Orthodrop
an
Introduction for a general audience

by
Douglas Moreman

On your PC you can now solve systems of 1000 linear equations in 1000 unknowns. You can do this with an algorithm that you can program and modify yourself. There is an opportunity for enterprising experimentalists to discover improvements in this new method.

The new Orthodrop Method is introduced here in comparison with the venerable Gauss-Seidel Method. Both methods are implemented herein in BASIC. Some practical problems give rise to huge systems of equations whose coefficients are mostly zeros. For these problems, we apply the methods in a way that saves memory by not storing the zeros.

EXAMPLE.

A small illustration can show how an engineering problem might give rise to a huge, sparse (mostly zeros) system of linear equations. To find by how much a machine part may expand or contract, changing critical clearances, one may wish to know the temperature distribution on that part under given conditions. As a two-dimensional introduction to heat problems, suppose that the temperatures around the boundary of a rectangle are held to known values and the eventual, "steady-state", temperatures T(x,y) at points (x,y) in the interior are desired.

To solve this problem, the engineer can use a result of the law that the rate of change of temperature, T, $\nabla^2 T$. You need not follow all this example, in this paragraph, to appreciate most of the remained of this article. $\nabla^2 T$ is $\partial_{11}T + \partial_{22}T$ and, at steady state, is 0 at every point. $\partial_{11}T$ denotes the second derivative of T in a direction parallel to one pair of edges of the rectangle, and $\partial_{22}T$ is the second derivative of T parallel to the other pair of edges of the rectangle. We create a grid of "nodal points" on the rectangle, label the points (x_1,y_1), (x_2,y_2) and so on, and, for each point create an approximating equation based on the equilibrium law given above.

I'll skip the details. The important observation here is that if you have a 10 by 10 inch rectangle and put in a 1 by 1 inch grid, you'll get 81 equations in 81 unknowns, though each of those equations will be sparse, involving no more than 5 of the unknowns. Increasing the resolution to a 0.1 by 0.1 grid generates a much larger system having 9,801 unknowns, no more than 5 of which occur in any one of the 9,801 equations.

TWO METHODS.

The famous Gauss Method fails on sufficiently large problems, due in part to excessive storage requirements. More insidiously, "round-off error" inherent in all computers, can cause the Gauss Method to give wrong answers. The next most famous method, misnamed "Gauss-Seidel" is self-correcting of round-off errors and converges rapidly to the solution for many practical problems but explodes catastrophically for many more.

Let's begin with a simple example of two linear equations in two unknowns:

$$A_{11}\cdot X_1 + A_{12}\cdot X_2 = B_1$$
$$A_{21}\cdot X_1 + A_{22}\cdot X_2 = B_2.$$

where the only unknowns are X_1 and X_2. Assume that our system has exactly one solution (X_1,X_2).

It is easy to imagine the graphs of our two equations. They are two straight lines H_1 and H_2 that meet at the unknown point $X=(X_1,X_2)$.

The Gauss-Seidel algorithm in Two Space is: start with some point P_0 then repeat the following steps in a loop {move parallel to Axis 1 to a point on H_1 then move parallel to Axis 2 to H_2}. Figure 1 illustrates how the points of successive iterations of the Gauss-Seidel Method can converge and, importantly, how they can race off "towards infinity" if the equations are not in an appropriate order. In large systems, finding a successful order can be, practically, impossible. I have computed that if the number of equations is 1000 then the number of possible orderings exceeds the number of atoms in the Milky Way!

