A Contemporary Variation on a Classic Robotics and Declarative Programming Problem

Revision 1.4

Sam Burkhart
Electrical and Computer Engineering
Portland State University
Portland, OR
Email: skb@pdx.edu

Armaan Roshani
Electrical and Computer Engineering
Portland State University
Portland, OR
Email: armaan@pdx.edu

Dr. Marek Perkowski
Electrical and Computer Engineering
Portland State University
Portland, OR
Email: h8mp@pdx.edu

Matt Fleetwood
Electrical and Computer Engineering
Portland State University
Portland, OR
Email: fleet@pdx.edu

David Hernandez
Electrical and Computer Engineering
Portland State University
Portland, OR
Email: dhern2@pdx.edu

***Abstract*—The purpose of this project is to re-create a classic experiment in robotics and Artificial Intelligence (A.I.) known informally as the monkey and banana experiment. Instead of a monkey, we employ a 4-wheeled robot with a front claw. Instead of a banana, the goal is a tall soda can. We utilize Python to interface with sensors (an overhead USB camera) and actuators (robot controls) while Prolog is used to represent the model world and goal rules. A vision system using OpenCV in Python and Aruco markers determines where relevant objects are located. Python queries Prolog with position and angular information in order to update the state of the world and returns the actions the robot should take in order to acquire the can based on the Prolog rules describing the scenario. The robot actions are then sent over a network socket to the robot, which is listening for incoming action commands.**

***Index Terms*—OpenCV, Aruco, computer vision, Prolog, Python, sockets, client, server, declarative programming, Object-Oriented Programming, robotics**

I. Introduction

Traditionally there are at least two significant distinctions between programming languages: declarative and imperative. Prolog, or Programming Logic, is used as a declarative language. For instance, Foit explains these programming languages as having "no algorithm that solves [a] problem. Instead, there is a description of a problem ... so the system can deduce the solving of that problem" [1]. In contrast, Python for example can be used in an imperative style. This means instead of describing what the program should do to solve a problem, a description of how the program should solve the problem is provided instead. Using a declarative approach to describing an autonomous robot, one could say the robot avoids obstacles. On the other hand using an imperative description, one could say the robot uses a sonic sensor to detect if an object is within 15 centimeters or less, otherwise it continues along its current path. Declarative descriptions describe what a phenomena does while imperative descriptions entail how a phenomena occurs. Prolog can also be described procedurally, which ultimately "specifies how Prolog answers questions ... the procedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program" [2]. A more detailed discussion on the procedural and declarative aspects of our Prolog program is described below in the Design and Experiment section.

Prolog can be treated as a database that is queried from Python. Pyswip is an existing Python 2 standard for handling this interface [6], [7]. From the Python side of things, the Prolog database is instantiated through an object or Prolog class. That is, to instantiate the Prolog class in Python as the variable prolog, the instruction is:

$$prolog = Prolog() \tag{1}$$

This can be seen in the Pyswip Appendix B figures, where the Pyswip class Prolog is imported on line 1 and an object instantiated on line 3. Lines 5 and 6 add two rules in our knowledge base relating michael to john and gina through the father predicate. Line 8 queries our knowledge base, asking michael is the father of who? and finally print the results on line 10. The output can be seen at the terminal at the bottom of the screen, where a list of python dictionaries is returned, where X is given the value of each valid response (aka john and gina). You can also retrieve a prolog program using the prolog.consult() function.

This can also be seen on line 89 in the sudoko.py program, which is posted on our GitHub. The major differences across different applications in this regard are two-fold:

- The filename will typically be different for applications, and will need to be modified in Python.

- The handling of query results will also vary for distinct applications, since for one query it might make sense for Python to simply check the result as a bool (true or false), to check if the result is a string ('turn left 5 degrees'), or to check if there are multiple solutions (e.g. "John is the father of Mary; John is the father of Jane", etc.).

