Equal opportunities

Mike Liardet investigates some equation-solving techniques, using both symbolic and numerical methods, and includes a practical and efficient Basic implementation of a numerical method that can be applied in a large number of problem areas.

Most computer programs are based in part on theories that include some mathematical equations. For example, financial applications encode the equations for profit, loss, compound interest, and so on. Many games and graphics programs use the equations of projective geometry. Scientific software typically borrows the equations of physics, with anything from an elementary Fahrenheit to Celcius formula through to the equations of atomic theory and beyond. Even a simple accounting program must encapsulate the equations for VAT.

In many cases it's possible for the programmer to solve the equations while the program is being written, hard-wiring his solution explicitly in the code with the appropriate formulae. Experienced programmers do this almost subconsciously, forgetting that they are actually implementing something that's based on a firm mathematical theory. For example, the equations for VAT might be written in this form:

```
VAT_on_Goods = Invoice_Total -
  Goods_Value
VAT_on_Goods = Goods_Value * VAT_Rate
VAT_Rate = .175 (at present!)
```

For a typical invoicing system it is likely that, when the program needs to work with these equations, the Goods_Value is already known (it is simply the sum of the prices of all the items on the invoice, which were presumably typed in by the user) and that the VAT_on_Goods and Invoice_Total are the unknowns. In this context these are the values for which the equations must be solved. With the aid of a trivial manipulation of the first equation and some straightforward re-ordering of the equations themselves, the VAT calculations can be (pseudo-) coded as:

```
Sub VATCalc(Goods_Value,VAT_on_Goods,
        Invoice_Total)
  Set VAT_Rate = .175
  Set VAT_on_Goods = Goods_Value *
        VAT_Rate
  Set Invoice_Total = Goods_Value +
        VAT_on_Goods
End
```

This may be enough for most accounting software, but it's interesting to note that this simple little routine does only half the job of the original equations, since it can work in only one direction. Specifically it can only determinine VAT_on_Goods and Invoice_Total when given Goods_Value, but it is unable to determine, for example, Goods_Value, given Invoice_Total. This is something that a VAT Inspector may want to know, and the only way to provide it with a conventional program is for the programmer to code another routine to provide this functionality, working with the same set of equations as before, but using them in a different way.

For the small set of VAT equations above it is simple enough for the programmer to hand-solve them and then implement all the necessary routines to support the application. For more complicated situations this may not be feasible. Either the formulae may become too complicated for reliable hand solution, or the "direction" of use may not be known in advance — for example, spreadsheet systems often have a "back-solver", where the user can specify any combination of variables in a formula as input, to derive the remainder as output.

In these situations the program itself needs equation-solving capability. There are two ways of tackling the problem: by symbolic manipulation or by a numerical method. A symbolic solution generally provides 100 percent accurate answers, whereas there is always an error factor with a numerical method. Neither approach guarantees a solution in every case — symbolic manipulation can get overloaded by the complexities of symbol manipulation, and numerical methods can miss the mark either because of accumulated rounding errors, or else because the underlying iterative method diverges away from the theoretical solution instead of converging onto it.

Outside of the Artificial Intelligence laboratories there is comparatively little equation-solving software that uses symbolic manipulations; Wolfram's Mathematica is one notable exception. Most other equation-solving software uses numerical techniques. There are a number of reasons for this, not least because most of the common programming languages (e.g. C, C++, Cobol, Fortran, Basic) are better equipped to handle numbers rather than symbols.

In this month's Low Level we will explore some of the equation-solving techniques, using both symbolic and numerical methods, and finish off with a practical and efficient Basic implementation of a numerical method that can be applied in a wide number of problem areas.

To get a flavour of what's involved we will start work with a set of three simple simultaneous equations (Fig 1), which are to be solved for the three unknowns: x, y and z. To solve these equations symbolically by computer, it is essential to first represent them in an appropriate data structure, such as the "nested list" which is a standard construct in programming languages like Lisp or Prolog. Fig 2 shows how these equations can be converted into lists.

Once held in list form it's much easier to operate on the equations, but this begs the question of what operations can usefully be performed on them. There are various rules which are commonly used when attempting to solve equations. Fig 3 outlines the common manipulations, although we have glossed over some of the mathematical niceties: for example, you have to be careful not to divide equations by zero. To show how these rules are used in practice the solution process for the equations is given in Fig 4.

Using rule 1 we can see that there are three variables and three equations, so it's likely that it will in fact be possible to reach a solution. Fewer equations would leave the problem under-constrained, with not enough information to reach a solution. More equations would over-constrain the problem, and, unless there was some redundancy (such as one of the extra equations being a trivial variant of another equation) there is likely to be an inconsistency with no solution possible.

