Autonomous motion planning for NVIDIA JetBot

Zachary A Stoebner[a]

[a]Vanderbilt University School of Engineering, Nashville, TN, USA

ABSTRACT

Not only is autonomous driving a major technological fixation of society in the current day but it is also a booming area of research, with obvious practical application. For vehicles to be fully autonomous, a number of problems related to perception, decision-making, and notably path planning must be solved in tandem. This project addresses the game-theoretic path planning aspect of autonomous driving and realizing path planning on an NVIDIA JetBot, an open-source AI robot. Once the path is planned, the vehicular model must then move accordingly in reality to complete the path; implementing such motion planning is the primary endgoal of this project. It was found that JetBot can plan paths with contemporary software and can be programmed to move in a rudimentary fashion along those paths on a real grid layout. Although the results of this project are not groundbreaking, they are at least a proof-of-concept and elucidate promising future improvements for motion planning on JetBot.

Keywords: motion, autonomy, game theory, robotics

1. INTRODUCTION

1.1 Background

Path planning in autonomous vehicles is a booming research area with significant developments.[1–3] Although computer vision and machine learning are often employed to plan motion in autonomous vehicles,[4–6] computationally solving the optimization problems, that arise from in scenarios of motion planning, through a game theoretic is a lightweight alternative to path solving.[7] In this scenario, the path planning optimization problem is formulated as a nonlinear complementarity problem (NCP) constrained by physics and simple car dynamics, which cannot necessarily be directly and exactly solved. Instead the NCP can be approximated by linear mixed complementarity problems (LMCPs), iteratively computing partial paths that together approximate the solution to the NCP and yield a motion planned trajectory for an autonomous vehicle.

The problem formulation is a non-visual scenario where stationary obstacles are laid out on a grid, in a predetermined fashion, and an optimal path must be computed through these obstacles to some goal point without exceeding bounds. Such paths are often nonlinear and can be closely approximated by solving linear mixed complementarity problems via a pathsolver algorithm. Once the path is determined, the JetBot must then move in a real setting, of which the software representation of the grid space is a projection. Realizing the

1.2 Motivation

The main goals of this project were to 1. assemble to the NVIDIA JetBot into a working condition to implement programmatic operation, and 2. implement non-visual path planning based on prior knowledge of an obstacle course. Initially, a reach goal for visual path planning in a dynamic environment was set, which is set for the future due to current limitations. To implement non-visual path planning, one subgoal was to utilize a pathsolver[7] and CasADi[8] to generate a path through a predetermined grid of stationary obstacles. With the path points in hand, the next, and most important, subgoal was to implement motion planning To my knowledge, no existing implementation of game-theoretic motion planning exists for NVIDIA JetBot and it appears that, opting for computer vision-based methods, few have proposed implementations of motion planning from pathsolvers on JetBot. Therefore, this project is a novel endeavor for JetBot users and applications.

Further author information: (Send correspondence to Z.A.S.)

Z.A.S.: E-mail: zachary.a.stoebner@vanderbilt.edu

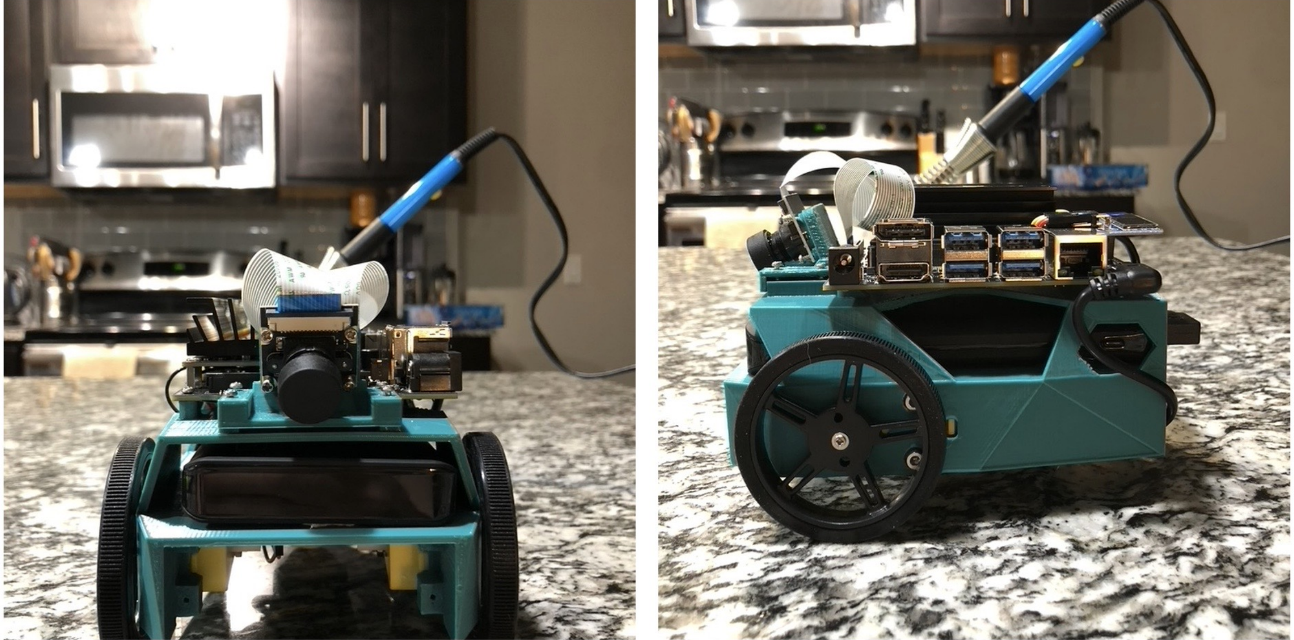

Figure 1. Fully assembled JetBot. The two views show the camera, ports, wheels, and overall build structure of the JetBot. In the background is the soldering iron used during assembly.

