CHAPTER

10 Interpolation

OUTLINE

Islands in the stream
That is what we are
No one in between
How can we be wrong?

–"Islands in the Stream" by Dolly Parton and Kenny Rogers

CHAPTER POINTS

- Polynomials can be used to approximate an unknown function between known values.
- High-degree polynomial interpolation is subject to oscillations known as the Runge phenomenon.
- Spline interpolants can give accurate and smooth interpolating functions when many points of the function are known.

In this chapter we will look at how we can take functions evaluated at particular points and fit polynomials to them. The points which we know the function could be from a table of values, measurements, etc. The process of creating a function that fits observed data is called inter-

DOI: 10.1016/B978-0-12-812253-2.00012-1

polation: the interpolation function will pass through the given points and make it possible to give values between the known data.

Polynomials are useful for interpolation because they are easy to evaluate and can be readily generated. As we will see, high-degree polynomials are not always the best choice, but a set of low-degree polynomials can be quite useful when we have many data points.

10.1 POLYNOMIALS

We denote a polynomial that is of degree n as $P_n(x)$. A polynomial of degree n has $n+1$ coefficients and is written as

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i.$$

It is possible to evaluate polynomials efficiently because there is a natural recursion. To compute the value of x^n we need to compute x^{n-1}, x^{n-2}, ..., x. Therefore, if we compute a term at a time, we only have to perform n multiplications to compute the values $\{x^n, x^{n-1}, \dots, x\}$. Below is a function that evaluates polynomials in this efficient manner.

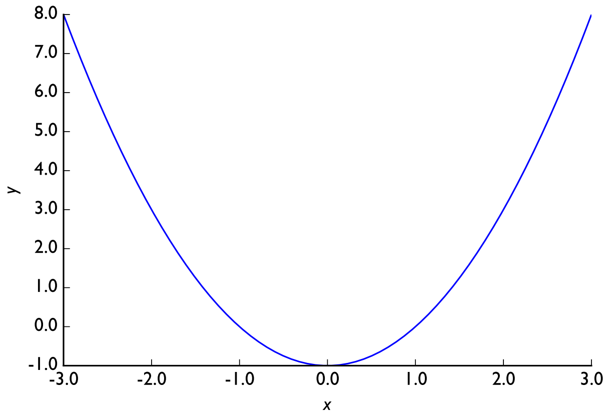
```
In [1]: def polynomial(a,x):
            """Evaluate a general 1-D polynomial at point
            Args:
                a: array of the n+1 coefficients of a polynomial
                x: the point to evaluate the polynomial at
            Returns:
                Pn(x)
            """
            num_coefficients = a.size
            answer = a[0]
            xpower = 1
            for i in range(1,num_coefficients):
                #the next power of x is x*previous power
                xpower *= x
                answer += a[i]*xpower
            return answer
```

The total number of floating point operations in this calculation is comprised of $2n$ multiplications (n from computing x^n and n more from the coefficient multiplications) and n additions. Testing out this function reveals that in practice there is basically no difference in the time it takes to evaluate a degree 10 polynomial and a degree 10^6 polynomial.

To demonstrate how this function works, we will test it on a simple quadratic polynomial:

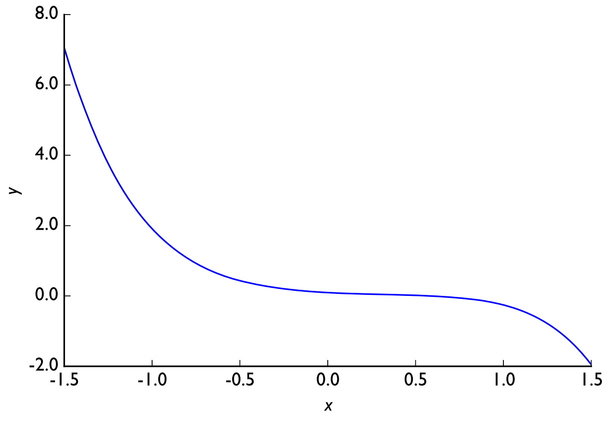
$$P_2(x) = (x-1)(x+1) = x^2 - 1.$$

```
In [2]: a = np.array([-1,0,1])
        X_points = np.linspace(-3,3,100)
        y = polynomial(a,X_points)
        plt.plot(X_points,y)
        plt.xlabel("x")
        plt.ylabel("y")
        plt.show()
```



A more complicated polynomial can also be evaluated. Next we evaluate a quintic, or fifth degree, polynomial with random coefficients.

```
In [4]: poly_degree = 5
        a = np.random.uniform(-2,2,poly_degree + 1)#+1 because of 0
        y = polynomial(a,X_points)
        plt.plot(X_points,y)
        plt.xlabel("x")
        plt.ylabel("y")
        plt.show()
```



Polynomials have the property that any function can be approximated by a polynomial to any desired degree of accuracy. This result, known as the Weierstrass approximation theorem, is written as

Weierstrass approximation theorem. *For any function $f(x)$ defined on the interval $a \le x \le b$, there exists a degree n polynomial with n finite where*

$$|P_n(x) - f(x)| < \epsilon,$$

for all $x \in [a,b]$ and any $\epsilon > 0$.

Therefore, any function you name, we can approximate it as well as you like with a polynomial inside a given interval.

