Convex Optimization

Exercise 2 (110/500)

Report Delivery Date: 17 April 2018

Instructor: Athanasios P. Liavas

Student: Konidaris Vissarion 2011030123

A. (a) In the first part of the exercise, we shall consider a simple quadratic optimization problem. Let $f : \mathbb{R}^n \to \mathbb{R}$. For fixed $x \in \mathbb{R}$, let $g : \mathbb{R} \to \mathbb{R}$ be defined as

$$g_x(y) := f(x) + \nabla f(x)^T(y - x) + \frac{m}{2}||y - x||_2^2.$$

i. The gradient of $g(x)$ is

$$\nabla g_x(y) = \frac{\partial}{\partial y}(\nabla f(x)^T(y - x)) + \frac{m}{2}\frac{\partial}{\partial y}(||y - x||_2^2) =$$

$$\nabla f(x) + \frac{m}{2}\frac{\partial}{\partial y}(y^T y - 2y^T x + x^T x) = \nabla f(x) + m(y - x).$$

ii. The value y_* that minimizes the function g is

$$y_* = \arg\min_y g_x(y)$$

$$\nabla g_x(y_*) = 0 \Leftrightarrow f(x) + m(y_* - x) = 0 \Leftrightarrow my_* = mx - \nabla f(x)$$

$$\Leftrightarrow y_* = x - \frac{1}{m}\nabla f(x).$$

So the minimum value of the function g is

$$g_x(y_*) = f(x) + \nabla f(x)^T(x - \frac{1}{m}\nabla f(x) - x) + \frac{m}{2}||x - \frac{1}{m}\nabla f(x) - x||_2^2 =$$

$$= f(x) - \frac{1}{m}||\nabla f(x)||_2^2 + \frac{1}{2m}||\nabla f(x)||_2^2 = f(x) - \frac{1}{2m}||\nabla f(x)||_2^2.$$

The relation (6.33) in section 6.5 , $p_* \geq f(x) - \frac{1}{2m}||\nabla f(x)||_2^2$, is a consequence of the convergence analysis of strongly convex functions. Using the relation (6.31) which is a quadratic global underestimator of f and equal to our function $g_x(y)$ we can prove the relation (6.33) which is similar to our result $g_x(y_*)$. Thus we have $f(y) \geq g_x(y_*)$.

B. In the second part of the exercise we shall solve a convex quafratic problem using a descent method with exact and back-tracking line search. Consider the problem

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) = \frac{1}{2}x^T P x + q^T x$$

where $P \in \mathbb{R}^{n \times n}$, $P = P^T \succ \mathbf{0}$ and $q \in \mathbb{R}^n$.

(a) We begin by constructing a random positive definite matrix P. In order to fully control the condition number of the matrix we do the following steps.

i. First we generate a random $(n \times n)$ matrix A and calculate its singular value decomposition. This is done for the purpose of generating a random orthonormal matrix U. By computing UU^T and U^TU we confirm that this is the case as both of them are equal to the identity matrix I_n.

ii. Then we construct the eigenvalues λ_i, for $i = 1, ..., n$. We select the λ_{min} and λ_{max} in a way that enables us to get the desired condition number for our problem. Afterwards, we generate the rest $n - 2$ eigenvalues uniformly at random from within the interval $[\lambda_{min}, \lambda_{max}]$. Lastly, we compute P as $U\Lambda U^T$, where Λ is the diagonal matrix with our randomly generated eigenvalues.

(b) The closed form solution of the problem is the following.

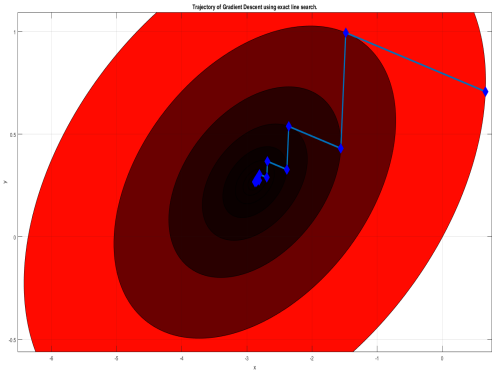
$$\nabla f(x_*) = 0 \Leftrightarrow Px_* + q = 0 \Leftrightarrow x_* = -P^{-1}q$$

We proceed by minimizing f for various dimensions ($n = 2$, 50, 500, 1000) and for various condition numbers of P ($K = 10$, 100, 1000), using the gradient algorithm with exact and back-tracking line search. We use the quantity $log(f(x_k) - p_*)$, where p_* the closed form solution, to compare the two algorithms. In all cases, the parameters α and β of the back-tracking line search algorithm where chosen to be 0.4999 and 0.5 respectively. The reason for this is that the algorithm with these values tends to perform better for this problem.

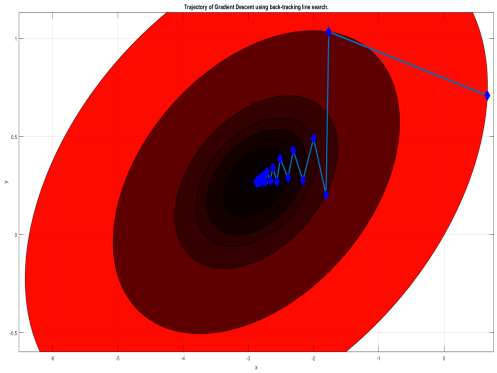
In Figure 1 we can see the trajectories of the descent algorithm for the exact and back-tracking line search in the $n = 2$ case. We can see that in this low dimensional case, the exact line search converges faster to the optimal solution, while the back-tracking line search does much more extra steps, especially as in the $K = 100$ and $K = 1000$ cases. The are of course some random cases where $K = 10$, where the two algorithms perform similarly, converging after a few number of iterations.

On the other hand, looking at Figure 2, we can see that for problems of larger dimensions ($n = 50$, 500, 1000), back-tracking line search tends to perform better that the exact line search.

