Project 3

ECE 10

Connor Guzikowski (cguzikow)
Jacob Sickafoose (jsickafo)
Sriram Venkataraman (svenka12)

June 3, 2022

Part 1: 2D Path Planning

Overview

For this section, we were expected to load the map from the provided argument and calculate a path between the *'start'* and *'goal'* coordinates stored in the Python pickle. This path was to be calculated using either *A** or *Dijkstra's Algorithm* and overlaid over the map.

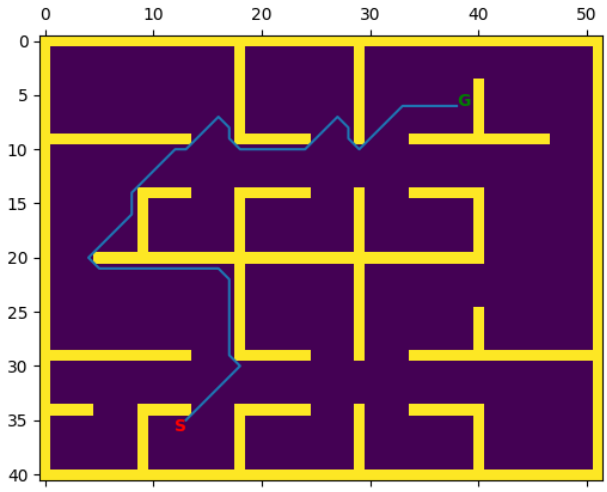
Approach

We utilized *A** in order to calculate the path between the *'start'* and *'goal'*. We observed that the *'map'* field in the Pickle was stored as a numpy matrix of 0′*s* and 1′*s*. The 0′*s* represented the walkable space the robot could travel across and the 1′*s* were the walls preventing movement.

We started out by being able to plot a line on top of the map we were given so we could plot the path when one was calculated. This was done by taking the list of points that defined the *A** path that was generated, and then storing all of the x and y values into separate lists. After that, we can just plot these two lists. Once this was accomplished, we found an implementation on the web of *A** in Python. This code was used as a baseline for our algorithm.

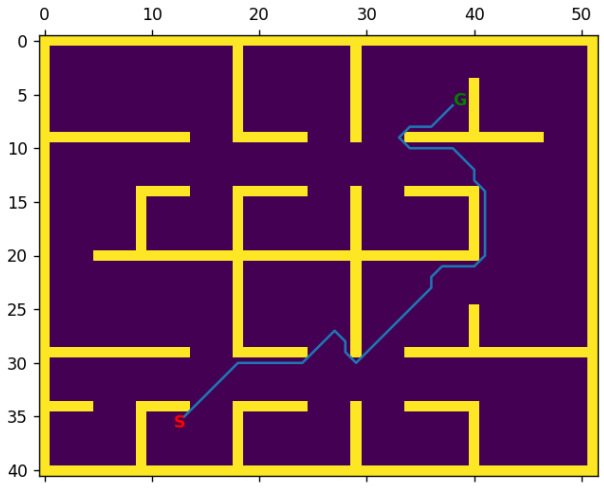
`https://medium.com/@nicholas.w.swift/`
`easy-a-star-pathfinding-7e6689c7f7b2`

There were some issues that we had with our code, for example, the way the matrix is indexed for the pickle is y, x instead of x, y. Example shown in Path 1:



(a) Path 1

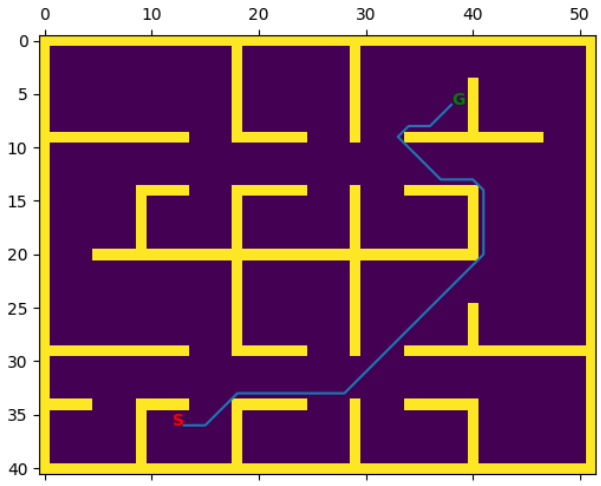
Another issue we had is that we were not creating a higher cost to move diagonally. This made it so the algorithm would place a diagonal path in unnecessary areas, such as gaps that were in between walls, but didn't lead anywhere. This is shown in Path 2:



