

On a 2D Quadrotor with a load attached through a Chord control problem: Window-passing

Gabriel Enrique García Chávez

In this control problem we have a 2D Quadrotor with a load attached through a chord, which is a Hybrid System. The system is very similar to a 2D Quadrotor with a pendulum, with the difference that, when the inner tension forces inside the “pendulum” (chord in this case) are negative, the system will behave in a very different way. In fact, the load will be “detached” from the quadrotor and will behave as a free-falling ball, while the quadrotor will behave as a single and simple Quad. This behavior will take place as long as the distance from the load to the quad is less than l_0 , the length of the chord. When the equality takes place, the instantaneous tension of the chord, will be modeled as a common impact between objects. If the collision is inelastic, the two bodies will behave as a quad with pendulum again, this is the first mode of the hybrid system. If the collision is elastic, the load will bounce, going back to the second mode (load “detached” of “flying”), but with a reset map applied.

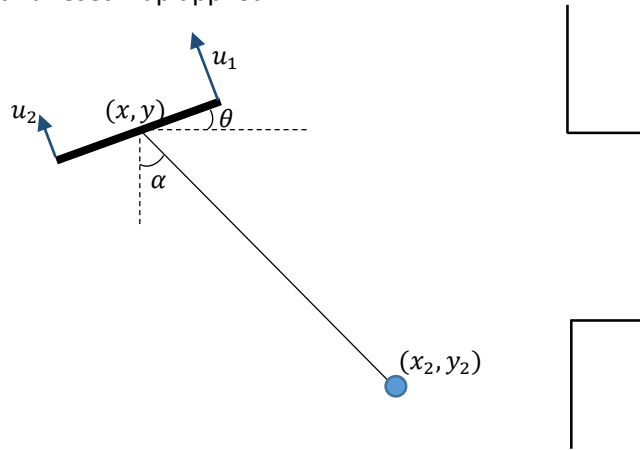


Figure 1. Hybrid System Quadrotor-Chord-Load and window

Let's see some variables:

m_1 : Mass of the Quadrotor

m_2 : Mass of the Load

u_i : Forces from Thrusters.

α : Angle of chord (only when it's tensed)

θ : Angle of Quadrotor

l : Length of the chord

a : Height of the window

d : Distance from center of mass to each thruster.

(x, y) : Position of Quadrotor,

(x_2, y_2) : Position of Load (only when it's flying, although it can be computed when chord is tensed)

*Note: we are supposing that chord is attached at center of mass of the quadcopter

The window has a height of a , with $a < l$, so we have now the control problem: Let's make the quadrotor cross the window in order to take the system to the other side of the window.

In fact the robot must throw the ball and instantaneously follow it, keeping the distance between them less than l , until they cross the window, after that, the robot must "recover" the ball, so there will be an impact produced by the chord.

In the rest of the work, we will consider for impact, inelastic collision and that mass of load is negligible respect to quadcopter. "QuadLoad_e" is the system considering elastic collision (so there is a new variable e , coefficient of restitution).

Let's start. Let's see the first mode of the hybrid dynamical system. It is when we have the chord tensed. This is called "QuadLoadAsOne".

$$\begin{aligned}\ddot{x} &= -\frac{\left(\left(2m_1 + \frac{3m_2}{2}\right)\sin(\theta) - \frac{m_2}{2}\sin(2\alpha - \theta)\right)(u_1 + u_2) - (m_2 + 2m_1)m_2l\sin(\alpha)\dot{\alpha}^2}{(m_1 + m_2)(2m_1 + m_2)} \\ \ddot{y} &= -g - \frac{-\left(\left(2m_1 + \frac{3m_2}{2}\right)\cos(\theta) - \frac{m_2}{2}\cos(2\alpha - \theta)\right)(u_1 + u_2) + (m_2 + 2m_1)m_2l\cos(\alpha)\dot{\alpha}^2}{(m_1 + m_2)(2m_1 + m_2)} \\ \ddot{\theta} &= \frac{d}{I_1}(u_1 - u_2) \\ \ddot{\alpha} &= -\frac{1}{l(2m_1 + m_2)}\sin(\alpha - \theta)(u_1 + u_2)\end{aligned}$$

We will assume that in this case, the chord will behaves as a rod, or as a link, making all the system similar to a double Pendulum. The angle of the chord and the vertical corresponds to α . I've obtained these expression by using the first part of the script ExtractDynamics.m. Order of the system is 8 (counting velocities).

