Practice Exercises for Installation and Python Tips

Sean Matthew Nolan

February 23, 2021

1 Install Beluga

Problem

On your local machine, install beluga per the instructions given in the lecture. Use a virtual environment. Navigate to the folder that contains beluga, and to verify installation, run:

```
> python .\examples\Classic\MoonLander\MoonLander.py
```

Solution

If installation was successful, output should look like:

```
Cont. #1 (      h_f -> 0, v_f -> 0      ): 100%|#########| 10/10 [00:00<00:00, 73.58trajectories/s]
Cont. #2 ( u_lower -> 0, u_upper -> 4 ): 100%|#########| 10/10 [00:00<00:00, 73.10trajectories/s]
Cont. #3 (      epsilon1 -> 1e-05       ): 100%|#########| 30/30 [00:01<00:00, 20.73trajectories/s]
beluga.py:172: Continuation process completed in 1.7493 seconds.
```

2 Morse Code Translator

Problem

Write a program in Python that automatically converts text to Morse code and vice versa.

Solution

Below is an implementation of a Morse code translator:

```
# Define character to morse mappings here
lookup_dict = {
    'a': '.-',    'b': '-...',  'c': '-.-.',  'd': '-..',   'e': '.',
    'f': '..-.',  'g': '--.',   'h': '....',  'i': '..',    'j': '.---',
    'k': '-.-',   'l': '.-..',  'm': '--',    'n': '-.',    'o': '---',
    'p': '.--.',  'q': '--.-',  'r': '.-.',   's': '...',   't': '-',
    'u': '..-',   'v': '...-',  'w': '.--',   'x': '-..-',  'y': '-.--',
    'z': '--..',  '1': '.----', '2': '..---', '3': '...--', '4': '....-',
    '5': '.....', '6': '-....', '7': '--...', '8': '---..', '9': '----.',
    '0': '-----', ' ': '|'
}


def text_to_morse(text):
    # Make text lowercase because case doen't matter for Morse code
    text = text.lower()

    # Intialize output string
    out = ''

    # Iterate through each character in the string
    for char in text:
        # If char is letter, number, or space
        if char in lookup_dict.keys():
            out += lookup_dict[char] + ' '
```

```
    return out


if __name__ == '__main__':

    # Test function
    test_strings = [
        'Hello World',
        'The quick brown fox jumps over the lazy dog',
        'In principio erat Verbum, et Verbum erat apud Deum, et Deus erat Verbum.'
    ]

    morse_results = [text_to_morse(test_string) for test_string in test_strings]

    for input_string, output_string in zip(test_strings, morse_results):
        print('{0}\n{1}\n\n'.format(input_string, output_string))
```

3 Catapult Launch Problem

Problem

Using Python, numerically maximize the range of a projectile launched by a catapult by finding the optimal release angle γ_0. After the projectile is released, it travels with dynamics:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{v} \\ \dot{\gamma} \end{bmatrix} = \begin{bmatrix} v\cos\gamma \\ v\sin\gamma \\ -\frac{D(v)}{m} - g\sin\gamma \\ -\frac{g}{v}\cos\gamma \end{bmatrix} \tag{3.1}$$

Using fixed initial velocity $v_0 = 100\,\text{ft/s}$, mass $m = 10\,\text{slugs}$, gravity $g = 32.17\,\text{ft/s}$, drag equation

$$D(v) = \tfrac{1}{2}\rho v^2 C_D A\,, \tag{3.2}$$

density $\rho = 0.00235\,\text{slugs/ft}^3$, drag coefficient $C_D = 0.5$, and reference area $A = 15\,\text{ft}^2$

Assume that projectile lands at the same altitude from which it was launched.

