CHAPTER 4

Interpolation

4.1. Introduction

We will cover sections 4.1 through 4.12 in the book. Read section 4.1 in the book on your own.

The basic problem of one-dimensional *interpolation* is this:

Given a set of data points x_i, $i = 0, \dots, m$, given function values y_i, and given a class of admissible functions, find a function f in this class so that the graph of f goes through these points:

$$f(x_i) = y_i.$$

More generally, we could prescribe not just function values, but also derivative values. An example is so-called *Hermite interpolation*, where we demand

$$\begin{aligned} f(x_i) &= y_i, \\ f'(x_i) &= y_i'. \end{aligned}$$

Depending on the class of admissible functions, we get different kinds of interpolation. The classes most frequently used in practice are

- *Polynomials* of a certain degree n, so that

$$f(x) = \sum_{k=0}^{n} c_k x^k.$$

- *Piecewise Polynomials* or *Splines*. We will talk about these later.
- *Rational Functions*, that is, quotients of polynomials.

$$f(x) = \frac{p(x)}{q(x)} = \frac{\sum_{i=0}^{m} c_i x^i}{\sum_{j=0}^{n} d_j x^j}$$

- *Trigonometric Polynomials*, which are defined as

$$f(x) = \sum_{k=0}^{n} a_k \cos(kx) + \sum_{k=1}^{n} b_k \sin(kx)$$

  This type of interpolation is related to Fourier series.

We will only consider polynomial and piecewise polynomial interpolation in this course.

A problem related to interpolation is the problem of *approximation.* Again, we are given a set of data points and a class of admissible functions. However, we do not require that the approximating function actually goes through these points. Rather, we are looking for an admissible function which gets "as close as possible" to these points. The exact meaning of "as close as possible" must be specified.

A typical example of approximation is trying to put a straight line through 10 data points. We will see some examples of this in chapter 6. All classes of functions used in interpolation can also be used in approximation.

Most interpolation problems are *linear.* This means that the class of admissible functions consists of linear combinations of some basis functions:

$$f(x) = \sum_{k=0}^{n} \alpha_k b_k(x).$$

The set of basis functions is not unique. For polynomial interpolation, one possible basis is $b_k(x) = x^k$. Other possible bases are the Lagrange polynomials (see below) or the polynomials used for Newton interpolation.

In trigonometric interpolation, the basis functions are sines and cosines. One possible basis for splines is the set of *B-splines* defined in section 4.14 (we skip that section).

Rational interpolation is an example of nonlinear interpolation. (Note that when you add two rational functions, the sum will usually have higher degree polynomials in numerator and denominator).

The general questions for any type of interpolation (and many other types of problems, too), are

Existence: Is there a solution?
Uniqueness: Is the solution unique?
Algorithms: How do you find the solutions in practice?

4.1.1. Solving Linear Interpolation Problems. There are two simple ways to attack linear interpolation problems. They are not always good ways to do it, but they often work. If you are faced with a linear interpolation problem with an unusual set of basis functions, you may want to start with one of these.

The Matrix Approach. Since we are solving a linear problem, any admissible function is of the form

$$f(x) = \sum_{k=0}^{n} \alpha_k b_k(x),$$

where the b_k are known basis functions, and the α_k unknown coefficients to be determined. Suppose we want to satisfy

$$f(x_i) = y_i \quad \text{for } i = 0, \ldots, m.$$

This leads to a linear system of equations

$$\begin{aligned} b_0(x_0)\alpha_0 + \cdots + b_n(x_0)\alpha_n &= y_0 \\ b_0(x_1)\alpha_0 + \cdots + b_n(x_1)\alpha_n &= y_1 \\ &\cdots \\ b_0(x_m)\alpha_0 + \cdots + b_n(x_m)\alpha_n &= y_m \end{aligned}$$

The coefficients α_j are the unknowns, the rest is known. If we have derivative information, we get equations containing $b_i'(x_j)$ or higher derivatives. These are also known, since we know the b_k.

This approach reduces the interpolation problem to a matrix problem. If $m = n$, we can hope to find a unique solution.