(b) Path 2

The final main problem we were having was that we were not calculating the estimated distance correctly. We were calculating it by using the formula: $(x_{goal} - x_{node})^2 + (y_{goal} - y_{node})^2$. This should seem like an obvious misrepresentation of

the Pythagorean theorem (as it pretty much is), but this was used because in a video about A^*, the creator of the video explained that the square root was unnecessary. In hindsight this wasn't a very smart idea, and we should have used our intuition instead. Using $\sqrt{(x_{goal} - x_{node})^2 + (y_{goal} - y_{node})^2}$, we get the result:



(c) Path 3

This result was achieved with the following code:

```
#!/usr/bin/env python3
import pickle
from turtle import position
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt

f = open('project3_part1.pickle','rb')
res = pickle.load(f)

# Class to create a node to represent every pixel of the math
class Node():
    """A node class for A* Pathfinding"""
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
```

```
        self.g = 0 # Current Distance Traveled
        self.h = 0 # Estimated Distance to goal
        self.f = 0 # Estimated total distance of path

    def __eq__(self, other):
        return self.position == other.position


def astar(maze, start, end):
    # Beginning and end of the path
    start_node = Node(None, start)
    end_node = Node(None, end)
    # Create 2 lists for open and closed:
        # Open means the node has not been explored yet,
        # Closed means the node has been explored, and is the lowest
            cost it can be
    open = []
    closed = []

    open.append(start_node)
    # Look for more nodes while they are available
    while len(open) > 0:
        # Initialize to first node
        current_node = open[0]
        current_index = 0
        # Look for item in the open list with the lowest f cost
            (Total distance of path with point)
        for index, item in enumerate(open):
            if item.f < current_node.f:
                current_node = item
                current_index = index
        # When selecting the current node, select the node with
            lowest cost in open
        open.pop(current_index)
        closed.append(current_node)

        # If the current node is the end node, then the path has
            been found
            # Go through the parent nodes to find the path
        if current_node == end_node:
```

```
    path = []
    point = current_node
    while point.parent:
        path.append(point.position)
        point = point.parent
    return path
children = []
for neighbor in [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1),
    (1, -1), (-1, 1), (-1, -1)]:
# for neighbor in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
    new_position = (current_node.position[0] + neighbor[0],
        current_node.position[1] + neighbor[1])
    # Check for bounds
    if new_position[1] > len(maze) - 1 or new_position[0] < 0
        or new_position[0] > len(maze[len(maze)-1]) -1 or
        new_position[1] < 0:
        continue
    # Check to see if already evaluated
    if Node(current_node, new_position) in closed:
        continue
    # Check to see if there is a wall
    if maze[new_position[1]][new_position[0]] != 0:
        continue
    # Create new node if it is a valid neighbor
    new_node = Node(current_node, new_position)
    # Add the child of current node to list
    children.append(new_node)
# Loop through the nodes connected to current node
for child in children:
    # Add more distance if it is a diagonal reach
    if(abs(current_node.position[0] - child.position[0]) ==
        abs(current_node.position[1] - child.position[1])):
        child.g = current_node.g + 14
    else:
        child.g = current_node.g + 10
    # Get the estimated distance to travel
    child.h = sqrt(((child.position[0] -
        end_node.position[0])**2) + ((child.position[1] -
        end_node.position[1])**2))
    child.f = child.g + child.h
```

```
            # Check to make sure you have lowest cost version of the
                node
            if child in open:
                for open_node in open:
                    if child == open_node and child.g < open_node.g:
                        open_node = child
            else:
                open.append(child)
# Create the maze from the pickle
maze = res['map'].tolist()
# Set start and end points
start = (12, 36)
end = (38, 6)
# Create path
path = astar(maze, start, end)
# Get values of path to plot them
xvals = []
yvals = []
for x,y in path:
    xvals.append(x)
    yvals.append(y)
# Plot maze, path, and start/end points
plt.matshow(res['map'])
start = res['start']
plt.text(start[0], start[1], 'S', color='r',fontweight='bold')
goal = res['goal']
plt.text(goal[0], goal[1], 'G', color='g',fontweight='bold')
plt.plot(xvals, yvals)
plt.show()
```

