Supplementary notes on interpolation

The interpolation problem:
Given a set of ordered pairs (or points in the plane) \((x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n),\) find a function \(\Phi(x)\) from a given family of functions such that \(\Phi(x_i) = y_i, i = 0, \ldots, n.\)
Graphically, this means that the graph of the function \(\Phi\) passes through the points \((x_i, y_i)\) in the plane. (We assume that the \(x_i\) represent distinct values.)

Terminology: We say that the function \(\Phi\) above "interpolates" the data and that \(\Phi\) is the "interpolant" of the data. The points \(x_0, x_1, \ldots, x_n\) are sometimes called the "interpolation nodes".

Very often the family from which \(\Phi\) is drawn is a so-called "linear space" of functions, or "vector space" of functions. This means a family of functions of the form \(\Phi(x) = c_0\phi_0(x) + \ldots + c_k\phi_k(x)\) where \(\phi_0, \ldots, \phi_k\) are specific functions. Initially we will be concerned with the case where \(k = n\), where we have exactly the same number of parameters (coefficients) as datapoints. Other cases, are, however important: When \(k\) is much less than \(n\) we have cannot interpolate and in that case we usually settle for the "least-squares" approximation of the data; when \(k\) is more than \(n\) there are many interpolants and we can use the extra parameters to choose the most desirable interpolant from some viewpoint. Examples of a family of functions which is not a linear space:

a) rational functions: \(\Phi(x) = \frac{a_0 + a_1x + \ldots + a_mx^m}{b_0 + b_1x + \ldots + b_nx^n}\)

b) exponential functions: \(\Phi(x) = a_0e^{b_0x} + a_1e^{b_1x} + \ldots + a_ke^{b_kx}\)

When \(\Phi\) is drawn from a linear space the interpolation problem reduces to the solution of a system of linear equations:
\[
\begin{align*}
y_0 &= c_0\phi_0(x_0) + \ldots + c_k\phi_k(x_0) \\
y_1 &= c_0\phi_0(x_1) + \ldots + c_k\phi_k(x_1) \\
&\vdots \\
y_n &= c_0\phi_0(x_n) + \ldots + c_k\phi_k(x_n)
\end{align*}
\]

In matrix form, we have
\[
\begin{bmatrix}
y_0 \\
y_1 \\
\vdots \\
y_n
\end{bmatrix} =
\begin{bmatrix}
\phi_0(x_0) & \phi_0(x_1) & \cdots & \phi_0(x_n) \\
\phi_1(x_0) & \phi_1(x_1) & \cdots & \phi_1(x_n) \\
\vdots & \vdots & \ddots & \vdots \\
\phi_k(x_0) & \phi_k(x_1) & \cdots & \phi_k(x_n)
\end{bmatrix}
\begin{bmatrix}
c_0 \\
c_1 \\
\vdots \\
c_k
\end{bmatrix}
\]
where the matrix \(A\) is given by \((A)_{ij} = \phi_j(x_i)\). Thus, interpolation is a fairly easy problem in this case.

Cardinal functions: In the case where \(\Phi(x)\) is drawn from a linear space with the same number of coefficients as datapoints, it is generally the case that the interpolant exists and is unique for any set of datapoints, or if that is not true, the set of datapoints is
restricted to those cases for which the interpolant exists and is unique. Whenever the interpolant exists and is unique, we can construct the so-called cardinal functions for the problem. These are functions \( l_i(x) \) that depend on the interpolation nodes \( \{x_0, \ldots, x_n\} \) and are defined so as to interpolate data which is zero at all the nodes except that \( l_i(x) = 1 \) at node \( i \). Thus we have \( l_i(x_j) = 0 \) for all nodes \( x_0, x_1, \ldots, x_n \) except \( l_i(x_i) = 1 \). There is one cardinal function corresponding to each node. If we compute all the individual cardinal functions \( l_i(x) \) for a given interpolation problem, the interpolant of the original data \( (x_i, y_i) \), \( i = 0, \ldots, n \) is given by

\[
\Phi(x) = y_0 l_0(x) + \ldots + y_n l_n(x)
\]

(this is easy to check: If we set \( x = x_i \) on both sides, then only \( l_i(x) \) is nonzero at \( x_i \) and is equal to 1, so that \( \Phi(x_i) = y_i l_i(x_i) = y_i \)). The importance of cardinal functions is that, once computed for a given problem, it is easy to write down the interpolant for any given set of \( y \) values; moreover the cardinal functions show precisely how the interpolant in a given problem depends on, or is sensitive to, the \( y \) values: If \( y_i \) changes by \( \Delta y_i \) then the interpolant \( \Phi(x) \) changes by \( \Delta y_i l_i(x) \).

