# A Limited Memory Steepest Descent Method[1]

Yan Cheng[2]

April 19, 2021

[1] Fletcher, R. A limited memory steepest descent method. Math. Program. 135, 413-436 (2012). https://doi.org/10.1007/s10107-011-0479-6

[2] Department of Applied Physics and Applied Mathematics, Columbia University

# A Brief Review

From class, we know the steepest descent method computes

$$p_k = -\nabla f(x_k)$$

and uses linear search to update

$$x_{k+1} = x_k + \alpha_k p_k.$$

The SDM has certain advantages. However, it is known to be inefficient.

# Motivation

$$\underset{\text{Large-scale Nonlinear Programming}}{\text{NLP}} \Longrightarrow \underset{\text{Sequential Linear Programming}}{\text{SLP}}$$

Some reasonable requirements:

- Not have to store Hessian
- Can somehow make use of information from previous SLP iterations

Some possibilities:

- Conjugate-Gradient method
- 1-BFGS

Some possibilities:

- Conjugate-Gradient method
- 1-BFGS
- Or could we perhaps reconsider the steepest descent method?

# Some step-length choices for SDM

If $A$ is the Hessian of $f$, then

- (Line Search)

$$\alpha_k = \frac{p_k^T p_k}{p_k^T A p_k}$$

- (Barzilai-Borwein)

$$\alpha_k = \frac{p_{k-1}^T p_{k-1}}{p_{k-1}^T A p_{k-1}}$$

Here $p_k = -\nabla f(x_k)$ is the search direction in the $k$-th iteration.

# Quadratic Problem

- Problem:
$$\min\ f = \frac{1}{2}\mathbf{x}^T A \mathbf{x}$$
where $A$ is symmetric, positive definite. For the rest of this talk, denote
$$g_k = \nabla f(x_k).$$

- Using the steepest-descent update formula

$$x_{k+1} = x_k - \alpha_k g_k \iff x_{k+1} - x_k = -\alpha_k p_k,$$

  we have

$$g_{k+1} - g_k = A(x_{k+1} - x_k) = -\alpha_k A g_k.$$

  We can write

$$g_{k+1} = g_k - \alpha_k A g_k$$

- Since $A$ is symmetric, there exists an orthogonal transformation that transforms $A$ into be a diagonal matrix $\Lambda = \text{diag}(\lambda_i)$. Then component-wise,

$$g_i^{k+1} = (1 - \alpha_k \lambda_i) g_i^k, \quad i = 1, 2, \ldots, n.$$

We can make two simple observations from the formula

$$g_i^{k+1} = (1 - \alpha_k \lambda_i) g_i^k, \quad i = 1, 2, \ldots, n.$$

- If $g_i^k = 0$ for some $k$, then all subsequent $g_i^{\tilde{k}} = 0$, $\tilde{k} \geq k$.
- If $\alpha_k = \lambda_i^{-1}$ at step $k$, then $g_i^{k+1} = 0$.

Then if we choose $\alpha_i = \lambda_i^{-1}$, $i = 1, 2, \ldots, n$, we will get $g_n = 0$. So let us assume that we cannot somehow get the eigenvalues of $A$ all at once.

- Now let us introduce the limited-memory setting: suppose $n$ is larger and we store our data in a long vector of size $1 \leq m \leq n$
- Repeatedly applying the formulae

  $$x_k = x_{k-1} - \alpha_{k-1} g_{k-1}, \quad g_k = g_{k-1} - \alpha_{k-1} A g_{k-1},$$

  it is easy to see

  $$x_k - x_{k-m} \in \operatorname{span}\left\{g_{k-m}, Ag_{k-m}, A^2 g_{k-m}, \ldots, A^{m-1} g_{k-m}\right\}$$

  and

  $$g_k - g_{k-m} \in \operatorname{span}\left\{Ag_{k-m}, A^2 g_{k-m}, \ldots, A^m g_{k-m}\right\}.$$

- The columns of

$$K_m = \left[g_{k-m}, Ag_{k-m}, A^2 g_{k-m}, \ldots, A^{m-1} g_{k-m}\right]$$

is in general not orthogonal, but since $A$ is symmetric, we can apply the Lanczos algorithm to find an orthonormal basis of $K_m$.

- The Lanczos algorithm:

  Input:

  1. $n \times n$ symmetric matrix $A$
  2. some number $m$, $1 \leq m \leq n$
  3. a starting vector $q_1$

  Output:

  1. $n \times m$ matrix $Q$ whose columns $q_1, \ldots, q_m$ are orthogonal
  2. $m \times m$ matrix $T = Q^T A Q$ whose eigenvalues we will need

# The Lanczos algorithm

- 1. First, normalize $q_1$ if it is not a unit vector already
  2. Let $w_1 = Aq_1 - q_1^T A q_1$
  3. for $j = 2, \ldots, m$:
     - 3.1 Let $\beta_j = \|w_{j-1}\|$
     - 3.2 Let $q_j = w_{j-1}/\beta_j$
     - 3.3 Let $w_j = Aq_j - q_j^T A q_j - \beta_j q_{j-1}$

- The eigenvalues of

$$T = Q^T A Q$$

are known as the *Ritz values*. Since $T$ is $m \times m$, there are $m$ such Ritz values.