Let's see the dynamical system when we don't have the chord tensed. This is called "QuadLoadFlight". In this case, the load simply behaves as a ball in free fall. We no longer will use α but we'll use the coordinates of ball in plane (x_2, y_2) . Order of the system is 10 (counting velocities).

$$\begin{aligned}\ddot{x} &= -\frac{1}{m_1}\sin\theta(u_1 + u_2) \\ \ddot{y} &= \frac{1}{m_1}(-m_1g + \cos\theta(u_1 + u_2)) \\ \ddot{\theta} &= \frac{d}{I_1}(u_1 - u_2) \\ \ddot{x}_2 &= 0 \\ \ddot{y}_2 &= -g\end{aligned}$$

We are using in total 12 variables. It is possible to replace α by x_2 and y_2 in the first mode, but there is a problem: The dynamics are even messier than they are currently. In a particular, in order to performance Trajectory Optimization, the solvers many times require the gradient of the guards, transition and dynamics of a hybrid system. So we need to compute those gradient, which will be considerable bigger if we don't use α .

From mode 1 to 2:

Guards:

$$h_{12} = \frac{m_2}{m_1 + m_2} (m_1 l \dot{\alpha}^2 + \cos(\alpha - \theta)(u_1 + u_2)) < 0 \rightarrow \text{Change}$$

Transition Dynamics:

$$\begin{aligned} x_2 &= x + l \sin \alpha \\ y_2 &= y - l \cos \alpha \\ \alpha &= 0 \\ \dot{\alpha} &= 0 \end{aligned}$$

Rest of variables are the same.

From mode 2 to 1:

Guards:

$$h_{21} = \sqrt{(x_2 - x)^2 + (y_2 - y)^2} - l \geq 0 \wedge \frac{dh_{21}}{dt} > 0 \rightarrow \text{Change}$$

Transition Dynamics:

$$\begin{aligned} \alpha &= \frac{\pi}{2} + \text{atan2}(y_2 - y, x_2 - x) \\ \dot{\alpha} &= \frac{1}{l} ((\dot{x}_2 - \dot{x}) * \cos \alpha + (\dot{y}_2 - \dot{y}) * \sin \alpha) \alpha = 0 \\ x_2 &= 0 \\ y_2 &= 0 \\ \dot{x}_2 &= 0 \\ \dot{y}_2 &= 0 \end{aligned}$$

Rest of variables are the same.

*Let's note that we are simplifying the collision, assuming mass negligible for the load respect to the quadrotor, and also inelastic collision. If we want to have a more realistic transition, we can use the following dynamics (current only in QuadLoad_e)

Guards:

$$h_{21} = \sqrt{(x_2 - x)^2 + (y_2 - y)^2} - l \geq 0 \wedge v_{trig} > \frac{dh_{21}}{dt} > 0 \rightarrow \text{Change}$$

Transition Dynamics (Basically $e = 0$):

$$\begin{aligned} \alpha &= \frac{\pi}{2} + \text{atan2}(y_2 - y, x_2 - x) \\ v_{2i} &= \dot{x}_2 \sin(\alpha) - \dot{y}_2 \cos(\alpha) \\ v_{1i} &= \dot{x} \sin(\alpha) - \dot{y} \cos(\alpha) \end{aligned}$$

$$v2f = \frac{(m2 * v2i + m * v1i)}{m + m2}$$

$$v1f = v2f$$

$$v2ort = \dot{x}_2 \cos(\alpha) + \dot{y}_2 \sin(\alpha)$$

$$v1ort = \dot{x} \cos(\alpha) + \dot{y} \sin(\alpha)$$

$$\dot{x}_2 = v2ort \cos(\alpha) + v2f \sin(\alpha) \text{ (temporal)}$$

$$\dot{y}_2 = v2ort \sin(\alpha) - v2f \cos(\alpha) \text{ (temporal)}$$

$$\dot{x} = v1ort \cos(\alpha) + v1f \sin(\alpha)$$

$$\dot{y} = v1ort \sin(\alpha) - v1f \cos(\alpha)$$

$$\dot{\alpha} = \frac{1}{l} ((\dot{x}_2 - \dot{x}) * \cos \alpha + (\dot{y}_2 - \dot{y}) * \sin \alpha) \alpha = 0$$

$$x_2 = 0$$

$$y_2 = 0$$

$$\dot{x}_2 = 0$$

$$y_2 = 0$$

Rest of variables are the same.