Cardinal functions are associated with the name of the mathematician Lagrange, which prompts the notation \( l_i(x) \).

Polynomial interpolation: In polynomial interpolation the function \( \Phi(x) \) is given by a polynomial \( p(x) = a_0 + a_1 x + \ldots + a_n x^n \) of degree \( n \), with as many coefficients as datapoints. While polynomial interpolants can be easily calculated by setting up the linear equations above, this space of functions has special features and properties that make the interpolant particularly easy to calculate and the importance of polynomials justifies a detailed analysis of the properties of the interpolant. The first nice facts about polynomial interpolation is that it is easy to show that the interpolant is unique (and so exists) for any set of data and it is very easy to write down a formula for the associated cardinal functions. We do this next.

Uniqueness: Polynomials are a linear space. If we consider the linear system \( y = Ac \) as described above for the coefficients \( c_0, c_1, \ldots, c_n \) of \( 1, x, x^2, \ldots, x^n \) respectively then we know from linear algebra that solutions are unique if \( y = 0 \) implies that \( c = 0 \) must hold. But if \( y = 0 \) then we are looking for a polynomial of degree \( n \) which is zero at \( n + 1 \) points \( x_0, \ldots, x_n \). This is only possible for the zero polynomial, in which case \( c = 0 \) must follow. We also know that for square linear systems, uniqueness implies existence, so polynomial interpolants are always unique and always exist.

Lagrange cardinal polynomials: We can just write down the cardinal functions for polynomial interpolation. We have

\[
l_i(x) = \frac{(x-x_0)(x-x_1) \cdots (x-x_{i-1})(x-x_{i+1}) \cdots (x-x_n)}{(x_i-x_0)(x_i-x_1) \cdots (x_i-x_{i-1})(x_i-x_{i+1}) \cdots (x_i-x_n)}
\]

Note that the numerator is zero whenever \( x = x_0, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \) and when \( x = x_i \) the numerator and denominator are identical and so \( l_i(x_i) = 1 \). Here is a function in MATLAB for computing Lagrange cardinal polynomials:

```matlab
function L=card(x,i,X)
%function L=card(x,i,X)

```
%X is the array of interpolation nodes
n=length(X);
K=[1:(i-1) (i+1):n];
L=1;
for j=K
    L=L.*(x-X(j))/(X(i)-X(j));
end

It is important to note the behavior of these cardinal functions when there are a large number of nodes. One observes that the values of these functions can be much larger than one, and that these large values can occur far from the node corresponding to the cardinal function. This means that a small change in a given y value can produce a large change, far away from that y value, in the interpolant. For that reason - the extreme sensitivity of high-degree polynomial interpolants to small changes or errors in the data - we seldom interpolate with high-degree polynomials even under conditions when the theory tells us that they behave well (and there are situations in which the theory proves that they behave terribly!)

Terminology: The **Lagrange form of the interpolating polynomial** refers to the formula $p(x) = y_0\ell_0(x) + ... + y_n\ell_n(x)$ for the interpolating polynomial, where the $\ell_i(x)$ are the Lagrange cardinal polynomials defined above.

Newton form of the interpolating polynomial:

Now we present another approach to the construction of the interpolating polynomial which results in the so-called Newton form. In the process we introduce the idea of divided differences, sort of a discrete version of derivatives.

In this development we consider the $y_i$ as coming from a function $f(x)$, so that $y_i = f(x_i)$.

Given the datapoints $(x_i, f(x_i))$, $i = 0, ..., n$, define a sequence of polynomials as follows:

$p_0(x) = a_0$ where $a_0 = f(x_0)$.