The Basis Function Approach. Suppose we could find basis functions $B_i(x)$ with the special property

$$B_i(x_j) = \delta_{ij}.$$

The δ_{ij} is called the *Kronecker delta* and is defined as

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $B_i(x)$ has the value 1 at x_i, and value 0 at the other x_j. The function

$$f(x) = \sum_{k=0}^{m} y_k B_k(x)$$

then has the required property

$$f(x_j) = y_j.$$

All we need to do is to find the $B_i(x)$ once and for all, which is often possible.

This approach can be modified if derivative information is given. For example, for Hermite interpolation you use the basis functions B_i and $\hat{B}_i$ which satisfy

$$\begin{aligned} B_i(x_j) &= \delta_{ij}, \\ B_i'(x_j) &= 0, \\ \hat{B}_i(x_j) &= 0, \\ \hat{B}_i'(x_j) &= \delta_{ij}. \end{aligned}$$

4.2. Polynomial Interpolation

Read this section in the book on your own. I have combined the material with the following section.

4.3. Using Other Basis Functions

I will cover the book sections 4.2 and 4.3 here. The book covers only the two approaches I mentioned in the introduction. I will also cover Newton's approach to solving the polynomial interpolation problem, which is not in the book. In my opinion, Newton's approach is preferrable to the other two.

Before we look at algorithms, let us answer the existence/uniqueness question.

Fact: Assume we are given a sequence of distinct interpolation points x_0, x_1, … x_m. At the point x_j, we are given n_j conditions

$$\begin{aligned} f(x_j) &= y_j, \\ f'(x_j) &= y'_j, \\ &\dots \\ f^{(n_j-1)}(x_j) &= y_j^{(n_j-1)}, \end{aligned}$$

for a total of $N = n_0 + n_1 + \dots n_m$ conditions. Then there is a unique interpolating polynomial of degree $N-1$ which satisfies these conditions.

This fact covers both standard interpolation ($n_j = 1$ for all j) and Hermite interpolation ($n_j = 2$ for all j).

The polynomial can be found using either one of the two approaches mentioned in the introduction.

Remark: I would like to emphasize two points that may not be obvious.

First of all, the theorem mentioned above states that *all possible methods produce exactly the same polynomial*, except for roundoff error. This polynomial can be written in a number of different ways, but there is only one. There is no reason to favor one approach over the other because it produces a better or worse interpolating polynomial.

Second, none of the methods treated here requires that the points x_j are in order. This is the way they usually come, but all three algorithms can handle points in any order.

From now on, we only consider the *standard polynomial interpolation problem.* We assume we have $(n+1)$ distinct points x_0, … ,x_n, in any order. We are given y_0, … ,y_n, but no derivative values. We want to interpolate with a polynomial of degree n (which has $(n+1)$ coefficients). We know that there is a unique solution.

4.3.1. The Matrix Approach.

We use any convenient basis for the set of polynomials. Usually, this means $b_j(x) = x^j$ or some shifted version $b_j(x) = (x - x_0)^j$.

Example: Put a polynomial through the points $(-1, 1)$, $(1, 1)$, $(3, 2)$ and $(5, 3)$.

We have 4 conditions, so we use a cubic polynomial (with 4 coefficients). We set up

$$p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

and get the equations

$$\begin{aligned} c_0 - c_1 + c_2 - c_3 &= 1 \\ c_0 + c_1 + c_2 + c_3 &= 1 \\ c_0 + 3c_1 + 9c_2 + 27c_3 &= 2 \\ c_0 + 5c_1 + 25c_2 + 125c_3 &= 3 \end{aligned}$$

The solution is

$$p(x) = (39 + x + 9x^2 - x^3)/48.$$

□

4.3.2. The Basis Function Approach.

The basis functions with the property $B_i(x_j) = \delta_{ij}$ for standard polynomial interpolation are called *Lagrange polynomials.*

They have an explicit representation

$$L_{n,k}(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)}$$

where k is between 0 and n. You can check by inspection that the $L_{n,k}$ have the required properties

- Each $L_{n,k}$ is a polynomial of degree n. (Notice there are n factors in the numerator, each with one x, and only constants in the denominator).
- $L_{n,k}$ has the value 0 at all the x_i except at x_k, where it has the value 1.

