3.4 Why shifting and scaling?

Suppose you have a system and you are reading a sensor periodically, perhaps at 10 Hz, so once every 100 ms. To give a more concrete example, suppose you have a physical vapor deposition (PVD) device at your place of employment. It's one-of-a-kind in-house solution, so the temperature sensor in this case happens to be both sufficiently accurate and precise to allow polynomial interpolation to be used to estimate future values. Suppose the temperature must be maintained between 710 K and 730 K and the temperature sensor returns a value to one digit after the decimal point.

Assuming that the device is turned on at time $t_0 = 0$, so after a while, the clock may end up at a value such as $t_{25527} = 2552.7$ s (so the machine has been on for just over seven hours). Suppose that the last three readings are

$$T_{25525} = 717.2 \text{ K}, T_{25526} = 719.4 \text{ K, and } T_{25527} = 720.7 \text{ K}$$

with the last being the most recent.

The temperature is clearly increasing, but you'd like to estimate what the temperature will be at time 2552.75 (50 ms into the future) and 2552.8 (100 ms into the future). To do this, we will find an interpolating polynomial and evaluate this polynomial at these two new points, as shown in Figure 1.

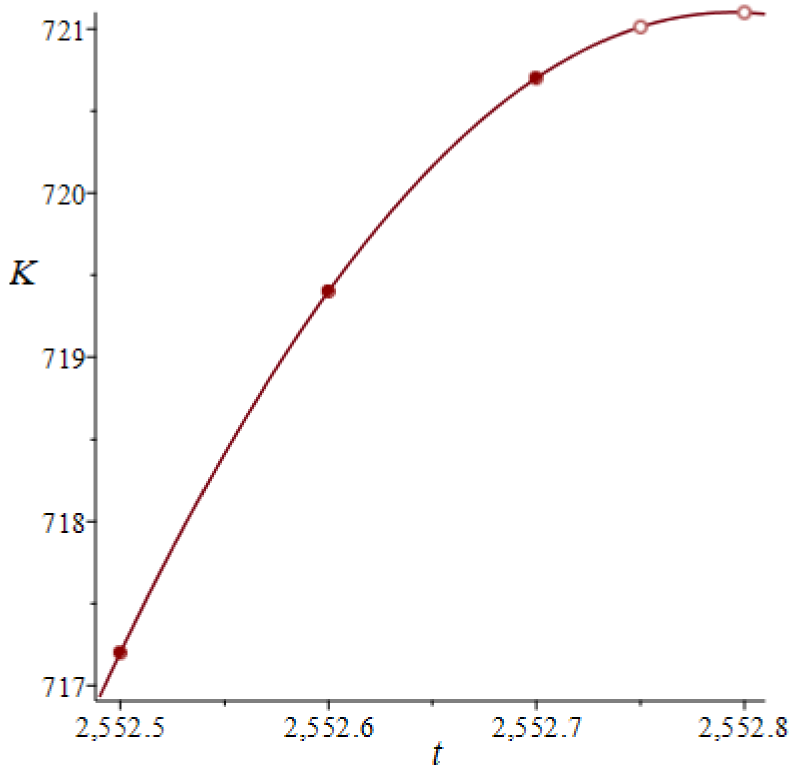

Figure 1. An interpolating polynomial through three points extrapolated to 2552.75 and 2552.8.

This is described as the process of *extrapolation*, as we are extrapolating into the future based on currently available information. This is only really useful and even possible if you have precise and accurate information, and later we will see more appropriate techniques for approximating values in the future. However, if you were to find the interpolating polynomial $at^2 + bt + c$, you would simply create the Vandermonde matrix and solve the system of linear equations

$$\begin{pmatrix} 2552.7^2 & 2552.7 & 1 \\ 2552.6^2 & 2552.6 & 1 \\ 2552.5^2 & 2552.5 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 720.7 \\ 719.4 \\ 171.2 \end{pmatrix}.$$