2. METHODS

2.1 Hardware Setup

The JetBot was built following the documentation on the JetBot homepage. For the parts with multiple options: the IMX219-160 listed as the second option for cameras, the M2 card + antennas listed as the first option for wifi, and the 65mm wheels listed as the second option for wheels were used. The total cost was approximately $300. The hardware setup time was approximately twelve hours spread between two days. A significant portion of the time was spent extracting a screw terminal from the motor board that was placed incorrectly. 1 shows the completed JetBot hardware assembly.

2.2 Software Setup

The OS for the JetBot was flashed onto a 64 GB SD card in a two step process. First, the NVIDIA Jetson Nano OS was flashed to initialize the Jetson and its the drivers. Second, the JetBot OS was flashed over the Jetson OS on the SD card to initialize the JetBot. On first startup, wifi was configured from the command line; on subsequent startups, the JetBot would automatically connect to the network and could be interfaced through JupyterLab on a browser at the JetBot's IP address. Total software setup time took about 3.5 hours.

2.2.1 CasADi on JetBot

The JetBot OS (Ubuntu 18.04 LTS - aarch64) is not supported by any current binary distributions of CasADi.[8] With much investigative effort, it was possible to build CasADi from source, mostly following the instructions found on the CasADi GitHub wiki. Total build time took about an hour to complete. The command that yielded a successful build on the JetBot, once all prerequisites and source were installed, was:

```
cmake -DWITH_PYTHON=ON -DWITH_PYTHON3=ON ..
```

2.3 Motion Planning

2.3.1 Path Solver

This notebook contains code for non-visual motion planning – the primary objective of the project. The code relies on an LMCP solver that takes an LMCP formulation in M, q, l, u, x_0 and returns a path of points of z, w, v, t. A pathsolver then iteratively solves LMCPs for Newton points along an overarching path, performing backward linesearch to progress sufficiently down each of these paths towards the predefined goal point.

Solving many LMCPs approximates a nonlinear path, which can be formulated as an NMCP for which the KKT conditions must first be derived. The KKT conditions are formulated symbolically so that KKT function as well as the Jacobian of the KKT can be passed to the pathsolver for sparse JIT evaluation, accelerating runtime. Once the point sequence is acquired, it is passed to a module for JetBot motion planning to attempt to move the JetBot along the equivalent trajectory on a real grid space.

2.3.2 Motion Algorithms

To achieve the best motion possible on the JetBot, various motion planning algorithms were implemented: linear approximation, Manhattan (aka wiggle) motion, and proportional / integrative / derivative (PID) control.[9] For some of these algorithms, the *arctan*2 function is used to compute the angle for turning from one orientation to another. (The full code for the JetBotMotion class is included in the appendix.)

$$\arctan 2(\Delta y, \Delta x) \tag{1}$$

The approximate relevant specifications measured for the JetBot were:

1. With two obstacles, sometimes the pathsolver fails if dt is too small $\implies dt > 0.1$
2. Confirmed that the solved states $[x, y, v_x, v_y]$ closely approximate the dynamics of horizontal motion
3. JetBot moves forward 40cm in 0.75 sec at speed=1
4. JetBot rotates 360 degrees in 1 sec at speed=1

Linear approximation is the approximation of the linear movement along the line between the current point and the next point in the path sequence. The algorithm first computes the angle difference, turns, and then moves in a line toward the next point. This algorithm is intuitive but does not account well for vehicle dynamics.

```
def MoveTo(self,x,y,vx,vy):
    x_diff = x - self.pos[0]
    y_diff = y - self.pos[1]

    if abs(x_diff) > 0.2 or abs(y_diff) > 0.2:
        self.x_slider.value = x_diff
        self.y_slider.value = y_diff

        angle = np.arctan2(y_diff, x_diff)
        self.TurnBy(angle - self.orient)
        self.orient = angle

        time.sleep(0.01)

        dist = sqrt(x_diff**2 + y_diff**2) # takes 0.75 seconds to go 40 cm forward
        t = dist*0.75
        self.ForwardFor(t)

        self.pos = (x,y)
        print('Moved to ', self.pos)
```

Manhattan motion: Manhattan motion is the grid-like travel between two points. It is inspired by the Manhattan distance. It effectively travels the line between subsequent points by traveling the legs of the right triangle formed by the distance between them. Unfortunately, this type of movement leaves a huge margin for error due to lack of smoothness in trajectory and more imprecise movement.

```
def Manhattan(self,x,y,vx,vy):
    x_diff = x - self.pos[0]
    y_diff = y - self.pos[1]

    tx = abs(x_diff)
    ty = abs(y_diff)
    if x_diff > 0.0:
        self.LeftFor(tx)

        if y_diff > 0.0:
            self.LeftFor(ty)
        elif y_diff < -0.0:
            self.RightFor(ty)

    elif x_diff < -0.0:
        self.RightFor(tx)

        if y_diff > 0.0:
            self.LeftFor(ty)
        elif y_diff < -0.0:
            self.RightFor(ty)

    self.pos = (x,y)
    print('Moved to ', self.pos)
```

