LP-QP

Abouhussein, Harris, El Mistiri

January 16, 2024

Contents

1 LP Problems

The team coded the feasible and the infeasible primal-dual interior point algorithms to solve two problems: the example problem from the excerpt of the Nash & Sofer book and the production planning problem in file prodplan.pdf.

1.1 Nash Problem

The example problem can be presented as follows:

$$\min_{x\in\mathbb{R}^2} z = -x_1 - 2x_2 \qquad \text{s.t.} \qquad \begin{cases} -2x_1 + x_2 & \leq & 2, \\ -x_1 + 2x_2 & \leq & 7, \\ x_1 + 2x_2 & \leq & 3, \\ x_1, x_2 & \geq & 0. \end{cases}$$

If we introduce slack variables x_3, x_4, x_5 then the problem in standard format is presented as follows:

$$\min_{x\in\mathbb{R}^2} z = -x_1 - 2x_2 \qquad \text{s.t.} \qquad \begin{cases} -2x_1 + x_2 + x_3 & = & 2, \\ -x_1 + 2x_2 + x_4 & = & 7, \\ x_1 + 2x_2 + x_5 & = & 3, \\ x_1, x_2, x_3, x_4, x_5 & \geq & 0. \end{cases}$$

Moreover the dual of this problem can be presented as follows:

$$\min_{x\in\mathbb{R}^2} w = 2y_1 + 7y_2 + 3y_3 \qquad \text{s.t.} \qquad \begin{cases} -2y_1 - y_2 + y_3 + s_1 &= -1, \\ y_1 + 2y_2 + 2y_3 + s_2 &= -2, \\ y_1 + s_3 &= 0, \\ y_2 + s_4 &= 0, \\ y_3 + s_5 &= 0, \\ s_1, s_2, s_3, s_4, s_5, &\geq 0. \end{cases}$$

The team first solves the problem using a feasible algorithm with a strictly feasible starting point defined as follows:

$$x_0 = \begin{pmatrix} 0.5 \\ 0.5 \\ 2.5 \\ 6.5 \\ 1.5 \end{pmatrix}, \qquad y_0 = \begin{pmatrix} -1 \\ -1 \\ -5 \end{pmatrix}, \qquad s_0 = \begin{pmatrix} 1 \\ 11 \\ 1 \\ 1 \\ 5 \end{pmatrix} \tag{1}$$

Running the feasible algorithm with this above starting point produces the following results:

```
Results_x0_Feasible =

  10x6 table

    k       z          w            xs          xs_min_nmu       mu
    -    -------    -------    ----------    -----------    ------

    0       -1.5        -24          22.5          -27.5       NaN
    1         -3    -40.681        37.681        -12.319        10
    2     -2.898    -4.1745        1.2766        -3.7234         1
    3    -2.9097    -3.2108        0.3011        -0.1989       0.1
    4    -2.9927     -3.022      0.029248      -0.020752      0.01
    5    -2.9997    -3.0019     0.0021736     -0.0028264     0.001
    6    -2.9999    -3.0002    0.00030785    -0.00019215    0.0001
    7         -3         -3    2.1608e-05    -2.8392e-05     1e-05
    8         -3         -3    3.1838e-06    -1.8162e-06     1e-06
    9         -3         -3    2.1483e-07    -2.8517e-07     1e-07
```

where k, z, w, xs, xs_min_nmu, and mu represent the iteration count, solution of the primal problem, solution of the dual problem, duality gap, complimentary slackness residual, and barrier parameter μ respectively. Moreover, we have

$$x^* = \begin{pmatrix} 2.1794 \\ 0.4103 \\ 0.9486 \\ 8.3588 \\ 0.0000 \end{pmatrix}, \qquad y^* = \begin{pmatrix} 0.0000 \\ 0.0000 \\ -1.0000 \end{pmatrix}, \qquad s^* = \begin{pmatrix} 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 1.0000 \end{pmatrix}.$$

We see that our results are consistent with the table of results found in the `nash_ipm.pdf` file. This gives us confidence that the algorithm is working as intended. The team then solved the problem with the same feasible algorithm but with an infeasible starting point defined as follows:

$$x_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \qquad y_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \qquad s_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{2}$$

Running the feasible algorithm with this above starting point produces the following results:

```
Results_x1_Feasible =

  10x6 table

    k       z          w            xs          xs_min_nmu       mu
    -    -------    ------    ----------    -----------    ------

    0         -3        12             5            -45       NaN
    1         -2    6.0001            20            -30        10
    2    -1.8414     15.04        9.3573         4.3573         1
    3    -3.5081    13.471        2.4591         1.9591       0.1
    4    -3.4347     10.39       0.36796        0.31796      0.01
    5    -3.3269    10.811       0.10413       0.099128     0.001
    6    -3.3333    11.001    0.00054404     4.4045e-05    0.0001
    7    -3.3333        11     3.145e-05     -1.855e-05     1e-05
    8    -3.3333        11    2.1716e-06    -2.8284e-06     1e-06
    9    -3.3333        11    3.3397e-07    -1.6603e-07     1e-07
```

As expected, the algorithm does not converge to a meaningful solution because we did not begin at a feasible starting point. We see that the primal solution does not match the dual solution. The feasible algorithm can be found in `PDM_f.m`.

`PDM_f.m`

```
function [x,y,s,Z,W,k,T,G,R,M] = PDM_f(A,b,c,x0,y0,s0,mu0,theta,tol,N)
% Uses the Feasible Primal-Dual Method to solve
%           min z=c'*x  s.t.  A*x=b     (Primal Problem)
%           max w=b'*y  s.t.  A'*y+s=c  (Dual Problem)
% INPUT:
%       A = Matrix             = Constraint Coefficient Matrix
%       b = Column Vector      = Constraint Constant Vector
%       c = Column Vector      = Cost Vector
%      x0 = Column Vector      = Initial Guess for Primal Variables
%      y0 = Colume Vector      = Initial Guess for Dual Variables
%      s0 = Column Vector      = Initial Guess for Slack Vector
%     mu0 = Positive Real      = Initial Barrier Parameter
%   theta = Real: 0<theta<1    = Barrier Reduction Parameter
%     tol = Positive Real      = Duality Gap Error Tolerance
%       N = Positive Integer = Maximum Number of Iterations
```

```
% OUTPUT:
%      x = Column Vector = Primal Minimizer
%      y = Column Vector = Dual Minimizer
%      s = Column Vector = Slack Vector for Dual Problem
%      Z = Column Vector = Solution History for Primal Problem
%      W = Column Vector = Solution History for Dual Problem
%      k = Integer       = Number of Iterations
%      T = Positive Real = Computation Time
%      G = Column Vector = Duality Gap History
%      R = Column Vector = Complimentary Slackness Residual History
%      M = Column Vector = Barrier Parameter History
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    tic;                % Start Algorithm Timer
    n = length(x0);     % Problem Size
    x = x0;             % Initial Primal Variables
    y = y0;             % Initial Dual Variables
    s = s0;             % Initial Slack Vector
    m = mu0;            % Initial Barrier Parameter
    e = ones(size(x0)); % Vector with all elements equal to one
    [Z,W,G,R,M] = deal(zeros(N,1)); % Allocate Memory
    Z(1) = c'*x;                    % Store Initial Primal Solution
    W(1) = b'*y;                    % Store Initial Dual Solution
    G(1) = x'*s;                    % Store Initial Duality Gap
    R(1) = G(1)-n*m;                % Store Initial C.S. Residual
    M(1) = NaN;                     % Store Initial Barrier Parameter
    k=1;                            % In Case G(1) <= tol
    if G(1) > tol                   % If duality gap is large enough
      for k = 1:N                     % Then execute the method
        % Compute Directions
        X  = spdiags(x,0,n,n);    % Diagonal matrix w/ jth diag term = x_j
        S  = spdiags(s,0,n,n);    % Diagonal matrix w/ jth diag term = s_j
        Si = spdiags(1./s,0,n,n); % Calculate Inverse of S
        D = Si*X;                 % Diag Matrix w/ jth diag term = x_j/s_j
        v = m*e-X*S*e;            % Complementary Slackness Residual
        dy = -(A*D*A')\(A*Si*v);  % Compute Direction of y
        ds = -A'*dy;              % Compute Direction of s
        dx = Si*v-D*ds;           % Compute Direction of x

        % Update Estimates
        ix = find(dx<0);               % Indices of negative x components
        alpha_p = min(-x(ix)./dx(ix)); % Calculate Step Length for Primal
        is = find(ds<0);               % Indices of negative s components
        alpha_s = min(-s(is)./ds(is)); % Calculate Step Length for Dual
        alpha_max = min(alpha_p,alpha_s); % Take Maximum Alpha
        alpha = 0.99999*alpha_max;        % Stay below threshold
        if isempty(alpha)                 % If there were no negative x,s
            alpha = 1;                      % Set alpha to 1
        end
        x = x + alpha*dx;    % Update x Variables
        y = y + alpha*dy;    % Update y Variables
        s = s + alpha*ds;    % Update Slack Vector

        % Calculate Residuals
        Z(k+1) = c'*x;       % Store Primal Solution
```

```
        W(k+1) = b'*y;        % Store Dual Solution
        G(k+1) = x'*s;        % Store Duality Gap
        R(k+1) = G(k+1)-n*m;  % Store C.S. Residual
        M(k+1) = m;           % Store Barrier Parameter
        if G(k+1) <= tol      % If Solution within Tolerance
            break;             % Exit the Loop
        end
        m = theta*m;          % Reduce Barrier Parameter by a Factor of theta
      end
    end
    Z = Z(1:k+1); % Cut off Unused Elements of Primal Solution History
    W = W(1:k+1); % Cut off Unused Elements of Dual Solution History
    G = G(1:k+1); % Cut off Unused Elements of Duality Gap History
    R = R(1:k+1); % Cut off Unused Elements of C.S. Residual History
    M = M(1:k+1); % Cut off Unused Elements of Barrier Parameter History
    T = toc;      % Record Algorithm Time
end
```