Thus $p_0(x)$ is the polynomial of degree 0 that interpolates the single datapoint $(x_0, f(x_0))$

Now set $p_1(x) = p_0(x) + a_1(x - x_0)$ and determine the value of $a_1$ from the condition $f(x_1) = p_1(x_1) = p_0(x_1) + a_1(x_1 - x_0)$. Note two important things: First $p_1(x)$ automatically maintains the interpolation property of $p_0(x)$, namely $p_1(x_0) = p_0(x_0) = y_0$. Second, it is clearly possible to solve for $a_1$, indeed we find $a_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$. This is not surprising as $a_1$ is just the slope of the line connecting the first two datapoints. However, we are not particularly concerned with the value of $a_1$, just the form of $p_1(x)$ and the fact that $a_1$ can be calculated. So now we have

$p_1(x) = p_0(x) + a_1(x - x_0)$ is the polynomial of degree 1 that interpolates
Continuing, define
\[ p_2(x) = p_1(x) + a_2(x-x_0)(x-x_1) \] and choose \( a_2 \) so that \( p_2(x_2) = f(x_2) \). Then
\[ p_2(x) = p_1(x) + a_2(x-x_0)(x-x_1) \] is the polynomial of degree 2 that interpolates
\((x_i,f(x_i)), i = 0,1,2\)

Finally we arrive at \( p_n(x) = p_{n-1}(x) + a_n(x-x_0) \cdot \cdots (x-x_{n-1}) \) where \( a_n \) is chosen so that
\( p_n(x_n) = f(x_n) \). The end result is our interpolating polynomial \( p_n(x) \) which we can write in the form
\[ p_n(x) = a_0 + a_1(x-x_0) + a_2(x-x_0)(x-x_1) + \cdots + a_n(x-x_0) \cdot \cdots (x-x_{n-1}) \]

Any polynomial written in a form like this is said to be in Newton form, where the points \( x_0,..,x_{n-1} \) are called the "centers". So if we write
\[ p(x) = 2 + 3(x-1) - (x-1)(x+1) + 7(x-1)(x+1)x \] then \( p(x) \) is written in Newton form, with centers \( x = 1,-1,0 \)

Of course the specific \( p_n(x) \) above comes from the interpolation problem and so the \( a_k \) have particular significance. Each \( a_k \) depends on the data \((x_i,f(x_i)), i = 0,..,k \). We will indicate this dependence using the specific notation \( a_k \equiv f[x_0,..,x_k] \) and can make the following definition and characterization:

Definition: \( a_k \equiv f[x_0,..,x_k] \) is the leading (i.e. highest-power) coefficient in the polynomial \( p_k(x) \) that interpolates the data \((x_i,f(x_i)), i = 0,..,k \).

The quantity \( f[x_0,..,x_k] \) is referred to as a "\( k \)th order divided difference" for reasons that will become clear later. So, for instance, if we have a function \( f(x) \) and we write \( f[-1,2,1,-3] \) then this is a 3rd order divided difference and \( f[-1,2,1,-3] \) represents the coefficient of \( x^3 \) in the cubic polynomial that interpolates the data \((x_i,f(x_i))\) at the interpolation nodes \( x = -1,2,1,-3 \). Note as a special case that \( f[x_i] = f(x_i) \).

In terms of this notation, we can now write the interpolating polynomial as:
\[ p_n(x) = f[x_0] + f[x_0,x_1](x-x_0) + f[x_0,x_1,x_2](x-x_0)(x-x_1) + \cdots + f[x_0,..,x_n](x-x_0) \cdot \cdots (x-x_{n-1}) \]

The formula above is called the **Newton form of the interpolating polynomial**.

Recursive formula for divided differences: A divided difference of order \( k + 1 \) can be expressed as a first order divided difference of two divided differences of order \( k \). Specifically, we have the following recursive formula:
\[ f[x_0,..,x_{k+1}] = \frac{f[x_1,..,x_{k+1}] - f[x_0,..,x_k]}{x_{k+1} - x_0} \]

Divided differences can be rapidly and efficiently calculated using the idea of the divided difference table. Given nodes \( x_0,..,x_n \), we begin with the zeroth order divided differences \( f[x_i] = f(x_i) \) and then successively compute first order, second order, ..., \( n \)th order divided differences among consecutive points using the recursion above.

Here’s what the table looks like:
Note that, working backwards from the right, we can see that in order to compute \( f[x_0,...,x_n] \) recursively, we need the values of all the divided differences in the table to its left. Of course, we compute from left to right, beginning with the values that we know, namely \( f[x_k] = f(x_k) \).

