Name: Max Chamberlin, Candidate Number: MEC68

Question 1: Hybrid Bisection Interpolation Algorithm

The bracket program is implemented in simple_bracket(), listed in the appendix:

Given an initial input, we search for a bracket along the direction of descent from the given input. We can think of the direction of descent as being like our positive x-axis to get our bearings. The input point then has a **negative** gradient along this descent direction. My algorithm searches for another point with a **positive** gradient along the direction of decrease. Once such a point has been found, we must have a bracket for the minimum. This is a consequence of the simple mathematical fact that for a continuous function, there must be a point x st: f'(x)=0 that is a minimum between a 'left' point with a negative gradient and the 'right' point with a positive gradient[1]

The search program is implemented in search(), listed in the appendix.

I used two different methods of interpolation in the search phase. The first method I implemented was a simple bisection. We update the bracket [L,R] for the minimum by interpolating midway between the bounds of the bracket. Then we select either [L,I] or [I,R] to form the new bracket. A minimum must be contained between a bracket where the leftmost point has a negative gradient and the rightmost point has a positive gradient, as previously explained; we select [L,I] or [I,R] based on this criterion.

However, I later adapted this method to produce a hybrid algorithm, which would give better convergence. If the function we hope to minimise is roughly parabolic near the minimum, then moving to the abscissa of a parabolic approximation should take us much closer to the true minimum - since smooth functions are well approximated by parabolas close to their minima. The parabola was fitting using gradient information. This typically resulted in faster convergence, but had the disadvantage that occasionally degenerate parabolas were created (ones were the interpolation point was at the same location as one of the previous bracket bounds). To remedy this problem, I opted to use a hybrid method that averaged between the midpoint and the parabolic interpolation point.

Question 2: Testing the Algorithm

Below I plot the performance of the two algorithms described. I decided to initialise the points at various places of interest: for instance at a minima 0, for very large values of x, where the function oscillates much less and also for very small values of x. In the graphs, the bracket and initialisation point are given in the title. What can be seen is that both algorithms converge to the minimum. The hybrid algorithm I developed is faster than the midpoint algorithm, as can be seen from the relative steepness of the curves in the diagram below. Kinks in the curves are due to the fact that there are often several minima in a given bracket, and so the gradient of the interpolation point may not decrease as smoothly as we would expect. As bisection reduces the interval length by half on each iteration, the bisection method has precisely linear convergence. It remains to determine whether the hybrid algorithm has linear or super-linear convergence.

[1]***NB**: if the point is at an optima, my algorithm displaces the point slightly from the optima so that a direction of descent can be found, and a minimum can subsequently be bracketed.

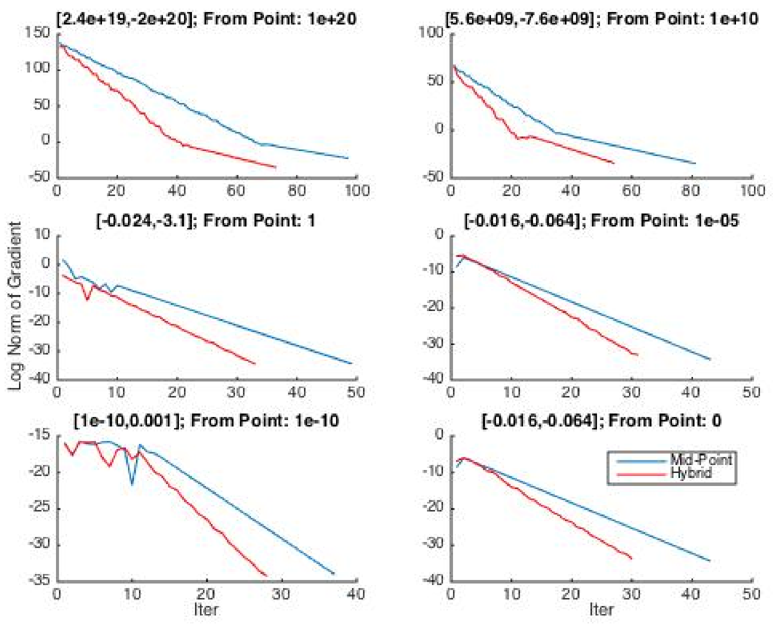


Figure 1: Convergence for Mid-Point and Hybrid Algorithm: Log Norm Gradient

I produced the plot below, which charts the distances between successive interpolation points. Since, the bisection method reduces the interval length by exactly half on each iteration, we should expect the completely linear rate of decrease as shown . The hybrid methods in some case looks like it may have reached a super-linear rate of convergence: for instance in the the graph from point 1e-10. However, most of the curves show seem to display a typical linear rate of decrease.