The team then solved the problem using an infeasible algorithm with the same strictly feasible starting point (1). The results are presented below:

```
Results_x0_Infeasible =

  10x6 table

    k       z          w           xs          xs_min_nmu       mu
    -    -------    -------    ----------    -----------    ------

    0       -1.5        -24          22.5          -27.5       NaN
    1         -3    -40.681        37.681        -12.319        10
    2     -2.898    -4.1745        1.2766        -3.7234         1
    3    -2.9097    -3.2108        0.3011        -0.1989       0.1
    4    -2.9927     -3.022      0.029248      -0.020752      0.01
    5    -2.9997    -3.0019     0.0021736     -0.0028264     0.001
    6    -2.9999    -3.0002    0.00030785    -0.00019215    0.0001
    7         -3         -3    2.1608e-05    -2.8392e-05     1e-05
    8         -3         -3    3.1838e-06    -1.8162e-06     1e-06
    9         -3         -3    2.1483e-07    -2.8517e-07     1e-07
```

The algorithm converged to the same solution obtained by the feasible algorithm as expected. The team then solved the problem using the infeasible algorithm with the infeasible starting point (2). The results are presented below:

```
Results_x1_Infeasible =

  10x6 table

    k       z          w           xs          xs_min_nmu       mu
    -    -------    -------    ----------    -----------    ------
```

0	-3	12	5	-45	NaN
1	-2.2222	3.3334	15.926	-34.074	10
2	-2.1693	-8.549	5.5732	0.57325	1
3	-2.4822	-3.5103	0.95662	0.45662	0.1
4	-2.936	-3.0327	0.093384	0.043384	0.01
5	-3	-3.003	0.0030542	-0.0019458	0.001
6	-2.9999	-3.0001	0.0002071	-0.0002929	0.0001
7	-3	-3	3.4438e-05	-1.5562e-05	1e-05
8	-3	-3	1.9055e-06	-3.0945e-06	1e-06
9	-3	-3	3.5983e-07	-1.4017e-07	1e-07

This time we see that the algorithm converges to the optimal solution despite starting from an infeasible starting point. This gives us confidence going forward that the algorithm is working as intended. The infeasible algorithm can be found in `PDM_i.m`.

`PDM_i.m`

```
function [x,y,s,Z,W,k,T,G,R,M] = PDM_i(A,b,c,x0,y0,s0,mu0,theta,tol,N)
% Uses the Infeasible Primal-Dual Method to solve
%          min z=c'*x  s.t.  A*x=b     (Primal Problem)
%          max w=b'*y  s.t.  A'*y+s=c  (Dual Problem)
% INPUT:
%       A = Matrix              = Constraint Coefficient Matrix
%       b = Column Vector       = Constraint Constant Vector
%       c = Column Vector       = Cost Vector
%      x0 = Column Vector       = Initial Guess for Primal Variables
%      y0 = Colume Vector       = Initial Guess for Dual Variables
%      s0 = Column Vector       = Initial Guess for Slack Vector
%     mu0 = Positive Real       = Initial Barrier Parameter
%   theta = Real: 0<theta<1     = Barrier Reduction Parameter
%     tol = Positive Real       = Duality Gap Error Tolerance
%       N = Positive Integer = Maximum Number of Iterations
% OUTPUT:
%       x = Column Vector = Primal Minimizer
%       y = Column Vector = Dual Minimizer
%       s = Column Vector = Slack Vector for Dual Problem
%       Z = Column Vector = Solution History for Primal Problem
%       W = Column Vector = Solution History for Dual Problem
%       k = Integer       = Number of Iterations
%       T = Positive Real = Computation Time
%       G = Column Vector = Duality Gap History
%       R = Column Vector = Complimentary Slackness Residual History
%       M = Column Vector = Barrier Parameter History
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    tic;                   % Start Algorithm Timer
    n = length(x0);        % Problem Size
    x = x0;                % Initial Primal Variables
    y = y0;                % Initial Dual Variables
    s = s0;                % Initial Slack Vector
    m = mu0;               % Initial Barrier Parameter
    e = ones(size(x0)); % Vector with all elements equal to one
```

```
    [Z,W,G,R,M] = deal(zeros(N,1)); % Allocate Memory
    Z(1) = c'*x;                    % Store Initial Primal Solution
    W(1) = b'*y;                    % Store Initial Dual Solution
    G(1) = x'*s;                    % Store Initial Duality Gap
    R(1) = G(1)-n*m;                % Store Initial C.S. Residual
    M(1) = NaN;                     % Store Initial Barrier Parameter
    k=1;                            % In Case G(1) <= tol
    if G(1) > tol                   % If duality gap is large enough
      for k = 1:N                     % Then execute the method
        % Compute Directions
        X  = spdiags(x,0,n,n);    % Diagonal matrix w/ jth diag term = x_j
        S  = spdiags(s,0,n,n);    % Diagonal matrix w/ jth diag term = s_j
        Si = spdiags(1./s,0,n,n); % Calculate Inverse of S
        D = Si*X;                 % Diag Matrix w/ jth diag term = x_j/s_j
        v = m*e-X*S*e;            % Complementary Slackness Residual
        rp = b-A*x;               % Primal Constraint Residual
        rd = c-A'*y-s;            % Dual Constraint Residual
        dy = -(A*D*A')\(A*Si*v-A*D*rd-rp); % Compute Direction of y
        ds = -A'*dy+rd;                    % Compute Direction of s
        dx = Si*v-D*ds;                    % Compute Direction of x

        % Update Estimates
        ix = find(dx<0);               % Indices of negative x components
        alpha_p = min(-x(ix)./dx(ix)); % Calculate Step Length for Primal
        is = find(ds<0);               % Indices of negative s components
        alpha_s = min(-s(is)./ds(is)); % Calculate Step Length for Dual
        alpha_max = min(alpha_p,alpha_s); % Take Maximum Alpha
        alpha = 0.99999*alpha_max;        % Stay below threshold
        if isempty(alpha)                 % If there were no negative x,s
            alpha = 1;                      % Set alpha to 1
        end
        x = x + alpha*dx;    % Update x Variables
        y = y + alpha*dy;    % Update y Variables
        s = s + alpha*ds;    % Update Slack Vector

        % Calculate Residuals
        Z(k+1) = c'*x;       % Store Primal Solution
        W(k+1) = b'*y;       % Store Dual Solution
        G(k+1) = x'*s;       % Store Duality Gap
        R(k+1) = G(k+1)-n*m; % Store C.S. Residual
        M(k+1) = m;          % Store Barrier Parameter
        if G(k+1) <= tol     % If Solution within Tolerance
            break;             % Exit the Loop
        end
        m = theta*m;         % Reduce Barrier Parameter by a Factor of theta
      end
    end
    Z = Z(1:k+1); % Cut off Unused Elements of Primal Solution History
    W = W(1:k+1); % Cut off Unused Elements of Dual Solution History
    G = G(1:k+1); % Cut off Unused Elements of Duality Gap History
    R = R(1:k+1); % Cut off Unused Elements of C.S. Residual History
    M = M(1:k+1); % Cut off Unused Elements of Barrier Parameter History
    T = toc;      % Record Algorithm Time
end
```

A performance summary of the two algorithms applied with the different initial guesses is given below.

```
Comparison =

  4x4 table

                           Sol_Prim    Sol_Dual    Iterations       Time
                           --------    --------    ----------    ---------

    Feasible with x0             -3          -3            9           0.0929
    Feasible with x1        -3.3333          11            9        0.0071356
    Infeasible with x0           -3          -3            9         0.014815
    Infeasible with x1           -3          -3            9           0.0929
```

We see that the algorithms perform at roughly the same level. It takes 9 iterations for all the algorithms to converge. Also, the computation times are very similar. All algorithms converge to the same solution except the feasible algorithm with the infeasible starting point which is expected. Finally, the driver code which sets up the problem and calls the solvers can be found in `LP_Nash.m`.