Aside from the two differences listed above, the use for Pyswip should be exactly the same across different applications that interface Prolog from Python. For instance, consider the sudoku.py and sudoku.pl programs in the Pyswip GitHub repository. This example shows how one can provide a legal sudoku puzzle in Python and then query the Prolog program in order to solve the puzzle. Other than creating and querying the Prolog object and database, Python simply prints a nice picture (using pretty print) of the puzzle's solution. The result of this example puzzle in the provided source code is shown below in Fig. 1. This demonstrates a simple test to ensure Prolog is working correctly when it is called from Python. Python 3 was used in the Anaconda environment and the Spyder IDE (Integrated Development Environment) for part of this project, while Visual Studio Code was used for the majority of it.

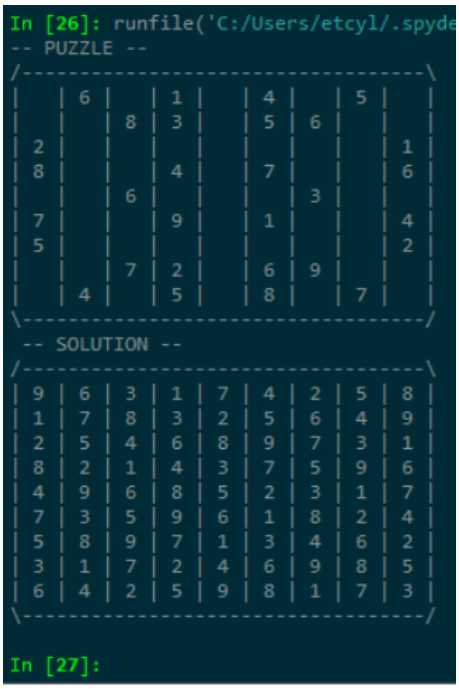

Fig. 1: Input and output sudoku puzzle using Prolog from Python.

A. Related Works

There are several notable examples of related research to this project. For instance, Pineda et. al. since at least 2001 have developed and used an interpreter written in Prolog, called SitLog, which implements what their team calls Dialogue Models (DM) based on their cognitive inspired research [3]. Our approach is different from Pineda's because we employ relatively simple Prolog rules. We also use OpenCV differently to accomplish our vision with Aruco markers, while Pineda's group use OpenCV for face and head detection, tracking, and recognition.

In 1985 Brooks at MIT described a layered control system with "a number of levels of competence for an autonomous mobile robot" [4]. The primary difference between Brooks's physical robot and ours is that we limit our sensor to one USB webcam and our compute platforms to a laptop for Python and OpenCV, and a RaspberryPi 3 for the robot. Brooks used a mobile robot equipped with an array of camera sensors along with an Intel 8031 for the main processor. Brooks also utilizes levels of competence in order to achieve higher overall competence. In contrast, our team developed declarative programs partly procedurally and declaratively at the same time without special regard to levels of competence. We did however develop our project in a somewhat similar sense using distinct phases, which is described in the Design and Experiment section.

Later in 1991, Brooks summarized mobile robots as exploiting "at least a nominal path through the world model", which is extrapolated from sensors [5]. Our work also follows the notion of building a model through sensors. We describe how position and angular data create a vector for different identified objects within the robot's model in the Aruco Markers section below. It suffices to say that the robot's model of the world uses position and angular information for Prolog rules, which determine where the robot should go next to achieve its goal (grabbing the can on a box with a ramp).

II. Description of Prolog Backtracking

A. Family Tree Program in Prolog

Prolog is a powerful declarative language because of its inherent ability for backtracking [17], [18]. As an example, a simple trace output for the family tree program is shown in Figure below. (The complete Prolog program can be found on our GitHub.) For brevity, only one rule is investigated to demonstrate backtracking in this example - the related rule.

```
related(X, Y) :-
    parent(Y, X);
    grandparent(X, Y);
    grandparent(Y, X);
    greatgrandparent(X, Y);
    greatgrandparent(Y, X);
    sibling(X, Y);
    aunt(X, Y);
    aunt(Y, X);
    uncle(X, Y);
    uncle(Y, X);
    sibling(Y, X).
```

Fig. 2: Family tree program rule for defining if two people are related.

