Rocket Landing - Gradient-based Algorithms and Differentiable Programming

PROJECT 1 REPORT

YONESHWAR BABU

ASU ID:1220454365

MAE 598: Design Optimization (Fall 2021)

OCTOBER 23, 2021

Under the guidance of

Yi Ren

Assistant Professor of aerospace and mechanical engineering

School for Engineering of Matter, Transport and Energy

Arizona State University

ACKNOWLEDGEMENT

I consider myself highly fortunate for the opportunity to do this project under the guidance of **YI (MAX) REN** who provided us a sample template code to work on.

ABSTRACT

The primary objective of this project **“ROCKET LANDING”** is develop a program using Gradient-Based algorithms and Differentiable Programming to safely land the rocket considering different parameters which in fluence the trajectory of the rocket while landing.

1.Introduction

I have considered a simple formulation of rocket landing, where the rocket state **x(t)**, is represented by its distance to the ground **d(t)** and its velocity **v(t)**, i.e., $\mathbf{x(t)=[d(t),v(t)]^T}$, where **t** specifies time. The control input of the rocket is its acceleration **a(t)**. The discrete-time dynamics follows

$$\mathbf{d(t+1) = d(t)+v(t)\Delta t},$$

$$\mathbf{v(t+1) = v(t)+a(t)\Delta t},$$

where $\mathbf{\Delta t}$**,** is a time interval. Further, let the closed-loop controller be

$$\mathbf{a(t) = f_\theta(x(t))}$$

where $\mathbf{f_\theta(.)}$**,** is a neural network with parameters θ, which are to be determined through optimization.

For each time step, we assign a loss as a function of the control input and the state: **l(x(t), a(t)).** In this example, we will simply set **l(x(t), a(t)) =0** for all **t=1,…,T-1** where **T,** is the final time step and $\mathbf{l(x(T),a(T))=\|x(T)\|^2=\|d(T)\|^2+\|v(T)\|^2}$**.** This loss encourages the rocket to reach **d(T)=0** and **v(T)=0**, which are proper landing conditions.

The optimization problem is now formulated as

$$\mathbf{Min}\ \ \mathbf{\|x(T)\|^2}$$

$$\mathbf{d(t+1) = d(t)+v(t)\Delta t},$$

$$\mathbf{v(t+1) = v(t)+a(t)\Delta t},$$

$$\mathbf{a(t)=f_\theta(x(t)),\ \forall\ t=1,\ldots,T-1}$$

While this problem is constrained, it is easy to see that the objective function can be expressed as a function of x(T-1) and a(T-1), where x(T-1) as a function of x(T-2) and a(T-2), and so on. Thus, it is essentially an unconstrained problem with respect to θ.

In the following, we code this problem up with PyTorch, which allows us to only build the forward pass of the loss (i.e., how we move from x(1) to x(2) and all the way to x(T) and automatically get the gradient Δ .

Sample code:

```
# overhead

import logging
import math
import random
import numpy as np
import time
import torch as t
import torch.nn as nn
from torch import optim
from torch.nn import utils
import matplotlib.pyplot as plt

logger = logging.getLogger(__name__)

# environment parameters

FRAME_TIME = 0.1  # time interval
GRAVITY_ACCEL = 0.12  # gravity constant
BOOST_ACCEL = 0.18  # thrust constant

# # the following parameters are not being used in the sample code
# PLATFORM_WIDTH = 0.25  # landing platform width
# PLATFORM_HEIGHT = 0.06  # landing platform height
# ROTATION_ACCEL = 20  # rotation constant
# define system dynamics
# Notes:
# 0. You only need to modify the "forward" function
# 1. All variables in "forward" need to be PyTorch tensors.
# 2. All math operations in "forward" has to be differentiable, e.g., default
PyTorch functions.
# 3. Do not use inplace operations, e.g., x += 1. Please see the following section
for an example that does not work.

class Dynamics(nn.Module):

    def __init__(self):
        super(Dynamics, self).__init__()

    @staticmethod
    def forward(state, action):

        """
        action: thrust or no thrust
        state[0] = y
        state[1] = y_dot
        """

        # Apply gravity
        # Note: Here gravity is used to change velocity which is the second
element of the state vector
        # Normally, we would do x[1] = x[1] + gravity * delta_time
```

```
        # but this is not allowed in PyTorch since it overwrites one variable
(x[1]) that is part of the computational graph to be differentiated.
        # Therefore, I define a tensor dx = [0., gravity * delta_time], and do x =
x + dx. This is allowed...
        delta_state_gravity = t.tensor([0., GRAVITY_ACCEL * FRAME_TIME])

        # Thrust
        # Note: Same reason as above. Need a 2-by-1 tensor.
        delta_state = BOOST_ACCEL * FRAME_TIME * t.tensor([0., -1.]) * action

        # Update velocity
        state = state + delta_state + delta_state_gravity

        # Update state
        # Note: Same as above. Use operators on matrices/tensors as much as
possible. Do not use element-wise operators as they are considered inplace.
        step_mat = t.tensor([[1., FRAME_TIME],
                             [0., 1.]])
        state = t.matmul(step_mat, state)

        return state

# Demonstrate the inplace operation issue

class Dynamics(nn.Module):

    def __init__(self):
        super(Dynamics, self).__init__()

    @staticmethod
    def forward(state, action):

        """
        action: thrust or no thrust
        state[0] = y
        state[1] = y_dot
        """

        # Update velocity using element-wise operation. This leads to an error
from PyTorch.
        state[1] = state[1] + GRAVITY_ACCEL * FRAME_TIME - BOOST_ACCEL *
FRAME_TIME * action

        # Update state
        step_mat = t.tensor([[1., FRAME_TIME],
                             [0., 1.]])
        state = t.matmul(step_mat, state)

        return state
```

