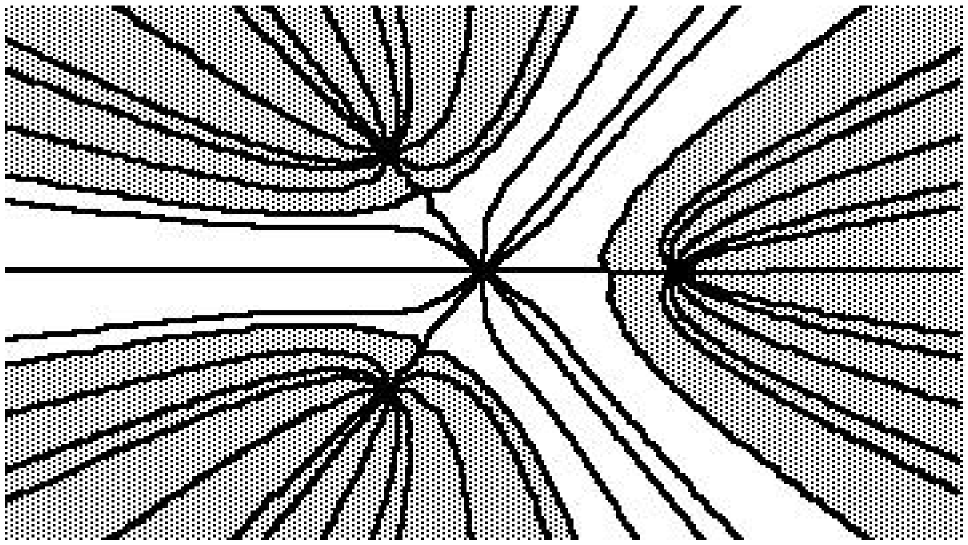
Theory of Equations

Lesson 6

by

Barry H. Dayton
Northeastern Illinois University
Chicago, IL 60625, USA

`www.neiu.edu/˜bhdayton/theq/`



2.11 Newton's Method, Real Case

Today, the method of choice for finding roots of polynomials is Newton's method. This method is sometimes known under other names such as the "Newton-Raphson" method, but I believe it is proper to call this Newton's method as it first appears in full generality in Newton's "Method of Fluxions." In the case of quadratics of the form $x^2 - a$ this method (whose result is $\sqrt{a}$) was known to the Babylonians.

Newton's method is well adapted to the calculator or computer, is fast, will generally be as accurate as the internal accuracy of the calculator or computer and, although this method does not work as well on roots of multiplicity greater than 1, it is the only algorithm that we will study extensively that works at all for multiple roots.

The algorithm goes as follows. Let $f(x)$ be your polynomial, choose an approximate root x_0. Then (hopefully) a better approximation is $x_1 = x_0 - f(x_0)/f'(x_0)$. A still better approximation is $x_2 = x_1 - f(x_1)/f'(x_1)$. In general we get a sequence of approximations defined by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

After a relatively small number of iterations, the sequence $\{x_n\}$ will usually converge to a root.

Consider the example we used to illustrate Horner's method. Here $f(x) = x^4 - 2x^3 + x^2 - 3$, $f'(x) = 4x^3 - 6x^2 + 2x$. Choose as a starting point $x_0 = 2$. Then

$$\begin{aligned}
x_1 &= \quad 2 - .083333333 &&= 1.916666667\\
x_2 &= 1.916666667 - .008723748 &&= 1.907942919\\
x_3 &= 1.907942919 - .000089648 &&= 1.907853271\\
x_4 &= 1.907853271 - .000000009 &&= 1.907853262
\end{aligned}$$

At this point our computer tells us $f(x_4) = 0$ so the terms x_n have stabilized at x_4 and so we may stop.

