```
clc;
clear;
close all;
clear variables
format shortG;
```

1 Quadractic programming solver

1.a Standard dual projected gradient algorithm

Equation in the question is

$$3x_1^2 + x_2^2 + 2x_1x_2 + x_1 + 6x_2 + 2$$

the 2 can be ignored from the optimization problem as it is a constant the resulting matrices that produce this equation are

```
H = [6 2;
     2 2];
q = [1; 6];
```

constraints are

$$2x_1 + 3x_2 \geq 4$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

have to reverse the sign of the inequality to make it $\leq$

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

% run with MATLAB's quadprog
[x, fval, exitflag, output, lam] = quadprog(H, q, A, b);
disp('1.a Standard dual projected gradient algorithm')
disp('MATLAB quadprog:');
disp('Solution =');
disp(x);
disp('Lagrange multipliers =');
disp(lam.ineqlin);

[x_my, lam_my] = myQP(H, q, A, b, zeros(size(A, 1), 1));
disp('myQP:');
disp('Solution =');
disp(x_my);
disp('Lagrange multipliers =');
disp(lam_my);
```

```
Minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

1.a Standard dual projected gradient algorithm
MATLAB quadprog:
Solution =
          0.5
            1

Lagrange multipliers =
            3
   6.8148e-10
   3.6335e-12
```

1.c Discretize the mass-spring-damper system

```
disp('1.c Discrete time model of the mass-spring system')
% have to re-declare A and B with consants for m and k
Ts = 0.1;
m = 1;
k = 1;
Ac = [0 1;
      -k / m 0];
Bc = [0;
      1 / m];

[Ad, Bd, Cd, Dd] = c2dm(Ac, Bc, [], [], Ts, 'zoh');
disp('Ad =');
disp(Ad);
disp('Bd =');
disp(Bd);
```

```
1.c Discrete time model of the mass-spring system
Ad =
        0.995     0.099833
    -0.099833        0.995

Bd =
    0.0049958
     0.099833
```

1.d Tracking MPC formulation

$\Delta x_{1,k+1}$ and $\Delta x_{2,k+1}$ can be represented as x_{k+1}

Then $\Delta x_{k+1} = x_{k+2} - x_{k+1}$

$= Ax_{k+1} + Bu_{k+1} - (Ax_k + Bu_k)$

$= A(x_{k+1} - x_k) + B(u_{k+1} - u_k)$

Using definitions of Δx_k and Δu_k

$\Delta x_{k+1} = A\Delta x_k + B\Delta u_k$

The error e can be calculated by:

$x_{k+1} = x_k + \Delta x_k$ -- (1)

$e_k = x_k - r$ -- (2)

$e_{k+1} = x_{k+1} - r$ -- (3)

Substituting (1) into (3)

$e_{k+1} = x_k + \Delta x_k - r$

Using (2) to reduce this equation

Thus: $e_{k+1} = e_k + \Delta x_k$

$u_{k+1} = u_k + \Delta u_k$, so it just uses the A_d to pick u and B_d to pick Δu.

$x_{1,k+1} = x_{1,k} + \Delta x_{1,k}$ by just rearranging the given equation for Δx_k

```
A = [Ad, zeros(2, 3);
     1, 0, 1, 0, 0;
     0, 0, 0, 1, 0;
     1, 0, 0, 0, 1];
B = [Bd; 0; 1; 0];
```

1.e LQ-MPC Problem formulation

```
Q = diag([0, 0, 1, 0, 0]);
R = 1;
[K, Pdxu, E] = dlqr(A(1:3, 1:3), B(1:3), Q(1:3, 1:3), R);
P = blkdiag(Pdxu, zeros(2, 2));
disp('1.e LQ-MPC Problem formulation')
disp('P =');
disp(P);
```

```
1.e LQ-MPC Problem formulation
P =
       304.52        85.262        44.817             0             0
       85.262        40.051        9.9917             0             0
       44.817        9.9917        10.033             0             0
            0             0             0             0             0
            0             0             0             0             0
```

1.f Represent constraints