Note that the $L_{n,k}$ depend on the points x_j, so they have to be calculated from scratch for every interpolation problem.

Example: With the same numbers as in the last example, we get

$$\begin{aligned}
L_{3,0}(x) &= \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} \\
&= \frac{(x-1)(x-3)(x-5)}{(-1-1)(-1-3)(-1-5)} \\
&= (15 - 23x + 9x^2 - x^3)/48, \\
L_{3,1}(x) &= (15 + 7x - 7x^2 + x^3)/16, \\
L_{3,2}(x) &= (-5 + x + 5x^2 - x^3)/16, \\
L_{3,3}(x) &= (3 - x - 3x^2 + x^3)/48
\end{aligned}$$

and our solution is

$$p(x) = 1 \cdot L_{3,0} + 1 \cdot L_{3,1} + 2 \cdot L_{3,2} + 3 \cdot L_{3,3} = (39 + x + 9x^2 - x^3)/48,$$

like before. □

4.3.3. Newton's Approach.

Newton's method of finding the interpolating polynomial is better than either one of the methods mentioned above, in my opinion. I will explain the reasons later, when I compare the methods.

Instead of calculating the interpolating polynomial in one fell swoop, Newton does it in stages. Define $p_k(x)$ to be the polynomial of degree k which interpolates the data at $x_0, x_1, \ldots, x_k$, and build up p_0, p_1, and so on. Once you get to p_n, you are done.

A suitable representation is the following:

$$\begin{aligned}
p_0(x) &= b_0 \\
p_1(x) &= p_0(x) + b_1(x-x_0) \\
p_2(x) &= p_1(x) + b_2(x-x_0)(x-x_1) \\
&\cdots \\
p_n(x) &= p_{n-1}(x) + b_n(x-x_0)\ldots(x-x_{n-1})
\end{aligned}$$

It is obvious that p_k is a polynomial of degree k. The b_j are coefficients to be determined. We need to find suitable values so that p_k interpolates the data at x_0 through x_k.

We do that by induction:

p_0 is a polynomial of degree 0, i.e. a constant. We need to determine b_0 so that $p_0(x_0) = y_0$. The obvious solution is $b_0 = y_0$.

When we determine p_k, we already know that p_{k-1} interpolates at x_0 through x_{k-1} (induction hypothesis). Since $p_k = p_{k-1}+$ (a term that vanishes at x_0 through x_{k-1}), p_k also automatically interpolates at x_0 through x_{k-1}. The remaining condition $p_k(x_k) = y_k$ can be satisfied by a suitable choice of b_k.

It is easy to verify that

$$
\begin{aligned}
b_0 &= y_0, \\
b_1 &= \frac{y_1 - y_0}{x_1 - x_0},
\end{aligned}
$$

but after that it gets pretty messy.

Now, let us take a detour. Suppose we are given some points x_0 through x_n, and function values y_0 through y_n. We define *divided differences* recursively as follows:

Zeroth Divided Difference: $y[x_j] = y_j$.

First Divided Difference: $y[x_j, x_{j+1}] = \dfrac{y[x_{j+1}] - y[x_j]}{x_{j+1} - x_j}$.

Second Divided Difference: $y[x_j, x_{j+1}, x_{j+2}] = \dfrac{y[x_{j+1}, x_{j+2}] - y[x_j, x_{j+1}]}{x_{j+2} - x_j}$.

In general,

$$
y[x_j, \ldots, x_{j+k}] = \frac{y[x_{j+1}, \ldots, x_{j+k}] - y[x_j, \ldots, x_{j+k-1}]}{x_{j+k} - x_j}.
$$

These divided differences can be calculated very rapidly in a triangular table:

$$
\begin{array}{c|ccccccc}
x_0 & y[x_0] & & & & & & \\
 & & y[x_0, x_1] & & & & & \\
x_1 & y[x_1] & & y[x_0, x_1, x_2] & & & & \\
 & & y[x_1, x_2] & & y[x_0, x_1, x_2, x_3] & & & \\
x_2 & y[x_2] & & y[x_1, x_2, x_3] & & & & \\
 & & y[x_2, x_3] & & & & & \\
x_3 & y[x_3] & & & & & &
\end{array}
$$

Each entry is calculated as the difference of its left neighbors, divided by the difference of the corresponding x values.