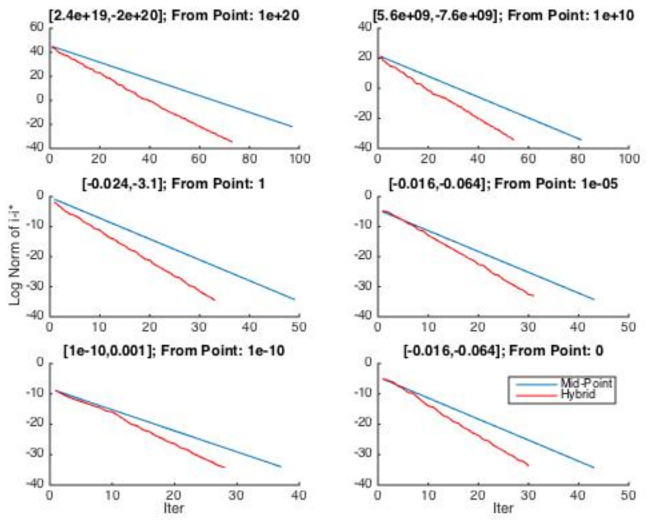
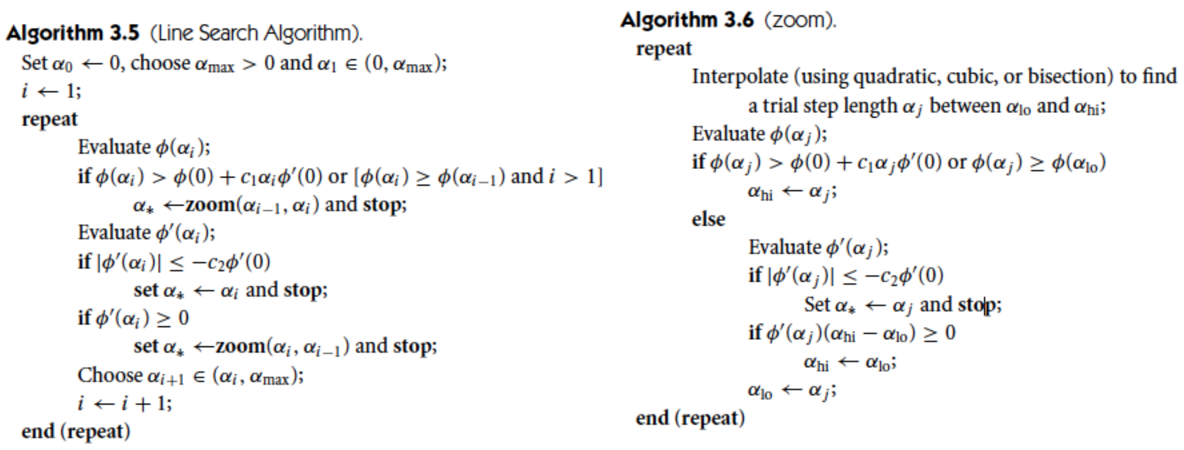


Figure 2: Convergence for Mid-Point and Hybrid Algorithm: Distances between Interpolation Points

Another interesting point to note, but that I decided not to produce graphs for, is that: since this function is completely symmetric about the 0-point, (and thus gradients and directions of descent are symmetric) my algorithm would produce exactly the same performance curves for negative initialisation points as for the corresponding positive points.

Question 3

Since my line-search algorithm in question 1 only used gradient information to bracket and search for the minimum, a great many structural changes had to take place for the function to use the Wolfe conditions as a criterion for stopping. Ultimately, I implemented the strong Wolfe conditions from the outline as described in Nocedal in Numerical Optimisation chapter 3, under: 'A Line Search Algorithm For the Wolfe Conditions' .This is given in the pseudo-code below. Zoom searches for the minimum, and LineSearch does the bracketing.



```
Algorithm 3.5 (Line Search Algorithm).
  Set $\alpha_0 \leftarrow 0$, choose $\alpha_{\max} > 0$ and $\alpha_1 \in (0, \alpha_{\max})$;
  $i \leftarrow 1$;
  repeat
      Evaluate $\phi(\alpha_i)$;
      if $\phi(\alpha_i) > \phi(0) + c_1\alpha_i\phi'(0)$ or $[\phi(\alpha_i) \geq \phi(\alpha_{i-1})$ and $i > 1]$
          $\alpha_* \leftarrow$ zoom$(\alpha_{i-1}, \alpha_i)$ and stop;
      Evaluate $\phi'(\alpha_i)$;
      if $|\phi'(\alpha_i)| \leq -c_2\phi'(0)$
          set $\alpha_* \leftarrow \alpha_i$ and stop;
      if $\phi'(\alpha_i) \geq 0$
          set $\alpha_* \leftarrow$ zoom$(\alpha_i, \alpha_{i-1})$ and stop;
      Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{\max})$;
      $i \leftarrow i + 1$;
  end (repeat)
```

```
Algorithm 3.6 (zoom).
  repeat
      Interpolate (using quadratic, cubic, or bisection) to find
          a trial step length $\alpha_j$ between $\alpha_{lo}$ and $\alpha_{hi}$;
      Evaluate $\phi(\alpha_j)$;
      if $\phi(\alpha_j) > \phi(0) + c_1\alpha_j\phi'(0)$ or $\phi(\alpha_j) \geq \phi(\alpha_{lo})$
          $\alpha_{hi} \leftarrow \alpha_j$;
      else
          Evaluate $\phi'(\alpha_j)$;
          if $|\phi'(\alpha_j)| \leq -c_2\phi'(0)$
              Set $\alpha_* \leftarrow \alpha_j$ and stop;
          if $\phi'(\alpha_j)(\alpha_{hi} - \alpha_{lo}) \geq 0$
              $\alpha_{hi} \leftarrow \alpha_{lo}$;
          $\alpha_{lo} \leftarrow \alpha_j$;
  end (repeat)
```

Figure 3: Strong Wolfe Line Search: Nocedal, Chapter 3

As a result of structural changes to the program, I decided to change the condition for bracketing from using derivative information to conditions on step lengths. According to Nocedal, we should bracket to ensure the step lengths bracket: (α_{i-1}, α_i) contains step lengths satisfying the strong Wolfe conditions. This occurs if one of the following three conditions is satisfied: (i) α_i violates the sufficient decrease condition; (ii) $(\alpha_i\ )\ \ (\alpha_{i-1}\ )$; (iii) $(\alpha_i\ )\ \ 0$ - where $\phi(\alpha) = f(x + dir \cdot \alpha)$; $\phi'(\alpha) = f'(x + dir \cdot \alpha)$. These changes are also reflected in my code, for when one of those conditions is found to hold we undertake the search that leads us to a minimum.

