MS 4327 Project

Cathaoir Agnew 16171659

March 2020

1 Introduction

The task at hand for this project was to implement the following 6 algorithms in Matlab to solve for the Rosenbrock function:

- Steepest Decent Method, SDM
- Newton's Method
- Fletcher-Reeves Conjugate Gradient Method, FR CGM
- Polak-Ribiere Conjugate Gradient Method, PR CGM
- Dai-Yaun Conjuagte Gradient Method, DY CGM
- "Hybrid" Conjugate Gradient Method,

2 Algorithms

This section will consist of the algorithms I will implement in Matlab, along with a short explanation of each of the methods, with the exception of the Fletcher-Reeves Algorithm which was available from the lecture slides of the module.

2.1 Steepest Decent Method

This is the simplest of the gradient methods.The descent direction is the negative of the gradient, as the gradient points in the direction of the steepest assent. This algorithm is guaranteed to find the minimum point (if it exists) through numerous iterations. Although its convergence can be slow as it "zig zags" to the minimum point, the SDM has linear convergence.

Advantages	Disadvantages
Always "downhill"	Slower close to minimum
Avoids "saddle points"	Linear search may cause problems
Efficient further from the minimum	Might "zig zag" down valleys

Algorithm 1: ***Steepest Decent Method***

1. x_0 is the initial point
2. f function
3. $g = \nabla\ f(x_0)$
4. Search Direction $p = -g$
5. **while** $p^T g \neq 0$ **do**
6. α_k from line search, which satisfies Wolfe Conditions
7. $x_k + 1 = x_k + \alpha_k * p$
8. update g and p
9. **end while**

2.2 Newton's Method Algorithm

Newton's Method is usually faster than the Steepest Descent Method, as Newton's Method typical has a quadratic rate of convergence. Newton's algorithm is an unconstrained algorithm derived from a second order Taylor series expansion. However, Newton's Method requires the calculation of the Hessian Matrix, which can be computationally expensive.

Advantages	Disadvantages
Fast rate of convergence	Necessity of calculating the hessian
Typically quadratic convergence	Calculating the Hessian is computationally costly

Algorithm 2: ***Newton's Method***

1. x_0 is the initial point
2. f function
3. $g_0 = \nabla f(x_0)$
4. $H_0 = \nabla^2 f(x_0)$
5. step $= 1$
6. $i = 0$
7. **while** $step > 0$ **do**
8. Calc eigenvalues, λ, of H_i
9. minimum eigenvalue $= \min(\lambda)$
10. **if** minimum eigenvalue < 0 **do**
11. $p_i = (-(\frac{3}{2} * minimumeiegenvalue * I + H_i))^{-1} * g_i$
12. **else**
13. $p_i = -{H_i}^{-1} g_i$
14. **end if**
15. α_i from line search, which satisfies Wolfe Conditions
16. Update x_{i+1}, g_{i+1} and H_{i+1}
17. step $= p_{i+1}^T g_{i+1}$
18. **end while**

2.3 Fletcher-Reeves Algorithm

The Fletcher-Reeves algorithm is a conjugate gradient method for non linear problems. A scale factor, β^{FR}, is calculated by using the current and previous iteration. It generates descent directions and is globally convergent. However, there is a flaw in the Fletcher-Reeves approach, the Fletcher-Reeves method "sticks" if it ever generates a bad search direction p_i.

Advantages	Disadvantages
Globally convergent	Bad search direction causes method to "stick"

Algorithm 3: ***Fletcher-Reeves Method***

1. x_0 is the initial point
2. f function
3. $g_0 = \nabla\ f(x_0)$
4. $p_0 = -g_0$
5. $i = 0$
6. **while** $g_i \neq 0$ **do**
7. α_i from line search, which satisfies Wolfe Conditions
8. Update x_{i+1}, $x_{i+1} = x_i + \alpha_i\ p_i$
9. Update g_{i+1}, $g_{i+1} = \nabla f_{i+1}$
10. Calculate $\beta_{i+1}^{FR} = \frac{\left\|g_{i+1}\right\|^2}{\|g_i\|^2}$
11. Update p_{i+1}, $p_{i+1} = -g_{i+1} + \beta_{i+1}^{FR}\ p_i$
12. $i = i + 1$
13. **end while**

2.4 Polak-Ribiere Algorithm

The Polak-Ribiere method is the same as the Fletcher Reeves method with the exception of how β is calculated. The Polak-Ribere method is more robust than the Fletcher Reeves. The advantage the Polak-Ribere method has is an "automatic restart" property, which means the method will not "stick" if a poor choice of p_k is made.

