Lecture 14

Interpolation I

1 Introduction

A common problem faced in engineering is that we have some physical system or process with input *x* and output *y*. We assume there is a functional relation between input and output of the form $y=f(x)$, but we don't know what *f* is. For *n* particular inputs $x_1<x_2<\cdots<x_n$ we experimentally determine the corresponding outputs $y_i=f(x_i)$. From these we wish to estimate the output *y* for an arbitrary input *x* where $x_1\le x\le x_n$. This is the problem of *interpolation*.

If the experiment is easy to perform then we could just directly measure $y=f(x)$, but typically this is not practical. Experimental determination of an input-output relation is often difficult, expensive and time consuming. Or, $y=f(x)$ may represent the result of a complex numerical simulation that is not practical to perform every time we have a new *x* value. In either case the only practical solution may be to estimate *y* by interpolating known values.

To illustrate various interpolation methods, we will use the example of

$$y=f(x)=3e^{-x}\sin x$$

sampled at $x=0,1,2,3,4,5$. These samples are the black dots in Fig. 1.

2 Nearest neighbor or "staircase" interpolation

A simple interpolation scheme for estimating $f(x)$ is to find the sample value x_i that is nearest to *x* and then assume $y=f(x)=y_i$. This is called *nearest neighbor* interpolation, and we can describe it as follows.

```
Nearest-neighboor interpolation
  Find the value of i that minimizes the distance $|x-x_i|$
  Set $y=y_i$
```

The interpolation will be constant near a sample point. As we move towards another sample point the interpolation will discontinuously jump to a new value. This produces the "staircase" appearance of the solid line in Fig. 1. Except for the special case where all y_i are equal (a constant function), nearest neighbor interpolation produces a discontinuous output.

Although this may not look like a "realistic" function, the fact is that without more information than just the sample points we don't have any basis to declare one interpolation better than another. Digital systems often produce output with piecewise-constant, discrete levels such as this. On the other hand, if we know that the function $f(x)$ is the result of some continuous physical process, then a nearest-neighbor interpolation is almost certainly a poor representation of the "true" function.

3 Linear interpolation

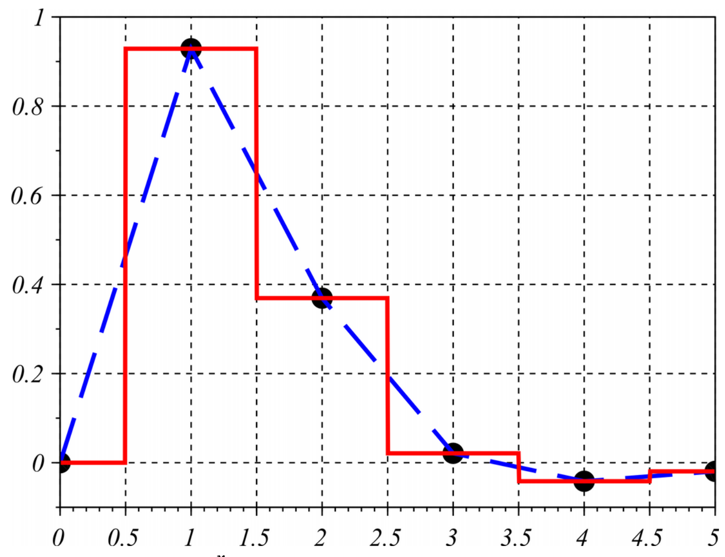

Fig. 1 circles: samples of $3e^{-x}\sin x$ *, solid line: nearest-neighbor interpolation; dashed line, linear interpolation*

An obvious and easy way to interpolate data is to "connect the dots" with straight lines. This produces a continuous interpolation (Fig. 1) but which has "kinks" at the sample points where the slope is discontinuous. The algorithm is

Linear interpolation

Find k such that $x_k < x < x_{k+1}$

Set $y = y_k + \dfrac{y_{k+1} - y_k}{x_{k+1} - x_k}(x - x_k)$

If the samples are closely spaced, linear interpolation works quite well. In fact it's used by default for the `plot()` routine in Scilab/Matlab. However, when sampling is sparse (as in Fig. 1), linear interpolation is unlikely to give an accurate representation of a "smooth" function. This motivates us to investigate more powerful interpolation methods.