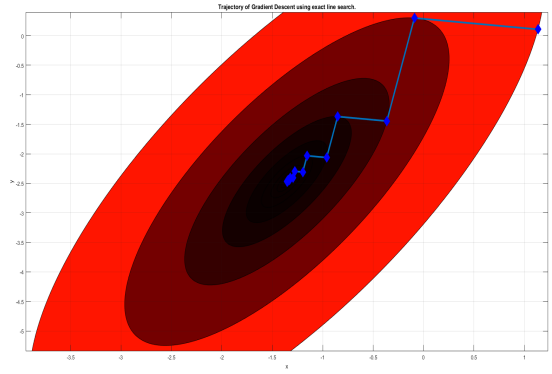
Finally, on Figure 3 we see the theoretical versus the actual minimum number of iterations that guarantees solution within accuracy ϵ. In all cases, meaning for all algorithms, in all problem dimensions and for all the condition numbers, the actual number of iterations before the algorithms converge, are far less that the theoretical ones. The theoretical minimum number of iterations come from the convergence analysis results of strongly convex functions and are $k_e \approx Klog\frac{f(x_0)-p_*}{\epsilon}$ for exact line search and $k_e \approx= \frac{log\frac{f(x_0)-p_*}{\epsilon}}{log(\frac{1}{c})}$, where $c = 1 - min(2m\alpha, \frac{2\beta\alpha m}{M})$, $m = \lambda_{min}$, $M = \lambda_{max}$, for the back-tracking line search.



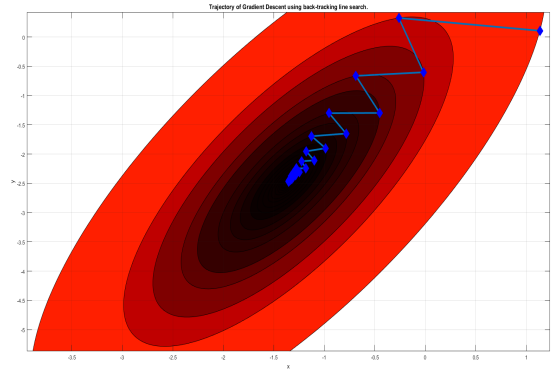
(a) $n = 2$, $K = 10$ Exact line search.



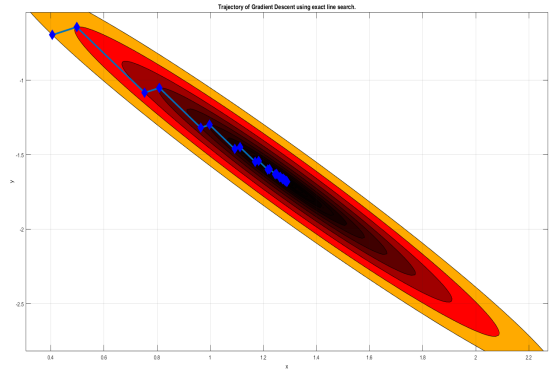
(b) $n = 2$, $K = 10$ Back-tracking line search.



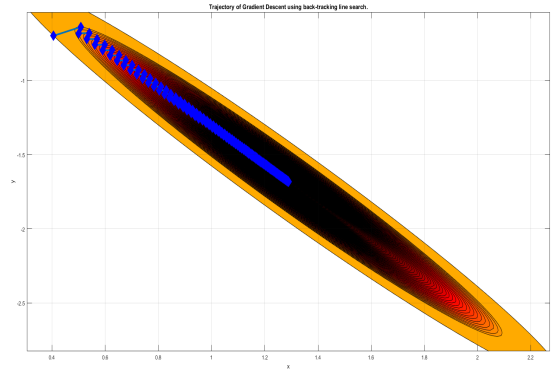
(c) $n = 2$, $K = 10$ Exact line search.



(d) $n = 2$, $K = 10$ Back-tracking line search.

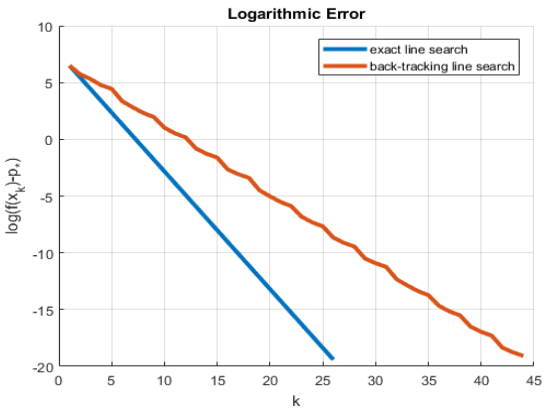


(e) $n = 2$, $K = 100$ Exact line search.

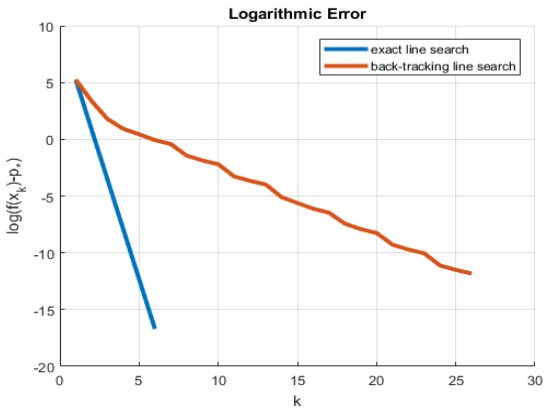


(f) $n = 2$, $K = 100$ Back-tracking line search.

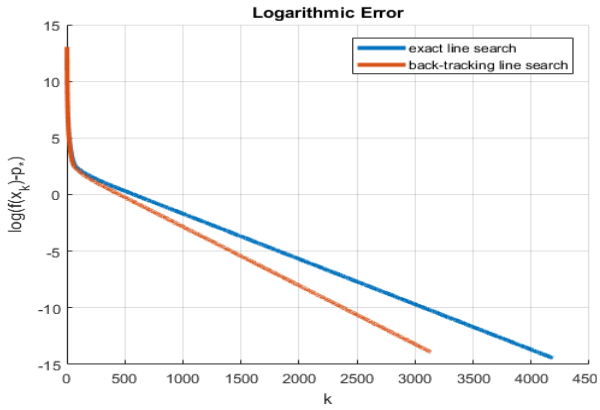
Figure 1: Trajectories of $\{x_k\}$ produced by the two algorithms for $n = 2$.



