Performance Comparison of Optimization Methods

Utkarsh Deepak
17137675

April 2018

1 Introduction

The project shows the implementation of six different optimization methods using line search algorithm. The line search algorithm finds an alpha value satisfying the Wolfe conditions and the code is based on the pseudo-code in the paper *Algorithm 851: CG DESCENT, a Conjugate Gradient Method with Guaranteed Descent* by *William W. Hager and Hongchao Zhang*. The methods are - **Steepest Descent**, **Newton's** method and four **Quasi Newton Methods** - **BFGS**, **Inverse BFGS**, **DFP** and **Inverse DFP**.

2 Implementation

The top level script sets the number of iterations, dimensions and random start points which would be the same for each method and then call each method script. After each method is called once and we obtain the time taken by each method, performance function is called to draw the stairs diagram for comparison. The objective function which we are taking into account is Rosenbrock's function. We start by creating a top level utility file.

```
% Top Level Script which sets some parameters and calls each method
% After all the methods are run, performance function perf is
    called to draw a comparison plot


n=2;% dimension of problem
N=10; % number of iterations
x0=10*rand(n,1)-5;% start points same for each method

fprintf('\nNewton\n');
Newton
ElapsedTime = tElapsed';
Iterations = counter';

```

```
fprintf('\nBFGS\n');
BFGS
ElapsedTime = horzcat(ElapsedTime,tElapsed');
Iterations = horzcat(Iterations,counter');

fprintf('\nInverse BFGS\n');
InverseBFGS
ElapsedTime = horzcat(ElapsedTime,tElapsed');
Iterations = horzcat(Iterations,counter');

fprintf('\nDFP\n');
DFP
ElapsedTime = horzcat(ElapsedTime,tElapsed');
Iterations = horzcat(Iterations,counter');

fprintf('\nInverse DFP\n');
InverseDFP
ElapsedTime = horzcat(ElapsedTime,tElapsed');
Iterations = horzcat(Iterations,counter');

fprintf('\nAll Methods run. DONE!!!!\n');

perf(ElapsedTime,N);
```

Listing 1: Main.m

As Steepest Descent method works on a very limited scope, I have not included that here. Below is the code for Perf.m

```
function perf(T,N,logplot)
%PERF    Performace profiles
%
% PERF(T,logplot)-- produces a performace profile as described in
%   Benchmarking optimization software with performance profiles,
%   E.D. Dolan and J.J. More',
%   Mathematical Programming, 91 (2002), 201--213.
% Each column of the matrix T defines the performance data for a
    solver.
% Failures on a given problem are represented by a NaN.
% The optional argument logplot is used to produce a
% log (base 2) performance plot.
%
% This function is based on the perl script of Liz Dolan.
%
% Jorge J. More', June 2004

if (nargin < 3) logplot = 0; end

colors  = ['m' 'b' 'r' 'g' 'c' 'k' 'y'];
% lines   = [':' '-' '-.' '--'];
markers = ['x' '*' 's' 'd' 'v' '^' 'o'];

[np,ns] = size(T);

% Minimal performance per solver
```

```

minperf = min(T,[],2);% dimension 2 - row wise

% Compute ratios and divide by smallest element in each row.

r = zeros(np,ns);
for p = 1: np
  r(p,:) = T(p,:)/minperf(p);
end

if (logplot) r = log2(r); end

max_ratio = max(max(r));

% Replace all NaN's with twice the max_ratio and sort.

r(find(isnan(r))) = 2*max_ratio;
r = sort(r);

% Plot stair graphs with markers.

clf;
figure(1);
for s = 1: ns
 [xs,ys] = stairs(r(:,s),(1:np)/np);
 option = ['-' colors(s) markers(s)];
 plot(xs,ys,option,'MarkerSize',3);
 hold on;
end
grid on;
title(sprintf('Performence Profile: Number of Iterations: %d',N));
legend('Newton','BFGS','Inv BFGS','DFP','Inv DFP');
legend('Location','southeast');
xlabel('Tau'); ylabel('Performence Profile');
% Axis properties are set so that failures are not shown,
% but with the max_ratio data points shown. This highlights
% the "flatline" effect.
axis([ 0 1.1*max_ratio 0 1 ]);
```

Listing 2: Perf.m

3 Conclusions

After running the code for different values of - number of dimensions n and number of iterations N, we conclude:

1. All methods produce comparable results except Steepest Descent. Plotting for $n = 2$ and $N = 10$ we can see that time taken by Steepest Descent is more than 10 times as compared to other methods. Steepest Descent doesn't compute hessian but has linear convergence.

2. Newton's Method is the best among all methods for any n and N. This is because Newton direction have a fast rate of convergence typically quadratic. The drawback of Newton is computing Hessian, which is computationally expensive.

3. Quasi Newton Methods (like Steepest Descent) don't require computation of the Hessian and are thus more efficient than Newton's method, but achieve superlinear convergence.

4. BFGS is better than DFP. Each iteration in BFGS takes $O(n^2)$ arithmetic operations (plus the cost of function and gradient operations) while for DFP it is $O(n^3)$ as solving linear systems is expensive.