4 Polynomial interpolation

Through two points $(x_1, y_1), (x_2, y_2)$ we can draw a unique line, a 1st order polynomial. Through three points $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ we can draw a unique parabola, a 2nd order polynomial. In general, through *n* points $(x_i, y_i), i = 1, 2, \ldots, n$ we can draw a unique polynomial of order $(n-1)$. Although this polynomial is unique, there are different ways to represent and derive it. We start with the most obvious approach, the so-called monomial basis.

4.1 Monomial basis

A term in a single power of *x*, such as x^k, is called a *monomial*. Adding two or more of these together produces a *polynomial*. A polynomial of order $(n-1)$ with arbitrary coefficients is a linear combination of monomials:

$$y=p(x)=c_1+c_2x+c_3x^2+\cdots+c_nx^{n-1}$$

We want this polynomial to pass through our samples $(x_i, y_i), i=1,2,\ldots,n$. We therefore require

$$\begin{gathered}c_1+c_2x_1+c_3x_1^2+\cdots+c_nx_1^{n-1}=y_1\\c_1+c_2x_2+c_3x_2^2+\cdots+c_nx_2^{n-1}=y_2\\\vdots\\c_1+c_2x_n+c_3x_n^2+\cdots+c_nx_n^{n-1}=y_n\end{gathered}$$

which has the form of *n* equations in *n* unknown coefficients c_i . We can express this as the linear system

$$\mathbf{A}\,\mathbf{c}=\mathbf{y}$$

where

$$\mathbf{A}=\begin{pmatrix}1 & x_1 & x_1^2 & \cdots & x_1^{n-1}\\1 & x_2 & x_2^2 & \cdots & x_2^{n-1}\\\vdots & \vdots & \vdots & \vdots & \vdots\\1 & x_n & x_n^2 & \cdots & x_n^{n-1}\end{pmatrix}\ ,\ \mathbf{c}=\begin{pmatrix}c_1\\c_2\\\vdots\\c_n\end{pmatrix}\ ,\ \mathbf{y}=\begin{pmatrix}y_1\\y_2\\\vdots\\y_n\end{pmatrix} \tag{1}$$

A matrix of the form **A**, in which each column contains the sample values to some common power is called a *Vandermonde* matrix. The coefficient vector **c** is easily calculated in Scilab/Matlab as

```
n = length(x);
A = ones(x);
for k=1:n-1
  A = [A,x.^k];
end
c = A\y;
```

Here we've assumed that the vector **x** is a column vector. If it is a row vector replace it by the transpose (`x'`). Likewise for the vector **y**.

Example 1:Suppose $\mathbf{x}^T=[0,1,2]$ *and* $\mathbf{y}^T=[2,1,3]$. *The coefficients of the 2nd order polynomial that passes through these points are found from*

$$\begin{pmatrix}1 & 0 & 0\\1 & 1 & 1\\1 & 2 & 4\end{pmatrix}\begin{pmatrix}c_1\\c_2\\c_3\end{pmatrix}=\begin{pmatrix}2\\1\\3\end{pmatrix}$$