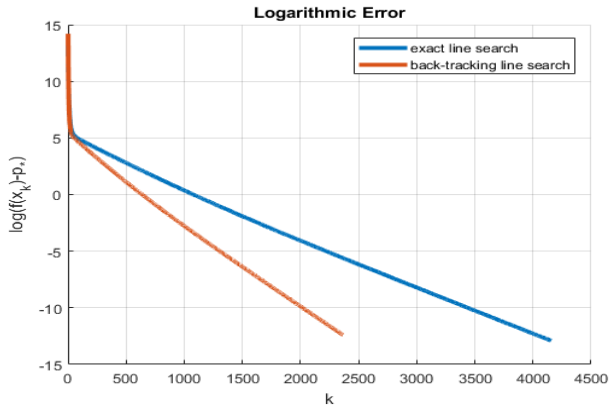
(a) $n = 2$, $K = 10$



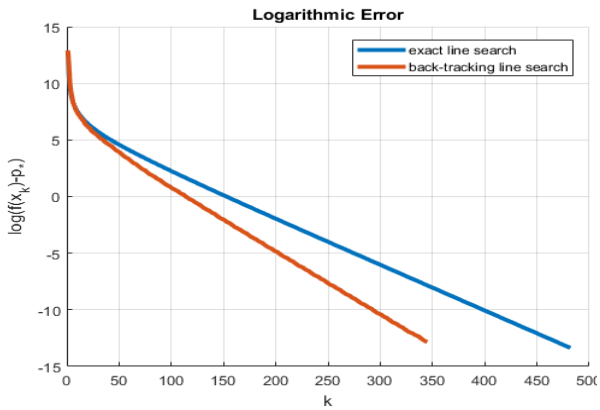
(b) $n = 2$, $K = 1000$ Back-tracking line search.



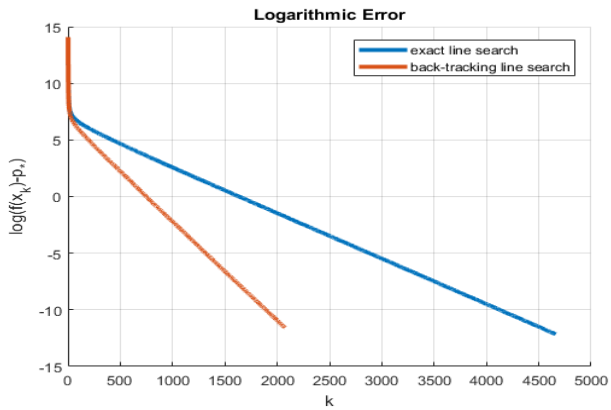
(c) $n = 50$, $K = 1000$ Exact line search.



(d) $n = 500$, $K = 1000$ Back-tracking line search.

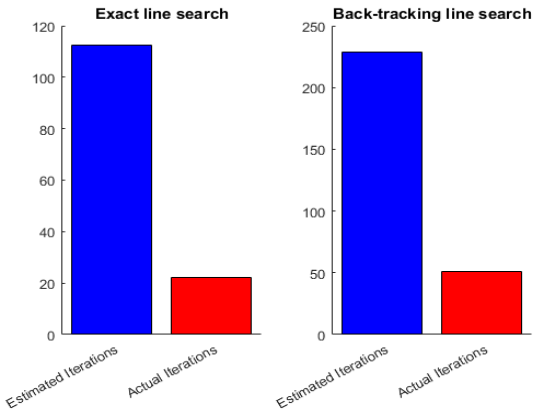


(e) $n = 1000$, $K = 100$ Back-tracking line search.

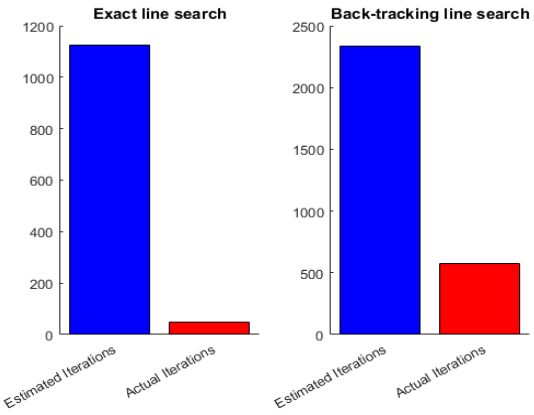


(f) $n = 1000$, $K = 1000$ Exact line search.

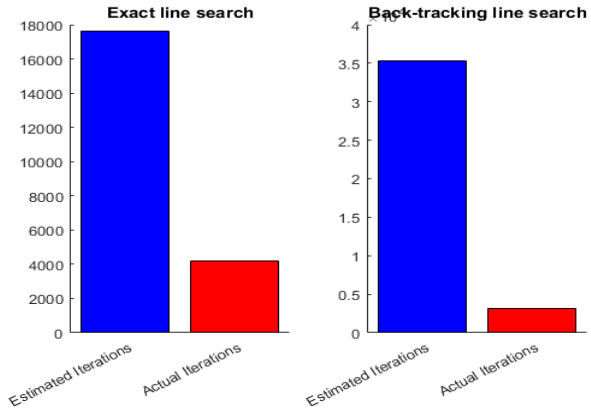
Figure 2: $log(f(x_k) - p_*)$, produced ny the two algorithms, versus k.