Advantages	Disadvantages
Has restart property so it does not "stick"	Only generates descent directions and is globablly convergent when a two-stage version of the Strong Wolfe conditions is applied

Algorithm 4: ***Polak-Ribiere Method***

1. x_0 is the initial point
2. f function
3. $g = \nabla\ f(x_0)$
4. $p_0 = -g_0$
5. $i = 0$
6. **while** $g_i \neq 0$ **do**
7. α_i from line search, which satisfies Wolfe Conditions
8. Update x_{i+1}, $x_{i+1} = x_i + \alpha_i\ p_i$
9. Update g_{i+1}, $g_{i+1} = \nabla f_{i+1}$
10. Calculate $\beta_{i+1}^{PR} = \frac{g_{i+1}^T(g_{i+1}-g_i)}{\|g_i\|^2}$
11. Update p_{i+1}, $p_{i+1} = -g_{i+1} + \beta_{i+1}^{PR}\ p_i$
12. $i = i + 1$
13. **end while**

2.5 Dai-Yuan Algorithm

The Dai-Yaun method is also a variation of the Fletcher Reeves method, with the exception of how β is calculated. It is globally convergent. However, the flaw in the Fletcher-Reeves approach with a bad search direction causing the algorithm to "stick" is also present in the Dai-Yuan method. For a bad search direction p_i, the algorithm "sticks".

Advantages	Disadvantages
Globally convergent	Bad search direction causes method to "stick"

Algorithm 5: ***Dai-Yuan Method***

1. x_0 is the initial point
2. f function
3. $g = \nabla\ f(x_0)$
4. $p_0 = -g_0$
5. $i = 0$
6. **while** $g_i \neq 0$ **do**
7. α_i from line search, which satisfies Wolfe Conditions
8. Update x_{i+1}, $x_{i+1} = x_i + \alpha_i\ p_i$
9. Update g_{i+1}, $g_{i+1} = \nabla f_{i+1}$
10. Calculate $\beta_{i+1}^{DY} = \frac{\left\|g_{i+1}\right\|^2}{p_{i+1}^T(g_{i+1}-g_i)}$
11. Update p_{i+1}, $p_{i+1} = -g_{i+1} + \beta_{i+1}^{DY}\ p_i$
12. $i = i + 1$
13. **end while**

2.6 "Hybrid" Algorithm

Like the Dai-Yuan method, the "hybrid" algorithm is globally convergent. The Hybrib method does not "stick" like the Fletcher-Reeves and Dai-Yuan methods. The scale factor β is calculated by comparing the minimum of the Dai-Yuan β and the Hestenes and Stiefel β to zero, and then taking the maximum value of the above comparison.

Advantages	Disadvantages
Globally convergent and does not "stick"	Extra computations as two β's are calculated

Algorithm 6: ***Hybrid Method***

1. x_0 is the initial point
2. f function
3. $g = \nabla\ f(x_0)$
4. $p_0 = -g_0$
5. $i = 0$
6. **while** $g_i \neq 0$ **do**
7. α_i from line search, which satisfies Wolfe Conditions
8. Update x_{i+1}, $x_{i+1} = x_i + \alpha_i\ p_i$
9. Update g_{i+1}, $g_{i+1} = \nabla f_{i+1}$
10. Calculate $\beta_{i+1}^{DY} = \frac{\|g_{i+1}\|^2}{p_{i+1}^T(g_{i+1}-g_i)}$
11. Calculate Hestenes and Stiefel, $\beta_{i+1}^{HS} = \frac{g_{i+1}^T(g_{i+1}-g_i)}{p_{i+1}^T(g_{i+1}-g_i)}$
12. Hybrid $\beta_{i+1}^{H} = \max(0, \min(\beta_{i+1}^{DY}, \beta_{i+1}^{HS}))$
13. Update p_{i+1}, $p_{i+1} = -g_{i+1} + \beta_{i+1}^{H}\ p_i$
14. $i = i + 1$
15. **end while**

