Conjugate Gradient Tutorial

Jon F. Claerbout

ABSTRACT

The conjugate-gradient method is a general purpose simultaneous equation solving method ideal for geophysical inversion and imaging. A simple form of the algorithm iteratively searches the plane of the gradient and the previous step.

INTRODUCTION

The solution time for simultaneous linear equations grows cubically with the number of unknowns. For equations with hundreds of unknowns the solutions require minutes to hours. What can we do about geophysical problems with unknowns numbering in the millions? The number of unknowns somehow must be reduced by theoretical means, or else we must turn to numerical approximation methods. A numerical technique known as the *conjugate-gradient method* provides good approximations.

The conjugate-gradient method is an all-purpose optimizer and simultaneous equation solver. It is useful for systems of arbitrarily high order because its iterations can be interrupted at any stage and the partial result is an approximation that is often useful. Like most simultaneous equation solvers, the exact answer (assuming exact arithmetic) is attained in a finite number of steps.

The conjugate-gradient method is really a family of methods. There are perhaps a dozen or more forms of the conjugate-gradient algorithm. The various methods differ in treatment of underdetermined systems, accuracy in treating ill-conditioned systems, space requirements, and numbers of dot products. Conjugate-gradient methods are an active area of computer science research. A popular program by Michael Saunders has 625 lines. My plan here is to present the geometrical concepts and a one-page program. The program should handle most practical cases. Since you will understand the one-

page program, it will be convenient for alterations such as making snapshots of the residual during the descent, or adapting to nonlinear optimization. The principal disadvantage of the program presented here is that it uses more auxiliary memory vectors than most other versions.

CHOICE OF DIRECTION

Any collection of search lines can be used for function minimization. Even if the lines are random, the descent can reach the desired extremum because if the value does not decrease when moving one way along the line, it almost certainly decreases when moving the other way.

Why steepest descent is a slow way to go

Before we can understand why the conjugate-gradient method is so fast, we need to see why the steepest descent method is so slow. Visualize a contour map of a mountainous terrane. The gradient is perpendicular to the contours. Contours and gradients are *curved lines.* In the steepest descent method you start at a point where you compute the gradient direction at that point. Then you begin a *straight line* descent in that direction. The gradient direction curves away from your direction of travel, but you continue on your straight line until you have stopped descending and are about to ascend. There you stop, compute another gradient vector, turn in that direction, and descend along a new straight line. The process repeats until you get to the bottom, or until you get tired.

What could be wrong with such a direct strategy? The difficulty is at the stopping locations. The stops occur where the descent direction becomes *parallel* to the contour lines. (That is where the path becomes horizontal). So after each stop, you turn 90° from parallel to perpendicular to the local contour line for the next descent. What if the ultimate goal is at a 45° angle to your path? A 45° angle turn can not be made. Instead of moving like a rain drop down the centerline of a rain gutter, you move a fine-toothed zigzag path crossing and recrossing the centerline. The gentler the slope of the rain gutter, the finer the teeth on the zigzag path.

From the rain gutter example you see the problem of steepest descent is that you overshoot the centerline before stopping. If you must use the steepest decent method, it seems you should make one steepest descent step. After that you should undershoot the local minimum, say by 2/3 of the distance. Conventional wisdom is to overshoot all steps by about 3/2.

CONTOURS AND LINE SEARCHES

Let us minimize the sum of the squares of the components of the residual vector given by

$$\textit{residual} \;=\; \textit{data space} \;-\; \textit{transform}\quad \textit{model space} \tag{1a}$$

$$\begin{pmatrix} \\ R \\ \\ \end{pmatrix} \;=\; \begin{pmatrix} \\ Y \\ \\ \end{pmatrix} \;-\; \begin{pmatrix} \\ \mathbf{A} \\ \\ \end{pmatrix} \begin{pmatrix} \\ x \\ \\ \end{pmatrix} \tag{1b}$$

How is equation (1) related to a contour plot? A contour plot is based on an altitude function of space. The altitude is the dot product $R \cdot R$. By finding the lowest altitude we are driving the residual vector R as close as we can to zero. If the residual vector R reaches zero, then we have solved the simultaneous equations $Y = \mathbf{A}x$. In a two-dimensional world the vector x has two components, (x_1, x_2). A contour is a curve of constant $R \cdot R$ in (x_1, x_2)-space. These contours have a statistical interpretation as contours of uncertainty in (x_1, x_2) given measurement errors in Y.

