9 Interpolation

Let's start with the definition. Interpolation is the method of constructing new data points within the range of a known data points.

Suppose we are given a dataset with the average temperature for each month of the year. The dataset for Toronto is printed in Table 1 and plotted in Figure 20. Each data point represents the average temperature during one month. Note that the months are enumerated with integers starting at 0 (January). There is no information given about the temperature on a particular day.

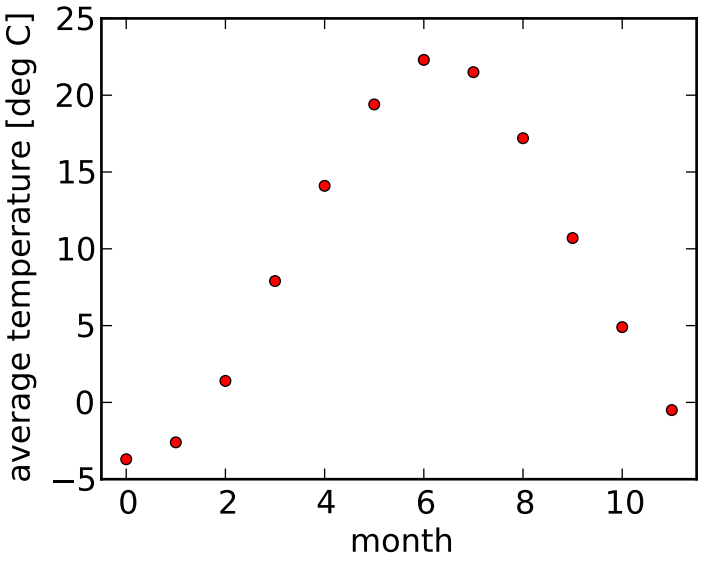

Figure 20: Average monthly temperature in Toronto (0 = January). Source: Environment Canada.

Month	Average temperature [C]
0	-3.7
1	-2.6
2	1.4
3	7.9
4	14.1
5	19.4
6	22.3
7	21.5
8	17.2
9	10.7
10	4.9
11	-0.5

Table 1: Temperatures in Toronto (0 = January).

If someone asks, "What is the temperature at the *end* of January", the honest answer would be "I don't know". The dataset that was given to us only knows about the temperature averaged over the entire month of January. However, one can see by eye, that there is a clear trend: it is coldest in January and warmest in July.

If you want to be able to give an answer but don't have any additional data available, we use a method called **interpolation**. In general, you will be given a set of N function values of an unknown function f in the form

$$(x_0, f(x_0)),\ (x_1, f(x_1)),\ \ldots\ ,\ (x_{N-1}, f(x_{N-1}))$$

and your task is to estimate $f(x)$ for an arbitrary x in $x_0 \leq x \leq x_{N-1}$.

9.1 Interpolation with a piece-wise constant function

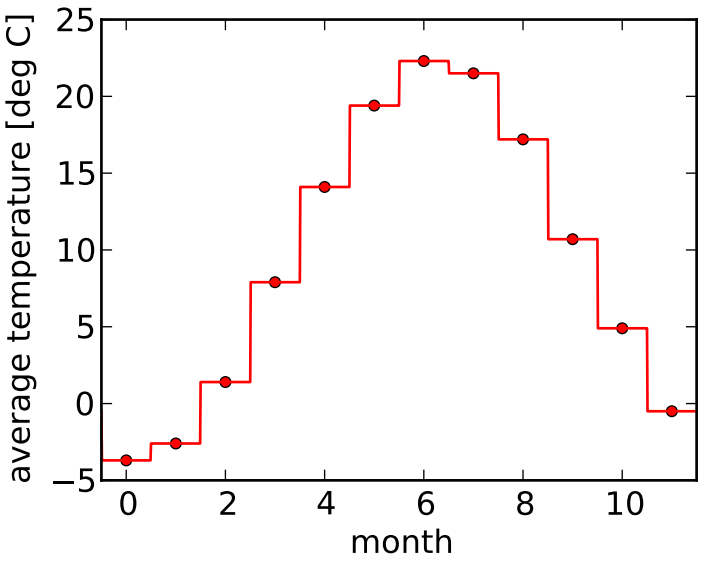

Figure 21: Temperature in Toronto (0 = January). Piece-wise constant interpolation.

To come back to the question from above, a reasonable answer would be, "Well, I don't know what the typical temperature at the end of January is, but it is probably not drastically different from the average temperature in January.". This idea corresponds to the simplest interpolation one can do: using piece-wise constant functions. It is sometimes also referred to as nearest-neighbour interpolation.