PID control: Similar to linear approximation, PID control computes the angle difference between two points. However, this method does not use integration. Rather, the the PID control is simply the sum of the proportion of the angle by current steering value and derivative of the change in angle, which incorporates the change in steering into the equation, computed in line 10 of this algorithm. PID control is more sophisticated than the other two algorithms; however, it is more advanced and therefore may be more challenging to debug and fully utilize in practice.

```
def MoveWithPIDTo(self,x,y, vx, vy):
    x_diff = x - self.pos[0]
    y_diff = y - self.pos[1]
    self.x_slider.value = x_diff
    self.y_slider.value = y_diff

    self.speed_slider.value = self.speed_gain_slider.value

    angle = np.arctan2(y_diff, x_diff)
    pid = angle * self.steering_gain_slider.value + (angle - self.orient) * self.
   steering_dgain_slider.value
    self.orient = angle
    print('Turned by angle ', angle)

    self.steering_slider.value = pid + self.steering_bias_slider.value

    self.robot.left_motor.value = max(min(self.speed_slider.value + self.
   steering_slider.value, 1.0), 0.0)
    self.robot.right_motor.value = max(min(self.speed_slider.value - self.
   steering_slider.value, 1.0), 0.0)

```

```
    t = self.sleep_slider.value if vx+vy==0 else sqrt((x_diff**2 + y_diff**2) / (vx**2
     + vy**2)) # time to travel to next point at current velocity
    time.sleep(t)

    self.pos = (x,y)
    print('Moved to ', self.pos)
```

3. RESULTS

3.1 Basic Movement

Prior to attempting the solution trajectories from the pathsolver, basic movement with linear approximation was first implemented. 2 shows the movement on the grid on a test sequence of points. Although the JetBot moves to the first point correctly, subsequent movements are mostly incorrect.

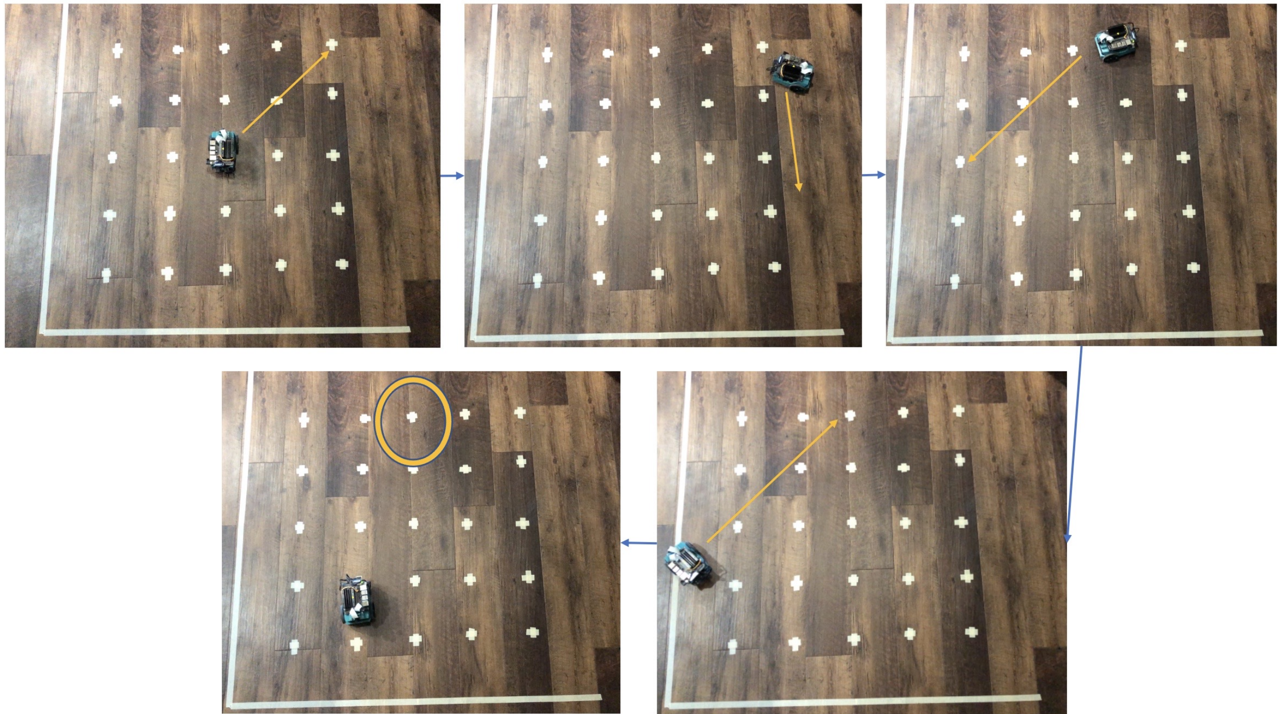

Figure 2. JetBot movement sequence for $\{(1,1),(1,0),(-1,0),(-1,-1),(0,1)\}$. Blue arrows indicate scene flow. The yellow arrow indicates the point that the JetBot should go to next; the yellow circle indicates the final location.

3.2 Linear Approximation

The primary algorithm used for computing movement along the pathsolver trajectories, linear approximation was put against single- and double-obstacle courses. 3 and 4 display the results of attempted motion with linear approximation on these courses. As with basic motion, the JetBot makes the correct first move in both and then enters the endless circle of death, never to reach its goal.