`LP_Nash.m`

```
clear; close all; clc;

% Problem Parameters:
c = [-1; -2; 0; 0; 0]; % Cost Vector
A = [-2 1 1 0 0;       % Constraint Coefficient Matrix
     -1 2 0 1 0;
      1 2 0 0 1];
b = [2; 7; 3];         % Constraint Constant Vector

% Method Parameters:
mu0   = 10;   % Initial Barrier Parameter
theta = 1/10; % Barrier Reduction Parameter
tol   = 1e-6; % Duality Gap Tolerance
N     = 10;   % Maximum Number of Iterations

% Initial Guesses:

% From Example:
x0 = [0.5; 0.5; 2.5; 6.5; 1.5];
y0 = [-1; -1; -5];
s0 = [1; 11; 1; 1; 5];
% All Elements Set to One
[x1,s1] = deal(ones(size(x0)));
y1 = ones(size(y0));

%% Solve Problem Using Feasible Primal-Dual Algorithm
clc;
```

```
% Using x0, y0, and s0:
[x0f,y0f,s0f,Z0f,W0f,k0f,T0f,G0f,R0f,M0f] ...
    = PDM_f(A,b,c,x0,y0,s0,mu0,theta,tol,N);

% Using x1, y1, and s1:
[x1f,y1f,s1f,Z1f,W1f,k1f,T1f,G1f,R1f,M1f] ...
    = PDM_f(A,b,c,x1,y1,s1,mu0,theta,tol,N);

%% Solve Problem Using Infeasible Primal-Dual Algorithm
clc;

% Using x0, y0, and s0:
[x0i,y0i,s0i,Z0i,W0i,k0i,T0i,G0i,R0i,M0i] ...
    = PDM_i(A,b,c,x0,y0,s0,mu0,theta,tol,N);

% Using x1, y1, and s1:
[x1i,y1i,s1i,Z1i,W1i,k1i,T1i,G1i,R1i,M1i] ...
    = PDM_i(A,b,c,x1,y1,s1,mu0,theta,tol,N);

%% Display Results
clc;

% Create Iteration Information:
K0f = (0:k0f)'; K1f = (0:k1f)'; K0i = (0:k0i)'; K1i = (0:k1i)';

% Gather Method Information:
Sol_Prim   = [Z0f(end); Z1f(end); Z0i(end); Z1i(end)]; % Primal Solutions
Sol_Dual   = [W0f(end); W1f(end); W0i(end); W1i(end)]; % Dual Solutions
Iterations = [k0f; k1f; k0i; k1i];                     % Iteration Numbers
Time       = [T0f; T1f; T0i; T0f];                     % Computation Times

% Display Results in Tables:
Comparison = table(Sol_Prim,Sol_Dual,Iterations,Time,...
               'RowNames',{'Feasible with x0','Feasible with x1',...
                           'Infeasible with x0','Infeasible with x1'})
Results_x0_Feasible   = table(K0f,Z0f,W0f,G0f,R0f,M0f,...
                        'VariableNames',{'k','z','w','xs',...
                                         'xs_min_nmu','mu'})
Results_x1_Feasible   = table(K1f,Z1f,W1f,G1f,R1f,M1f,...
                        'VariableNames',{'k','z','w','xs',...
                                         'xs_min_nmu','mu'})
Results_x0_Infeasible = table(K0i,Z0i,W0i,G0i,R0i,M0i,...
                        'VariableNames',{'k','z','w','xs',...
                                         'xs_min_nmu','mu'})
Results_x1_Infeasible = table(K1i,Z1i,W1i,G1i,R1i,M1i,...
                        'VariableNames',{'k','z','w','xs',...
                                         'xs_min_nmu','mu'})
```

1.2 prodplan Problem

The Infeasible Primal-Dual Method constructed above in `PDM_i.m` was then utilized to solve a production planning problem as an LP. The problem was presented as part d) in prodplan.pdf, and the production of three different machines (Refrigerators, Stoves, & Dishwashers) is to be scheduled in a way that minimizes total expenses due to backlog and inventory. Each different machine requires a different amount of time for the production of one unit (2 hours for the refrigerators, 4 hours for stoves, and 3 hours for dishwashers), with a total of 15,000 hours of production time available for each quarter. Moreover, there is no inventory of any product at the start of production in the first quarter, and management is requiring that the inventory level of each of the products to be at least 150 by the end of the fourth quarter. Due to plans to change the assembly line tooling for refrigerators, manufacturing of refrigerators in the second quarter is not possible. A backlog charge of $20 per item per quarter is associated with delays in the refrigerators and stoves, and a $10 per item per quarter backlog charge for dishwashers. Also, each item that is left in the inventory at the end of a quarter would cost $5 for holding. The expected sales of each of the products for each quarter are presented in the table below:

Product	Q1	Q2	Q3	Q4
Refrigerators	1500	1000	2000	1200
Stoves	1500	1500	1200	1500
Dishwashers	1000	2000	1500	2500

From the information given above, we construct four constraints that ensure the total production hours for each quarter do not exceed 15,000. Note that the following constraints are implemented in row 1, 5, 9, and 13 of the constraint coefficient matrix A, which is defined on line 12 in the code (seen further below).

Constraints on the Hours of Production:

$$
\begin{array}{lcccccccccc}
A_1 & = & 4s_1 & + & 3d_1 & + & 2r_1 & + & x_1 & = & 15000, \\
A_5 & = & 4s_2 & + & 3d_2 & & & + & x_2 & = & 15000, \\
A_9 & = & 4s_3 & + & 3d_3 & + & 2r_3 & + & x_3 & = & 15000, \\
A_{13} & = & 4s_4 & + & 3d_4 & + & 2r_4 & + & x_4 & = & 15000.
\end{array}
$$

where

$$
\begin{array}{lcl}
s_i & = & \text{The number of stoves produced in quarter i,} \\
d_i & = & \text{The number of dish washers produced in quarter i,} \\
r_i & = & \text{The number of refrigerators produced in quarter i,} \\
x_i & = & \text{Slack Variable.}
\end{array}
$$

We note that in the above production constraints, we are unable to produce refrigerators in the second quarter, so $r_2 = 0$ is fixed which allows us to exclude it from the problem.

Next, we look at the constraints derived from the expected sales for each quarter. Since we are trying to minimize storage costs, it makes sense to produce a number of items equal to the total amount of expected sales for all four quarters, plus the number of products that you want in storage (150 each) at the end of the last quarter. With this in mind, we create the first quarter's sales constraints so that the sales quotas are satisfied exactly. They are implemented in rows 2, 3, and 4 of matrix A.

Constraints on Production Amounts for Quarter 1:

$$\begin{array}{ccccccccc} A_2 & = & s_1 & - & \hat{s}_1 & + & \bar{s}_2 & = & 1500, \\ A_3 & = & d_1 & - & \hat{d}_1 & + & \bar{d}_2 & = & 1000, \\ A_4 & = & r_1 & - & \hat{r}_1 & & & = & 1500. \end{array}$$

where

$$\begin{array}{ccl} \hat{s}_i & = & \text{The number of stoves stored in quarter i,} \\ \bar{s}_i & = & \text{The number of stoves backlogged in quarter i,} \\ \hat{d}_i & = & \text{The number of dish washers stored in quarter i,} \\ \bar{d}_i & = & \text{The number of dish washers backlogged in quarter i,} \\ \hat{r}_i & = & \text{The number of refrigerators stored in quarter i,} \\ \bar{r}_i & = & \text{The number of refrigerators backlogged in quarter i.} \end{array}$$

We note at this time that storing the items created in a quarter decreases the available products in that quarter, while increasing the available products in the next quarter. In contrast, backlogging items increase the number of available products in previous quarter, while decreasing the number of items in the current quarter. So any items that are stored in quarter 1 are subtracted from the total amount, and any items backlogged in quarter 2 are added. However, these actions increase the cost function, and so the goal is to avoid taking them whenever possible. Also, since we are unable to produce refrigerators during quarter 2, the variable $\bar{r}_2 = 0$ can be omitted from the problem.

Next, we show the second quarter's sales constraints. They are implemented in row 6, 7, and 8 of matrix A.

Constraints on Production Amounts for Quarter 2:

$$\begin{array}{ccccccccccccc} A_6 & = & \hat{s}_1 & - & \bar{s}_2 & + & s_2 & - & \hat{s}_2 & + & \bar{s}_3 & = & 1500, \\ A_7 & = & \hat{d}_1 & - & \bar{d}_2 & + & d_2 & - & \hat{d}_2 & + & \bar{d}_3 & = & 2000, \\ A_8 & = & \hat{r}_1 & & & & & & & + & \bar{r}_3 & = & 1000. \end{array}$$

Now, the number of stored products from quarter 1 can also be used to meet the expected number of sales. However, any products created in quarter 2 that are backlogged cannot be used to meet the current sales requirements since they were used to meet the previous quarter's product availability requirements. Also, in addition to $\bar{r} = 0$, the inability to manufacture refrigerators in quarter 2 implies that $r_2 = 0$ and $\hat{r}_2 = 0$, so these variables are also omitted from the problem.

The next sales constraints are for quarter 3, and are implemented in row 10, 11, and 12 of matrix A.

Constraints on Production Amounts for Quarter 3:

$$\begin{array}{ccccccccccccc} A_{10} & = & \hat{s}_2 & - & \bar{s}_3 & + & s_3 & - & \hat{s}_3 & + & \bar{s}_4 & = & 1200, \\ A_{11} & = & \hat{d}_2 & - & \bar{d}_3 & + & d_3 & - & \hat{d}_3 & + & \bar{d}_4 & = & 1500, \\ A_{12} & = & & - & \bar{r}_3 & + & r_3 & - & \hat{r}_3 & + & \bar{r}_4 & = & 2000. \end{array}$$

Here, we omit the variable $\hat{r}_2 = 0$, but the constraints are otherwise straight forward.

Finally, we have the sales constraints for the fourth quarter implemented in rows 14, 15, and 16 of matrix A.

Constraints on Production Amounts for Quarter 4:

$$\begin{array}{ccccccccc} A_{14} & = & \hat{s}_3 & - & \bar{s}_4 & + & s_4 & = & 1650, \\ A_{15} & = & \hat{d}_3 & - & \bar{d}_4 & + & d_4 & = & 2650, \\ A_{16} & = & \hat{r}_3 & - & \bar{r}_4 & + & r_4 & = & 1350. \end{array}$$

Since we want 150 of each product in storage at the end of the fourth quarter, the variables $\hat{s}_4 = 150$, $\hat{d}_4 = 150$, and $\hat{r}_4 = 150$ are fixed. This means we can omit them from the problem as long as we add 150 to each of the components on the right hand sides of the above equations.