BOX 10.1 NUMERICAL PRINCIPLE

The Weierstrass approximation theorem states that we can approximate any function well using polynomials. It does not state what degree the polynomial needs to be to make a good approximation.

In addition to the approximation properties of polynomials, these functions are also unique. For a given set of $n+1$ pairs of points $[x_i, f(x_i)]$, there is only one polynomial of degree n that passes through all those points. This makes sense because a polynomial of degree n has $n+1$ coefficients. Therefore, the $n+1$ data points define a unique polynomial. We will now discuss a method to compute the coefficients of polynomials given data.

10.2 LAGRANGE POLYNOMIALS

Lagrange polynomials are the simplest way to interpolate a set of points. This approach is not necessarily the most efficient for generating polynomial interpolating functions, but the difference is minimal for most applications. Regardless of the method used to compute the polynomial, the polynomial coefficients will be the same due to the uniqueness of interpolating polynomials.

The equations to construct a linear Lagrange polynomial are straightforward. Given points a_0 and a_1 and $f(a_0)$ and $f(a_1)$, the linear Lagrange polynomial formula is

$$P_1(x) = \frac{x - a_1}{a_0 - a_1} f(a_0) + \frac{x - a_0}{a_1 - a_0} f(a_1).$$

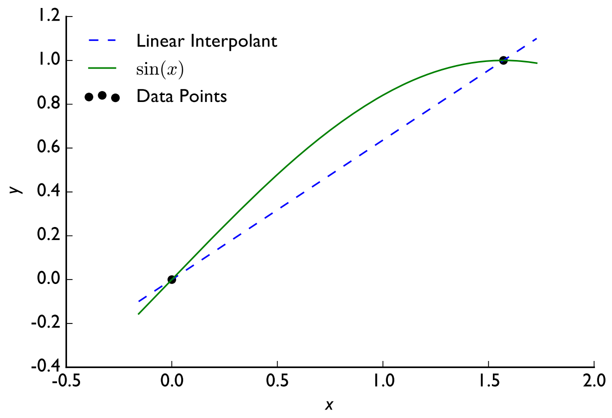
It is clear that $P_1(a_0) = f(a_0)$ and $P_1(a_1) = f(a_1)$ because in each case one of the numerators goes to 0. Also, it is clear that the function is a linear polynomial. We can translate this formula for linear interpolation into a function for Python:

```
In [5]: def linear_interp(a,f,x):
            """Compute linear interpolant
            Args:
                a: array of the 2 points
                f: array of the value of f(a) at the 2 points
```

```
    Returns:
        The value of the linear interpolant at x
    """
    assert a.size == f.size
    answer = (x-a[1])/(a[0]-a[1])*f[0] + (x-a[0])/(a[1]-a[0])*f[1]
    return answer
```

We can check this on an example when the function is $\sin(x)$ and we interpolate $x = 0$ and $x = \pi/2$.

```
In [6]: a = np.array([0,np.pi*0.5])
        f = np.sin(a)
        x = np.linspace(-0.05*np.pi,0.55*np.pi,200)
        y = linear_interp(a,f,x)
        plt.plot(x,y,linestyle="--",label="Linear Interpolant")
        plt.plot(x,np.sin(x),",label="$\sin(x)$")
        plt.scatter(a,f,c="black",label="Data Points")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.legend(loc="best")
        plt.show()
```



Note how the line passes through the two points we passed to the function. The line captures the overall trend of $\sin(x)$ between the two points, but we might want a better approximation.

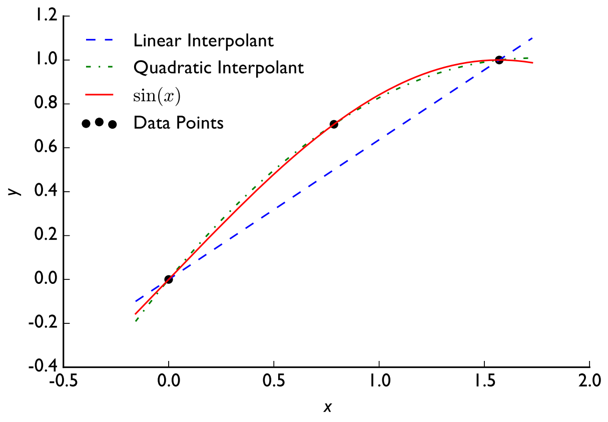
The formulation of the quadratic Lagrange polynomial is also straightforward. Given points a_0, a_1, and a_2 and $f(a_0)$, $f(a_1)$, and $f(a_2)$, the quadratic Lagrange polynomial formula is

$$P_2(x) = \frac{(x-a_1)(x-a_2)}{(a_0-a_1)(a_0-a_2)} f(a_0) + \frac{(x-a_0)(x-a_2)}{(a_1-a_0)(a_1-a_2)} f(a_1) + \frac{(x-a_0)(x-a_1)}{(a_2-a_0)(a_2-a_1)} f(a_2).$$

This formula now has three terms that each have the product of two linear polynomials. Similar to the linear Lagrange polynomial, the numerator vanishes on two of the terms when we evaluate the formula at one of the data points. A function to implement the quadratic Lagrange polynomial is given next.

```
In [7]: def quadratic_interp(a,f,x):
            """Compute the quadratic interpolant
            Args:
                a: array of the 3 points
                f: array of the value of f(a) at the 3 points
            Returns:
                The value of the quadratic interpolant at x
            """
            answer = (x-a[1])*(x-a[2])/(a[0]-a[1])/(a[0]-a[2])*f[0]
            answer += (x-a[0])*(x-a[2])/(a[1]-a[0])/(a[1]-a[2])*f[1]
            answer += (x-a[0])*(x-a[1])/(a[2]-a[0])/(a[2]-a[1])*f[2]
            return answer
```

We will do the same test as before on $\sin(x)$, the point we add is in the middle and compare our result to the linear interpolant.



