ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

Course "Optimization" Report

Exercise 2

Georgios Frangias 2018030086
gfrangias@tuc.gr

Course Professor:
Athanasios Liavas

Technical University of Crete

School of Electrical & Computer Engineering

November 2022

A. We consider a simple quadratic optimization problem.

(a) Let $f : \mathbb{R}^n \to \mathbb{R}$. For fixed $\mathbf{x} \in \mathbb{R}^n$ and $m > 0$, let $g_{\mathbf{x}} : \mathbb{R}^n \to \mathbb{R}$ be defined as

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

i. Compute $\nabla g_{\mathbf{x}}(\mathbf{y})$

$$\nabla f(\mathbf{x})^T = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} & \frac{\partial f(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

$$\nabla f(\mathbf{x})^T\mathbf{y} = \frac{\partial f(\mathbf{x})}{\partial x_1}\mathbf{y_1} + \frac{\partial f(\mathbf{x})}{\partial x_2}\mathbf{y_2} + \ldots + \frac{\partial f(\mathbf{x})}{\partial x_n}\mathbf{y_n}$$

$$\nabla\left(\nabla f(\mathbf{x})^T\mathbf{y}\right) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \dots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix} = \nabla f(\mathbf{x}) \qquad (1)$$

$$\nabla\left(\frac{m}{2}||\mathbf{x} - \mathbf{y}||_2^2\right) = \frac{m}{2}\nabla\left((y_1 - x_1)^2 + (y_2 - x_2)^2 + \ldots + (y_n - x_n)^2\right) =$$

$$= \frac{m}{2}\begin{bmatrix} 2y_1 - 2x_1 \\ 2y_2 - 2x_2 \\ \vdots \\ 2y_n - 2x_n \end{bmatrix} = m(\mathbf{y} - \mathbf{x}) \qquad (2)$$

$$\nabla g_{\mathbf{x}}(\mathbf{y}) = \nabla\left(f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + \frac{m}{2}\|\mathbf{y} - \mathbf{x}\|_2^2\right) \Rightarrow$$

$$\Rightarrow \nabla g_{\mathbf{x}}(\mathbf{y}) = \nabla\left(f(\mathbf{x}) + \nabla f(\mathbf{x})^T\mathbf{y} - \nabla f(\mathbf{x})^T\mathbf{x}\right) + \nabla\left(\frac{m}{2}\|\mathbf{y} - \mathbf{x}\|_2^2\right) \Rightarrow$$

$$\Rightarrow \nabla g_{\mathbf{x}}(\mathbf{y}) = \nabla\left(\nabla f(\mathbf{x})^T\mathbf{y}\right) + \nabla\left(\frac{m}{2}\|\mathbf{y} - \mathbf{x}\|_2^2\right) \overset{(1),(2)}{=} \nabla f(\mathbf{x}) + m(\mathbf{y} - \mathbf{x})$$

ii. Compute the optimal point, $\mathbf{y}_* = \underset{\mathbf{y}}{\text{argmin}} g_{\mathbf{x}}(\mathbf{y})$, and the optimal value, $g_{\mathbf{x}}(\mathbf{y}_*)$.

$$\nabla g_{\mathbf{x}}(\mathbf{y}_*) = 0 \Rightarrow$$

$$\Rightarrow \nabla f(\mathbf{x}) + m(\mathbf{y}_* - \mathbf{x}) = 0 \Rightarrow$$

$$\Rightarrow \mathbf{y}_* - \mathbf{x} = -\frac{1}{m}\nabla f(\mathbf{x}) \Rightarrow$$

$$\Rightarrow \mathbf{y}_* = \mathbf{x} - \frac{1}{m}\nabla f(\mathbf{x})$$

$$g_{\mathbf{x}}(\mathbf{y}_*) = f(\mathbf{x}) + \nabla f(\mathbf{x})^T\left(-\frac{1}{m}\nabla f(\mathbf{x})\right) + \frac{m}{2}||-\frac{1}{m}\nabla f(\mathbf{x})||_2^2 \Rightarrow$$

$$\Rightarrow g_{\mathbf{x}}(\mathbf{y}_*) = f(\mathbf{x}) - \frac{1}{m}\nabla f(\mathbf{x})^T\nabla f(\mathbf{x}) + \frac{1}{2}||\nabla f(\mathbf{x})||_2^2 \Rightarrow$$

$$\Rightarrow g_{\mathbf{x}}(\mathbf{y}_*) = f(\mathbf{x}) + \frac{m-2}{2m}||\nabla f(\mathbf{x})||_2^2$$

B. We proceed to the solution of convex quadratic problems. Our goal is to study the behavior of the gradient method, with exact and backtracking line search.

