COMP0120 Numerical Optimisation

Assignment 4

1 Exercise 2

The constraint minimisation problem with its objective function and constraint equation:

$$\begin{aligned} \min_{(x,y)} f(x,y) = (x-2y)^2 + (x-2)^2 \\ x - y = 4 \end{aligned} \tag{1}$$

Because we do not have dual constraints, λ^* term but only have primary constraints $h_i(x,y)$ term, Karush-Kuhn-Tucker conditions are simplified and shown as below:

$$\begin{aligned} \nabla f(x,y) + \sum_{i=1}^{p} \nu_i^* \nabla h_i(x,y) = 0 \\ h_i(x,y) = x - y - 4 = 0 \end{aligned} \tag{2}$$

where,

$$\begin{aligned} \nabla f(x,y) = \begin{cases} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{cases} = \begin{cases} 4x - 4y - 4 \\ 8y - 4x \end{cases} \\ \nabla h(x,y) = \begin{cases} \frac{\partial h(x,y)}{\partial x} \\ \frac{\partial h(x,y)}{\partial y} \end{cases} = \begin{cases} 1 \\ -1 \end{cases} \end{aligned} \tag{3}$$

Inserting the equation 3 into the equation 2, we can obtain:

$$\begin{cases} 4x^* - 4y^* - 4 + \nu_i^* = 0 \\ 8y^* - 4x^* - \nu_i^* = 0 \\ x^* - y^* - 4 = 0 \end{cases} \rightarrow \begin{cases} x^* = 5 \\ y^* = 1 \\ \nu^* = -12 \end{cases} \tag{4}$$

2 Exercise 4

The signal can be generated as:

$$\tilde{y} = Ax + e \tag{5}$$

where, sparse signal $x = \{x_n\}_{n=1...N}$ of length $N = 2^{13}$ consisting of:(T = 100 randomly distributed spikes with values ± 1 and the remaining(N-T), values equal to 0), the measurement matrix A $\in; KxN$ and e$\in N(0, \sigma)$ is the normally distributed noise vector.
We consider two different measurement types:

- A is a Gaussian random matrix with size K x N. We obtain A(K x N) by first generating a Gaussian random matrix A(N x N) and then make it orthonormal by $A = randn(orth(A))$. After that, we can randomly subsample K from the N rows. We then generate the signal according to the equation (5). Due to the transformation of the matrix A, the adjoint (inverse) of matrix A is its transpose matrix A^T.
- A is a subsampled Welsh-Hadmard transform. In this case, we obtain the signal y by multiplying the matrix S of size K x N and WH transform matrix (x). Due to the transformation of the matrix A, the adjoint (inverse) of matrix A is:$At = @(x)ifwhyt(S' * x)$.

ISTA and FISTA:

The regularisation term λ is used to control the weight of L1 in the objective function. we use the equation $= max[A^T y]$ to scale the regularization term. I have tried several values for α (0.001, 0.01, 0.1 and 1). The obtained results of MSE are 0.0101, 0.0071, 0.0007 and 0.0122 repectivel. Therefore, $\alpha = 0.1$ is chosen for this case.

1. Lipschitz constant =1 as it is an compressing sensing problem.
2. Proximal operator

$$\begin{aligned} f(x) &= ||x||_1 \\ prox_{\lambda f}(v) &= S_\lambda(v) \end{aligned} \tag{6}$$

where, the elementwise soft thresholding:

$$S_\sigma(x) = \begin{cases} x - \sigma \; x > \sigma \\ 0 \; x \in [-\sigma, \sigma] \\ x + \sigma x < -\sigma \end{cases} \tag{7}$$

In the code, the function called softThresh is used for the proximal operator purpose.
And then I plot the histogram of the output of the optimal x^* to show that most entries. I realized that the absolute values of entries are quite small. Some of them are even smaller than 0.1. Therefore, I manually change those values (greater than 0.1) to 1, those values (smaller than -0.1) to -1 and the remaining value to 0. In the end, it turns out 50 entries having value 1 and the other 50 entries having -1. The remaining entries has value 0. After that, I can compare the obtained result with the original sparse signal x. According to the results, the reconstruction are very successful.