One way we may justify this method is graphically. Given a value x_0 relatively close to a root with $y_0 = f(x_0)$ we construct the tangent line to the graph at (x_0, y_0). The slope is $f'(x_0)$ and using the point-slope form of the equation of a line the equation of the tangent line is

$$(y - y_0) = f'(x_0)(x - x_0).$$

If x_0 is close to a root the tangent line crosses the x-axis even closer to that root. To find this point, set $y = 0$ in the equation above and solve for x. We get $f'(x_0)x =$

$f'(x_0)x_0 + y_0$ so dividing by $f'(x_0)$ and recalling that $y_0 = f(x_0)$ we get

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

which is the formula for the next iterate in Newton's method.

A question that arises whenever using an iterative algorithm such as Newton's method or one of the fixed point iteration methods of the last section to solve a polynomial equation $y = f(x)$ is "when do you stop?" There are essentially three different possibilities for a stopping criterion. One is when $|f(x)|$ becomes close enough to 0. For some practical applications when it is more important that $f(x)$ be smaller than some error allowance than that x be an exact root this would be a good choice. A second possibility is to stop when $|x_{n+1} - x_n|$ is small for two consecutive iterates. However the best is probably to have the relative error $|x_{n+1} - x_n|/|x_n|$ small. This is particularly relevant when using a calculator or computer since your answer can never be more accurate than the internal number of significant digits. Essentially this relative error can measure significant digit accuracy and thus a good stopping criterion is to stop when

$$\frac{|x_{n+1} - x_n|}{|x_n|} < 0.5 * 10^{-t}$$

where 10^{-t} is the machine precision.

A drawback of the real Newton's method is that if the initial point is not picked close to a real root then the algorithm may not converge quickly. For example, one might investigate the polynomial $f(x) = 1 + 4x - 12x^2 + 13x^3 - 6x^4 + x^5$. This has one real root which is negative. Just about any choice of a positive number for x_0 will take 30 or more iterations to converge. Some choices, such as $x_0 = 6$ may never converge. The issue of non-convergence of Newton's method will be more fully discussed in the next chapter. However, if one first uses a graph or Descartes rule or Newton's upper bound to approximatly locate a root and then use several iterations of the bisection method to get closer, Newton's method will then converge in a very few iterations. For simple roots (multiplicity one) Newton's method close to the root converges ”quadratically”, i.e. the error in the next iteration will be on the order of the square of the previous error. Thus if x_0 is within .1 of the actual root, x_1 will have an error around .01, x_2 an error around .001, x_3 around .00001 and so on.

As mentioned above, Newton's method will not work as well for a multiple root, but it will work. If c is a multiple root of $f(c)$ then the multiplicity of the root c of $f'(x)$ is one less than the multiplicity of c as a root of $f(x)$. Thus if x is chosen near c but not exactly c then $f(x)/f'(x)$ is defined and goes to 0 as x goes to c. For instance, after a change of variables we can assume $c = 0$ and $f(x) = x^m + \cdots$. For x close to 0 we can ignore the higher order terms so we can take $f(x) = x^m$. A calculation shows that the iteration formula is $x_{n+1} = ((m-1)/m) * x_n$ or $x_n = ((m-1)/m)^n * x_0$. As $(m-1)/m < 1$ we know that $((m-1)/m)^n$ goes to 0 as n gets large and thus Newton's method will converge to the root. For a root of multiplicity two the error is halved each time so Newton's method is about the same speed as the bisection method (as applied to simple roots, recall that bisection will not work for roots of even multiplicity), for higher multiplicity convergence is even slower. There are variations of Newton's method which will speed convergence in this case, or better yet one can use the Euclidean algorithm to reduce your polynomial to one where the multiplicities of the roots are smaller. As a practical matter, however, polynomials with multiple roots are rare in nature and thus this problem will not occur often.

Maple Implementation

MAPLE does not have a built in Newton's method, using instead more sophisticated solving methods. It is easy enough to write your own Newton's method procedure to watch the method converge.

```
Newt := proc(f,z0,n) local g,z,j;
   g := x - f/diff(f,x);
   z := evalf(z0);
   for j to n do
     z := evalf(subs(x=z,g));
     print(z);
   end do;
end;
```

Here `f` is the polynomial in indeterminate `x` whose root is to be found, `z0` is the initial point and n is the number of iterations (10 is a good number to start with). As I mentioned earlier in regards to iteration, you are warned to not use an indeterminate or algebraic expression as your initial value!