Consider the problem

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

where $\mathbf{P} \in \mathbb{R}^{n\times n}$, $\mathbf{P} = \mathbf{P}^T \succ \mathbf{O}$ and $\mathbf{q} \in \mathbb{R}^n$ (indicate values of n are $n = 2, 50, 10^2, 10^3$).

i. A random positive definite matrix P can be constructed in many ways. To fully control the condition number of the matrix, we can act as follows. Every positive definite P can be expressed as

$$\mathbf{P} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$$

with $\mathbf{U}, \mathbf{\Lambda} \in \mathbb{R}^{n\times n}$, $\mathbf{U}\mathbf{U}^T = \mathbf{U}^T\mathbf{U} = \mathbf{I}_n$, and $\mathbf{\Lambda} = \mathbf{diag}(\lambda_\mathbf{1}, \ldots, \lambda_\mathbf{n})$, with $\lambda_i > 0$, for $i = 1, \ldots, n$. The columns of U are the eigenvalues of P and the elementsof the diagonal of $\mathbf{\Lambda}$ are the eigenvalues of P.

(a) To create a random orthonormal U, create a random $(n \times n)$ matrix A as follows

$$\texttt{A} = \texttt{randn(n,n);}$$

and then compute its singular value decomposition as

$$\texttt{[U,S,V]} = \texttt{svd(A);}$$

Compute $\texttt{U} * \texttt{U}'$ and $\texttt{U}' * \texttt{U}$. What do you observe?

It is observed that $\texttt{U} * \texttt{U}'$ and $\texttt{U}' * \texttt{U}$ are equal to the identity matrix $\mathbf{I_n}$ for any random value of matrix **A**. This indicates that the matrix **U** is orthonormal.

(b) To construct the eigenvalues λ_i, for $i = 1, \ldots, n$, select the smallest and largest, λ_{min} and λ_{max}, respectively. You can create the other eigenvalues in any way you wish. For example, you can create $(n - 2)$ random numbers, uniformly distributed in the interval $[\lambda_{min}, \lambda_{max}]$ using the command

$$\texttt{z} = \lambda_{min} + (\lambda_{max} - \lambda_{min}) * \texttt{rand(n-2,1);}$$

and the vector of the n eigenvalues as

$$\texttt{eig\_P} = [\lambda_{min}; \lambda_{max}; \texttt{z}];$$

Matrix $\mathbf{\Lambda}$ may be constructed as

$$\mathbf{\Lambda} = \texttt{diag(eig\_P);}$$

The condition number of the problem is $\mathcal{K} := \frac{\lambda_{min}}{\lambda_{max}}$.

The requested matrices and vectors are created with the following MATLAB code.

```
%%%%%%%%%%%%%%%%%B.i.(b)%%%%%%%%%%%%%%%%%%%%

K = 1000;
l_min = abs(randn(1,1));
l_max = K*l_min;

z = l_min + (l_max-l_min)*rand(n-2,1);
eig_P = [l_min;l_max;z];
Lambda = diag(eig_P);
```

ii. Construct random vector q and random positive definite matrix P, with condition number $\mathcal{K}$ (indicative values, $\mathcal{K} = 10, 10^2, 10^3$).

```
q = 100 * rand(n,1);
P = U * Lambda * U';
```

iii. Solve problem (2) using the closed-form solution, and compute the optimal point, $\mathbf{x}_*$, and the optimal value, $p_* = f(\mathbf{x}_*)$.

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x}$$

For $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$, $\mathbf{q} = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{bmatrix}$ and $\mathbf{P} = \begin{bmatrix} p_{11} & p_{12} & \dots & p_{1n} \\ p_{21} & p_{22} & \dots & p_{2n} \\ \vdots & & \ddots & \vdots \\ p_{m1} & p_{m2} & \dots & p_{mn} \end{bmatrix}$

$$f(\mathbf{x}) = \frac{1}{2}\left(x_1(x_1p_{11} + x_2p_{21} + \ldots + x_np_{n1}) + \ldots + x_n(x_1p_{1n} + x_2p_{2n} + \ldots + x_np_{nn})\right) + q_1x_1 + q_2x_2 + \ldots + q_nx_n$$

$$\frac{\partial f(\mathbf{x})}{\partial x_1} = \frac{1}{2}(2x_1p_{11} + x_2p_{21} + \ldots + x_np_{n1} + x_2p_{12} + \ldots + x_np_{1n}) + q_1 \overset{\mathbf{P}=\mathbf{P}^T}{=} x_1p_{11} + x_2p_{21} + \ldots + x_np_{1n} + q_1$$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix} = \begin{bmatrix} x_1p_{11} + x_2p_{21} + \ldots + x_np_{1n} + q_1 \\ x_1p_{21} + x_2p_{22} + \ldots + x_np_{2n} + q_2 \\ \vdots \\ x_1p_{n1} + x_2p_{n2} + \ldots + x_np_{nn} + q_n \end{bmatrix} = \mathbf{P}\mathbf{x} + \mathbf{q}$$

