Controlling the Compass Gait

Controlling the Compass Gait (Part I)

0.0/8.0 points (ungraded)

In this problem we will stabilize the passive limit cycle we obtained in the previous problem using a time-varying LQR (TVLQR) controller. The first step in order to do this is to modify the standard continuous-time TVLQR approach to work for hybrid systems. The Riccati equation associated with this problem is the so-called "Jump Riccati Equation" (JRE):

$$-\dot{S}(t) = Q - S(t)\,B(t)\,R^{-1}B(t)^T S(t) + S(t)\,A(a) + A(t)^T S(t) \quad (1)$$
$$S(t^-) = A_d^T S(t^+)\,A_d. \quad (2)$$

Here, equation (1) is the usual evolution of the cost (same as the continuous time case), which we solve backwards in time with a final-value constraint $S(t_f) = Q_f$. The second equation propagates the cost backwards through a hybrid event ("jump"). Here A_d is the linearization of the discrete transition dynamics (in our case, it is the linearization of the collision dynamics for the compass gait).

The second modification we need to the standard continuous-time TVLQR is to make its solution periodic. In other words, we want the cost-to-go at $t = 0$ to be the same as the cost-to-go at the final time $t = t_f$ (since the limit cycle is periodic). It is well-known (but not trivial to prove!) that repeatedly solving the equations (1) and (2) backwards in time will result in convergence to a periodic $S(t)$. Hence, we solve (1) backwards in time for the interval $[t^+, t_f]$ with a final-value constraint $S(t_f) = Q_f$. Then, we solve (2) using the $S(t^+)$ obtained from (1). We then solve (1) again with the final value constraint $S(t_f) = S(t^-)$, and so on until convergence.

We have provided stub code here that solves the continuous time equation (1) using Drake. Your task is to fill in the section corresponding to the discrete jump equation (2), and to check for convergence of $S(t)$ to a periodic solution. It might be helpful to take a look at the function **collisionDynamics** in CompassGaitPlant.m. The inputs to this function are the hybrid mode ("1" in our case), time, pre-impact state (a 4×1 vector) and control input. The first output is the post-impact state. The columns of the last output (dxp) are the gradients of the post-impact state with respect to the different inputs of the function. So, the columns of interest to us are the columns 3 through 6 (since these are the gradients with respect to the pre-impact state).

Note: This problem relies on your answer(s) from the previous problem. To be safe, make sure to use all the digits you comptued in Matlab in the previous part. Also set your tolerances much tighter than the ones we list in the problems (since there is almost no computational cost to doing this).

(a) What is the value of Q_f when the solution has converged (the variable "Qf_converged" in the code)? **Our tolerance for this answer is 0.01.**

```
Qf_converged = ;