Exercise 2.11.1 [20 points] Find all 5 real roots of $f(x) = x^5 - 8x^3 + 10x + 1$ to 6 significant digits using Newton's method. For each root give the initial point used and the number of iterations needed.

2.12 Newton' Method, Complex Case

Perhaps the best thing about Newton's method is that not only does the algorithm work in the complex case but it works much better in the complex plane, converging promptly to a root from almost any initial point.

The method is the same:

1. Choose an initial point z_0.
2. Iterate using $z_{n+1} = z_n - f(z_n)/f'(z_n).$

Here z is the variable we often use to denote a complex number and the arithmetic here is that of complex numbers. Starting with an imaginary initial point the sequence $\{z_n\}$ almost always will converge quickly to a root (real or imaginary) of $f(z).$ In fact, if $f(z)$ has real coefficients Newton's method will often converge more quickly to a real root starting from an imaginary initial point. The reason is that the real Newton's method can get hung up on a critical point (i.e. a zero of $f'(z)$) whereas the complex Newton's method has room to go around it.

It is more difficult to estimate the location of imaginary roots than real roots. Fortunately with the complex Newton's method this is not so vital. Using Newton's upper bound one can find upper bounds on the modulus of complex roots. Often convergence is more direct if one starts with an initial estimate slightly larger in modulus than the desired root. It should be remembered that for polynomials with real coefficients, the imaginary roots come in conjugate pairs.

The complex Newton's method is easiest to implement using a computer language such as MAPLE or FORTRAN which supports complex arithmetic, however with care the algorithm can be implemented using other languages.

Generally implementations of the complex Newton's method use Horner's process to calculate $f(z)$ and $f'(z)$ efficiently. This method works the same in the complex case as in the real case. Sometimes because Horner's process is used, algorithms to do the complex Newton's method are, incorrectly, called Horner's algorithm.

When Newton's method finds a root c of $f(z)$ then $f(z)$ factors as $f(z) = (z - c)q(z)$. $q(z)$ is the first quotient calculated by Horner's process when calculating $f(c)$. The method of "deflation", once a root is found, replaces $f(z)$ with $q(z)$. Then after finding a root of $q(z)$ deflation is used again and so on, each time we get a polynomial of lower degree to solve. By the time a linear or quadratic polynomial is reached then it can be solved directly. In this way one gets quickly and in an orderly fashion all the complex roots of a polynomial. The quotients $q(z)$ obtained by deflation are only approximate quotients since the roots found are only approximate roots, thus one may wish to go back to the original polynomial and start Newton's iteration from these points. After an iteration or two any inaccuracies will be corrected. In practice, however, the roots found by deflation are already as accurate as can be expected, given the machine precision.

For example, starting with $f(z) = z^3 + z + 1$ an initial point of $-3 + 3i$ gives after 8 iterations the root $\alpha = -.34164 + 1.16154i$. The deflated polynomial is $(-.233 - .793i) + (-.341 + 1.162i)z + z^2$. An initial point $-.4 - i$, which is close to $\bar{\alpha}$, gives after 4 iterations the root $\bar{\alpha}$. The deflated polynomial is then $-.682328 + z$ which gives immediately the root $\beta = .682328$. Using β as an initial point in Newton's method applied to the original polynomial confirms β as an accurate root.

Exercise 2.12.1 [20 points] Write a program in your favorite programming language to do Newton's method with deflation. If your language does not support complex arithmetic just do a real Newton's method. Illustrate with a few polynomials.