Here is a MATLAB function for computing a divided difference table from data \((x, y)\) where \(x\) and \(y\) are columns.

```matlab
function D = divdiff(x, y);
% x and y are columns
m = length(y);
D(:, 1) = y;
for j = 2:m
    D(1:m-j+1, j) = (D(2:m-j+2, j-1) - D(1:m-j+1, j-1))./(x(j:m) - x(1:m-j+1));
end
```

The array \( D \) computed above arranges the divided differences in the upper triangular part of the array.

\[
\begin{array}{cccccc}
  x_i & f[x_i] & f[x_i,x_{i+1}] & f[x_i,x_{i+1},x_{i+2}] & \ldots & f[x_i,x_{i+1},x_{i+n-1}] & f[x_0,...,x_n] \\
  x_0 & f(x_0) & \downarrow & f[x_0,x_1] & \downarrow & f[x_0,x_1,x_2] & \ldots & f[x_0,x_1,x_{n-1}] & f[x_0,...,x_n] \\
  x_1 & f(x_1) & \downarrow & f[x_1,x_2] & \downarrow & f[x_1,x_2,x_3] & \ldots & f[x_1,...,x_n] & \uparrow \\
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \uparrow & \vdots & \uparrow \\
  x_{n-1} & f(x_{n-1}) & \downarrow & f[x_{n-1},x_n] & \uparrow & \vdots & \vdots & \vdots & \vdots \\
  x_n & f(x_n) & \uparrow & \uparrow & \vdots & \vdots & \vdots & \vdots & \vdots \\
\end{array}
\]
Once we have computed the divided difference table, the coefficients of \( p_n(x) \) in Newton form can be read off or obtained from the top diagonal in the divided difference table (which is the top row in D). Note however, that other Newton forms of \( p_n(x) \) can be written by proceeding on any path of successively higher-order divided differences beginning from a function value \( f(x_k) \) on the left to the \( n^{th} \) order divided difference \( f[x_0,\ldots,x_n] \) at the far right. This is because a divided difference is independent of the ordering of its arguments, e.g. \( f[1, -2, 3, -1] = f[3, -1, 1, -2] \). An example is further below.

Evaluating \( p_n(x) \) in Newton form using recursion/nested multiplication: A polynomial in Newton form can be efficiently evaluated by adapting our previous Horner’s algorithm for polynomials in standard power form. Namely, if
\[
p_n(x) = a_0 + a_1(x-x_0) + a_2(x-x_0)(x-x_1) + \ldots + a_n(x-x_0) \cdots (x-x_{n-1}) \]
we write
\[
p_n(x) = a_0 + (x-x_0)[a_1 + (x-x_1)[a_2 + \ldots + (x-x_{n-2})[a_{n-1} + (x-x_{n-1})[a_n]]]]
\]
and arrange the calculations from the innermost parentheses to the outermost. In MATLAB we can write:

```matlab
function y=newtpoly(x,c,X)
%centers are in X, c is the coefficient array
n=length(c);
y=c(n)*ones(size(x));
for k=n-1:-1:1
    y=c(k)+(x-X(k)).*y;
end
```

**Example:**

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( f[x_i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>35</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>93</td>
</tr>
<tr>
<td>83</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>259</td>
</tr>
</tbody>
</table>

Two different forms of the same interpolating polynomial \( p(x) \) can be read off, the first using the top diagonal and the second on the path indicated by the arrows.

\[
p(x) = 1 + 8(x-0) + 3(x-0)(x-1) + 1(x-0)(x-1)(x-2) + 0(x-0)(x-1)(x-2)(x-4) = 1 + 7x + x^3
\]

\[
p(x) = 93 + 35(x-4) + 12(x-4)(x-2) + 1(x-4)(x-2)(x-6) = 1 + 7x + x^3
\]
This second polynomial represents the form
\[ p(x) = f[4] + f[4,2](x - 4) + f[4,2,6](x - 4)(x - 2) + f[4,2,6,1](x - 4)(x - 2)(x - 6) + f[4,2,6,1,0](x - 4)(x - 2)(x - 6)(x - 1) \]

where, for instance, the number 12 in the table represents \( f[2,4,6] = f[4,2,6] \).