To implement the algorithm above neatly, I decided that all the information relating to a step-size should be held in one place. As such, I created a 'step size' class that stored all the information related to different step sizes. This included: the locations stepped to; the vector x they were stepping away from; the descent directions, their gradient and function values at the current location; and their gradients along the descent direction. I also gave each step object a counter to count their function evaluations.

Interpolation: because the algorithm was previously constructed on the basis of derivative information, which is no longer possible in this setting, it would not be possible to implement the simple parabolic interpolation as presented in question 1. The bracket no longer contains a negative and positive gradient for parabolic interpolation. Consequently, I decided to use the simple bisection method that was initially discussed and **tested** for in question 2. However, I have also included a modified hybrid bisection algorithm which interpolates based on different principles from those expounded in question 1.

Question 4:

The code to perform steepest descent and conjugate gradients is listed within the appendix. My programs perform an inexact line search using the line search method that had been developed in question 3. In this question we use bisection to interpolate.

In the diagrams below, I plot the trajectories of the respective functions for the point initialised at (x,y)=[-1,0.5].

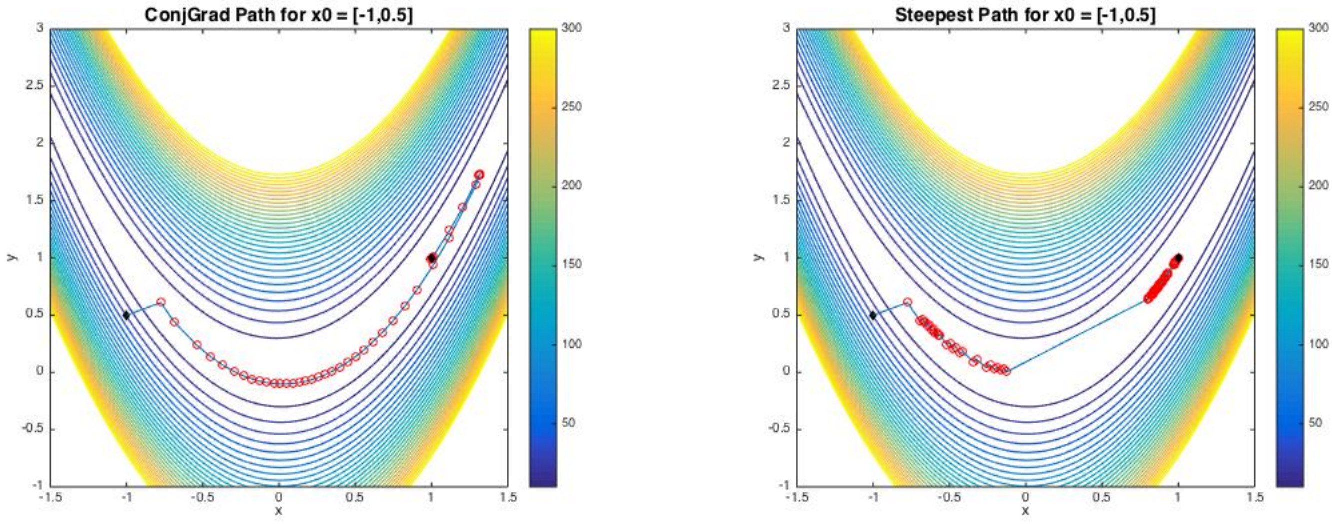


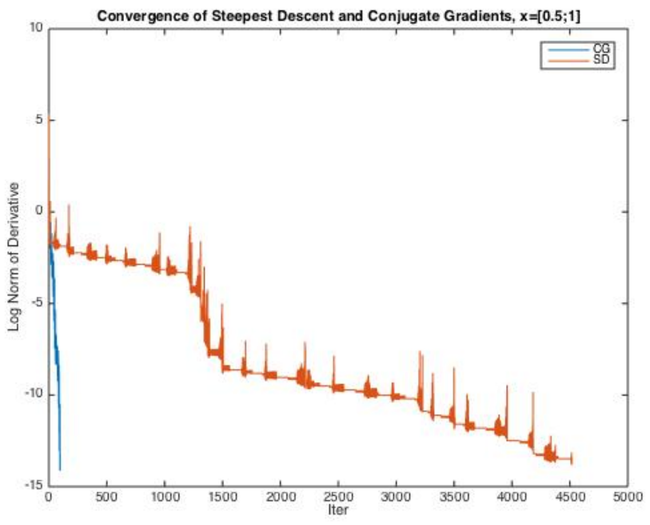
Figure 4: Paths over Rosenbrock Valley for SD and CG Algorithms

From the of the respective paths above, we can see that the conjugate gradients takes a smoother route along the valley floor, following a path that makes larger jumps between successive sampled points. The steepest descent method tends to spend a lot of time moving perpendicular to the contour lines of the figure, making very little progress towards the true minimum as it follows the paths of steepest descent. Because the directions of conjugate gradients are somewhat orthogonal, there is much less wasted movement as it progresses towards the global minimum.