So what does that have to do with anything? It turns out that

$$
b_k = y[x_0, x_1, \ldots, x_k],
$$

so that the top row in this triangular table contains exactly the correct Newton coefficients.

Example: Same example as before. We get

$$
\begin{array}{r|llll}
-1 & 1 & & & \\
 & & 0 & & \\
1 & 1 & & 1/8 & \\
 & & 1/2 & & -1/48 \\
3 & 2 & & 0 & \\
 & & 1/2 & & \\
5 & 3 & & &
\end{array}
$$

Therefore, Newton's solution is

$$
p(x) = 1 + 0 \cdot (x+1) + \frac{1}{8}(x+1)(x-1) - \frac{1}{48}(x+1)(x-1)(x-3).
$$

If you multiply it out, you get the same polynomial as before. □

4.3.4. Comparison of Methods. Let me repeat the point I made before: *All methods produce exactly the same polynomial*, except for roundoff error. The only difference between the methods is in efficiency (less work) and stability (less round-off error).

The matrix approach is easy to understand, but that is where its advantages end. It is not very efficient, and with increasing number of points, it rapidly becomes unstable.

The Lagrange basis function approach is very stable (the most stable of all methods, as a matter of fact), but again not very efficient.

The Newton approach is much faster than the other two, it is pretty stable, and it has one other advantage: it lets you add extra points to the existing ones without starting over again. You simply add another line or several to the divided difference table. This is useful if you don't know how many points you need.

4.4. How Good is Polynomial Interpolation?

If the interpolation data comes out of the blue, there is no point in talking about error, since there is no "true solution" to compare with the result.

Let us therefore assume that the data values y_j are values of a known function $g(x)$ at the points x_j, $j = 0, \ldots, n$. The *interpolation error* is the difference between the original function and the interpolating polynomial (ignoring roundoff error). How large can the interpolation error get?

There are two questions I want to address here:

- For a fixed set of points x_j, how large is the error?
- What happens as we increase the number of points more and more?

4.4.1. Fixed Interpolation Points. Assume the points x_j are fixed. They may be table values of a function $g(x)$, or measured values.

If g has $(n+1)$ continuous derivatives, then it can be shown that

$$g(x) - p(x) = \frac{\omega(x)}{(n+1)!} g^{(n+1)}(\xi),$$

where

$$\omega(x) = (x - x_0)(x - x_1) \ldots (x - x_n),$$

and ξ is some point between x_0 and x_n.

If $g(x)$ is known, we can estimate the error, at least in principle.

One thing to observe is that as long as x is somewhere among the x_j, $\omega(x)$ is not too bad. If we move x around, some terms $x - x_j$ will get larger, others smaller. However, if we move x outside, $\omega(x)$ will grow very rapidly (all terms get larger). *Polynomial interpolation becomes extremely unreliable if we try to extrapolate into an area where we don't have any data points.*

4.4.2. Increasing the Number of Points. We can see that if we keep the function $g(x)$ fixed and increase the number of interpolation points, the error estimate involves higher and higher derivatives of g. Even for nice functions g, higher derivatives may become large. Look at the Runge example in the book for a bad case.

How should we choose the points to keep the error growth under control?

Using equally spaced points is pretty bad. Using a different type of error estimate than shown here, one can show that the worst case error grows exponentially in n.

If one chooses the so-called *Chebyshev points*, the error grows only like $\log(n)$, which is much better. On the interval $[-1, 1]$, the Chebyshev points are given by

$$x_j = \cos(\frac{2j-1}{2n}\pi), \quad j = 1, \ldots, n.$$

For other intervals, these points need to be scaled and shifted. (Note that the numbering is from 1 to n here, instead of from 0 to n). The Chebyshev points are not the best possible, but they are close to the best possible, and they are easy to find.