ADMM

for the initialisation in ADMM, I have set Ex+Fz = b, where x and z are actually the same vector. The function handler can be shown as:

$$\begin{aligned} E(x) &= @(x)x \\ F(x) &= @(x) \\ &\rightarrow b = 0 \end{aligned} \tag{8}$$

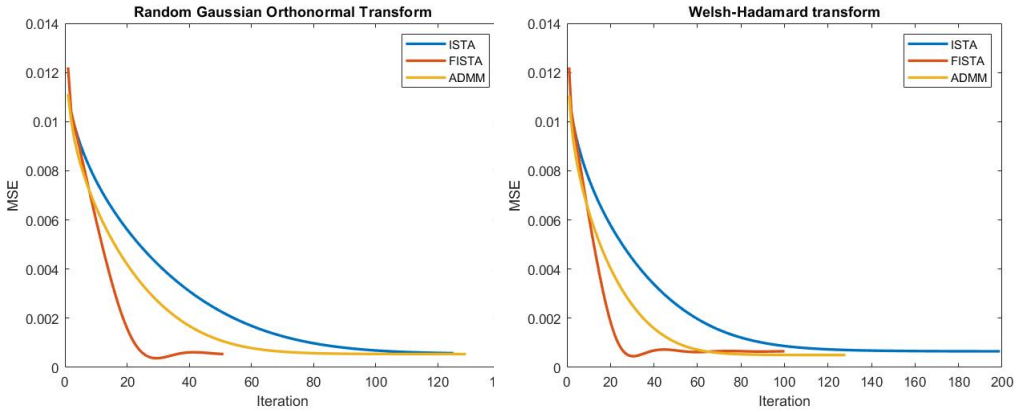
The shrinkage operator can be shown as:

$$proxy@(x, rho)softThresh(x, lambda * rho); \tag{9}$$

The parameter λ is used as the same value using for ISTA and FISTA. ($\lambda = 0.1xmax[A^T y]$). `stopTolerance` $= 10^{-6}$; `maxIter` $= 200; \rho = 1; \mu = 5;$ `overRelaxPara` $= 1.5$. With having these parameters, the optimal x^* can be obtained by running ADMM.

Result

In the figure 1, it shows mean square error for three different methods (ISTA, FISTA and ADMM) with two different measurement types (Gaussian random matrix and Welsh-Hadamard transform). As you can see in the both figures, FISTA has quicker convergence rate than the other two does. ADMM can converge to the steady state faster than ISTA does but it takes more iterations to reach the final iteration. I guess ADMM can converge with a lot less iterations if I can set better parameters. Conclusively, FISTA works the best for calculating the optimal convergence rate for smooth functions.



(a) Mean Square Error using Gaussian random matrix

(b) Mean Square Error using Welsh-Hadamard transform

Figure 1: Mean Square Error with two different measurements types

3 Exercise5

Consider the constraint optimization problem. The objective function is:

$$f(x,y) = (x-a)^2 + \frac{1}{2}(y-b)^2 - 1 \tag{10}$$

with its constraint function:

$$x^2 + y^2 = 2 \tag{11}$$

where, a=1, b=1.5

I implemented a simple version of the quadratic penalty and augmented Lagrangian methods to solve for this problem.

- A line search method with a Newton direction and backtracking line search are used to solve the unconstraint problem at each step for both methods
- And I used $||x_k - x_{k-1}|| < \epsilon$ as a stopping criterion for both of methods. The final tolerance is set $\epsilon = 1e-10$.

The Quadratic Penalty Method:
$\mu_o = 1$ is chosen for the initial penalty weight.
The merit function:

$$Q(x;\mu) := f(x) + \frac{\mu}{2}\sum_{i=1}^{p} h_i^2(x) \tag{12}$$

where, $\mu > 0$ is the penalty parameter.