Below, I have also tabulated the mean function calls and main-loop iterations taken for the respective algorithms to converge to the minimum, with initial points sampled from 0 to 2 randomly in both x and y. I averaged over 30 values. The criterion for convergence is that the norm of the gradient must be less that 1e-6. As we can see, on average Steepest Descent takes 1000 times more function calls and 100 times as many main loop iterations as conjugate gradients to converge to the minimum. But why should SD perform worse when scrutinised by function call than by iteration? This is probably due to the fact that it's very difficult to find a small enough step-size with a sufficient decrease along a descent direction when we approach the minimum, since descent directions are almost parallel to one another. This makes progress increasingly difficult. In contrast, CG, which looks for orthogonal directions, is not blighted by the same problem and can easily find large step sizes between successive points .

Tolerance =1e-6	Conjugate Gradients	Gradient Descent
Mean f Calls	970.3000	129,230
Std Deviation f Calls	910.8231	50,989
Mean Iters	42.4000	4,562.1
Std Iters	36.5547	1,819.4

I have also graphed the results of running the algorithm on a test point, so that convergence can be seen visually. As can be seen, conjugate gradients converges much more rapidly than steepest descent. Both algorithms tend to converge to the minimum when initialised fairly close to the global minimum. However, whilst Conjugate gradients seems to converge even from afar (perhaps due to the robust search directions), the steepest descent algorithm seems to cycle aimlessly through a set of points, taking smaller and smaller steps towards a minimum.



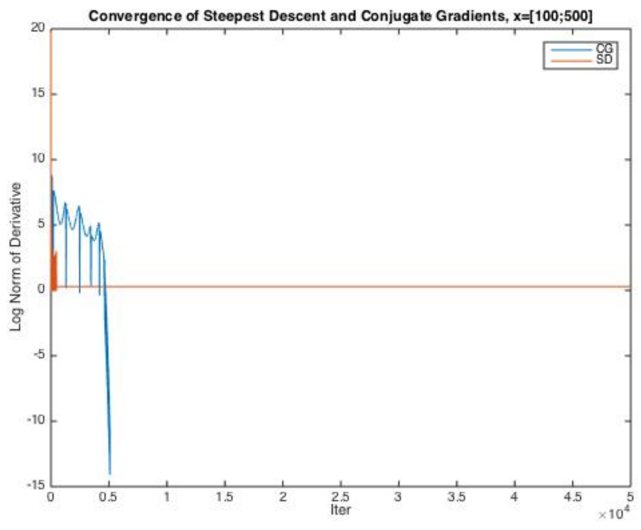


Figure 5: Graph of Convergence for Conjugate Gradients and Steepest Descent

Note:

The Rosenbrock function can be trivially solved by applying a change of variables: $z_1 = (1 - x),\ z_2 = 10 * (y - x^2)$. In which case $f(x, y) = g(z_1, z_2) = \mathbf{z^T I z}$. This latter function can be solved with an exact line search, which would be much less computationally expensive. Because the shape of the quadratic cost is uniform, steepest descent would solve the problem in 2 iterations. However, I take it that the question was asking for a robust method to measure the performance of the algorithms, rather than exploiting other mathematical manipulations.

Question 5:

The programs to perform an exact line-search Steepest Descent and Conjugate Gradients are listed in the appendix.

In general the B matrices are better suited to gradient descent than the A matrices: the contours of their objective function are more 'spherical' than they are like 'ellipsoids'. This can be seen from the tabulation of their eigen-values below. The largest eigen-value in B is only slightly greater (by a factor of 10) than the smallest eigen -value. In the A matrices, the largest eigen-value is more than 10,000 times as great as the smallest. This will indeed produce a very narrow elliptical objective function. As a result, we should expect that steepest descent would perform much more poorly than conjugate gradients on the B matrices, as it gets stuck in the narrow valleys (see figure below).

matrix/eigen_values	λ_1	λ_2	λ_3	λ_4	λ_5	λ_6	λ_7	λ_8	λ_9	λ_{10}
matrix A10	1.0825	1.0460	0.1082	0.1046	0.0105	0.0108	0.0011	0.0010	0.0001	0.0001
matrix B10	1.0266	1.0267	10.4256	10.4256	10.2667	10.2667	1.0267	1.0426	1.0426	1.0267

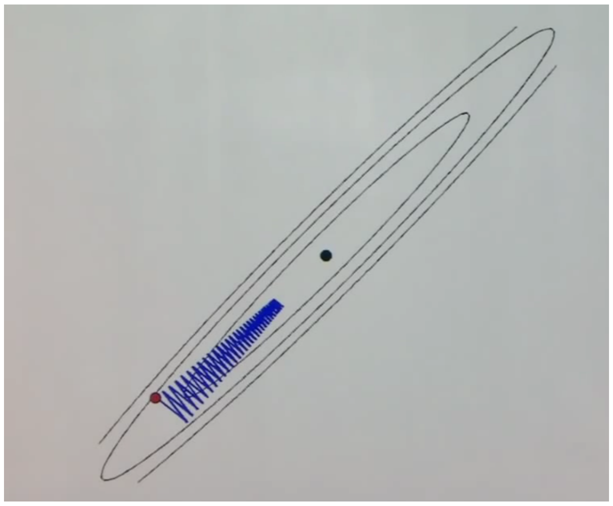


Figure 6: An example of Steepest Descent stuck in a Narrow Ellipse, From: tumblr_naubqg3kYK1qc38e9o1_1280.png

Below I have plotted graphs showing the performance of both SD and CG from or an initial point x, whose elements are identically set to 1, and with the b vector in the $b^T x$ term of the objective function randomised as suggested. As can be seen, for the more elliptical A matrices, CG tends to reach the minimum thousands of times faster. SD does converge, however the rate is rather slow- with the number of iterations until convergence thousands of times larger than the dimensions of A10. For the more spherical matrices, CG still outperforms steepest descent, however the performance of the two methods is much closer.