The problem statement was translated into a mathematical LP optimization problem in MATLAB, as it can be seen in the `LP_prodplan.m` file below:

`LP_prodplan.m`

```
clear; close all; clc;

% Cost Vector:
%   Selling=$0, Storing=$5, Backlog_d=$10, Backlog_s_r=$20, Slack=$0
c = [0 5 0 5 0 5 20 0 5 10 0 5 20 0 5 10 0 5 20 0 5 20 0 10 0 20 0 0 0 0 0]';

% Constraint Coefficient Matrix:
%   s/d/r => Produced,   - => Backloged,   ^ => Stored
%   | s1 | d1 | r1 |  s2   |  d2   |   s3   |   d3   |   r3   | s4 | d4 | r4 |slack
%   |s  ^ d  ^ r  ^| - s  ^  - d  ^| - s  ^  - d  ^  - r  ^| - s  - d  - r|
%   |              |               |                       |              |
```

```
A = [4   0 3   0 2   0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   0 0   0 0 1 0 0 0;
     1  -1 0   0 0   0   1 0   0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   0 0   0 0 0 0 0 0;
     0   0 1  -1 0   0   0 0   0   1 0   0   0 0   0   0 0   0   0 0   0   0 0   0 0   0 0 0 0 0 0;
     0   0 0   0 1  -1   0 0   0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   0 0   0 0 0 0 0 0;
     0   0 0   0 0   0   0 4   0   0 3   0   0 0   0   0 0   0   0 0   0   0 0   0 0   0 0 0 1 0 0;
     0   1 0   0 0   0  -1 1  -1   0 0   0   1 0   0   0 0   0   0 0   0   0 0   0 0   0 0 0 0 0 0;
     0   0 0   1 0   0   0 0   0  -1 1  -1   0 0   0   1 0   0   0 0   0   0 0   0 0   0 0 0 0 0 0;
     0   0 0   0 0   1   0 0   0   0 0   0   0 0   0   0 0   0   1 0   0   0 0   0 0   0 0 0 0 0 0;
     0   0 0   0 0   0   0 0   0   0 0   0   0 4   0   0 3   0   0 2   0   0 0   0 0   0 0 0 0 1 0;
     0   0 0   0 0   0   0 0   1   0 0   0  -1 1  -1   0 0   0   0 0   0   1 0   0 0   0 0 0 0 0 0;
     0   0 0   0 0   0   0 0   0   0 0   1   0 0   0  -1 1  -1   0 0   0   0 0   1 0   0 0 0 0 0 0;
     0   0 0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   0  -1 1  -1   0 0   0 0   1 0 0 0 0 0;
     0   0 0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   0   0 4   0 3   0 2 0 0 0 1;
     0   0 0   0 0   0   0 0   0   0 0   0   0 0   1   0 0   0   0 0   0  -1 1   0 0   0 0 0 0 0 0;
     0   0 0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   1   0 0   0   0 0  -1 1   0 0 0 0 0 0;
     0   0 0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   0   0 0   1   0 0   0 0  -1 1 0 0 0 0];

% Constrait Constant Vector:
%    hours    s     d     r
b = [15000; 1500; 1000; 1500;  % Quarter 1
     15000; 1500; 2000; 1000;  % Quarter 2
     15000; 1200; 1500; 2000;  % Quarter 3
     15000; 1650; 2650; 1350]; % Quarter 4

% Initial Guesses:
x0 = 500*ones(size(c));
y0 = 500*ones(size(b));
s0 = 500*ones(size(c));

% Other Parameters:
mu0   = 10;   % Initial Barrier Parameter
theta = 1/10; % Barrier Reduction Parameter
tol   = 1e-8; % Duality Gap Tolerance
N     = 100;  % Maximum Number of Iterations

% Solve Problem 3 from prodplan.pdf Using Infeasible Primal-Dual Method:
[x,y,s,Z,W,k,T,G,R,M] = PDM_i(A,b,c,x0,y0,s0,mu0,theta,tol,N);

%% Display Results
clc;

% Create Numbering Information:
K = (0:k)'; % Iterations
Q = (1:4)'; % Quarters

% Extract Product Information:

% Stoves:
%   |B.L. |Prod |Store|
S = [   0  x( 1) x( 2); % Quarter 1
     x( 7) x( 8) x( 9); % Quarter 2
     x(13) x(14) x(15); % Quarter 3
     x(22) x(23)  150]; % Quarter 4

```

```
% Dishwashers:
%   |B.L. |Prod |Store|
D = [   0  x( 3) x( 4); % Quarter 1
      x(10) x(11) x(12); % Quarter 2
      x(16) x(17) x(18); % Quarter 3
      x(24) x(25)  150]; % Quarter 4

% Refrigerators:
%   |B.L. |Prod |Store|
F = [   0  x( 5) x( 6); % Quarter 1
        0     0     0 ; % Quarter 2
      x(19) x(20) x(21); % Quarter 3
      x(26) x(27)  150]; % Quarter 4

% Calculate Production Time:
H = 4*S(:,2)+3*D(:,2)+2*F(:,2); % Production Hours

% Calculate Product Totals:
% Stove
TS(1)   =          -S( 1 ,1)+S( 1 ,2)-S( 1 ,3)+S( 2 ,1);
TS(2:3) = S(1:2,3)-S(2:3,1)+S(2:3,2)-S(2:3,3)+S(3:4,1);
TS(4)   = S( 3 ,3)-S( 4 ,1)+S( 4 ,2)-S( 4 ,3);
% Dishwasher
TD(1)   =          -D( 1 ,1)+D( 1 ,2)-D( 1 ,3)+D( 2 ,1);
TD(2:3) = D(1:2,3)-D(2:3,1)+D(2:3,2)-D(2:3,3)+D(3:4,1);
TD(4)   = D( 3 ,3)-D( 4 ,1)+D( 4 ,2)-D( 4 ,3);
% Refrigerator
TF(1)   =          -F( 1 ,1)+F( 1 ,2)-F( 1 ,3)+F( 2 ,1);
TF(2:3) = F(1:2,3)-F(2:3,1)+F(2:3,2)-F(2:3,3)+F(3:4,1);
TF(4)   = F( 3 ,3)-F( 4 ,1)+F( 4 ,2)-F( 4 ,3);

% Calculate Quota Information:
QS = b(2:4:end); QS(end) = QS(end)-150;
QD = b(3:4:end); QD(end) = QD(end)-150;
QF = b(4:4:end); QF(end) = QF(end)-150;

% Create Tables
Convergence_Results_for_prodplan = table(K,Z,W,G,R,M)
Production_Results               = table(Q,S(:,2),D(:,2),F(:,2),H)
Stove_Production_Results         = table(Q,S(:,1),S(:,2),S(:,3),TS',QS)
Dishwasher_Production_Results    = table(Q,D(:,1),D(:,2),D(:,3),TD',QD)
Refrigerator_Production_Results  = table(Q,F(:,1),F(:,2),F(:,3),TF',QF)
```

As mentioned above, and can be seen in the code, `PDM_i.m` was called to solve the problem on hand, and the obtained performance results are shown below:

Output:

```
Convergence_Results_for_prodplan =

  15x6 table

    K       Z            W             G              R            M
    --    ------    ----------    ----------    ----------    ------

     0     85000    3.9425e+07      7.75e+06    7.7497e+06       NaN
     1     82865    3.6192e+07    4.3991e+06    4.3988e+06        10
     2     79545    2.4415e+07    2.6686e+06    2.6685e+06         1
     3     76154    1.0522e+07    1.0201e+06    1.0201e+06       0.1
     4     70659     1.804e+06    1.9138e+05    1.9138e+05      0.01
     5     52702    4.5374e+05         67662         67662     0.001
     6     36787    2.3326e+05         37330         37330    0.0001
     7     16741         42286        9335.5        9335.5     1e-05
     8     12182         22766        4038.8        4038.8     1e-06
     9    9663.1         12840        1260.5        1260.5     1e-07
    10    8614.6        8889.1         116.6         116.6     1e-08
    11    8508.6        8528.8        8.6542        8.6542     1e-09
    12    8500.1        8500.2      0.050727      0.050727     1e-10
    13      8500          8500    6.1662e-07    6.1631e-07     1e-11
    14      8500          8500    3.1263e-11    2.6329e-13     1e-12
```

The meaning of the column names are as follows:

$$\begin{aligned} K &= \text{Iterations}, \\ Z &= \text{Primal Solution}, \\ W &= \text{Dual Solution}, \\ G &= \text{Duality Gap}, \\ R &= \text{Complementary Slackness Residual}, \\ M &= \text{Barrier Parameter } \mu. \end{aligned}$$

We see that the method finished in 14 iterations with the primal and dual solutions converging to the same value. It takes a time on the order of 10^{-2} seconds for the algorithm to finish the problem, and the results are within the specified error tolerance on the duality gap. According to the results, the minimum storage cost is \$8500. We note that this solution does not include the cost of storing the products at the end of the fourth quarter. So we add \$2250 to the solution to get the total cost:

$$\text{Storage Cost} = \$10750.00$$

Next, the values of the minimizer x are considered. First, we consider the number of items produced during each quarter, and the required amount of time needed for their manufacture:

```
Production_Results =

  4x5 table

    Quarter      s_i       d_i      r_i      Req_Time
    -------     ------     ----     ----     --------

       1          1500     1000     2500      14000
       2        1637.5     2000        0      12550
       3          1625     1500     2000      15000
       4        1087.5     2650     1350      15000
```

There are two major obstacles to overcome in this problem. The first is meeting the expected number of refrigerator sales in the second quarter while not being able to produce the product in the same quarter. The refrigerator sales requirements must be achieved only by using refrigerators that were stored from the first quarter, or by using refrigerators backlogged from the third quarter. We also note that quarter 2 has the lowest expected refrigerator sales, which is probably why the company chose to do the tool modifications during this period.