So the optimal value can be found for

$$\nabla f(\mathbf{x}_*) = 0 \Rightarrow \mathbf{P}\mathbf{x}_* + \mathbf{q} = 0$$

In MATLAB this is translated into

```
x_opt = linsolve(P,-q);
f_opt = 0.5*x_opt'*P*x_opt+q'*x_opt;
```

iv. Solve problem (2) using cvx

Using cvx the problem is solved with

```
cvx_begin
    variable x(n)
    minimize( 0.5*x'*P*x+q'*x )
cvx_end
```

It should be marked that the cvx solution is equal to the closed-form one.

v. Solve problem (2) using the gradient algorithm (with exact and backtracking line search). How many iterations are necessary for convergence, if $\mathcal{K} = 1$ and you use the exact line search?

For the backtracking line search algorithm the given code from Amir Beck's book is used. Now, for the exact line search algorithm it should be computed what the step should be for this particular function. In order to do that, there should be a vector $\Delta\mathbf{x} \in \mathbb{R}^n$ which is a descent direction of f at $\mathbf{x}$ and the step should be

$$\tilde{\mathbf{t}}_{\mathbf{exact}} = \min_{\mathbf{t}\geq 0} f(\mathbf{x} + \mathbf{t}\Delta\mathbf{x})$$

So

$$f(\mathbf{x} + \mathbf{t}\Delta\mathbf{x}) = \frac{1}{2}(\mathbf{x} + \mathbf{t}\Delta\mathbf{x})^T \mathbf{P}(\mathbf{x} + \mathbf{t}\Delta\mathbf{x}) + \mathbf{q}^T(\mathbf{x} + \mathbf{t}\Delta\mathbf{x}) =$$

$$= \frac{1}{2}(\mathbf{x}^T + \mathbf{t}^T\Delta\mathbf{x}^T)(\mathbf{P}\mathbf{x} + \mathbf{P}\mathbf{t}\Delta\mathbf{x}) + \mathbf{q}^T\mathbf{x} + \mathbf{q}^T\mathbf{t}\Delta\mathbf{x} =$$

$$= \frac{1}{2}(\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{x}^T\mathbf{P}\mathbf{t}\Delta\mathbf{x} + \mathbf{t}^T\Delta\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{t}^T\Delta\mathbf{x}^T\mathbf{P}\mathbf{t}\Delta\mathbf{x}) + \mathbf{q}^T\mathbf{x} + \mathbf{q}^T\mathbf{t}\Delta\mathbf{x} =$$

$$= \frac{1}{2}(\Delta\mathbf{x}^T\mathbf{P}\Delta\mathbf{x})\mathbf{t}^2 + (\Delta\mathbf{x}^T\mathbf{P}\mathbf{x} + \Delta\mathbf{x}^T\mathbf{q})\mathbf{t} + \frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x} =$$

$$= \frac{1}{2}(\Delta\mathbf{x}^T\mathbf{P}\Delta\mathbf{x})\mathbf{t}^2 + (\Delta\mathbf{x}^T\mathbf{P}\mathbf{x} + \Delta\mathbf{x}^T\mathbf{q})\mathbf{t} + f(\mathbf{x})$$

For $\tilde{\mathbf{t}}_{\mathbf{exact}}$

$$\frac{\mathrm{d}f(\mathbf{x} + \tilde{\mathbf{t}}_{\mathbf{exact}}\Delta\mathbf{x})}{\mathrm{d}\mathbf{t}} = 0 \Rightarrow$$

$$\Rightarrow (\Delta\mathbf{x}^T\mathbf{P}\Delta\mathbf{x})\tilde{\mathbf{t}}_{\mathbf{exact}} + \Delta\mathbf{x}^T(\mathbf{P}\mathbf{x} + \mathbf{q}) = 0 \Rightarrow$$

$$\tilde{\mathbf{t}}_{\mathbf{exact}} = -\frac{\Delta\mathbf{x}^T(\mathbf{P}\mathbf{x} + \mathbf{q})}{\Delta\mathbf{x}^T\mathbf{P}\Delta\mathbf{x}} = -\frac{\Delta\mathbf{x}^T\nabla f(\mathbf{x})}{\Delta\mathbf{x}^T\mathbf{P}\Delta\mathbf{x}}$$

