Conjugate Gradient Methods for Multidimensional Optimization

Stephen D. Butalla & Valerie Kobzarenko
October 7, 2019

CSE-5400
Florida Institute of Technology
Recall the Taylor expansion of a function $f(x)$ around point $P$:

$$f(x) \approx f(P) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \cdots$$

$$\approx c - b \cdot x + \frac{1}{2} x \cdot A \cdot x$$

which is (approximately) quadratic, with

$$c \equiv f(P), \quad b \equiv -\nabla f|_P, \quad A_{ij} = \left. \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j \right|_P$$

- Gradient produces $N$ parameters; we can reduce the order of the line minimizations from direction set methods ($O(N^2)$ $\rightarrow$ $O(N)$) if we use gradients
Naïve First Guess at an Algorithm

- At point $P_0$, calculate $-\nabla f(P_0)$ and begin the line search along the “steepest” changing direction.
- Recalculating the gradient at each point $P_i$ and continue until minimum is reached.
- Known as the Steepest Descent Method.
- Results in making many small steps because the gradient at the new point $P_{i+1}$ will result in an orthogonal direction change.

Steepest Descent Method. (a) A long, narrow valley, (b) the resulting orthogonal direction change [1].
• More efficient method: Conjugate Gradient Method
• Couple the conjugate direction method like in Powell’s Method, but use gradient information instead of
• Use two sets of vectors that form a recurrence relation:

\[
g_{i+1} = g_i - \lambda_i A \cdot h_i, \quad h_{i+1} = g_{i+1} + \gamma_i h_i, \quad i = 0, 1, 2, \cdots
\]

where \( g_i, \ h_i \) satisfy the conjugate direction relations

\[
g_i \cdot g_j = 0, \quad h_i \cdot A \cdot h_j = 0, \quad g_i \cdot h_j = 0, \quad i < j
\]

and where

\[
\lambda_i = \frac{g_i^2}{h_i \cdot A \cdot h_i}, \quad \frac{g_i \cdot h_i}{h_i \cdot A \cdot h_i}, \quad \gamma_i = \frac{g_{i+1} \cdot g_{i+1}}{g_i \cdot g_i}
\]
Polak-Ribiere Method

- Polak and Ribiere modified the Fletcher-Reeves algorithm
- Set

\[ \gamma_i = \frac{(g_{i+1} - g_i) \cdot g_{i+1}}{g_i \cdot g_i} \]

- There is some evidence that choosing \( \gamma_i \) above is more efficient
- Note that the function and its gradient must be known to use this method
Basic Polak-Ribiere Example

- Define your function and compute its gradient:

\[ f(x, y) = x^2 + y^2 \]
\[ \nabla f = 2(\hat{x}x + \hat{y}y) \]

- Provide (arbitrary) initial guess of minimum: \( \mathbf{P}_0^T = (1 \ 2) \)
- Define tolerance \( \tau \) and a tolerance to distinguish between zero and a very small number, \( \nu \)

\[ f(x, y) = x^2 + y^2 \] and vector \( \mathbf{P}_0 \), produced in MATLAB
Basic Polak-Ribiere Example

- Calculate gradient at $P_0^2 = (1 \ 2)$

$$f(x, y) = x^2 + y^2$$

$$\nabla f|_{P_0} = \hat{x}2(1) + \hat{y}2(2) = \hat{x}2 + \hat{y}4$$

- Assign components of the negative gradient to the $g$ and $h$ vectors

$$g_i = h_i = -\nabla f|_i$$

- Use the line minimization algorithm to find the minimum along the first direction, which returns $f(P_{\text{lin min}})$

- If

$$2|f(P_{\text{lin min}}) - f(P_0)| \leq \tau(|f(P_{\text{lin min}})| + |f(P_0)| + \nu)$$

minimum is at $f(P_{\text{lin min}})$, return vector $P_{\text{lin min}}$

- If this statement is not true, assign $P_1 = P_{\text{lin min}}$
const Int ITMAX = 200; // maximum iterations
const Doub EPS = 1.0e-18; // used to test if function min is zero
const Doub GTOL = 1.0e-8; // tolerance for zero gradient
Doub gg, dgg;
Int n = pp.size(); // number of dimensions
p = pp; // reassign vector
VecDoub g(n), h(n); // initialize g and h vectors
xi.resize(n);
Doub fp = func(p); // copy of original function at p
func.df(p, xi); // calculate gradient and return in vector xi
for (Int j = 0; j < n; j++) {
    g[j] = -xi[j]; // negative gradient
    xi[j] = h[j] = g[j]; // flip gradient sign and store in h and gradient storage
}
// Main loop for minimization
for (Int its = 0; its < ITMAX; its++) {
    iter = its;
    fret = linmin(); // calls linmin for line minimization
    if (2.0 * abs(fret - fp) <= ftol * (abs(fret) + abs(fp) + EPS))
        return p;
    fp = fret;
    func.df(p, xi);
}
Basic Polak-Ribiere Example

• Test to see if gradient is zero (minimum)
• Define:

\[ \beta_j = \frac{|\nabla P_i^{(j)}| \ast \max(|P_i^{(j)}|, 1)}{\max(|f(P_i)|, 1)} \]

• If \( \beta_i > 0 \), compare with the tolerance variable \( \nu \) defined earlier

Doub test=0.0; // Initialize zero gradient test value
Doub den=MAX(abs(fp),1.0);
for (Int j=0; j<n; j++) {
    Doub temp=abs(xi[j]) * MAX(abs(p[j]),1.0)/den;
    if (temp > test) test=temp;
}
if (test < GTOL) return p;