Notice that the quadratic interpolant does a much better job than a linear function. Part of this is due to the fact that by adding the point in the middle we have decreased the distance between known values of the function. Moreover, between the known data points, the curvature of the quadratic interpolant follows the original function more closely.

Comparing the linear and quadratic Lagrange interpolation formulas, we can begin to see a pattern in the Lagrange interpolation formulas. Each term has in the numerator the product of $(x - a_i)$ where the a_i's are different than the point we evaluate $f(x)$ at. Also, the denominator is the product of the a point minus each other a point. This leads us to the general Lagrange interpolation formula:

$$P_n(x) = \sum_{i=0}^{n} f(a_i) \frac{\prod_{j=1, j\neq i}^{n} (x - x_j)}{\prod_{j=1, j\neq i}^{n} (x_i - x_j)},$$

```
Data: Degree: $n$, Points: $a_0, a_1, \ldots, a_n$, Function values: $f(a_0), f(a_1), \ldots, f(a_n)$,
      Evaluation Point: $x$
Result: The value of the $n$th degree Lagrange interpolant at point $x$
answer $= 0$;
for $i \in [0, n]$ do
    product $= 1$;
    for $j \in [0, n]$ do
        if $i \neq j$ then
            product $=$ (product) $\times \frac{x-a_j}{a_i-a_j}$;
        end
    end
    answer $=$ answer $+$ (product) $\times f(a_i)$;
end
```

Algorithm 10.1: Lagrange Polynomial Interpolation

where we have used the product notation:

$$\prod_{i=1}^{n} a_i = a_1 a_2 \ldots a_n.$$

The general Lagrange polynomial interpolation algorithm is given in pseudocode in Algorithm 10.1.

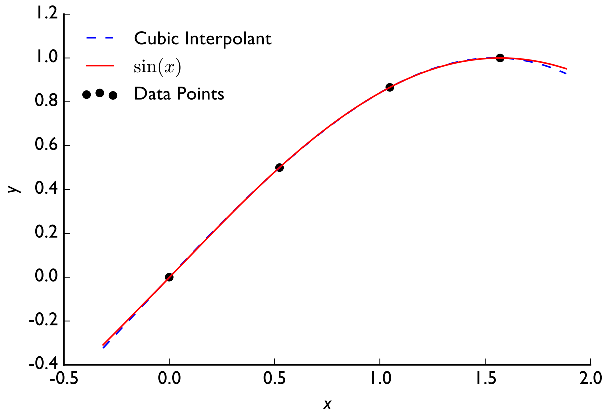
This algorithm is implemented in Python below.

```
In [9]: def lagrange_interp(a,f,x):
            """Compute a lagrange interpolant
            Args:
                a: array of n points
                f: array of the value of f(a) at the n points
            Returns:
                The value of the Lagrange interpolant at x
            """
            answer = 0
            assert a.size == f.size
            n = a.size
            for i in range(n):
                product = 1
                for j in range(n):
                    if (i != j):
                        product *= (x-a[j])/(a[i]-a[j])
                answer += product*f[i]
            return answer
```

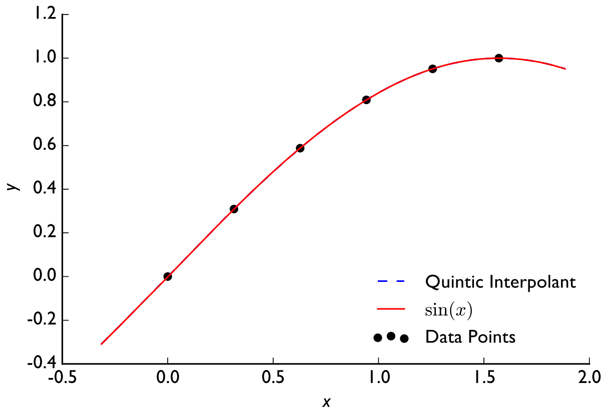
We can use this function to compute the interpolation of $\sin(x)$ using a degree 3, or cubic, polynomial.

```
In [10]: a = np.linspace(0,np.pi*0.5,4)
         f = np.sin(a)
```

```
x = np.linspace(-0.1*np.pi,0.6*np.pi,200)
y = lagrange_interp(a,f,x)
```



This interpolant is nearly indistinguishable from the original function over this range. With a general Lagrange interpolation routine, we can go even further and create a quintic interpolant. The result of running the code above with 6 points instead of 4 gives the following result:

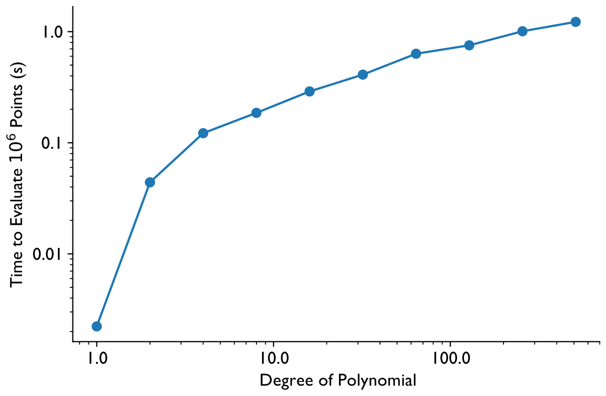


In this case we cannot distinguish between the original function and the interpolating polynomial. These results seem to indicate that higher degree polynomial interpolants are superior to lower degree, but there is a problem with high-degree interpolation called the Runge phenomenon that we discuss later.