```
lN = 10;
% large number
xlim.max = [lN, lN, lN, 0.2, 0.2]; % [dx1, dx2, e, u, x_1]
xlim.min = -xlim.max;
xlim20.max = [lN, lN, lN, 0.25, 0.2]; % [dx1, dx2, e, u, x_1]
xlim20.min = -xlim20.max;
% Note our "control" is "control increment", actual control is the fourth
state
ulim.max = [lN];
ulim.min = -ulim.max;
```

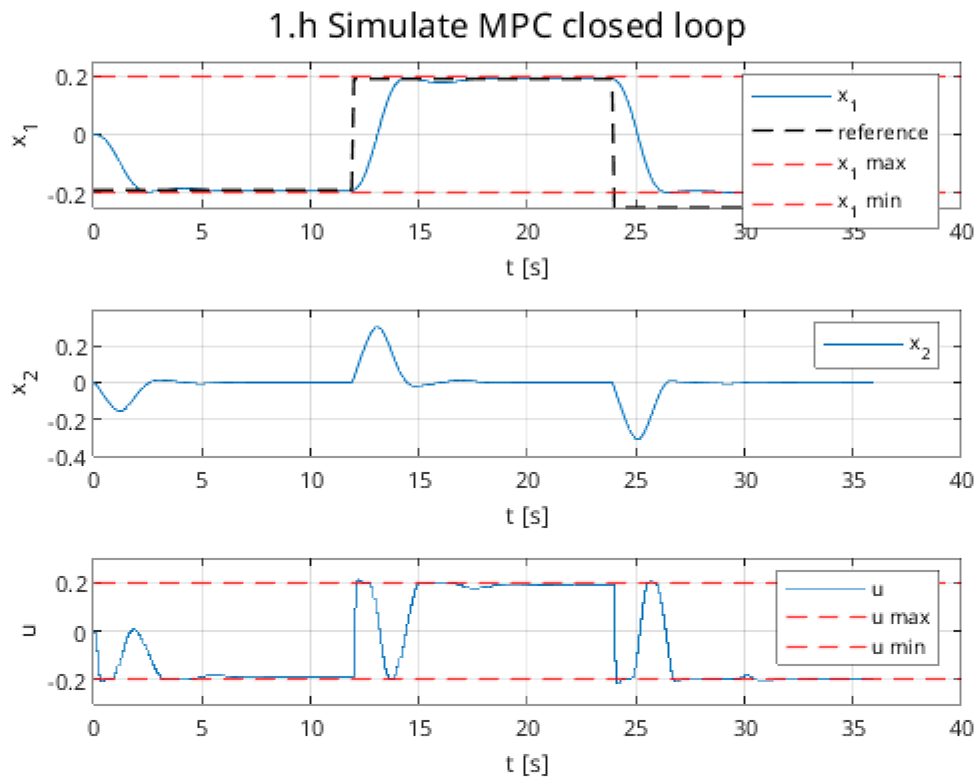
1.h/i Simulate MPC closed loop

disp('1.h Simulate MPC closed loop') m = 1; k = 1;

```
N = 15;
cmds = [-0.19 0.19 -0.25];
cmd = cmds(1);
prev_cmd = cmd;
% start with everything as 0, so the error is now the step reference command
xk_msd = [0; 0];
xk_msd20 = [0; 0];
xk = [0; 0; -cmd; 0; 0];
xk20 = [0; 0; -cmd; 0; 0];
Tcmd = 12;
Tsim = Tcmd * length(cmds);
times = 0:Ts:Tsim;
fidelity = Ts / 10;
% form the matrices needed for QP
[H, L, G, W, T, IMPC] = formQPMatrices(A, B, Q, R, P, xlim, ulim, N);
[H20, L20, G20, W20, T20, IMPC20] = formQPMatrices(A, B, Q, R, P, xlim20,
ulim, N);
lam = ones(size(G, 1), 1);
lam20 = ones(size(G20, 1), 1);

data.x_msd = zeros(2, Tsim / Ts);
data.x_msd20 = zeros(2, Tsim / Ts);
data.u = zeros(1, Tsim / Ts);
data.u20 = zeros(1, Tsim / Ts);
data.r = zeros(1, Tsim / Ts);
data_idx = 1;

for t = times
  % get the current command
  cmd = cmds(min(1 + floor(t / Tcmd), length(cmds)));

  if cmd ~= prev_cmd
    % update the error in the state
    xk(3) = xk(3) - cmd + prev_cmd;
    xk20(3) = xk20(3) - cmd + prev_cmd;
  end
```