• \( \beta_i \) is sufficiently small, the minimum has been found
• Loop breaks here. If not, continue to updating coefficients
Basic Polak-Ribiere Example

• Update coefficients:

Calculate $\gamma$

$$\gamma = \frac{\sum_i (\nabla f|_i + g_i) \nabla f|_i}{\sum_i g_i^2}$$

Reassign values of the vector that stores the gradient and each component of $h_i$:

$$h_i = g_i + \gamma h_i$$

• Restart line search at the new point $P_1 = P_{\text{lin min}}$
Basic Polak-Ribiere Example

// Calculate components necessary for gamma
for (Int j=0; j<n; j++) {
    gg += g[j]*g[j];
    // dgg += xi[j]*xi[j]; // Fletcher-Reeves
    dgg += (xi[j]+g[j])*xi[j]; // Polak-Ribiere
}

if (gg == 0.0) // if gradient is zero, return point p
    return p;
Doub gam = dgg/gg;
for (Int j=0; j<n; j++) {
    g[j] = -xi[j];
    xi[j]=h[j]=g[j]+gam*h[j];
}

Main minimization loop then restarts
Eventually, at one of the tests between the loops, the gradient will be sufficiently close to zero
Minimum returned from this example is (0,0)

\[ f(0,0) = 0 \]
Let's find the minimum of the function

\[ f(x, y) = \sin(x) \cos^2(y) \]
The gradient is

$$\nabla f(x, y) = \hat{x} \cos(x) \cos^2(y) - \hat{y} \cos(y) \sin(x) \sin(y)$$
Code Demonstration

- Initial guess: $P_0 = 2(\hat{x} + \hat{y})$

$$f(x, y) = \sin(x)\cos^2(x)$$
Code Demonstration

- Initial guess: $P_0 = 2(\hat{x} + \hat{y})$
- Returned minimum: $(2.0, 1.57, 0)$

$$f(x, y) = \sin(x) \cos^2(y)$$
Code Demonstration

- Initial guess: $P_0 = -\hat{x}2.75$
- Returned minimum: $(-1.58, 0, -1)$

$$f(x, y) = \sin(x) \cos^2(y)$$
Code Demonstration

- Find the minimum of:

\[ f(x, y) = \frac{\sin(x)(10 + \cos(10 + xy) \cos(x))}{10 \sin(xy)} \]
Code Demonstration

- If we zoom, a local minimum (on the domain) appears around \((x, y) = (1.5, 2.25)\)

\[
f(x, y) = \frac{\sin(x) \left( 10 + \cos(10 + xy) \cos(y) \right)}{10 \sin(xy)}
\]

\[
2.35
\]

\[
2.3
\]

\[
2.25
\]

\[
2.2
\]

\[
2.15
\]

\[
2.1
\]

\[
1.35
\]

\[
1.4
\]

\[
1.45
\]

\[
1.5
\]

\[
1.55
\]
Code Demonstration

- In 3 dimensions, we see more structure:

\[ f(x, y) = \frac{\sin(x) \left( 10 + \cos(10 + y) \cos(x) \right)}{10 \sin(xy)} \]
Code Demonstration

- Zoomed near the point of the minimum:

\[ f(x, y) = \frac{\sin(x)}{10 \sin(xy)} \left( 10 + \cos(10 + xy) \cos(x) \right) \]
Code Demonstration

- Choose an initial guess at $P_0 = \hat{x} + \hat{y}^2$

\[
f(x, y) = \frac{\sin(x) \left(10 + \cos(10 + 2y) \cos(x)\right)}{10 \sin(xy)}
\]
• Compute the gradient:

\[
\nabla f(x, y) = \hat{x} \left[ \frac{1}{10} \csc(xy) \sin(x)^{(\cos(x) \cos(xy+10)+10)} \left( \cot(x)(\cos(x) \cos(xy + 10) + 10) \\
+ \log(\sin(x)) \left( \sin(x)(- \cos(xy + 10)) - y \cos(x) \sin(xy + 10) \right) \right) \\
- \frac{1}{10} y \cot(xy) \csc(xy) \sin(x)^{(\cos(x) \cos(xy+10)+10)} \right] \\
+ \hat{y} \left[ - \frac{1}{10} x \cot(xy) \csc(xy) \sin(x)^{(\cos(x) \cos(xy+10)+10)} \\
- \frac{1}{10} x \cos(x) \sin(xy + 10) \csc(xy) \log(\sin(x)) \sin(x)^{(\cos(x) \cos(xy+10)+10)} \right]
\]

\[
\]
Code Demonstration

- Odd behavior results for very complex functions
- Input: (1.5, 2.5); returned minimum: (-2.24, -0.41)
Code Demonstration

- Troubleshooting: Input guess of (0, 0) returns ($nan, nan$)
- Need to include the complex library and pick only real component of returned value
- Problem remains; Input (2, 3), Minimum ($7.0 \times 10^{33}, 9.5 \times 10^{33}$)
Let’s find the minimum of the function

\[ f(x, y) = x^2 \log(y^2) \]
• The same issue of a returned minimum being \((nan, nan)\) persists
• The input function \(f(x, y) = x^2 \log(y^2)\) is real-valued \(\forall x, y \in \mathbb{R}\)
• \(\Rightarrow\) This algorithm is not useful for functions that have logarithms in either the function or the derivative!