*Note that this transition dynamics avoids Zero Phenomenon by setting $e = 0$ at some point of QuadLoad_e, this will be triggered only when $v_{trig} > \frac{dh_{21}}{dt}$, i.e. when the velocity in the collision action is really low.

From mode 2 to 2 (only for QuadLoad_e):

Guards:

$$h_{22} = \sqrt{(x_2 - x)^2 + (y_2 - y)^2} - l \geq 0 \wedge \frac{dh_{22}}{dt} > v_{trig} \rightarrow \text{Change}$$

Transition Dynamics:

$$\alpha = \frac{\pi}{2} + \text{atan2}(y_2 - y, x_2 - x) \text{ (temporal)}$$

$$v2i = \dot{x}_2 \sin(\alpha) - \dot{y}_2 \cos(\alpha)$$

$$v1i = \dot{x} \sin(\alpha) - \dot{y} \cos(\alpha)$$

$$v2f = \frac{(m2 * v2i + m * v1i) - m * e * (v2i - v1i)}{m + m2}$$

$$v1f = v2f + e * (v2i - v1i)$$

$$v2ort = \dot{x}_2 \cos(\alpha) + \dot{y}_2 \sin(\alpha)$$

$$v1ort = \dot{x} \cos(\alpha) + \dot{y} \sin(\alpha)$$

$$\dot{x}_2 = v2ort \cos(\alpha) + v2f \sin(\alpha)$$

$$\dot{y}_2 = v2ort \sin(\alpha) - v2f \cos(\alpha)$$

$$\dot{x} = v1ort \cos(\alpha) + v1f \sin(\alpha)$$

$$\dot{y} = v1ort \sin(\alpha) - v1f \cos(\alpha)$$

$$\alpha = 0$$

The configuration is defined in file “runDircolQuadLoad2I_wdw.m”. We are running a mode sequence as: AsOne-Flight-AsOne. We are also defining the window not as the square that is showed in the first image, because we want it to have derivative for the solvers (corners has no derivative). So we define the function

$$f(x,y) = \left(\frac{y}{0.5}\right)^2 - \left(\frac{x-2}{0.2}\right)^4 - 1$$

When $f(x,y) = 0$, we have the corner of the new window. $f(x,y) < 0$ is outside of window and $f(x,y) > 0$ is forbidden because it's inside. We want this for (x,y) being coordinates of the center of the robot, the thruster coordinates, and the Load coordinates. Note that the height of the system in the program is 1 and the length of the chord is 2.

We define the cost function as a LQR also.

Running “runDircolQuadLoad2I_wdw.m” we will obtain ideally a local optimal trajectory. We use a Direct Collocation program from Hybrid Trajectory Optimization. The important here is the control signal *utraj*. You can save this and see its performance by running “simnplotwfb.m”.