As calculated in MATLAB, the divided difference table above looks like:
```matlab
>> x=[0 1 2 4 6]; y=[1 9 23 93 259];
>> divdiff(x,y)
ans =
    1     8     3   1   0
    9    14     7   1   0
   23    35    12   0   0
   93    83     0   0   0
  259   259     0   0   0
```

Proof of the recursion formula for divided differences:
We want to prove
\[ f[x_0,\ldots,x_{k+1}] = \frac{f[x_1,\ldots,x_{k+1}] - f[x_0,\ldots,x_k]}{x_{k+1} - x_0} \]

Let \( p(x) \) represent the polynomial of degree \( k \) that interpolates \( f(x) \) at \( x_1,\ldots,x_{k+1} \)

Let \( Q(x) \) represent the polynomial of degree \( k \) that interpolates \( f(x) \) at \( x_0,\ldots,x_k \)

Consider now the polynomial \( p(x) \) defined by
\[ p(x) = \frac{(x-x_0)P(x) + (x_{k+1} - x)Q(x)}{x_{k+1} - x_0} \]

When \( x = x_0 \) it is easy to verify that \( p(x_0) = Q(x_0) = f(x_0) \)

When \( x = x_{k+1} \) it is easy to verify that \( p(x_{k+1}) = Q(x_{k+1}) = f(x_{k+1}) \)

When \( x = x_j \) for \( 0 < j < k + 1 \), we have
\[ p(x_j) = \frac{(x_j-x_0)P(x_j) + (x_{k+1} - x_j)Q(x_j)}{x_{k+1} - x_0} = \frac{(x_j-x_0)f(x_j) + (x_{k+1} - x_j)f(x_j)}{x_{k+1} - x_0} = f(x_j) \]

Thus \( p(x) \) interpolates \( f(x) \) at \( x_0,\ldots,x_{k+1} \). It’s leading coefficient is \( f[x_0,\ldots,x_{k+1}] \) by definition. On the other hand, considering that \( p(x) = \frac{(x-x_0)P(x) + (x_{k+1} - x)Q(x)}{x_{k+1} - x_0} \) we can see that the leading coefficient (the coefficient of \( x^{k+1} \) ) on the right hand side is \( \frac{f[x_1,\ldots,x_{k+1}] - f[x_0,\ldots,x_k]}{x_{k+1} - x_0} \) and setting these two equal gives the result
\[ f[x_0,\ldots,x_{k+1}] = \frac{f[x_1,\ldots,x_{k+1}] - f[x_0,\ldots,x_k]}{x_{k+1} - x_0} \].

Adding an additional interpolation point: If we calculated a divided difference table and wish to add another datapoint that is to be interpolated, we can just add that point to the bottom of the list and compute the upward diagonal of divided differences. In the example, if we wish to add the point \((x, y) = (3, 67)\) we would compute (follow the arrows)
and then using the top diagonal (which now has one additional number)

\[
p(x) = 1 + 8(x - 0) + 3(x - 0)(x - 1) + 1(x - 0)(x - 1)(x - 2) + 0(x - 0)(x - 1)(x - 2)(x - 4) + 1(x - 0)(x - 1)(x - 2)(x - 4)(x - 6)
\]

It is possible to show that adding a new point in this way requires computations essentially equivalent to computing

\[
a_5 = \frac{f(x_5) - p_4(x_5)}{(x_5 - 0)(x_5 - 1)(x_5 - 2)(x_5 - 4)(x_5 - 6)}
\]

so the divided difference table doesn't actually save any computational work above that involved in the original Newton form development. However it does display the computations in a very clear structure.

Divided differences and derivatives:

In what follows we assume that the function \( f(x) \) has as many continuous derivatives as we need for the result. We prove the following:

\[
f[x_0, \ldots, x_n] = \frac{1}{n!} f^{(n)}(\xi) \text{ where } \xi \text{ is a point between the largest and smallest of the } x_i
\]

Proof: Let \( p(x) \) interpolate the function \( f(x) \) at \( x = x_0, \ldots, x_n \). Remember that the leading coefficient of \( p(x) \) is given by \( f[x_0, \ldots, x_n] \), i.e. \( p(x) = \ldots + f[x_0, \ldots, x_n]x^n \). Consider now the "error" \( e(x) = f(x) - p(x) \). We have \( e(x_i) = 0 \), \( i = 0, \ldots, n \). Now remember Rolle's theorem: between every adjacent pair of zeros of a function, the derivative of the function must be zero somewhere between the adjacent zeros. Since \( e(x) \) is zero at \( n + 1 \) points, \( e'(x) \) is zero at (at least) \( n \) points; applying the same argument to \( e'(x) \), we have that \( e''(x) \) is zero at \( n - 1 \) points, and so on, until we arrive at the fact that \( e^{(n)}(x) \) is zero at least one point that lies between the largest and smallest values of the \( x_i \); we call this point \( \xi \). Now \( 0 = e^{(n)}(\xi) = f^{(n)}(\xi) - p^{(n)}(\xi) = f^{(n)}(\xi) - nf[x_0, \ldots, x_n] \) and we obtain \( f[x_0, \ldots, x_n] = \frac{1}{n!} f^{(n)}(\xi) \).