3 MatLab Code of Algorithms

3.1 Steepest Descent Method

```
function [x_final,counter_ca_total,t] = SDM(x,acc)
%Steepest Descent Method algorithm implemented on the
   Rosenbrock function
tic
%Gradient of f
[~,g]=obj(x);
%search direction
p=-g;
counter=0;

while max(abs((p.')*g)) > acc
    counter = counter+1;
   %Getting alpha
    alpha = ls_V2(0,x, p,[]);
   %Updating x
    x = x(:) + alpha*p;
   %Updating the values of g and p
     [~,g]=obj(x);
      p=-g;
end

counter_ca_total = counter;
x_final=x;
t=toc;
seconds = num2str(t);
fprintf( '\n \t Total Time Taken in Seconds: \t %s \n'
    , seconds)
end
```

3.1.1 SDM Explanation

There are 2 inputs for the function, x which is the initial guess vector and acc, which is the desired level of accuracy to be used. In the above code I tried to implement and follow the algorithm for the Steepest Descent Method. The Tic and Toc function is used to record the run time, and then converted into a string to be outputted in a clear and nice structure with the fprintf function. For the conditions of the while loop, I used $\max(\text{abs}(p^T g)) > \text{acc}$. This was to save both computational effort and time as running the function to true 0 could take a lot of time. In the Steepest Descent Method, the descent direction is the negative of the gradient, as the gradient points in the direction of the steepest assent. This is implemented in lines 5, gradient of the function is gotten and line 7, the search direction p is let equal to -gradient. These values are then checked in the while loop, and if the condition of $\max(\text{abs}(p^T g)) > \text{acc}$ is met, it enters the while loop. Here an α is calculated by the Is V2 function, provided on the project description in a link. The x value is updated with the new found α, line 15 , and the gradient is taken of this new updated value of x, with the search direction also being updated, lines 17 and 18. This is repeated until the while loop condition is not satisfied, once its not satisfied the method is complete, and the minimum value is found, within the accuracy level chosen acc. The toc function will then stop the timer, and the fprintf function prints out the run time.

3.2 Newton's Method

```
function [x_final,counter_ca_total,t] = Newton(x,acc)
%Newtowns Method algorithm implemented on the
   Rosenbrock function
tic
step = 1;
counter=0;

while abs(step) > acc
    %Get gradient of f and gradient squared of f
    [~,g,h]=obj(x);
    %Calculating eigenvalue of h
    eigen_h = eig(h);
    %Getting min eigenvalue
    min_eigen_h = min(eigen_h);

    if min_eigen_h < 0
        H = -(1.5*min_eigen_h*eye(size(h)) + h);
        p = inv(H) *g;
    else
        p = -inv(h)*g;
    end

    [alpha, ] = ls_V2(0, x, p, []);
    x = x(:) + alpha*p;

    %Updating g and h values with new x
    [ ~ ,gnew,hnew]=obj(x);
    eigen_hnew = eig(hnew);
    min_eigen_hnew = min(eigen_hnew);

    %Calculate p+1 and g+1
    if min_eigen_hnew < 0
        H = -(1.5*min_eigen_hnew*eye(size(h)) + hnew);
        p_new = inv(H)*gnew;
    else
        p_new = -inv(hnew)*gnew;
    end

    step = (p_new.')*gnew;
    counter = counter+1;
end

counter_ca_total = counter;
x_final=x;
t=toc;
seconds = num2str(t);
fprintf( '\n \t Total Time Taken in Seconds: \t %s \n'
    , seconds)
end
```