10.2.1 Efficiency of Lagrange Polynomials

The Lagrange polynomial formula is not the most efficient way to compute an interpolating polynomial. Nevertheless, for almost any application the speed of modern computers means that our Lagrange polynomial is fast enough. This is especially true because using

NumPy we can compute the value of the polynomial at many x points at the same time for a given set of input data points. As shown in the figure below, it is possible to evaluate degree 512 polynomials at one million points in less than a second.



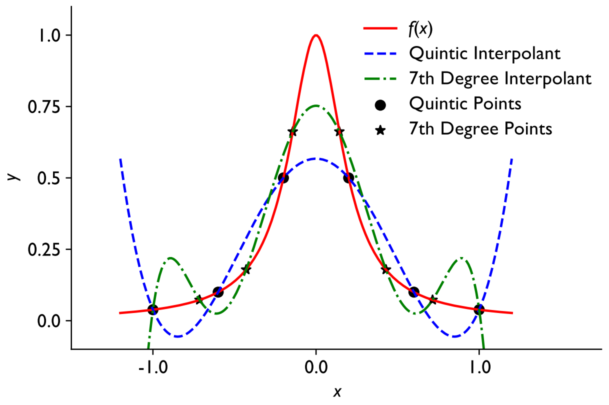
If the Lagrange polynomial formula is too slow for a particular application, there are other approaches discussed at the end this chapter.

10.2.2 The Runge Phenomenon

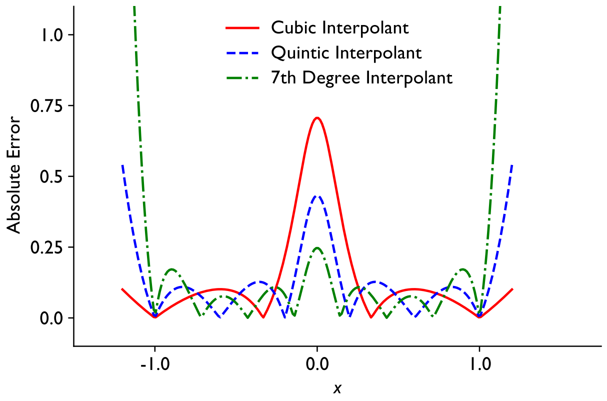
At degrees higher than 3, polynomial interpolation can be ill-behaved between the interpolation data. Specifically, the interpolating polynomial can have large oscillations. We can see this in a simple example with the function

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

This function seems to be fairly innocuous, but it tortures polynomial interpolation. We will look at 5th and 7th degree polynomials. Using our Lagrange polynomial function defined above, we produce the interpolating polynomials we get the following result:



Notice how the polynomial interpolants are very inaccurate near the edges of the domain and the behavior of the interpolating polynomials are completely different than the underlying function. In particular the higher degree polynomial has a larger maximum error and larger average error than the lower degree polynomial. This can be seen in the plot of the absolute value of the interpolation error below:



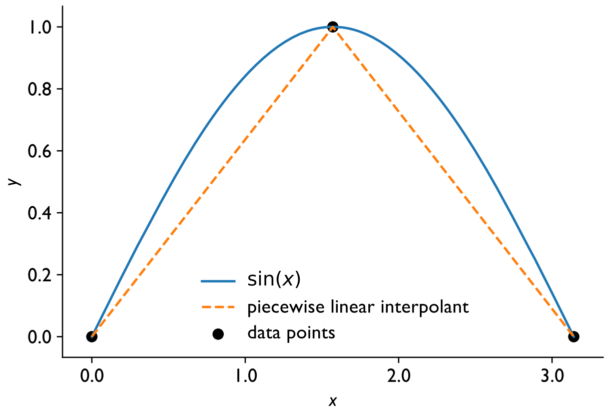
This behavior of high degree interpolants is called the Runge phenomenon: with equally spaced points a polynomial interpolant can have large oscillations going both high and below the actual function. This also makes extrapolating, that is evaluating the function outside the range of the data a dangerous endeavor. In this example, whether the function increases or decreases beyond the data depends on the degree of the interpolating polynomial. This high sensitivity to the choices made in interpolation make extrapolation untrustworthy outside the domain of the data, especially when the polynomial degree is high.

In general, performing high degree polynomial interpolation is a bad idea. In general it is better to fit low-degree polynomials to subsets of the data and then connect those polynomials. This is the idea behind spline interpolation.

10.3 CUBIC SPLINE INTERPOLATION

A cubic spline is a piecewise cubic function that interpolates a set of data points and guarantees smoothness at the data points. Before we discuss cubic splines, we will develop the concept of piecewise linear fits.

If we have several points, but do not want to have a high degree polynomial interpolant because of fear of the Runge phenomenon, we could do linear interpolation between each of the points.