Solving the system we obtain

$$\mathbf{c}=\begin{pmatrix}2\\-\frac{5}{2}\\\frac{3}{2}\end{pmatrix}$$

so the interpolating polynomial is

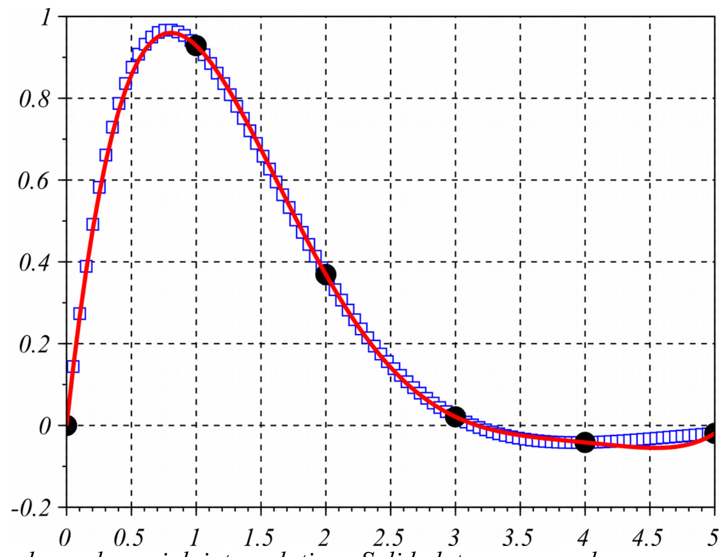

Fig. 2 5th order polynomial interpolation. Solid dots are samples; squares are actual function values (for comparison); line is interpolation.

$$y = 2 - \frac{5}{2}x + \frac{3}{2}x^2$$

A Scilab program to interpolate n samples appears in Appendix 2 as `interpMonomial()`. Applying this to the sample data shown in Fig. 1 produces the results shown in Fig. 2. The resulting interpolation gives a good representation of the underlying function except near $x=4.5$.

Numerically that is all there is to polynomial interpolation. However, it does require the solution of a linear system which for a high-order polynomial must be done numerically. The monomial approach starts having problems for very high-order polynomials due to round-off error. We'll come back to this.

In additional to numerical considerations, there are times when we would like to be able to write the interpolating polynomial directly, without solving a linear system. This is particularly true when we use polynomial interpolation as part of some numerical method. In this case we don't know what the x and y values are because we are developing a method that can be applied to any data. Therefore we want to express the interpolating polynomial coefficients as some algebraic function of the sample points. There are two classic way to do this: Lagrange polynomials and Newton polynomials.

4.2 Lagrange polynomials

Through n points we can pass a polynomial of order $n-1$. Lagrange polynomials are an ingenious way to write this polynomial down by inspection. Here's the idea.

Suppose we have three data points: $(x_1, 1), (x_2, 0), (x_3, 0)$. There is a unique second-order polynomial $p(x)$ which interpolates these data. Since $p(x_2)=0$, x_2 is a root of the polynomial. That means the polynomial must have a factor $(x-x_2)$. Likewise, $p(x_3)=0$ so it also has a factor $(x-x_3)$. But the two factors $(x-x_2)(x-x_3)$ by themselves form a 2nd order polynomial $x^2-(x_2+x_3)x+x_2x_3$. Therefore to within a multiplicative constant $p(x)$ must simply be the product $(x-x_2)(x-x_3)$, that is

$$p(x)=c(x-x_2)(x-x_3)$$

The constant is fixed by the third constraint $p(x_1)=1$ to give us

$$p(x)=\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}$$

From the form of this quadratic it is immediately clear that $p(x_1)=1$ and $p(x_2)=p(x_3)=0$. We'll call this

$$L_1(x)=\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}$$

and $L_1(x_1)=1, L_1(x_2)=0, L_1(x_3)=0$. Now consider the data points $(x_1,0),(x_2,1),(x_3,0)$. By the same logic the interpolating polynomial must be

$$L_2(x)=\frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}$$

and $L_2(x_1)=0, L_2(x_2)=1, L_2(x_3)=0$. Finally consider the data points $(x_1,0),(x_2,0),(x_3,1)$. These are interpolated by

$$L_3(x)=\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}$$

and $L_3(x_1)=0, L_3(x_2)=0, L_3(x_3)=1$.

Now suppose we have three data points $(x_1,y_1),(x_2,y_2),(x_3,y_3)$. The claim is that the second-order interpolating polynomial is

$$p(x)=y_1L_1(x)+y_2L_2(x)+y_3L_3(x)$$

Since L_1, L_2, L_3 are each second-order, a linear combination of them is also. We can verify that it interpolates our three data points by simply calculating $p(x_1), p(x_2), p(x_3)$. For example

$$p(x_2)=y_1L_1(x_2)+y_2L_2(x_2)+y_3L_3(x_2)=y_1(0)+y_2(1)+y_3(0)=y_2$$

The Lagrange polynomial is not in the monomial form, but if we expand out the terms we will (to within round-off error) get the same result we obtain with the monomial-basis method.