Obviously the graphical argument we used to justify Newton's method in the real case won't work here in the complex case, although the use of the derivative to get a linear approximation to $f(z)$ is still valid. There are several other arguments one can use to justify Newton's method in the real or complex case for use near a root. We can think of Newton's method as a fixed point iteration process with $g(z) = z - f(z)/f'(z)$. But then

$$g'(z) = 1 - \frac{f'(z)f'(z) - f(z)f"(z)}{f'(z)^2} = \frac{f(z)f"(z)}{f'(z)^2}$$

Hence if c is a root of $f(z)$ we have $g'(c) = 0$. Thus c is an attractive fixed point in the strongest possible way for g. Another way to justify Newton's method around a root is to calculate the Taylor series for $f(z)/f'(z)$ about $z = c$, this is an actual infinite series which looks, in the case of a simple root c, like

$$(z - c) + b_2(z - c)^2 + b_3(z - c)^3 + \cdots$$

Thus for z_n close to c,

$$z_{n+1} = c + b_2(z_n - c)^2 + b_3(z_n - c)^3 + \cdots$$

and this error term is on the order of $(z_n - c)^2$. Thus the error in z_{n+1} is approximately the square of the error in z_n.

Thus Newton's method can be easily justified locally. We said earlier that the complex Newton's method converges in practice for just about any choice of z_0. It would be nice to have a justification of this statement but this turns out to be extremely difficult and not entirely true. This is the topic of much current research in mathematics. The reasons for this will be examined in the next chapter.

Exercise 2.12.2 [10 points] Use the complex Newton's method with deflation (if you use Maple you can deflate manually) to calculate the 7 complex roots of $p(x) = 2 - 3x + x^4 + 8x^7$. In a previous Exercise we showed all roots α satisfy $|\alpha| < 1$. For each root α of $p(x)$ find an initial point z_0 with $|z_0| > 1$ for which Newton's method applied to $p(x)$, not a deflation, converges to α and give the number of iterations required.

2.13 The Newton–Barstow algorithm

Because the complex Newton's method requires complex arithmetic it is a little bit awkward to implement in a computer language, such as versions of BASIC or C, which does not support complex arithmetic. Since the coefficients of the original polynomial are usually real the Newton-Barstow algorithm which can find imaginary roots of real polynomials using only real arithmetic is sometimes an appealing alternative.

The idea behind the Newton-Barstow method is interesting, although a bit complicated. We only sketch it here. We start with a real polynomial $p(x)$ so imaginary roots occur in conjugate pairs. In particular each imaginary root corresponds to a real quadratic factor of $p(x)$, thus the goal of this algorithm is to find such factors of $p(x)$. Thus we start with a guess $x^2 + ax + b$ of a quadratic factor. Dividing $p(x)$ by this guess we get a quotient and a linear remainder $cx + d$. We want this remainder to be 0 so the procedure is to refine our initial guess in order to get a smaller remainder. We can regard the coefficients of the remainder c, d as functions of the two variables a, b. By a variation on Horner's procedure we can calculate the partial derivatives of c, d in terms of a, b. We can then apply a two dimensional real version of Newton's method to produce new values of a, b which will tend to give smaller values of c, d. Iterating this procedure c, d become small enough to be considered 0 and so $x^2 + ax + b$ is a quadratic factor of $p(x)$. The roots of $x^2 + ax + b$ can then be found by the quadratic formula.

Although the idea is complicated, the implementation of this algorithm is simple. We give a MAPLE procedure.

Maple Implementation

```
NBarstow := proc(f,a1,a0)  local i,k, u,n,s1,s2,
              jac,b0,b1,b2,c0,c1,c2,d1,d2;
 n := degree(f,'x');
 if n < 3 then RETURN(f); fi;
 s1 := -a1;  s2 :=-a0;
 for k to 80 do
```

```
          b1 := coeff(f,'x',n); c1 := 0;
          b0 := coeff(f,'x',n-1) + s1*b1; c0 := b1;
          for i from n-2 by -1 to 0 do
             b2 := b1; c2 := c1;
             b1 := b0; c1 := c0;
             b0 := coeff(f,'x',i) + s2*b2 + s1*b1;
             c0 := b1 + s2*c2 + s1*c1;
          od;
          jac := c1^2 - c0*c2;
          d1 := (b1*c1 -b0*c2)/jac;
          d2 := (b0*c1 -b1*c0)/jac;
          s1 := evalf(s1-d1); s2 := evalf(s2-d2);
          if evalf(d1^2 + d2^2) < 10^(-12)*(s1^2+s2^2)
             then RETURN( ('x')^2-s1*'x' - s2);
          fi;
       od;
       RETURN(FAIL);
    end;
```

For this procedure, `f` is given as an expression in the indeterminate `x` and the initial guess of a quadratic factor is `x^2 + a1 x + a0` . If successful, the procedure returns an actual (numerically approximate) quadratic factor.

