Tutorial 12: Project Introduction

Gerasimos Maltezos, Dylan Vogel

24.05.2023

Overview

Project: Simulation of the SpaceX Falcon 9 vertical landing phase

Goal: Develop controller(s) to land the rocket safely

This tutorial: Simulator introduction, PID hovering example, linearization example

For the tutorial, go to the project link, save a copy to your Drive (File -> Save a Copy in Drive) and work on your copy.

Project submission

To submit your project, upload to Moodle the following files:

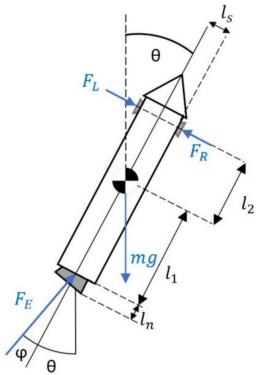
- The pdf of your slides
- Your code as a .ipynb file (you can export from colab to a .ipynb file)
  - Make sure your code generates the requested videos with the names given on slide 9
  - You do not need to upload the videos. We will run the code to generate the videos.
- A pdf (e.g. a Word or Google Doc) of the code documentation requested in the project description
  - This PDF should be a copy of the documentation in your code. We ask for this copy to make the grading easier.
  - The documentation requested in the project description answers the following questions:
    - How the failure scenario has been modeled in the simulation?
    - What parameters are available to define the failure scenario?
    - How the proposed controller is implemented?
    - What parameters need to be tuned in the controller?
    - How you tuned them and how would you recommend tuning them?

Actions and States

Actions

Three inputs are available for steering the rocket:

- F_E: the thrust produced by the main engine in Newtons (N), acting directly on the rocket body at the point where the nozzle pivots;
- F_S: the thrust produced by the side gas thrusters in Newtons (N), defined as the difference $F_L - F_R$, acting at a distance l_2 from the rocket center of mass;
- ϕ: the angle of the nozzle with respect to the rocket body in radians (rad), which changes the direction of F_E. It can be discontinuous and it has instant response up to the 60 fps frame rate of our model (sampling time of the environment).



State Variables

The state of the rocket is defined by the following variables:

All in radians

- (x, y): the 2D position of the rocket center of mass in meters (m);
- $(\dot{x}, \dot{y})$: the velocity of the rocket in meters per second (m/s);
- θ: the angle of the rocket in radians (rad);
- $\dot{\theta}$: the angular velocity of the rocket in radians per second (rad/s);
- c_L, c_R: binary variables indicating whether the left or right legs are in contact with the environment, respectively (equal to 0 otherwise).

The environment returns these variables as an observation:

$$\mathbf{x} = \begin{bmatrix} x & y & \dot{x} & \dot{y} & \theta & \dot{\theta} & c_L & c_R \end{bmatrix}$$

Simulator Inputs/Outputs

```
next_obs, rewards, done, _, info = env.step(action)
```

$$\mathbf{x} = \begin{bmatrix} x & y & \dot{x} & \dot{y} & \theta & \dot{\theta} & c_L & c_R \end{bmatrix} \qquad \mathbf{u} = \begin{bmatrix} F_E & F_S & \phi \end{bmatrix}$$

- **Controller provides actions, environment outputs observations**
  - Conforms to the OpenAI Gym/Gymnasium API
- **Done**: true/false variable indicating whether the simulation has terminated
  - True == terminated
- **Rewards**: float variable, could be used to train RL agents (not necessary for this project)

We incorporate this normalization for you in the provided linearized system matrices (discussed later)

IMPORTANT: The controller should return *normalized* actions, as a fraction of full scale. The main engine thrust is clipped to $[0, 1]$, while the side engine thrust and nozzle angle is clipped to $[-1, 1]$. The environment then scales these to the appropriate ranges (see *Limits on the Actuators*).

Simulator Inputs/Outputs

```
next_obs, rewards, done, _, info = env.step(action)
```

Limits on the State

The following limits should be assumed for the state:

- $x \in [0, 33.333]$;
- $y \in [0, 26.666]$;
- $\theta \in [-0.6108, 0.6108]$ (or $\pm 35^\circ$)

These are the *maximal* limits for the simulation, and the simulation will terminate if these are exceeded. Your controller should at least consider these limits on the state space.

IMPORTANT: All variables are with respect to world frame, which is located in the *bottom-left* corner of the simulation window. The x-axis increases going right (horizontally) along the window and the y-axis increases going up (vertically). Positive angles are measured counter-clockwise from the y-axis.

Passing Arguments

```
@dataclass
class UserArgs:
   """User arguments for tweaking the environment"""

   initial_position: Optional[Tuple[float, ...]] = None  # 3-tuple (x, y, theta)
   initial_state: Optional[Tuple[float, ...]] = None  # 6-tuple (x, y, x_dot, y_dot, theta, theta_dot)
   initial_barge_position: Optional[Tuple[float, ...]] = None  # 2-tuple (x, theta)
```

IMPORTANT: The values x and y are provided as a fraction of the screen width and height, for simplicity. All other values $(\theta, \dot{\theta}, \dot{x}, \dot{y})$ use their normal units of rad, rad/s, and m/s, respectively.