Framework: for a sequence $\{\mu_k\}$:$\mu_k \to \infty$ as k $\to \infty$, increasingly penalising the constraint compute the (approximate, $||\nabla_x Q(x_k;\mu_k)|| \leq \tau_k, tau \to 0$ sequence $\{x_k\} \to x^*$ of minimisers $x_k of Q(x;\mu_k)$.

The gradient of the merit function can be expressed as:

$$\nabla_x Q(x_k;\mu_k) = \nabla f(x_k) + \sum_{i=1}^{p} \mu_k h_i(x_k) \nabla h_i(x_k) \tag{13}$$

However, it might be ill-conditioning of Hessian. The hessian of the merit function can be expressed as:

$$\nabla^2_{xx} Q(x;\mu_k) p_n = -\nabla^2_x f(x) + \sum_{i=1}^{p} \mu_k h_i(x) \nabla^2 h_i(x) + \mu_k \nabla h(x) \nabla h(x)^T \tag{14}$$

if x is sufficiently close to the minimiser of$Q(.;\mu_k)$

$$\nabla^2_{xx} Q(x;\mu_k) \approx \nabla^2_{xx} \mathcal{L}(x;\nu^*) + \mu_k A(x)^T A(x) \tag{15}$$

AS $\mu_k \to \infty$ the Hessian is dominated by the second term (with eigenvalues 0 and $\mathcal{O}(\mu_k)$) and hence increasely ill-conditioned. There is alternative formulation used to avoid ill-conditioning, $\xi = \mu_k A(x) p_n$, but it is not part of the question. It is because we ignore the ill-conditioning of the Hessian in our case.

The Augmented Lagrangian Method:
$\mu = 10 and \nu_0 = 1$ is chosen as a fixed penalty weight and initial Lagrange mutiplier.
The merit function:

$$\mathcal{L}_{\mathcal{A}}(x,\nu,\mu) := f(x) + \sum_{i=1}^{p} \nu_i h_i(x) + \frac{\mu}{2} \sum_{i=1}^{p} h_i^2(x) \tag{16}$$

The gradient of the merit function can be expressed as:

$$\nabla_x \mathcal{L}_{\mathcal{A}}(x_k,\nu^k;\mu_k) = \nabla f(x_k) + \sum_{i=1}^{p} [\nu_i^k + \mu_k h_i(x_k)] \nabla h_i(x_k) \tag{17}$$

The optimality condition for the Lagrangian of (COP:E):

$$0 \approx \nabla_x \mathcal{L}(x_k,\nu^*) = \nabla f(x_k) + \sum_{i=1}^{p} \nu^* \nabla h_i(x_k) \tag{18}$$

where, comparison yields (an update scheme for ν):

$$\nu^* \approx \nu_i^k + \mu_k h_i(x_k),\ i = 1, ...p \tag{19}$$

The hessian of the merit function can be expressed as:

$$\nabla^2_{xx} \mathcal{L}_{\mathcal{A}}(x_k,\nu^k;\mu_k) = \nabla^2 f(x_k) + \sum_{i=1}^{p} [\nu^* \nabla^2 h_i(x_k)] + \mu \nabla h_i(x_k) . \nabla h_i(x_k)^T \tag{20}$$

The results:

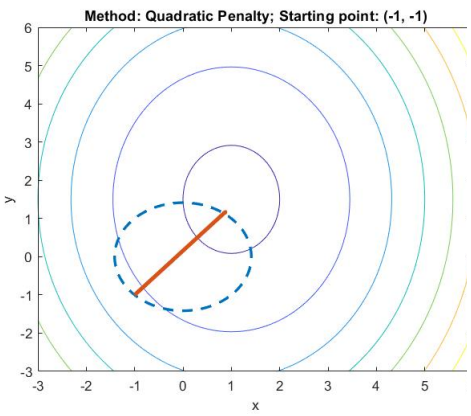
The figure2 and 3 shows the contour graphs with a constraint function and the rate of convergence graphs with a feasible starting point (-1,-1) for both methods. The parameters set for the Quadratic Penalty method are: $\mu_0 = 1, t = 2, \texttt{tol} = 10^{-10}, \texttt{maxIter} = 1000$ where t is the rate of increase for μ_k s.t. $\mu_{k+1} = t\mu_k$ for all $k = 0, 1, 2, ...$ For the rate of convergence graphs, it can be quantified as $\frac{||x_k - x^*||_2}{||x_{k-1} - x^*||_2}$ against iteration k. As you can see, it shows a linear convergence in the figure2b with a 0.5 convergence rate.
The parameter set for the Augmented Lagrangian method are: $\mu = 10, \nu_0 = 1, \texttt{tol} = 10^{-10}, \texttt{maxIter} = 1000$. Here μ is kept fixed rather than being updated.