The Newton-Barstow algorithm has several drawbacks. It does not work at all well in the case of multiple roots and if a polynomial of odd degree has only one real root, this real root is not a root of any real quadratic factor and hence is invisible to the method, although it can be found by deflation. And unlike Newton's method, this method cannot be adapted to find zeros of complex functions other than polynomials. However for polynomials the drawbacks can be compensated for, for instance one can use the real Newton's method to find invisible real roots. And if one suspects multiple roots then these multiple roots are also roots of $p'(x)$ so one strategy is to apply the Newton-Barstow method to the lower degree $p'(x)$, and if that still doesn't work to $p''(x)$ etc.

The Newton-Barstow algorithm is a useful alternative to Newton's Method in the D'Alembert's Theorem situation, i.e., one wants real quadratic factors rather than roots. See, for example, section 5.5.

Exercise 2.13.1 [10 points] Factor $x^5 - 6x + 3$ as a product of real linear and quadratic factors using the Newton-Barstow algorithm.

2.14 Other root finding algorithms

Several root finding methods often discussed in Numerical Analysis texts are the Secant Method and Muller's method. With complex arithmetic these will calculate complex roots, if desired. The advantage in these methods is that it is not necessary to calculate the derivative. But for polynomials the derivative is no problem so these methods offer no advantage and they are slower than Newton's method. Another method, Leguerre's Method, is often used in commercial software. This method, which is fairly complicated, is an iteration method which converges quickly for polynomials with only real roots. It also works for more general polynomials.

A method of historical interest, but of no practical modern value, is Graeffe's method which was developed in the early 19th century by a number of mathematicians, one being the geometer Lobachevsky. This method was used to find the imaginary roots of a real polynomial until recently. The idea is the following. Given a monic polynomial $f(x)$ it is easy to show that $f(\sqrt{x}) * f(-\sqrt{x})$ is a monic polynomial of the same degree but with roots the squares of the original roots. For example if $f(x) = 6 - 5x + x^2$ which has roots 2 and 3 then

$$\begin{aligned} f(\sqrt{x}) * f(-\sqrt{x}) &= (6 - 5\sqrt{x} + x)(6 + 5\sqrt{x} + x) \\ &= (6 + x)^2 - 25x \\ &= 36 - 13x + x^2 \end{aligned}$$

which has roots 4 and 9. Repeating this process spreads out the moduli of the roots and one can eventually deduce the moduli from the derived equations (see, for instance, Uspensky for details or Example 4.6.2 below). Knowing the moduli of the roots one can then deduce the roots (again refer to Uspensky). In the case when one conjugate pair of roots has larger modulus than other conjugate pairs the method is not too difficult. If there are several pairs of roots with the same largest modulus the method can get quite involved, using lots of logarithms and trigonometry. Uspensky says of this method that "it is the only practical method for the computation of imaginary roots" but it is hard for the mathematician of today to see how that could have been true.

Exercise 2.14.1 [50 points] Look up Graeffe's method in Uspensky's book or some other classical theory of equations book. Use this method to find the roots of $f(x) = x^4 - x^3 + 7x^2 - 5x + 10$. In keeping with the spirit of the time you should look up all logarithms in tables but I will allow you to use your calculator whenever possible! Show all steps.

In some Linear Algebra books it is suggested that to find the eigenvalues of a square matrix one should calculate the characteristic polynomial and find its roots. For those who still insist on this method we will discuss an efficient way to calculate the characteristic polynomial in section 4.7. In practice, however, the modern mathematician will use iterative methods to directly calculate the eigenvalues of a matrix. Since a computer is essential in such a calculation, and software for this problem is widely available, the modern mathematician who wishes to find roots often finds that the quickest way (in human time, not computer time) to find all roots of a polynomial is to construct a matrix whose characteristic polynomial is the desired polynomial and then to pop this matrix into the computer and find the eigenvalues.