Example 2: Suppose as in Example 1 $\mathbf{x}^T=[0,1,2]$ *and* $\mathbf{y}^T=[2,1,3]$. *We have*

$$L_1(x)=\frac{(x-1)(x-2)}{(0-1)(0-2)}\ ,\ L_2(x)=\frac{(x-0)(x-2)}{(1-0)(1-2)}\ ,\ L_3(x)=\frac{(x-0)(x-1)}{(2-0)(2-1)}$$

so

$$y=\frac{2}{2}(x-1)(x-2)+\frac{1}{-1}x(x-2)+\frac{3}{2}x(x-1)$$

Expanding out the terms and collecting like powers we obtain

$$y=2-\frac{3}{2}x+\frac{3}{2}x^2$$

which is the result we obtained in Example 1.

In general, if we have *n* sample points $(x_i, y_i), i=1,2,\ldots,n$ we form the *n* polynomials

$$L_k(x)=\prod_{i\neq k}\frac{x-x_i}{x_k-x_i} \tag{2}$$

for $k=1,2,\ldots,n$, where the Π symbol signifies the product of the $n-1$ terms indicated. The interpolating polynomial is then

$$y=p(x)=\sum_{k=1}^{n} y_k L_k(x) \tag{3}$$

Lagrange polynomials are very useful analytically since they can be written down by inspection without solving any equations. They also have good numerical properties. A Scilab program to interpolate *n* samples using the Lagrange method appears in Appendix 2 as `interpLagrange()`. It's important to remember that there is a unique polynomial of order $n-1$ which interpolates a given *n* points. Whatever method we use to compute this *must* produce the same polynomial (to within our numerical limitations).

4.3 Newton polynomials

Newton polynomials are yet another way to obtain the $n-1$ order polynomial that interpolates a given *n* points. Suppose we have a single sample (x_1, y_1). The zeroth-order polynomial (a constant function) passing through this point is

$$p_0(x)=c_1$$

where $c_1=y_1$. Now suppose we have an additional point (x_2, y_2). The polynomial that interpolates these two points will now be first order. Newton's idea was to write it in the form

$$p_1(x)=c_1+c_2(x-x_1)$$

because this guarantees that $p_1(x_1)=c_1=p_0(x_1)=y_1$, so our coefficient c_1 is unchanged. We need only calculate a single "new" coefficient c_2 from

$$p_1(x_2)=y_2=c_1+c_2(x_2-x_1)$$

This gives us

$$c_2=\frac{y_2-c_1}{x_2-x_1}=\frac{y_2-y_1}{x_2-x_1}$$

so

$$p_1(x)=y_1+\frac{y_2-y_1}{x_2-x_1}(x-x_1)$$

Notice that this looks somewhat like $f(x)=f(x_1)+f'(x_1)(x-x_1)$ where $f'(x_1)$ is approximated by "change in *y* over change in *x*." That is it has the form of a *discrete Taylor* *series*.

Now suppose we add a third point (x_3, y_3). Our interpolating polynomial will now need to be quadratic, but we want to write it in a way that "preserves" the fit we've already obtained for the first two points. Therefore we write

$$p_2(x)=c_1+c_2(x-x_1)+c_3(x-x_1)(x-x_2)$$

which guarantees that $p_2(x_1)=c_1=y_1=p_0(x_1)$ and $p_2(x_2)=c_1+c_2(x_2-x_1)=y_2=p_1(x_2)$. The single new coefficient c_3 is obtained from

$$p_2(x_3)=y_3=c_1+c_2(x_3-x_1)+c_3(x_3-x_1)(x_3-x_2)$$

so that

$$c_3=\frac{y_3-c_1-c_2(x_3-x_1)}{(x_3-x_1)(x_3-x_2)}=\frac{\frac{y_3-y_1}{x_3-x_1}-c_2}{x_3-x_2}=\frac{\frac{y_3-y_1}{x_3-x_1}-\frac{y_2-y_1}{x_2-x_1}}{x_3-x_2}$$