If you are using more than 4 or 5 points, you can't expect good results from equally spaced points. Use the Chebyshev points.

Summary¿: Even in the best case, we have to assume that with increasing n the error in polynomial interpolation will grow. This is a basic shortcoming of polynomial interpolation.

4.5. Historical Perspective

Read this section on your own.

4.6. Evaluation of Polynomials

How do you evaluate

$$p(x) = a_n x^n + \cdots + a_1 x + a_0$$

on a computer, if the a_j and x are known?

The direct "multiply it out" approach is not very efficient. Even if you are smart enough to calculate x^j as $x^{j-1} * x$, instead of from scratch, it takes $(n-1)$ multiplications to generate all powers of x, and another n multiplications to multiply them by the coefficients, for a total of $(2n-1)$.

The best known method is *Horner's Rule*, which is based on writing

$$p(x) = (\ldots(((a_n x + a_{n-1})x + a_{n-2})x \ldots)x + a_1)x + a_0.$$

This takes only n multiplications.

With a slight multiplication, the same thing works for Newton's approach:

$$p(x) = (\ldots(b_n(x - x_{n-1}) + b_{n-1})(x - x_{n-2}) + \ldots)(x - x_0) + b_0.$$

Altogether, Newton's approach takes about $n^2/2$ divisions to generate the divided difference table, and then n multiplications per evaluation. The interpolating polynomial itself is never written down.

Here are two little subroutines I wrote to calculate Newton's interpolating polynomial. They might come in handy for a homework assignment. I will put these routines in `/home/mathclasses/keinert/473` on PV and in the `class:` directory on Vax.

```
      subroutine ncoef(x,y,n)
*
*     calculate coefficients for Newton interpolating polynomial
*     for interpolating x1..xn, y1..yn.
*     Results are returned in y.
*
      integer  n, i, j
      real     x(n), y(n)
*
      do i = 2, n
         do j = n, i, -1
            y(j) = (y(j) - y(j-1)) / (x(j) - x(j-i+1))
         enddo
      enddo
      end

      real function neval(x,b,n,x0)
*
*     evaluate the Newton polynomial given by x1..xn,
*     coefficients b1..bn, at the point x0
*
      integer n, i
      real    x(n), b(n), x0
*
      neval = b(n)
      do i = n-1, 1, -1
         neval = neval * (x0 - x(i)) + b(i)
      enddo
      end
```

To use them, you have to call `ncoef` once, which will construct the divided difference table. It turns out you can do that "in place", so that the table itself is never stored, just its top row, which is all you need. When you enter `ncoef`, array `y` contains the function values. On return, it contains the coefficients b_j.

The second routine `neval` evaluates the polynomial at a point x_0, using the modified Horner scheme described above.

4.7. Piecewise Linear Interpolation

Read section 4.7 in the book.

Assume we are given an increasing sequence of points x_j, $j = 0, \ldots, n$, called the *knots*. A *piecewise polynomial* is a function which reduces to a polynomial on each subinterval $[x_j, x_{j+1}]$. Different subintervals can contain different polynomials. Usually we are only interested in piecewise polynomials which reduce to polynomials of the same degree k on every subinterval.

Meta-theorem: If the function $g(x)$ has $(k+1)$ continuous derivatives on $[a, b]$, if we divide $[a, b]$ into n equally spaced subintervals of width $h = (b-a)/n$, and interpolate g at the knots by a piecewise polynomial $s(x)$ of degree k, then the error satisfies

$$|s(x) - g(x)| \le Ch^{k+1} |g^{(k+1)}(\xi)|,$$

where C is some constant (depending on the type of piecewise polynomials we use) and ξ is some point in (a, b).

By "meta-theorem" I mean that the statement is much too vague to be a theorem. It is more a "rule of thumb", standing for a collection of theorems, one for each type of piecewise polynomials. For each type of piecewise polynomials, you have to prove it from scratch, producing a theorem for this particular case.

One way to remember the meta-theorem and to make it plausible is to observe that this looks remarkably like the remainder term of a Taylor series. If I used a Taylor polynomial of order k on each subinterval, I would get this error estimate with $C = 1/(k + 1)!$.