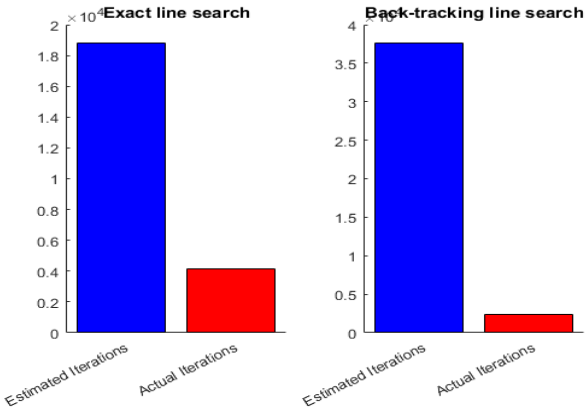
(a) $n = 2$, $K = 10$



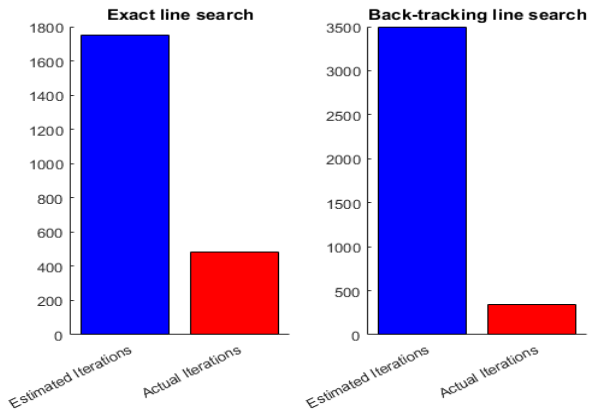
(b) $n = 2$, $K = 100$ Back-tracking line search.



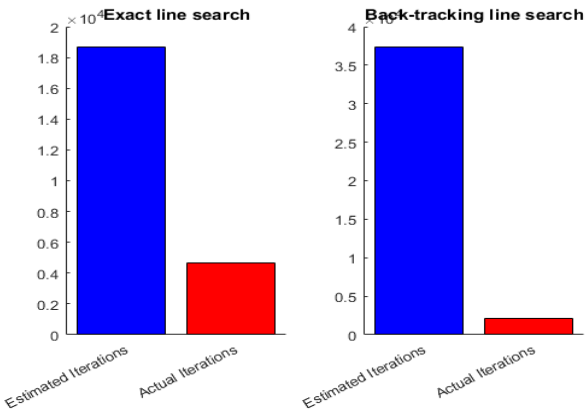
(c) $n = 50$, $K = 1000$ Exact line search.



(d) $n = 500$, $K = 1000$ Back-tracking line search.



(e) $n = 1000$, $K = 100$ Back-tracking line search.



(f) $n = 1000$, $K = 1000$ Exact line search.

Figure 3: Theoretical vs actual iterations before converging within accuracy ϵ.

C. In the third part of the exercise, we will solve general unconstrained problems using first order and second order descend methods. We consider the function

$$f(x) = c^T x - \sum_{i=1}^{m} log(b_i - a_i^T x) = c^T x - sum(log(b - Ax))$$

with $x \in \mathbb{R}^n$ and $m > n$ (or $m \gg n$).

(a) The domain of f is the following.

$$domf = \{x \in \mathbb{R}^n \mid a_i^T x < b_i,\ i = 1, ..., m\}$$

Each if the i arguments of the logarithms must be positive. The above domain is the intersection of the i open halfspaces $a_i^T x - b_i < 0$ which are convex sets. Hence the $domf$ is itself convex.

(b) f is differentiable in its domain. We will show that the hessian of this function is positive semidefinite.

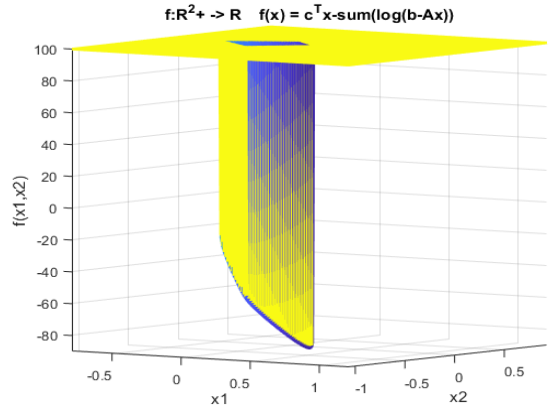
$$\nabla f(x) = c - \sum_{i=1}^{m} \frac{d}{dx}(log(b_i - a_i^T x)) = c + \sum_{i=1}^{n}(b_i - a_i^T x)^{-1} a_i$$

$$\nabla^2 f(x) = \sum_{i=1}^{m} a_i \frac{d}{dx}((b_i - a_i^T x)^{-1}) = -\sum_{i=1}^{m} a_i (b_i - a_i^T x)^{-2} \frac{d}{dx}(b_i - a_i^T x)$$

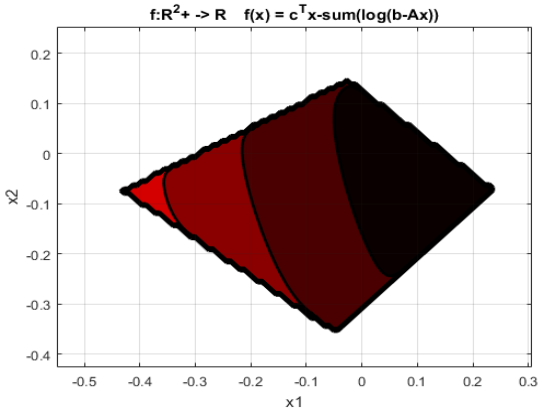
$$\Leftrightarrow \nabla^2 f(x) = \sum_{i=1}^{m}(b_i - a_i^T x)^{-2} a_i a_i^T$$

The Hessian is positive semidefinite as $a_i a_i^T$ is positive semidefinite and $(b_i - a_i^T x)^{-2} > 0$ inside the $domf$. Hence f is convex.

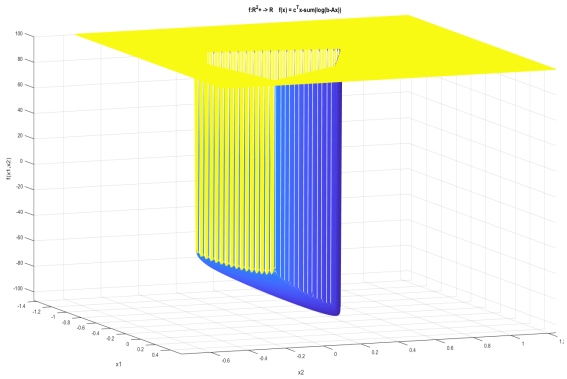
We minimize this function w.r.t x using the cvx tool, the gradient algorithm using back-tracking line search and with the Newton algorithm using also back-tracking line search for (n,m)=(2,20),(50,200),(300,800). On Figure 4 we can see the plot and level sets of f for $n = 2$. We compare the two algorithms by plotting the semilogy of $(f_{gradient}(x_k) - p_*)$ and $(f_{newton}(x_k) - p_*)$ as a function of step k. As we can see from Figure 5, the Newtons method is far superior to the gradient algorithm, especially in high dimensional problems. In low dimensional problems, the two algorithms have smaller differences. Newtons method descends along f at each step much more that the gradient algorithm does, with the cost of computing the Hessian matrix.