3.3 Manhattan Motion

Manhattan motion did not effectively follow the pathsolver trajectory, even at the initial point. Often, the wiggle motion would push the JetBot in reverse if it wiggled too fast. Although the angles should have been identical as the JetBot is only making right turns, often times the JetBot would overshoot and send itself off course during Manhattan motion. 5 shows the results of Manhattan motion for a trajectory through a double-obstacle course.

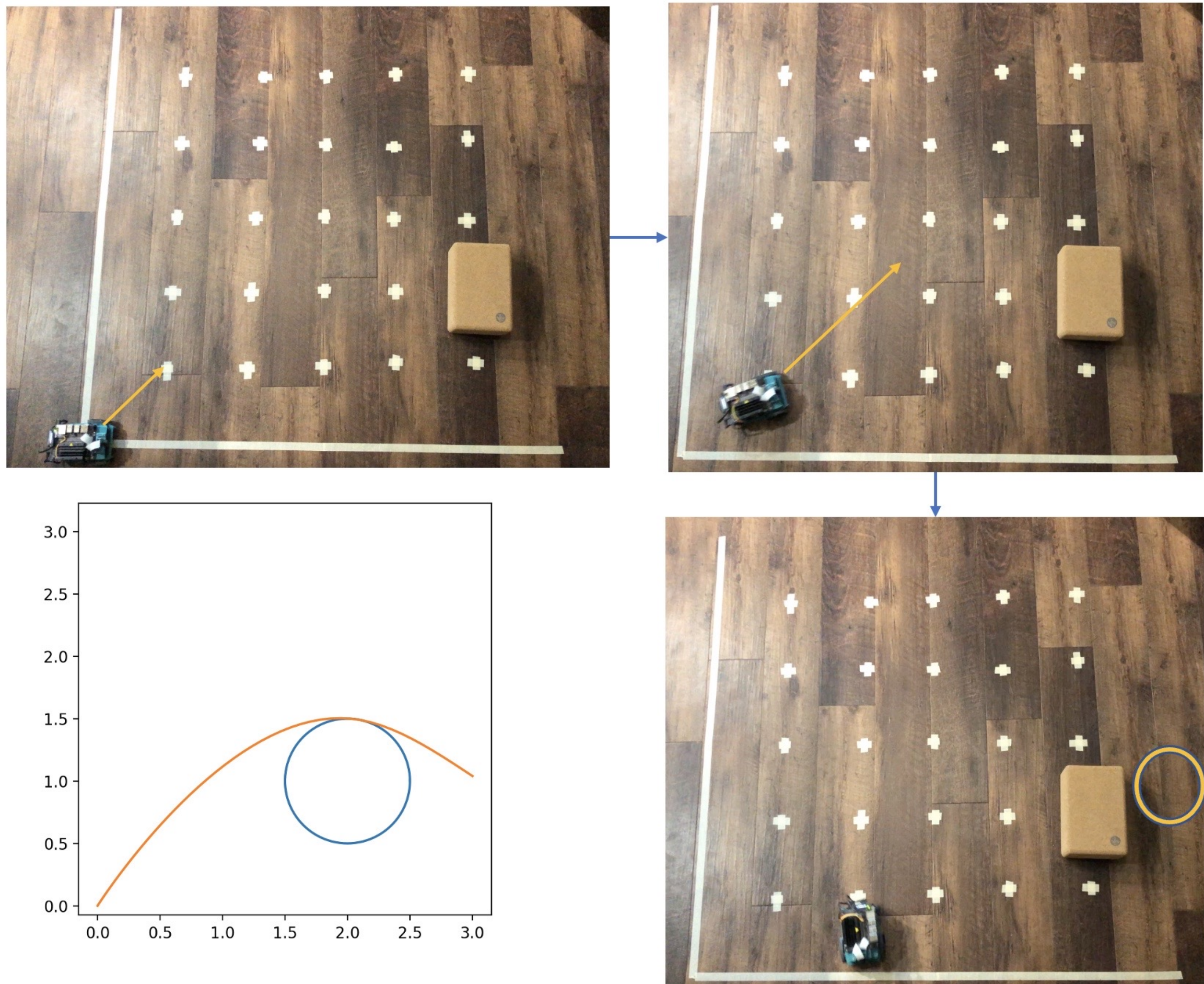

Figure 3. Attempted JetBot linear approximation movement on a single obstacle course. The bottom left pane is the predicted trajectory from the pathsolver. Blue arrows indicate scene flow. The yellow arrow indicates the point that the JetBot should go to next; the yellow circle indicates the final location.

3.4 PID Control

Luckily, PID control resulted in a decent realistic trajectory. However, after analysis, it appears that the lame right wheel assisted in a harder right turn, bringing it closer to the goal. Technically, PID should have gone further straight and overshoot the top of the grid. 6 shows the results of PID control movement for a trajectory through a double-obstacle course.