We simply choose the x_i and the corresponding $f(x_i)$ which is closes to the x that we are interested in. The result is a discontinuous series of constant functions. If you know nothing about the data points that are given to you, to get a value of the interpolated function $f(x)$, you need to go through the entire list of known data points to find the one that is closest. For the temperature data from above, we can simply things quite a bit. First, we have the data already sorted. Second, the known data points have integers (the months) as their x value. So we only need to find the closes integer to the x that we are interested in and then look up the known data point. Figure 21 shows a piece-wise constant function interpolation to the Toronto weather data. The following function shows a possible implementation in python, assuming the known data points are given in the array `temperature[]`.

```
def temperature_constant_interpolation(x):
    x = int(math.round(x))
    return temperature[x]
```

Code 53: Python listing implementing piece-wise constant interpolation of the temperature data.

We can also apply this method to a function that depends on two variables x and y. In this two dimensional case, we'll end up with a Voronoi diagram as shown in Figure 22

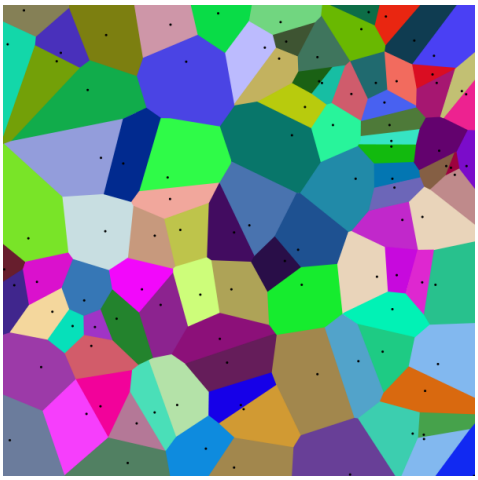

Figure 22: Example of a piece-wise constant interpolation in two dimensions. Source: Wikimedia (CCSA).

9.2 Interpolation with a piece-wise linear function

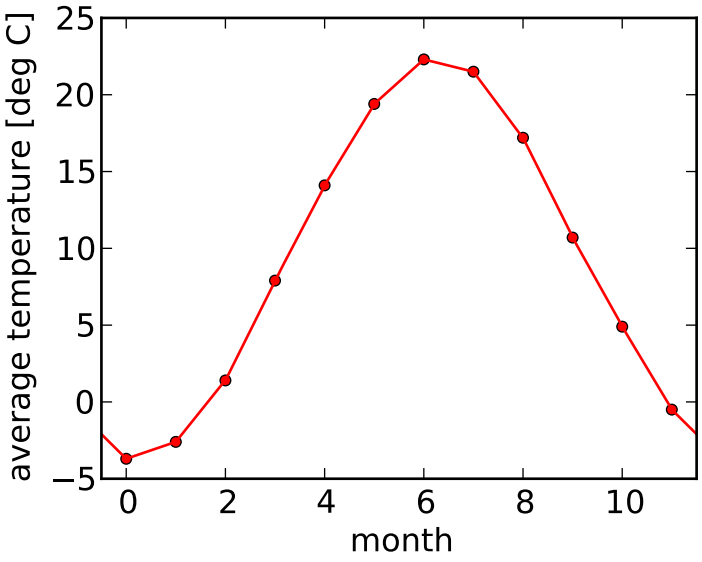

Figure 23: Temperature in Toronto (0 = January). Piece-wise linear interpolation.

There are several disadvantages with a pice-wise constant interpolation. First of all, it's not a very good approximation. Most likely, it is warmer at the end of April than at the beginning of April. However, we have assumed a constant temperature across the entire month. Furthermore, at the transition from end of April to beginning of May the temperature suddenly jumps by 5 degrees. This seems unrealistic. Mathematicians call the piece-wise constant interpolation discontinuous.

We can indeed do better. What if we just draw a straight line from one data point to the next? This is called piece-wise linear interpolation, sometimes just referred to as linear interpolation or lerp. The result looks much smoother, as shown in Figure 23. Now the function is continuous.

However, some people will still not be happy with this. If you look closely, you'll see that the function has a sharp kink at every data point. That means we can't calculate the derivative of the interpolating function at those points. Hence the piece-wise linear interpolation is non-differentiable. We'll later discuss another interpolation scheme which is both continuous and differentiable.

But before we do that, let's look at how we calculate the piece-wise linear function. First, we need to find the given data point to the left and right of the value x that we are interested in. Let's call them x_l and x_r. Then, the interpolation function $\tilde{f}(x)$ is

$$\tilde{f}(x) = f(x_l) + \big(f(x_r) - f(x_l)\big) \cdot \frac{x - x_l}{x_r - x_l}$$

Remember that the values $f(x_l)$ and $f(x_r)$ are given to us. What we don't know is how the function f looks like, which is why we are constructing an approximation (an interpolation), which we call $\tilde{f}$.