- The eigenvalues of
$$T = Q^T A Q$$
are known as the *Ritz values*. Since $T$ is $m \times m$, there are $m$ such Ritz values.
- It is easy to see that when $m = 1$, the only Ritz value is
$$\theta = \frac{p_{k-1}^T A p_{k-1}}{p_{k-1}^T p_{k-1}}$$
so $1/\theta$ is the Barzilai-Borwein step size.
- Also, if $m = n$, then the Ritz values are just the eigenvalues of $A$ (this is a property of the Krylov subspace)

# The Limited Memory Steepest Descent Method

- The sequence of steepest descent iterations is divided into groups of $m$ iterations, refered to as *sweeps*. Suppose $m$ Ritz values are calculated

$$\theta_{j,k-1}, \quad j = 1, 2, \ldots, m$$

- At each sweep, we use

$$x^{j+1,k} = x^{j,k} - \alpha_{j,k} g^{j,k}, \quad j = 1, 2, \ldots, m$$

where

$$x^{1,k} := x^k, \quad \alpha_{j,k} = (\alpha_{j,k-1})^{-1}, \quad g^{j,k} = Ax^{j,k},$$

and let

$$x^{m+1,k} =: x^{k+1}.$$

# Summary of the Algorithm

- 1. Fix $m$ such that $1 \leq m \leq n$
  2. Loop the following:
     2.1 Use $x$ to define $g = Ax$
     2.2 Use $(A, m, g)$ to find $m$ Ritz values $\theta_j$, $j = 1, \ldots, m$
     2.3 For $j = 1, \ldots, m$, do steepest descent with step size $\alpha_j = 1/\theta_j$

# Key Ingredients

Recall that we had

$$x_k - x_{k-m} \in \operatorname{span}\left\{g_{k-m}, Ag_{k-m}, A^2 g_{k-m}, \ldots, A^{m-1} g_{k-m}\right\}.$$

- When $m = 1$, we have the usual steepest descent with the Barzilai-Borwein step size
- When $m = n$, we have a complete set of eigenvalues for $A$
- So we are, in a sense, interpolating between the two cases
- Also, with the knowledge of $m$ Ritz values, we can jump $m$ iterations each time (from $x_{k-m}$ to $x_k$), which the author calls a "sweep"

HOME PLOTS APPS EDITOR PUBLISH VIEW

/ ▸ Users ▸ yan ▸ MATLAB

Editor – /Users/yan/MATLAB/sweep_method_quadratic.m

```matlab
function [x, fun_val, X] = sweep_method_quadratic(A, c, m, x0, epsilon)
% use steepest descent method to find the optimal solution for following
% quadratic model
%                min f(x) = 0.5 x^T*A*x - c^T*x
% input : A ------- the positive definite matrix associated with the objective
%                   function
%         c  ------ a column vector associated with the linear part of the
%                   objective function
%         x0  ----- starting point of the method
%         epsilon - tolerance parameter
%
% output : x -------- an optimal solution (up to a tolerance)
%          fun_val -- the optimal value up to a tolerance
%          X -------- contain all x_k

X = [];
X = [X x0];
x = x0;
iter = 0;
grad = A*x - c;
p = -grad;
%p = A\(-grad);
fun_val = 0.5*x'*A*x - c'*x;
for i = 1:3
    iter = iter + 1;
    %alpha = -(grad)'*p/(p'*A*p); % exact line search
    %x = x + alpha*p;
    grad = A*x - c;
    p = -grad;

    q1 = grad / norm(grad);
    [Q, H] = ritz(A, q1, m);
    theta = eig(Q' * A * Q);

    for i = 1:length(theta)
       p = - A*x;
       x = x + (1 / theta(i)) * p;
    end

    %p = A\(-grad);
    fun_val = 0.5*x'*A*x - c'*x;
    fprintf('sweep = %3d norm_grad = %2.6f fun_val = %2.6f\n', iter,norm(grad),
    X = [X x];
end
```

gradient_method_qu...

test_steepest_desce...

ritz.m

sweep_method_quad...

Command Window

```
>> test_steepest_descent

Q =

     1     0     0     0
     0     2     0     0
     0     0     3     0
     0     0     0     4

sweep =   1 norm_grad = 5.477226 fun_val = 0.012632
sweep =   2 norm_grad = 0.209852 fun_val = 0.000032
sweep =   3 norm_grad = 0.013837 fun_val = 0.000000
>>
```

Workspace

UTF-8 sweep_method_quadratic Ln 42 Col 15

Thank you!