Fig 1 *A set of three simultaneous equations, with three unknowns: x, y and z*

```
x y z  = 450      (A)
5 y² + 5 x y = 50 x    (B)
3 x² - 30 x = -75      (C)
```

Fig 2 *A Lisp-like internal representation of the equations in Fig 1, using list structures*

```
(= (* (* x y) z) 450)                          (A)
(= (+ (* 5 (^ y 2)) (* (* 5 x) y)) (* 50 x))   (B)
(= (- (* 3 (^ x 2)) (* 30 x)) 45)              (C)
```

Fig 3 *Ten rules of thumb for solving equations*

1. To be solvable the number of equations should equal the number of unknowns.
2. Equations involving powers of 2 (e.g. x^2 - 7 x = - 12) have two solutions (x = 3 or 4 in this case). With powers of three there are three solutions, and so on.
3. Equations can be shuffled into any order without affecting their solution.
4. Sub-expressions within an equation can be shuffled and re-arranged in various ways, controlled by mathematical rules and identities.
5. A value, or expression (possibly including unknowns) can be added to or subtracted from both sides of any equation.
6. A value or expression can be used to multiply or divide an equation. This is done by applying it to both sides simultaneously.
7. The left- and right-hand sides of an equation can be swapped.
8. Two or more equations can be added (or subtracted) to make a new equation by adding (or subtracting) the left-hand sides of each, giving one side of the new equation, and performing the same operation on the right-hand sides of each for the other side of the new equation.
9. Two or more equations can be multipled together (or divided) to make a new equation by simultaneously applying the appropriate operation to the respective equation, adding (or subtracting) the left-hand sides of each for one side of the new equation, and the right-hand sides of each for the other.
10. When an unknown is isolated on one side of an equation, it can be eliminated from all the equations by replacing its every occurence in the other equations by the expression that it's equal to. The original equation need play no further part in the solution process.

Fig 4 *Solving the equations in Fig 1*

```
Divide (C) by 3
x² - 10 x = -25   (C1)
Add 25 to each side of (C1)
x² - 10 x + 25 = 0  (C2)
Rearrange LHS of (C2)
(x - 5)(x - 5) = 0  (C3)
So x = 5, substitute in (A) and (B)
5 y z  = 450      (A1)
5 y² + 25 y = 250      (B1)
Divide (B1) by 5
y² + 5 y = 50      (B2)
Subtract 50 from each side of (B2)
y² + 5 y - 50 = 0  (B3)
Apply identity
 a x² + b x + c = 0 => x = (-b +-sqrt(b² - 4 a c))/ 2 a

to (B3)

y = -5 +- sqrt(25 + 200) /2 (B4)

So y = 5 or y = -10   (B5)

Substitute in (A1)

25 z = 450 or -50 z = 450  (A2)

Divide one equation in A2 by 25 and the other by -50

So z = 18 or z = -9

Solutions: (x = 5, y = 5, z =18) or (x = 5, y = -10, z = -9)
```

Rule 2 tells us that there may be multiple solutions for x and for y as well. Most of the other rules are used in the solution process itself. With an appropriate programming language, such as Lisp or Prolog, each rule can easily be encoded as an operation that can be performed on a list structure. For example, adding a quantity Q to each side of an equation (= LHS RHS), converts it to the form (= (+ LHS Q) (+ RHS Q)). List processing languages can easily manipulate lists in this fashion.

Notice that, contrary to expectations, the equations produce only one solution for x (x = 5). Equation C3 shows why — there are in fact two solutions, but they are both the same!

With a symbolic solver it's not enough simply to implement the equation manipulation rules. The solver needs some strategy to guide the solution process and apply the rules in the right order and in the right way. This is the hardest part to implement, and the area in which symbolic solvers are most likely to fail.

We won't go into the strategy in much detail here, but in general a good overall aim is to isolate any one variable on one side of one equation so that it can be substituted in all the other equations, effectively reducing the number of variables by one. If all goes well, after repeating this process several times, a single equation in one variable is left, and hopefully this is solvable directly, and its solution can then help get solutions for the other remaining variables. This may not always work out as planned — for example, there's a well known mathematical result that there's no general solution method for polynomials of degree 5 or more (e.g. equations like x^5 +.. = ..).

Numerical methods are less likely to founder in this way, but often require some complex participation by the user as a part of the solution process. For example, the user may have to provide a first guess for the solution, and if the guess is a bad one then the method may fail to arrive at a solution, or if there are multiple solutions it may only find the one nearest to the guess.

With a numerical method it is also sometimes necessary for the user to differentiate the equations by hand and then write a function that calculates the differential. This is a pity because it's not too difficult to implement a symbolic differentiation routine that could do the job automatically, but the programming languages commonly used to implement numerical methods cannot readily be used in this way. Firstly they are not very good at symbol manipulation in any case, but the main problem is that programs written in languages like C and Pascal are compiled before they are used, and cannot then create new compiled routines dynamically; they simply cannot be used in this way.

There is one special area of equation-solving, where a numerical method works particularly well. This is the Gaussian elimination technique, which can be applied to systems of linear equations. A linear equation can contain any number of unknowns but these can only be combined in a very simple way. There are no unknowns multiplied together and exponentiation, trigonometry or other awkward expressions are not allowed.

Fig 5 shows a typical set of simultaneous linear equations, with three unknowns: x, y and z. These equations could be solved symbolically, using the rules given in Fig 3, but Gaussian elimination formalises the process, so that the solution, if it exists, is found quickly, efficiently and accurately.