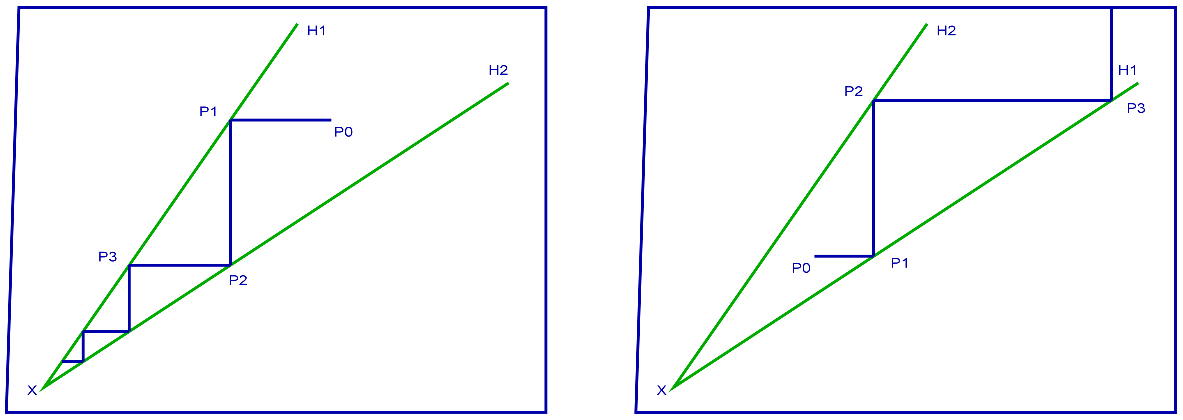

Figure 1

[Text for Figure 1: Success of the Gauss-Seidel Method depends upon a lucky order of labeling the given equations.]

<u>ORTHODROP.</u>

The Orthodrop Method, illustrated in Figure 2, seems to solve a broader set of problems.

By successively dropping perpendiculars from one line to the next, the "orthodrops" P_n approach the point of intersection. This fact is independent of the ordering of the equations.

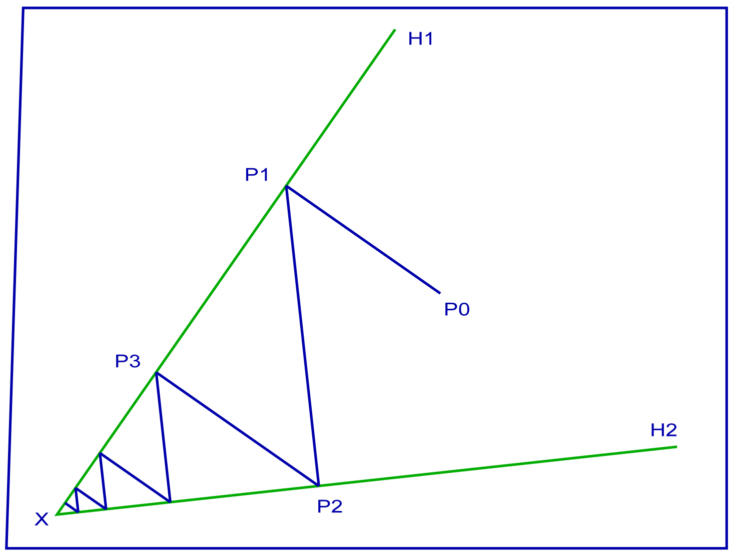

Figure 2

Using the Orthodrop Method on our system of two equations, you proceed like this:

Let P_1 denote any point on H_1 except X.

Let P_2 be the **orthodrop** of P_1 onto line H_2. That is, P_2 is the point of H_2 such that the line through P_1 and P_2 is perpendicular to H_2. P_2 is also the closest point of H_2 to P_1.

Let P_3 denote the orthodrop of P_2 onto H_1.

Let P_4 denote the orthodrop of P_3 onto H_2.

And so on... .

The idea is simple and seems to work in all cases for which there is a unique solution, regardless of the order in which the equations are listed. But how do we implement it for n unknowns?

We will use the concept of the "dot product" $C \bullet D$ of a "point" $C=(C_1,C_2,...,C_n)$ with point $D=(D_1,D_2,...,D_n)$: the sum of the products $C_k \cdot D_k$ for k=1 to n. For a system of n-linear equations of the form $A_{k1} \cdot X_1 + A_{k2} \cdot X_2 + ... + A_{kn} \cdot X_n = B_n$, each equation may be written more compactly in the form $A_k \bullet X = B_k$. The graph of that equation in n-Space is an object H_k, which is a line if n=2 (in Two Space) and is a plane if n=3 (in Three Space). I cannot visualize H_k when n>3, but I know that H_k is very much like a plane (and is called a "hyperplane").