```
  % solve the QP
  [U, lam] = myQP(H, L * xk, G, W + T * xk, lam);
  [U20, lam20] = myQP(H20, L20 * xk20, G20, W20 + T20 * xk20, lam20);
  % get the first control increment
  delta_uk = IMPC * U;
  delta_uk20 = IMPC20 * U20;
  uk = xk(4) + delta_uk; % nu is just 1 so this works
  uk20 = xk20(4) + delta_uk20; % nu is just 1 so this works
  % simulate the system
  [~, xk1_msd_ode] = ode45(@(t, x) msd(t, x, uk, 1, 1), [t +
fidelity:fidelity:t + Ts], xk_msd);
  [~, xk1_msd_ode20] = ode45(@(t, x) msd(t, x, uk20, 0.8, 1.2), [t +
fidelity:fidelity:t + Ts], xk_msd20);
  xk1_msd = xk1_msd_ode(end, :)';
  xk1_msd20 = xk1_msd_ode20(end, :)';
  % update the state
  delta_xk_msd = xk1_msd - xk_msd;
  xk1 = [delta_xk_msd; % delta x1 and x2
         xk(3) + delta_xk_msd(1); % e
         uk; % u
         xk1_msd(1)]; % x1

  delta_xk_msd20 = xk1_msd20 - xk_msd20;
  xk120 = [delta_xk_msd20; % delta x1 and x2
           xk20(3) + delta_xk_msd20(1); % e
           uk20; % u
           xk1_msd20(1)]; % x1
  % update
  xk_msd = xk1_msd;
  xk_msd20 = xk1_msd20;
  xk = xk1;
  xk20 = xk120;
  prev_cmd = cmd;
  % save data
  data.x_msd(:, data_idx) = xk_msd;
  data.x_msd20(:, data_idx) = xk_msd20;
  data.u(data_idx) = uk;
  data.u20(data_idx) = uk20;
  data.r(data_idx) = cmd;

  data_idx = data_idx + 1;
end

% plot the results
figure();
sgtitle('1.h Simulate MPC closed loop');
subplot(3, 1, 1);
plot(times, data.x_msd(1, :), 'DisplayName', 'x_1');
grid on;
hold on;
plot(times, data.r, '--k', 'DisplayName', 'reference', "LineWidth", 1);
yline(0.2, '--r', 'DisplayName', 'x_1 max', "LineWidth", 1);
yline(-0.2, '--r', 'DisplayName', 'x_1 min', "LineWidth", 1);
```

```
xlabel('t [s]');
ylabel('x_1');
ylim([-0.25, 0.25]);
legend();

subplot(3, 1, 2);
plot(times, data.x_msd(2, :), 'DisplayName', 'x_2');
grid on;
xlabel('t [s]');
ylabel('x_2');
ylim([-0.40, 0.40]);
legend();