The first part is easy, for given a monic polynomial $p(x) = a_0+a_1x+a_2x^2+\cdots+x_n$ the *companion* matrix is the $n \times n$ matrix

$$C_p = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots 0 & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ -a_0 & -a_1 & -a_2 & -a_3 & \cdots & -a_{n-1} \end{bmatrix}$$

and $\det(C_p) = (-1)^n p(x)$ (see, for instance, Birkhoff and MacLane for more details.) In either case the roots are the same as those of $p(x)$ so the eigenvalues of the companion matrix are the roots of the polynomial.

Having discussed a variety of methods, some of which show how roots should not be found, some of which show how roots should be found, and this last which shows how roots usually are found we have completed our discussion of root finding algorithms.

2.15 Polynomial Interpolation

No discussion of Numerical Analysis and polynomials would be complete without mentioning polynomial interpolation. Since polynomials are the simplest type of functions (other than linear functions) it is desirable to approximate functions by polynomials. For any suitably differentiable function a good way to get a polynomial aproximation near a point c is to take the first n terms of the Taylor series. However this is only accurate near c, usually, and we would like an approximation that is close in an entire interval.

The method used is interpolation. Given a function $f(x)$ and an interval $[a, b]$ one chooses $n + 1$ points usually evenly spaced in an interval $a = x_0 < x_1 < x_2 <$

$\cdots < x_n = b$ and then constructs a polynomial $p(x)$ of degree n so that $p(x_j) = f(x_j)$ for $j = 0, 1, ..., n$. One nice feature of this is that it is not necessary to know much about the function $f(x)$ except its values $y_j = f(x_j)$ at these $n+1$ points. This is particularly useful when $f(x)$ is known only by experimental data, for we can then find a polynomial which agrees with our data and if necessary we can calculate values of the function for values of x inside our interval other than our experimental values of x. Extrapolation is the word used to describe the method of approximating the value of $f(x)$ for x outside the interval by calculating $p(x)$. However closely the interpolating polynomial approximates the value of $f(x)$ inside the interval $[a, b]$ it usually diverges quickly from $f(x)$ outside the interval. Thus while interpolation is extremely accurate, those who extrapolate usually make fools of themselves.

In light of the above discussion our problem is: given $n+1$ values of x, $x_0, x_1, x_2, \ldots, x_n$ and $n+1$ values of y, $y_0, y_1, \ldots y_n$ to find a polynomial $p(x)$ of degree n so that $p(x_j) = y_j$ for $j = 0, 1, \ldots, n$. The following theorem of Lagrange shows that this is always possible.

Theorem 2.15.1 *Given $x_0, x_1, \ldots, x_n$ distinct real or complex numbers and $y_0, y_1, \ldots, y_n$ arbitrary real or complex numbers (several may be the same) , there is a unique polynomial $p(x)$ of degree n or less satisfying $p(x_j) = y_j$ for all j.*

Proof: We define the kth Lagrange interpolating polynomial by

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

noting that these depend only on the x_j's, not the y_j's. $L_{n,k}(x)$ is a polynomial that takes the value 0 at $x_0, \ldots x_{k-1}, x_{k+1}, \ldots, x_n$ but the value 1 at x_k. The desired interpolating polynomial is then

$$p(x) = y_0 L_{n,0}(x) + y_1 L_{n,1}(x) + \cdots + y_n L_{n,n}(x)$$

Uniqueness follows from Theorem 1.7.2 as in the proof of 1.7.3.

From a practical standpoint although one can use the Lagrange interpolating polynomials in the proof above to calculate the interpolating polynomial however there are better methods (see a Numerical Analysis text for some.) The best way, however, is to treat this as a linear algebra problem. Letting $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ we can regard the $n+1$ equations

$$\begin{aligned} p(x_0) &= y_0 \\ p(x_1) &= y_1 \\ \vdots &\quad \vdots \\ p(x_n) &= y_n \end{aligned}$$

as $n+1$ linear equations in the $n+1$ unknowns $a_0, a_1, \ldots a_n$. You can then use one of the linear equation algorithms, e.g. Gauss–Jordan to find the coefficients.