Part 2: Local Jacobian Planning with a Keyboard Interface

Overview

For this section, we were first expected to code the controller to make the black and yellow ball movable with buttons on the keyboard. Once this was accomplished, we were then tasked with calculating the error between the current pose of the end

effector and the black and yellow ball. This error was then scaled and left multiplied by the inverse Jacobian to calculate the arm joint motor speeds to move the end effector to the desired pose.

Approach

We expanding the included keyboard reading code in order to include (q,w) for *X axis*, (a,s) for *Y*, (z,x) for *Z*, (Comma,Period) for *roll*, (up arrow, down arrow) for *pitch*, and (left arrow, right arrow) for *yaw*. Since Webot's represents rotations through the axis-angle notation, we had to convert between the keyboard input (which was read as Euler angles) to axis-angle in order to offset the previous value. Additionally, the x and y axes needed to be flipped as a result of Webot's translation semantics. When each of these buttons was pressed, the supervisor object for the black and yellow ball had the corresponding transformation and rotation applied.

In order to apply inverse kinematics to have the end effector track the target, the desired end effector translational and rotational rates of change need to be calculated (this provides us with the format we can use to find the joint angle velocities, which is what Webots needs as input parameters for the UR10). Since the error between current and desired state is essentially a vector that the robot needs to travel, this can be estimated to be the "$\dot{state}$" vector. Finally, the $\dot{state}$ can multiplied with the inverse Jacobian to find the motor speeds needed to track the target. We were able to calculate the Jacobian using the homogenous transform matrices from the base of the arm to the end effector. Using the 3x3 block in the upper left hand corner - the rotation matrix (abbreviated as R) and the right most 3x1 column vector - the position vector abbreviated as d:

$$J = \begin{bmatrix} J_v \\ J_\omega \end{bmatrix} = \begin{bmatrix} R_0^0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times (d_n^0 - d_0^0) & \cdots & R_{n-1}^0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times (d_n^0 - d_{n-1}^0) \\ R_0^0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} & \cdots & R_{n-1}^0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \end{bmatrix}$$

Part 3: Global Inverse Kinematic Planning (Robotic Fruit Ninja)

Overview

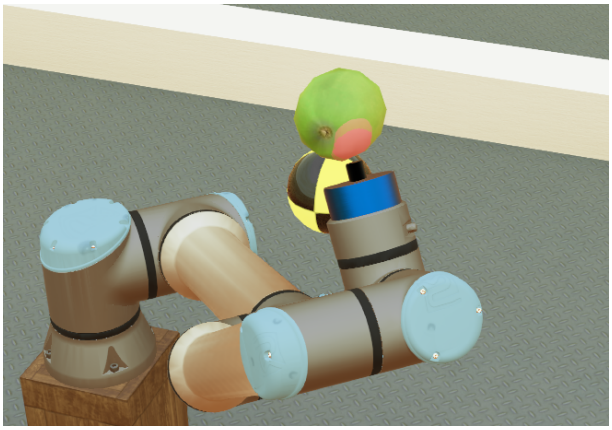
For this part, we were provided with code to launch the fruit into the air as well as a pen on the robot which automatically marks the apples when it is within $2cm$ of the fruit. We were tasked with moving the arm to mark the fruit in the air by calculating the trajectory of the fruit, deciding on a reasonable interception point and moving the arm to the point.