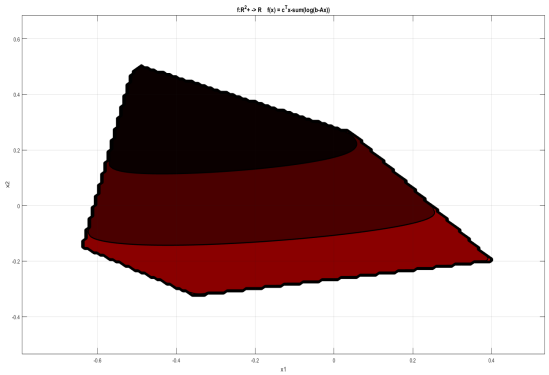
(a) Plot of f for $(n, m) = (2, 20)$



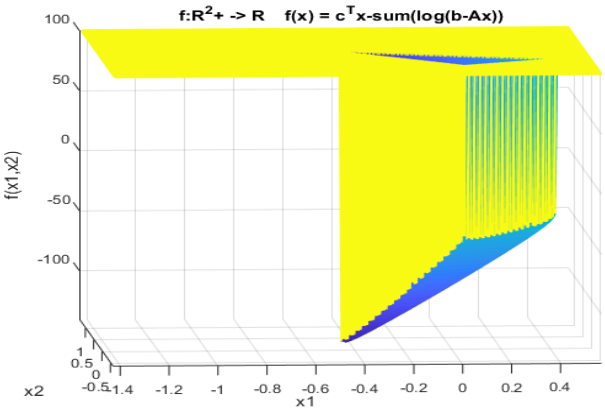
(b) Level sets of f for $(n, m) = (2, 20)$



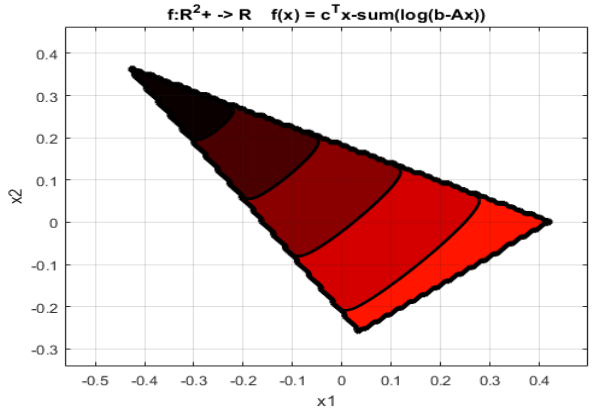
(c) Plot of f for $(n, m) = (2, 20)$



(d) Level sets of f for $(n, m) = (2, 20)$

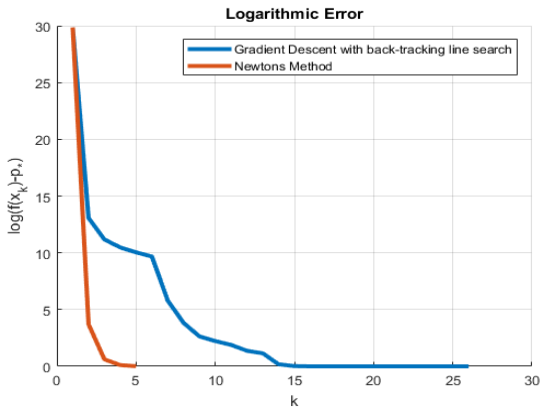


(e) Plot of f for $(n, m) = (2, 20)$

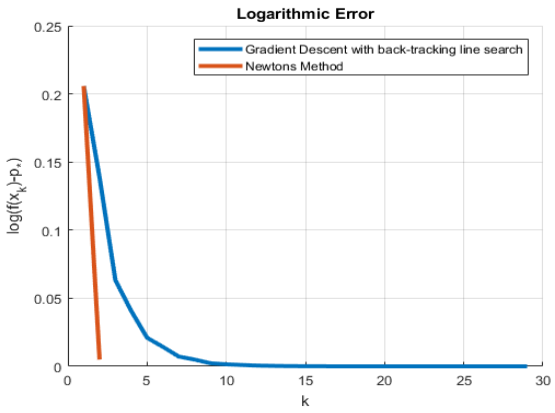


(f) Level sets of f for $(n, m) = (2, 20)$

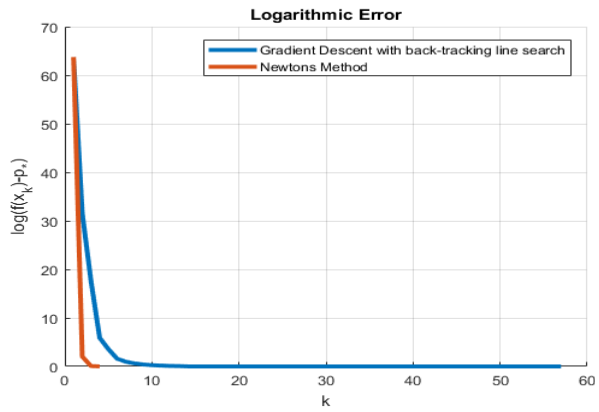
Figure 4: Plot and level sets of f.