Exercise 2.15.1 [10 points] On July 20, 1989 the O'Hare temperatures were 67 at Midnight, 66 at 6AM, 69 at Noon and 66 at 6PM. Thinking of midnight as hour 0, 6AM as hour 6, 6PM as hour 18 etc find a degree 3 polynomial which interpolates this data, i.e. $p(j) = y_j$ where y_j is the temperature at hour j. Using interpolation estimate the temperature at 4PM (hour 16). [It was actually 67.] Now using extrapolation, estimate the temperature at 2PM on July 21, 1989 (i.e. hour 38). [The actual temperature was 70, it was a cool July week.]

Often it turns out that in a practical situation we have a lot of data but we want an interpolating polynomial of only small degree. From the uniqueness assertion of Theorem 2.15.1 it is clear that if we have d data points and want a polynomial of degree n it is probably impossible for $p(x_j) = y_j$ for all j if $d > n+1$. The next best thing (actually a better thing since we have more data) is to make $p(x_j)$ as close as possible to y_j for all j. In practical terms we wish to make

$$\sum_{j=0}^{n} (p(x_j) - y_j)^2$$

as small as possible. The polynomial that does this is called the *least squares polynomial of degree* n for this data.

One way to calculate the least squares polynomial for some data is to minimize the sum above using calculus. That people would do this in practice (for degree greater than 1) is as unbelievable as people actually using Graeffe's method in the 21st century. There are nice methods using linear algebra, some dating back to Gauss in the early 19th century. In fact both Gauss and Jordan (an obscure Jordan, not the famous Jordan) worked on their Gauss-Jordan method not to solve ordinary systems of equations but least squares problems. For more information on these methods consult a good Linear Algebra text, for example the text by Gilbert Strang.

$\boxed{\textbf{Maple Implementation}}$

The Maple procedure `interp` solves the interpolation problem. For example, to find a polynomial $p(x)$ of degree 3 with $p(0) = 18, p(1) = 15, p(3) = -2$ and $p(5) = 17$ one would enter

```
p:= interp([0,1,3,5],[18,15,-2,17],x);
```

The least squares problem can be solved using Maple by either using the `regression` procedure in the `stats` package or by setting up an under-determined system of linear equations and solving by using the `leastsqrs` procedure in the `linalg` package.

One can also generalize by setting conditions on the derivatives, second derivatives etc. to find polynomials displaying desired features, such as roots of various multiplicity. The important thing is that to generate a polynomial of degree n one should have $n + 1$ conditions. But only n of these can involve derivatives, $n - 1$ second or higher derivatives, and so on. Also at least one condition must not involve a value being zero, otherwise the zero polynomial will be your solution. One gets a system of linear equations which can be solved by one of Maple's solving procedures. You should be warned, however, that the coeficients and/or values on the graph of the polynomial may be large or weird. To try to get a nice example with integer coefficients and reasonable size values the Maple `lattice` procedure may be helpful as in the following example. Given a real matrix with independent rows the `lattice` procedure attempts to find a basis for the rowspace consisting of "small" vectors which are integer linear combinations of the original rows, we will see more applications of this procedure in Chapter 5.

Maple Implementation

Here we wish to find a polynomial $p(x)$ of degree 4 with root of multiplicity 2 at $x = 2$, a turning point at $x = 0$ and a turning point at $x = -1$ with as small a y value as possible. Thus we want the value of the polynomial at 2 to be zero, and the derivative to be zero at $x = 2, 0$ and -1. To avoid the zero polynomial as our solution we ask that the value $p(-1)$ not be zero, but be as small as possible. We actually also want our second derivatives at $x = -1, 0$ and 2 to not be zero, but we will trust to luck that this will happen.