The orthodrop of a point P onto H_k is the point $P+t*A_k$ where t is the number $(B_k-A_k \bullet P)/A_k \bullet A_k$, called the orthodrop coefficient.

Explaining why this simple formula is true would require a diversion into linear geometry which I will skip.

In the Orthodrop Method you start with any point P, let the new P be the orthodrop of the current P onto H_1, then let the new P be the orthodrop of the current P onto H_2, and so on. From H_n, P is dropped back onto H_1 to resume the cycle.

It would be nice to stop the cycling when the current point P differs from the solution-point X by some amount that we consider small enough. But, not knowing X, we cannot know that difference. We will use here a "stopping criterion" that is simple, but not always accurate. We want to find a point X so that the matrix product AX is equal to the point B. We will stop our loop when the maximum deviation of the terms of AP from the corresponding terms of B is less than some chosen number Er. (The smaller you make Er, the longer the program will run. If Er is small enough the program may, due to round-off errors, never stop itself).

"AP" in the preceding paragraph means the product of the square matrix A and a point P having exactly as many, namely n, coordinates as A has columns. AP is a point with n coordinates, the k-th of which is the dot product, A_k●P, of the k-th row of A with the point P. The k-th row of A may be denoted by (A_{k1},A_{k2},...,A_{kn}), and is called a "**normal** vector for the hyperplane H_k." Letting O denote the origin, Line OA_k is perpendicular or "normal" to H_k.

Listing 1 is a nearly complete, but overly-simple implementation of the Orthodrop Method. It is intended to be read for understanding and insight. It will run, but it is not suitable for huge, sparse systems and has no mechanism for increasing the speed of convergence towards the solution.

[Listing One]

```
' Ortho1.bas by Douglas Moreman 1988, 89, 90, 91.
' The language is a version of BASIC
DEFINT C, I-N, R ' Make some integer-variables.
DIM A(50, 50), P(50), B(50)' set aside some memory.
CLS  'clear the screen
GOSUB GetData
K = 0: FinishedYet = 0  'The variable K indexes the equations.
WHILE FinishedYet = 0 'This loop illustrates the Orthodrop Method
  K = K + 1: IF K > NumEqs THEN K = 1
  ' Recall that Ak is a row of the matrix of coefficients.
  ' Ak is read "A sub k".
  ' Compute Ak●P :
    AkDotP = 0
    FOR J = 1 TO NumXs
      AkDotP = AkDotP + A(K, J) * P(J)
    NEXT J
  t = B(K) - AkDotP  ' Since Ak●Ak = 1, as mentioned in GetData,
                  ' there is no need to divide by Ak●Ak.
  ' Compute new P as P + t*Ak:
    FOR J = 1 TO NumXs  ' NumEqs may exceed NumXs
      P(J) = P(J) + t * A(K, J)
    NEXT J
  GOSUB CheckError   ' This can change FinishedYet.
  GOSUB PrintP       ' You can write this yourself.
WEND  'Short, isn't it?
END

CheckError: ' This subroutine determines whether the maximum
' difference between a term of the point AP and the corresponding
' term of B exceeds Er.
' The R and C used below remind me of Row and Column.
FinishedYet = -1
FOR R = 1 TO NumEqs
  ArDotP = 0
  FOR C = 1 TO NumXs
    ArDotP = ArDotP + A(R, C) * P(C)
  NEXT C
  E = ArDotP - B(R)  'compare the Rth term of AP to that of B
  IF E < -Er OR E > Er THEN
    FinishedYet = 0
    EXIT FOR
  END IF
NEXT R
RETURN

GetData
 ' Write this subroutine yourself:
 ' Get NumEqs, the number of equations which is ordinarily
 ' the same as NumXs, the number of unknowns, and get
 ' the coefficients A(K,J), the constants B(K) and
 ' the stopping number Er.
 ' Normalize the rows of A by dividing each equation by the
 ' square root of the sum of the squares of the terms of A in
 ' that row. Each row Ak of A will now be such that Ak●Ak = 1.
 ' Each term of the first approximate P will be zero by
 ' default.
RETURN
```

[End of Listing One]

The rate at which the solution is approached can be fast or excruciatingly slow. To see how it can be slow, imagine a two-equation case, as in Figure 1, in which the two intersecting lines meet in a small angle. The smaller the angle, the slower the rate of convergence.