This fit is OK, but it has some problems. The primary one is that it is not smooth at the data points: the function has a discontinuous derivative at some of the points. This would be the case even if we knew that the underlying function should be smooth. Also, a linear interpolant is not a good fit to the function: above we had much better luck with quadratics and cubics.

To guarantee a degree of accuracy, avoid the oscillations we have seen before, and get smooth functions we can, and should, use cubic spline interpolants. Cubic spline interpolants are continuous in the zeroth through second derivatives and pass through all the data points. The name spline comes from thin sticks, called splines, that were used in drafting in the days before computers. One also could imagine that these flexible sticks were used to strike colleagues in moments of merriment or anger.

To set up our algorithm we begin with the $n+1$ data points (x_i, y_i) (also called knot points), this implies n intervals and $n-1$ interior (not at the beginning or end) points. We will denote the cubic on the interval from point $(i-1)$ to i as

$$f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3.$$

The cubics need to match at the knot points so

$$f_1(x_1) = y_1,$$

$$f_i(x_{i+1}) = f_{i+1}(x_{i+1}) = y_{i+1} \qquad 1 \le i < n,$$

$$f_n(x_{n+1}) = y_{n+1},$$

which are $2n$ total conditions, when the end points are included. In equation form these become

$$a_1 + b_1 x_1 + c_1 x_1^2 + d_1 x_1^3 = y_1,$$

$$a_{i+1} + b_{i+1} x_{i+1} + c_{i+1} x_{i+1}^2 + d_{i+1} x_{i+1}^3 = a_i + b_i x_{i+1} + c_i x_{i+1}^2 + d_{i+1} x_{i+1}^3 = y_{i+1},$$

$$a_n + b_n x_{n+1} + c_n x_{n+1}^2 + d_n x_{n+1}^3 = y_{n+1}.$$

Also, we need to make the derivatives continuous at the interior knot points,

$$f_i'(x_{i+1}) = f_{i+1}'(x_{i+1}) \qquad 1 \le i < n.$$

The $n-1$ equations for this are

$$b_{i+1} + 2c_{i+1}x_{i+1} + 3d_{i+1}x_{i+1}^2 = b_{i+1} + 2c_{i+1}x_i + 3d_{i+1}x_i^2 \qquad 1 \le i < n.$$

Finally, we need to make the second derivatives continuous at the interior knot points,

$$f_i''(x_{i+1}) = f_{i+1}''(x_{i+1}) \qquad 1 \le i < n.$$

The $n-1$ equations for equality are

$$2c_{i+1} + 6d_{i+1}x_{i+1} = 2c_i + 6d_i x_{i+1}.$$

For the n intervals there are $4n$ unknowns (4 coefficients for each cubic). We have $4n-2$ equations at this point so we need two more equations. The natural choice is to set the second derivative to be zero at the two endpoints:

$$f_1''(x_1) = 0, \qquad f_n''(x_{n+1}) = 0.$$

We now have $4n$ equations and $4n$ unknowns.

We will build a cubic spline for $\sin(x)$ using $x = (0, \pi/2, \pi)$. This spline will have two intervals, meaning that there are 8 cubic coefficients we need to find. We fill a matrix with the equations for matching the function at the knot points first. The code below does this.

```
In [11]: #knot points are sin(x) at 0, pi/2,pi
         n = 2 #2 intervals
         data = np.zeros((n+1,2))
         data[:,0] = a
         data[:,1] = y_a
         coef_matrix = np.zeros((4*n,4*n))
         rhs = np.zeros(4*n)
         #set up the 2n equations that match the data at the knot points
         #first point
         x = data[0,0]
         coef_matrix[0,0:4] = [1,x,x**2,x**3]
         rhs[0] = data[0,1]
         #second point
         x = data[1,0]
         coef_matrix[1,0:4] = [1,x,x**2,x**3]
         rhs[1] = data[1,1]
         x = data[1,0]
         coef_matrix[2,4:8] = [1,x,x**2,x**3]
         rhs[2] = data[1,1]
         #third point
         x = data[2,0]
         coef_matrix[3,4:8] = [1,x,x**2,x**3]
         rhs[3] = data[2,1]
         print(coef_matrix[0:4,:])
```

```
[[ 1.         0.         0.         0.         0.         0.         0.         0.        ]
 [ 1.         1.57079633  2.4674011   3.87578459  0.         0.         0.         0.        ]
 [ 0.         0.         0.         0.         1.         1.57079633        2.4674011   3.87578459]
 [ 0.         0.         0.         0.         1.         3.14159265        9.8696044   31.00627668]]
```

These are the first four rows of the matrix to determine the 8 unknowns in our spline fit.