The second obstacle is that there is not enough available production hours in quarter 4 to meet the product demand. The requirement that 150 of each product be in storage at the end of the fourth quarter adds more difficulty to the issue. Additional items must be produced and stored before quarter 4 in order to achieve the goal.

To overcome the first obstacle, we see that 2500 refrigerators are produced during the first quarter. This is enough to meet the requirements of both the first quarter (1500) and the second quarter (1000), and is the optimal choice because it requires less money to store a refrigerator (\$5) than to backlog it (\$20). The exact number of required stoves and dishwashers is produced to avoid any additional storage costs. The amount of time required to manufacture the items is 14,000 hours, which is well below the available time. The total storage cost for the first quarter is \$5000 (\$5 x 1000 refrigerators), making the company's tool modification project the biggest source of storage expense in the problem.

In the second quarter, we start to prepare for the high product demand of the fourth quarter (2nd obstacle). We see that 137.5 extra stoves are produced in order to start creating a back-stock of items. Stoves are selected as the surplus item in order to maximize the total number of products (of any kind) created in the fourth quarter. Since stoves take longer to make than the other products (4h), more items are created by focusing on dishwashers (3h) and refrigerators (2h) whenever time is the limiting resource. If this is done in the final quarter, then the number of items that must be stored over the course of preceding quarters is minimized, which minimizes the total storage costs. For the other products, the exact number of dishwashers is created so that no storage is required, and there are no refrigerators produced during this period due to tool modifications. As mentioned before, the demand for refrigerators is met by using the refrigerators created in the first quarter. It requires 12,500 production hours to create the items for the second quarter, which is even less than the previous quarter (14,000h), and is below the maximum. The storage cost for this quarter is the lowest of all, totaling to $687.50 ($5 x 137.5 stoves). This brings the grand total to $5687.50 for quarter 1 and 2.

Next, we consider the third quarter. We immediately see that the number of production hours is at the maximum allowed. In fact, both quarter 3 and quarter 4 use the maximum amount of production hours because creating products too early will incur additional storage fees. So it makes sense to manufacture products as close to when you will sell them as possible. Again, the exact number of refrigerators and dishwashers are created to save on storage fees, while a surplus of 562.5 stoves (taken from the stove table below) is created and stored. This incurs a fee of $2812.50, which brings the total expense to $8500 for the first three quarters. Note that this matches the result produced by the algorithm.

Finally, in the fourth quarter, the exact amount of dishwashers and refrigerators are created with the idea that 150 of each will be put into storage. The remaining production hours are used to create as many stoves as possible. These stoves combined with the stoves in storage will satisfy the sales requirements, while leaving 150 stoves to be placed in storage. The cost of storing 450 items is $2250, and when added to the previous quarter's expenses, gives us a final result of $10750.00 in storage costs.

The following tables summarize the variable values for each particular product. We note that any answers that are on the order of 10^{-13} and 10^{-14} can be approximated as zero since they are smaller than the error tolerance. We see that in all cases, the total amount of product sold matches the given quota. Also 150 items are in the fourth quarter's storage column as required.

```
Stove_Production_Results =

  4x6 table

    Quarter     Backlog      Production      Store       Total Product    Quota
    -------    ----------    ----------    ----------    -------------    -----

       1                0         1500     2.0461e-13        1500          1500
       2       5.0752e-14       1637.5          137.5        1500          1500
       3       4.0667e-14         1625          562.5        1200          1200
       4       4.0768e-14       1087.5            150        1500          1500
```

```
Dishwasher_Production_Results =

  4x6 table

    Quarter      Backlog       Production       Store        Total Product     Quota
    -------     ----------     ----------     ----------     -------------     -----

       1                 0        1000        2.0445e-13         1000          1000
       2        1.0154e-13        2000        8.3568e-13         2000          2000
       3        7.3805e-14        1500        8.1708e-13         1500          1500
       4        7.4058e-14        2650               150         2500          2500



Refrigerator_Production_Results =

  4x6 table

    Quarter      Backlog       Production       Store        Total Product     Quota
    -------     ----------     ----------     ----------     -------------     -----

       1                 0        2500              1000         1500          1500
       2                 0           0                 0         1000          1000
       3        5.7669e-14        2000        4.0743e-13         2000          2000
       4        4.5273e-14        1350               150         1200          1200
```

2 QP Problems

In this section, we create an interior-points predictor-corrector algorithm based on algorithm 6.4 in the book and use it to solve various problems. However, the algorithm solves problems with non-negativity inequality constraints rather than general linear inequality constraints. In other words, problems of the form

$$\min_x q(x) = \frac{1}{2}x^T G x + x^T c \qquad \text{s.t.} \qquad x \geq 0,$$

where G is symmetric and positive semidefinite.

2.1 Algorithm Design

Rewriting the KKT Conditions and introducing the slack vector $y = (y_1, y_2, ..., y_n)^T$, we derive the following linear system

$$\begin{bmatrix} G & 0 & -I \\ I & -I & 0 \\ 0 & \Lambda & Y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -r_d \\ -r_p \\ \sigma\mu e - \Lambda Y e \end{bmatrix}. \tag{3}$$

where (with n being the problem size)

$$\begin{aligned} e &= (1, 1, ..., 1)^T \text{ is the n-dimensional vector with all components 1,} \\ \lambda &= (\lambda_1, \lambda_2, ..., \lambda_n)^T \text{ is the vector of Lagrangian multipliers,} \\ I &= \text{diag}(1, 1, ..., 1) \text{ is the } n \times n \text{ identity matrix,} \\ \Lambda &= \text{diag}(\lambda_1, \lambda_2, ..., \lambda_n), \\ Y &= \text{diag}(y_1, y_2, ..., y_n), \\ r_d &= Gx - \lambda + c, \\ r_p &= x - y, \\ \mu &= \frac{y^T \lambda}{n}. \end{aligned}$$

The parameter σ is the centering parameter, and the variables Δx, Δy, and $\Delta\lambda$ are the step directions by which the new iterates

$$x_{k+1} = x_k + \alpha_k \Delta x_k, \qquad y_{k+1} = y_k + \alpha_k \Delta y_k, \qquad \lambda_{k+1} = \lambda_k + \alpha_k \Delta\lambda_k$$

are defined. Here α_k is a step length determined by the algorithm. The algorithm requires us to find the solution (using substitution) for a variation of (3) on three different occasions. In the end, we will be required to solve a linear system for Δx using the conjugate gradient method, then using the result to obtain Δy and $\Delta\lambda$.

On the first occasion, we take an affine scaling step by setting $\sigma = 0$ and solving the third row of (3) for Δy^{aff} which yields

$$\Delta y^{\text{aff}} = -\Lambda^{-1} Y \Delta\lambda^{\text{aff}} - y.$$

Substitute the above into the second row of (3) and rearrange to get the system

$$\begin{bmatrix} G & -I \\ I & \Lambda^{-1}Y \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta \lambda^{\text{aff}} \end{bmatrix} = \begin{bmatrix} -r_d \\ -y - r_p \end{bmatrix}.$$

Finally, we can solve for Δx^{aff} by adding $Y^{-1}\Lambda$ times the second row of the above equality to the first row. This gives us the following equations which are solved/calculated on line 56-58 of the code (below).

$$\begin{aligned} (G + Y^{-1}\Lambda)\Delta x^{\text{aff}} &= -\lambda - Y^{-1}\Lambda r_p - r_d, \\ \Delta y^{\text{aff}} &= \Delta x^{\text{aff}} + r_p, && \text{[From 2nd row of (3)]} \\ \Delta \lambda^{\text{aff}} &= G\Delta x^{\text{aff}} + r_d. && \text{[From 1st row of (3)]} \end{aligned}$$

On the second occasion, we must improve upon the affine step by computing a corrector step. To do this, we solve the system

$$\begin{bmatrix} G & 0 & -I \\ I & -I & 0 \\ 0 & \Lambda & Y \end{bmatrix} \begin{bmatrix} \Delta x^{\text{cor}} \\ \Delta y^{\text{cor}} \\ \Delta \lambda^{\text{cor}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\Lambda^{\text{aff}} Y^{\text{aff}} e \end{bmatrix}, \tag{4}$$

where

$$\Lambda^{\text{aff}} = \text{diag}(\Delta\lambda_1^{\text{aff}}, \Delta\lambda_2^{\text{aff}}, ..., \Delta\lambda_n^{\text{aff}}) \qquad \text{and} \qquad Y^{\text{aff}} = \text{diag}(\Delta y_1^{\text{aff}}, \Delta y_2^{\text{aff}}, ..., \Delta y_n^{\text{aff}}).$$

We follow the same procedure as before by first solving for Δy^{cor} to obtain

$$\Delta y^{\text{cor}} = -\Lambda^{-1}\Lambda^{\text{aff}} Y^{\text{aff}} e - \Lambda^{-1} Y \Delta\lambda^{\text{cor}},$$

then substitute the results into the second row of (4)

$$\begin{bmatrix} G & -I \\ I & \Lambda^{-1}Y \end{bmatrix} \begin{bmatrix} \Delta x^{\text{cor}} \\ \Delta \lambda^{\text{cor}} \end{bmatrix} = \begin{bmatrix} 0 \\ -\Lambda^{-1}\Lambda^{\text{aff}} Y^{\text{aff}} e \end{bmatrix},$$

and solving for Δx^{cor} by adding $Y^{-1}\Lambda$ times the second row to the first row, which results in following equations which are solved/calculated on line 59-61 of the code.