This rule says X and Y are related if any of the following are true: Y is a parent of X; Y is a grandparent of X, Y is a sibling of X; and so on (the opposite is also true: if X is a parent of Y, X and Y are also related). In Fig. 3, Prolog repeatedly tries to prove X and Y are related by using the definition of the rules. For instance, Prolog tries to see if matthew is related to

tim by first checking if tim is a parent of matthew. Some X is a parent of some Y if X is a mother or a father of Y. The parent rule is checked first because it is defined as the first goal with a semicolon. The semicolon indicates to Prolog that if this rule succeeds, then the rest of the rules (grandparent, greatgrandparent, sibling, etc.) for the related predicate are not attempted. Prolog backtracks from a rule if it fails, trying the parent rule, then the grandparent rule, and finally the uncle rule, which succeeds for this program.

While backtracking, Prolog references other possible terms by using an underscore and number (some constant value like 2708). This means Prolog is looking for some term that makes a predicate true. For instance, Call (11) sibling(matthew, 2708) ? creep means Prolog is attempting to unify some term with the sibling predicate and the term matthew. In other words, it is checking to see if there is a person who is defined as being the sibling of matthew. Fig. 4 below shows a graphical representation of Prolog processing a query, which uses unification and resolution.

```
?- consult(family_tree).
true.

?- trace.
true.

[trace]  ?- related(matthew, tim).
   Call: (8) related(matthew, tim) ? creep
   Call: (9) parent(tim, matthew) ? creep
   Call: (10) mother(tim, matthew) ? creep
   Fail: (10) mother(tim, matthew) ? creep
   Redo: (9) parent(tim, matthew) ? creep
   Call: (10) father(tim, matthew) ? creep
   Fail: (10) father(tim, matthew) ? creep
   Fail: (9) parent(tim, matthew) ? creep
   Redo: (8) related(matthew, tim) ? creep
   Call: (9) grandparent(matthew, tim) ? creep
   Call: (10) parent(matthew, _2708) ? creep
   Call: (11) mother(matthew, _2708) ? creep
   Fail: (11) mother(matthew, _2708) ? creep
   Redo: (10) parent(matthew, _2708) ? creep
   Call: (11) father(matthew, _2708) ? creep
   Fail: (11) father(matthew, _2708) ? creep
   Fail: (10) parent(matthew, _2708) ? creep
   Fail: (9) grandparent(matthew, tim) ? creep
   Redo: (8) related(matthew, tim) ? creep
   Call: (9) grandparent(tim, matthew) ? creep
   Call: (10) parent(tim, _2708) ? creep
   Call: (11) mother(tim, _2708) ? creep
   Fail: (11) mother(tim, _2708) ? creep
   Redo: (10) parent(tim, _2708) ? creep
   Call: (11) father(tim, _2708) ? creep
   Fail: (11) father(tim, _2708) ? creep
   Fail: (10) parent(tim, _2708) ? creep
   Fail: (9) grandparent(tim, matthew) ? creep
   Redo: (8) related(matthew, tim) ? creep
   Call: (9) greatgrandparent(matthew, tim) ? creep
   Call: (10) grandparent(matthew, _2708) ? creep
   Call: (11) parent(matthew, _2708) ? creep
   Call: (12) mother(matthew, _2708) ? creep
   Fail: (12) mother(matthew, _2708) ? creep
   Redo: (11) parent(matthew, _2708) ? creep
   Call: (12) father(matthew, _2708) ? creep
   Fail: (12) father(matthew, _2708) ? creep
   Fail: (11) parent(matthew, _2708) ? creep
   Fail: (10) grandparent(matthew, _2708) ? creep
   Fail: (9) greatgrandparent(matthew, tim) ? creep
   Redo: (8) related(matthew, tim) ? creep
```

Fig. 3: Trace output for the family tree program.

```
   Redo: (8) related(matthew, tim) ? creep
   Call: (9) uncle(tim, matthew) ? creep
   Call: (10) brother(tim, _2708) ? creep
   Call: (11) male(tim) ? creep
   Exit: (11) male(tim) ? creep
   Call: (11) sibling(tim, _2708) ? creep
   Exit: (11) sibling(tim, victoria) ? creep
   Exit: (10) brother(tim, victoria) ? creep
   Call: (10) parent(victoria, matthew) ? creep
   Call: (11) mother(victoria, matthew) ? creep
   Exit: (11) mother(victoria, matthew) ? creep
   Exit: (10) parent(victoria, matthew) ? creep
   Exit: (9) uncle(tim, matthew) ? creep
   Exit: (8) related(matthew, tim) ? creep
true .

[trace]  ?-
```

Fig. 4: Trace output showing successful unification of the related rule.

