HW 3: Motion Planning and Constructive Controllability

Due Feb 28, 2020

We dont mind if you work with other students on your homework. However, each student must write up and turn in their own assignment (i.e. no copy & paste). If you worked with other students, please **acknowledge who you worked with** at the top of your homework.

1. **Optimization-Based Motion Planning**

   When designing project 2, we realized that Python lacks user-friendly tools for general nonlinear optimization, so we've decided to have you do that portion of the project in Matlab as part of your homework. Hopefully we'll figure out a good Matlab-Python interface next week so that you can test these plans on hardware.

   In this problem, you'll be using YALMIP, a versitile optimization modeling toolbox for Matlab developed by Johan Löfberg, as well as IPOPT, a powerful open-source nonlinear optimizer developed by Andreas Wächter and Lorenz T. Biegler and maintained by the COIN-OR project.

   Installation instructions for YALMIP are available here, though I highly suggest installing it through tbxmanager.

   In order to use IPOPT, you need to have a set of MEX binaries for the solver. MEX binaries are essentially C++ code compiled to work with Matlab. We've provided a set of mex binaries in `ipopt.zip`, but if those don't work you can potentially try these. Unzip `ipopt.zip` in the same folder as the matlab code for this problem. There's some code at the beginning of `test_bicycle_planning.m` that should correctly add it to Matlab's path. You can test if the mex binary files work with your system by running some of the examples in the ipopt folder (you'll have to add the path first). However don't try to duplicate the code in those examples for this homework. We'll be using the much friendlier YALMIP interface instead.

   Some of the parts of this problem will have an autograder, but the constraint definitions aren't easy to grade autonomously. We'll be looking at the plot of the plan you output and using that to determine the correctness of your constraint formulation.

   For this problem, you'll be converting the path planning problem into the following nonlinear optimization problem:

$$
\begin{aligned}
q^*, u^* = \operatorname{argmin} & \sum_{i=1}^{N} \left(q_i^T Q q_i + u_i^T R u_i \right) + q_{N+1}^T P q_{N+1} \\
s.t.\ & q_i \geq q_{min},\ \forall i \\
& q_i \leq q_{max},\ \forall i \\
& u_i \geq u_{min},\ \forall i \\
& u_i \leq u_{max},\ \forall i \\
& q_{i+1} = F(q_i, u_i),\ \forall i \leq N \\
& (x_i - \text{obs}_{j_x})^2 + (y_i - \text{obs}_{j_y})^2 \geq \text{obs}_{j_r}^2,\ \forall i, j \\
& q_1 = q_{start} \\
& q_{N+1} = q_{goal}
\end{aligned}
$$

Here, we *discretize* the problem over $N+1$ timesteps, with the first indicating the current time, and the last indicating the time at which we intend to reach the goal. q is an array of $N+1$ states (x, y, θ, ϕ) with $q_i = (x_i, y_i, \theta_i, \phi_i)$ for $i = 1, ..., N+1$, and u is an array of N inputs (v, ω). We don't have an input at the last state. obs is an array of obstacles. In order to simplify computation, we assume that all obstacles are circles with center (x, y) and radius r. We also assume that our robot is circular, and that its radius has been incorporated into the radii of all obstacles. $F(q, u)$ is the discrete dynamics of our system, $q_{k+1} = F(q_k, u_k)$. We use a standard quadratic cost function here, though you could choose any appropriate cost function.

Answer the following questions

(a) Our cost function here is quadratic. Why do we need to use a nonlinear optimizer?

(b) Work through the list of constraints and explain what each does in common English. Feel free to use a little math if needed.

(c) In order to do path planning with an optimizer, we need to *discretize* our dynamics. In order to do this, we'll use an Euler (first order) discretization:

$$q(t + \delta t) = q(t) + \dot{q}(t)\delta t$$

While we could use an integral, a first order approximation is accurate if δt is small, and requires much less computation time than an integral. Since the optimizer will be running this on the order of hundreds of thousands of times (or more), speed is very important.

Open `discrete_bicycle_dynamics.m` and implement the discretized dynamics of a bicycle-modeled robot. Then run the autograder to make sure you've gotten it correct.

(d) The seed or initial condition of our optimizer is very important. A good seed can make the optimization faster, and a bad seed can make the optimization much, much slower. Ideally, we would initialize the optimizer with a feasible path to the goal, but we have no way of easily generating one. So we'll be doing the next best thing, and initializing the path as a straight line in configuration (state) space, with evenly distributed waypoints. Even though our system's nonholonomic constraints make this infeasible, the gradients on these constraints are very clear and the solver can pull itself out more easily than other potential initializations.