In the gradient algorithm the descent direction $\Delta\mathbf{x}$ is equal to $-||\nabla f(\mathbf{x})||$. So

$$\tilde{\mathbf{t}}_{\mathbf{exact}} = \frac{||\nabla f(\mathbf{x})||^2}{||\nabla f(\mathbf{x})||^T\mathbf{P}||\nabla f(\mathbf{x})||}$$

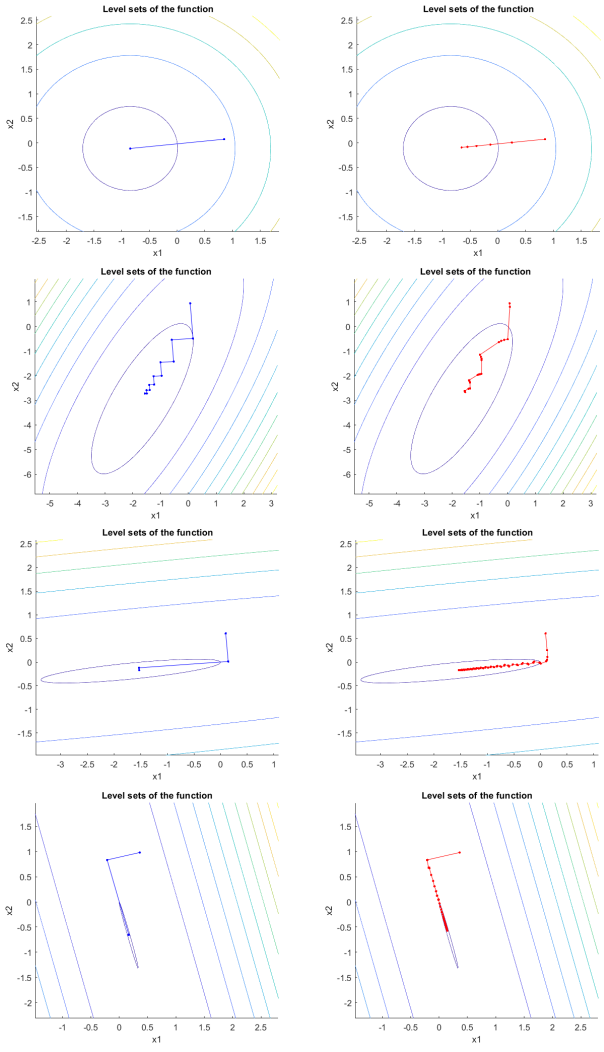
In MATLAB this is

```
while(norm(grad)>epsilon)
    iter=iter+1;
    t=norm(grad)^2/(grad'*P*grad);
    x=x-t*grad;
    grad=g(x);
    fun_val=f(x);
    fprintf('iter_number = %3d norm_grad = %2.6f funval = %2.6f \n',...
            iter, norm(grad), fun_val);
end
```

For $\mathcal{K} = 1 \Rightarrow \lambda_{min} = \lambda_{max}$. This means that the eigenvalues generated will all be equal to each other and that entails that the matrix $\mathbf{\Lambda} = c\mathbf{I}_n$ where $c \in \mathbb{R}^{++}$. Subsequently, $\mathbf{P}$ will have the same property. This means that the norm of the gradient of f will be zero, ending the exact line search on the first iteration.

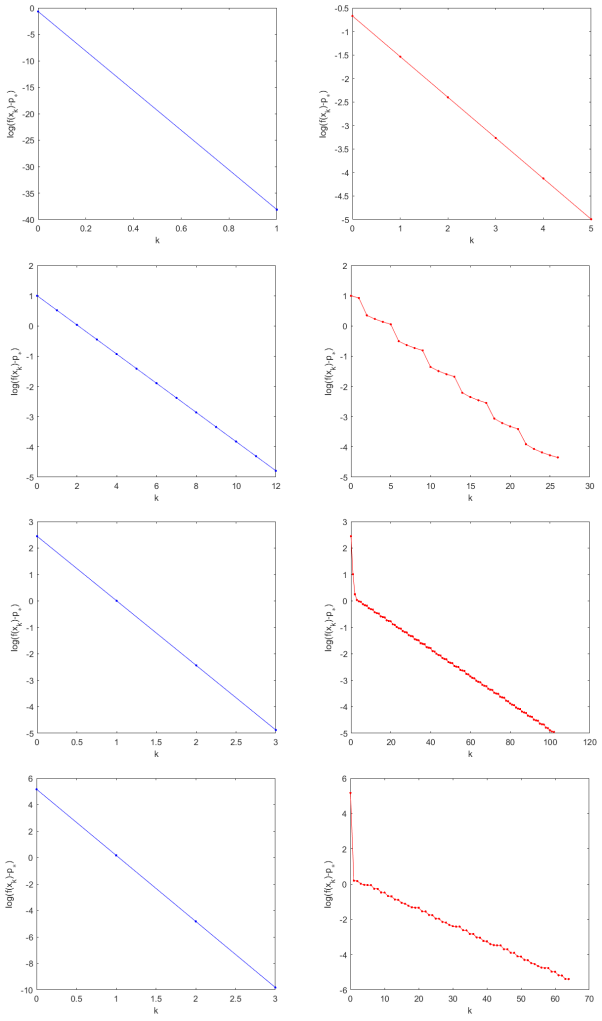
vi. For $n = 2$, construct a contour plot, with levels the values $f(\mathrm{x}_k)$. Then, on top of this plot, plot the trajectories of $\{\mathrm{x}_k\}$ produced by the two algorithms. (For problems with large condition number, you must observe the "zig-zag" effect.)

