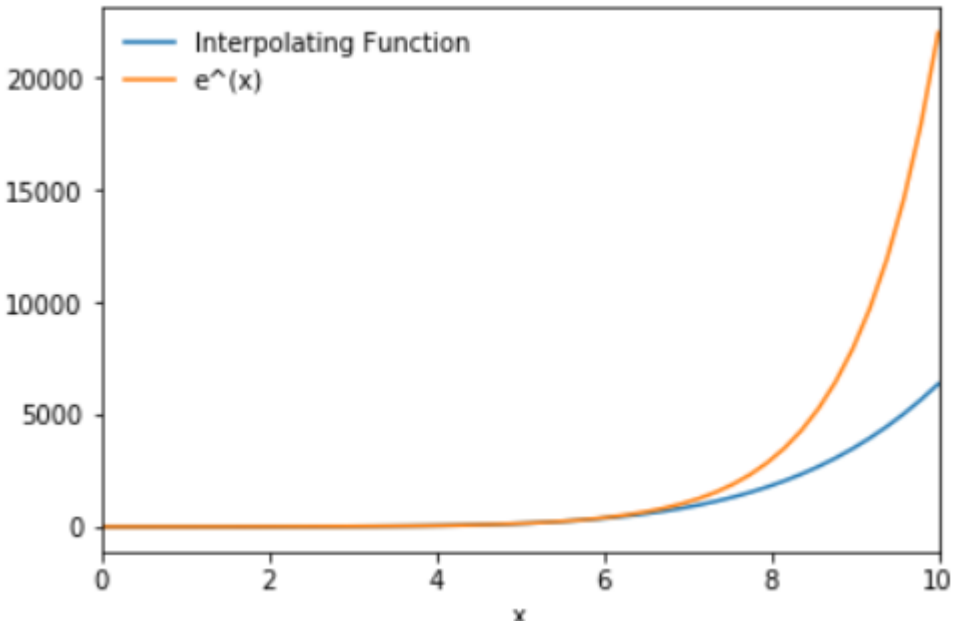
```
dataPoints = [0,2,4,6,8,10]
x = linspace(0,10)
y = getAllLagrangeInterpolatingPolynomial(x, dataPoints, math.cos)
```



Based on the graph above when even x values are known the approximation is quite close; however, when odd the interpolating function loses accuracy. It seems that between the interval of [8,10] there's seems to be a huge gap. Another observation about the interpolating function I found was that if there are no known data points ahead. Based on the slopes current trajectory, the interpolating function will continue on that trajectory on an increasing or decreasing rate depending on the slope of the previously known data point.

I decided to try to approximate the function e^x with it's positive slope, to see if my suspensions about the interpolating function's slope was correct.

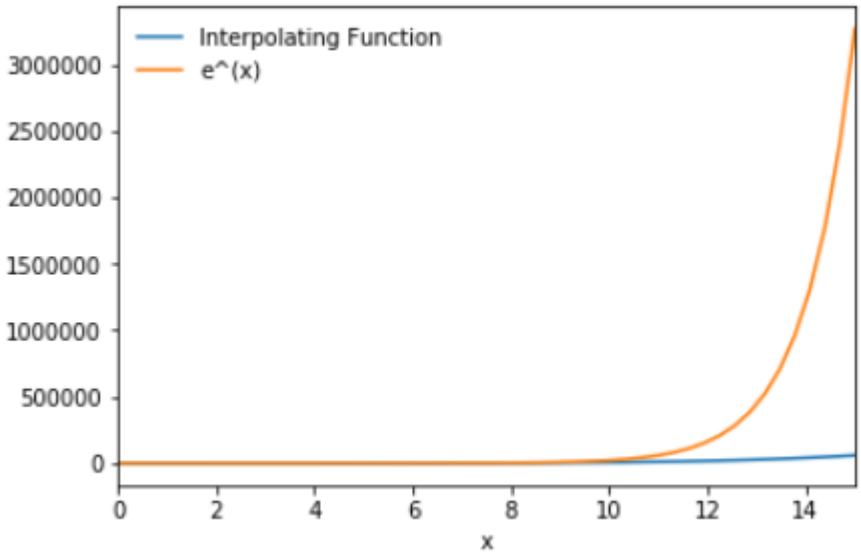
```
dataPoints = [0,1,2,3,4,5]
x = linspace(0,10)
y = getAllLagrangeInterpolatingPolynomial(x, dataPoints, exp)
```