4. DISCUSSION

The big achievements that resulted from this project were a working JetBot assembly, a working CasADi resource on the JetBot, and motion planning fundamentals. For basic motion, the JetBot can in fact move, even with only linear approximation and not a more sophisticated motion algorithm. However, it is likely that there are bugs, beyond lack of physics consideration, within the algorithm, particularly in the angle computation from the prior point given the current orientation, that contribute to movement inaccuracy. Starting out in the trajectories, the JetBot typically moves in the right direction by a corresponding angle. However, poor understanding of the physics of the JetBot likely lead to inefficiencies in the turning and other operations that ultimately lead to an

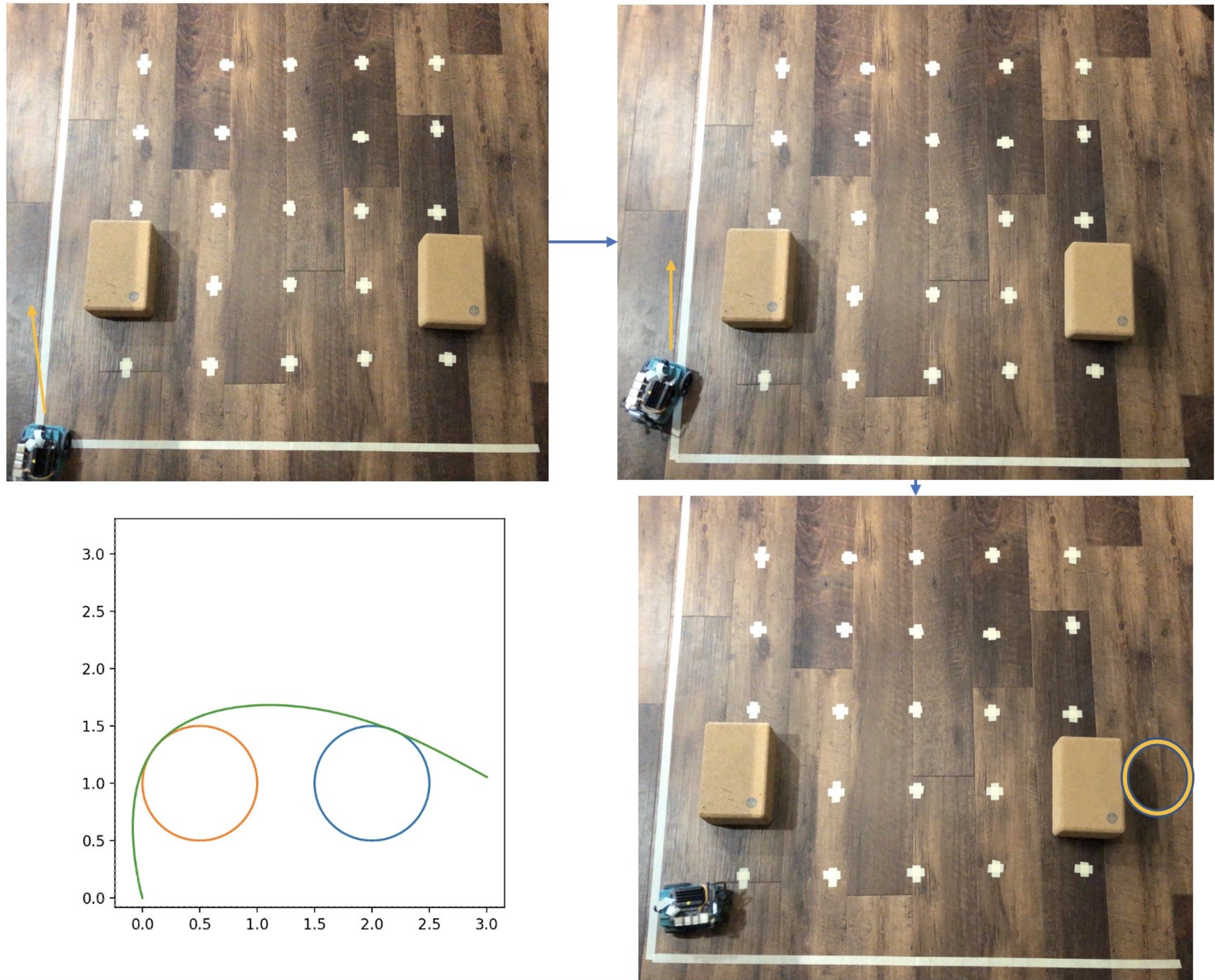

Figure 4. Attempted JetBot linear approximation movement on a double obstacle course. The bottom left pane is the predicted trajectory from the pathsolver. Blue arrows indicate scene flow. The yellow arrow indicates the point that the JetBot should go to next; the yellow circle indicates the final location.

incorrect trajectory. Although the work needed to accomplish optimal motion planning for JetBot is immense, the current pitfalls and next steps to resolve them are fairly straightforward.

4.1 Limitations

JetBot's memory is limited to 1.4 GB after the full OS flash, not including many basic Python packages. Moving forward, flash OS to a much larger SD card, i.e., a 128 GB SD card. JetBot OS doesn't support available CasADi binaries, which, although ultimately not a problem for this project, attributed some strain to the project as it required careful management of any additional packages downloaded to the JetBot, especially in tandem with low memory.

JetBot has extremely limited, primitive motion. The SDK only defines functions for moving each individual motor at a certain speed. Especially from a novice's standing, lack of predefined intelligent motion, and generally any impressive pre-implemented functionality, for the JetBot is inhibitory in defining higher-level function, e.g., comprehensive motion planning. Not a lame wheel induced a turn bias that empirically skewed results but lack of specifications for torque, friction, weight, etc. also hindered the acquisition of .

