CS 4/6545 Autonomous Robotics:
Inverse dynamics control

Submission instructions: Your submission will consist of all code and plots. **You will submit eight plots: one each for sinusoidal tracking and setpoint holding (2), one for before and after inverse dynamics is implemented (2), and one each for joint position and motor torques** ($2 \times 2 \times 2 = 8$)**.**

One important note: *To make things a little tricky, the dynamics derivation we use in this assignment uses $\theta_1 = 0, \theta_2 = 0$ to denote the pendulum at the vertical position. In* GAZEBO*, the same configuration is denoted by $\theta_1 = -\frac{\pi}{2}, \theta_2 = 0$. You should never have to directly account for this difference: if you see an addition of $\frac{\pi}{2}$ to the first joint angle at points in the assignment, this is why.*

1. You will now use two particular versions of `track_sinusoidal.cpp` and `PD_hold.cpp`, modified from Assignment #1 to use low PD gains. `track_sinusoidal.cpp` has the double pendulum track the sinusoidal trajectory $\theta_1 = \sin t, \theta_2 = \sin 3t$. Plot desired and actual joint angles over ten seconds for this controller.

   ```
   inverse_dynamics $ cd build
   build $ make
   build $ cd ..
   inverse_dynamics $ gazebo --verbose track_sinusoidal.world
   inverse_dynamics $ python plot_track_sinusoidal.py
   inverse_dynamics $ python plot_track_sinusoidal_torques.py
   ```

2. `PD_hold.cpp` controls the double pendulum to go to the setpoint $\theta_1 = \frac{\pi}{2}, \theta_2 = 0$. Plot desired and actual joint angles over ten seconds for this controller.

   ```
   inverse_dynamics $ cd build
   build $ make
   build $ cd ..
   inverse_dynamics $ gazebo --verbose PD_hold.world
   inverse_dynamics $ python plot_PD_hold.py
   inverse_dynamics $ python plot_PD_hold_torques.py
   ```

3. Using MATHEMATICA, we will now go through computing the dynamics equations for the double pendulum. I will show you multiple ways of doing the same thing, when possible, so that you can pickup enough MATHEMATICA syntax. One quirk about MATHEMATICA: it does not evaluate anything until you press `Shift-Enter`.

   (a) Compute U_1, the potential energy for the first link of the pendulum. You can set this up multiple ways in MATHEMATICA. For instance, you could say:

   ```
   U1 = -m1*g*l1*Cos[theta1]
   ```

   You could also setup a frame at the end of the link and get MATHEMATICA to compute the endpoint for you:

   ```
   f_0T0x = {{Cos[theta1], -Sin[theta1], 0}, {Sin[theta1], Cos[theta1], 0}, {0, 0, 1}}
   f_0xT1 = {{1, 0, 0}, {0, 1, -l1}, {0, 0, 1}}
   o1 = {{0}, {0}, {0}}
   p1 = f_0T0x . f_0xT1 . o1
   ```

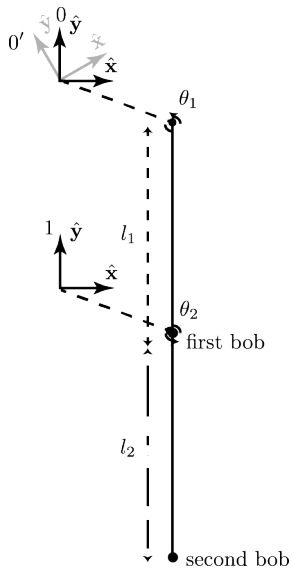

Figure 1: *Depiction of the double pendulum used in the derivation. We assume that all mass is concentrated at the distal end of each link. The first joint of the pendulum is located at* $(0, 0, 0)$.

where `f_0T0x` and `f_0xT1` are defined as ${}_0T_{0'}$ and ${}_{0'}T_1$ (using Figure 1), and $o1$ is defined as the origin of Frame 1. Then, compute the potential energy using the vertical position of the link endpoint:

```
U1 = p1[[2]]*m1*g
```

(b) Compute U_2 similarly. This calculation will obviously be more involved. **(Undergraduates can ask me for my MATHEMATICA commands to compute U_1 and U_2).**

(c) Compute T_1. This is much easier:

```
T1 = 1/2 * m1 * theta1'[t]^2
```

The `theta1'[t]` statement tells `Mathematica` that θ is a function of time and the apostrophe tells `Mathematica` that we want the derivative of this function (*i.e.*, we want $\dot{\theta}_1$).

(d) Compute T_2. This is still easy.

(e) Now compute the Lagrangian. The potential energy of the double pendulum is the sum of the potential energies of all of its links. The kinetic energy of the double pendulum is the sum of the kinetic energies of all of its links:

```
L = T1 + T2 - U1 - U2
```

(f) Here is where MATHEMATICA shines. The complicated statement below tells MATHEMATICA that we want to differentiate the Lagrangian with respect to $\dot{\theta}_1$ first (`D[L, theta1'[t]]`) and then we want to differentiate *that* with respect to time (`D[...], t]`). The next statement (`D[L, theta1[t]]`) tells MATHEMATICA that we want to differentiate the Lagrangian with respect to the θ_1. The next `D[...]` statements function identically.

After MATHEMATICA has differentiated these functions, a (2×2) system of linear equations remains. We want to solve this system for $\ddot{\theta}_1$ and $\ddot{\theta}_2$. That is what the statement `Solve[tau1 == ..., tau2 == ..., theta1''[t], theta2''[t]` does. Finally, the `Simplify[...]` statement applies some clever substitutions to reduce the resulting formulae to a much more manageable form.

```
Simplify[Solve[{tau1 == D[D[L, theta1'[t]], t] - D[L, theta1[t]],
  tau2 == D[D[L, theta2'[t]], t] - D[L, theta2[t]], {theta1''[t], theta2''[t]}]]
```

(g) Plug some values in for $\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2$ for which you should be able to know answers immediately. Do the values output for $\ddot{\theta}_1$ and $\ddot{\theta}_2$ check out?

(h) You now have formulae in terms of $\ddot{\theta}_1$ and $\ddot{\theta}_2$. Now modify the MATHEMATICA command above to solve for `tau1` and `tau2` instead.

(i) Finally, take the result of this last step and put it into C++ (actually C) code. Do this using:

```
CForm[%]
```

The % tells `Mathematica` to use the result of the last step. Finally, embed this code in your program, by putting it into the function I've made for you in `inverse_dynamics.cpp`. You'll need to clean this code up a little from what MATHEMATICA gave you; for example, remove extraneous braces, change `Cos(.)` to `cos(.)`, *etc.*

4. Implement the composite controller by coding the equations for τ_1 and τ_2 into `track_sinusoidal.cpp` and `hold_PD.cpp`. Build the plugins *without retuning PID/PD gains*.

5. Plot desired and actual joint angles *again* over ten seconds for both plugins (saving your last plots). Also plot motor torques again. If your feedback motor torques are not small compared to the inverse dynamics torques, you have a bug in your code.