Approach

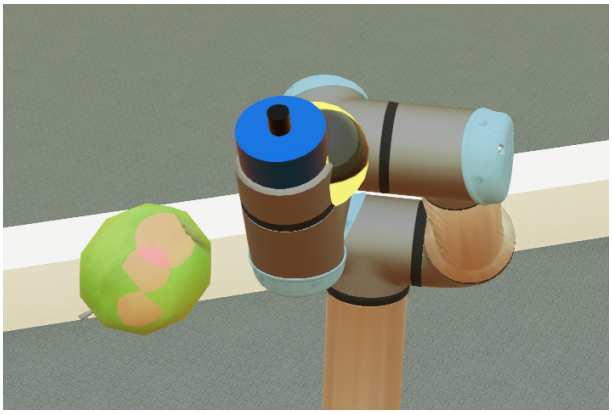
We chose a simple approach to this part. Rather than planning the path in configuration space, we simply set the goal position of the end effector to make contact with each apple manually. This was done by first creating a function which calculated the location of the apples based on their trajectory. The black and yellow ball which the end effector follows was then moved to follow the same trajectory at each time step. At the time we knew the first apple was marked, we start looking at the trajectory for the next ball.

This method involved some guessing and checking to find where in the trajectory of each apple to place the target ball in order to intersect the path. Guessing and checking was also used to find what time to switch focus to the next apple because we observed the last being marked. While going through this process, we realized

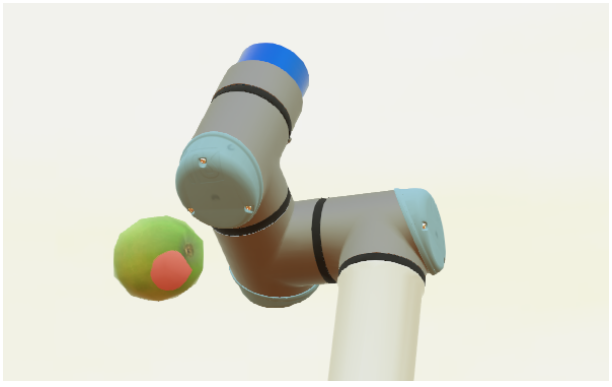
that we could not track the apples' trajectories exactly with the end effector, as this would often lead to hitting the apple with a link of the arm instead of marking the apple. This issue was counteracted by tracking a slightly delayed path on apples rising, and a slightly ahead path for the falling balls. After accounting for all of this, we were able to successfully mark all of the balls:



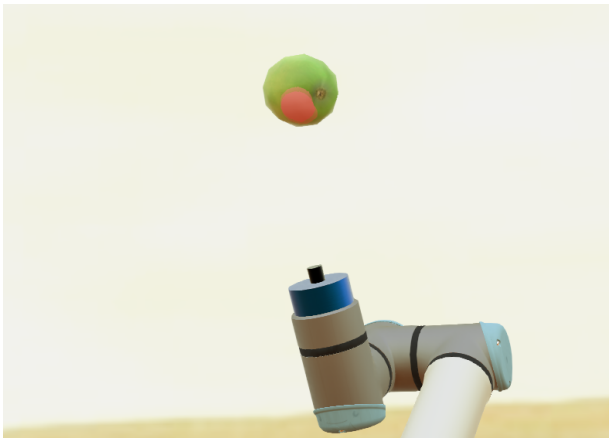
(d) First Apple Marked



(e) Second Apple Marked



(f) Third Apple Marked



(g) Fourth Apple Marked

Part 4: Questions

1. What are the names of everyone in your lab group?

Jacob Sickafoose (jsickafo) -Master repo
Connor Guzikowski (cguzikow)
Sriram Venkataraman (svenka12)

2. How did your group work together to solve each part?

For part 1, we initially were all trying independently implement the path solving with A*. We were each on our own computer but talking together working through it. At some point this transitioned into three brains coding on one computer as we focused on getting it to work.

Rather than breaking down the rest of the parts into functions for each of us to write, we continued on in the same way. This meant working on the same section of code together when we could, and separately otherwise and updating our Discord repo at the end of each work session.

3. Roughly how much time did you spend programming this lab?

Easily the longest assignment so far, we had a combined total man hours of **25**

4. Does your implementation work as expected? If not, what problems do you encounter?

Kinda sorta... As mentioned previously, for part 2 we suspect we are not calculating joint velocities using the inverse Jacobian as we should. The final result works as we think it is supposed to, it's just the methodology we are not sure is correct.

Also mentioned previously, our part 3 achieves the goal of marking each apple with a much more simplistic approach than path planning. Rather than calculating the optimal path between each of the apples, we did that ourselves and essentially hard coded locations for the arm to move to in order to mark the apples. This means that relaunching the apples again would not automatically do anything and the implementation only works in this specific circumstance.