How is equation (1) related to a line search? Let R be the residual at any random value of x. Suppose a random perturbation dx is proposed. If we add dx to x then the residual R gets augmented by $-\mathbf{A}\,dx$. Since we have no idea whether this perturbation is a good one or not, we multiply it by a scaling factor α which could be positive, negative, or zero. Let us find α. Let $G = \mathbf{A}\,dx$. We seek to minimize the dot product $(R - \alpha G) \cdot (R - \alpha G)$. Set to zero the derivative of the dot product with respect to α. This gives $\alpha = (R \cdot G)/(G \cdot G)$. Geometrically and algebraically the new residual $R' = R - \alpha G$ is perpendicular to the "fitting function" G.

CONJUGATE GRADIENT

In the conjugate-gradient method a line is not searched. Instead a plane is searched. A plane is made from an arbitrary linear combination of two vectors. One vector will be chosen to be the gradient vector, say g. The other vector will be chosen to be the previous descent step vector, say $s = x_j - x_{j-1}$. Instead of $\alpha\, g$ we need a linear combination, say $\alpha\, g + \beta\, s$. For minimizing quadratic functions the plane search requires only the solution of a two-by-two set of linear equations for α and β. The equations will be specified along with the program. (For *nonquadratic* functions a plane search is considered intractable, whereas a line search proceeds by bisection).

Magic

There are a number of properties of the conjugate-gradient approach that are well known, but harder to explain. Luenberger's book is a good place to look for formal explanations of this magic. (His book also provides other forms of the conjugate-gradient algorithm). Another helpful book is that of Strang [1986]. Known properties are:

1. The conjugate-gradient descent method gets the exact answer (assuming exact arithmetic) in n descent steps, where n is the number of unknowns.
2. Since it is helpful to use the previous step, you might wonder why not use the previous two steps, since it is not hard to solve a three-by-three set of simultaneous linear equations. It turns out that the third direction would not help. The distance moved on the extra direction would be zero.

Conjugate-gradient theory for programmers

Fourier transformed variables are often capitalized. Here we will let vectors transformed by the $\mathbf{A}$ matrix be capitalized. A *matrix* such as $\mathbf{A}$ is denoted by **boldface** print. For example:

$$X = \mathbf{A}\, x \tag{2a}$$

$$S_j = \mathbf{A}\, s_j \tag{2b}$$

$$G_j = \mathbf{A}\, g_j \tag{2c}$$

Obviously a linear combination in solution space, say $s+g$, corresponds to $S+G$ in the conjugate space. The residual is

$$R = Y - \mathbf{A}\, x = Y - X \tag{3}$$

The solution x is obtained in steps s_j.

$$x = s_1 + s_2 + s_3 + \cdots \tag{4}$$

The last stage of each iteration is to update the solution and the residual.

$$\begin{aligned} &\text{solution update:} && x \leftarrow x + s \\ &\text{residual update:} && R \leftarrow R - S \end{aligned}$$

The gradient g is a vector of length of x. One such vector is

$$g = \mathbf{A}^{\mathrm{T}} R = \textit{gradient} \tag{5a}$$

$$G = \mathbf{A}\, g = \textit{conjugate gradient} \tag{5b}$$

You don't need to know what g is the gradient of, but if you would like to learn, note that

$$-\frac{\partial}{\partial x^*}\,(Y^* - x^*\,\mathbf{A}^*)\,(Y - \mathbf{A}\,x) \;=\; \mathbf{A}^*\,R \tag{6}$$