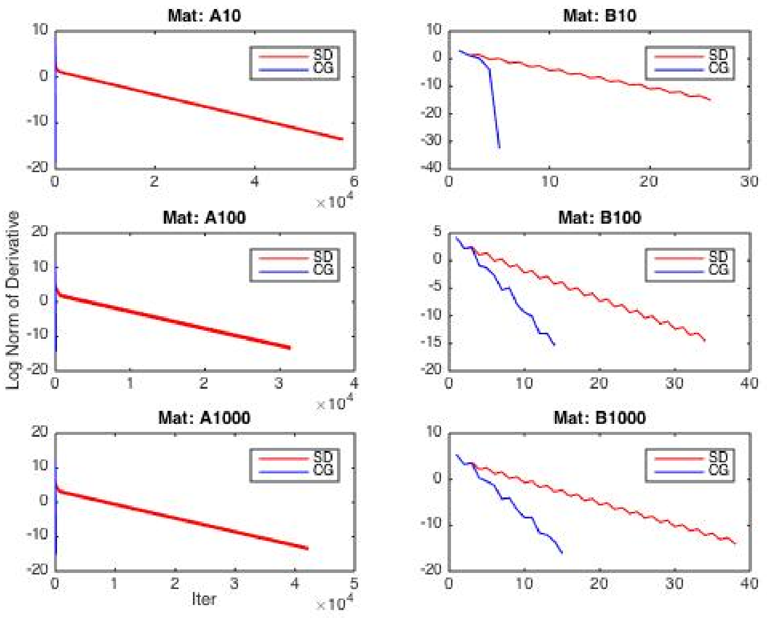


Figure 7: Convergence of the Conjugate Gradients and Steepest Descent over A,B matrices

To produce a more general set of results, I also tabulated an average number of function calls until convergence- specified by the tolerance 1e-4. Another interesting point to note is that the standard deviation of the number of function calls for conjugate gradients is very low, often 0. From a theoretical perspective, conjugate gradients should find the exact minimum after a number of evaluations roughly equal to A's or B's dimension (Similar to how it takes less than N steps to find the minimum of an N dimensional matrix by moving along the direction of it's eigenvectors). However, to reach the tolerance threshold, it takes a little more time for 2 A matrices and much less time for the B matrices. From the table below, we can see that SD performs only marginally worse (by a factor of 2 or 3 times the number of f calls) for fairly spherical matrices, but many thousands of times worse for elliptical matrices.

Tolerance =1e-4	A10	B10	A100	B100	A1000	B1000
CJG Mean f Calls	15.8	5	94.53	14	136	15
GJG Std Deviation f Calls	0.43	0	2.74	0	0	0
SD Mean f Calls	10,930	22.6	15,809	34	17,018	38
SD Std Deviation f Calls	1,343	3.32	272.74	0	91.28	0

Question 6:

For this question, I used the same algorithms used in question 4, but supplied with arguments for the objective and gradient functions of the A and B matrices. The graph below shows the performance of the two algorithms from or an initial point x, whose elements are identically set to 1 and with b in the $b^T x$ term randomised.

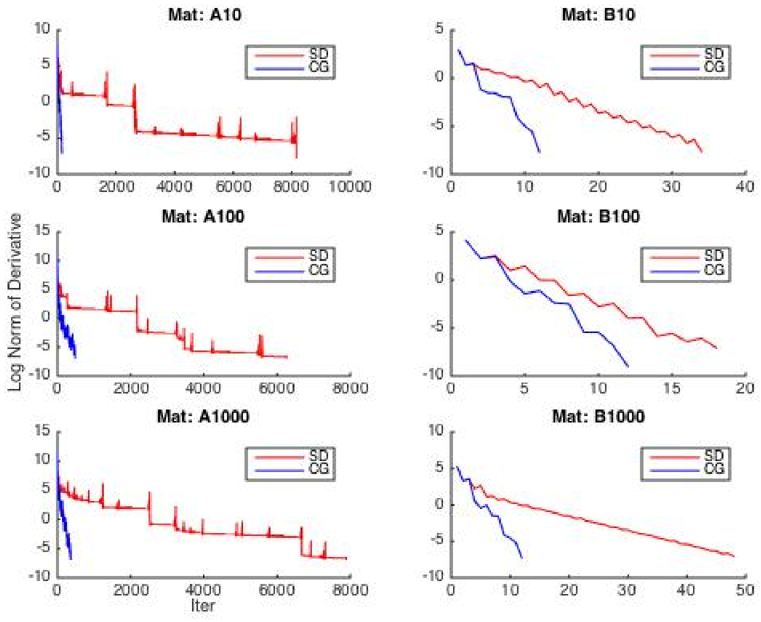


Figure 8: Convergence of the Conjugate Gradients and Steepest Descent over A,B matrices

The graphs show that rate of convergence is generally much less smooth for the A matrices, which is perhaps to be expected since the method uses an inexact line search. This is because when the contours of the objective function are plotted it can be shown to have a very narrow elliptical shape, and as such even small movements along a direction of descent can result in much larger changes in gradient. The graphs of convergence for the B matrices are much smoother because the ellipses are not generally moving into a 'valley' of the 'ellipse', but more in away which is parallel to the contours. This results in a much smaller change in gradient.