B. The Original Banana and Monkey Experiment by Bratko

The banana and monkey experiment is discussed in Chapter 2 of the Bratko text. As the author notes, the Prolog program is "a simple example of problem solving" to "show how the mechanisms of matching and backtracking can be used" in Prolog [2]. In the variation described by Bratko, a monkey standing in a room needs to grab a hanging banana, which is suspended somehow from the ceiling. Since the monkey cannot reach the banana, it needs to locate a box in the room that can be pushed below the banana such that the monkey can climb on the box. Once standing on the box, the monkey can grab the banana, which represents the goal of this scenario.

Bratko describes how the world state of this example can be viewed. For instance, "the initial state of the world is determined by:

- Monkey is at door.
- Monkey is on floor.
- Box is at window.
- Monkey does not have banana."

Similarly, only "four types of moves are allowed:

- Grasp banana.
- Climb box.
- Push box.
- Walk around."

The complete program for this example can be found in PDF format online here. Fig. 5 procedurally (or imperatively) describes how Prolog solves a query for this program. A tree is shown to graphically represent this description. For example, at the root query, the monkey is at the door on the floor, the box is at the window, and the monkey does not have the banana.

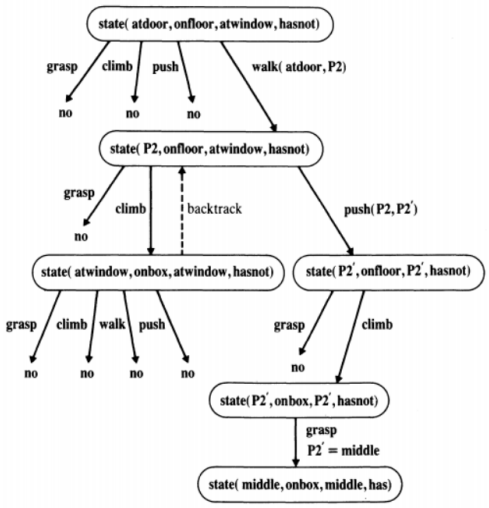

Fig. 5: Original Prolog search chart from Chapter 2 in Bratkos book, [2]

III. Claw Robot and Object-Oriented Programming

The claw robot is a four-wheeled robot using a RaspberryPi to control four TT hobby motors and a front grabber servo. It

is shown below in Fig 6. A white box of the robot can be seen in Fig. 8. The robot is mobile and uses a RaspberryPi 3. The software for this robot was written in Python and facilitates a relatively simple OOP class interface.

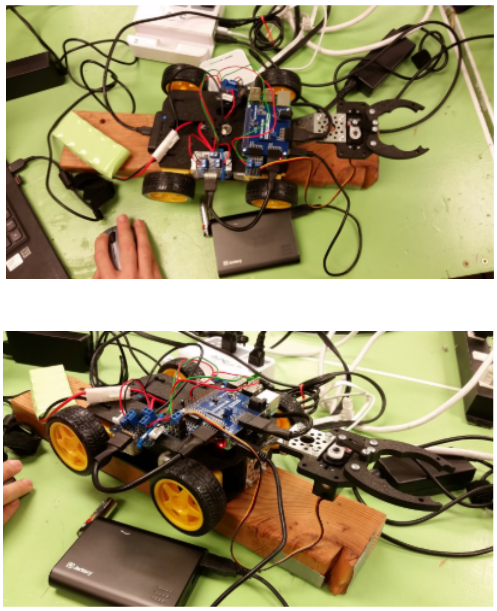

Fig. 6: Images of the claw robot.