The important thing to observe is that as $n \to \infty$, the error goes to zero, since h goes to zero, and the rest stays bounded. This is the big advantage piecewise polynomials have over polynomials.

4.8. Piecewise Cubic Functions

I will cover book sections 4.8, 4.10 and 4.11 in this section. Read sections 4.8 and 4.11 in the book on your own. Reading 4.10 is optional.

As you probably guessed already, a piecewise cubic function is a piecewise polynomial which reduces to a cubic polynomial in each subinterval. Two important types are *Hermite cubics* and *cubic splines*.

In piecewise polynomial interpolation, it is important that the points are in order: x_0 ¡ x_1 ¡ $\ldots$ ¡ x_n.

We will stick to the standard problem (function values given, but no derivatives).

4.8.1. Hermite Cubics.

A Hermite cubic is a piecewise cubic polynomial with one continuous derivative. Thus, at every interior point where two cubic polynomials come together (at each *knot*), we demand that the function values and the first derivative fit together.

Suppose we are given data points (x_j, y_j), $j = 0, \ldots, n$ to interpolate. We have n subintervals and a cubic polynomial on each. That makes $4n$ coefficients to determine.

We have $2n$ constraints from the interpolation data (note that the interior points put constraints on *two* polynomials). The other $2n$ constraints come from prescribing derivative values y'_j. (In case you haven't noticed it yet: the word "Hermite" in the context of interpolation usually means you are given function values and derivatives). Prescribing derivatives automatically makes sure that the derivatives fit together at the knots.

If you know the derivatives, that's fine. What if you don't? In that case, you can estimate the derivates from the y_j. To estimate y'_j, you put a parabola through the interpolation points $(j-1)$, j and $(j+1)$, and differentiate that. Details are on page 103. I have seen this process called *Bessel interpolation*.

So now we have data y_j and y'_j. Since we have $4n$ constraints and $4n$ coefficients to determine, we expect to find a unique solution.

One possible method of attack would be to create a system of $4n$ equations in $4n$ unknowns (the "matrix method"). A better method is the "basis function approach" mentioned earlier. Details for that are given in section 4.10 in the book.

We will skip the details.

4.8.2. Cubic Splines. A cubic spline is a piecewise cubic polynomial with two continuous derivatives. Thus, at every interior point where two cubic polynomials come together, we demand that the function values and the first and second derivative fit together.

Remark: We can't require that more than two derivatives fit together: if the function values and three derivatives of two cubic polynomials fit together, they are identical polynomials.

Remark: The reason for the name *spline* is the following: A spline originally was a flexible metal rod used as a drafting tool. In order to produce a smooth curve passing through a set of points, the draftsman would pin the rod down on the paper at these points and trace the shape of the rod.

The rod has a tendency to assume the shape which minimizes the internal energy. The energy is related to the curvature, and the curvature is related to the second derivative. Approximately, the rod would take on a shape which minimizes the average second derivative. The splines we consider here have the same property: they minimize the average second derivative. This is where they got their name.

The more derivatives a function has, the smoother it is. Hermite polynomials had one continuous derivative. We expect cubic splines to look smoother.

Suppose again that we are given data points (x_j, y_j), $j = 0, \ldots, n$ to interpolate. We have n subintervals and a cubic polynomial on each. That makes $4n$ coefficients to determine.

Specifying the values y_i gives us a total of $2n$ conditions, like before. Matching first derivatives left and right at the interior points gives $(n-1)$ conditions; likewise for the second derivatives. The total is $(4n-2)$ conditions, so we need two extra conditions.

There are many ways to get these conditions, and each one leads to a different kind of spline.

- Set the second derivative equal to zero at the endpoints:
$$s''(x_0) = 0$$
$$s''(x_n) = 0$$
This is the so-called *natural* or *free spline*. This is the type of curve the draftsman got out of his metal spline by letting the ends stick out free.
- If we prescribe the derivatives at the endpoints
$$s'(x_0) = y_0'$$
$$s'(x_n) = y_n'$$
we get the *clamped* or *complete spline*. Physically, this corresponds to clamping the ends of the metal spline into a certain position.