In blue are the points of the exact line search algorithm and in red of the backtracking algorithm. Each row of diagrams corresponds to the different $\mathcal{K}$(with $\mathcal{K} = 1, 10, 100, 1000$) and in that order.



The zig-zag effect is more evident for $\mathcal{K} = 10$ for the exact line algorithm and for $\mathcal{K} = 100$ for the backtracking algorithm.

vii. Plot quantity $\log\left(f\left(\mathbf{x}_k\right)-p_*\right)$, produced by the two algorithms, versus k. What is the slope of this plot? What do you observe for different condition numbers?



The slope of the exact line search algorithm is linear, which means that the convergence is happening exponentially. For the backtracking algorithm more iterations are needed and the slope of the plot isn't stable but has a rippling effect except for when $\mathcal{K}=1$. For bigger condition numbers $\mathcal{K}$ both algorithms need more iterations to converge. The frequency of the rippling effect is not affected by $\mathcal{K}$.

viii. Using the convergence analysis results for strongly convex functions, and the values of $f\left(\mathrm{x}_0\right), p_*$ and ϵ, compute the minimum number of iterations that guarantees solution within accuracy ϵ, i.e., $f\left(\mathbf{x}_k\right)-p_* \leq \epsilon$, and compare it with the number of iterations performed by the algorithms. What do you observe?

$$k_\epsilon \approx \mathcal{K} \log\left(\frac{f\left(\mathbf{x}_0\right)-p_*}{\epsilon}\right)$$

```
k_max = K*log((fun_val_hist_e(1)-f_opt)/epsilon);
```

After attempting to calculate using different $\mathcal{K}$ and `n` values, `k_max` is always greater than the number of iterations performed.

C.

Let $\mathbf{c} \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^m$, and $\mathbf{A} \in \mathbb{R}^{m \times n}$, with rows $\mathbf{a}_i^T$, for $i = 1, \ldots, m$. Consider the function $f : \operatorname{dom} f \subseteq \mathbb{R}^n \to \mathbb{R}$, defined as

$$f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} - \sum_{i=1}^{m} \log\left(b_i - \mathbf{a}_i^T\mathbf{x}\right) = \mathbf{c}^T\mathbf{x} - \mathtt{sum}(\log(\mathbf{b} - \mathbf{A}\mathbf{x}))$$

with $\mathbf{x} \in \mathbb{R}^n$ and $m > n$ (or $m \gg n$) (indicative value pairs $(n, m) = (2, 20), (50, 200)$. If you have patience and enough memory in the computer, you can set $(n, m) = (300, 800)$ or larger values)

Some observations are as follows (see Boyd-Vandenberghe, pages 141, 419-422, 458, 459, 472, 492)

- The set $\mathbf{dom}f$ contains only the points $\mathbf{x} \in \mathbb{R}^n$ for which the arguments of the logarithms are positive. Prove that

  (a) The set $\mathbf{dom}f$ is convex

  $$\mathbf{dom}f = \left\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}_i^T\mathbf{x} < b_i,\ \forall i = 1, 2, \ldots, m\right\}$$

  Let $\mathbf{x_1}, \mathbf{x_2} \in \mathbf{dom}f$ and $\theta \in [0, 1]$. For $\mathbf{dom}f$ to be a convex set the following statement should hold.

  $$\begin{aligned}
  &\theta\mathbf{x_1} + (1-\theta)\mathbf{x_2} \in \mathbf{dom}f \Rightarrow \\
  &\Rightarrow \mathbf{a}_i^T(\theta\mathbf{x_1} + (1-\theta)\mathbf{x_2}) < b_i,\ \forall i = 1, 2, \ldots, m \Rightarrow \\
  &\Rightarrow \theta\mathbf{a}_i^T\mathbf{x_1} + (1-\theta)\mathbf{a}_i^T\mathbf{x_2} < b_i,\ \forall i = 1, 2, \ldots, m \Rightarrow \\
  &\Rightarrow \theta\mathbf{a}_i^T\mathbf{x_1} + (1-\theta)\mathbf{a}_i^T\mathbf{x_2} < \theta b_i + (1-\theta)b_i,\ \forall i = 1, 2, \ldots, m
  \end{aligned}$$