Hints

The cost function should numerically propagate the trajectory of the projectile to obtain the range. Use SciPy's `solve_ivp()` with a terminal event (see: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`)

The optimization problem can be solved with SciPy's `minimize` solver (see: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html`)

Solution

The optimal launch angle is 42.6103 degrees. Below is a code that solves the problem:

```
import numpy as np
from math import sin, cos, pi
from scipy import integrate, optimize
import matplotlib.pyplot as plt

v_0 = 150.
m = 10
g = 32.17
rho = 0.00235
c_d = 0.5
a_ref = 15


# Define drag equation
def drag(v):
    return 0.5 * rho * v**2 * c_d * a_ref
```

```
# Define dynamics of projectile
def dynamics(_, states):
    x, y, v, gam = states
    return np.array([
        v * cos(gam),
        v * sin(gam),
        -drag(v) / m - g * sin(gam),
        -g / v * cos(gam)
    ])


# Define terminal event (note the addition of attributes to the function)
def hit_ground(_, states):
    return states[1]


hit_ground.terminal = True
hit_ground.direction = -1


# Define cost function
def cost(gam_0):
    # Propagate trajectory
    prop_sol = integrate.solve_ivp(
        dynamics, [0., 50.], np.array([0., 0., v_0, gam_0]), events=hit_ground)

    # Extract and return flight range
    x_f = prop_sol.y[0, -1]

    # Invert range for minimization
    return -x_f


# Solve optimization problem (Guess 45 deg which is no drag solution)
opt_sol = optimize.minimize(cost, np.array([pi/4]))
opt_gam_0 = opt_sol.x[0]

print('Optimal release angle is {0:.4f} degrees'.format(opt_gam_0 * 180 / pi))

################################################
# Create plots for illustration (not needed)
fig = plt.figure(0, figsize=(6.5, 3.5))

gam_0_range = np.linspace(0, pi/2, 15)
cost_range = np.array([cost(_gam_0) for _gam_0 in gam_0_range])

ax1 = fig.add_subplot(1, 2, 1)
ax1.plot(gam_0_range * 180 / pi, -cost_range, label='range')
ax1.plot(opt_gam_0 * 180 / pi, -opt_sol.fun, 'x', label='optimal')
ax1.legend()
ax1.set_xlabel(r'launch angle, $\gamma_0$ [deg]')
ax1.set_ylabel('range, $x_f$ [ft]')

ax2 = fig.add_subplot(1, 2, 2)
for _gam_0 in gam_0_range:
    _prop_sol = integrate.solve_ivp(dynamics, [0., 50.], np.array([0., 0., v_0, _gam_0]),
                                    max_step=0.1, events=hit_ground)
    ax2.plot(_prop_sol.y[0, :], _prop_sol.y[1, :], color='C0')

_prop_sol = integrate.solve_ivp(dynamics, [0., 50.], np.array([0., 0.01, v_0, opt_gam_0]),
                                max_step=0.1, events=hit_ground)
ax2.plot(_prop_sol.y[0, :], _prop_sol.y[1, :], color='C1')

ax2.set_xlabel(r'$y$ [deg]')
ax2.set_ylabel('$x$ [ft]')

fig.subplots_adjust(wspace=0.33, bottom=0.15)

plt.show()
```

Figure 1 shows how the trajectory varies with launch angle verifying our answer.

