CS 294-40, Fall 2008 Problem Set #2: Autonomous Helicopter Flight and Quadruped Locomotion

Due 11am on October 21.

NOTE:

- Please refer to the class webpage (http://inst.eecs.berkeley.edu/˜cs294-40/fa08/) for the homework policy.

1 Autonomous Helicopter Flight [18 + 6 points]

Questions 1-5 are all formulated inside the matlab m-files that can be downloaded from the class webpage—namely, the m-files Q1.m, Q2.m, Q3.m, Q4.m, Q5.m. The code is commented such as to be self-contained, but if you have any questions, please do not hesitate to send me email or stop by in office hours.

The helicopter model used is of the following form: The state is represented by the position (n, e, d), orientation (quaternion q), velocity $(\dot{n}, \dot{e}, \dot{d})$, angular rate ($(p, q, r)$, in the helicopter's coordinate frame). For control purposes, we also augment the state with the past control inputs, and the past change in control inputs. We treat the change in control input as the control input in our model.

The inputs to control a helicopter are:

- u_1 and u_2: The latitudinal (left-right) and longitudinal (front-back) cyclic pitch controls. They are also often referred to by elevator and aileron. They change the pitch angle of the main rotor throughout each cycle and can thereby cause the helicopter to pitch forward/backwards or to roll sideways. By pitching and rolling the helicopter, the pilot can affect the direction of the main thrust, and hence the direction in which the helicopter moves.
- u_3: The tail rotor collective pitch control. This control is also often referred to by rudder and it affects the tail rotor thrust. It can be used to yaw (turn) the helicopter.
- u_4: The main rotor collective pitch control, similarly to the cyclic controls, causes the main rotor blades to rotate along an axis that turns along the length of the rotor blades, and thereby affects the angle at which the main rotor's blades are tilted relative to the plane of rotation. As the main rotor blades sweep through the air, they generate an amount of upward thrust that (generally) increases with this angle. By varying the collective pitch angle, we can affect the main rotor's thrust. For inverted flight, by setting a negative collective pitch angle, we can cause the helicopter to produce negative thrust.

We let F_x, F_y, F_z denote the forces working on the helicopter along the helicopters forward (x) axis, its sideways to the right (y) axis, and its downward (z) axis. Similarly, we let T_x, T_y, T_z denote the torques working on the helicopter along each of these axes. We let (u, v, w) denote the velocity of the helicopter in the helicopter's coordinate frame. Then we model the helicopter dynamics as follows:

$$
\begin{aligned}
F_x &= G_x + C_x * [u] \\
F_y &= G_y + C_y * [1; v] \\
F_z &= G_z + C_z * [1; w; u_4] \\
T_x &= C_p * [1; p; u_1] \\
T_y &= C_q * [1; q; u_2] \\
T_z &= C_r * [1; r; u_3]
\end{aligned}
$$

Here (G_x, G_y, G_z) denotes gravity in the frame of the helicopter; $C.$ are vectors that parameterize the helicopter model. These parameters are instantiated in the code and have been determined by fitting them to flight data.

From forces and torques we find linear and angular acceletations, which in turn we can integrate over time to obtain the helicopter's state. The files: heli.m and init_setup.m contain the implementation and according comments.

1. **[6 points] Autonomous Helicopter Flight: Hover with LQR**
2. **[3 points] Autonomous Helicopter Flight: Hover with DDP**
3. **[3 points] Autonomous Helicopter Flight: Trajectory with DDP**
4. **[6 points] Autonomous Helicopter Flight: Aerobatic Funnel with DDP**
5. **[6 points] Autonomous Helicopter Flight: Helicopter with a Load**[Challenge problem—extra credit.]

2 Value Iteration for Quadruped Footstep Planning [18 points]

The code referred to below is available from the class website.

In this problem you'll implement value iteration to plan footsteps for a quadruped robot, similar to the setup discussed in the quadruped lecture. The `footstep_planner.cpp` file in the `code/` directory contains the starter code for implementing value iteration. You need to add your own code in the section labelled

```
////////////////////////////////////////////////////////////////////////
// YOUR CODE HERE
////////////////////////////////////////////////////////////////////////
```

While there is quite a bit of starter code that sets up the footstep planning problem, you only need to use a few of the variables:

- The `num_states` variable is an integer listing the number of (valid) states.
- The `rewards` variable is an array of doubles, with `num_states` elements, that indicates the reward for each state. As discussed in class, this reward is negative everywhere, and zero at the goal state.

- The `values` variable is also an array of doubles, with `num_states` elements, that you will fill in with the value function by performing value iteration. It is initialized with all values set to negative infinity, except the goal state which had value 0.
- The `num_transitions` and `transitions` arrays specify the valid transitions for the MDP. In particular, `num_transitions[i]` indicates the number of valid transitions from the ith state, and `transitions[i]` is itself an array, with `num_transitions[i]` elements, that lists all the states which can be reached from the ith state.
- The `state_idx_to_feet()` function converts a state index to a list of four foot positions, described as a 2D double array `double feet[4][2]`. In the foot array, `foot[i][0]` is the x position of the ith foot, and `foot[i][1]` is the y position of the ith foot. In addition, the `feet_to_state_idx()` function performs the inverse operation, converts four foot positions to a state index, or returns -1 if those four feet are not a valid state.
- The `start_feet` and `goal_feet` variables are also 2D arrays, as described above, that indicate the start and goal states as foot positions — these can therefore be converted to state indexes via the `feet_to_state_idx()` function.

6. **[15 points] Value iteration for footstep planning**

   Implement value iteration to find the optimal set of footsteps from the start location to the goal. Your code should first perform value iteration until convergence (you can assumed that value iteration has converged if no entry changes by more than 10^-8 after a full sweep through the states), then use this value function to find an optimal sequence of foot steps that begins at the `start_feet` location, and ends at `goal_feet`.

   After finding the optimal footsteps, output these footsteps to a file named `steps.feet` — each line in this file should contain a full set of footsteps in the format:

```
feet[0][0] feet[0][1] feet[1][0] feet[1][1] feet[2][0] feet[2][1] feet[3][0] feet[3][1]
```

   After outputting this file, you can visualize the results with the included Matlab script, `plot_footsteps.m`.

   Hand in: (i) Print-out of relevant part of your code. (ii) Email me your footstep output file as steps.feet.yourname (I will replay with the matlab script you also have available for visualization).

7. **[3 points] Footstep choices**

   In the previous problem, you were allowed to pick footsteps in any order you desired. In practice, quadruped locomotion can often be optimized when the footsteps are always performed in a specific order — for example: back-right, front-right, back-left, front-left. Describe how you would augment the MDP for this problem to ensure that the feet always moved in a specific order (you don't need to implement this part for the problem, just describe the modification).