In dot product notation, the plane spanned by the gradient and the previous step vector is searched when we minimize with respect to α and β the quadratic:

$$Q\,(\alpha,\,\beta) \;=\; (R - \alpha G - \beta S)\cdot(R - \alpha G - \beta S) \tag{7}$$

The minimum is found at $\partial Q\,/\partial\alpha = 0$ and $\partial Q\,/\partial\beta = 0$, namely

$$0 \;=\; G\cdot(R - \alpha G - \beta S) \tag{8a}$$

$$0 \;=\; S\cdot(R - \alpha G - \beta S) \tag{8b}$$

You are ready to tackle the program.

PROGRAM

```
# minimize   res(m) = y(m) - aaa(m,n) * x(n)   by conjugate gradients.

subroutine cg( n, x, g, s, m, y, res, gg, ss, aaa, niter)
integer i, j, n, m, iter, niter
real x(n), y(m), res(m), aaa(m,n)
real g(n), s(n), gg(m), ss(m)          # space vectors
real dot, sds, gdg, gds, determ, gdr, sdr, alfa, beta
do i = 1, n
     x(i) = 0.              # clear solution
do i = 1, m
     res(i) = y(i)               # load the residual vector
do iter = 0, niter {             # niter is the specified number of iterations
     do j = 1, n {
          g(j) = 0.          # g = A^T * res = grad
          do i = 1, m
               g(j) = g(j) + aaa(i,j) * res(i)
          }
     do i = 1, m {
          gg(i) = 0.         # gg = A * g = conjugate gradient
          do j = 1, n
               gg(i) = gg(i) + aaa(i,j) * g(j)
          }
```

```
    if( iter == 0 ) {       # one step of steepest descent
        alfa =  dot(m,gg,res) / dot(m,gg,gg)
        beta = 0.
        }
    else {                  # search plane by solving 2-by-2
        gdg = dot(m,gg,gg)    #  G · (R - G α - S β) = 0
        sds = dot(m,ss,ss)    #  S · (R - G α - S β) = 0
        gds = dot(m,gg,ss)
        determ = gdg * sds - gds * gds + 1.e-30
        gdr = dot(m,gg,res)
        sdr = dot(m,ss,res)
        alfa = ( sds * gdr - gds * sdr ) / determ
        beta = (-gds * gdr + gdg * sdr ) / determ
        }
    do i = 1, n                                  # s = model step
        s(i)  = alfa * g(i)  + beta * s(i)
    do i = 1, m                                  # ss = conjugate
        ss(i) = alfa * gg(i) + beta * ss(i)
    do i = 1, n                                  # update solution
        x(i)  =  x(i) +  s(i)
    do i = 1, m                                  # update residual
        res(i) = res(i) - ss(i)
    }
return; end

real function dot( n, x, y )
integer i, n
real val, x(n), y(n)
val = 0.
do i=1, n
    val = val + x(i) * y(i)
dot = val
return;     end
```

EXAMPLE

y transpose

3.00	3.00	5.00	7.00	9.00

A transpose

1.00	1.00	1.00	1.00	1.00
1.00	2.00	3.00	4.00	5.00
1.00	0.00	1.00	0.00	1.00
0.00	0.00	0.00	1.00	1.00

```
for iter = 0, 4
x    0.43457383  1.56124675  0.27362058  0.25752524
res  0.73055887 -0.55706739 -0.39193439  0.06291389  0.22804642
x    0.51313990  1.38677311  0.87905097  0.56870568
res  0.22103608 -0.28668615 -0.55250990  0.37106201  0.10523783
x    0.39144850  1.24044561  1.08974123  1.46199620
res  0.27836478  0.12766024 -0.20252618  0.18477297 -0.14541389
x    1.00001717  1.00006616  1.00001156  2.00000978
res -0.00009474 -0.00014952 -0.00022683 -0.00029133 -0.00036907
x    0.99999994  1.00000000  1.00000036  2.00000000
res -0.00000013 -0.00000003  0.00000007  0.00000018 -0.00000015
```

REFERENCES

Luenberger, D. G., 1973, Introduction to linear and nonlinear programming: Addison-Wesley.