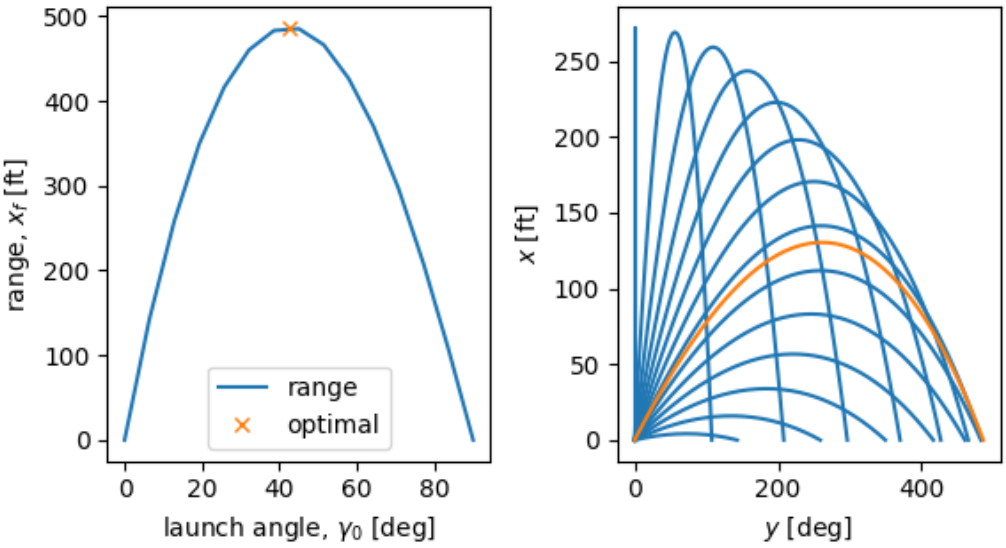

Figure 1: Left: Terminal ranges for different launch angles (Orange x denotes optimal) Right: Projectile trajectories for different launch angles (Orange is optimal)

4 Tarjan's Algorithm

Problem

Tarjan's strongly connected components algorithm finds groups of strongly connected components in a directed graph. Strongly connected refer to groups in which every node is reachable from every node.

For algorithm details, refer to `https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm`

Implement this algorithm using a custom "node" class in Python. It is suggested that you make a function to construct a graph from an adjancy matrix. Test the implementation with the adjacency matrix (where a non-zero element A_{ij} indicates a edge pointing from i to j):

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Solution

Below is an implementation of the algorithm:

```
import numpy as np


class Node:
    def __init__(self, identifier):
        self.identifier = identifier

        self.in_nodes = []
        self.out_nodes = []
```

```
        self.index = None
        self.low = None

    def __repr__(self):
        return 'Node_{}'.format(self.identifier)


# Function to create graph from adjacency matrix
def create_graph(adj_mat):
    # Initialize nodes
    num_nodes = adj_mat.shape[0]
    node_list = [Node(_k) for _k in range(num_nodes)]

    # Set in and out nodes (edges)
    for i, node in enumerate(node_list):
        node.in_nodes = [node_j for node_j, cond in zip(node_list, adj_mat[:, i]) if cond]
        node.out_nodes = [node_j for node_j, cond in zip(node_list, adj_mat[i, :]) if cond]

    return node_list


# Create class to encapsulate index and stack of Trajan algorithm
class TarjanClass:
    def __init__(self):
        self.index = 0
        self.stack = []
        self.strongly_connected_components = []

    def tarjan(self, node_set):
        self.index = 0
        self.stack = []

        for node in node_set:
            if node.index is None:
                self.strong_connect(node)

        return self.strongly_connected_components

    def strong_connect(self, node):
        node.index = self.index
        node.low = self.index
        self.index += 1

        self.stack.append(node)

        # Consider out nodes of current node
        for out_node in node.out_nodes:
            if out_node.index is None:
                self.strong_connect(out_node)
                node.low = min(node.low, out_node.low)

            # Check if out_node is already on stack. Bypasses need for onStack
            # flag used in psuedo-code
            # If out_node not on stack, current edge points to already identified
            # strongly connected components
            elif out_node in self.stack:
                node.low = min(node.low, out_node.index)

        # If root node returned to, pop the node from stack and generate
        # strongly connected components
        if node.low == node.index:
            end_node = self.stack.pop()
            scc = [end_node]

            while end_node is not node:
                end_node = self.stack.pop()
                scc.append(end_node)

            self.strongly_connected_components.append(scc)


# Define tarjan as a single function
tarjan = TarjanClass().tarjan

if __name__ == '__main__':
    test_adj_mat = np.array([
        [0, 0, 0, 0, 1, 0, 0, 0],
        [1, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 1, 0, 0, 0, 0],
        [0, 0, 1, 0, 1, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0],
```

```
    [0, 1, 0, 0, 1, 0, 1, 0],
    [0, 0, 1, 0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0, 0, 1, 1]
])

graph = create_graph(test_adj_mat)
sccs = tarjan(graph)
```

The strongly connected components of the given network are

1. Nodes: 1, 2, 5
2. Nodes: 3, 4
3. Nodes: 6, 7
4. Node: 8

References