Assignment 07: Ballistic Trajectory Calculation

Optimal Control for Robotics

Assigned: February 27 — Due: March 7 at 11:55pm

Introduction

In this assignment you will solve a simple boundary value problem using single shooting. The example problem will be that of aiming a canon: compute the launch velocity vector with the minimum launch speed such that the projectile reaches the specified target.

Deliverables

Implement the function `getBallisticParameters` using the template provided. Also submit any additional Matlab code that you write for the assignment.

Write-Up:

In addition to the Matlab code described above, upload a short write-up for this assignment:
`hw_07_studentName_writeup.txt`

- Header: full name, date, assignment name and number
- List any other students that you worked with.
- How long did this assignment take you?
- Briefly describe your approach to testing your code.
- Answers to the questions below.

One of the best ways to learn about trajectory optimization is to experiment with the inputs and outputs of an optimization. Here are a few things to investigate.

- How does varying the number of grid points and simulation method affect the number of iterations, solve time, and final accuracy of the NLP solver?
- Adjust the various problem parameters to see how the solution changes. What happens if the drag is large and the distance is large? Why?
- How many function calls does `fmincon` make on each iteration? What is going on here?

`computeBallisticTrajectory.m`

For this assignment you will implement one Matlab function: `computeBallisticTrajectory`. This function takes a set of paramters that describe a specific boundary value problem and the computes the optimal ballistic trajectory. In general there are many ways to solve this problem. For this assignment I will restrict a few of the details to keep things consistent across the class.

- Use the Matlab function `fmincon` to solve the underlying non-linear program. Create a `problem` data structure to define the NLP and then pass it as a single argument to `fmincon`.
- Use the function `runSimulation` from the code library to run the simulation.
- Use a uniform time grid.
- Use all of the inputs and return all of the outputs that are specified in the template code.
- The decision variables in the optiization should be the initial velocity vector (three elements) and the total duration of the simulation.
- The objective function should be the initial speed squared.

Inputs:

There are four inputs to this function, which are described in detail in the source code. The first input, `param`, describes the boundary value problem specification: boundary positions and physics model parameters. The function `getBallisticParameters` in the assignment source code can be used to provide a candidate set of parameters. The second input, `nGrid`, gives the number of gridpoints that should be used for the underlying simulation. Next is `method` which is passed directly to `runSimulation`. The final input, `nlpOpt`, is a set of options that are passed to `fmincon`. See the template code for how to get the default parameters.

Outputs:

There are two outputs, which describe the optimal trajectory that the ballistic projectile takes. The first should be the uniform time grid that was used by the simulation for the dynamics. The second should be the full state (position and velocity) at each point in time.

Template Code:

```
function soln = computeBallisticTrajectory(param, nGrid, method, nlpOpt)
%  soln = computeBallisticTrajectory(param, nGrid, method, nlpOpt)
%
% Computes a ballistic trajectory, given a model of
% the projectile motion and the start and target locations.
%
% INPUTS:
%   param = parameters for the simulation and ballistic model:
%       .gravity = scalar = gravity constant
%       .wind = [3,1] = [w1; w2; w3] = wind velocity
%       .drag = scalar = quadratic drag constant
%       .mass = scalar = mass of the projectile
%       .start = [3,1] = initial position of the trajectory
%       .target = [3,1] = target final position on the trajectory
%   nGrid = scalar = number of gridpoints for the integration method
%           --> use a uniform time grid
%   method = string = name of the desired method
%        'euler' = Euler's method (first-order)
%        'heun' = Heun's method (second-order)
%        'midpoint' = the midpoint method (second-order)
%        'ralston' = Ralston's method (second-order)
%        'rk4' = "The" Runge--Kutta method (forth-order)
%   nlpOpt = solver options object, created by:
%        >> nlpOpt = optimset('fmincon')
%       Useful options (set using "." operator, eg: nlpOpt.Display = 'off')
%           --> Display = {'iter', 'final', 'off'}
%           --> OptimalityTolerance = {1e-3, 1e-6}
%           --> ConstraintTolerance = {1e-3, 1e-5, 1e-10}
%
% OUTPUT:
%   soln = struct with the solution to the ballistic trajectory:
%     .tGrid = [1, nTime] = time grid for the ballistic trajectory
%     .zGrid = [6, nTime] = [x1;x2;x3;  dx1; dx2; dx3] = state trajectory
%
% NOTES:
%
%   Method: Single Shooting with uniform time grid
%
%   NLP Solver:  fmincon()
%
%   Simulation Method:  runSimulation()
%       - explicit Runge--Kutta methods, order 1, 2, or 4.
%
%   Physics Model:  projectileDynamics()
%

%%%% TODO:  implement this function

%%%% HINT:  a good implementation will have at least one sub-function

end
```

Dynamics Model

We will use a simple model of projectile motion: the projectile is acted on by only gravity and a quadratic drag force. The dynamics are implemented in the code library:
`ME149/codeLibrary/modelSystems/projectile/projectileDynamics.m`
The equations of motion are given below for the curious. The constant parameters are: d = coefficient of quadratic drag, g = acceleration due to gravity, m = projectile mass, $\vec{\boldsymbol{w}}$ = wind velocity. There are three time-varying scalar functions: $x = x(t)$, $y = y(t)$, and $z = z(t)$ that describe the position of the projectile. Finally, there are three constant unit vectors: $\hat{\boldsymbol{i}}$, $\hat{\boldsymbol{j}}$, $\hat{\boldsymbol{k}}$.

$$\vec{\boldsymbol{p}} = x\,\hat{\boldsymbol{i}} + y\,\hat{\boldsymbol{j}} + z\,\hat{\boldsymbol{k}} \tag{1}$$

$$\vec{\boldsymbol{v}} = \dot{\vec{\boldsymbol{p}}} - \vec{\boldsymbol{w}} \tag{2}$$

$$\vec{\boldsymbol{F}}_{\boldsymbol{D}} = -d\,||\vec{\boldsymbol{v}}||\,\vec{\boldsymbol{v}} \tag{3}$$

$$\vec{\boldsymbol{F}}_{\boldsymbol{G}} = -m\,g\,\hat{\boldsymbol{k}} \tag{4}$$

$$\ddot{\vec{\boldsymbol{p}}} = (\vec{\boldsymbol{F}}_{\boldsymbol{D}} + \vec{\boldsymbol{F}}_{\boldsymbol{G}})/m \tag{5}$$