In [5]:

```
# a deterministic controller
# Note:
# 0. You only need to change the network architecture in "__init__"
# 1. nn.Sigmoid outputs values from 0 to 1, nn.Tanh from -1 to 1
```

```python
# 2. You have all the freedom to make the network wider (by increasing
"dim_hidden") or deeper (by adding more lines to nn.Sequential)
# 3. Always start with something simple

class Controller(nn.Module):

    def __init__(self, dim_input, dim_hidden, dim_output):
        """
        dim_input: # of system states
        dim_output: # of actions
        dim_hidden: up to you
        """
        super(Controller, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(dim_input, dim_hidden),
            nn.Tanh(),
            nn.Linear(dim_hidden, dim_output),
            # You can add more layers here
            nn.Sigmoid()
        )

    def forward(self, state):
        action = self.network(state)
        return action
# the simulator that rolls out x(1), x(2), ..., x(T)
# Note:
# 0. Need to change "initialize_state" to optimize the controller over a
distribution of initial states
# 1. self.action_trajectory and self.state_trajectory stores the action and state
trajectories along time

class Simulation(nn.Module):

    def __init__(self, controller, dynamics, T):
        super(Simulation, self).__init__()
        self.state = self.initialize_state()
        self.controller = controller
        self.dynamics = dynamics
        self.T = T
        self.action_trajectory = []
        self.state_trajectory = []

    def forward(self, state):
        self.action_trajectory = []
        self.state_trajectory = []
        for _ in range(T):
            action = self.controller.forward(state)
            state = self.dynamics.forward(state, action)
            self.action_trajectory.append(action)
            self.state_trajectory.append(state)
        return self.error(state)

    @staticmethod
    def initialize_state():
```

```
        state = [1., 0.]  # TODO: need batch of initial states
        return t.tensor(state, requires_grad=False).float()

    def error(self, state):
        return state[0]**2 + state[1]**2
# set up the optimizer
# Note:
# 0. LBFGS is a good choice if you don't have a large batch size (i.e., a lot of
initial states to consider simultaneously)
# 1. You can also try SGD and other momentum-based methods implemented in PyTorch
# 2. You will need to customize "visualize"
# 3. loss.backward is where the gradient is calculated (d_loss/d_variables)
# 4. self.optimizer.step(closure) is where gradient descent is done

class Optimize:
    def __init__(self, simulation):
        self.simulation = simulation
        self.parameters = simulation.controller.parameters()
        self.optimizer = optim.LBFGS(self.parameters, lr=0.01)

    def step(self):
        def closure():
            loss = self.simulation(self.simulation.state)
            self.optimizer.zero_grad()
            loss.backward()
            return loss
        self.optimizer.step(closure)
        return closure()

    def train(self, epochs):
        for epoch in range(epochs):
            loss = self.step()
            print('[%d] loss: %.3f' % (epoch + 1, loss))
            self.visualize()

    def visualize(self):
        data = np.array([self.simulation.state_trajectory[i].detach().numpy() for
i in range(self.simulation.T)])
        x = data[:, 0]
        y = data[:, 1]
        plt.plot(x, y)
        plt.show()
# Now it's time to run the code!

T = 100  # number of time steps
dim_input = 2  # state space dimensions
dim_hidden = 6  # latent dimensions
dim_output = 1  # action space dimensions
d = Dynamics()  # define dynamics
c = Controller(dim_input, dim_hidden, dim_output)  # define controller
s = Simulation(c, d, T)  # define simulation
o = Optimize(s)  # define optimizer
o.train(40)  # solve the optimization problem
```

2.Problem Formulation

I have considered a formulation of rocket landing of Falcon 9. I included a better formulation with thrust in x and y direction and a fuel constraint, if the rocket has some fixed amount of fuel and will be depleted based proportional to the acceleration. The added constraint is that there is only a fixed amount of fuel, and therefore any deviation from that value would be considered as a loss. The goal is to consume fuel, but to prioritize other parameters first. Therefore, the weightage to the fuel would be kept low.