  If we don't know what the derivatives are, we could estimate them. This is the same idea as with Hermite interpolation.
- We could prescribe the second derivatives at the endpoints
$$s''(x_0) = y_0''$$
$$s''(x_n) = y_n''$$
  Again, if you don't know the values, you could estimate them. The previous edition of your textbook contained a program based on this.
- We could demand that three derivatives match instead of two at the first and last interior points:
$$s'''(x_1 - 0) = s'''(x_1 + 0)$$
$$s'''(x_{n-1} - 0) = s'''(x_{n-1} + 0)$$
This is the same as saying that we want the first two and last two cubic polynomials to be identical. (If two cubic polynomials match in value and three derivatives at a point, they are identical). This is called the *not-a-knot spline*.

  The reason for the name is this: A *knot* is a place where different polynomials come together. Since the first two polynomials are actually the same, x_1 is not a knot, even though it may look like one. Likewise for x_{n-1}.

  The spline portion of program PCHEZ in NMS uses this type of spline.

- For the *periodic spline* we demand that the first two derivatives match at the endpoints

  $$f'(x_0) = f'(x_n)$$
  $$f''(x_0) = f''(x_n)$$

  (and of course $y_0 = y_n$, otherwise this does not make sense). This spline is obviously only useful to interpolate periodic functions.

We can choose a different type of conditions at each endpoint.

There are various ways to go about calculating a cubic spline.

The "matrix approach" is possible, but not very good.

The "basis function approach" works well. The basis functions in this case are called *B-splines.* Look at section 4.14 in the book, if you want, but don't expect to understand much from the few crumbs of information the authors are doling out in that section.

The way it is done in the `PCHEZ` program in the book is by calculating the derivatives of the cubic spline at all x_j. Once we have those, we are done: on each subinterval, we have 4 coefficients and 4 constraints (function value and derivative at each end). We can write down a formula for evaluating the spline s and its derivative s' at any point from this data.

The derivative values are found by solving a tridiagonal system of equations of size $(n+1) \times (n+1)$. As you recall, this is very fast.

By doing it this way, most of the programming is identical for Hermite cubics and for cubic splines: For Hermite cubics, we estimate the derivatives by interpolation. For cubic splines, we solve a tridiagonal system. After that, everything is identical.

A similar approach with which I am more familiar is to calculate the *second* derivatives at all x_j. This also leads to a tridiagonal system, and we also get formulas to calculate the 4 coefficients on each subinterval from 4 constraints (2 function values and 2 second derivatives, in this case).

4.9. PCHIP

Read this section in the book on your own.

Note that there are two programs: `PCHEV` is called once at the beginning, to get the coefficients set up. Afterwards, `PCHEZ` is called repeatedly, once or for every set of values where you want the interpolating cubic evaluated.

This is not unlike the internal division between `SGEFA` and `SGESL` in Gaussian elimination, or the division between `ncoef` and `neval` in my own subroutines.

4.10. Cubic Hermite Interpolation: Details

You can skip this section in the book, or read it for your own amusement.

4.11. Cubic Splines

Read this section in the book. I have covered the material under section 4.8 above.

4.12. Practical Differences Between Splines and cubic Hermite polynomials

Read this section on your own.

The most important piece of information you should remember from this section is that the error estimate for both cubic splines and cubic Hermites looks like

$$s(x) - g(x) \leq Ch^4 |g^{(4)}(\xi)|.$$

(Remember the meta-theorem from earlier?). For equally spaced points, the h is the subinterval size. For unequally spaced points, h is the size of the largest subinterval.

In contrast to polynomial interpolation, the derivative estimate stays fixed for any number of points. If we take more and more points, the error goes to zero (and quite rapidly, too), as long as the length of each subinterval goes to zero.

Remark: In addition to better error estimates, piecewise polynomials have another advantage over regular polynomials: If you change the data at one point, this usually produces significant changes in the interpolating polynomial over the entire interval. In contrast, the changes in the interpolating piecewise polynomial tend to "die out" quite rapidly away from the point of change. Changes tend to remain localized.