Paige, C.C., and Saunders, M.A., 1982, LSQR: an algorithm for sparse linear equations and sparse least squares: ACM Transactions on Mathematical Software, **8,** 43-71.

Paige, C.C., and Saunders, M.A., 1982, Algorithm 583, LSQR: sparse linear equations and least squares problems: ACM Transactions on Mathematical Software, **8,** 195-209.

Strang, Gilbert, 1986, Introduction to Applied Mathematics, Wellesley-Cambridge Press, Box 157, Wellesley MA 02181

```
/*
 * IBM to DEC floating point conversion
 *
 * ibmflt(from,to,nword,idirec)
 *
 * float from[nword] - array containing byte swapped (via swab (3)) input
 *                     32 bit IBM floating point numbers.
 * float to[nword]   - output array to contain converted floating point
 *                     values in standard internal DEC format.
 * int nword         - number of values to be converted.
 * int idirec        - conversion direction: 1 for IBM to DEC, -1 for DEC to IBM
 *
 * Author: Stewart A. Levin  3/24/83  ...decvax!hanauma!stew  4.2BSD VAX
 *
 ******************************************************************************
 #
 * Technical Discussion
 *
 * After byte swapping,  IBM floating point numbers have the 32 bit form
 *
 *
 *       gggggggggggggggseeeeeeeffffffff
 *
 *           16 bit      | 7 bit
 *         low mantissa  | exponent
 *                       |
 *                    1 bit       8 bit
 *                    sign    high mantissa
 *
 * where the value represented is
 *
 *        s     4E-256
 *    (-1)  x  2   x .FG
 *
 *
 * On the other hand, a DEC floating point number has the form
 *
 *
 *       gggggggggggggggseeeeeeeeffffffff
 *
 *           16 bit      | 8 bit
 *         low mantissa  | exponent
 *                       |
 *                    1 bit       7 bit
 *                    sign    high mantissa
 *
 *
 * and represents the value
 *
 *        s     E-128
 *    (-1)  x  2   x .1FG
 *
 * where the assumed initial mantissa bit 1 is called the "hidden normalization
 * bit" or some similar nomer.
 *
 * If we went ahead and simply treated the IBM number as a DEC number one of
```



```
 * the following two values would be obtained:
 *
 * Leading IBM (low) mantissa bit=1:
 *
 *        s     2E-127
 *    (-1)  x  2   x .FG
 *
 * Leading IBM (low) mantissa bit=0:
 *
 *        s     2E-128
 *    (-1)  x  2   x  (.1 + .FG)
 *
 * Comparing these to the value the IBM number is supposed to represent,
 * we see that the conversion can be accomplished by the formulas:
 *
 * Leading IBM (low) mantissa bit=1:
 *
 *                2E-129                2E-127
 *  multiply by  2        =  1/2  x  (2     x  .100)
 *
 * Leading IBM (low) mantissa bit=0:
 *
 *                s     2E-128
 *  subtract  (-1)  x  2    x  .1
 *
 *                2E-128              2E-128
 *  multiply by  2        =  2  x  (2     x  .100)
 *
 * The factorizations are given in terms of values that can be obtained
 * by masking all but the leading mantissa bit, exponent, and possibly
 * sign of the IBM bit pattern.
 *
 * Certain IBM number cannot be represented in DEC floating point.
 * The code allows for this happening and replaces values that have
 * overflowed with an appropriate signed maximum quantity.  Values too
 * small are replaced by zero.
 *
 */
```