Objective Function

$$\mathbf{Min}_{\theta} \, \|\mathbf{X(T)}\|^2$$

Subject to:

$$fc(t+1) = A(t)\Delta t$$

$$x(t+1) = x(t)+v_x(t)\Delta t$$

$$y(t+1) = y(t)+v_y(t)\Delta t$$

$$v_x(t+1)= v_x(t)+a_x(t)\,\Delta t$$

$$v_y(t+1)= v_y(t)+a_y(t)\,\Delta t$$

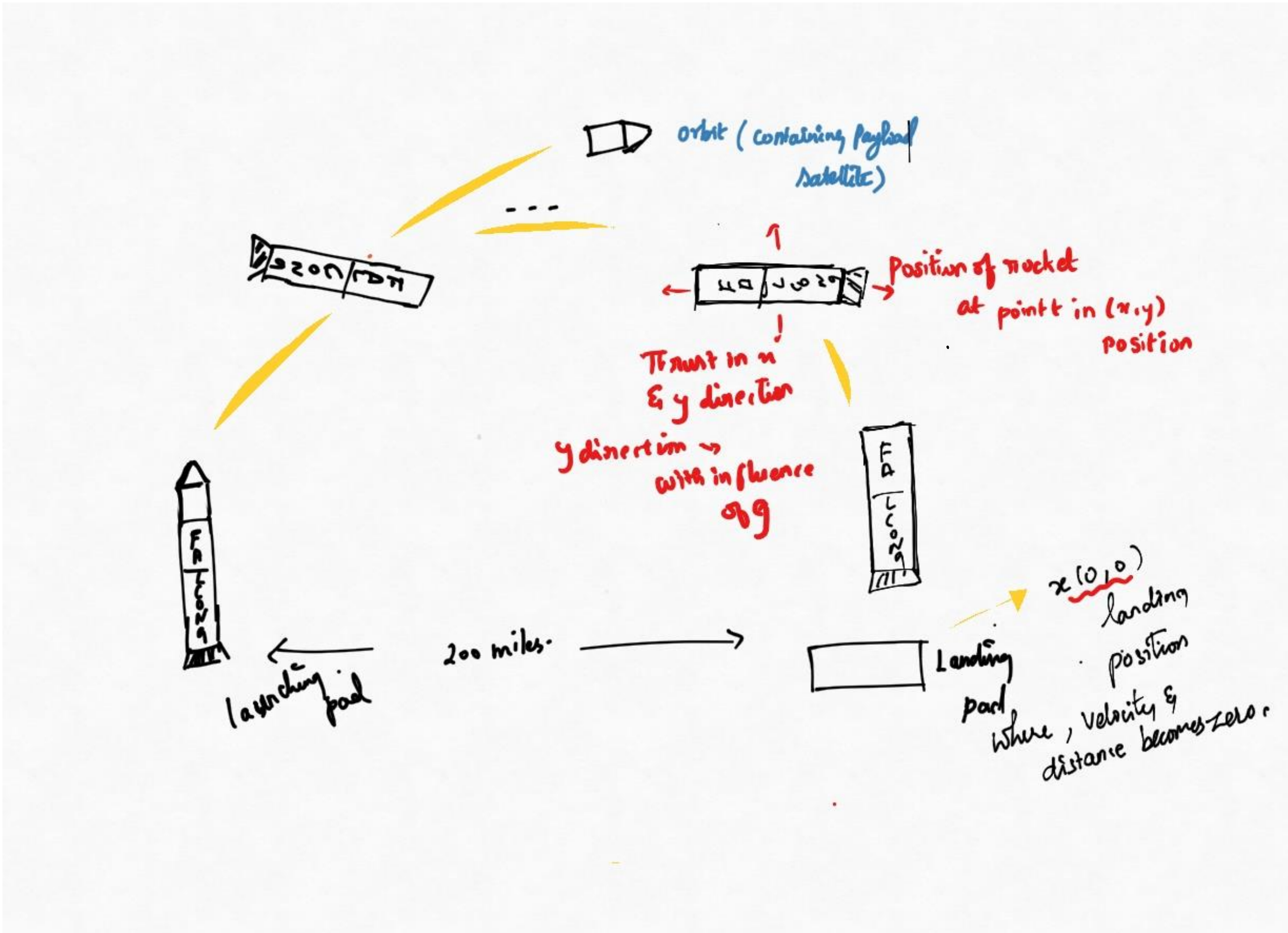
$$A(t)= a_x(t)\,\Delta t+ a_y(t)\,\Delta t$$

$$A(t),fc(t)= f_\theta(x(t)),\ \forall\ t=1,\ldots,T-1$$

$X(T)=(x(T),y(T),v_{xy}, fc(T))$

fc is fuel consumption, influenced by acceleration towards the landing pad. This loss encourages the rocket to reach $\mathbf{v_x(T)=0, v_y(T)=0}$, which are proper landing conditions. the objective function can be expressed as a function of x(T-1),y(T-1), a_y(T-1), and a_x (T-1), where x(T-1) as a function of x(T-2),y(T-2) and a_x (T-2), a_y (T-2), and so on.

Pictorial representation



The initial position of the rocket is taken at 15 at x-axis and 12 in y-axis having zero velocity at that instance.
I have considered the total fuel capacity as 300 tons.