5. Superiority of the BFGS algorithm over the DFP method is also due to the its robustness w.r.t the scalar γ_k becoming large. BFGS tends to correct itself after a few iterations while DFP is slow to recover.

6. Corroborated theoretically, good properties of BFGS also manifest in Inverse BFGS and bad properties of DFP are also seen in Inverse DFP.

7. We also conclude that performence of BFGS is midway between Steepest Descent and Newton's Method.

4 Appendix

Below are the codes for each method.

```
tElapsed = zeros(1,N); % Time taken for each outer loop
counter = zeros(1,N);% Inner Loop counter for each outer iteration
for i = 1:N
    tStart = tic;  % Start Time counter for i
    fprintf('Iteration Number = %i\n',i);
    x=x0;
    [f,g] = obj(x);% Evaluating values of function and gradient

    k = 0;
    alpha=[];
    K=abs(f);
    tol=sqrt(eps);
    df=1;
    ndx=1;

    count = 0;
    while norm(g)>tol && df>100*K*eps && ndx>tol
        count = count + 1;
        p = -g; % Steepest Descent
        [alpha]=ls_V2(k,x,p,alpha);% Line Search algorithm
        dx = alpha*p;
        ndx = norm(dx);
        x = x+dx;

        [fnew,gnew] = obj(x);
        df = abs(fnew-f);

        p = -gnew;
        if p'*gnew>=0
            fprintf('Not a descent direction, p''*g: %g\n', p'*gnew)
            disp('Finished processing this problem.');
            break
        end
        f = fnew;
        g = gnew;
        k = k+1;
    end
    tElapsed(i) = toc(tStart);  % Stop Time counter
    counter(i) = count;
end
```

Listing 3: SteepestDescent.m

```matlab
tElapsed = zeros(1,N); % Time taken for each outer loop
counter = zeros(1,N);% Inner Loop counter for each outer iteration
for i = 1:N
    tStart = tic;  % Start Time counter for i
    fprintf('Iteration Number = %i\n',i);
    x=x0;
    [f,g,h] = obj(x);% Evaluating values of function, gradient and hessian

    k = 0;
    alpha=[];
    K=abs(f);
    tol=sqrt(eps);
    df=1;
    ndx=1;

    count = 0;
    while norm(g)>tol && df>100*K*eps && ndx>tol
        count = count + 1;
        p = -inv(h)*g;% Newton direction method
        [alpha]=ls_V2(k,x,p,alpha);% Line Search algorithm
        dx = alpha*p;
        ndx = norm(dx);
        x = x+dx;

        [fnew,gnew,hnew] = obj(x);
        df = abs(fnew-f);

        p = -inv(hnew)*gnew;
        if (p'*gnew>=0)
            fprintf('Not a descent direction, p''*g: %g\n', p'*gnew);
            fprintf('Finished processing this iteration\n');
            break
        end
        f = fnew;
        g = gnew;
        h = hnew;
        k = k+1;
    end
    tElapsed(i) = toc(tStart);  % Stop Time counter
    counter(i) = count;
end
```

Listing 4: Newton.m

```
tElapsed = zeros(1,N); % Time taken for each outer loop
counter = zeros(1,N);% Inner Loop counter for each outer iteration
for i = 1:N
    tStart = tic;  % Start Time counter for i
    fprintf('Iteration Number = %i\n',i);
    x=x0;
    [f,g] = obj(x);% Evaluating values of function & gradient

    k = 0;
    alpha=[];
    K=abs(f);
    tol=sqrt(eps);
    df=1;
    ndx=1;

    j=eye(n);% Initialising inverse hessian j as Identity matrix
    count = 0;
    while norm(g)>tol && df>100*K*eps && ndx>tol
        count = count + 1;
        p = -j*g;% Descent Direction for BFGS
        [alpha]=ls_V2(k,x,p,alpha);% Line Search algorithm
        dx = alpha*p;
        ndx = norm(dx);
        x = x+dx;

        [fnew,gnew] = obj(x);
        df = abs(fnew-f);

        Y = (gnew -g);
        if(dx'*Y>0)
            % Calculating new j for BFGS
            gama = 1/(dx'*Y); % gama is a scalar
            A = eye(n) - gama*(dx*Y');
            jnew = A*j*A' + gama*(dx*dx');
            p = -jnew*gnew; % new Descent Direction
            if (p'*gnew>=0)
                fprintf('Not a descent direction, p''*g: %g\n', p'*gnew);
                fprintf('Finished processing this iteration\n');
                break
            end
        else
            fprintf('Condition dx''*Y>0 not satisfied dx''*Y: %g\n',dx'*Y);
        end
        f = fnew;
        g = gnew;
        j = jnew;
        k = k+1;
    end
    tElapsed(i) = toc(tStart);  % Stop Time Counter
    counter(i) = count;
end
```