The coefficient c_3 has a form that suggests "difference in the derivative of *y* over difference in *x*" – the form of a second derivative. Our interpolating polynomial

$$p_2(x)=y_1+\frac{y_2-y_1}{x_2-x_1}(x-x_1)+\frac{\frac{y_3-y_1}{x_3-x_1}-\frac{y_2-y_1}{x_2-x_1}}{x_3-x_2}(x-x_1)(x-x_2)$$

is roughly analogous to a second-order Taylor series

$$f(x)=f(x_1)+f'(x_1)(x-x_1)+\frac{1}{2}f''(x_1)(x-x_1)^2$$

We might think of $p_2(x)$ as a second-order discrete Taylor series.

If we now add a fourth data point (x_4, y_4) we will need a third-order interpolating polynomial $p_3(x)$ which we write as

$$p_3(x)=c_1+c_2(x-x_1)+c_3(x-x_1)(x-x_2)+c_4(x-x_1)(x-x_2)(x-x_3)$$

As before all the coefficients except the "new one" will be unchanged, so we need only calculate the single new coefficient c_4.

Suppose instead we skip calculating the polynomials $p_0(x), p_1(x), p_2(x)$ and want to calculate $p_3(x)$ directly from the four data points. The four coefficients are determined by the conditions

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & (x_2-x_1) & 0 & 0 \\ 1 & (x_3-x_1) & (x_3-x_1)(x_3-x_2) & 0 \\ 1 & (x_4-x_1) & (x_4-x_1)(x_4-x_2) & (x_4-x_1)(x_4-x_2)(x_4-x_3) \end{pmatrix}\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}=\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \tag{4}$$

The matrix is lower-triangular, so we can solve it using forward substitution. However, if we solve it using Gauss-Jordan elimination an interesting pattern emerges.

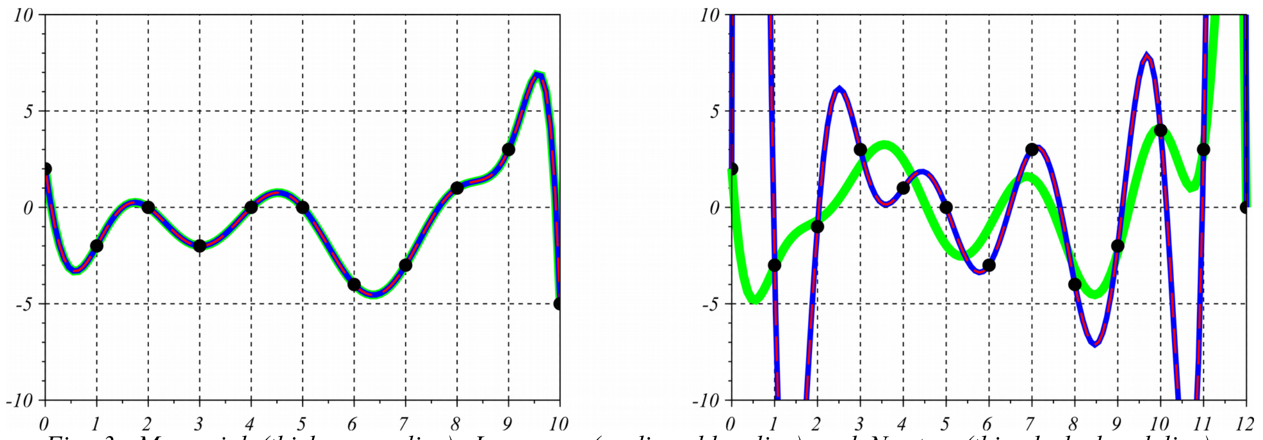

Fig. 3: Monomial (thick green line), Lagrange (medium blue line) and Newton (thin dashed red line) polynomials. At left N=11 point (hece 10th order polynomials). Agreement is good. At right N=13 points (12th order polynomials). The monomial polynomial fails to interpolate the data. The y data values were chosen randomly. The x values are 0,1,2,...,N-1.