subplot(3, 1, 3);
stairs(times, data.u, 'DisplayName', 'u');
grid on;
xlabel('t [s]');
ylabel('u');
yline(0.2, '--r', 'DisplayName', 'u max', "LineWidth", 1);
yline(-0.2, '--r', 'DisplayName', 'u min', "LineWidth", 1);
ylim([-0.30, 0.30]);
legend();
```

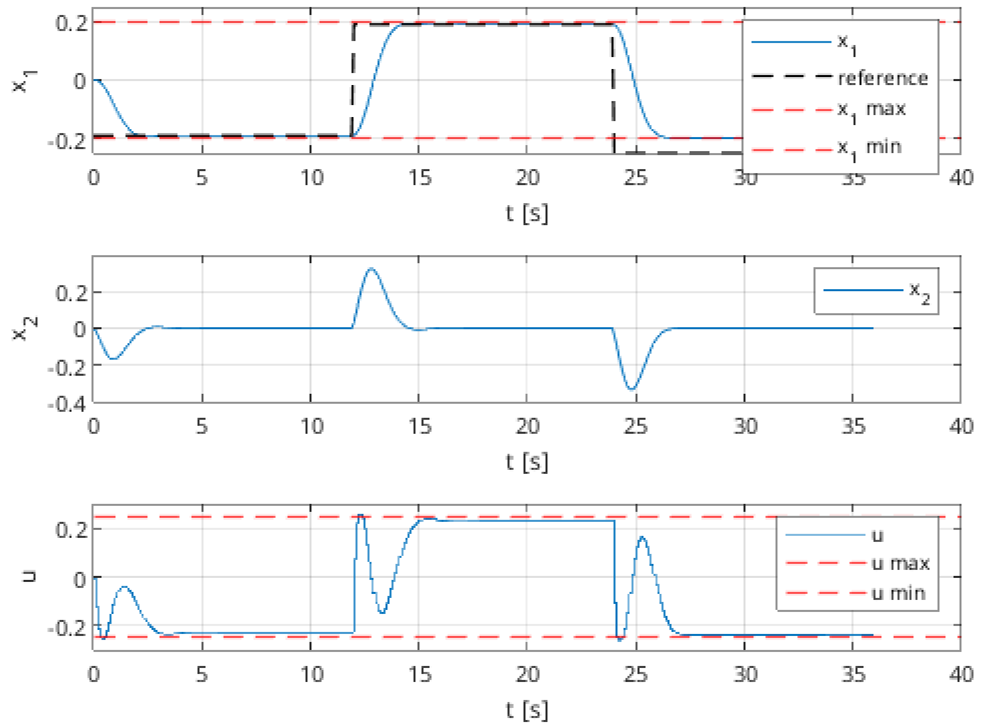


1.i Simulate MPC closed loop and comment on differernces

After reducing the mass an increasing the stiffness, we can see that the controller is not only robust to these changes, but it also performs better. We can notice that from the initial state, the controller now is able to reach the reference faster at 2.2s vs 2.6s. Additionally, we can observe that the oscillations in the state and input are reduced. However, these differences are mostly due to the fact that the mass is reduced and the stiffness is increased, making the dynamics of the mass spring system faster. The controller is also given more control authority so it can reach the reference by pushing the control input harder. Overall, the controller is still robust while facing a model mismatch by being able to push the control input more.

```
figure();
sgtitle('1.i Simulate MPC closed loop with different mass and stiffness');
subplot(3, 1, 1);
plot(times, data.x_msd20(1, :), 'DisplayName', 'x_1');
grid on;
hold on;
plot(times, data.r, '--k', 'DisplayName', 'reference', "LineWidth", 1);
yline(0.2, '--r', 'DisplayName', 'x_1 max', "LineWidth", 1);
yline(-0.2, '--r', 'DisplayName', 'x_1 min', "LineWidth", 1);
xlabel('t [s]');
ylabel('x_1');
ylim([-0.25, 0.25]);
legend();

subplot(3, 1, 2);
plot(times, data.x_msd20(2, :), 'DisplayName', 'x_2');
grid on;
xlabel('t [s]');
ylabel('x_2');
ylim([-0.40, 0.40]);
legend();

subplot(3, 1, 3);
stairs(times, data.u20, 'DisplayName', 'u');
grid on;
xlabel('t [s]');
ylabel('u');
yline(0.25, '--r', 'DisplayName', 'u max', "LineWidth", 1);
yline(-0.25, '--r', 'DisplayName', 'u min', "LineWidth", 1);
ylim([-0.30, 0.30]);
legend();
```

1.i Simulate MPC closed loop with different mass and stiffness



1.a Standard dual projected gradient Function

```
function [U, lam] = myQP(H, q, A, b, lam0)
  % This function implements the dual projected
  % gradient algorithm for solving a QP problem.
  % Minimize 1/2 * U' * H * U + q' * U subject to G * U <= Wtilde