3.2.1 Newton's Explanation

There are 2 inputs for the function, x which is the initial guess vector and acc, which is the desired level of accuracy to be used. In the above code I tried to implement and follow the algorithm for Newton's Method. The Tic and Toc function is used to record the run time, and then converted into a string to be outputted in a clear and nice structure with the fprintf function. For Newton's method the initial step should be when possible be set equal to 1, which is done on line 4. For the conditions of the while loop, I used step $>$ acc. This was to save both computational effort and time as running the function to true 0 could take a lot of time. Newton's algorithm is an unconstrained algorithm derived from a second order Taylor series expansion. The first and second derivatives are calculated on line 9, using the obj function provided on the project description. Newton's Method requires the calculation of the Hessian Matrix, which is done in line 9, and is assigned the variable h. The eigenvalues of the Hessian matrix are calculated and the minimum eigenvalue is used in an if else statements to calculate the new search direction, seen in lines 11-20. Then a line search is undertaken to find α which satisfies the Wolfe Conditions calculated by the Is V2 function, provided on the project description in a link. The x value is updated with the new found α and search direction, line 23. The gradient is taken of this new value of x with the obj function. The updated hessian is then used to find the next search direction, and the step is updated. This process is repeated until the while loops conditions is not satisfied. Once the while loop conditions are not satisfied a minimum is found, within the accuracy level chosen by acc. The toc function will then stop the timer, and the fprintf function prints out the run time.

3.3 Fetcher-Reeves Algorithm

```
function [x_final,counter_ca_total,t] = FR(x,acc)
%Fletcher-Reeves algorithm implemented on the
    Rosenbrock function
tic
[~,g]=obj(x);
p=-g;
counter=0;

while max(abs(g)) > acc

    %Getting alpha
    [alpha, ] = ls_V2(0, x, p, []);
    %Updating x
    x = x + alpha*p;
    %Updating g
    [ ~ ,new_g] = obj(x);
    %Getting beta, breaking up to avoid any errors with
        brackets
    %beta_top_FR=(norm(new_g)).^2;
    %beta_bottom_FR = (norm(g)).^2;
    %beta_FR = beta_top_FR/beta_bottom_FR;
    beta_FR = ( (norm(new_g)).^2 / (norm(g)).^2 );

    %Updating p
    p = -new_g + beta_FR*p;
    g = new_g;
    counter = counter+1;
end

counter_ca_total=counter;
x_final=x;
t=toc;
seconds = num2str(t);
fprintf( '\n \t Total Time Taken in Seconds: \t %s \n'
    , seconds)
end
```

3.3.1 FR Explanation

There are 2 inputs for the function, x which is the initial guess vector and acc, which is the desired level of accuracy to be used. In the above code I tried to implement and follow the algorithm for the Fletcher Reeves Method. The Tic and Toc function is used to record the run time, and then converted into a string to be outputted in a clear and nice structure with the fprintf function. For the conditions of the while loop, I used $\max(\text{abs}(g_i)) > \text{acc}$. This was to save both computational effort and time. The initial search direction is gotten by taken the gradient of the function at the initial guess and taking the negative of this, seen on lines 4 and 5. The value of g_i is then checked in the while loop, and if the condition of $\max(\text{abs}(g_i)) > \text{acc}$ is met, it enters the while loop. Here a line search that satisfies the Wolfe Conditions is ran and the α is calculated by the Is V2 function, provided on the project description in a link. The x value is updated with the new found α, line 13 , and the gradient is taken of this new updated value of x. The gradient of the new value x is calculated, line 15. The FR β scale factor is calculated by using the current gradient and previous gradient, as seen in line 20. The search direction is then updated with the new gradient value and scaling factor, line 23. This is repeated until the while loop condition is not satisfied. Once the the while loop conditions are not satisfied, a minimum is found within the accuracy level chosen by acc. The toc function will then stop the timer, and the fprintf function prints out the run time.

3.4 Polak-Ribiere

```
1  function [x_final,counter_ca_total,t] = PR(x,acc)
2  %Polak-Ribiere algorithm implemented on the Rosenbrock
       function
3  tic
4  [~,g]=obj(x);
5  p=-g;
6  counter=0;
7
8  while max(abs(g)) > acc
9     %Getting alpha
10    [alpha, ] = ls_V2(0, x, p, []);
11    %Updating x
12    x = x + alpha*p;
13    %Updating g
14    [~,new_g] = obj(x);
15    %Getting beta, breaking into parts so not to have
         an error with the brackets
16    % Beta_top_PR=(new_g.')*(new_g-g);
17    % Beta_bottom_PR = ((norm(g).^2));
18    % Beta_PR = Beta_top_PR/Beta_bottom_PR;
19   Beta_PR = ((new_g.')*(new_g-g) / ((norm(g).^2)) );
20    %Updating p
21    p = -new_g + Beta_PR*p;
22    g = new_g;
23    counter = counter+1;
24 end
25
26 counter_ca_total=counter;
27 x_final=x;
28 t=toc;
29 seconds = num2str(t);
30 fprintf( '\n \t Total Time Taken in Seconds: \t %s \n'
       , seconds)
```

3.4.1 PR Explanation

As the Polak-Ribiere only differs from the Fletcher-Revees Algorithm by the scaling factor β. The coding is the same, except for line 19, where the β is calculated. The comments above where used to avoid any misplaced brackets.