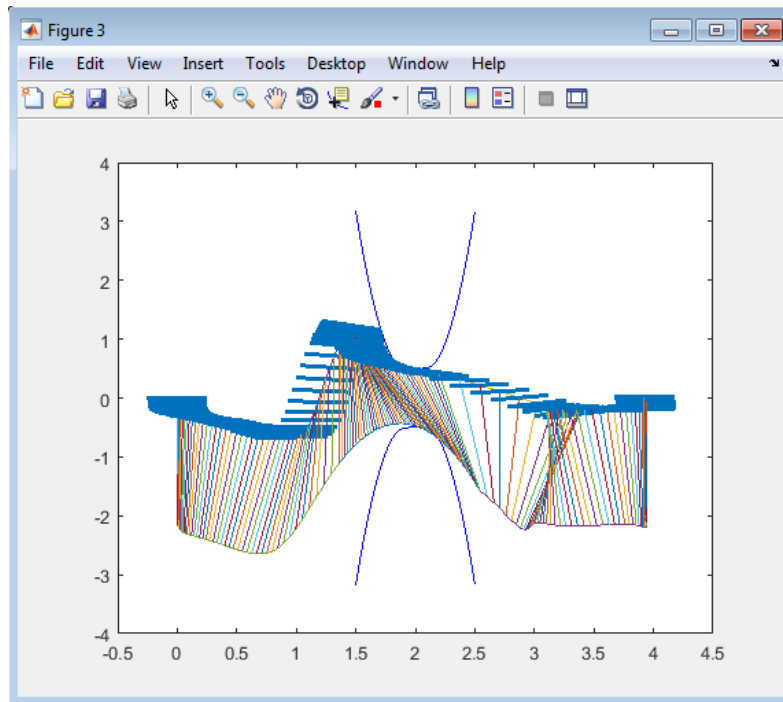


Figure 2. Performance of the control signal – Direct Collocation

Note that in this performance there are some apparent collision, this is because many times in optimal control, the optimal solution is in the boundaries of the restrictions. Also, the solver only cares about the states in collocation point, not the states between them. So one solution is increase the number of collocation points, or better, set the forbidden region with a slack space for safety.

Finally we have a control that leads the system to the other side of the window. But this is an optimal trajectory for a specific initial condition. So we need to make it robust to possible slightly different initial conditions. In order to do that, we can make an TV-LQR, but with some modifications because this is an hybrid trajectory.

We are going to perform TV-LQR on each mode, and we are going to propagate the cost related quadratic function $S(t)$ backwards with the Riccati Differential Equation, but in the transitions, we are going to make a jump involving the Linearization of the transition dynamics at that point:

$$S(t^-) = A_d^T S(t^+) A_d$$

And everywhere else we will run the classic Riccati Differential Equation:

$$-\dot{S}(t) = Q - S(t)B(t)R^{-1}B(t)^TS(t) + S(t)A(t) + A(t)^TS(t)$$

With final condition $S(t_f) = Q_f$, the penalization matrix to the final states.

This is called in Underactuated Robotics 6.832x from edX the “Jump Riccati Equation”. After these we will get the optimal control signal:

$$\bar{u} = -K(t)\bar{x} + y_0(t)$$

Where $y_0(t)$ involves very little terms related with numerical issues of linearization and integration. So:

$$u - u_0(t) = -K(t)(x - x_0(t)) + y_0(t)$$

We have finally the control signal:

$$u = -K(t)x + K(t)x_0(t) + u_0(t) + y_0(t)$$

This control signal is applied directly to the hybrid system.

We have a good performance for the feedback control when a perturbation is applied at the beginning.