```

Unanswered

```
% The Qf_converged we compute is:
Qf_converged =   1.0e+03 * [6.490617561949686  -2.137322594070590 -0.122512533982038   1.1995577
  -2.137322594070590   0.732684500175522   0.039887204353096  -0.391443040435211;
  -0.122512533982038   0.039887204353096   0.002331333037853  -0.022717723218988;
   1.199557787078113  -0.391443040435211  -0.022717723218988   0.222366589377603];

% Our full code is shown below:

% Compass Gait plant
p = CompassGaitPlant();

%% Get limit cycle initial condition (FILL ME IN) %%%%%%%%%%%%%%%%%%%%%%%%%
x0 = [-0.32338855;
0.21866879;
-0.37718213;
-1.0918269]; % Your x0 from part (b) of the problem "Poincare analysis on compass gait"
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x0 = [  -0.323388544149323
   0.218668789029664
  -0.377182130914653
  -1.091826912735189
];

% Simulate forwards from x0
xf = strideFunction(p,x0); % Your strideFunction from the previous problem

% Check that the resulting trajectory is periodic
if max(abs(xf-x0)) > 1e-4
    error('Simulating x0 does not result in a limit cycle');
end

% Simulate forwards from x0 to get the trajectory
xtraj = simulate(p,[0 1], [1; x0]);
xtraj = xtraj.traj{1}; % Extract first part of the trajectory (before collision)
ts = xtraj.getBreaks(); % Get time samples of trajectory
utraj = PPTrajectory(spline(ts,zeros(1,length(ts)))); % Define nominal u(t),
                                                      % which is all zeros since our
                                                      % system was passive

% Set frames
xtraj = xtraj.setOutputFrame(p.modes{1}.getStateFrame);
utraj = utraj.setOutputFrame(p.modes{1}.getInputFrame);


%% Stabilize using tvlqr (FILL ME IN) %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define Q, Qf, R
```

```
Q = diag([10 10 1 1]);
R = 1;
Qf = Q;

options = struct();

converged = false;

while ~converged
% tvlqr for continuous phase
[tv,V] = tvlqr(p.modes{1},xtraj,utraj,Q,R,Qf,options);
QfV = Qf;

% Jump equation (FILL ME IN)
S_t_plus = V.S.eval(0);
xend = xtraj.eval(ts(end));
[~,~,~,dxp] = p.collisionDynamics(1,0,xend,0);
Ad = dxp(:,3:end-1);

% S_t_minus = inv(Ad'*inv(S_t_plus)*Ad + Q);
S_t_minus = Ad'*S_t_plus*Ad;


% Set Qf = to S_t_minus
Qf = S_t_minus;

% Check for convergence (FILL ME IN)
if all(all(abs(Qf - QfV) < 1e-4))
    converged = true;
end


end

Qf_converged = Qf

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Setup tvlqr controller and simulate from x0

% Extend nominal limit cycle a little bit (see footnote at the bottom for
% an explanation of this)
tExtended = 1.0;
xtrajExtended = p.modes{1}.simulate([0 tExtended],x0);
utrajExtended = PPTrajectory(spline(linspace(0,tExtended,100),zeros(1,100)));
xtrajExtended = xtrajExtended.setOutputFrame(p.modes{1}.getStateFrame);
utrajExtended = utrajExtended.setOutputFrame(p.modes{1}.getInputFrame);
```

```
[tv,V] = tvlqr(p.modes{1},xtrajExtended,utrajExtended,Q,R,Qf,options);

% Set frames of tvlqr controller
tv = tv.inOutputFrame(p.getInputFrame);
tv = tv.inInputFrame(p.getOutputFrame);

pmodel = SimulinkModel(p.getModel());

tv = tv.setInputFrame(pmodel.getOutputFrame);
tv = tv.setOutputFrame(pmodel.getInputFrame);

% Closed loop system (i.e., feedback system with TVLQR controller)
sysClosedLoop = feedback(pmodel,tv);

% Visualizer
v = CompassGaitVisualizer(p);

%% Simulate from x0 (syntax is useful for part (b)) %%%%%%%%%%%%%%%%%%%%%%%
xtrajSim = sysClosedLoop.simulate([0 1.0], [1;xtraj.eval(0)]);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Play trajectory back
xtrajSim = xtrajSim.setOutputFrame(p.getOutputFrame);
v.playback(xtrajSim);

%% Footnote (feel free to ignore) %%
% Since the continuous portion of our nominal trajectory is only defined for a finite amount of
% [0,ts(end)], we need to deal with cases where the compass gait doesn't
% make contact with the ground before ts(end). Our solution here is to
% extend the nominal trajectory (xtrajExtended) a little beyond ts(end) (by just simulating
% the passive system forwards for longer). This is a reasonable thing to do,
% but still somewhat of a hack. However, this is an important
% implementation issue and worth thinking about in practice.
%%
```

Run Code

Submit You have used 0 of 3 attempts

Answers are displayed within the problem

Controlling the Compass Gait (Part II)

0.0/7.0 points (ungraded)

(b) Next, implement a function similar to the function strideFunction in the previous problem in order to simulate the closed-loop system for one step. The code we have provided you in stabilizeCompassGait.m shows you how to simulate the closed loop system ("sysClosedLoop in the code"). Use this to compute the Floquet multipliers for the closed-loop system just as you did in the previous problem for the passive system. Type in your answer below as a 4×1 vector of complex numbers. Don't worry about ordering the multipliers in any particular order. **Our tolerance for this answer is 0.05 (in terms of the magnitude of each element in the vector).** (Compare the largest Floquet multiplier with the largest multiplier you obtained for the passive system in part (d) of the previous problem. Is it smaller or bigger in magnitude? Is this what you expected?)

```
f = ;

```

Unanswered

```
% We can compute the monodromy matrix and Floquet multipliers just like the last problem.
% The monodromy matrix is:
A = [0.1281    1.7540   -0.0694    0.5891
    -0.1281   -1.7540    0.0694   -0.5891
    -1.5076   -2.9420    0.2520   -1.2411
    -0.2308    2.6514   -0.0817    0.9266];

% The Floquet multipliers are:
f = [-0.4729 + 0.0000i
      0.0128 + 0.1589i
      0.0128 - 0.1589i
     -0.0000 + 0.0000i];
```

Run Code

Submit You have used 0 of 3 attempts

Answers are displayed within the problem