  Which is true since $\mathbf{x_1}, \mathbf{x_2} \in \mathbf{dom}f \Rightarrow \begin{cases} \mathbf{a}_i^T\mathbf{x_1} < b_i & \forall i = 1, 2, \ldots, m \\ \mathbf{a}_i^T\mathbf{x_2} < b_i & \forall i = 1, 2, \ldots, m \end{cases} \Rightarrow$

  $$\Rightarrow \begin{cases} \theta\mathbf{a}_i^T\mathbf{x_1} < \theta b_i & \forall i = 1, 2, \ldots, m \\ (1-\theta)\mathbf{a}_i^T\mathbf{x_2} < (1-\theta)b_i & \forall i = 1, 2, \ldots, m \end{cases}$$

  (b) Function f is convex

  It should be proven that the Hessian matrix of f is positive semi-definite ($\nabla^2 f(x) \succeq 0$).

  $$\nabla^2 f(\mathbf{x}) = \nabla^2\left(\mathbf{c}^T\mathbf{x} - \sum_{i=1}^{m} \log\left(b_i - \mathbf{a}_i^T\mathbf{x}\right)\right) \Rightarrow$$

  $$\Rightarrow \nabla^2 f(\mathbf{x}) = \nabla^2\left(\mathbf{c}^T\mathbf{x}\right) - \sum_{i=1}^{m} \nabla^2\left(\log\left(b_i - \mathbf{a}_i^T\mathbf{x}\right)\right)$$

  $$\nabla^2\left(\mathbf{c}^T\mathbf{x}\right) = 0$$

$$\nabla^2 \left(\log \left(b_i - \mathbf{a}_i^T \mathbf{x} \right) \right) = - \frac{\mathbf{a}_i \mathbf{a}_i^T}{\left(b_i - \mathbf{a}_i^T \mathbf{x} \right)^2}$$

So the Hessian would be

$$\nabla^2 f(\mathbf{x}) = \sum_{i=1}^{m} \frac{\mathbf{a}_i \mathbf{a}_i^T}{\left(b_i - \mathbf{a}_i^T \mathbf{x} \right)^2}$$

Let vector $\mathbf{v} \in \mathbb{R}^n$, then the $n \times n$ matrix $\mathbf{a}_i \mathbf{a}_i^T$ is positive semi-definite if and only if

$$\mathbf{v}^T \mathbf{a}_i \mathbf{a}_i^T \mathbf{v} \geq 0 \Rightarrow$$
$$\Rightarrow \mathbf{a}_i^T \mathbf{v} \mathbf{a}_i^T \mathbf{v} \geq 0 \Rightarrow$$
$$\Rightarrow (a_1 v_1 + a_2 v_2 + \ldots + a_n v_n)^2 \geq 0$$

Which is true and therefore $\nabla^2 f(x) \succeq 0$ and f is convex.

- Observe that if $b_i > 0$, for $i = 1, \ldots, m$, then $\operatorname{dom} f \neq \emptyset$, because $\mathbf{x} = \mathbf{0}$ is a feasible point (in the experiments, it makes sense to always use this convention).
- Function f may be unbounded from below. In this case, the problem has no solution (no need to do something for it - cvx will help you to identify these cases).
- If a solution $\mathbf{x}_*$ exists, then it lies in the interior of $\operatorname{dom} f$, because when we approach the boundary of $\operatorname{dom} f$ the value of the function increases without bound. This means that a necessary and sufficient condition for $\mathbf{x}_*$ to be an optimal point is $\nabla f(\mathbf{x}_*) = \mathbf{0}$.

Our study will proceed as follows.

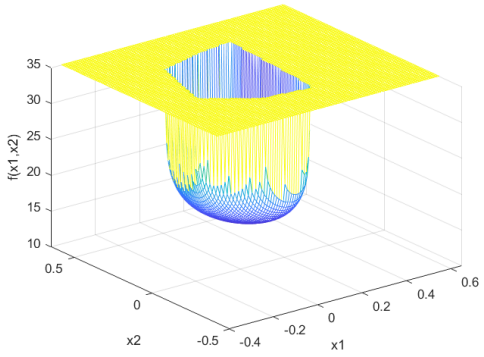
(a) Minimize f using the `cvx`. If the problem has a solution, then `cvx` will compute it, otherwise it will display a message saying that the problem has no solution.

Using `cvx` for the minimization of f the domain should be defined in the `subject to` clause.