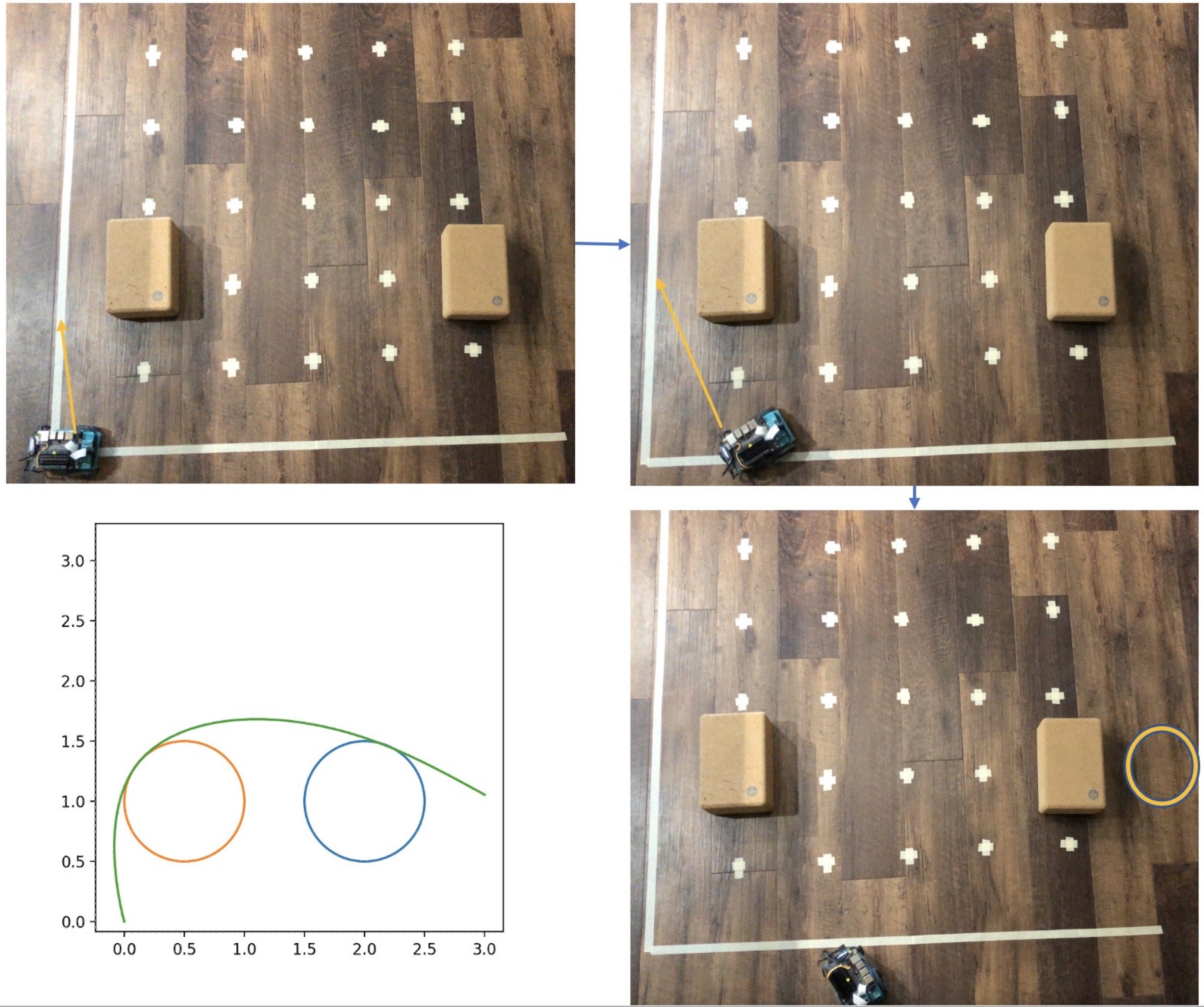

Figure 5. Attempted JetBot Manhattan movement on a double obstacle course. The bottom left pane is the predicted trajectory from the pathsolver. Blue arrows indicate scene flow. The yellow arrow indicates the point that the JetBot should go to next; the yellow circle indicates the final location.

The primary obstacle preventing accomplishment of the reach goal, the JetBot's camera functionality is currently bugged and unable to gather visual data. Without the ability to see, the JetBot cannot even complete most of the baseline examples for collision avoidance, road following, and object following. The results of these examples would act as useful resources for any visual path planning algorithms that might be developed for JetBot in the future.

5. FUTURE IMPROVEMENTS

Given the trajectory of this project, it could benefit from a number of future improvements that will ideally solve critical issues currently facing motion planning on JetBot. First and foremost, fixing camera and memory issues will easily elevate the capabilities of the JetBot, which will lead to new software that greatly improves its motion. Secondly, more precise measurements and accounting for real-world physics could make or break effective motion planning. Thirdly, extension to N-player path solving that updates based on vision data could robustify the motion planning performance in the real world. As always, the motion planning method could benefit from more optimized algorithmic control and additional sensors to offer more information that could be