3.5 Dai-Yuan

```
1  function [x_final,counter_ca_total,t] = DY(x,acc)
2  %Dai-Yuan algorithm implemented on the Rosenbrock
       function
3  tic
4  [~,g]=obj(x);
5  p=-g;
6  counter=0;
7
8  while max(abs(g)) > acc
9
10    %Getting alpha
11    [alpha, ] = ls_V2(0, x, p, []);
12    %Update x
13    x = x + alpha*p;
14    %Update g
15    [~,new_g] = obj(x);
16    %Getting beta, breaking into parts so not to have
          an error with the brackets
17    %beta_top_DY= norm(new_g).^2 ;
18    %beta_bottom_DY=(p.')*(new_g-g);
19    %beta_DY = beta_top_DY / beta_bottom_DY;
20    beta_DY = ( (norm(new_g).^2) / ((p.')*(new_g-g)) );
21    %Update p
22    p = -new_g + beta_DY*p;
23    g = new_g;
24
25    counter = counter+1;
26 end
27
28 counter_ca_total=counter;
29 x_final=x;
30 t=toc;
31 seconds = num2str(t);
32 fprintf( '\n \t Total Time Taken in Seconds: \t %s \n'
       , seconds)
33 end
```

3.5.1 DY Explanation

As the Dai-Yuan only differs from the Fletcher-Revees Algorithm by the scaling factor β. The coding is the same, except for line 20, where the β is calculated. The comments above where used to avoid any misplaced brackets.

3.6 Hybrid

```
1  function [x_final,counter_ca_total,t] = Hybrid(x,acc)
2  %Hybrid Method implemented for the Rosenbrock function
3  tic
4  [~,g]=obj(x);
5  p=-g;
6  counter=0;
7
8  while max(abs(g)) > acc
9     %Getting alpha
10    [alpha, ] = ls_V2(0, x, p, []);
11    %Updating x
12    x = x + alpha*p;
13    %Updating g
14    [~,new_g] = obj(x);
15
16    %Calculating beta DY
17    %beta_Top_DY= (norm(new_g).^2);
18    %beta_Bottom_DY = ((p.')*(new_g-g));
19    %beta_DY = beta_Top_DY/beta_Bottom_DY;
20
21    beta_DY = ( (norm(new_g).^2) / ((p.')*(new_g-g)) );
22
23    %Calculating hestenes and stiefel beta, beta HS
24    %beta_Top_HS = ((new_g.')*(new_g-g));
25    %beta_Bottom_HS = ((p.')*(new_g-g));
26    %beta_HS = beta_Top_HS/beta_Bottom_HS;
27
28    beta_HS = ( ((new_g.')*(new_g-g))/((p.')*(new_g-g))
         );
29
30    %Getting hybrid beta, beta HY
31    beta_HY = max(0,min(beta_DY,beta_HS));
32    %Update p
33    p = -new_g + beta_HY*p;
34    g = new_g;
35    counter = counter+1;
36 end
37
38 counter_ca_total=counter;
39 x_final=x;
40 t=toc;
41 seconds = num2str(t);
42 fprintf( '\n \t Total Time Taken in Seconds: \t %s \n'
       , seconds)
43 end
```

3.6.1 Hybrid Explanation

As the Hybrid algorithm only differs from the Fletcher-Revees Algorithm and the Dai-Yuan by the scaling factor β and the choice of which β to use. The coding is the same, except for lines 21-31, where the DY β_{i+1}^{DY} is calculated and the Hestenes and Stiefel β_{i+1}^{HS} is calculated. The Hybrid β_{i+1}^{H} is the maximum of the set (0, min(β_{i+1}^{DY}, β_{i+1}^{HS})). The comments above where used to avoid any misplaced brackets.