The next step is defining the equations for the first derivative at the interior point. This adds one more equation.

```
In [12]: #now the first derivative equations
         #second point
         x = data[1,0]
         coef_matrix[4,0:4] = [0,1,2*x,3*x**2]
         rhs[4] = 0
         coef_matrix[4,4:8] = [0,-1,-2*x,-3*x**2]
```

The last step in the construction of the equations is to create the equations for the second-derivatives at the knot points. One of these equations will be at the middle point, $x = \pi/2$, and the other two specify that the second derivative goes to zero at the endpoints.

```
In [13]: #now the second derivative equations
         #second point
         x = data[1,0]
         coef_matrix[5,0:4] = [0,0,2,6*x]
         rhs[5] = 0
         coef_matrix[5,4:8] = [0,0,-2,-6*x]
         #set first point to 0
         x = data[0,0]
         coef_matrix[6,0:4] = [0,0,-2,6*x]
         rhs[6] = 0
         #set last point to 0
         x = data[2,0]
         coef_matrix[7,4:8] = [0,0,2,6*x]
         rhs[7] = 0
         print(coef_matrix)
```

```
[[  1.          0.          0.          0.          0.          0.          0.          0.        ]
 [  1.          1.57079633   2.4674011    3.87578459   0.          0.
 0.        0.        ]
 [  0.          0.          0.          0.          1.          1.57079633
    2.4674011    3.87578459]
 [  0.          0.          0.          0.          1.          3.14159265
     9.8696044   31.00627668]
 [  0.          1.          3.14159265   7.4022033    0.          -1.
    -3.14159265  -7.4022033 ]
 [  0.          0.          2.          9.42477796   0.          0.
    -2.          -9.42477796]
 [  0.          0.          -2.         0.          0.          0.          0.          0.        ]
 [  0.          0.          0.          0.          0.          0.          2.          18.84955592]]
```

Finally, we solve the system of equations using Gauss elimination (from Chapter 7) and get the coefficients of the cubic spline interpolant.

```
In [14]: #solve for the cubic coefficients
         coefs = GaussElim(coef_matrix,rhs)
         print(coefs)
```

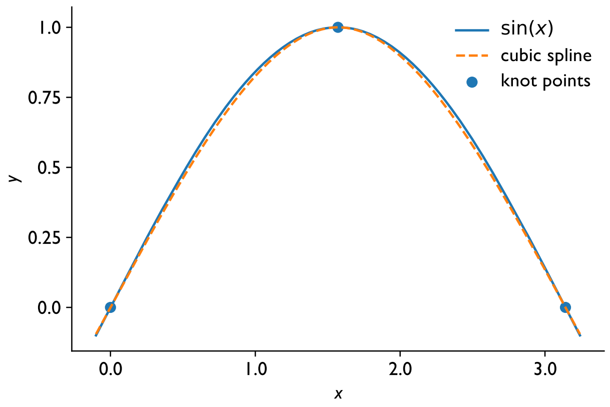
```
[ 0.          0.95492966          0.          -0.12900614
 -1.          2.86478898          -1.2158542          0.12900614]
```

Therefore, our approximation is

$$f(x) = \begin{cases} 0.95492966x - 0.12900614x^3 & x \leq \frac{\pi}{2} \\ -1 + 2.86478898x - 1.2158542x^2 + 0.12900614x^3 & x \geq \frac{\pi}{2} \end{cases}.$$

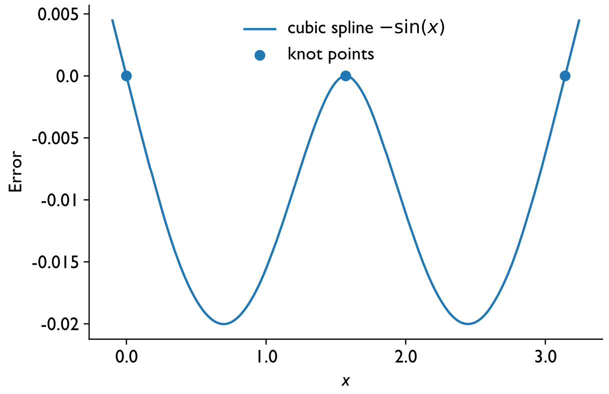
The beauty of the cubic spline interpolant is how well it approximates a function. If we look at the sine wave example from before, it is hard to distinguish the cubic spline interpolant from the original function. In the code below we evaluate the cubic spline fit. For a given point we need to determine which spline to use and this logic is expressed in the `if-elif-else` block.

```
In [15]:  #evaluate function
          points = 200
          X = np.linspace(-0.1,np.pi+0.1,points)
          y_interp = np.zeros(points)
          for i in range(points):
              if (X[i] < np.min(data[:,0])):
                  spline = 0
              elif (X[i] > np.max(data[0:n,0])):
                  spline = n-1
              else:
                  #knot to the left is spline
                  spline = np.argmax(X[i]-data[:,0])
              y_interp[i] = np.sum(coefs[4*spline:(4*spline+4)] *
                                   [1,X[i],X[i]**2,X[i]**3])
          plt.plot(X,np.sin(X), label="$\sin(x)$")
          plt.plot(X,y_interp, linestyle="--", label="Cubic Spline")
          plt.scatter(data[:,0],data[:,1],label="knot points")
```



In the following graph, the error, as defined by the difference between the true function and the interpolant, is quantified so that we can see how small it is:

```
In [16]:  plt.plot(X,y_interp-np.sin(X), label="cubic spline $- \sin(x)$")
          plt.scatter(data[:,0],0*data[:,1],label="knot points");
          plt.xlabel("x")
          plt.ylabel("Error")
          plt.legend(loc="best")
```



Making the cubic spline was straightforward, but tedious. Additionally, evaluating the splines involved determining which spline to use and then evaluating the function. Thankfully, the package `SciPy`, a companion package for NumPy that implements many numerical algorithms, has a cubic spline function that we can use.