Everything pass the x value 5 is unknown, and the interpolating function does it's best to approximate the function e^x. At x=5 and $f(5)$, the slope is slowly increasing and as we go pass 5 the rate of the slope is positively increasing; based on that, It seems that the last known true slope determines the slope of future interpolating function.

x = 4.285714285714286	72.873708	0.219284	0.003018
x = 4.4897959183673475	89.493698	0.390438	0.004382
x = 4.6938775510204085	109.744231	0.468148	0.004284
x = 4.8979591836734695	134.293985	0.277986	0.002074
x = 5.1020408163265305	163.896750	0.460237	0.002800
x = 5.3061224489795915	199.396739	2.170385	0.010768
x = 5.510204081632653	241.733884	5.467687	0.022118
x = 5.714285714285714	291.949141	11.218436	0.037004
x = 5.918367346938775	351.189795	20.614396	0.055444
x = 6.122448979591837	420.714758	35.265256	0.077339
x = 6.326530612244898	501.899874	57.313223	0.102489
x = 6.530612244897959	596.243224	89.574748	0.130610
x = 6.73469387755102	705.370424	135.715546	0.161358
x = 6.938775510204081	831.039930	200.466442	0.194343
x = 7.142857142857143	975.148342	289.889282	0.229155
x = 7.346938775510204	1139.735704	411.704241	0.265369
x = 7.551020408163265	1326.990809	575.692446	0.302569
x = 7.755102040816327	1539.256498	794.190941	0.340351
x = 7.959183673469388	1779.034968	1082.700920	0.378337
x = 8.16326530612245	2048.993069	1460.634854	0.416179
x = 8.36734693877551	2351.967612	1952.233978	0.453565
x = 8.571428571428571	2690.970668	2587.694689	0.490218
x = 8.775510204081632	3069.194869	3404.551158	0.525901
x = 8.979591836734695	3490.018717	4449.372169	0.560417
x = 9.183673469387756	3957.011882	5779.843315	0.593605
x = 9.387755102040817	4473.940503	7467.321802	0.625338
x = 9.591836734693878	5044.772496	9599.970859	0.655523
x = 9.795918367346939	5673.682853	12286.604967	0.684098
x = 10.0	6365.058945	15661.406850	0.711027

As you can see the absolute error grows rapidly when x > 5 and is approaching 10. If we try to approximate higher x values, it'll be a pretty useless approximation.



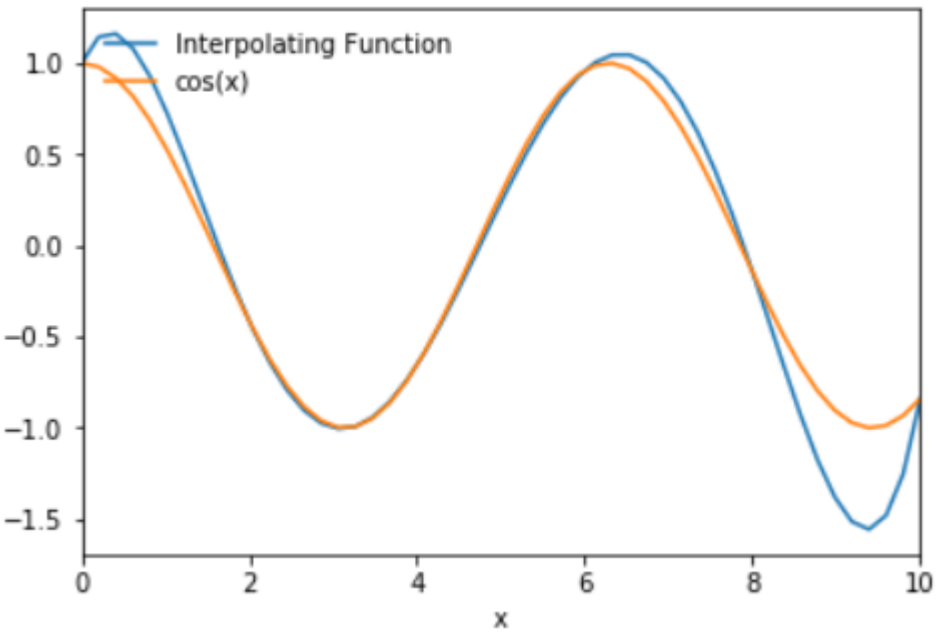
For fun, I decided to see what the interpolating function will do pass the x when x = 10.