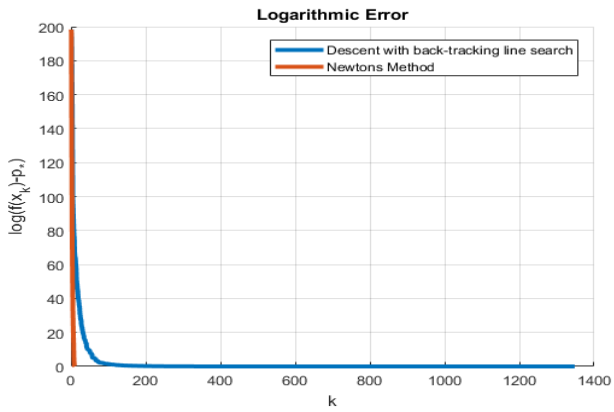
(a) $f_{algo}(x_k) - p_*$ versus k for $(n, m) = (2, 20)$



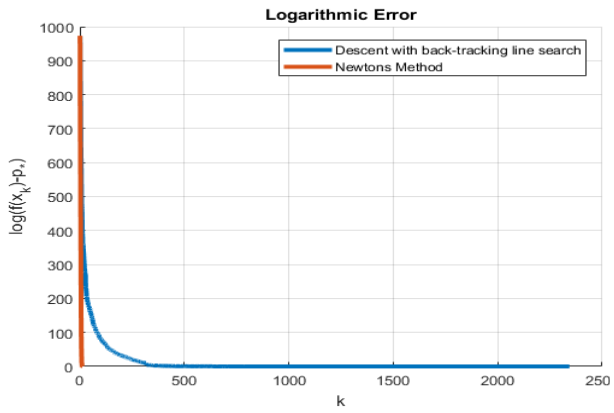
(b) $f_{algo}(x_k) - p_*$ versus k for $(n, m) = (2, 20)$



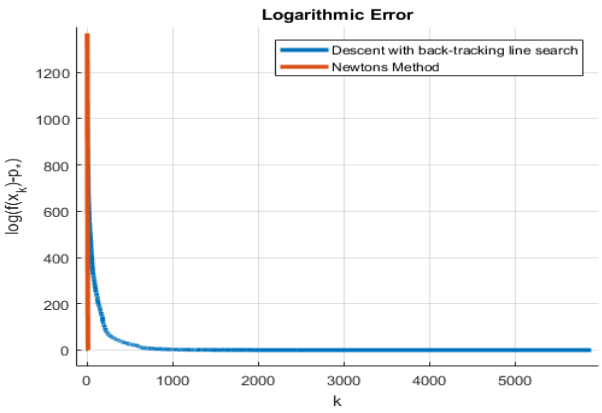
(c) Plot of f for $(n, m) = (50, 200)$



(d) Level sets of f for $(n, m) = (50, 200)$



(e) Plot of f for $(n, m) = (300, 800)$



(f) Level sets of f for $(n, m) = (300, 800)$

Figure 5: Plot and level sets of f.

Exercise B Matlab implementation

Main.m

```
clear ; close all; clc

format long;
disp('Minimization of a quadratic function');
disp(' with positive definite matrix.');
disp('');

% Dimensions of the problem. Indicative
% values are n = 2, 50, 500, 1000.
n=0;
while(n~=2 && n~=50 && n~=500 && n~=1000)
   disp('Give the dimensions of the problem.');
   disp('Acceptable number of dimensions are ');
   disp('2, 50, 500 and 1000.');
   n=input('n : ');
   disp('');
end

% Condition Number
K=0;
while(K~=10 && K~=100 && K~=1000)
   disp('Give the condition number of matrix P.');
   disp('Acceptable number of condition numbers are ');
   disp('10, 100 and 1000.');
   K=input('K : ');
   disp('');
end

% The acceptable error of the optimizer.
e = -1.;
```

```
while(e<0.)
  disp('Give the acceptable error of the optimizer.');
  disp('Acceptable number of error are ');
  disp('the all the positive real numbers.');
  e=input('e : ');
  disp('');
end

% Number used for computing the range
% of the random values for the matrix A.
limit=100;

% The random matrix A constructed
% by n rand vectors of n dimensions.
A=2*limit*rand(n,n)-limit;

% Calculating the singular value decomposition of A.
[U,S,V]=svd(A);

% Constructing the eigenvalues for the matrix P.
lmin = randi(limit);
lmax = lmin*K;
L=diag([lmin;lmax;lmin+(lmax-lmin)*rand(n-2,1)]);

% Our positive definite matrix P with condition number K.
P=U*L*U';
q=2*limit*rand(n,1)-limit; % A random vector q.

% Closed for solution of the quadratic problem
sol=-inv(P)*q;

% Gradient Descent using exact line search.
disp('Gradient Descent using exact line search.');
```

```
xo = randn(n,1);
x = xo;
k1=0;
grad=P*x+q;
grad_norm=norm(grad);
traj1=x;
% Estimating the number of iterations through convergence
    analysis.
k1_e=K*log(((0.5*xo'*P*xo+xo'*q) -...
                (0.5*sol'*P*sol+sol'*q))...
                /e);
while(grad_norm>e)
   x = x-(grad_norm^2/(grad'*P*grad))*grad;
   k1=k1+1;
   grad=P*x+q;
   grad_norm=norm(grad);
   traj1 = [traj1 x];
end
disp(['Estimated iterations : ',num2str(k1_e)]);
disp(['Actual iterations : ',num2str(k1)]);
disp(['Error : ',num2str(norm(sol-x))]);
disp('')
disp('')


% Plot the trajectory of the minimisation algorithm.
if(n==2)
   PlotTrajectory(k1,traj1,P,q,1);
   title('Trajectory of descent using exact line search.');
end

% Gradient Descent using back-tracking line search.
disp('Gradient Descent using back-tracking line search.');
```

```
alpha=0.5;
beta=1.;
while(alpha>=0.5 || alpha<=0.)
  disp('Give a value for the parameter alpha.');
  disp('Acceptable values are in the open interval (0,0.5).')
     ;
  alpha=input('alpha : ');
  disp('');
end
while(beta>=1. || beta<=0.)
  disp('Give a value for the parameter beta.');
  disp('Acceptable values are in the open interval (0,1).');
  beta=input('beta : ');
  disp('');
end

x=xo;
k2=0;
grad=P*x+q;
grad_norm=norm(grad);
traj2=[x];
% Estimated number of iterations using convergence analysis.
k2_e=-(1/log(1-min(2*lmin*alpha,...
       (2*beta*alpha*lmin)/lmax)))*...
       log(((0.5*xo'*P*xo+xo'*q)-...
       (0.5*sol'*P*sol+sol'*q))...
       /e);
while(grad_norm>e)
  t=1;
  while(0.5*t^2*grad'*P*grad-t*grad_norm^2+0.5*x'*P*x+q'*x...
        > 0.5*x'*P*x+q'*x-alpha*t*grad_norm^2)
    t=beta*t;
```

```
    end
    x=x-t*grad;
    k2=k2+1;
    grad=P*x+q;
    grad_norm=norm(grad);
    traj2 = [traj2 x];
end
disp(['Estimated iterations : ',num2str(k2_e)]);
disp(['Actual iterations : ',num2str(k2)]);
disp(['Error : ',num2str(norm(sol-x))]);
disp('')
disp('')

% Plot the trajectory of the minimisation algorithm.
if(n==2)
    PlotTrajectory(k2,traj2,P,q,2);
    title('Trajectory of descent using back-tracking line
        search.');
end

% Plot log(f(x_k)-sol) against each iteration.
figure(3);hold on;
PlotError(k1,traj1,P,q,sol);
PlotError(k2,traj2,P,q,sol);
legend('exact line search','back-tracking line search');

% Plot theoretical and practical iterations
figure(4);
subplot(1,2,1);
hold on;
bar(categorical({'Estimated Iterations'}),k1_e,'FaceColor','
    blue');
bar(categorical({'Actual Iterations'}),k1,'FaceColor','red');
```

```
hold off;
title('Exact line search');
ax2=subplot(1,2,2);
hold on;
bar(categorical({'Estimated Iterations'}),k2_e,'FaceColor','
    blue');
bar(categorical({'Actual Iterations'}),k2,'FaceColor','red');
hold off;
title('Back-tracking line search');

disp('End of experiment.');
disp(['n : ',num2str(n)]);
disp(['K : ',num2str(K)]);
disp(['e : ',num2str(e)]);
disp(['alpha : ',num2str(alpha)]);
disp(['beta : ',num2str(beta)]);
```