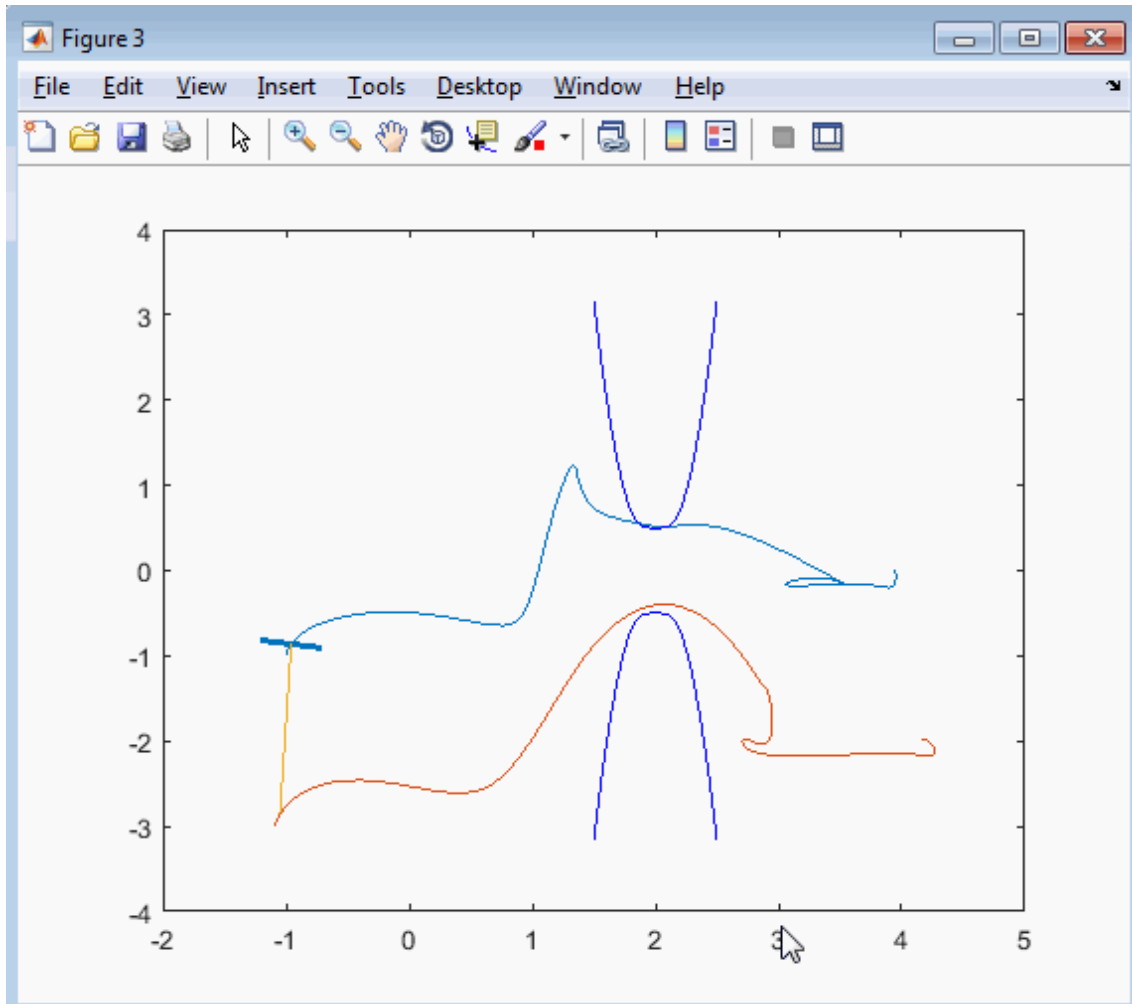


Figure 3. Performance of the control signal Direct Collocation + TVLQR for Robustness

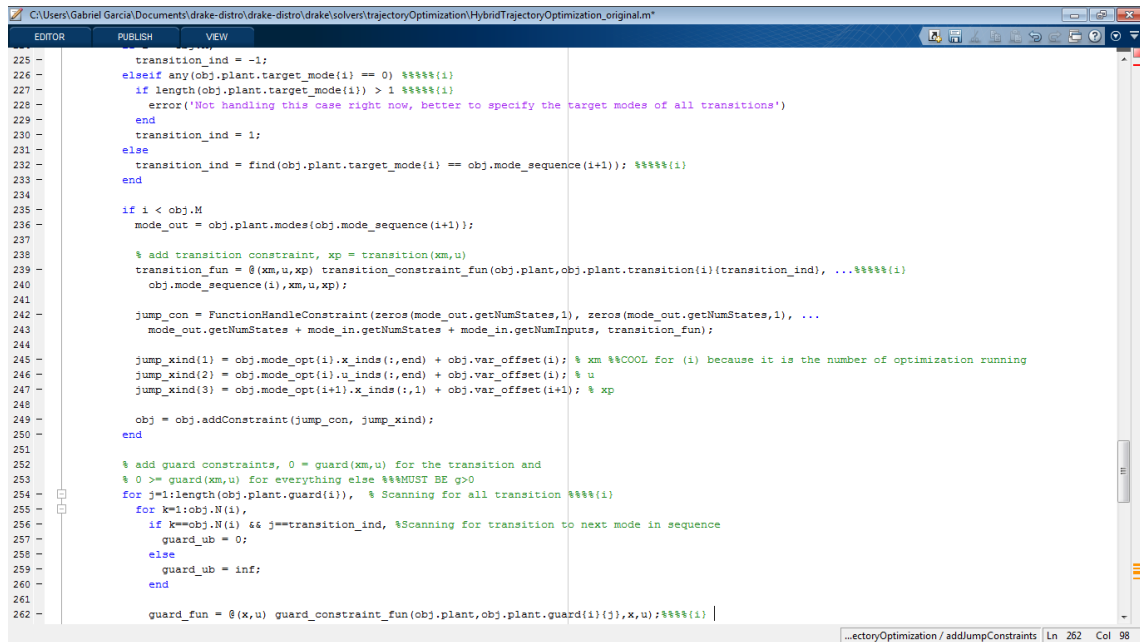
Currently we are also working in the transition issues that can arise when the guard is triggered before or after the time when it was triggered in the original computed trajectory.