$$\begin{aligned} (G + Y^{-1}\Lambda)\Delta x^{\text{cor}} &= -Y^{-1}\Lambda^{\text{aff}} Y^{\text{aff}} e, \\ \Delta y^{\text{cor}} &= \Delta x^{\text{cor}}, \\ \Delta \lambda^{\text{cor}} &= G\Delta x^{\text{cor}}. \end{aligned}$$

On the final occasion, we get the true iterate step directions by solving the following system

$$\begin{bmatrix} G & 0 & -I \\ I & -I & 0 \\ 0 & \Lambda & Y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -r_d \\ -r_p \\ \sigma\mu e - \Lambda Y e - \Lambda^{\text{cor}} Y^{\text{cor}} e \end{bmatrix}, \tag{5}$$

with

$$\Lambda^{\text{cor}} = \text{diag}(\Delta\lambda_1^{\text{cor}}, \Delta\lambda_2^{\text{cor}}, ..., \Delta\lambda_n^{\text{cor}}) \qquad \text{and} \qquad Y^{\text{cor}} = \text{diag}(\Delta y_1^{\text{cor}}, \Delta y_2^{\text{cor}}, ..., \Delta y_n^{\text{cor}}).$$

Once again following the same procedure, we have

$$\Delta y = \sigma\mu\Lambda^{-1}e - y - \Lambda^{-1}\Lambda^{\text{cor}}Y^{\text{cor}}e - \Lambda^{-1}Y\Delta\lambda,$$

which implies

$$\begin{bmatrix} G & -I \\ I & \Lambda^{-1}Y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -r_d \\ \sigma\mu\Lambda^{-1}e - y - \Lambda^{-1}\Lambda^{\text{cor}}Y^{\text{cor}}e - r_p \end{bmatrix},$$

from which we derive

$$\begin{aligned}(G + Y^{-1}\Lambda)\Delta x &= \sigma\mu Y^{-1}e - \lambda - Y^{-1}\Lambda^{\text{cor}}Y^{\text{cor}}e - Y^{-1}\Lambda r_p - r_d,\\ \Delta y &= \Delta x + r_p,\\ \Delta\lambda &= G\Delta x + r_d.\end{aligned}$$

The above equations gives us the final step directions, and are solved on lines 70-72 in the following code, which implements the interior points predictor-corrector method for quadratic problems with non-negativity inequality constraints.

`PC_QP.m`:

```
1  function [x,y,a,Z,k,T,M,R,C] = PC_QP(G,c,x0,y0,lambda0,tol,N,cgtol,cgN)
2  % Uses the Predictor-Corrector Method for QP to solve
3  %          min z=0.5*x'*G*x+c'*x  s.t.  x>=0
4  % INPUT:
5  %        G = Matrix            = G Matrix from Problem Statement
6  %        c = Column Vector     = c Vector from Problem Statement
7  %       x0 = Column Vector     = Initial Guess for Variables
8  %       y0 = Colume Vector     = Initial Guess for Slack
9  %  lambda0 = Column Vector     = Initial Guess for Lagrangian Multiplier
10 %      tol = Positive Real     = Complementarity Gap Error Tolerance
11 %        N = Positive Integer = Maximum Number of Iterations
12 % OUTPUT:
13 %        x = Column Vector = Primal Minimizer
14 %        y = Column Vector = Dual Minimizer
15 %        a = Column Vector = Slack Vector for Dual Problem
16 %        Z = Column Vector = Solution History for Primal Problem
17 %        k = Integer       = Number of Iterations
18 %        T = Positive Real = Computation Time
19 %        M = Column Vector = Complementarity Gap History
20 %        R = Column Vector = Solution Residual History
21 %        C = Column Vector = Constraint Residual History History
22 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
23     tic;             % Start Algorithm Timer
24     n = length(x0); % Problem Size
25     x = x0;          % Initial Variables
26     y = y0;          % Initial Slack
27     a = lambda0;     % Initial Lagrange Multipliers
28
29     % Ensure Good Starting Points
30     D  = spdiags(a./y,0,n,n);              % Diag. Matrix with lambda/y
```

```
  rd = G*x-a+c;                         % Solution Residual
  rp = x-y;                             % Constrait Residual
  dx = cglin(x,G+D,-a-D*rp-rd,cgtol,cgN); % Affine x Direction
  dy = dx+rp;                           % Affine y Direction
  da = G*dx+rd;                         % Affine a Direction
  dx = cglin(x,G+D,-da.*dy./y,cgtol,cgN); % Corrector for x Direction
  dy = dx;                              % Corrector for y Direction
  da = G*dx;                            % Corrector for a Direction
  x(1:n) = max(1,abs(x+dx));            % Adjust Initial x Condition
  y(1:n) = max(1,abs(y+dy));            % Adjust Initial y Condition
  a(1:n) = max(1,abs(a+da));            % Adjust Initial a Condition

  % Calculate Initial Solution and Residuals
  [Z,R,C,M] = deal(zeros(N,1)); % Allocate Memory
  Z(1) = 0.5*x'*G*x+x'*c;       % Store Initial Solution
  R(1) = norm(G*x+c);           % Store Initial Solution Residual
  C(1) = norm(rp);              % Store Initial Constraint Residual
  M(1) = y'*a/n;                % Store Initial Complementarity Measure
  k=1;                          % Set k, In Case R(1) <= tol
  if R(1) > tol                 % Store If Residual is large enough
    for k = 1:N                   % Then execute the method
      % Compute Affine Scaling Step
      D  = spdiags(a./y,0,n,n);             % Diag. Matrix with lambda/y
      rd = G*x-a+c;                         % Lagrangian Residual
      rp = x-y;                             % Constrait Residual
      dx = cglin(x,G+D,-a-D*rp-rd,cgtol,cgN); % Affine x Direction
      dy = dx+rp;                           % Affine y Direction
      da = G*dx+rd;                         % Affine a Direction
      dx = cglin(x,G+D,-da.*dy./y,cgtol,cgN); % Corrector for x Direction
      dy = dx;                              % Corrector for y Direction
      da = G*dx;                            % Corrector for a Direction

      % Calculate Centering Parameter
      m = M(k);                                % Complementarity Measure
      alpha = calc_max_alpha([y;a],[dy;da],1); % Affine Step Length
      ma = (y+alpha*dy)'*(a+alpha*da)/n;       % Affine Comp. Measure
      s = (ma/m)^3;                            % Centering Parameter

      % Compute Directions
      dx = cglin(x,G+D,s*m./y-a-a.*dy./y-D*rp-rd,cgtol,cgN); % x Direction
      dy = dx+rp;                                          % y Direction
      da = G*dx+rd;                                        % a Direction

      % Set Step Length
      t  = max(1-m,0.5);           % Distance From Max Step Length
      ap = calc_max_alpha(y,dy,t); % Primal Step Length
      ad = calc_max_alpha(a,da,t); % Dual Step Length
      alpha = min(ap,ad);          % Final Step Length

      % Update Estimates
      x = x + alpha*dx; % Update Variables
      y = y + alpha*dy; % Update Slack
      a = a + alpha*da; % Update Lagrange Multipliers

```

```
        % Calculate Solution and Residuals
        Z(k+1) = 0.5*x'*G*x+x'*c; % Store Solution
        R(k+1) = norm(G*x+c);     % Store Solution Residual
        C(k+1) = norm(x-y);       % Store Constraint Residual
        M(k+1) = y'*a/n;          % Store Complementarity Measure
        if R(k+1) <= tol          % If Solution within Tolerance
            break;                  % Exit the Loop
        end
      end
    end
    Z = Z(1:k+1); % Cut off Unused Elements of Solution History
    R = R(1:k+1); % Cut off Unused Elements of Residual History
    C = C(1:k+1); % Cut off Unused Elements of Constraint Residual History
    M = M(1:k+1); % Cut off Unused Elements of Comp. Measure History
    T = toc;      % Record Algorithm Time
end

function a = calc_max_alpha(x,dx,t)
% Calculates Maximum a such that
%    x+a*dx>=(1-t)*x   with   0<a<=1
% INPUT:
%   x = Column Vector
%  dx = Column Vector
%   t = Scalar: 0<t<1
% OUTPUT:
%   a = Scalar: 0<a<=1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    [wcs,i] = min(t*x+dx); % Value of Worst Case Scenario
    i = i(1);              % Index of Worst Case Scenario
    if wcs < 0             % If Worst Case Scenario is Negative
        a = -t*x(i)/dx(i);   % Then Find a  s.t.  x_i+a*dx_i==(1-t)*x_i
    else                   % Otherwise
        a = 1;               % Use the Maximum Value for a
    end
end
```

2.2 Poisson Matrix Problem

In this section, we test the algorithm on the problem where the G matrix is the 100×100 Poisson matrix and the c vector has all components equal to -1. The solution to this problem is known to be -250.5. The results of the algorithm are given in the following table. To save space, most of the iterations are left out of the table.

```
Results_Poisson =

  20x4 table
```

k	Sol	Residual	D_Gap
0	-146.39	6.3495	2.1867
10	-249.56	0.85565	0.31613
20	-250.38	0.31519	0.11123
30	-250.46	0.189	0.06595
40	-250.48	0.13432	0.046638
50	-250.49	0.104	0.036007
60	-250.5	0.084766	0.029298
100	-250.5	0.048606	0.016744
500	-250.5	0.0091675	0.0031464
900	-250.5	0.0050558	0.0017346
1300	-250.5	0.0034899	0.0011972
1700	-250.5	0.0026644	0.00091393
2100	-250.5	0.0021547	0.00073906
2500	-250.5	0.0018087	0.00062035
2900	-250.5	0.0015584	0.0005345
3300	-250.5	0.001369	0.00046951
3700	-250.5	0.0012206	0.00041862
4100	-250.5	0.0011012	0.00037768
4500	-250.5	0.0010031	0.00034403
4515	-250.5	0.00099979	0.00034288

We see that the algorithm converges in 4515 iterations to the desired result of -250.5. This confirms that the algorithm is working as intended. The following code reproduces the results.