The first issue is that the condition number of the matrix is very large: 8495036449221147.49 or approximately 8.5 quadrillion, so any error in the temperature ***could*** potentially be magnified significantly. However, if we apply Gaussian elimination with partial pivoting together with backward substitution, we get that the interpolating polynomial is

$$-44.999999t^2 + 229751.492501t - 293253445.728755$$

The exact interpolating polynomial is exactly

$$-45t^2 + 229751.5t - 293253455.3$$

so it may appear that the interpolating polynomial found numerically is reasonably accurate and precise. Next, let us substitute in a value such as $t = 2552.8$: you will note we are adding or subtracting three very large numbers (each on the order of half a billion), and out of this sum, we must get a value on the order of 720. For example, if we do evaluate this at 2552.8, we are now needing to calculate

$$-293255443.2265998 + 586509610.0553545 - 293253445.7287546$$

Fortunately enough, due to all the techniques used to mitigate the issues of numeric error, the answer is still very close to the correct answer: this calculates to 721.1000000834465, which is still very close[1] to the correct value of 721.1, but as you may suspect, subtractive cancellation may easily cause more significant errors. Similarly, if we evaluate this polynomial at 2552.75, we get an answer 721.0125000476837 which is very close to the correct answer 721.0125. Also, it is not necessarily known if this will continue to return accurate approximations for other values of t (such as for either different times, different time steps, or both).

Now, one more issue is that with each next time step, if we want to repeat the process, this requires a new Vandermonde matrix, and we must once again solve a new system of linear equations. Fortunately, if the system of linear equations is set up correctly (the most recent time in the first row), the process will always be the same:

1. Add appropriate multiples of Row 1 onto Rows 2 and 3.
2. Swap Rows 2 and 3 (the entry (3, 2) will always be greater in absolute value than (2, 2)).
3. Add an appropriate multiple of Row 2 onto Row 3.

This can, with intelligent coding, be reduced to five multiplications and additions, followed by backward substitution which requires three multiplications, three subtractions, and three divisions. This sums to 19 floating-point operations, or 19 FLOPs. Evaluating this interpolating polynomial at a point using Horner's rule pushes the number of operations to 23 FLOPs.

[1] The astute student may note that there are 10 digits correct in this approximation, that the numbers being added and subtracted are on the order of 10^6 larger, and thus this accounts for why the error begins with the 11th digit.

Can we, however, do better? And are there scenarios where this algorithm may fail?

More interestingly, suppose we ignore the actual times t = 2552.5, t = 2552.6 and t = 2552.7, and assume instead that the three most readings are at time $\tau = 0$, $\tau = -1$ and $\tau = -2$, so instead of interpolating

(2552.5, 717.2 K), (2552.6, 719.4 K), (2552.7, 720.7 K),

and finding a quadratic polynomial $p(\tau)$, we will instead interpolate

(–2, 717.2 K), (–1, 719.4 K), (0, 720.7 K),

and find a quadratic polynomial $q(\tau) = a\tau^2 + b\tau + c$. Note that the T-values are unchanged. Thus, we have the following system of linear equations

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & -1 & 1 \\ 4 & -2 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} T_k \\ T_{k-1} \\ T_{k-2} \end{pmatrix}$$

where T_k is the most recent reading. Note that condition number of this new matrix is small: only 28.

We can solve this at design time to get:

$$\begin{aligned} a &= \frac{T_k + T_{k-2}}{2} - T_{k-1} = \frac{T_k - 2T_{k-1} + T_{k-2}}{2} \\ b &= \frac{3T_k + T_{k-2}}{2} - 2T_{k-1} = \frac{3T_k - 4T_{k-1} + T_{k-2}}{2} \\ c &= T_k \end{aligned}$$

Why does this work?

The first question you may ask is: how and why can we do this? After all, if we plot the two interpolating polynomials on the same scale, they look very different, as is shown in Figure 2.