Recording & Playing Videos

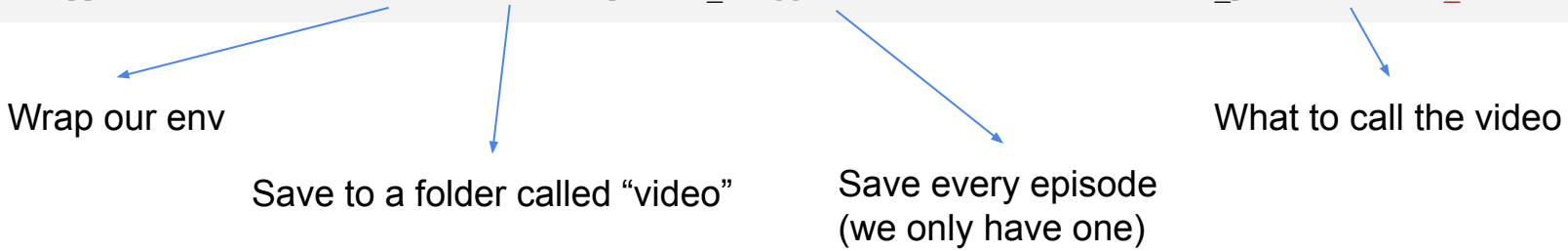
- Gym provides a **RecordVideo** wrapper which we can use to record videos from the episode
- We provide you a **show_video()** function to display the videos in Colab

```
import gymnasium as gym

env = gym.make("coco_rocket_lander/RocketLander-v0", render_mode="rgb_array", args={})
env = gym.wrappers.RecordVideo(env, 'video', episode_trigger = lambda x: True, name_prefix="video_name")

done = False
while not done:
 action = [0, 0, 0]
 next_obs, rewards, done, _, info = env.step(action)
 # etc …

env.close()  # video is saved at this step!
```

Recording & Playing Videos

- Gym provides a **RecordVideo** wrapper which we can use to record videos from the episode
- We provide you a **show_video()** function to display the videos in Colab

```
env = gym.wrappers.RecordVideo(env, 'video', episode_trigger = lambda x: True, name_prefix="video_name")
```



Important: save your videos and specify **name_prefix** as:

- `Firstname_Lastname_PID_Failure_Case` for the PID failure case
- `Firstname_Lastname_Controller_Demonstration` for your controller with the same case
- (optional) `Firstname_Lastname_Model_Free` if you do the bonus

Recording & Playing Videos

- Use **show_video()** to play the recorded videos

```
show_video(0)  # the number specifies which video to play

glob.glob('video/*.mp4')  # list your saved videos
# output: ['video/video_name-episode-0.mp4'
```

Successful Landings

Success:

- Both rocket legs are in contact with the landing platform
- Rocket completely stops moving (zero linear and angular velocity)
- *Environment will return reward=+100 and done=True*

Failure:

- Rocket legs make contact with the sea, or the rocket body contacts anything
- *Environment will return reward=-100 and done=True*

PID Hovering Exercise

Task 1:

Write some code which makes the rocket hover close to the center of the screen, with zero angle.

Hint: use the PID simulator example provided in the project as a template

Task 2:

Inject gaussian noise into the PID actions such that some movement is visible, but not enough to destabilize the rocket. Record the (noisy) inputs and outputs from the simulation into an array.

Hint: the PID actions may need to be clipped before adding noise

Linearization

A basic linearized model of the rocket dynamics has been derived and is provided to you, for the discrete-time update equation of the form:

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k.$$

Note:

- This equation assumes that we have discarded the last two elements (c_L, c_R) of the state observation $\mathbf{x}$ returned by the environment;
- The A matrix has shape $\mathbb{R}^{6\times 6}$;
- The B matrix has shape $\mathbb{R}^{6\times 3}$;
- We linearize around the upright equilibrium with action $\tilde{\mathbf{u}} = \begin{bmatrix} mg & 0 & 0 \end{bmatrix}$

Moreover, we scale B such that *normalized actions* (as a fraction of full scale) satisfy the update equation. The normalization of B is performed as follows:

$$B = B_{\text{unnormalized}} \begin{bmatrix} F_{E,\max} & 0 & 0 \\ 0 & F_{S,\max} & 0 \\ 0 & 0 & \phi_{\max} \end{bmatrix}$$

where $B_{\text{unnormalized}}$ is simply the usual linearized, discrete-time input matrix obtained from the system dynamics.

Linearization

Let A_c and B_c be the continuous-time linearized system matrices. The discrete-time version is obtained as:

$$A = e^{A_c T} \qquad B = \int_0^T e^{A_c \tau} B_c d\tau$$

Where T is the sampling time. Note that the system matrices are not time-varying.

```
from coco_rocket_lander.env import SystemModel

model = SystemModel(env)

model.calculate_linear_system_matrices()  # upright equilibrium with F_E = m*g
model.discretize_system_matrices(sample_time=0.1)  # you are free to change the sampling time

A, B = model.get_discrete_linear_system_matrices()
```

Linearization Simulation Exercise

Task 3:

Steer to the goal using the provided PID controller from initial position (0.45, 0.9, 0.2). Store the returned state observations in an array. At a time step k (you pick k), compare the output of the linearized dynamics for predicting x_k+1 with the true x_k+1 output by the simulator.

Can you explain any errors you observe? What might be the source of the errors? You can also plot the values over a horizon and see how well they align.

Hint 1: The PID controller outputs an action every 1/60 seconds, and may not respect the action limits
*Hint 2: The linearization is performed assuming F_E = m*g. How does this affect the state dynamics?*

Extra Slides: Reward Function

- If a more descriptive reward is necessary, the environment implements a similar reward to the Gym LunarLander environment
- The specific code which computes the reward can be found here (line 444)
- In words:
  - Reward change in position or velocity towards the landing point
  - Reward leg contact with the barge
  - Penalize non-zero rocket angle
  - Penalize thruster usage

Using this reward function is only necessary if you decide to use a RL solution for the model-free bonus