The augmented matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & y_1 \\ 1 & (x_2-x_1) & 0 & 0 & y_2 \\ 1 & (x_3-x_1) & (x_3-x_1)(x_3-x_2) & 0 & y_3 \\ 1 & (x_4-x_1) & (x_4-x_1)(x_4-x_2) & (x_4-x_1)(x_4-x_2)(x_4-x_3) & y_4 \end{pmatrix}$$

Subtracting the row 1 from rows 2, 3 and 4

$$\begin{pmatrix} 1 & 0 & 0 & 0 & y_1 \\ 0 & (x_2-x_1) & 0 & 0 & y_2-y_1 \\ 0 & (x_3-x_1) & (x_3-x_1)(x_3-x_2) & 0 & y_3-y_1 \\ 0 & (x_4-x_1) & (x_4-x_1)(x_4-x_2) & (x_4-x_1)(x_4-x_2)(x_4-x_3) & y_4-y_1 \end{pmatrix}$$

Normalizing rows 2, 3 and 4 by the element in the 2nd column

$$\begin{pmatrix} 1 & 0 & 0 & 0 & y_1 \\ 0 & 1 & 0 & 0 & \dfrac{y_2-y_1}{x_2-x_1} \\ 0 & 1 & (x_3-x_2) & 0 & \dfrac{y_3-y_1}{x_3-x_1} \\ 0 & 1 & (x_4-x_2) & (x_4-x_2)(x_4-x_3) & \dfrac{y_4-y_1}{x_4-x_1} \end{pmatrix}$$

Note the cancellation of the term (x_3-x_1) in row 3 and (x_4-x_1) in row 4. Now substract row 2 from rows 3 and 4.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & y_1 \\ 0 & 1 & 0 & 0 & \dfrac{y_2-y_1}{x_2-x_1} \\ 0 & 0 & (x_3-x_2) & 0 & \dfrac{y_3-y_1}{x_3-x_1}-\dfrac{y_2-y_1}{x_2-x_1} \\ 0 & 0 & (x_4-x_2) & (x_4-x_2)(x_4-x_3) & \dfrac{y_4-y_1}{x_4-x_1}-\dfrac{y_2-y_1}{x_2-x_1} \end{pmatrix}$$

Normalizing rows 3 and 4 by the element in the 3rd column

$$\begin{pmatrix} 1 & 0 & 0 & 0 & y_1 \\ 0 & 1 & 0 & 0 & \dfrac{y_2-y_1}{x_2-x_1} \\ 0 & 0 & 1 & 0 & \dfrac{\dfrac{y_3-y_1}{x_3-x_1}-\dfrac{y_2-y_1}{x_2-x_1}}{x_3-x_2} \\ 0 & 0 & 1 & (x_4-x_3) & \dfrac{\dfrac{y_4-y_1}{x_4-x_1}-\dfrac{y_2-y_1}{x_2-x_1}}{x_4-x_2} \end{pmatrix}$$

Note the cancellation in row 4. Subtract row 3 from row 4

$$\begin{pmatrix} 1 & 0 & 0 & 0 & y_1 \\ 0 & 1 & 0 & 0 & \dfrac{y_2-y_1}{x_2-x_1} \\ 0 & 0 & 1 & 0 & \dfrac{\dfrac{y_3-y_1}{x_3-x_1}-\dfrac{y_2-y_1}{x_2-x_1}}{x_3-x_2} \\ 0 & 0 & 0 & (x_4-x_3) & \dfrac{\dfrac{y_4-y_1}{x_4-x_1}-\dfrac{y_2-y_1}{x_2-x_1}}{x_4-x_2}-\dfrac{\dfrac{y_3-y_1}{x_3-x_1}-\dfrac{y_2-y_1}{x_2-x_1}}{x_3-x_2} \end{pmatrix}$$