Listing 5: BFGS.m

```
tElapsed = zeros(1,N); % Time taken for each outer loop
counter = zeros(1,N);% Inner Loop counter for each outer iteration
for i = 1:N
    tStart = tic;  % Start Time counter for i
    fprintf('Iteration Number = %i\n',i);
    x=x0;
    [f,g] = obj(x);% Evaluating values of function & gradient

    k = 0;
    alpha=[];
    K=abs(f);
    tol=sqrt(eps);
    df=1;
    ndx=1;

    h=eye(n);% Initialising hessian h as Identity matrix
    count = 0;
    while norm(g)>tol && df>100*K*eps && ndx>tol
        count = count + 1;
        p = -h\g;% Descent Direction for Inverse BFGS
        [alpha]=ls_V2(k,x,p,alpha);% Line Search algorithm
        dx = alpha*p;
        ndx = norm(dx);
        x = x+dx;

        [fnew,gnew] = obj(x);
        df = abs(fnew-f);

        Y = (gnew -g);
        if(dx'*Y>=0)
            % Calculating new h for Inverse BFGS
            hnew = h + (Y*Y')/(dx'*Y) - ((h*dx)*dx'*h)/(dx'*h*dx);
            p = -hnew\gnew;% New Descent Direction
            if (p'*gnew>=0)
                fprintf('Not a descent direction, p''*g: %g\n', p'*gnew);
                fprintf('Finished processing this iteration\n');
                break
            end
        else
            fprintf('Condition dx''*Y>0 not satisfied dx''*Y: %g\n',dx'*Y);
        end
        f = fnew;
        g = gnew;
        h = hnew;
        k = k+1;
    end
    tElapsed(i) = toc(tStart);  % Stop Time Counter
    counter(i) = count;
end
```

Listing 6: InverseBFGS.m

```
tElapsed = zeros(1,N); % Time taken for each outer loop
counter = zeros(1,N);% Inner Loop counter for each outer iteration
for i = 1:N
    tStart = tic;  % Start Time counter for i
    fprintf('Iteration Number = %i\n',i);
    x=x0;
    [f,g] = obj(x);% Evaluating values of function & gradient

    k = 0;
    alpha=[];
    K=abs(f);
    tol=sqrt(eps);
    df=1;
    ndx=1;

    h=eye(n);% Initialising hessian h as Identity matrix
    count = 0;
    while norm(g)>tol && df>100*K*eps && ndx>tol
        count = count + 1;
        p = -h\g;% Descent Direction for DFP
        [alpha]=ls_V2(k,x,p,alpha);% Line Search algorithm
        dx = alpha*p;
        ndx = norm(dx);
        x = x+dx;

        [fnew,gnew] = obj(x);
        df = abs(fnew-f);

        Y = (gnew -g);
        if(dx'*Y>=0)
            % Calculating new h for DFP
            gama = 1/(dx'*Y);
            % gama is a scalar
            A = eye(n) - gama*(dx*Y');
            hnew = A'*h*A + gama*(Y*Y');
            p = -hnew\gnew;% New Descent Direction
            if (p'*gnew>=0)
                fprintf('Not a descent direction, p''*g: %g\n', p'*gnew);
                fprintf('Finished processing this iteration\n');
                break
            end
        else
            fprintf('Condition dx''*Y>0 not satisfied dx''*Y: %g\n',dx'*Y);
        end
        f = fnew;
        g = gnew;
        h = hnew;
        k = k+1;
    end
    tElapsed(i) = toc(tStart);  % Stop Time Counter
    counter(i) = count;
end
```

Listing 7: DFP.m

```
tElapsed = zeros(1,N); % Time taken for each outer loop
counter = zeros(1,N);% Inner Loop counter for each outer iteration
for i = 1:N
    tStart = tic;  % Start Time counter for i
    fprintf('Iteration Number = %i\n',i);
    x=x0;
    [f,g] = obj(x);% Evaluating values of function & gradient

    k = 0;
    alpha=[];
    K=abs(f);
    tol=sqrt(eps);
    df=1;
    ndx=1;

    j=eye(n);% Initialising inverse hessian h as Identity matrix
    count = 0;
    while norm(g)>tol && df>100*K*eps && ndx>tol
        count = count + 1;
        p = -j*g;% Descent Direction for Inverse DFP
        [alpha]=ls_V2(k,x,p,alpha);% Line Search algorithm
        dx = alpha*p;
        ndx = norm(dx);
        x = x+dx;

        [fnew,gnew] = obj(x);
        df = abs(fnew-f);

        Y = (gnew -g);
        if(dx'*Y>=0)
            % Calculating new j for Inverse DFP
            jnew = j + (dx*dx')/(dx'*Y) - ((j*Y)*(Y'*j))/(Y'*j*Y);
            p = -jnew*gnew;% New Descent Direction
            if (p'*gnew>=0)
                fprintf('Not a descent direction, p''*g: %g\n', p'*gnew);
                fprintf('Finished processing this iteration\n');
                break
            end
        else
            fprintf('Condition dx''*Y>0 not satisfied dx''*Y: %g\n',dx'*Y);
        end
        f = fnew;
        g = gnew;
        j = jnew;
        k = k+1;
    end
    tElapsed(i) = toc(tStart);  % Stop Time Counter
    counter(i) = count;
end
```

Listing 8: InverseDFP.m