The python source code to create the piece-wise linear interpolation of the temperature data is shown in Code 54. Note that the code snippet is specific for the periodic temperature example from. It assumes the temperatures are stored in the one dimensional array `temperature` with indices ranging from 0 to 11. Note that the code does not check if x_l or x_r are exceeding the array bounds.

```
def temperature_linear_interpolation(x):
    xl = int(math.floor(x))
    xr = int(math.ceil(x))

    f_xl = temperature[xl]
    f_xr = temperature[xr]

    return f_xl + (f_xr-f_xl)*(x-xl)/(xr-xl)
```

Code 54: Python listing implementing piece-wise linear interpolation of the temperature data.

9.3 High order polynomial interpolation

So far, we have only considered the point directly to the left and right of the value x that we want to interpolate to. What if we take into account all the data points that we are given. If we have N data points, then we can find a polynomial of degree $N-1$ that goes through all the points and which we can evaluate at any point x to find the interpolated value we want to know. This is exactly what the Lagrange interpolation polynomial does.

Given a set of N data points of the form $(x_0, y_0), \ldots, (x_{N-1}, y_{N-1})$ the interpolation polynomial in Lagrange form is a linear combination

$$L(x) = \sum_{j=0}^{N-1} y_j \ell_j(x)$$

where the Lagrange basis polynomials are

$$\ell_j(x) = \prod_{\substack{0 \le m \le N-1 \\ m \neq j}} \frac{x - x_m}{x_j - x_m} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_{N-1})}{(x_j - x_{N-1})}$$

Have a look at the basis polynomials. They are constructed such that they are exactly zero at all data points points except the j-th one, where the basis polynomial is 1. This guarantees that all data

points lie on the curve. An example of a Lagrange polynomial, fitted to our temperature dataset is shown below.

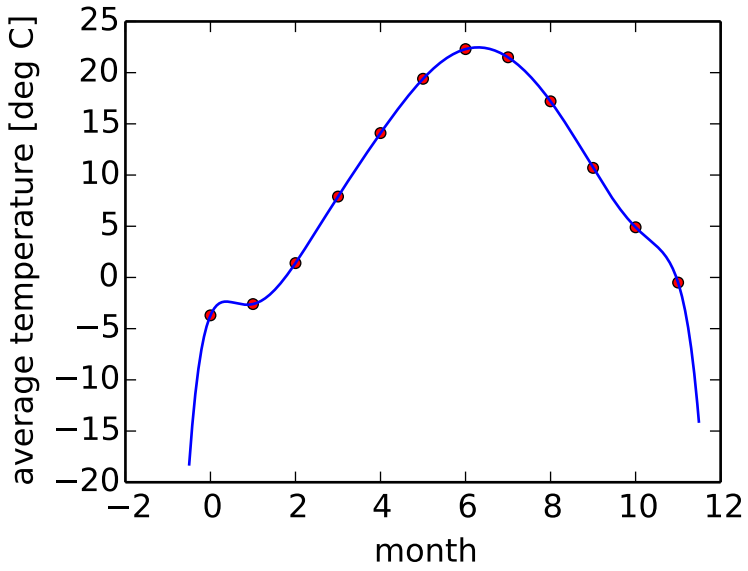

Figure 24: Lagrange interpolating polynomial to the temperature dataset.

This figure also illustrates a few of the problems associated with high order polynomial interpolation. First, you can see that outside of the interval where we have data points, the curve goes of very quickly. This makes sense, since it is a high order polynomial after all! Second, you can see a hint of *ringing*. Just between the data points at $x = 0$ and $x = 1$ you can see a local maximum. This is clearly not shown in our original data and purely an *artifact* of our interpolation routine. Ringing can be much worse than in the example shown above. Especially if the data points are spaced equidistantly or if there is some noise in the data (which will be for all measurements taken in a real world scenario). One way to avoid ringing is to choose the x points according to Chebyshev nodes. We will not cover this in the lectures, but you might want to keep the name in mind.

Nevertheless, you can see that the Lagrange polynomial is an excellent choice to get a good estimate of interpolated values in-between data points. The curve is very smooth, in fact, it is *infinitely differentiable*.

9.4 Cubic spline interpolation

Instead of going to higher and higher order, there is another way of creating a smooth function that interpolates data-points. A cubic spline is a piecewise continuous curve that passes through all of the values of a given dataset. This works particularly well for smooth datasets with no noise. Each of the piecewise curves is a cubic polynomial with coefficients a_i, b_i, c_i and d_i:

$$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i \qquad \text{for } x \in [x_i, x_{i+1}]$$

If we have N data-points, there are $N - 1$ intervals, hence $(N - 1) \cdot 4$ coefficients that we need to find. Two conditions in each interval arise because we have to match the two data-points at each end.

$$S_i(x_i) = y_i, \quad S_i(x_{i+1}) = y_{i+1}$$