The figure 4 and 5 shows the contour graphs with a constraint function and the rate of convergence graphs with a feasible starting point (2,3) for both methods. The parameters set for both methods are the same as shown before. The rate of convergence is quantified in the same way too.

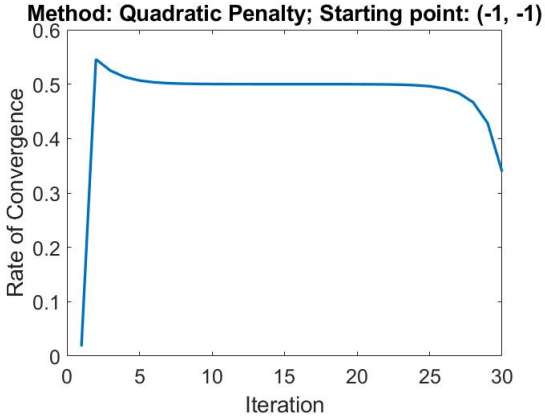
In conclusion, Two methods have quite different graphs of the rate of convergence. The Augmented Lagrangian takes much shorter iterations to converge than the Quadratic Penalty does. The reason might be that μ is fixed and ν_k is updated with h(x,y) in Augmented Lagrangian methods. However, the μ is updated brutally with$\mu = 1.2 * \mu$.

3.1 Code for Quadratic Penalty Method

```
function [xMin, fMin, t, nIter, infoQP] = Quadratic_Penalty(F, H, x0, mu, t, tol
    , maxIter)
% INTERIORPOINT_BARRIER function to minimise a quadratic form with constraints
% [xMin, fMin, nIter, info] = descentLineSearch(F, descent, ls, alpha0, x0, tol,
    maxIter)
```

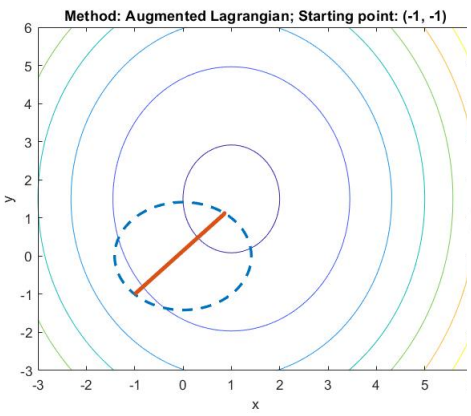


(a) contour graph with constraint function

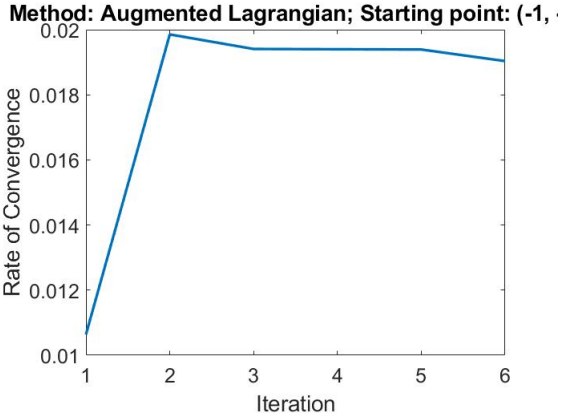


(b) rate of convergence

Figure 2: Quadratic Penalty with starting point (-1,-1)