The the inspiration for this robot came from a previous course where the frame was controlled by a PYNQ board from Digilent, utilizing Digilent PMODs to control the four hobby motors and the servo for the gripper arm [9]. With the DesignSpark PMOD HAT the motor drivers and servo could be reused [10]. The Raspberry Pi 3 instantiates a Robot class based heavily on the Dancing Hexapod design [12]. The main changes compared to the hexapod design are with respect to the robot controls, which utilize the DesignSpark PMOD HAT library for the Raspberry Pi [11]

The DesignSpark library comes with drivers for the HB3 motor controller. These were used for inspiration to design our own DHB1 driver and CON3 driver. These changes can be seen in the project GitHub here. Both drivers utilize the Raspberry Pi gpio python library PWM class to control the motors and servo. The ClawRobot.py class instantiates each of the three drivers (2x DHB1 and 1x CON3) and defines the interface between the user of the class and the PMOD drivers. There are certain restrictions on what ports that can be utilized to satisfy these requirement. Show in the below figures is how we connected these boards (DHB1 and CON3).

ClawRobot.py

```
16  class ClawRobot:
17      def __init__(self, motors1_port, motors2_port, servos_port):
18          # Left Motors
19          self.motors1 = createPmod('DHB1',motors1_port)
20
21          # Right Motors
22          self.motors2 = createPmod('DHB1',motors2_port)
23
24          # Claw
25          self.servos = createPmod('CON3',servos_port)
```

Fig. 7: ClawRobot.py class initialization code.

```
24  #def initialize_clawrobot(config_file):
25  def initialize_clawrobot():
26      my_clawrobot = ClawRobot(motors1_port='JC', motors2_port='JB', servos_port='JAB')
27      my_clawrobot.stop()
28
29      return my_clawrobot
30
```

Fig. 8: ClawRobot.py class initialization code.

We decided to have each DHB1 board control only one side (Left/Right) of the 4-wheel claw robot. This allowed for easier turning commands. We then assigned port locations to the motors connected to the DHB1 boards, i.e. port JC of PMOD HAT is connected to the left motors (BackLeft: motor1 and FrontLeft: motor2). Similarly CON3 board is connected to port JAB (bottom set of pins on port JA) which is then connected to servo1 the gripper arm.

The ClawRobot class is instantiate by a wrapper function that sets up a server socket on the host, which waits for incoming commands, and then utilizes the ClawRobot instance to send the commands to the robot. On our GitHub, the server application program can be found here. The valid commands available are forward, backward, left, right, open, close. The movement commands have flags available for delay (how long to perform the action) and power (how strong to drive the motors).

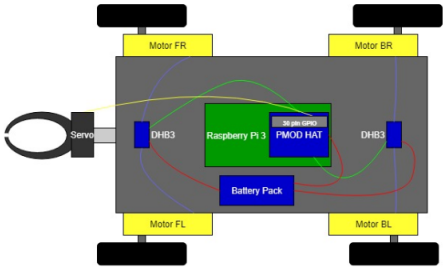

Fig. 9: White box diagram of the claw robot.

IV. ARUCO MARKERS IN OPENCV USING PYTHON

ArUco uses a binary square fiducial algorithm to find the markers. Through a series of steps, the ArUco library identifies markers in a scene, localizes them a 3D space, and provides OpenCV matrices of their ID numbers, coordinates of each markers four corners, and transformation matrices for each markers orientation and angle with respect to the cameras coordinate system. Details on the ArUco marker library design can be found in “Automatic generation and detection of highly reliable fiducial markers under occlusion” [15].

In order to use the OpenCV ArUco library effectively, the camera must be calibrated. The calibration can be performed using a classic chess board pattern, or with a merge of an ArUco board and chess board pattern called a ChArUco board. We used this method and the program can be found here. The print_ ChArUco_board.py was used to generate the charuco.png image shown below.

Fig. 10: Image of the Aruco markers for the can and robot.

We used a central overhead camera to orient our robots with objects around them using OpenCV and ArUco. OpenCV is an OpenSource Computer Vision Library, and ArUco is an OpenSource library for pose estimation. ArUco was originally intended to estimate the pose of a camera in relation to a scene using grids of special square markers. Features of it can be used in reverse though: to use special square markers attached to objects in a scene to orient those objects in relation to the grid pixels in the camera. OpenCV and the OpenCV ArUco library are both implemented in Python as wrappers. We are using those wrappers to generate our ArUco markers, and find them in video from our camera.