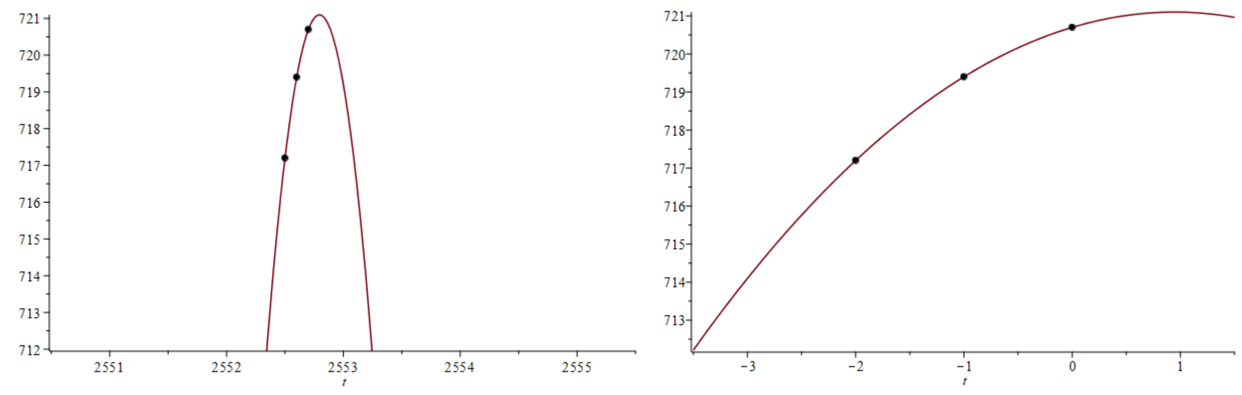

Figure 2. The polynomial interpolating (2552.5, 717.2), (2552.6, 719.4) and (552.7, 720.7) on the left, and the polynomial interpolating (–2, 717.2), (–1, 719.4) and (0, 720.7) on the right.

The polynomial on the left is significantly shrunk along the abscissa (the *t*-direction); however, if you were to evaluate both polynomials one time step into the future (0.1 on the left, and 1.0 on the right), you get the same result: 721.1. Thus, while the polynomial on the left is shrunk by a factor of 10 along the abscissa, the value going forward or going back one time step or even half or any multiple of a time step is the same.

Similarly, the derivative at the right-most point in the left-hand plot looks significantly larger than the derivative at the right-most point of the right-hand plot; however, if you were to divide the slope on the right (0.85) by the time step on the left, (0.1 in this case), you will get the slope of the derivative of the point on the left (8.5), as shown in Figure 3.

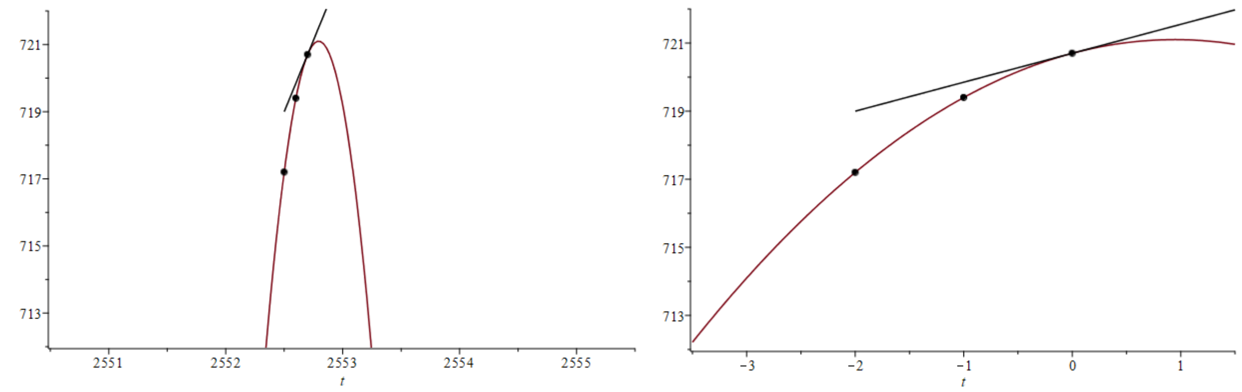

Figure 3. The derivative of the interpolating polynomial at the third point.

Similarly, the integral of the interpolating polynomial on the left across the right two points is 72.0125, while the integral of the interpolating polynomial on the right between –1 and 0 is 720.1250000, the one on the right being greater by a factor of ten, and thus, finding the integral of the interpolating polynomial on the right allows us to find the corresponding integral by multiplying the result by 0.1, as is shown in Figure 4.