Changes in source code:

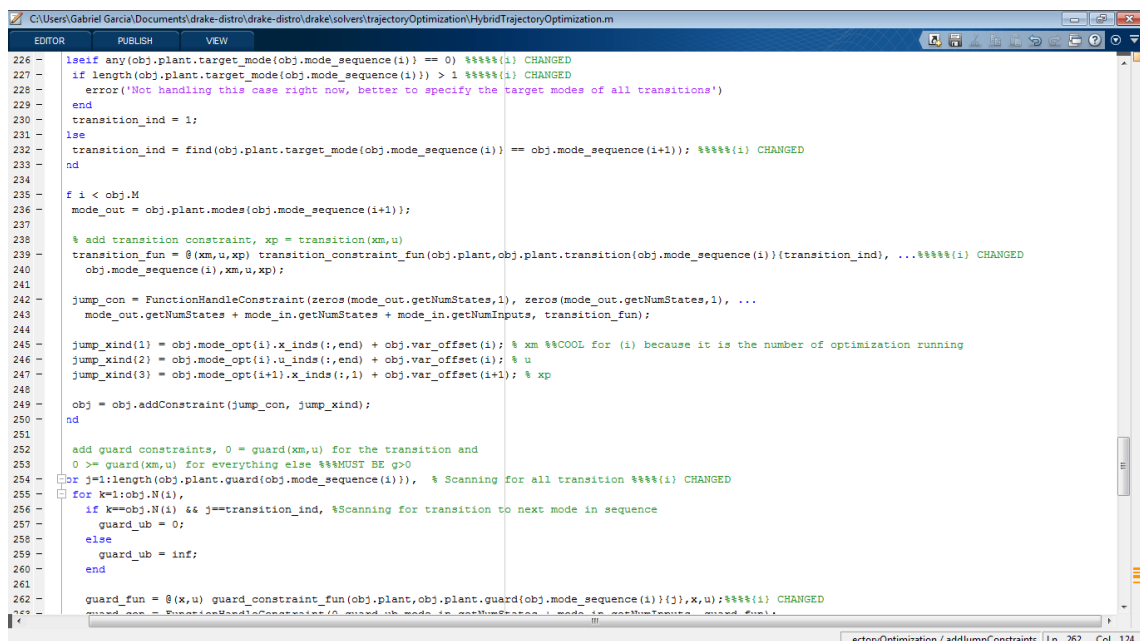
- Fixed a bug where the optimization confuses the i -th index for mode of the plant with the correct i -th index from the mode sequence.

In:
drake\drake\solvers\trajectoryOptimization\HybridTrajectoryOptimization.m

Change $i \rightarrow \text{obj.mode_sequence}(i)$ in specific places



```
225 - transition_ind = -1;
226 - elseif any(obj.plant.target_mode(i) == 0) %%%%{i}
227 - if length(obj.plant.target_mode(i)) > 1 %%%%{i}
228 - error('Not handling this case right now, better to specify the target modes of all transitions')
229 - end
230 - transition_ind = 1;
231 - else
232 - transition_ind = find(obj.plant.target_mode(i) == obj.mode_sequence(i+1)); %%%%{i}
233 - end
234 -
235 - if i < obj.M
236 - mode_out = obj.plant.modes(obj.mode_sequence(i+1));
237 -
238 - % add transition constraint, xp = transition(xm,u)
239 - transition_fun = @(xm,u,xp) transition_constraint_fun(obj.plant,obj.plant.transition(i){transition_ind}, ...%%%%{i}
240 - obj.mode_sequence(i),xm,u,xp);
241 -
242 - jump_con = FunctionHandleConstraint(zeros(mode_out.getNumStates,1), zeros(mode_out.getNumStates,1), ...
243 - mode_out.getNumStates + mode_in.getNumStates + mode_in.getNumInputs, transition_fun);
244 -
245 - jump_xind(1) = obj.mode_opt(i).x_inds(:,end) + obj.var_offset(i); % xm %%COOL for (i) because it is the number of optimization running
246 - jump_xind(2) = obj.mode_opt(i).u_inds(:,end) + obj.var_offset(i); % u
247 - jump_xind(3) = obj.mode_opt(i+1).x_inds(:,1) + obj.var_offset(i+1); % xp
248 -
249 - obj = obj.addConstraint(jump_con, jump_xind);
250 - end
251 -
252 - % add guard constraints, 0 = guard(xm,u) for the transition and
253 - % 0 >= guard(xm,u) for everything else %%%MUST BE q>0
254 - for j=1:length(obj.plant.guard(i)), % Scanning for all transition %%%%{i}
255 - for k=obj.N(i),
256 - if k==obj.N(i) & j==transition_ind, %Scanning for transition to next mode in sequence
257 - guard_ub = 0;
258 - else
259 - guard_ub = inf;
260 - end
261 -
262 - guard_fun = @(x,u) guard_constraint_fun(obj.plant,obj.plant.guard(i){j},x,u); %%%%{i}
```