For the input initialization, we'll set everything to zeros to hopefully bias the solver into finding a local optima with low inputs.

Open `initial_condition.m`, and implement these initializations. The `linspace` and `zeros` commands will likely be useful. Then run the autograder to make sure you've gotten it correct.

(e) We're using a standard quadratic cost function for this problem with three cost matrices P, Q, and R. Q and R collectively make up the *stage cost*, the cost at every timestep. Q penalizes the distance from the goal of the system at every timestep, while R penalizes the input at each timestep. P is called the *terminal cost*, and it penalizes the distance from the goal of the last timestep. This cost isn't actually necessary in our case, since we have a constraint that the last state be the goal. However, sometimes this constraint isn't possible to fulfill (we haven't given the system enough time or the state isn't reachable). In this case, we can remove that constraint and rely on our P cost to get us as close as possible. Usually P is set as Q or some scalar multiple of it.

Open `cost_function.m` and implement the cost function. Then run the autograder to make sure you've gotten it correct.

(f) Finally, we need to program the constraints. YALMIP provides a very intuitive, friendly way to imput constraints into the optimization. Open `bicycle_planning.m` and read over the file to see how we're setting up the optimization problem. Scroll down to where the constraints are defined, and fill in the rest of the constraints. We've defined a couple constraints for you, so hopefully you'll be able to pattern-match the syntax, but feel free to use the YALMIP tutorial if needed. This question is not autograded.

(g) Now we can run the optimization. Run `test_bicycle_planning.m` to run the path planning problem with a couple randomly-defined obstacles. If you've defined your constriants properly, your final path should look something like this (with different colors):

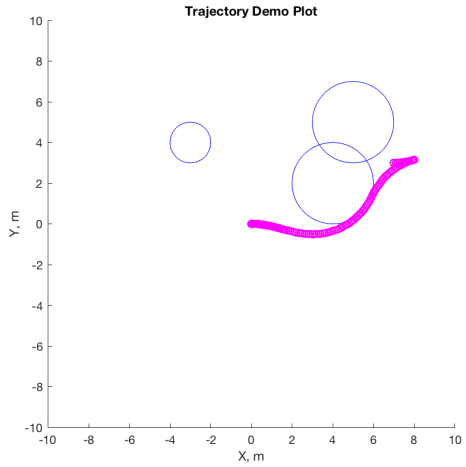

Figure 1: Expected plot of path plan

Submit all three displayed plots to gradescope with the rest of your assignment.

2. **Nonholonomic Kinematics**

(a) Suppose that we have the following model of a turtlebot:

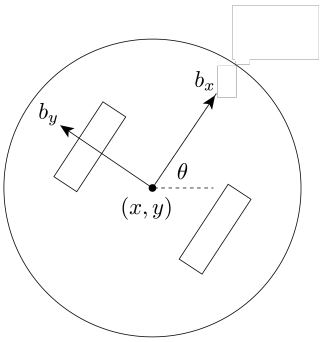

Figure 2: Unicycle Model of a Car

The dynamics of the system are given by:

$$\dot{x} = v\cos(\theta) \tag{1}$$
$$\dot{y} = v\sin(\theta) \tag{2}$$
$$\dot{\theta} = \omega \tag{3}$$

Suppose we can control the speed of the car v and the yaw rate ω.

i. Find the Pfaffian constraints $w(q, \dot{q})\dot{q} = 0$ and rewrite the dynamics in the form of

$$\dot{q} = g_1(q)u_1 + g_2(q)u_2$$

ii. Use Lie brackets to find the Lie algebra of the system. What's the highest order of the Lie bracket you need to use? Is this system nonholonomic?

iii. Are any of the Pfaffian constraint(s) integrable? If so, what are the integrated constraints?

iv. The presence of a holonomic constraint indicates that the system can be represented by a smaller set of state variables (thus eliminating the constraint). If this system is holonomic, what's the minimum number of state variables needed? What are the dynamics of the system in those state variables?