I tabulated the average number of iterations and function calls until convergence below. Generally, conjugate gradients performed comparably as well as before for the more spherical objective functions as defined by the B matrices. However, Conjugate gradients took much longer (10 times as long) to converge with the more elliptical A matrices. With an exact line search, we had a theoretical guarantee that convergence would take place in less than n iterations (where n is the dim of A or B), with an inexact line search there is no such guarantee and convergence takes considerably longer.

With Steepest Descent, there seemed to be a reduction in the number of calls to the line search function, by a factor of around 2/3. This is to be expected since an inexact line search should takes us further away from a narrow valley within an ellipse, than an exact line search. This means that we can move more closely to the contours of the ellipse as conjugate gradients does, rather than getting stuck iterating between the contours of a narrow ellipse. However, if we look at the number of function calls in their totality (rather than comparing calls to line search within the main loop) the number of function evaluations has actually increased. This can be seen in the second table below.

Main Loop Iters:

Tolerance =1e-4	A10	B10	A100	B100	A1000	B1000
CJG Mean iters	187.7333	16.10	258.2667	17.0667	288	19.6667
GJG Std Deviation iters	83.6095	3.0212	26.8057	1.7991	87.7765	1.2411
SD Mean f iters	3,376	17.2000	2174	14.5000	2528	39.6000
SD Std Deviation f iters	327	6.8097	198	4.9532	237	9.6369

Function Calls:

Tolerance =1e-3	A10	B10	A100	B100	A1000	B1000
CJG Mean f Calls	3,833	146.9	5.2236e+03	162.4333	5.5659e+03	185.8667
GJG Std Deviation f Calls	1,993	54.73	540.0611	29.1366	2.0026e+03	18.5300
SD Mean f Calls	17,330	214.8667	39,659	163.4333	43,975	591.6000
SD Std Deviation f Calls	4,565	93.9269	3,872	62.7764	5,667	150.8817

Appendix:

Q1: simple_bracket()

```
function [left,right,g_eval]=simple_bracket(gradF,x)
% INPUT
% gradF - gradient of function
% x     - initial point
% OUTPUT
% left   - left bracket
% right  - right bracket
% g_eval - gradient evaluations


% Preliminaries: Find a descent direction
%-----------------------------------------------------------------
g   = gradF(x);
dir = -sign(g);
% Code below handles rare 'bad inputs', eg saddle points.
% i1 counts the number of function calls
i1=1;
while(dir==0 && i1<10)
    % The displacement increases geometrically to escape saddle points.
    x=x+(1e10^i1)*eps;
    g  =    gradF(x);
    dir=-sign(g);
    i1=i1+1;
end
% Did we escape, do we have a descent direction?
if (dir==0)
    error('Could not escape optima');
end
%-----------------------------------------------------------------




% Main Algorithm:
%-----------------------------------------------------------------
alpha              = 0.1;     % Initial step length
fac                = 4;       % Increase factor
alpha_prev         =0;        % Prior step length
```

```
i2                    =0;          % Counts gradient calls of main loop

        % Main Loop:
        %------------------------------------------------------------------
        % Terminate on 1000th function call
        while(i2 <1000)
            % Take a step in the descent direction
            g_alpha=gradF(x+alpha*dir);
            i2=i2+1; g_eval=i2+i1;
            % Do we have a bracket for the minimum?
            if g_alpha*dir > 0
                 right=x+alpha*dir;
                 left=x+alpha_prev*dir;
                 return;
            end;
            %Increase step sizes
            alpha_prev=alpha;
            alpha = fac*alpha;
        end
        error('error: Search exceeds 1000 iterations');
%--------------------------------------------------------------------------
```

Q1: search algorithm()

- Both bisection and hybrid methods performed

```
function [a,interp,b,numEvals,gradDiff, Diff] = search(gradF,a,b,type)
%[a,x,b,h,i]=simple_bracket(@f,@gradF,1e19,@bisect)
% INPUT
% gradF - gradient of function to be minimised
% a     - left bracket
% b     - right bracket
% type  - 'hybrid' for hybrid interpolation, otherwise mid point used

% OUTPUT
% a       - updated left bracket
% b       - updated right bracket
% interp  - interpolation point* estimate for minimum
% numEvals- number of gradient evaluations
% gradDiff- log norm of successive gradients - for plotting
% Diff    - absolute difference between interps - for plotting


%Preliminaries
%----------------------------------------------------------------------
% tolerance for minimum acceptance
tol=eps*1e1;

% gradient of end points
gradA=gradF(a);
gradB=gradF(b);
% interpolation point
interp=(a+b)/2;
% number of function evaluations so far
numEvals=2;
% Store absolute difference in gradient to plot
gradDiff=nan(500); gradI=nan;
Diff=nan(500);


% Main Loop
%----------------------------------------------------------------------
% Bisect until convergence  within tolerence or
while (abs(a-b)>tol && numEvals <1000)
    prev=interp;
    %1. Interpolate between the bracket
    %--------------------------------------------------
    % take mid point as default interpolation
    interp=(a+b)/2;

    % update using hybrid or mid point criterion
    if strcmp(type,'hybrid')
        % construct a parabola y=px^2+qx+d through a and b, from deriv info
        p=0.5*(gradA-gradB)/(a-b);
        q=0.5*(gradA+gradB-2*p*(a+b));
        % estimation of min point of parabola
        min=-0.5*q/p;
```

```matlab
        % hybrid estimation offsets averages parabolic with mid point
        interp=((a+b)/2 +min)/2;
    end


    %2. Store the absolute difference in gradients to plot
    %-------------------------------------------------
     gradIPrev=gradI;
     gradI=gradF(interp);
     gradDiff(numEvals)=abs(gradIPrev-gradI);
     Diff(numEvals)=abs(prev-interp);

    %3. Update the bracket of the minimum
    %--------------------------------------------------
    if (gradI==0)
        return
    % Does the interpolation point have the same gradient as A
    elseif (sign(gradA)==sign(gradI))
        a=interp; gradA=gradI;
    % If not it must have the same sign as B
    else
         b=interp; gradB=gradI;
    end
    numEvals=numEvals+1;
```