In conclusions, the Lagrange Interpolation Polynomial is useful if you have a big enough sample. It does a poor job if the sample size is small. It can't predict the future. The last known data point will determine if the slope increases or decreases, but that is it. The true slope could be doing the exact opposite of the what the interpolation function approximated.

Lagrange Polynomial Approximation Error

$$\frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)^{n+1}.$$

Looking at the error term above, it looks like the error is minimize if the value you are approximating is close to one of the known data points. This is evident based on what is seen when approximating cosine with Lagrange polynomial.



As you can see when x is moving away from 8 the approximation error grows, but as x approaches 10 the error is minimized. So when Lagrange polynomial approximation its best to take a large sample of data. That way you can minimized the error throughout the whole interval.

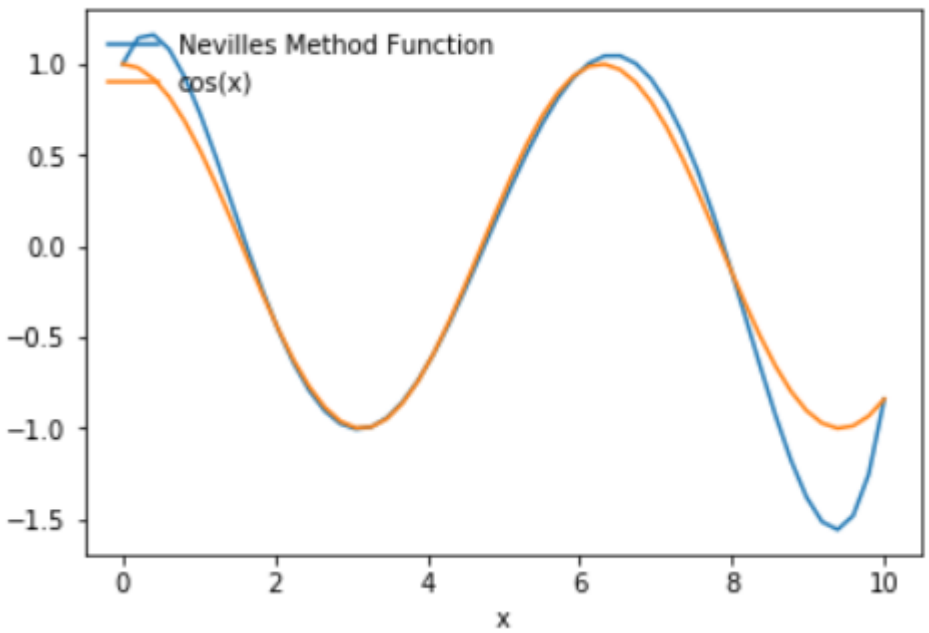
Neville's Method

Previously with Lagrange polynomial approximating, it would interpolate the data. In that case an explicit representation isn't needed, just the values of the polynomial at specified points. This is where Neville's Method comes in. Neville's method uses a programming technique known as dynamic programming. Before Lagrange polynomial would calculate the first approximation and then the second approximation. However, calculating the first approximation doesn't lessen the work for calculating the second approximation.