`QP_Poisson.m`:

```matlab
% Create Matrices defined in Problem Statement
G = gallery('poisson',10); % 100x100 Poisson Matrix
n = size(G,1);             % Problem Size
c = -ones(n,1);            % Vector with all elements -1

% Create Initial Conditions
[x0,y0,lambda0] = deal(ones(n,1));

% Error and Iteration Parameters
```

```
tol   = 1e-3;  % Error Tolerance for PC Method
cgtol = 1e-12; % Error TOlerance for CG Method
N     = 20000; % Maximum Number of Iterations for PC Method
cgN   = 1000;  % Maximum Number of Iterations for CG Method

% Solve Problem Using-Interior Point Predictor-Corrector Method for QP
[x,y,a,Z,k,T,M,R,C] = PC_QP(G,c,x0,y0,lambda0,tol,N,cgtol,cgN);

%% Display Results

K = [(0:10:60)'; (100:400:k)'; k];
Z2 = Z(K+1); M2 = M(K+1); R2 = R(K+1); C2 = C(K+1);
Results_Poisson = table(K,Z2,R2,M2,...
                        'VariableNames',{'k','Sol','Residual','D_Gap'})
```

2.3 Formulating LASSO as QP Problem

We recast the LASSO problem as a QP problem and subsequently solve it using the predictor-correct algorithm in section 2.1. First, we introduce the LASSO problem

$$\min_x q(x) = \frac{1}{2}||b - Ax||_2^2 + \lambda_{pen}||x||_1 \qquad \text{s.t.} \qquad x \geq 0,$$

where λ_{pen} is a tuning parameter which controls regularization. In order to solve the LASSO problem as a QP, we start by introducing

$$x = x^+ - x^-$$

where $x^+ > 0$ and $x^- > 0$ represent the positive and negative components of the solution. Consequently these transformations follow:

$$x = \begin{pmatrix} x^+ \\ x^- \end{pmatrix}, \qquad G = \begin{pmatrix} A^TA & -A^TA \\ -A^TA & A^TA \end{pmatrix}, \qquad c = \left(\lambda_{pen} e_{2n} - \begin{pmatrix} A^Tb \\ -A^Tb \end{pmatrix} \right) \tag{6}$$

where e_{2n} is the 2n-dimensional unity vector.

2.4 ADMM and SuiteLasso

We obtain two additional solvers to solve the LASSO problem and all three solvers (QP, ADMM and SuiteLasso) are used on three different datasets from the UCI machinelearning repository, namely `abalone_scale`, `bodyfat_scale` and `mpg_scale`. The solvers and the datasets were obtained from the links provided from the problem statement. The call to ADMM solver is straight forward and relied on parameter values suggested by the authors. The call to SuiteLasso was more intricate and so a wrapper was designed to call the SuiteLasso algorithm. The tolerance and lambda values for all three algorithms were set to 10^{-6} and $10^{-4}||A^Tb||_\infty$, respectively, according to the problem statement. There are three aditional datasets which were solved using ADMM and SuiteLasso, namely `abalone_scale_expanded9`, `body_fat_expanded9` and `mpg_scale_expanded8`. The original datasets were expanded to higher order polynomials to create the last 3 datasets according to the problem statement. Additionally there is an extra dataset `E2006-log1p` which was expanded and stored in the 'SuiteLasso-0\UCIdata' directory.

2.5 Driver Code

The following driver code was used to set up the problem and call all the three solvers.

`Part2.m`

```
clear; close all; clc;

%% Poisson and LASSO Problem Setup & Predictor-Corrector QP Driver

% Create Matrices defined in Problem Statement
G = gallery('poisson',10);  % 100x100 Poisson Matrix
n0 = size(G,1);             % Problem Size
c = -ones(n0,1);            % Vector with all elements -1

% Create Matrices for LASSO problem from UCI datasets
[b1, A1] = libsvmread('mpg_scale');  % Import mpg_scale data
n1 = 2*size(A1,2);                   % Problem size
lambda_max = norm( A1'*b1, 'inf' );
lambdal = (1e-4)*lambda_max;         % lambda as defined in Problem
Al = A1'*A1;                         % matrix used in LASSO transformation
bl = A1'*b1;                         % vector used in LASSO transformation
Alasso = [ Al -Al; -Al Al ];         % QP to LASSO A matrix
blasso = (lambdal*ones(n1,1) - [bl; -bl]); % QP to LASSO b matrix

% Create Initial Conditions
[x0,y0,lambda0] = deal(ones(n0,1));
[x1,y1,lambda1] = deal(ones(n1,1));

% Error and Iteration Parameters
tol   = 1e-6;    % Error Tolerance for PC Method
cgtol = 1e-6;    % Error TOlerance for CG Method
N     = 700;     % Maximum Number of Iterations for PC Method
cgN   = 1000;    % Maximum Number of Iterations for CG Method

% Solve Problem Using Interior-Point Predictor-Corrector Method for QP
[x0,y0,a0,Z0,k0,T0,M0,R0,C0] = ...
    PC_QP(G,c,x0,y0,lambda0,tol,N,cgtol,cgN);
[x1,y1,a1,Z1,k1,T1,M1,R1,C1] = ...
    PC_QP(Alasso,blasso,x1,y1,lambda1,tol,N,cgtol,cgN);

% Adjust QP solution form to LASSO
xlasso = x1(1:n1/2) - x1(n1/2+1:end);
ylasso = y1(1:n1/2) - y1(n1/2+1:end);
alasso = a1(1:n1/2) - a1(n1/2+1:end);
objlasso = 0.5*norm([b1-A1*xlasso],2)^2 + lambdal*norm(xlasso,1);

%% LASSO Problem Setup & ADMM Driver

% import data
data11 = importdata('SuiteLasso-0/UCIdata/mpg_scale_expanded8.mat');
data12 = importdata('SuiteLasso-0/UCIdata/abalone_scale_expanded9.mat');
data13 = importdata('SuiteLasso-0/UCIdata/bodyfat_scale_expanded9.mat');
```

```
% Create matrices for LASSO problem from UCI datasets
[b2, A2] = libsvmread('abalone_scale');
[b3, A3] = libsvmread('bodyfat_scale');
A11 = data11.A;
b11 = data11.b;
A12 = data12.A;
b12 = data12.b;
A13 = data13.A;
b13 = data13.b;

% lambda as defined in Problem Statement
lambda_max_2 = norm( A2'*b2, 'inf' );
lambda_max_3 = norm( A3'*b3, 'inf' );
lambda_max_11 = norm( A11'*b11, 'inf' );
lambda_max_12 = norm( A12'*b12, 'inf' );
lambda_max_13 = norm( A13'*b13, 'inf' );
lambda2 = (1e-4)*lambda_max_2;
lambda3 = (1e-4)*lambda_max_3;
lambda11 = (1e-4)*lambda_max_11;
lambda12 = (1e-4)*lambda_max_12;
lambda13 = (1e-4)*lambda_max_13;


% Solve LASSO Problem Using ADMM
[x2 history2 T2] = lasso(A2, b2, lambda2, 1.0, 1.0);
[x3 history3 T3] = lasso(A3, b3, lambda3, 1.0, 1.0);
[x4 history4 T4] = lasso(A1, b1, lambda1, 1.0, 1.0);
[x11 history11 T11] = lasso(A11, b11, lambda11, 1.0, 1.0);
[x12 history12 T12] = lasso(A12, b12, lambda12, 1.0, 1.0);
[x13 history13 T13] = lasso(A13, b13, lambda13, 1.0, 1.0);

%% LASSO Problem Setup & SuiteLasso Driver

% Create list for LASSO problem from UCI datasets
dataset = {'abalone_scale_expanded9', ...
            'bodyfat_scale_expanded9', ...
            'mpg_scale_expanded8', ...
            'abalone_scale','bodyfat_scale','mpg_scale' };

% lambda Multipler as defined in Problem Statement
lambdaCoef = 1e-4;

% Solve LASSO Problem Using SuiteLasso
[obj5,y5,xi5,x5,info5,runhist5] = ...
    SuiteLassoDriverWrapper(dataset{1},lambdaCoef,tol,true);
[obj6,y6,xi6,x6,info6,runhist6] = ...
    SuiteLassoDriverWrapper(dataset{2},lambdaCoef,tol,true);
[obj7,y7,xi7,x7,info7,runhist7] = ...
    SuiteLassoDriverWrapper(dataset{3},lambdaCoef,tol,true);
[obj8,y8,xi8,x8,info8,runhist8] = ...
    SuiteLassoDriverWrapper(dataset{4},lambdaCoef,tol,false);
[obj9,y9,xi9,x9,info9,runhist9] = ...
    SuiteLassoDriverWrapper(dataset{5},lambdaCoef,tol,false);
```

```
[obj10,y10,xi10,x10,info10,runhist10] = ...
    SuiteLassoDriverWrapper(dataset{6},lambdaCoef,tol,false);

%% Display Results
clc;
%Problem Definition
problem = {'Gmatrix','mpg_scale', 'mpg_scale', 'mpg_scale', ...
           'mpg_scale_expanded8','mpg_scale_expanded8', ...
           'abalone_scale', 'abalone_scale', ...
           'abalone_scale_expanded9', 'abalone_scale_expanded9',...
           'bodyfat_scale', 'bodyfat_scale', ...
           'bodyfat_scale_expanded9', 'bodyfat_scale_expanded9'};

solver = {'PC_QP', 'PC_QP', 'ADMM', 'SuiteLasso', 'SuiteLasso', 'ADMM' ...
          'ADMM', 'SuiteLasso', 'SuiteLasso', 'ADMM', ...
          'ADMM', 'SuiteLasso', 'SuiteLasso', 'ADMM'};


%Final Solution and Time
time = [ T0, T1, T4, info10.time, info7.time, T11, ...
        T2, info8.time, info5.time, T12, T3, info9.time, info6.time, T13];


obj = [ Z0(end), objlasso, history4.objval(end), info10.obj(1), ...
        info7.obj(1), history11.objval(end), ...
        history2.objval(end), info8.obj(1), ...
        info5.obj(1), history12.objval(end), ...
        history3.objval(end), ...
        info9.obj(1), info6.obj(1), history13.objval(end) ];

sol = { x0, xlasso, x4, x10, x7, x11, x2, x8, x5, x12, x3, x9, x6, x13};

% Sparsity Calculation
p = length(sol);
zeroCount = zeros(p,1);
totCount = zeros(p,1);

for k = 1:p
    idx = sol{k} < 1e-4;
    zeroCount(k) = sum(idx);
    totCount(k) = length(sol{k});
end

spar = (zeroCount./totCount);
format long
% Results Comparison displayed in a table
Results_ComparisonA = table(problem',solver',obj',...
                   'VariableNames',{'Problem','Solver',...
                   'Optimal_Objective'})

Results_ComparisonB = table(problem',solver',spar*100,...
                   'VariableNames',{'Problem','Solver',...
                   'Sparsity_Precentage'})