Normalize row 4 by its element in column 4

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & y_1 \\ 0 & 1 & 0 & 0 & \dfrac{y_2-y_1}{x_2-x_1} \\ 0 & 0 & 1 & 0 & \dfrac{\dfrac{y_3-y_1}{x_3-x_1}-\dfrac{y_2-y_1}{x_2-x_1}}{x_3-x_2} \\ 0 & 0 & 0 & 1 & \dfrac{\dfrac{\dfrac{y_4-y_1}{x_4-x_1}-\dfrac{y_2-y_1}{x_2-x_1}}{x_4-x_2}-\dfrac{\dfrac{y_3-y_1}{x_3-x_1}-\dfrac{y_2-y_1}{x_2-x_1}}{x_3-x_2}}{x_4-x_3} \end{array}\right)$$

The 5th column now contains the coefficients of the Newton polynomial. Following through these steps we arrive at the very compact algorithm for calculating the Newton coefficients.

Calculating Newton coefficients

```
c = y(:); //c is a column vector of the y values
for i=1:n-1
  for j=i+1:n
    c(j) = (c(j)-c(i))/(x(j)-x(i));
  end
end
```

This is used in the Scilab program `interpNewton` which appears in Appendix 2. This interpolates n sample points using the Newton method.

4.4 Numerical considerations and polynomial "wiggle"

In principle, for a given set of data the monomial, Lagrange and Newton interpolating polynomials are identical. They are merely expressed in different "formats." However, numerically the monomial polynomial method suffers from round-off error more than the other two methods. This is illustrated in Fig. 3. In these examples we see that for higher than about a 10th order polynomial the monomial basis fails to accurately interpolate the data. The problem is that the Vandermonde matrix in (1) tends to become nearly singular for large n. For this reason it is not recommended for use except for low-order problems, if at all.

The Lagrange and Newton polynomials are both quite stable numerically. The form of the Newton algorithm allows it to be implemented with fewer floating-point operations, and so it tend to be faster, especially at high order. Numerically, therefore, Newton polynomials are arguably the best choice for polynomial interpolation.

Even without round-off error, high-order polynomials tend to produce unsatisfactory interpolation. The problem is "polynomial wiggle." This can be seen in the right graph of Fig. 3. Near the "middle" x values the interpolation seems reasonable, but at the left and right extremes it oscillates between very large y values – much larger than any of the sample y values. Very slight changes in the y values of the sample points can cause very large changes in this wiggle effect. Although we are guaranteed to be able to find an $(n\text{-}1)$th order polynomial passing through n samples, there is no guarantee that it will be "smooth." In fact for large n it will typically display these large oscillations. We consider alternative interpolation methods in the next lecture.

5 Appendix 1 – Lagrange polynomials for equally spaced points

Suppose we have *n* sample points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with uniformly spaced *x* values given by $x_i = x_1 + (i-1)h$. Define

$$t = \frac{x - x_1}{h}$$

so that

$$x = x_1 + h\,t$$

Written at (t_i, y_i) samples, our data have the form

$$(0, y_1), (1, y_2), \ldots, (n-1, y_n)$$

Suppose we fit a polynomial $y = p(t)$ through these data. Then the polynomial

$$y = q(x) = p(t) = p\left(\frac{x - x_1}{h}\right)$$

interpolates the (x_i, y_i) samples. We limit consideration to the Lagrange basis, as it is the most useful theoretically. Here we list the $p(t)$ polynomials of orders 0,1,2,3,4 which interpolate *n*=1,2,3,4,5 sample points. These can easily be derived from formulas (2) and (3).