x_0	P_0				
x_1	P_1	$P_{0,1}$			
x_2	P_2	$P_{1,2}$	$P_{0,1,2}$		
x_3	P_3	$P_{2,3}$	$P_{1,2,3}$	$P_{0,1,2,3}$	
x_4	P_4	$P_{3,4}$	$P_{2,3,4}$	$P_{1,2,3,4}$	$P_{0,1,2,3,4}$

The chart above shows Neville's Methods matrix. The first column is the known data points. The next columns will perform the calculations using the previous columns. The next columns will use the previous columns to lessen the work for the current column.

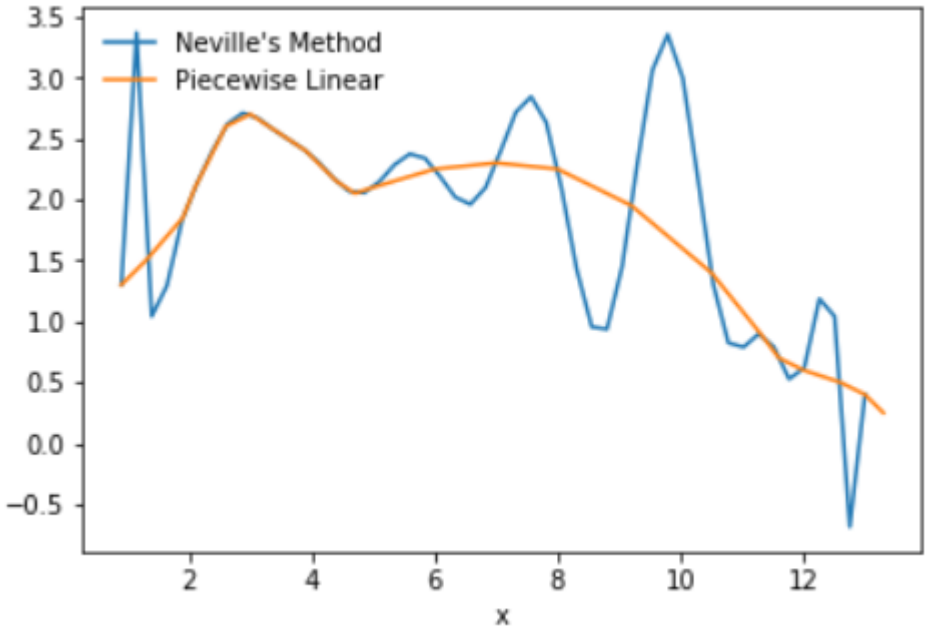
```
def nevillesMethod(xValues, yValues,x):
    sizeN = len(xValues)
    q = np.zeros((sizeN,sizeN))
    for i in range(len(yValues)):
        q[i][0] = yValues[i]

    #Neville's method...
    for i in range(1,sizeN):
        for j in range(1, i+1):
            numerator = ((x - xValues[i - j])*(q[i][j - 1]) - (x - xValues[i])*(q[i - 1][j
- 1]))
            demoninator = (xValues[i] - xValues[i - j])
            q[i][j] =  numerator/ demoninator
    return q[sizeN-1][sizeN-1]
```



Neville's method gives us the same result as Lagrange polynomial, but with the added help of dynamic programming. The disadvantage is that, Neville's method only uses half the matrix, thus there is wasted memory.