The condition that the derivative be zero at $x = 0$ tells us the linear coefficient must be zero so our polynomial can be written

$$p(x) = ax^4 + bx^3 + cx^2 + d$$

Our conditions then give the following equations:

$$\begin{array}{rcrl} p(2)=0 & \quad & 16a+8b+4c+d &= 0 \\ p'(2)=0 & & 32a+12b+4c \quad &= 0 \\ p(-1)=s & & a-b+c+d &= s \\ p(-1)=0 & & -4a+3b-2c \quad &= 0 \end{array}$$

Where s is to be a small, in absolute value, non-zero number. We set up a matrix as follows. Note the first 4 columns refer to the variables a, b, c, d and the last 4 are the equations. We multiply those equations where we want to get zero on the right by a large number (here 10) and those that we want small on the right by 1 or a not quite so large number. We enter the followng 4 x 8 matrix in Maple.

```
> A:=[[1,0,0,0,160,320,1,-40],
     [0,1,0,0,80,120,-1,30], [0,0,1,0,40,40,1,-20],
     [0,0,0,1,10,0,1,0]]:
```

Now we apply the procedure `lattice` which will provide us with ”small” vectors in the rowspace. The output is shown:

```
> lattice(A);

  [[0,0,0,1,10,0,1,0], [1,-1,-5,12,0,0,9,30],

   [1,-1,-4,8,0,40,6,10], [-2,3,7,-20,0,0,-18,30]]
```

Unfortunately we don’t get a solution, i.e., a vector with 0 in the 5,6 and 8th column. So we try again, multipy the 5,6 and 8th equations by a larger number, now 100, we are trying to force these entries, which would now have to be multiples of 100, to actually be zero. Apply `lattice` again:

```
> A:=[[1,0,0,0,1600,3200,1,-400],
   [0,1,0,0,800,1200,-1,300], [0,0,1,0,400,400,1,-200],
   [0,0,0,1,100,0,1,0]]:

> lattice(A);
```

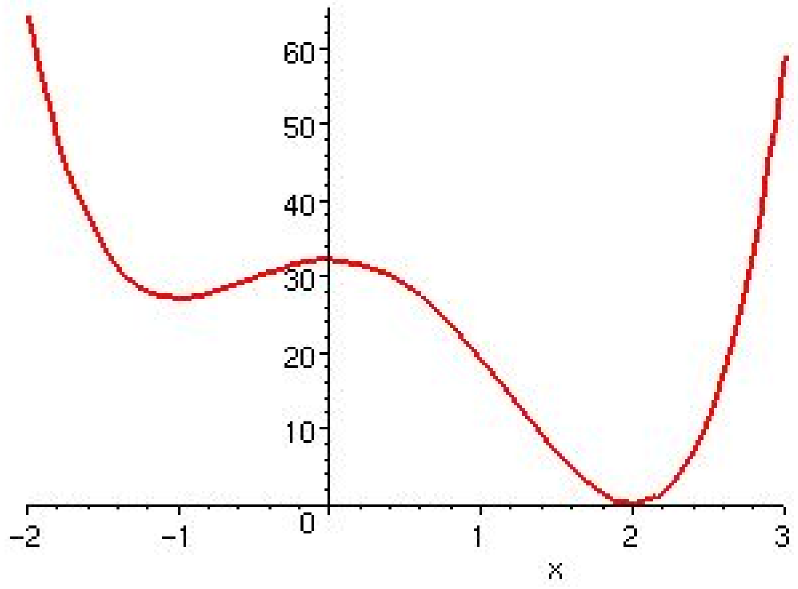
```
[[-3,4,12,-32,0,0,-27,0], [0,0,0,1,100,0,1,0],

  [1,-1,-5,12,0,0,9,300], [1,-1,-4,8,0,400,6,100]]
```

Here we see the first vector is our desired solution: $a = -3, b = 4, c = 12, d = -32$, the small right hand side of the third equation is -27. To get a polynomial with positive leading coefficient we multiply by -1 so our answer is:

$$p(x) = 3x^4 - 4x^3 - 12x^2 + 32$$

Figure 2.3: Graph of $p(x) = 3x^4 - 4x^3 - 12x^2 + 32$


60
50
40
30
20
10
0
-2
-1
1
2
3
x