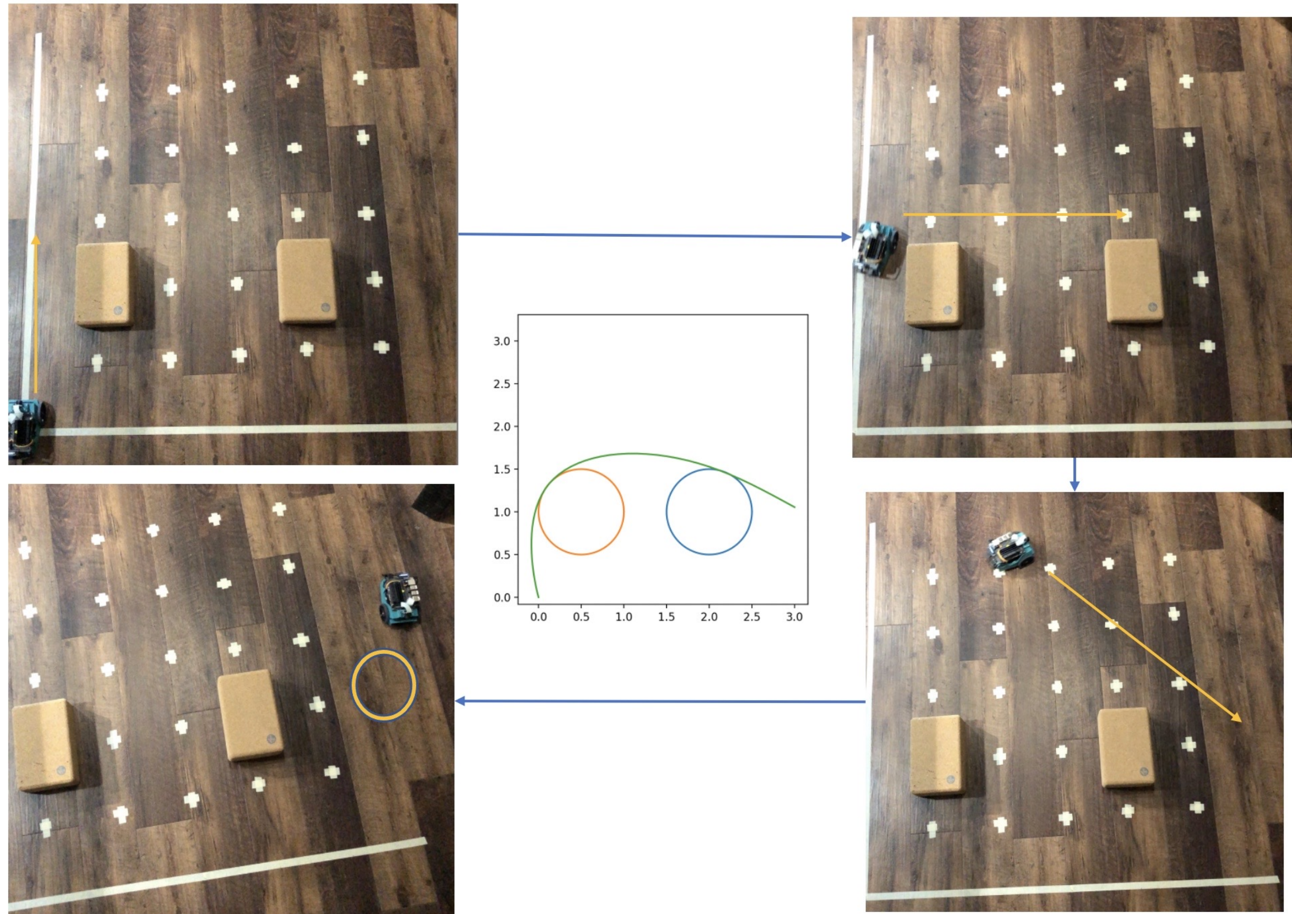

Figure 6. Attempted JetBot PID control movement on a double obstacle course. The middle pane is the predicted trajectory from the pathsolver. Blue arrows indicate scene flow. The yellow arrow indicates the point that the JetBot should go to next; the yellow circle indicates the final location.

essential to good movement for the JetBot. Despite the shortcomings, the path ahead for this project is clear and with some dedication and patience JetBot may outshine other autonomous drones.

REFERENCES

[1] Pepy, R., Lambert, A., and Mounier, H., "Path planning using a dynamic vehicle model," in [*2006 2nd International Conference on Information Communication Technologies*], **1**, 781–786 (2006).

[2] Choset, H., La Civita, M., and Park, J., "Path planning between two points for a robot experiencing localization error in known and unknown environments," (11 1999).

[3] Lozano-Perez, T., "A simple motion-planning algorithm for general robot manipulators," *IEEE Journal on Robotics and Automation* **3**(3), 224–238 (1987).

[4] Yonetani, R., Taniai, T., Barekatain, M., Nishimura, M., and Kanezaki, A., "Path planning using neural a* search," in [*International Conference on Machine Learning*], 12029–12039, PMLR (2021).

[5] Lee, L., Parisotto, E., Chaplot, D. S., Xing, E., and Salakhutdinov, R., "Gated path planning networks," in [*International Conference on Machine Learning*], 2947–2955, PMLR (2018).

[6] Mansouri, S. S., Kanellakis, C., Fresk, E., Kominiak, D., and Nikolakopoulos, G., "Cooperative coverage path planning for visual inspection," *Control Engineering Practice* **74**, 118–131 (2018).

[7] Dirkse, S. and Ferris, M., "The path solver: A non-monotone stabilization scheme for mixed complementarity problems," *Optimization Methods and Software* **5** (12 1993).

[8] Andersson, J. A. E., Gillis, J., Horn, G., Rawlings, J. B., and Diehl, M., "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation* (In Press, 2018).
[9] Araki, M., "Pid control," *CONTROL SYSTEMS, ROBOTICS, AND AUTOMATION* **2**.