PlotTrajectory.m

```
function Plottraj(iter,traj,P,q,fig)

  tr_traj=traj';
  v=zeros(1,size(traj,2));
  for i=1:size(traj,2)
        v(1,i)=0.5*tr_traj(i,:)*P*traj(:,i)+q'*traj(:,i);
  end

  v = sort(v,'ascend');

  [m,idx]=max(abs(traj(1,iter+1)-traj(1,1:iter)));
  distx=abs(traj(1,iter+1)-traj(1,idx))+0.1;
  [m,idx]=max(abs(traj(2,iter+1)-traj(2,1:iter)));
```

```
    disty=abs(traj(2,iter+1)-traj(2,idx))+0.1;
    x=linspace(traj(1,iter+1)-distx,traj(1,iter+1)+distx,150);
    y=linspace(traj(2,iter)-disty,traj(2,iter)+disty,150);
    [X,Y]=meshgrid(x,y);
    func=0.5*(P(1,1).*(X.^2)+P(1,2).*X.*Y+P(2,1).*X.*Y+P(2,2)
        .*(Y.^2))...
          +q(1,1).*X+q(2,1).*Y;

    figure(fig);
    contourf(X,Y,func,v);hold on;
    colormap hot;
    xlabel('x');
    ylabel('y');
    grid on;
    plot(traj(1,:),traj(2,:),'-d','MarkerSize',10,...
          'MarkerEdgeColor','b','MarkerFaceColor','b',...
          'linewidth',3);

    return
end
```

PlotError.m

```
function PlotError(iter,traj,P,q,x_star)

    tr_traj=traj';
    tr_traj=traj';
        v=zeros(1,size(traj,2));
        for i=1:size(traj,2)
                v(1,i)=0.5*tr_traj(i,:)*P*traj(:,i)+q'*traj
                    (:,i);
        end
```

```
    p_star=0.5*x_star'*P*x_star+x_star'*q;
    func = log(v-p_star);
    iter = linspace(1,iter+1,iter+1);

    plot(iter,func,'linewidth',3);
    title('Logarithmic Error');
    xlabel('k');
    ylabel('log(f(x_k)-p_*)');
    grid on;

    return
end
```