Question 3: Line Search

Step object

```matlab
classdef step  < handle
    properties
        size; %step size
        x;  % step from where
        f =[]; % function eval
        df =[];% derivative information
        dir; % direction normed
        count =0;
        loc; %location after step
        dfdir=[]; % gradient in line direction
    end

    methods
        % initialise the step size values
        function init(obj,size,x,dir)
            obj.size=size;
            obj.x=x;
            obj.dir=dir/norm(dir);
            obj.loc=obj.x+obj.dir*obj.size;
            obj.f=[];
            obj.df=[];
        end
        % evaluate the function at the step's location
         function evalF(obj,f)
          if (isempty(obj.f))
            obj.f=f(obj.loc);
            obj.count=obj.count+1;
          end
         end
        % initialise the gradient
         function evalDF(obj,df)
          if (isempty(obj.df))
            obj.df=df(obj.loc);
            obj.count=obj.count+1;
            obj.dfdir=dot(obj.df,obj.dir);
          end
         end
         % copy values from another step, but keep function evaluation counter the same
          function obj = copy(obj,obj2)
            obj.size=obj2.size;
            obj.x=obj2.x;
            obj.dir=obj2.dir;
            obj.loc=obj2.loc;
            obj.df=obj2.df;
            obj.f=obj2.f;
            obj.dfdir=obj2.dfdir;
         end
    end

end
```

Line Search() with Zoom() nested

```

 function [opt,z] = LineSearch(f,gradF,x,dir, type)

% implementation of wolfe line search
%
% f        - function handle of objective function
% gradF    - function handle of gradient
% x        - current iterate
% dir      - search direction must be a descent
% a        - the a vars are step objects
% type     - type of interpolation

% Output
% opt      - min point
% z        - number of function evaluations


a0 =step; ai=step; a_pre=step;

% c1= suff dec, c2= curvature condition NB: c1 <= c2
%c1 = 0.01; c2 = 0.1;
c1 = 0.01; c2 = 0.1;

a0.init(0,x,dir); a0.evalF(f); a0.evalDF(gradF);
ai.init(1,x,dir); a_pre=a_pre.copy(a0);

iter = 1;
while (iter<1000) %cap on iterations is 1000
    %'main'
    ai.evalF(f);
    % check if current iterate violates sufficient decrease
    if  ai.f > a0.f + c1*ai.size*a0.dfdir || (ai.f >= a_pre.f && iter > 1)
        %  Acceptable point between a_pre and a_i because (c1 > c2)
        [opt,z] = nocZoom(a_pre, ai);
        z=z+a0.count+ai.count+a_pre.count;
        return;
    end

    ai.evalDF(gradF);
    % current iterate has sufficient decrease, but are we too close?
    %if(ai.dfdir >=c2*a0.dfdir)
     if abs(ai.dfdir) <=-(c2*a0.dfdir)
        % Wolfe fullfilled, quit
        opt= ai.loc;
        z=a0.count+ai.count+a_pre.count;
        return;
    end
    % are we ahead of the minimum?
    if (ai.dfdir >= 0)
        % there has to be an acceptable point between a_pre and ai
        [opt,z] = nocZoom(ai,a_pre);
        z=z+a0.count+ai.count+a_pre.count;
        return;
    end
    a_pre=a_pre.copy(ai);
    %Double ?i step size ;
    ai.init(ai.size*2,x,dir);
    iter = iter + 1;
end



function [m,z] = nocZoom(aLo,aHi)
% Interpolated step size

aj=step;
aHi.evalDF(gradF);
aLo.evalDF(gradF);

  while 1
      'zoom';

  %update mid point with parabolic interpolation if different criterion used
    if strcmp(type,'hybrid')
        % construct parabola
        aLo.evalF(f);aHi.evalF(f);
        num = aLo.dfdir(aHi.size-aLo.size);
        denom = 2*(aLo.f+aLo.dfdir*(aHi.size-aLo.size)-aHi.f);
        % estimation of min point of parabola
        min= aLo.size+num/denom;
        % hybrid estimation offsets averages parabolic with mid point
        size=((aHi.size+aLo.size)/2 +min)/2;
    else
        size=(aHi.size+aLo.size)/2;
    end

     aj.init(size,x,dir);
    %aj.evalDF(gradF);
    aj.evalF(f);

    if aj.f > a0.f + c1*aj.size*a0.dfdir || aj.f >= aLo.f
       % No sufficient decrease for a => a= maximum of the interval
       aHi = aHi.copy(aj);
    else
       aj.evalDF(gradF);
       % strong wolfe fullfilled?
       %if aj.dfdir >=c2*a0.dfdir
       if abs(aj.dfdir) <=-(c2*a0.dfdir)

           m=aj.loc; z=aj.count;
           return;
       end
       % if slope positive and aHi > aLo
       if aj.dfdir*(aHi.size-aLo.size) >= 0
           aHi = aHi.copy(aLo);
           %aLo = aLo.copy(aj) ;
       end
         aLo=aLo.copy(aj);
    end

  end

end

end
```