6. APPENDIX

```
from math import pi, atan2, sqrt, degrees
import time
import numpy as np

from IPython.display import display
import ipywidgets
import traitlets

from jetbot import Robot, Camera, bgr8_to_jpeg
import cv2
import numpy as np
import torchvision

mean = 255.0 * np.array([0.485, 0.456, 0.406])
stdev = 255.0 * np.array([0.229, 0.224, 0.225])

normalize = torchvision.transforms.Normalize(mean, stdev)

def preprocess(camera_value):
    global device, normalize
    x = camera_value
    x = cv2.cvtColor(x, cv2.COLOR_BGR2RGB)
    x = x.transpose((2, 0, 1))
    x = torch.from_numpy(x).float()
    x = normalize(x)
    x = x.to(device)
    x = x[None, ...]
    return x

class JetBotMotion:
    def __init__(self, robot, dt, orient=0):
        self.robot = robot # jetbot robot object
        self.dt = dt
        self.pos = (0,0)
        self.init_orient = orient
        self.orient = orient # orientation relative to the origin

        self.speed_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01,
    value=0.1, description='speed gain')
        self.steering_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step
    =0.01, value=0.2, description='steering gain')
        self.steering_dgain_slider = ipywidgets.FloatSlider(min=0.0, max=0.5, step
    =0.001, value=0.0, description='steering kd')
        self.steering_bias_slider = ipywidgets.FloatSlider(min=-0.3, max=0.3, step
    =0.01, value=0.0, description='steering bias')

        display(self.speed_gain_slider, self.steering_gain_slider, self.
    steering_dgain_slider, self.steering_bias_slider)

        self.x_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='x')
```

```
46         self.y_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical',
     description='y')
47         self.steering_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='
    steering')
48         self.speed_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='
    vertical', description='speed')
49         self.sleep_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.001, value
    =0.1, description='sleep')
50 
51         display(ipywidgets.HBox([self.y_slider, self.speed_slider]))
52         display(self.x_slider, self.steering_slider, self.sleep_slider)
53 
54     def MoveTo(self,x,y,vx,vy):
55         x_diff = x - self.pos[0]
56         y_diff = y - self.pos[1]
57 
58         if abs(x_diff) > 0.2 or abs(y_diff) > 0.2:
59             self.x_slider.value = x_diff
60             self.y_slider.value = y_diff
61 
62             angle = np.arctan2(y_diff, x_diff)
63             self.TurnBy(angle - self.orient)
64             self.orient = angle
65 
66             time.sleep(0.01)
67 
68             dist = sqrt(x_diff**2 + y_diff**2) # takes 0.75 seconds to go 40 cm
    forward
69             t = dist*0.75
70             self.ForwardFor(t)
71 
72             self.pos = (x,y)
73             print('Moved to ', self.pos)
74 
75 
76     def MoveWithPIDTo(self,x,y, vx, vy):
77         x_diff = x - self.pos[0]
78         y_diff = y - self.pos[1]
79         self.x_slider.value = x_diff
80         self.y_slider.value = y_diff
81 
82         self.speed_slider.value = self.speed_gain_slider.value
83 
84         angle = np.arctan2(y_diff, x_diff)
85         pid = angle * self.steering_gain_slider.value + (angle - self.orient) * self.
    steering_dgain_slider.value
86         self.orient = angle
87         print('Turned by angle ', angle)
88 
89         self.steering_slider.value = pid + self.steering_bias_slider.value
90 
91         self.robot.left_motor.value = max(min(self.speed_slider.value + self.
    steering_slider.value, 1.0), 0.0)
92         self.robot.right_motor.value = max(min(self.speed_slider.value - self.
    steering_slider.value, 1.0), 0.0)
93 
94         t = self.sleep_slider.value if vx+vy==0 else sqrt((x_diff**2 + y_diff**2) / (
```

```
vx**2 + vy**2)) # time to travel to next point at current velocity
95          time.sleep(t)
96
97          self.pos = (x,y)
98          print('Moved to ', self.pos)
99
100     def TurnLeft(self):
101         self.robot.set_motors(0.4,-0.1)
102         time.sleep(1)
103         self.robot.stop()
104
105     def TurnRight(self):
106         self.robot.set_motors(-0.1,0.4)
107         time.sleep(1)
108         self.robot.stop()
109
110     def ForwardFor(self, t):
111         self.robot.forward(1)
112         time.sleep(t)
113         self.robot.stop()
114
115     def LeftFor(self,t):
116         self.TurnLeft()
117         self.ForwardFor(t)
118
119     def RightFor(self,t):
120         self.TurnRight()
121         self.ForwardFor(t)
122
123     def TurnBy(self,angle):
124         if angle < 0:
125             self.robot.right(0.9)
126         elif angle > 0:
127             self.robot.left(0.9)
128
129         time.sleep(abs(angle)/(2*pi))
130         self.robot.stop()
131
132         print('Turned by angle ', degrees(angle))
133
134     def Manhattan(self,x,y,vx,vy):
135         x_diff = x - self.pos[0]
136         y_diff = y - self.pos[1]
137
138         tx = abs(x_diff)
139         ty = abs(y_diff)
140         if x_diff > 0.0:
141             self.LeftFor(tx)
142
143             if y_diff > 0.0:
144                 self.LeftFor(ty)
145             elif y_diff < -0.0:
146                 self.RightFor(ty)
147
148         elif x_diff < -0.0:
149             self.RightFor(tx)
150
```

```
151             if y_diff > 0.0:
152                 self.LeftFor(ty)
153             elif y_diff < -0.0:
154                 self.RightFor(ty)
155 
156         self.pos = (x,y)
157         print('Moved to ', self.pos)
158 
159     def Reset(self):
160         self.pos = (0,0)
161         self.orient = self.init_orient
```