```

```
Results_ComparisonC = table(problem',solver',zeroCount,...
                      'VariableNames',{'Problem','Solver',...
                      'Sparsity_Count'})

Results_ComparisonD = table(problem',solver',time',...
                      'VariableNames',{'Problem','Solver',...
                      'Wall_Clock_Time'})
```

This following function script was used to call the SuiteLasso Solver.

SuiteLassoDriverWrapper.m

```
function [obj,y,xi,x,info,runhist] = ...
    SuiteLassoDriverWrapper(dataset,crhoadj,stoptoladj,expanded)
% Calls SuiteLasso alogrithm to solve LASSO problem
%         min 1/2*|| Ax - b ||_2^2 + \lambda || x ||_1
% INPUT:
%    dataset    = Cell Vector      = Matrix containting list of dataset
%    names
%    crhoadj    = Positive Real    = lambdamax adjustment factor
%    stoptoladj = Positive Real    = Error Tolerance
%    expanded   = Boolean          = True if data expanded, false otherwise
% OUTPUT:
%    obj        = Row Vector       = Optimal Primal and Dual Objective
%    y          = Column Vector    = Dual Minimizer
%    xi         = Column Vector    = Slack Vector for Dual Problem
%    x          = Column Vector    = Primal Minimizer
%    info       = Data Structure   = Problem Info and Final Solution
%    runhist    = Data Structure   = Variable and Solution History
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


 % look for dataset file
 if expanded
 filepath = strcat('SuiteLasso-0/UCIdata/',dataset,'.mat');
 else
 filepath = strcat(dataset);
 end

 if ~isfile(filepath)
     error('Solver Terminated. Can not find file %s.',filepath)
 end

 % import data

 if expanded
 data = importdata(filepath);
 A = data.A;
 b = data.b;
 else
 [b, A] = libsvmread(filepath);
 end
```

```


 m = size(A,1);
 n = size(A,2);
 Amap  = @(x) A*x;
 ATmap = @(x) A'*x;

 % tuning parameters

lambdamax=norm(ATmap(b),'inf');
eigsopt.issym = 1;
Rmap=@(x) A*x;
Rtmap=@(x) A'*x;
RRtmap=@(x) Rmap(Rtmap(x));
Lip = eigs(RRtmap,length(b),1,'LA',eigsopt);

fprintf('\n------------------------------------------------');
fprintf('------------------------------')
fprintf('\n Problem: n = %g,  m = %g    lambda(max) = %g ',n,m, lambdamax)
fprintf('\n Lip = %3.2e', Lip);
fprintf('\n------------------------------------------------');
fprintf('------------------------------')

stoptol = stoptoladj;%stopping tol

for crho = crhoadj
   lambda=crho*lambdamax;
   if (true)
      opts.stoptol = stoptol;
      opts.Lip = Lip; %can be set to 1
      Ainput.A = A;
      Ainput.Amap = @(x) Amap(x);
      Ainput.ATmap = @(x) ATmap(x);
      if (false) %% supply initial point
         x0 = zeros(n,1);
         y0 = zeros(n,1);
         xi0= zeros(m,1);
         [obj,y,xi,x,info,runhist] = Classic_Lasso_SSNAL(Ainput,b,n,lambda,nalop,y0,xi0,x0
      else %% no initial point available
         [obj,y,xi,x,info,runhist] = Classic_Lasso_SSNAL(Ainput,b,n,lambda,opts);
      end
   else
      [obj,y,xi,x,info,runhist] = Classic_Lasso_ADMM(Ainput,b,n,lambda,opts);
   end
end

end
```

2.6 Comparative Results

Results_ComparisonA =

14x3 table

Problem	Solver	Optimal_Objective
'Gmatrix'	'PC_QP'	-250.504512616031
'mpg_scale'	'PC_QP'	11801.9824000958
'mpg_scale'	'ADMM'	11801.9823988412
'mpg_scale'	'SuiteLasso'	11801.9823988412
'mpg_scale_expanded8'	'SuiteLasso'	881.370300163357
'mpg_scale_expanded8'	'ADMM'	881.370114240593
'abalone_scale'	'ADMM'	10938.5592518438
'abalone_scale'	'SuiteLasso'	10938.5592518438
'abalone_scale_expanded9'	'SuiteLasso'	9287.41498273849
'abalone_scale_expanded9'	'ADMM'	9282.90325695463
'bodyfat_scale'	'ADMM'	1.8054371138958
'bodyfat_scale'	'SuiteLasso'	1.80543702420266
'bodyfat_scale_expanded9'	'SuiteLasso'	0.0303024675471388
'bodyfat_scale_expanded9'	'ADMM'	0.0301555316920478

Results_ComparisonB =

14x3 table

Problem	Solver	Sparsity_Precentage
'Gmatrix'	'PC_QP'	0
'mpg_scale'	'PC_QP'	57.1428571428571
'mpg_scale'	'ADMM'	57.1428571428571
'mpg_scale'	'SuiteLasso'	57.1428571428571
'mpg_scale_expanded8'	'SuiteLasso'	98.9121989121989
'mpg_scale_expanded8'	'ADMM'	98.9121989121989
'abalone_scale'	'ADMM'	50
'abalone_scale'	'SuiteLasso'	50
'abalone_scale_expanded9'	'SuiteLasso'	99.8601398601399
'abalone_scale_expanded9'	'ADMM'	99.8601398601399
'bodyfat_scale'	'ADMM'	28.5714285714286
'bodyfat_scale'	'SuiteLasso'	28.5714285714286
'bodyfat_scale_expanded9'	'SuiteLasso'	99.9996328883124
'bodyfat_scale_expanded9'	'ADMM'	99.9996328883124

Results_ComparisonC =

14x3 table

Problem	Solver	Sparsity_Count
'Gmatrix'	'PC_QP'	0
'mpg_scale'	'PC_QP'	4
'mpg_scale'	'ADMM'	4
'mpg_scale'	'SuiteLasso'	4
'mpg_scale_expanded8'	'SuiteLasso'	6365
'mpg_scale_expanded8'	'ADMM'	6365
'abalone_scale'	'ADMM'	4
'abalone_scale'	'SuiteLasso'	4
'abalone_scale_expanded9'	'SuiteLasso'	24276
'abalone_scale_expanded9'	'ADMM'	24276
'bodyfat_scale'	'ADMM'	4
'bodyfat_scale'	'SuiteLasso'	4
'bodyfat_scale_expanded9'	'SuiteLasso'	817187
'bodyfat_scale_expanded9'	'ADMM'	817187

```
Results_ComparisonD =

  14x3 table
```

Problem	Solver	Wall_Clock_Time
'Gmatrix'	'PC_QP'	0.000232
'mpg_scale'	'PC_QP'	0.000199
'mpg_scale'	'ADMM'	0.00096
'mpg_scale'	'SuiteLasso'	0.0150810000000021
'mpg_scale_expanded8'	'SuiteLasso'	0.360639000000003
'mpg_scale_expanded8'	'ADMM'	2.906686
'abalone_scale'	'ADMM'	0.002408
'abalone_scale'	'SuiteLasso'	0.0426179999999974
'abalone_scale_expanded9'	'SuiteLasso'	5.36099
'abalone_scale_expanded9'	'ADMM'	93.975171
'bodyfat_scale'	'ADMM'	0.003272
'bodyfat_scale'	'SuiteLasso'	0.0215849999999982
'bodyfat_scale_expanded9'	'SuiteLasso'	8.313787
'bodyfat_scale_expanded9'	'ADMM'	13.856137

It is interesting to note that the QP solver almost matched the ADMM solver by needing roughly the same wall clock time to produce an optimal objective which differed by an order of 10^{-4}. Moreover, SuiteLasso required at least an order of magnitude higher than ADMM to solve each dataset. The SuiteLasso algorithm prowess is shown, however, as each dataset is expanded. Indeed SuiteLasso performs consistently better (in terms of lower wall clock time) than ADMM for all expanded datasets. Sparsity is reported as both a percentage and zero count. The sparsity and the optimal objective value did not differ between the two solvers for all datasets tested.