SAD NEWS/OPPORTUNITY.

Ironically, on some classic test systems Orthodrop is much slower than is Gauss-Seidel: in the simple heat example used in this paper, the superiority of the Gauss-Seidel Method increases with the number of equations. This may be a reason that Orthodrop, though probably re-invented by many people in the past 100 years, has not been properly appreciated and is not in text books. Fortunately, Orthodrop blazed through the engineering problem I invented it to solve, so I was encouraged, while earlier discoverers may have been discouraged. I have been seeking, with partial success, ways to accelerate convergence on classic test problems. You may wish to try your hand at this yourself.

EXTRAPOLATION.

In my own experimentation, I tried a simple method that, in my test problems, decreases the time of convergence: LNP extrapolation (linear-near-point). It is based on the idea that if P and Q are two approximates to X that occur on the same hyperplane H_k, and Q was obtained before P then Ray QP, the ray from Q through P, runs in a direction more or less towards X.

We guess at the point of Ray QP that is nearest to the solution point X. Points of Ray QP are of the form

Q + t*(P-Q) where t is a number not less than 0.

My guess for the best t is

[A(Q-P)●(AQ-B)]/[A(Q-P)●A(Q-P)] .

Again I will omit an explanation from linear geometry. But, if the subject is new to you, just try to see that as t increases from 0, the point Q + t*(P-Q) starts at Q and moves along Ray QP, passing P as t passes 1.

LNP extrapolation in the program uses points on H_1, the graph of the first equation. But, I have found by experimentation that time-to-convergence is usually reduced by not extrapolating at every return of P to the plane of extrapolation.

INDEXING.

The program in Listing 2 uses a one-dimensional array A to store the coefficients of the system and avoids storing those coefficients that are zeros. For huge systems, this method is more efficient than using a two-dimensional array when at least half of the coefficients are zeros.

The non-zero coefficients are stored in a long, one-index array A(). For each of the original equations, say Equation K, the starting position in A() for Row K is stored in IndexOfFirstTerm(K) and the number of non-zeros of Row K is NumOfNonZeros(K). So, A(IndexOfFirstTerm(2)) stores the value of the first non-zero term of Row 2 of the original system. The non-zero terms of the original A are counted by the variable Index and, the value of that term is stored in A(Index) and the column occupied by that term in the original system is stored in Column(Index).

Suppose that the first two rows of the original system of coefficients (excluding the constants to right of the "=") are:

0	3	0	5	0	7	0
0	9	0	0	0	0	4

The numbers stored to represent these two original rows are:
A(1)=3, A(2)=5, A(3)=7, A(4)=9, A(5)=4,
Column(1)=2, Column(2)=4, Column(3)=6, Column(4)=2, Column(5)=7,
IndexOfFirstTerm(1)=1, IndexOfFirstTerm(2)=4,
NumOfNonZeros(1)=3, NumOfNonZeros(1)=2.
This information is sufficient to re-create the two given rows exactly.

In BASIC, "real" variables require 4 bytes and integers require 2. For a system with 1000 equations averaging no more than 5 non-zero terms each, the two-index A(Row,Col) method requires 1,000,000*4 bytes of memory but the indexing method shown here requires 5*1000*4 + 5*1000*2 + 1000*2; a reduction, in this case, of 99%.

The data in Listing 2 comes from a heat problem. This problem is one of those classics on which Gauss-Seidel is much faster than Orthodrop. There are problems on which the reverse is true, but I offer you the given one as a challenge. An opportunity exists for programmer-experimenters to invent various modifications of the Orthodrop Method that make it faster in general and also for special classes of problems. I would like to hear from interested reader/experimenters and may be able to send you details omitted here for brevity.

Hints: 1) Orthodrop works even when there are more equations (redundant) than unknowns and introducing a clever linear combination of the original equations can increase the speed of convergence, and 2) the LNP extrapolation factor tends to be stable and need not be recomputed at every extrapolation.