Exercise C Matlab implementation

Main.m

```
% Vissarion Konidaris
% Convex Optimization Course
% 1/4/2018

clear ; close all; clc

format long;
disp('Minimization the convex function');
disp(' c''x-sum(log(b-Ax)).');

in=0;
while(in~=1 && in~=2 && in~=3)
  disp('Give the dimensions of the problem.');
  disp('Indicative value pairs are (n,m) = (2,50), ');
  disp('(50,200),(300,800)\nwhere m the number of the ');
  disp('logarithms and n the dimension of input x.');
```

```
  disp('Type 1 for (2,50), 2 for (5,200), 3 for (300,800).');
  in=input('n : ');
  if(in==1)
    n=2;
    m=20;
  elseif(in==2)
    n=50;
    m=200;
  else
    n=300;
    m=800;
  end
  disp('');
end

% The acceptable error of the optimizer.
e = -1.;
while(e<0.)
  disp('Give the acceptable error of the optimizer.');
  disp('Acceptable number of error are ');
  disp('the all the positive real numbers.');
  e=input('e : ');
  disp('');
end


% Number used for computing the range
% of the random values for the matrix A.
limit=100;

% The random matrix A constructed
% by n rand vectors of n dimensions.
A=2*limit*rand(m,n)-limit;
```

```
b=limit*rand(m,1); % A random vector b.
c=2*limit*rand(n,1)-limit; % A random vector c.

cvx_begin
    variable cv_x(n)
    minimize(c'*cv_x-sum(log(b-A*cv_x)))
cvx_end

optimal_value = c'*cv_x-sum(log(b-A*cv_x));

xaxis=linspace(cv_x(1,1)-1,cv_x(1,1)+1,250);
yaxis=linspace(cv_x(2,1)-1,cv_x(2,1)+1,250);
[X,Y]=meshgrid(xaxis,yaxis);
func = ones(size(X,1),size(X,2))*100;
if(n==2)
    for i=1:size(X,1)
        for j=1:size(X,2)
            log_argument = b-A*[X(i,j); Y(i,j)];
            feasible=1;
            for k=1:size(log_argument,1)
                if(log_argument(k,1)<=0)
                    feasible=0;
                    break;
                end
            end
            if(feasible==1)
                func(i,j) = c'*[X(i,j); Y(i,j)]-...
                            sum(log(log_argument));
            end
        end
    end

    figure(1);
```

```
    mesh(X,Y,func,'LineWidth',2);
    rotate3d on;
    axis tight;
    title('f:R^2+ -> R     f(x) = c^Tx-sum(log(b-Ax))');
    zlabel('f(x1,x2)');
    xlabel('x1');
    ylabel('x2');
    xaxis([]);
    grid on;

    figure(2);
    contourf(X,Y,func,10,'Linewidth',2);
    axis tight;
    colormap hot;
    title('f:R^2+ -> R     f(x) = c^Tx-sum(log(b-Ax))');
    xlabel('x1');
    ylabel('x2');
    grid on;

    figure(3);
    contour3(X,Y,func,'Linewidth',2);
    rotate3d on;
    axis tight;
    colormap hot;
    title('f:R^2+ -> R     f(x) = c^Tx-sum(log(b-Ax))');
    xlabel('x1');
    ylabel('x2');
    grid on;
end

% Gradient Descent using back-tracking line search.
disp('Gradient Descent using back-tracking line search.');
```

```
aplha=0.5;
beta=1.;
while(aplha>=0.5 || aplha<=0.)
  disp('Give a value for the parameter aplha.');
  disp('Acceptable values are in the open interval (0,0.5).')
     ;
  aplha=input('aplha : ');
  disp('');
end
while(beta>=1. || beta<=0.)
  disp('Give a value for the parameter beta.');
  disp('Acceptable values are in the open interval (0,1).');
  beta=input('beta : ');
  disp('');
end

xo =zeros(n,1);
x=xo;
k1=0;
grad=c+sum(A.*(b-A*x).^(-1))';
grad_norm=norm(grad);
traj1=[xo];
while(grad_norm>e)
  t=1;
  while(1)
       feasible=1;
       point = b-A*(x-t*grad);
       for i=1:size(point,1)
         if(point(i,1)<=0)
              feasible=0;
              break;
         end
       end
```

```
        if(feasible==0)
            disp(['Not in domain at loop ',num2str(k1+1)]);
            t=beta*t;
            continue;
        end
        break;
    end
    f=c'*x-sum(log(b-A*x));
    while(c'*(x-t*grad)-sum(log(b-A*(x-t*grad)))...
          > f-aplha*t*grad_norm^2)
      t=beta*t;
    end
    x=x-t*grad;
    k1=k1+1;
    grad=c+sum(A.*(b-A*x).^(-1))';
    grad_norm=norm(grad);
    traj1 = [traj1 x];
end
disp(['Actual iterations : ',num2str(k1)]);
disp(['Error : ',num2str(norm(cv_x-x))]);
disp(['Optimal value (cvx_optval): ',...
      num2str(optimal_value)]);
disp(['Optimal value (back-tracking): ',...
      num2str(c'*x-sum(log(b-A*x)))]);
disp('');
disp('');

% Newtons method
disp('Newtons method.');
x=xo;
k2=0;
grad=c+sum(A.*(b-A*x).^(-1))';
hessian=zeros(n,n);
```

```
for i=1:size(A,1)
    hessian=hessian+(A(i,:)'*A(i,:)).*...
            (norm(b(i,1)-A(i,:)*x).^2).^(-1);
end
traj2=[xo];
while(1)
  t=1;
  delta=-inv(hessian)*grad;
  lambda_squared=-grad'*delta;
  if(0.5*lambda_squared<=e)
      break;
  end
  while(1)
      feasible=1;
      point = b-A*(x+t*delta);
      for i=1:size(point,1)
        if(point(i,1)<=0)
            feasible=0;
            break;
        end
      end
      if(feasible==0)
          disp(['Not in domain at loop ',num2str(k2+1]));
          t=beta*t;
          continue;
      end
      break;
  end
  f=c'*x-sum(log(b-A*x));
  while(c'*(x+t*delta)-sum(log(b-A*(x+t*delta)))...
        > f-aplha*t*lambda_squared)
    t=beta*t;
  end
```

```
    x=x+t*delta;
    k2=k2+1;
    grad=c+sum(A.*(b-A*x).^(-1))';
    hessian=zeros(n,n);
    for i=1:size(A,1)
        hessian=hessian+(A(i,:)'*A(i,:)).*...
                (norm(b(i,1)-A(i,:)*x).^2).^(-1);
    end
    traj2 = [traj2 x];
end
disp(['Actual iterations : ',num2str(k2)]);
disp(['Error : ',num2str(norm(cv_x-x))]);
disp(['Optimal value (cvx_optval): ',...
        num2str(optimal_value)]);
disp(['Optimal value (Newtons): ',...
        num2str(c'*x-sum(log(b-A*x)))]);

v1=zeros(1,size(traj1,2));
for i=1:size(traj1,2)
    v1(1,i)=c'*traj1(:,i)-sum(log(b-A*traj1(:,i)));
end
iterations1 = linspace(1,k1+1,k1+1);

v2=zeros(1,size(traj2,2));
for i=1:size(traj2,2)
    v2(1,i)=c'*traj2(:,i)-sum(log(b-A*traj2(:,i)));
end
iterations2 = linspace(1,k2+1,k2+1);

figure(4);hold on;
semilogy(iterations1,v1-optimal_value,'linewidth',3);
semilogy(iterations2,v2-optimal_value,'linewidth',3);
title('Logarithmic Error');
```

```
xlabel('k');
ylabel('log(f(x_k)-p_*)');
legend('Descent with back-tracking line search','Newtons
   Method');
grid on;
```