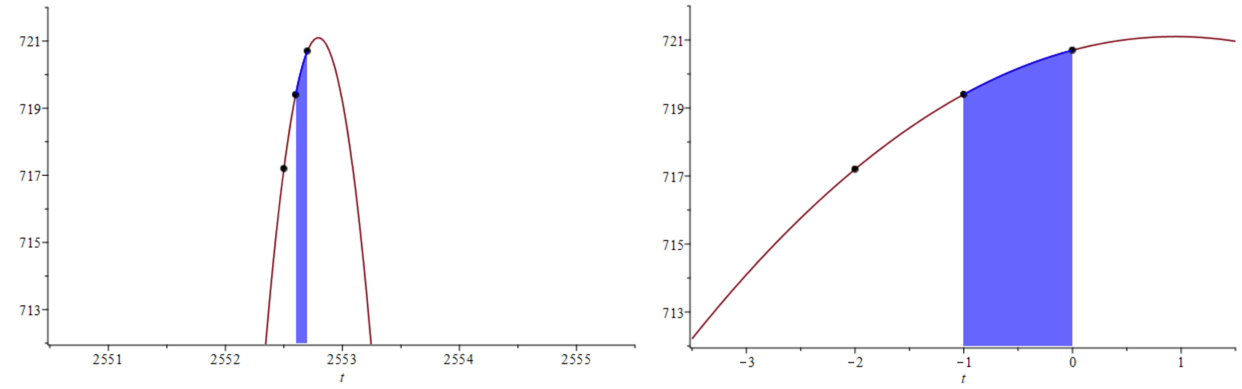

Figure 4. The integral of the interpolating polynomial between the two right-most points in each case, the integral on the left being 0.1 (the time step) being multiplied by the integral on the right.

Consequently, properties of the original interpolating polynomial can be derived from the properties of the interpolating polynomial on the polynomial that has the *t*-values shifted and scaled to match –2, –1 and 0; however, we can easily find the interpolating polynomial on the right and hard-code it, while the interpolating polynomial on the left requires us to solve a system of linear equations, and the corresponding matrix has a significantly larger condition number.

Consequently, if we wanted to evaluate the original polynomial

1. half a time-step into the future, we would now evaluate $q(0.5)$, and
2. a full time step into the future, we would now evaluate $q(1.0)$.

Similarly, if we wanted to evaluate the integral of the original interpolating polynomial over any time interval, we would multiply the integral over the corresponding time interval of our $q(\tau)$, then multiplying the result by the time step. Similarly, if we wanted to evaluate the derivative of the original interpolating polynomial at any particular time, we would simply find the derivative of the corresponding point in time of our $q(\tau)$, then divide the result by the time step.

You will also notice that we can actually code this interpolating polynomial at design time:

```
// double extrapolate( ... )
//
//   Parameters:
//      tau       A value -1 <= tau <= 1 at which we will
//                evaluate the interpolating polynomial
//      T[]       An array of observations
//       k        An index into the array so that
//                  T[k] is the most recent observation,
//                  T[k-1] is the next most recent observation, and
//                  T[k-2] is the most recent observation after T[k-1]
//
// Find the interpolating polynomial that passes through
//     (-2, T   ), (-1, T   ) and (0, T )
//           k-2         k-1           k
// and evaluate that interpolating polynomial at tau.

double extrapolate( double tau, double T[], std::size_t k ) {
    // We should not call this function if |tau| > 1
    //   - extrapolation has too large an error for large tau
    assert( (-1.0 <= tau) && (tau <= 1.0) );

    // This uses Horner's rule.
    return (
        (0.5*(T[k] + T[k - 2]) - T[k - 1])*tau
            + (1.5*T[k] + 0.5*T[k - 2]) – 2.0*T[k - 1])
    )*tau + T[k];
}
```

Finally, note that if we evaluate this interpolating polynomial at $\tau = 0.5$ and $\tau = 1$ using floating-point operations, we get exactly the values 721.0125 and 721.1 without any numeric error. Instead of 23 FLOPs, this reduces the number of floating-point operations to only twelve (12), so with approximately half the number of floating-point operations, we actually get a more precise and accurate value.