Question 4

Steepest descent()

```
function [x,logAbsG,evals,k]=steepest(x, gradF, f)
% Input:
% x      -  initial guess of the solution
% gradF -  gradient of function
% f      -  function to minimise

% Output:
% x            -  minimum of f
% logAbsG      -  the log of the norm of the gradient over iters
% evals        -  number of function evals
% k            - nuumber of iterations

%Output:

% tolerance for stopping algorithm
tol = 1e-4;
% iteration number

k = 1;
% maximum number of iterations
MAX=5000;

% g- gradient at x; x next location; p- conjugate gradient
g = gradF(x);

% num evals is 0
evals=0;
% log  norm of the gradient for plotting
logAbsG=nan(MAX,1);
logAbsG(k,1)=log(norm(g));

while norm(g) >tol & k<MAX
    % get new point guess from minimum and num evals from Line Search
    % search in direction of steeopest descent
    [x,evals1] =  LineSearch(f,gradF,x,-g);
    % update number of evaluations
    evals=evals+evals1;
    % get new gradient
    g=gradF(x);
    % update number of evaluations
    evals=evals+1;
    % update iter
    k=k+1;
    % store gradient info for plotting
    logAbsG(k,1)=log(norm(g));
end
end
```

Conjugate Gradients()

```
function [xk,logAbsG,evals,k]= conjGrad(x1, gradF, f)
% minimise along conjugate vectors: pk which are conjugate to A
% At each step we chose ak by an inexact line searc
% Input:
% x      -  initial guess of the solution
% gradF -  gradient of function
% f      -  function to minimise

% Output:
% x            -  minimum of f
% logAbsG      -  the log of the norm of the gradient over iters
% evals        -  number of function evals
% k            - nuumber of iterations

%Output:



% tolerance for stopping algorithm
tol = 1e-4;
% iteration number

k = 1;
% maximum number of iterations
MAX=5000;

% g- gradient at x; x next location; p- conjugate gradient
gk = gradF(x);

% search direction is -grad
pk=-gradF(x);

% num evals is 0
evals=0;

% log  norm of the gradient for plotting
logAbsG=nan(MAX,1);
logAbsG(k,1)=log(norm(g));

while norm(gk) >tol & k<max

            % Check conjugate gradient is a descent direction
            if gradF(xk)'*pk>0
                pk=-pk;
            end;
            % Perform line search
            [xk_next,evals1] = LineSearch(f,gradF,xk,pk, 'hybrid');
            % Update number of ealuations
            evals=evals+evals1;

            % This section updates the conjugate gradients
            % according to the method used in Barber, BRML
            gk_next=gradF(xk_next);
            evals=evals+1;
            bk =(gk_next'*gk_next)/(gk'*gk);
            pk_next=-gk_next+bk*pk;
            k=k+1; pk=pk_next; gk=gk_next; xk=xk_next;

            % we also apply the heuristic that the gradients are reset
            % every 100 iterations
            if floor(k/100)==1
                pk = -gk;
                bk=0;
            end;

            % update logAbs G for plotting
            logAbsG(k,1)=log(norm(gk));
end
end
```

Question 5 and 6

Steepest descent()

```
function [x,k,logAbsG]=steepestMat(x,A,seed)
% Input:
% x      -  initial guess of the solution
% A      -  A objective function
% note b is created with a random seed

% Output:
% x            -  minimum of f
% logAbsG      -  the log of the norm of the gradient over iters
% k            - nuumber of iterations


% tolerance for stopping algorithm
tol = 1e-6;

% Create Objective vector
rng(seed)
N=length(A);
b=2*rand(N,1)-1;

% g- gradient at x
g = A*x-b;
% negative of gradient
p = -g;

% number of iterations
k=1;

% Maximum number of iterations
MAX=10000;
% log Abs gradient for plotting
logAbsG=nan(MAX,1);
logAbsG(k,1)=log(norm(g));

% whle not converged do:
while norm(p) >tol & k<MAX

        % exact step size
        a= g'*g/(p'*A*p);
        % update x
        x=x+a*p;
        % update the search directions and conj grad
        g=A*x-b; p=-g;
        k=k+1;
        % store logabsG for plotting
        logAbsG(k,1)=log(norm(g));
end

end
```

Conjugate gradients

```
function [x,k,logAbsG]=conjGradMat(x,A,seed)
% Input:
% x      -  initial guess of the solution
% A      -  A objective function
% note b is created with a random seed

% Output:
% x            -  minimum of f
% logAbsG      -  the log of the norm of the gradient over iters
% k            - nuumber of iterations


% tolerance for stopping algorithm
tol = 1e-6;

% Create Objective vector
rng(seed)
N=length(A);
b=2*rand(N,1)-1;

% g- gradient at x
g = A*x-b;
% conjugate gradient
p = -g;

% number of iterations
k=1;

% Maximum number of iterations
MAX=10000;
% log Abs gradient for plotting
logAbsG=nan(MAX,1);
logAbsG(k,1)=log(norm(g));

% whle not converged do:
while norm(p) >tol & k<MAX

        % The method is given by Barber, BRML
        % Exact Line Search
        a= g'*g/(p'*A*p);
        % Update x
        x_next=x+a*p;
        % Update grad, conj
        g_next=A*x_next-b;
        b =(g_next'*g_next)/(g'*g);
        p_next=-g_next+b*p;
        k=k+1; p=p_next; g=g_next; x=x_next;

        % atore log nor grad for plotting
        logAbsG(k,1)=log(norm(g));
end

end
```