The function in the `interpolate` section of SciPy that we want to use is called `CubicSpline`. It takes in the data points `x` and `y` as inputs, and returns a function that we can evaluate. There are several options for the conditions at the beginning and end points. We have already discussed the natural spline conditions where the second derivative of the splines at the first and last knot point are set to zero, i.e.,

$$f_1''(x_1) = f_n''(x_{n+1}) = 0, \qquad \text{natural spline conditions.}$$

Another type of spline is the "clamped" spline where the first-derivative is set to zero at the end points:

$$f_1'(x_1) = f_n'(x_{n+1}) = 0, \qquad \text{clamped spline conditions.}$$

The default condition used by `CubicSpline` is the "not-a-knot" condition where the third derivative of the first and last splines is fixed so that it matches the third derivative at the nearest interior point:

$$f_1'''(x_2) = f_2'''(x_2), \qquad f_{n-1}'''(x_n) = f_n'''(x_n), \qquad \text{not-a-knot spline conditions.}$$

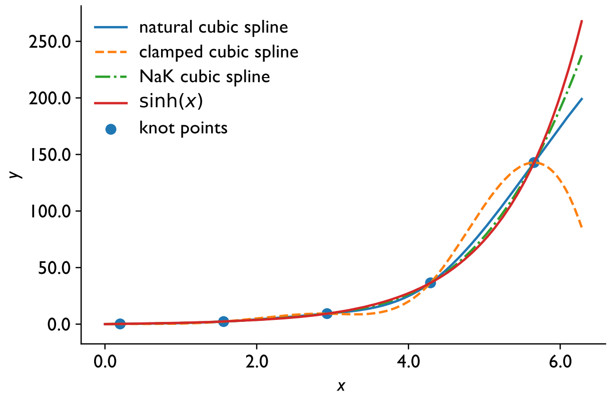
Plugging in the cubics to this equation we get the conditions

$$a_1 = a_2, \qquad a_{n-1} = a_n.$$

This implies that the spline in the first and nth interval are the same as the splines in the second and $(n-1)$ intervals, respectively. In this sense, the endpoints are not treated as knots, just a point that the interior spline must pass through.

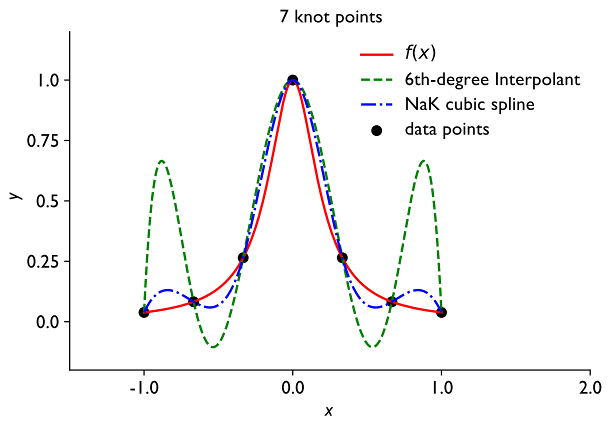
The code below compares these splines on an interpolation problem. This time we apply it to the hyperbolic sine function using five knot points, meaning there are four different splines.

```
In [17]:  from scipy.interpolate import CubicSpline
          #define data
          a = np.linspace(0.2,np.pi*1.8,5)
          data = np.zeros((5,2))
          data[:,0] = a
          data[:,1] = np.sinh(a)
          #define splines
          splineFunction = CubicSpline(data[:,0],data[:,1],bc_type='natural')
          splineFuncClamp = CubicSpline(data[:,0],data[:,1],bc_type='clamped')
          splineFuncNot = CubicSpline(data[:,0],data[:,1],bc_type='not-a-knot')
          #make plot
          points = 200
          X = np.linspace(0,np.pi*2,points)
          plt.plot(X,splineFunction(X),
                   label="Natural Cubic Spline")
          plt.plot(X,splineFuncClamp(X),linestyle="--",
                   label="Clamped Cubic Spline")
          plt.plot(X,splineFuncNot(X),linestyle="-.",
                   label="NaK Cubic Spline")
          plt.plot(X,np.sin(X), label="$\sinh(x)$")
          plt.scatter(data[:,0],data[:,1],label="knot points")
```

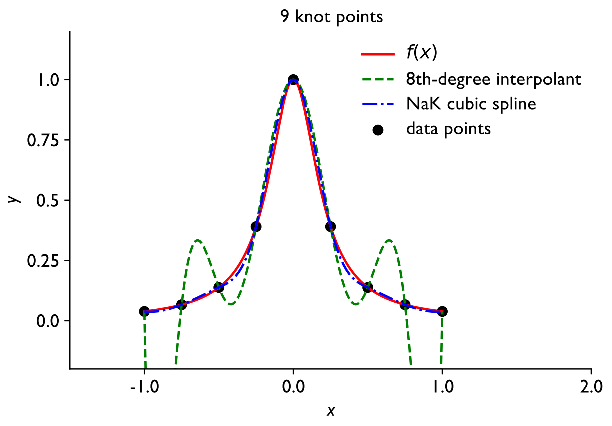


It is apparent that the choice of spline type does have an affect on the interpolation. The clamped splines force a local extreme point to be created at the endpoints because the derivative goes to zero. Similarly, the natural splines create an inflection point (i.e., the second-derivative is zero at the endpoints). For this particular problem the not-a-knot splines work best near the large values of x.