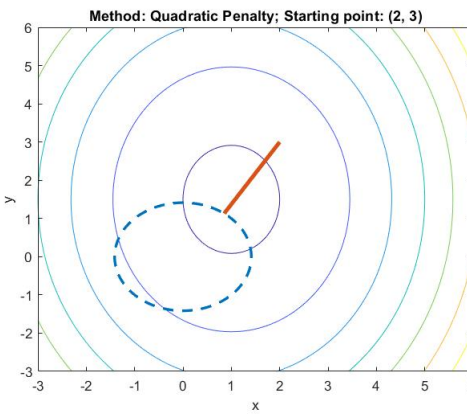
(a) contour graph with constraint function



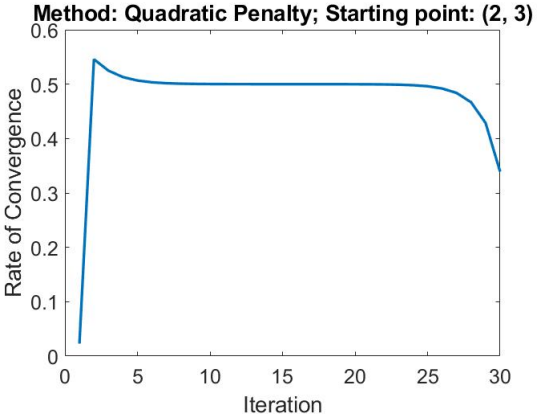
(b) rate of convergence

Figure 3: Quadratic Penalty with starting point (-1,-1)

```
%
% INPUTS
% F: structure with fields
%  - f: function to minimise
%  - df: gradient of function
%  - d2f: Hessian of function
% x0: initial iterate
% mu: initial mu
% t: increasing factor for mu
% tol: tolerance on two consecutive solutions x_t and x_{t+1}
% maxIter: maximum number of iterations
%
% OUTPUTS
% xMin, fMin: minimum and value of f at the minimum
% nIter: number of iterations
% infoBarrier: structure with information about the iteration
```

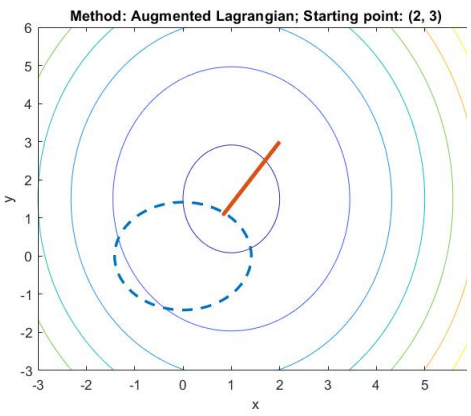


(a) contour graph with constraint function

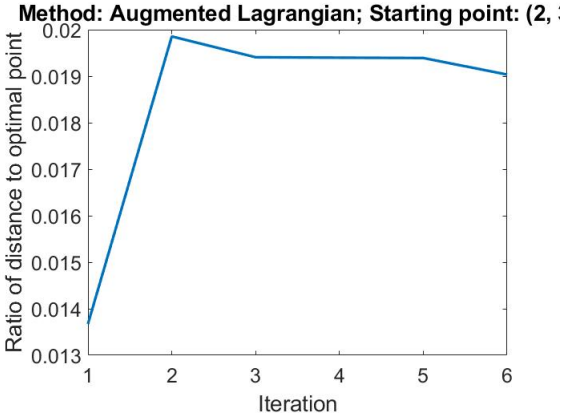


(b) rate of convergence

Figure 4: Quadratic Penalty with starting point (2,3)