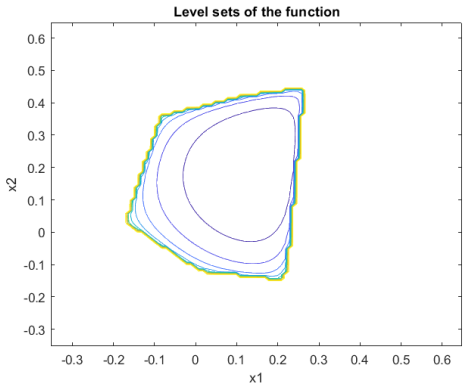
```
cvx_begin
    variable x(n)
    minimize( c'*x-sum(log(b-A*x)) )
    subject to
        min(b-A*x)>0;
cvx_end
```

(b) If $n = 2$, then plot f and its level sets in the neighborhood of the optimum point. You can check whether a point x belongs to the domain of f by checking the argument of the logarithm at this point. If a point x belongs to $\operatorname{dom} f$, then you can compute the value of f at this point. Otherwise, you can give an arbitrarily large value to f at this point (for example, $f(\mathbf{x}) = 10^3$).

On all points that are outside the domain f gets the value 35 to distinguish it from all other values inside the domain.



Plot of f in the neighborhood of $\mathbf{x}_*$



Contour of f in the neighborhood of $\mathbf{x}_*$

(c) Assuming that $\mathbf{x} = \mathbf{0}$ is a feasible point, minimize f using the gradient algorithm with backtracking line search, starting from $\mathbf{x}_0 = \mathbf{0}$. The implementation will have the following main difference from the baseline implementation.

- In step $(k+1)$, given the vectors $\mathbf{x}_k$ and $\Delta\mathbf{x}_k$, and having set $t = 1$, before starting the backtracking, you should check whether the point $\mathbf{x}_k + t\Delta\mathbf{x}_k$ belongs to $\operatorname{dom} f$ or not. If it does not belong to $\operatorname{dom} f$, then you must put $t := \beta t$ (β is the backtracking parameter) and repeat this process until you find a point that belongs to $\operatorname{dom} f$.[2] When you arrive at a point that belongs to $\operatorname{dom} f$, then you can proceed to the typical backtracking line search.

It is attested that $\mathbf{0}$ is a feasible point by creating $\mathbf{b}$ as a strictly positive vector. It is needed to compute $\nabla f(\mathbf{x})$.

$$\nabla\left(\mathbf{c}^T\mathbf{x}\right) = \mathbf{c}$$

$$\nabla\left(\sum_{i=1}^{m} log\left(b_i - \mathbf{a}_i^T\mathbf{x}\right)\right) = \sum_{i=1}^{m} \nabla\left(log\left(b_i - \mathbf{a}_i^T\mathbf{x}\right)\right)$$

$$\nabla log\left(b_i - \mathbf{a}_i^T\mathbf{x}\right) = -\frac{\mathbf{a}_i}{b_i - \mathbf{a}_i^T\mathbf{x}}$$

So the gradient should be

$$\nabla f(\mathbf{x}) = \mathbf{c} + \sum_{i=1}^{m} \frac{\mathbf{a}_i}{b_i - \mathbf{a}_i^T\mathbf{x}}$$

The MATLAB equivalent is

```
g = @(x) diag(c+sum(A./(b-A*x)));
```

And the only change on the backtracking line search algorithm is this extra condition, in order to check if the new point after the step is in the domain

```
    while (constr(x-t*grad)<=0)
        t=beta*t;
        fprintf('Out of domain! iter_number = %3d new step = %2.6f\n',...
        iter,t);
    end
```

It is observed that the furthest the point is from the optimal point $\mathbf{x}_*$, the more are the iterations required to get to a point inside the domain.

(d) Following the analogous procedure, minimize function f using the Newton algorithm.

Algorithm 9.5 from "Convex Optimization" by S.Boyd and L.Vandenberghe was used for the implementation of the Newton algorithm.

1. *Compute the Newton step and decrement.*

$$\Delta\mathbf{x}_{nt} := -\nabla^2 f(\mathbf{x})^{-1}\nabla f(\mathbf{x}); \quad \lambda^2 := \nabla f(\mathbf{x})^T \nabla^2 f(\mathbf{x})^{-1}\nabla f(\mathbf{x})$$

2. *Stopping criterion.*

$$\textbf{quit} \text{ if } \lambda^2/2 \leq \epsilon$$

3. *Line search.*

Choose step size t by backtracking line search.