Cubic Spline Interpolation

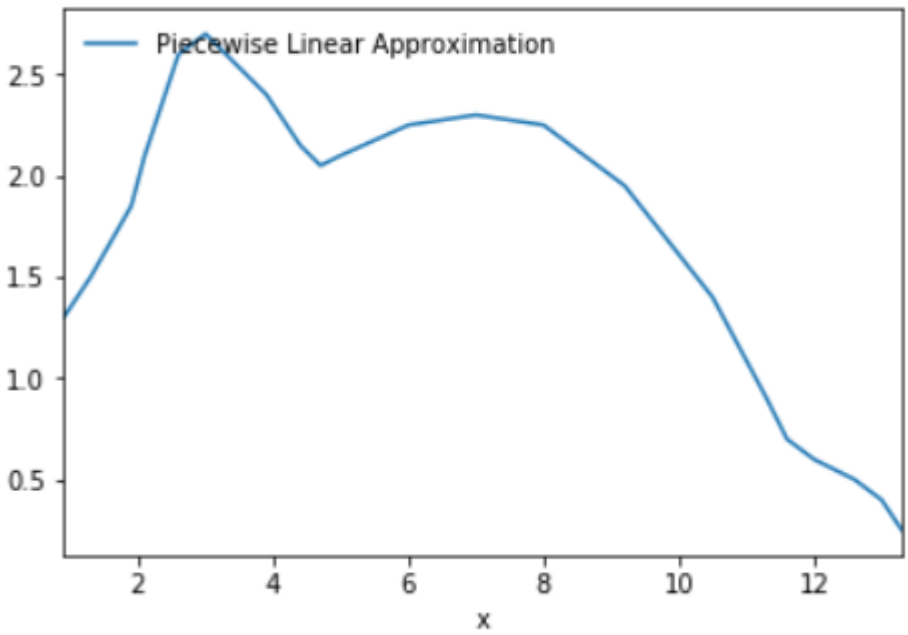
Throughout my project, I kept running into these wild oscillations when interpolating data. It turns out that high-degree polynomials can oscillate erratically; minor fluctuation over a small portion of the interval can cause large fluctuations over the entire range.



How do we fix this?

Piecewise-Polynomial Approximation

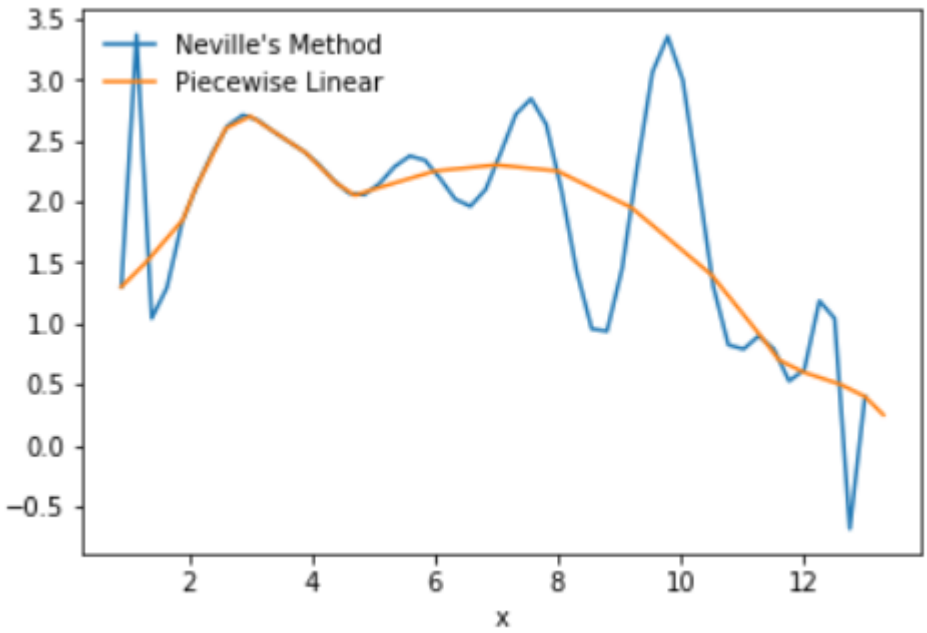
The simplest Interpolation is the linear interpolation method. Which is the linear interpolation between two points; however, what about linear interpolation between multiple points. The piecewise- linear interpolation consists of joining a set of data points {(x0,f(x0)), (x1,f(x1))...(xn-1,f(xn-1))} by a series of straight lines.