(a) contour graph with constraint function



(b) rate of convergence

Figure 5: Augmented Lagrangian with starting point (2,3)

```
%   - xs: iterate history for x

% Initilize
nIter = 0;stopCond = false;x_k = x0;infoQP.xs = x_k;infoQP.fs = F.f(x_k);
% Parameters for centering step
alpha0 = 1; opts.c1 = 1e-4;opts.c2 = 0.9;opts.rho = 0.5;tolNewton = 1e-12;
    maxIterNewton = 100;
% Loop
while (~stopCond && nIter < maxIter)
    % Create function handler for Q (needs to be redifined at each step because
        of changing "t")
    G.f = @(x) F.f(x) + mu / 2 * (H.f(x))^2;
    G.df = @(x) F.df(x) + mu .* H.f(x) .* H.df(x);
    G.d2f = @(x) F.d2f(x) + mu * H.f(x) .* H.d2f(x) + mu * H.df(x) * (H.df(x))';
    lsFun = @(x_k, p_k, alpha0) backtracking(G, x_k, p_k, alpha0, opts);
    x_k_1 = x_k;
```

```
    [x_k, f_k, nIterLS, infoIter] = descentLineSearch(G, 'newton', lsFun, alpha0
        , x_k, tolNewton, maxIterNewton);
    % Check stopping condition
    if norm(x_k - x_k_1) < tol; stopCond = true; end
    % Increase mu
    mu = mu*t;
    % Store info
    infoQP.xs = [infoQP.xs x_k];
    infoQP.fs = [infoQP.fs f_k];
    % Increment number of iterations
    nIter = nIter + 1;
end
% Assign values
xMin = x_k;
fMin = F.f(x_k);
```

3.2 Code for Augmented Lagrangian Method

```
function [xMin, fMin, nIter, infoAL] = Augmented_Lagrangian(F, H, x0, mu, v0,
    tol, maxIter)
% INTERIORPOINT_BARRIER function to minimise a quadratic form with constraints
% [xMin, fMin, nIter, info] = descentLineSearch(F, descent, ls, alpha0, x0, tol,
    maxIter)
%
% INPUTS
% F: structure with fields
%  - f: function to minimise
%  - df: gradient of function
%  - d2f: Hessian of function
% x0: initial iterate
% mu: initial mu
% t: increasing factor for mu
% tol: tolerance on two consecutive solutions x_t and x_{t+1}
% maxIter: maximum number of iterations
%
% OUTPUTS
% xMin, fMin: minimum and value of f at the minimum
% nIter: number of iterations
% infoBarrier: structure with information about the iteration
%   - xs: iterate history for x
% Initilize
nIter = 0;stopCond = false;x_k = x0;infoAL.xs = x_k;infoAL.fs = F.f(x_k);
%luke wrote
v_k=v0;
% Parameters for centering step
alpha0 = 1;
opts.c1 = 1e-4;
opts.c2 = 0.9;
opts.rho = 0.5;
tolNewton = 1e-12;
maxIterNewton = 100;
% Loop
while (~stopCond && nIter < maxIter)
    disp(strcat('Iteration ', int2str(nIter)));
```

```
    % Create function handler for Q (needs to be redifined at each step because
       of changing "t")
    G.f = @(x) F.f(x) + v_k.*H.f(x) + (mu / 2) * (H.f(x))^2;
    G.df = @(x) F.df(x) + (v_k+ mu.*H.f(x)) .* H.df(x) ;
    G.d2f = @(x) F.d2f(x) + (v_k+ mu.*H.f(x)) .* H.d2f(x) + mu * H.df(x) * (H.df
       (x))';
    lsFun = @(x_k, p_k, alpha0) backtracking(G, x_k, p_k, alpha0, opts);
    x_k_1 = x_k;
    [x_k, f_k, nIterLS, infoIter] = descentLineSearch(G, 'newton', lsFun, alpha0
       , x_k, tolNewton, maxIterNewton);
    % Increase v_k
    v_k = v_k + mu*H.f(x_k);
    % Check stopping condition
    if norm(x_k - x_k_1) < tol; stopCond = true; end
    % Store info
    infoAL.xs = [infoAL.xs x_k];
    infoAL.fs = [infoAL.fs f_k];

    % Increment number of iterations
    nIter = nIter + 1;
end
% Assign values
xMin = x_k;
fMin = F.f(x_k);
```