For $n=1$

$$p_0(t) = y_1$$

For $n=2$

$$p_1(t) = y_2 t - y_1(t-1)$$

For $n=3$

$$p_2(t) = \frac{1}{2} y_3 t(t-1) - y_2 t(t-2) + \frac{1}{2} y_1 (t-1)(t-2)$$

For $n=4$

$$p_3(t) = \frac{1}{6} y_4 t(t-1)(t-2) - \frac{1}{2} y_3 t(t-1)(t-3) + \frac{1}{2} y_2 t(t-2)(t-3) - \frac{1}{6} y_1 (t-1)(t-2)(t-3)$$

For $n=5$

$$\begin{aligned} p_4(t) = {} & \frac{1}{24} y_5 t(t-1)(t-2)(t-3) - \frac{1}{6} y_4 t(t-1)(t-2)(t-4) + \frac{1}{4} y_3 t(t-1)(t-3)(t-4) \\ & - \frac{1}{6} y_2 t(t-2)(t-3)(t-4) + \frac{1}{24} y_1 (t-1)(t-2)(t-3)(t-4) \end{aligned}$$

6 Appendix 2 – Scilab code

6.1 *Monomial-basis polynomial interpolation*

```
//////////////////////////////////////////////////////////////////////
// interpMonomial.sci
// interpLagrangeCoeff.sci
// 2014-06-25, Scott Hudson, for pedagogic purposes only
// Given n samples x(i),y(i), in the column vectors x,y
// calculate the coefficients c(i) of the (n-1) order
// monomial interpolating polynomial and evaluate at points xp.
//////////////////////////////////////////////////////////////////////
function yp = interpMonomial(x, y, xp)
  n = length(x); //x and y must be column vectors of length n
  A = ones(x); //build up the Vandermonde matrix A
  for k=1:n-1
    A = [A,x.^k]; //each column is a power of the column vector x
  end
  c = A\y; //solve for coefficients
  yp = ones(xp)*c(1); //evaluate polynomial at desired points
  for k=2:n
    yp = yp+c(k)*xp.^(k-1);
  end
endfunction
```

6.2 *Lagrange-basis polynomial interpolation*

```
//////////////////////////////////////////////////////////////////////
// interpLagrange.sci
// 2014-06-25, Scott Hudson, for pedagogic purposes only
// Given n samples x(i),y(i), in the column vectors x,y
// evaluate the Lagrange interpolating polynomial at points xp.
//////////////////////////////////////////////////////////////////////
function yp = interpLagrange(x, y, xp)
  n = length(x);
  yp = zeros(xp);
  for k=1:n //form Lagrange polynomial L_k
    L = 1;
    for i=1:n
      if (i~=k)
        L = L.*(xp-x(i))/(x(k)-x(i));
      end
    end
    yp = yp+y(k)*L;
  end
endfunction
```

6.3 Newton-basis polynomial interpolation

```
//////////////////////////////////////////////////////////////////////
// interpNewton.sci
// 2014-06-25, Scott Hudson, for pedagogic purposes
// Given n-dimensional vectors x and y, compute the coefficients
// c(1), c(2), ..., c(n) of the Newton interpolating polynomial y=p(x)
// and evaluate at points xp.
//////////////////////////////////////////////////////////////////////
function yp = interpNewton(x, y, xp)
  n = length(y);
  c = y(:);
  for i=1:n-1
    for j=i+1:n
      c(j) = (c(j)-c(i))/(x(j)-x(i));
    end
  end
  yp = ones(xp)*c(1);
  u = ones(xp);
  for i=2:n
    u = u.*(xp-x(i-1));
    yp = yp+c(i)*u;
  end
endfunction
```

6.4 Coefficients of Newton polynomial

```
//////////////////////////////////////////////////////////////////////
// interpNewtonCoeffs.sci
// 2014-06-25, Scott Hudson, for pedagogic purposes only
// Given n samples x(i),y(i), in the column vectors x,y
// calculate the coefficients c(i) of the
// (n-1) order interpolating Newton polynomial
//////////////////////////////////////////////////////////////////////
function c = interpNewtonCoeffs(x, y)
  n = length(y);
  c = y(:);
  for j=2:n
    for i=n:-1:j
      c(i) = (c(i)-c(i-1))/(x(i)-x(i-j+1));
    end
  end
endfunction
```