4. *Update.*

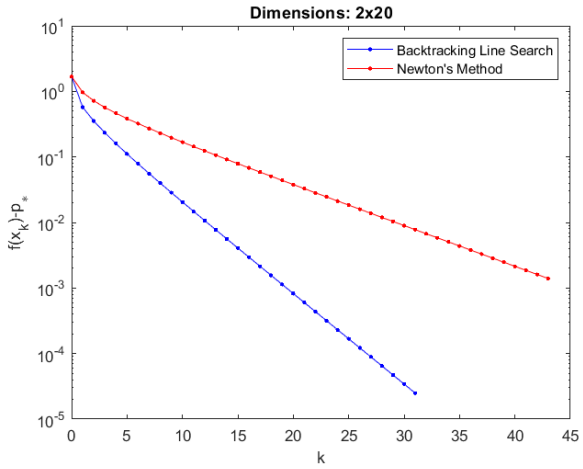
$$\mathbf{x} := \mathbf{x} + t\Delta\mathbf{x}_{nt}$$

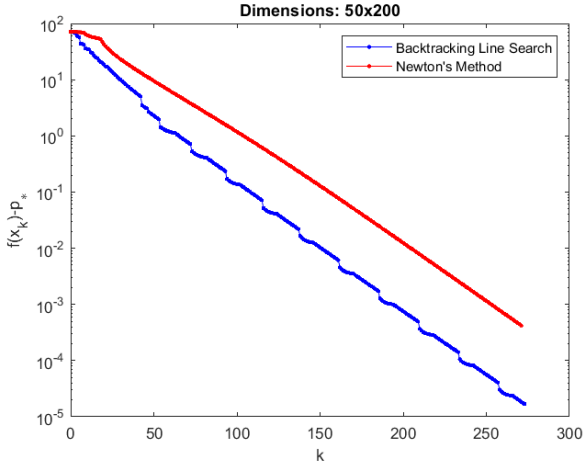
This can be converted to MATLAB code in the following way as written in the file `newton_method.m`

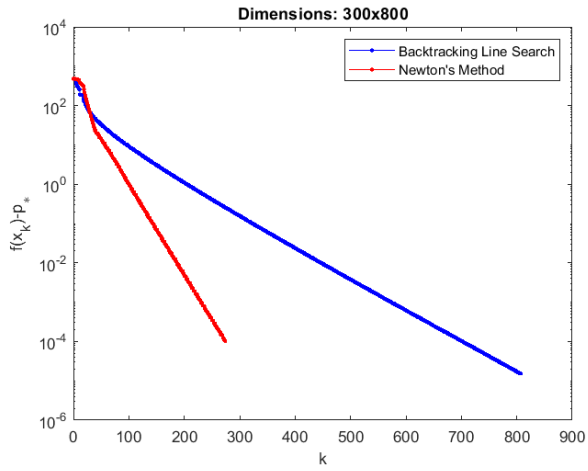
```
x=x0;
x_hist=x0;
grad=g(x);
fun_val=f(x);
hessian=h(x);
fun_val_hist=fun_val;
newton_s=-inv(hessian)*grad;    %Newton step
dec=grad'*inv(hessian)*grad;    %Newton decrement
iter=0;
while (dec/2>epsilon)
    iter=iter+1;
    t=s;
    while (constr(x-t*grad)<=0)
        t=beta*t;
        fprintf('Out of domain! iter_number = %3d new step = %2.6f\n',...
        iter,t);
    end
    while (fun_val-f(x-t*grad)<alpha*t*norm(grad)^2)
        t=beta*t;
    end
    x=x+t*newton_s;
    x_hist=[x_hist x];
    fun_val=f(x);
    fun_val_hist=[fun_val_hist fun_val];
    grad=g(x);
    hessian=h(x);
    newton_s=-inv(hessian)*grad;
    dec=grad'*inv(hessian)*grad;

    fprintf('iter_number = %3d decrement/2 = %2.6f fun_val = %2.6f \n',...
    iter,0.5*dec,fun_val);
```

(e) Plot, using `semilogy`, quantities $(f_{gradient}(\mathbf{x}_k) - p_*)$ and $(f_{newton}(\mathbf{x}_k) - p_*)$, as a function of step k. What do you observe for small and large values of the pair (n,m)?

The semilogy plot is located in the next page. It can be observed that for small dimensions and in the first iterations the backtracking line search algorithm is much faster that Newton's method. As the dimensions are becoming larger it seems that the quantity $(f_{newton}(\mathbf{x}_k) - p_*)$ is decreasing much faster although it lacks behind at first.


Dimensions: 2x20
Backtracking Line Search
Newton's Method
f(x_k)-p_*
k


Dimensions: 50x200
Backtracking Line Search
Newton's Method
f(x_k)-p_*
k


Dimensions: 300x800
Backtracking Line Search
Newton's Method
f(x_k)-p_*
k