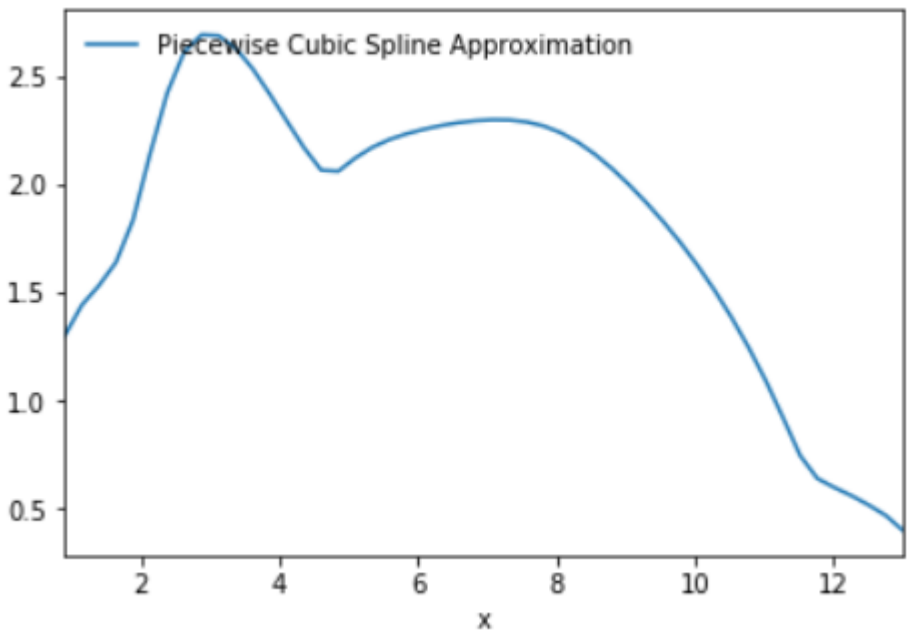
The draw back to the piecewise polynomial approximation is that there is no differentiability at the end points of each subinterval. So when representing the interpolating function graphically, the function isn't "smooth". Smoothness in a interpolating function is a requirement, meaning that the function approximating function MUST be continuously differentiable.

Cubic Spline

Another piecewise polynomial approximation is the cubic spline. Its somewhat similar to linear piecewise approximation in that you use sub intervals. Cubic spline interpolation uses cubic polynomials between pair of data points. Generally the cubic spline ensure that the interpolant is continuously differentiable on the interval; in addition, has a continuous second derivative. What does that mean? To be continuously differentiable is when taking the first derivative limit from both sides for a specific point will lead to the same limit. Taking the second derivative limit from both sides for a specific point will lead to the same result i.e local maximum or local minimum. That way the end points will give us that smooth curve we have been looking for.