3.7 Run File for n=2

```
%This m file will run the 6 algorithms for n=2 on the
    rosenbrock problem
%recording the times taken to run each method
%inital guess will be 15,15 with accuracy level 1e-15

% n = 2
x_initial = [15,15];
acc = 1e-15;

[x_SDM,counter_SDM,time_SDM] = SDM(x_initial, acc);
[x_Newton, counter_Newton,time_Newton] = Newton(
    x_initial, acc);
[x_FR,counter_FR,time_FR] = FR(x_initial, acc);
[x_PR,counter_PR,time_PR] = PR(x_initial,acc);
[x_DY,counter_DY,time_DY] = DY(x_initial,acc);
[x_Hybrid,counter_Hybrid,time_Hybrid] = Hybrid(
    x_initial,acc);
```

3.7.1 Run file for n=2 Explanation

This file will call and run the 6 algorithms for the n=2 case on the Rosenbrock function. I used the initial guess of (15,15) for my run with an accuracy level of $1e^{-15}$. One sample of the run times for this is shown below in the table.

Method	Run Time (sec)
SDM	**0.058015**
Newton	**.00553**
FR	**.1297**
PR	**.006987**
DY	**.073715**
Hybrid	**.01063**

From this sample, Newton's Method and Polak-Ribiere appear to be the fastest methods, with the Fletcher-Reeves method taking the longest.

3.8 Run File for n=10

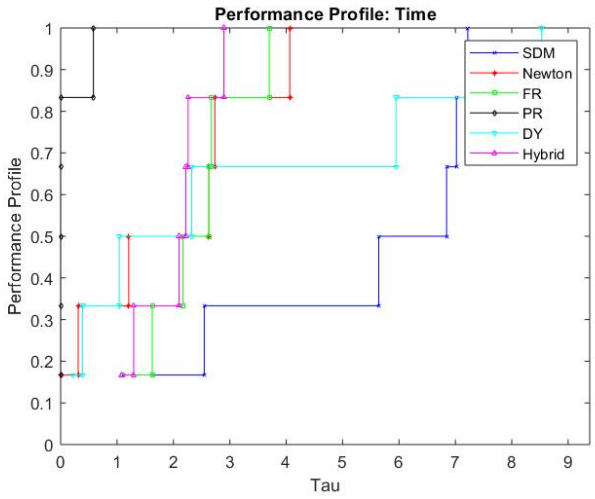
```
%This m file will run the 6 algorithms for n=10 on the
    rosenbrock problem.

%For loop creates a matrix so that each algorithm is
    assigned to a specific row
%Row 1: Steepest Decent Method
%Row 2: Newtons Method
%Row 3: Fletcher-Reeves Method
%Row 4: Polak-Ribiere Method
%Row 5: Dai-Yuan Method
%Row 6: Hybrid Method
%Each column of the matrix respresents the result of
    that run of the loop
%Col 1: first run
%Col 2: second run
%...
%Col 6: sixth run