Because of the simple structure of linear equations they do need to be held in list structures and can easily be represented by arrays containing the essential coefficients. The unknowns don't need to be mentioned at all. For example, the equations in Fig 5 can be adequately represented by the array shown in Fig 6, on the understanding that each row represents an equation, with the first three columns in the array giving the coefficients of the x, y and z unknowns, and the last column giving the right-hand side values of each equation.

Gaussian elimination systematically manipulates this array representation until it uncovers the solution. Fig 7 shows roughly how it works in practice. All the operations on the equations (or rather, rows in the array) are covered by one or other of the rules we mentioned earlier, but because linear equations are so highly structured the solution process moves forward very methodically. Notice that at the third stage, when we are at the point of deriving the value of z, all the array values below the array diagonal are zero. This is the so-called "triangulated" form, from which the solution tumbles out, in a straightforward fashion as shown in the example.

There is only one major complication (see Fig 8) with the method, which we have so far glossed over by carefully constructing the problem in the first place. But suppose the second and third equations are swapped, and the order of variables is changed to x, z, y. The solution method is in danger of breaking down here, because when working on the second variable (z) the pivot element turns out to be zero, and it cannot be used to eliminate the elements below. There's a simple work-around for this: search for the most appropriate pivot element, looking down the column from the diagonal to the bottom and swap the positions of the equations so that the best pivot element is in the frame and can then be used. If all the potential pivot elements are zero then there is no solution to the equations.

So much for the theory. The code to perform the Gaussian elimination is given in Fig 9. It can work on any number of variables/equations, although the solution process inevitably slows down as the problems get bigger. This is a result of the execution time being proportional to the cube of the array size (notice the three nested loops in the routine).

Fig 5 *Three simultaneous linear equations*

```
6 x + 6 y + 12 z = 54
3 x + 2 y +  4 z = 19
2 x + 3 y +  4 z = 20
```

Fig 6 *An array representation of the linear equations*

```
[ 6   6  12  54]
[ 3   2   4  19]
[ 2   3   4  20]
```

Fig 7 *Gaussian Elimination in action.* The technique systematically manipulates the array representation of the linear equations to obtain the solution

```
[ 6   6  12  54]
[ 3   2   4  19]
[ 2   3   4  20]
```

Pivoting on the x coefficient in the first equation, eliminate x from the the equations below...

```
[ 6   6  12  54]
[ 0  -1  -2  -8]
[ 0   1   0   2]
```

Pivoting on the y coefficient in the second equation, eliminate y from the equation below...

```
[ 6   6  12  54]
[ 0  -1  -2  -8]
[ 0   0  -2  -6]
```

The last equation now gives z = 3 immediately. Eliminating z from the equations above, and eliminating this equation from further consideration...

```
[ 6   6   0  18]
[ 0  -1   0  -2]
```

The last of the remaining equations now gives y = 2 immediately. Eliminating y from the equation above...

```
[ 6   0   0   6]
```

The last remining equation gives x = 1 immediately. The equations are solved

Fig 8
After a minor rearrangement of the equations there are problems with pivoting, when a zero element is found in the critical position.

```
6 x + 12 z + 6 y = 54
2 x +  4 z + 3 y = 20
3 x +  4 z + 2 y = 19
```

The equations have been rearranged, but their solution is the same. The array representation is:

```
[ 6  12   6  54]
[ 2   4   3  20]
[ 3   4   2  19]
```

As before, pivoting on the x coefficient in the first equation, eliminate x from the the equations below:

```
[ 6   6  12  54]
[ 0   0   1   2]
[ 0  -2  -1  -8]
Pivoting on the (second) z coeffi-
cient in the second equation is not
now possible as it is zero...
```

Fig 9 *A subroutine in Basic, that performs Gaussian elimination*

```
Sub Gauss (N As Integer, A() As Single, xans() As Single)
'Do Gaussian elimination on a(1..N,1..N) where RHS is in a(1..N,N+1)
'Answer is given in xans(1..N)
Dim i As Integer
Dim j As Integer
Dim k As Integer
Dim mx As Integer
Dim t As Single

'Eliminate...
   For i = 1 To N
      mx = i
      For j = i + 1 To N
         If Abs(A(j, i)) > Abs(A(mx, i)) Then mx = j
      Next j
      For k = i To N + 1
         t = A(i, k): A(i, k) = A(mx, k): A(mx, k) = t
      Next k
      For j = i + 1 To N
         For k = N + 1 To i Step -1
            A(j, k) = A(j, k) - A(i, k) * A(j, i) / A(i, i)
         Next k
      Next j
   Next i
'Substitute...
   For j = N To 1 Step -1
      t = 0#
      For k = j + 1 To N
         t = t + A(j, k) * xans(k)
      Next k
      xans(j) = (A(j, N + 1) - t) / A(j, j)
   Next j
End Sub
```

PCW Cover Disk
The full code for this month's Low Level is on the cover disk given with this issue of *PCW*.

PCW Contacts

Mike Liardet is a freelance programmer and writer. He can be contacted via the *PCW* editorial office or on email as **mliardet@cix.compulink.co.uk**