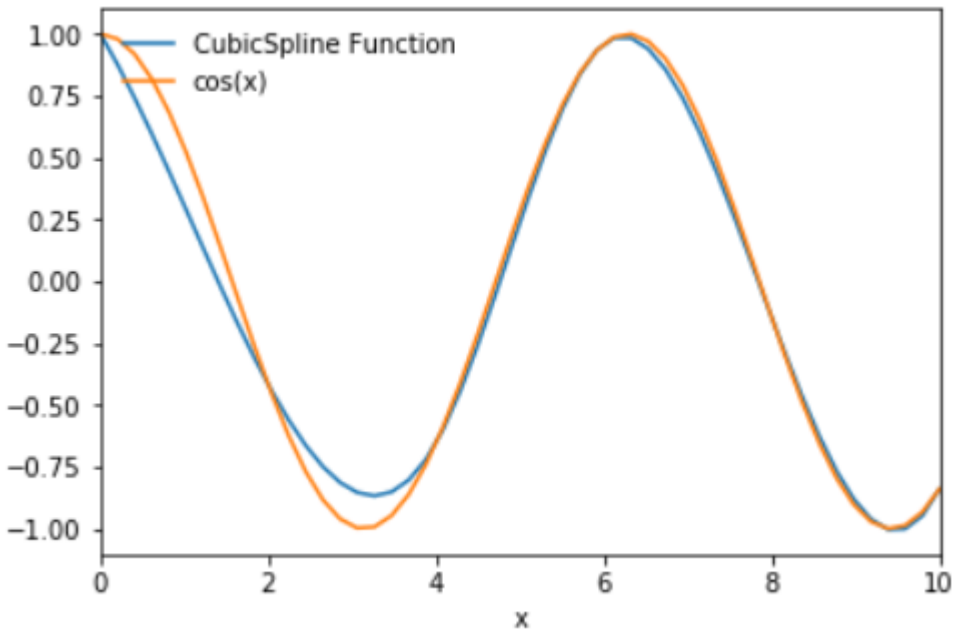


As you can see when using Neville's method it osculates wildly. Piecewise linear approximation doesn't give us that smooth curve we want.

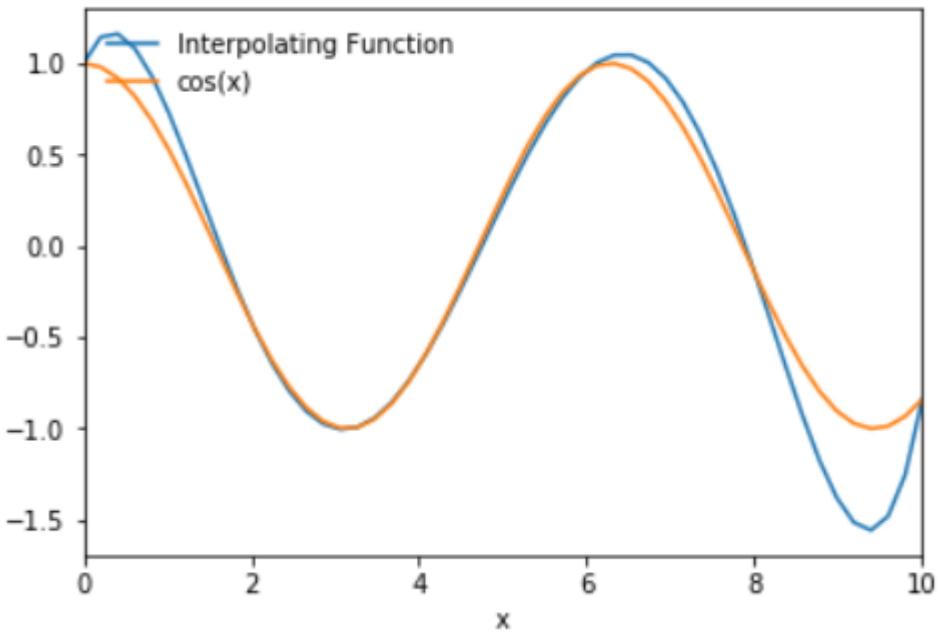


Cubic Spline Interpolation

When applying cubic spline we get a much nicer more smooth graphical representation.



Cubic Spline Interpolation.



Langrage Polynomial Interpolation.

When applying it to cosine function you can see a much greater fit for cosine. The problem is cubic spline is difficult to implement. So It seems that as long as you gather a large enough sample size for Lagrange polynomial, you can minizines the error, However; Cubic spline might be a better option if you're sample size isn't large enough.

Conclusion

Lagrange linear approximation is the easiest, but not very accurate. Lagrange Polynomial approximation is great if a large enough sample is collected. This will minimized the error and the wild oscillations. Neville's method will give the same results as Lagrange with the addition of dynamic programming since an explicit representation of the polynomial isn't needed. The wild oscillations will be solved with the Piecewise linear approximation, but it doesn't give the "smooth" look. The Piecewise cubic spline approximation will approximate the function with great accurately and remove those wild oscillations, but is difficult to implement.