```

```
%Using two matrices: 1)to store run times
%                    2)to store number of iteration

%Setting accuracy to 1e-5
acc = 1e-5;

%Creating mentioned above Matrices
run_times = zeros(6, 6);
no_iterations = zeros(6,6);

% These are the intial guesses
x_initial = [ -4 71 46 1.5  26.7 20  ] ;

for i = 1 :6
    %Outputting the run order, from 1 to 6
    fprintf( '\n <strong> Run %i </strong> \n' , i)

    %n=10, creating 10 copies of the initial guess
    x_initial_10= x_initial(i)*ones(1,10);

    %Steepest Decent Method
    [x_SDM_10,counter_SDM_10, time_SDM_10] = SDM(
       x_initial_10, acc);
    run_times(1, i ) = time_SDM_10;
    no_iterations(1, i) = counter_SDM_10;

    %Newton's Method
    [x_Newton_10,counter_Newton_10, time_Newton_10] =
       Newton(x_initial_10, acc);
    run_times(2, i) = time_Newton_10;
    no_iterations(2,i) = counter_Newton_10;

    %Fletcher-Reeves Method
    [x_FletR_10,counter_FletR_10, time_FletR_10] = FR(
       x_initial_10, acc);
    run_times(3, i) = time_FletR_10;
    no_iterations(3, i) = counter_FletR_10;

    %Polak-Ribiere Method
    [x_PR_10,counter_PR_10, time_PR_10] = PR(
       x_initial_10, acc);
    run_times(4, i) = time_PR_10;
    no_iterations(4,i) = counter_PR_10;

    %Dai-Yuan Method
    [x_DY_10,counter_DY_10, time_DY_10] = DY(
       x_initial_10, acc);
    run_times(5, i) = time_DY_10;
    no_iterations(5, i) = counter_DY_10;

    %Hybrid Method
    [x_Hyb_10,counter_Hyb_10, time_Hyb_10] = Hybrid(
       x_initial_10, acc);
    run_times(6, i) = time_Hyb_10;
    no_iterations(6, i) = counter_Hyb_10;
end
```

```
66
67 %Perf.m to plot Run Times
68 figure( 'Name' , 'Performance Profile: Time' )
69 perf(run_times,'Time')
70 legend( 'SDM' , 'Newton' , 'FR', 'PR' , 'DY' , 'Hybrid
       ')
71 title( 'Performance Profile: Time' )
72 xlabel( 'Tau' );
73 ylabel( 'Performance Profile' );
74
75 %Perf.m to plot the number of iterations
76 figure( 'Name' , 'Performance Profile:Iterations' )
77 perf(no_iterations,'Iterations' )
78 legend( 'SDM' , 'Newton' , 'FR', 'PR' , 'DY' , 'Hybrid
       ')
79 title( 'Performance Profile: Number of Iterations' )
80 xlabel( 'Tau' );
81 ylabel( 'Performance Profile' );
```

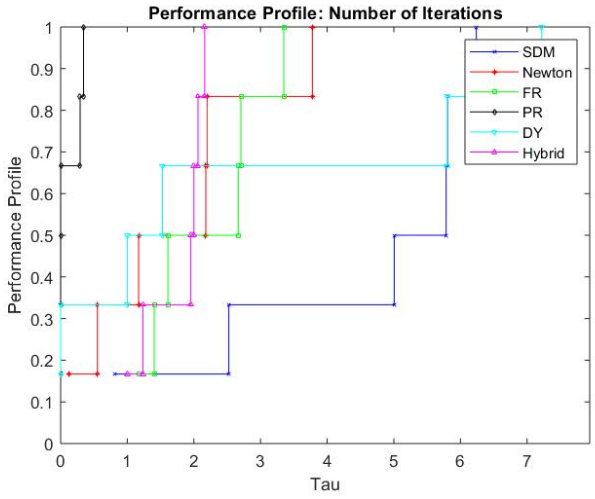
3.8.1 Run file for n=10 Explanation

I reduced the accuracy to $1e^{-5}$ for the n=10 case as this is a quite a large problem, for my laptop at least. I used 6 initial guesses, $-4, 71, 46, 1.5, 26.7, 20$. I then created matrices to store the number of iterations and the time taken by each method, seen on lines 23-24. Each row of the matrices represented a method. Then, the for loop runs through each of the 6 guesses and stores the run times and number of iterations taken to solve for the Rosenbrock function, lines 29-65. Finally, I used the perf.m file provided on the project description to produce the below graphs.



The plot above shows the 6 methods and ranks them on the time taken to solve the Rosenbrock function. It is quite clear the Polak-Ribiere method is the fastest, with the Steepest Descent Method and Dai-Yuan Method being the slowest of the 6 methods. In general the Steepest Decent Method would be considered the slowest method, but we can see the DY Method was the slowest. This might be explained by a poor choice of search direction which can cause the method to "stick", which is a disadvantage of the DY method.

The plot below shows the 6 methods and ranks them on the number of iterations taken to solve the Rosenbrock function. It is quite clear the Polak-Ribiere method takes the least number of iterations, with the Steepest Descent Method and Dai-Yuan Method taking the most iterations of the 6 methods. In general the Steepest Decent Method would be considered the highest number of iterations method, but we can see the DY Method had the highest number of iterations. This might be explained by a poor choice of search direction which can cause the method to "stick", which is a disadvantage of the DY method.



3.8.2 Conclusion

Overall, the Polak-Ribiere is the most efficient method to use when considering both run time and number of iterations taken to solve for the Rosenbrock function.