On the Runge phenomenon example from before, cubic spline interpolants perform better than high-degree polynomials as shown in this next figure.



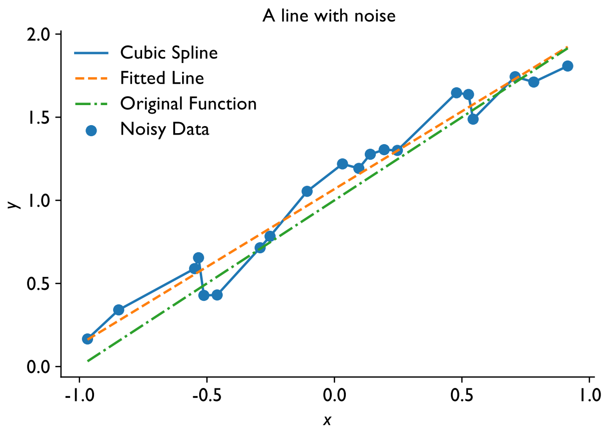
Adding two more knot points makes the spline fit better



Notice how the splines get better, but the full polynomial gets worse. That is why cubic splines are the go to method for fitting curves through a series of points.

CODA

Next time we will talk about those times when it is not desirable to have the function touch every point. These problems are curve-fitting problems and sometimes called regression. In these problems, as demonstrated below, we do not want the function to interpolate the data but to find some trend in the data. In the following figure a line that has had random noise added to it is shown, along with the original function, a cubic spline interpolant, and a line fitted by linear regression. We will cover this topic in the next chapter.



FURTHER READING

As mentioned above, there are other ways to compute interpolating polynomials than the Lagrange polynomial formula. All of the methods give the same the result, but different algorithms to compute them. Neville's algorithm and Newton's divided difference formula are two well-known polynomial construction techniques. Additionally, it is possible to use other functions for interpolation. Rational function interpolation writes the interpolant as the quotient of two polynomials and can give well-behaved interpolants where polynomial interpolation is too oscillatory. For periodic functions, trigonometric interpolation is another approach.

PROBLEMS

Short Exercises

10.1. Compute, by hand, the second-degree Lagrange polynomial for the points, $x = \{0, 2, 4\}$ and $f(x) = \{1, 0.223891, -0.39715\}$. If $f(3) = -0.260052$, how accurate is the interpolation at this point. Plot your interpolating polynomial for $x \in [0, 5]$. The points were generated from the Bessel function of the first kind, $J_0(x)$. Compare your interpolating polynomial to the actual function.

10.2. Repeat the previous exercise with a clamped cubic spline and a not-a-knot cubic spline.

Programming Projects

1. Root-Finding via Interpolation

Given the following function data

x	2.5	2.4	1.2	1	0.5	0.2	0.1
$f(x)$	−0.0600307	−0.0594935	−0.039363	−0.0279285	0.071541	0.43361	0.73361

Use interpolation to find the root of the function, that is find x such that $f(x) = 0$. You can do this by fitting interpolating functions to the data using $\hat{x}_i = f(x_i)$, and $\hat{f}(\hat{x}_i) = x_i$, and then evaluating $\hat{f}(0)$. Find the best approximation you can using an interpolating polynomial of maximum degree, and the three kinds of splines discussed above. The correct answer is $x = 0.75$. Comment on the results.

2. Extrapolation

Consider the following data for an unknown function. In the problems below $\log x$ denotes the natural logarithm of x.

x	1.2	1.525	1.85	2.175	2.5
$f(x)$	5.48481	2.3697	1.62553	1.28695	1.09136

Use interpolating polynomials of different degree, and three kinds of splines to estimate $f(3)$. Repeat the previous procedure after logarithmically transforming the data, i.e., set $\hat{x}_i = \log x_i$, and $\hat{f}(\hat{x}_i) = \log f(x_i)$. The correct answer is $f(3) = 0.910239226627$. Discuss which approaches performed best for both the linear and logarithmic interpolation.

3. Moderator Temperature Coefficient of Reactivity

The change in the reactivity for a nuclear reactor due to changes in the moderator temperature is called the moderator temperature coefficient of α_m is the logarithmic derivative of k_∞ for the reactor as

$$\alpha_\mathrm{m} = \frac{1}{k_\infty}\frac{\partial k_\infty}{\partial T_\mathrm{m}} = -\beta_\mathrm{m}\left(\log\frac{1}{p} - (1-f)\right),$$

where the subscript m denotes "moderator", p is the resonance escape probability for the reactor, f is the thermal utilization, and β_m is

$$\beta_\mathrm{m} = -\frac{1}{N_\mathrm{m}}\frac{\partial N_\mathrm{m}}{\partial T_\mathrm{m}},$$

with N_m the number density of the moderator.

Consider a research reactor that is cooled by natural convection. It has $p = 0.63$ and $f = 0.94$. Plot the moderator temperature coefficient from $T_\mathrm{m} = 285$ K to 373 K as a function of temperature for this reactor, using the data below from the National Institute for Standards and Technology (NIST) for T_m in K, and density, ρ_m, in mol/liter:

T_m	ρ_m	$\frac{\partial \rho_\mathrm{m}}{\partial T_\mathrm{m}}$
289.99	55.442	−0.00769
300.12	55.315	−0.0143
320.53	54.909	−0.0231
346.13	54.178	−0.0308
366	53.475	−0.0385
369.87	53.326	−0.0385