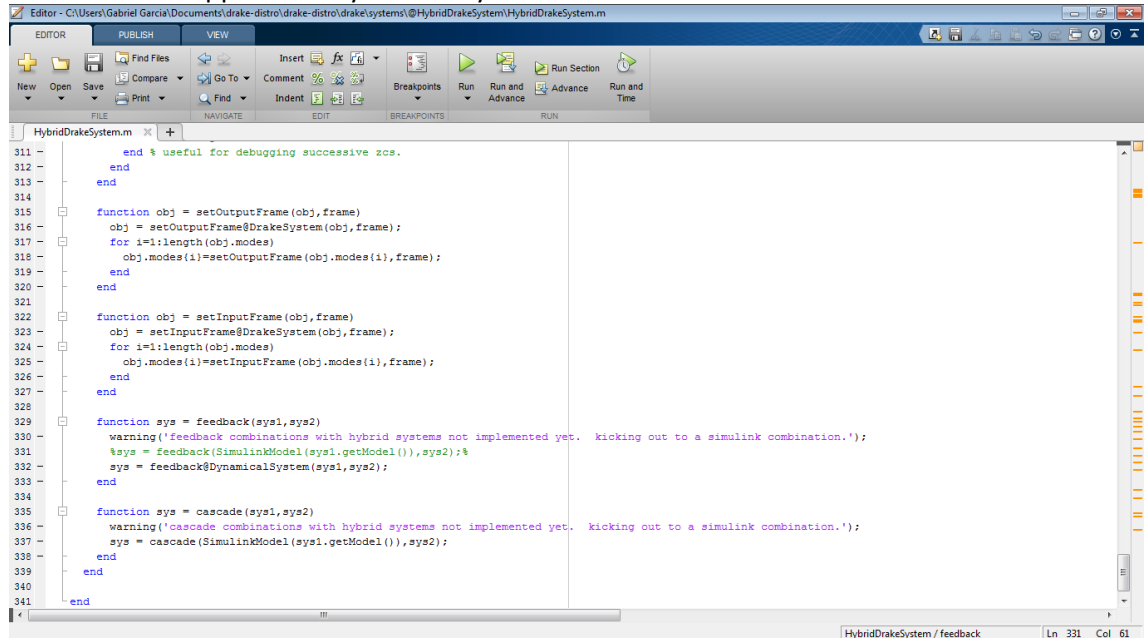


```
226 - %elseif any(obj.plant.target_mode(obj.mode_sequence(i)) == 0) %%%%{i} CHANGED
227 - if length(obj.plant.target_mode(obj.mode_sequence(i))) > 1 %%%%{i} CHANGED
228 - error('Not handling this case right now, better to specify the target modes of all transitions')
229 - end
230 - transition_ind = 1;
231 - else
232 - transition_ind = find(obj.plant.target_mode(obj.mode_sequence(i)) == obj.mode_sequence(i+1)); %%%%{i} CHANGED
233 - end
234 -
235 - if i < obj.M
236 - mode_out = obj.plant.modes(obj.mode_sequence(i+1));
237 -
238 - % add transition constraint, xp = transition(xm,u)
239 - transition_fun = @(xm,u,xp) transition_constraint_fun(obj.plant,obj.plant.transition(obj.mode_sequence(i)){transition_ind}, ...%%%%{i} CHANGED
240 - obj.mode_sequence(i),xm,u,xp);
241 -
242 - jump_con = FunctionHandleConstraint(zeros(mode_out.getNumStates,1), zeros(mode_out.getNumStates,1), ...
243 - mode_out.getNumStates + mode_in.getNumStates + mode_in.getNumInputs, transition_fun);
244 -
245 - jump_xind(1) = obj.mode_opt(i).x_inds(:,end) + obj.var_offset(i); % xm %%COOL for (i) because it is the number of optimization running
246 - jump_xind(2) = obj.mode_opt(i).u_inds(:,end) + obj.var_offset(i); % u
247 - jump_xind(3) = obj.mode_opt(i+1).x_inds(:,1) + obj.var_offset(i+1); % xp
248 -
249 - obj = obj.addConstraint(jump_con, jump_xind);
250 - end
251 -
252 - % add guard constraints, 0 = guard(xm,u) for the transition and
253 - % 0 >= guard(xm,u) for everything else %%%MUST BE q>0
254 - for j=1:length(obj.plant.guard(obj.mode_sequence(i))), % Scanning for all transition %%%%{i} CHANGED
255 - for k=obj.N(i),
256 - if k==obj.N(i) & j==transition_ind, %Scanning for transition to next mode in sequence
257 - guard_ub = 0;
258 - else
259 - guard_ub = inf;
260 - end
261 -
262 - guard_fun = @(x,u) guard_constraint_fun(obj.plant,obj.plant.guard(obj.mode_sequence(i)){j},x,u); %%%%{i} CHANGED
263 - guard_con = FunctionHandleConstraint(0,guard_ub,mode_in.getNumStates + mode_in.getNumInputs, guard_fun);
```