The error in polynomial interpolation:
Here we study \( e(x) = f(x) - p(x) \) as defined above. Let \( t \) be some value of \( x \) other than one of the \( x_i \). Let \( P(x) \) be the polynomial that interpolates \( f(x) \) at \( x = x_0, \ldots, x_n, t \) (i.e. we are adding on the point \( x = t \)). Then we know that

\[
P(x) = p(x) + f[x_0, \ldots, x_n, t](x - x_0) \cdot \cdot \cdot (x - x_n).
\]

We then have

\[
f(t) = P(t) = p(t) + f[x_0, \ldots, x_n, t](t - x_0) \cdot \cdot \cdot (t - x_n) \text{ and so}
\]

\[
e(t) = f(t) - p(t) = f[x_0, \ldots, x_n, t](t - x_0) \cdot \cdot \cdot (t - x_n).
\]

Finally, using the previous result relating divided differences to derivatives, we obtain the error formula:

\[
e(t) = \frac{1}{(n + 1)!} f^{(n+1)}(\xi)(t - x_0) \cdot \cdot \cdot (t - x_n)
\]

Note that this formula applies now even when \( t \) is one of the \( x_i \).

We can replace \( t \) by \( x \) since \( t \) is an arbitrary value of \( x \) and so obtain

\[
f(x) - p(x) = \frac{1}{(n + 1)!} f^{(n+1)}(\xi)(x - x_0) \cdot \cdot \cdot (x - x_n)
\]

where \( \xi \) is a value somewhere between the largest and smallest of the values \( \{x_i\}, i = 0, \ldots, n \) and \( x \). Note that \( x \) may be outside the interval containing the interpolation point. We use the term "extrapolation" when \( p(x) \) is used to estimate the value of \( f(x) \) outside the original interval containing the interpolation points.

Polynomial interpolation and Taylor polynomials:

Given that \( p_n(x) \) interpolates the function \( f(x) \) at \( x = x_0, \ldots, x_n \), consider the formula

\[
f(x) = p_n(x) + \frac{1}{(n + 1)!} f^{(n+1)}(\xi)(x - x_0) \cdot \cdot \cdot (x - x_n)
\]

If this reminds you of Taylor’s theorem

\[
f(x) = T_n(x) + \frac{1}{(n + 1)!} f^{(n+1)}(\xi)(x - a)^{n+1}
\]

this is not an accident. The interpolating polynomial

\[
p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \ldots + f[x_0, \ldots, x_n](x - x_0) \cdot \cdot \cdot (x - x_{n-1})
\]

represents a "discrete" Taylor polynomial, with divided differences taking the place of derivatives. One can show that if we consider a limit where all the \( x_i \) converge to a single point, say \( x_i \to a \) then one can show that \( p_n(x) \to T_n(x) \); for after all, we already know that \( f[x_0, \ldots, x_k] = \frac{1}{k!} f^{(k)}(\xi) \to \frac{1}{k!} f^{(k)}(a) \) in this case, and

\[
(x - x_0) \cdot \cdot \cdot (x - x_{k-1}) \to (x - a)^k.
\]

Taylor polynomials are a case where we interpolate derivatives of a function at a single point, rather than function values at several points and they are related by the limiting process described above. Note that approximating a function \( f(x) \) by \( T_n(x) \) corresponds to extrapolation when we consider approximating \( f(x) \) by \( p_n(x) \), so that extrapolation of polynomial interpolants can be seen to sometimes provide good approximations. But in general this is considered dangerous, especially when the degree of the polynomial is large. Even when extrapolation can be theoretically shown to behave well, the calculation of divided differences \( f[x_0, \ldots, x_k] \) over small intervals inherently involves a good deal of roundoff error, producing a huge error in \( f[x_0, \ldots, x_k](x - x_0) \cdot \cdot \cdot (x - x_{k-1}) \) if \( x \) far from where the \( x_i \) are located.

Estimating the error in polynomial interpolation: We present two applications in the linked document.