(b) Suppose we have the following model of a cart:

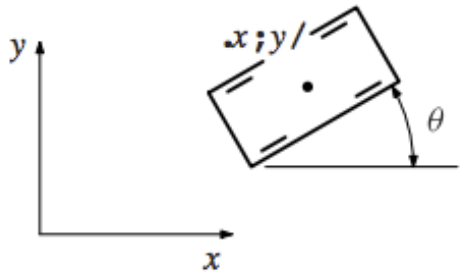

Figure 3: Model of a Cart

The dynamics of the system are given by:

$$\dot{x} = v\cos(\theta)$$
$$\dot{y} = v\sin(\theta)$$
$$\dot{\theta} = 0$$

Suppose we can control the speed of the cart v but not the angle θ.

i. Find the Pfaffian constraints $w(q,\dot{q})\dot{q} = 0$ and rewrite the dynamics in the form of

$$\dot{q} = g_1(q)u_1 + g_2(q)u_2$$

ii. Use Lie brackets to find the Lie algebra of the system. What's the highest order of the Lie bracket you need to use? Is this system nonholonomic?

iii. Are any of the Pfaffian constraint(s) integrable? If so, what are the integrated constraints?

iv. The presence of a holonomic constraint indicates that the system can be represented by a smaller set of state variables (thus eliminating the constraint). If this system is holonomic, what's the minimum number of state variables needed? What are the dynamics of the system in those state variables?

(c) Suppose we have the following model of a car:

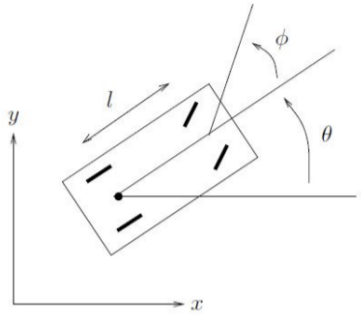

Figure 4: Bicycle Model of a Car

The dynamics of the system are given by:

$$\dot{x} = v\cos(\theta)$$
$$\dot{y} = v\sin(\theta)$$
$$\dot{\theta} = \frac{v}{l}\tan\phi$$

where l is the length of the car, and ϕ is the steering angle of with respect to the car body. Suppose we can control the speed of the car v, and the steering velocity $\dot{\phi}$.

i. Find the Pfaffian constraints $w(q,\dot{q})\dot{q} = 0$ and rewrite the dynamics in the form of

$$\dot{q} = g_1(q)u_1 + g_2(q)u_2$$

Note: this is a bit hairy. The constraints are reasonably easy to define, but representing them in terms of (x, y, θ, ϕ) and constants like l takes a fair amount of algebra.

ii. Use Lie brackets to find the Lie algebra of the system. What's the highest order of the Lie bracket you need to use? Is this system nonholonomic?

iii. Are any of the Pfaffian constraint(s) integrable? If so, what are the integrated constraints?

iv. The presence of a holonomic constraint indicates that the system can be represented by a smaller set of state variables (thus eliminating the constraint). If this system is holonomic, what's the minimum number of state variables needed? What are the dynamics of the system in those state variables?

3. The RRT algorithm is as follows:

Algorithm 1 The RRT algorithm

```
Graph.add_node(q_init)
while goal ∉ Graph do
    q_rand ⟵ Sample_Configuration()
    q_near ⟵ Nearest_Vertex(q_rand, Graph)
    q_new ⟵ Local_Planner(q_near, q_rand)
   if NOT Check_Collision(q_new) then
    Graph.add_node(q_new)
    Graph.add_edge(q_near, q_new)
   end if
end while
return Graph
```

What functions would need to be modified to make an RRT work with a nonholonomic robot model?

4. **Short Problems**

(a) An important part of the steering with sinusoids proof is the fact that sinusoids with integrally related frequencies are orthogonal. This means that

$$\langle \sin(\omega t), \sin(n\omega t)\rangle = \int_0^{\frac{2\pi}{\omega}} \sin(\omega t)\sin(n\omega t)dt = 0$$

for any integer n. Prove that this is true. *Hint: One of these might be useful*

(b) When would you use a sampling-based planner over an optimization-based planner? When would you use the optimization-based planner? Can you combine the two approaches?

5. **Research Comprehension**

CHOMP and TrajOpt are two major implementations of optimization-based motion planning for high-DOF robots (like arms). Skim both papers, and answer the following questions. CHOMP is a very long paper, and both are quite dense, so remember the paper reading techniques you've learned. If you try to read these papers like textbooks or novels, you'll be spending too much time on this question.

(a) Both papers spend quite a lot of space discussing their treatment of obstacles. In our optimization-based planner above, we deal with the obstacles with a single (rather simple) constraint. What makes this problem different to CHOMP and TrajOpt?

(b) How does each paper parameterize their paths?

(c) In our optimization-based path planner above, we assume that all our waypoints are close enough together that we don't need to check if the paths between them collide with obstacles. How do CHOMP and TrajOpt handle this problem?

(d) How do CHOMP and TrajOpt differ in their optimization methods?