- Changed a line in feedback function of Hybrid systems.

In: "drake-distro\drake\systems\@HybridDrakeSystem\HybridDrakeSystem.m"

Commented line is the original. Problem with using SimulinkModel is that this produces a lost in the Coordinate Frame of system 1. So when using feedback from an upper class, there is an error from mismatches of Coordinate Frames. It's better to use directly the feedback from the upper class Dynamical Systems.



- Added some lines in evaluation function of Hybrid trajectories.

In: "drake-distro\drake-distro\drake\systems\trajectories\HybridTrajectory.m"

When performing the feedback between the TLVQR signal and the Hybrid System, sometimes the feedback function will ask for evaluation to points that may not be defined in the TLVQR signal. This is a minimum observation, anyway take this with precaution, it can affect other programs involving feedback and Hybrid Trajectories. Let's set:

```

if any(t<obj.tspan(1))
    t(find(t<obj.tspan(1)))=obj.tspan(1);
    warning('before tspan')
end
if any(t>obj.tspan(end))
    t(find(t>obj.tspan(end)))=obj.tspan(end);
    warning('after tspan')
end
```


Editor - C:\Users\Gabriel Garcia\Documents\drake-distro\drake\systems\trajectories\HybridTrajectory.m

EDITOR PUBLISH VIEW

New Open Save Find Files Compare Go To Comment % Find Indent Breakpoints Run Run and Advance Run and Time

FILE NAVIGATE EDIT BREAKPOINTS RUN

```
107 /
108 end
109 function y = eval(obj,t)
110 % look into and evaluate correct trajectory based on the time
111 if (any(t<obj.tspan(1)) || any(t>obj.tspan(end)))
112     error('outside tspan');
113 end
114
115 if any(t<obj.tspan(1))
116     t(find(t<obj.tspan(1)))=obj.tspan(1);
117     warning('before tspan')
118 end
119 if any(t>obj.tspan(end))
120     t(find(t>obj.tspan(end)))=obj.tspan(end);
121     warning('after tspan')
122 end
123
124
125
126
127 if (length(t)==1)
128     trajind=find(t>=[obj.tspan(1),obj.te],1,'last');
129     y = obj.traj(trajind).eval(t);
130 else % vectorized version
131     t=sort(t);
132     ind = 1;
133     for i=1:length(obj.te)+1
134         nind = [];
135         if (i<=length(obj.te))
136             nind = find(t(ind:end)>obj.te(i),1,'first'); % first (relative) index beyond this segment
137         end
138         if (isempty(nind)) % the rest must belong to this segment
139             nind = ind;
140         end
141     end
142     y = obj.traj(nind).eval(t);
143 end
```

HybridTrajectory / eval Ln 124 Col 7