  % compared to his notes, G=A, U=x, W=b
  G = A; Wtilde = b;
  invH = inv(H); G_invH = G * invH; % see Note 1
  Hd = G_invH * G';
  % see Note 1
  qd = G_invH * q + Wtilde;
  Nit = 30;
  % maximum number of iterations
  lam = lam0;
  L = norm(Hd);
  k = 1;
  df = Hd * lam + qd;

  while k <= Nit % see Note 2
    lam = max(lam - 1 / L * df, 0);
    df = Hd * lam + qd;
    k = k + 1;
  end
```

```
  U = -invH * (G' * lam + q);
  return;
  % Note 1: Can pre-compute these quantities and pass as arguments to
  % myQP. In LQ-MPC setting, only q and Wtilde depend on x_0, makes no
  % sense to constantly re-compute these
  % Note 2: Change to another criterion as desired
end
```

```
myQP:
Solution =
          0.5
            1

Lagrange multipliers =
            3
            0
            0
```

1.b Mass-spring dynamics

```
function xdot = msd(t, x, u, m_msd, k_msd)
  xdot = [
          x(2);
          (-k_msd / m_msd) * x(1) + (u / m_msd);
          ];
end
```

1.g function that forms matrices needed for QP

```
function [H, L, G, W, T, IMPC] = formQPMatrices(A, B, Q, R, P, xlim, ulim, N)
  % This function forms the matrices needed for the constrained QP
  % Inputs:
  %   A, B: state-space matrices
  %   Q, R, P: cost function matrices
  %   xlim, ulim: state and input constraints
  %   N: prediction horizon

  nx = size(A, 1);
  nu = size(B, 2);

  S = zeros(N * nx, N * nu);
  % Compute the first column of S
  for i = 1:N
    rowStart = (i - 1) * nx + 1;
    rowEnd = i * nx;
    S(rowStart:rowEnd, 1:nu) = A^(i - 1) * B;
  end

  % Pad the first column and set it to other columns of S
  for i = 2:N
    colStart = (i - 1) * nu + 1;
    colEnd = i * nu;
```

```
    zeroRows = (i - 1) * nx;
    zeroCols = nu;
    S(:, colStart:colEnd) = [zeros(zeroRows, zeroCols); S(1:end - zeroRows,
1:nu)];
  end

  M = zeros(N * nx, nx);
  % Compute first row of M
  M(1:nx, :) = A;
  % Compute the rest of M
  for i = 2:N
    rowStart = (i - 1) * nx + 1;
    rowEnd = i * nx;
    % just multiply the previous rows by A to get higher powers
    M(rowStart:rowEnd, :) = A * M(rowStart - nx:rowEnd - nx, :);
  end

  Qbar = zeros(N * nx, N * nx);
  % Compute Qbar and set the last row to P
  for i = 1:N
    % Q is square so we can reuse indices
    rowStart = (i - 1) * nx + 1;
    rowEnd = i * nx;
    temp = Q;

    if i == N
      temp = P;
    end

    Qbar(rowStart:rowEnd, rowStart:rowEnd) = temp;
  end

  Rbar = zeros(N * nu, N * nu);
  % Compute Rbar
  for i = 1:N
    % R is square so we can reuse indices
    rowStart = (i - 1) * nu + 1;
    rowEnd = i * nu;
    Rbar(rowStart:rowEnd, rowStart:rowEnd) = R;
  end

  H = S' * Qbar * S + Rbar;
  L = S' * Qbar * M;
  G = [S;
       -S;
       eye(N * nu);
       -eye(N * nu)];
  W = [
       repmat(xlim.max', N, 1);
       repmat(-xlim.min', N, 1);
       repmat(ulim.max', N, 1);
       repmat(-ulim.min', N, 1);
       ];
  T = [
```

```
      -M;
      M;
      zeros(N * nu, nx);
      zeros(N * nu, nx);
      ];
  IMPC = [eye(nu, nu), zeros(nu, (N - 1) * nu)];

end
```

Published with MATLAB® R2023b