The Fig. 17 in Appendix D was used in various orientations while running the code in calibrate_camera.py. This script collects all of the images necessary to perform the calibration. Due to the length of time that calibration takes it was necessary to write the calibration input data to a yaml file (results.yml). The script in calibrate_camera_from_file.py reads in the file generated from calibrate_camera.py and performs the calibration, outputting the calibration matrix which is in turn stored in yaml files (camera_matrix.yml, camera_calibration_results.yml). This is used when getting the pose estimation in the main detection code. A sample of the aruco detection code is available in the directory here. For each marker detected we can retrieve the corners and ID of each. From this we are able to determine the orientation of the marker in 3D space and the ID.

V. Design and Experiment

We attempt to re-create a variation of the original banana and monkey experiment by using the claw robot, a box, a ramp, Prolog for the hierarchy of rules, and Python for I/O (Input/Output). A client-server interface sends commands from the computer to the robot and is shown below in Fig. 12.

A. Flowchart and Description

Our project flowchart can be seen below in Fig. 8. The system begins execution with reading from a sensor, in this case a USB camera. The image stream from the camera is then fed into a identification module within Python. This uses Arcuo markers in order to determine whether there are any object matches in the frame. The module checks if any known objects, such as the tall soda can, the robot, the ramp, or the box in our case, are present in the image frame. The (x, y) coordinates and orientation in degrees of any detected objects is also provided. Essentially, this provides the knowledge of where objects are (if any are detected) and what direction they are facing.

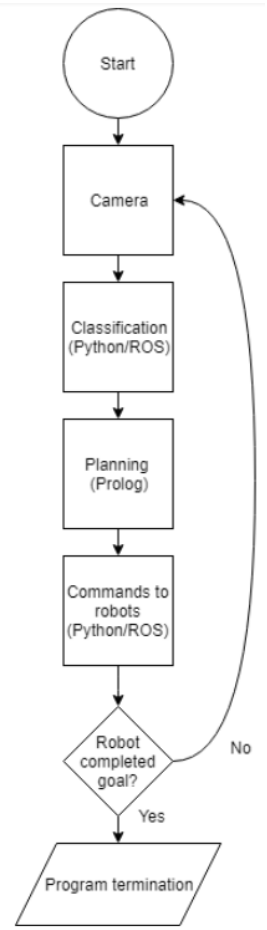

Fig. 11: High-level flowchart for the Python and Prolog system.

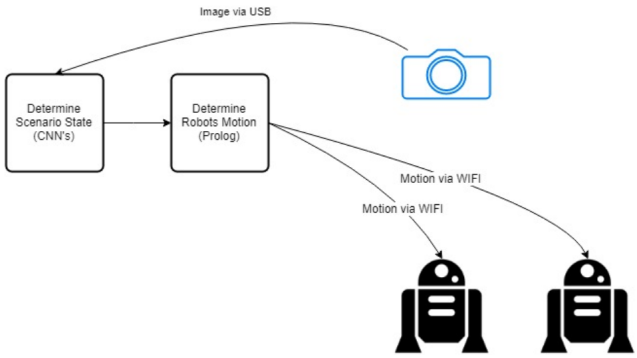

Fig. 12: Black-box for the Python and Prolog system.

This information is then used by the planning module within Prolog. Python queries Prolog using the Aruco information by asking where the robot should go next given the current positions and angles of detected objects. Prolog first checks if the robot has met the goal of the query. This means if the robot is at the goal, i.e. the soda can, then the planning module

returns false because there is nothing further to do. Otherwise, the robot has not yet achieved the goal.

Python receives the result of the query and checks what to do next. The commands are sent from one computer, in our case a laptop. This is achieved using Python sockets. The computer acts as the client by sending commands to the robots, which act as the server. In terms of Robot Operating System (ROS) topics, the robots are listening to the topic, and the computer is sending commands or publishing to this topic. The robots then simply use a Python program to check and decode commands. These include simple movements such as left, right, forwards, and backwards through the aforementioned class interface.

B. Design Phases

Our design goals can be split into the following phases below:

- **Phase 1: Grasping Goal**
  - Initial conditions:
    - Clawbot and Goal on flat surface.
    - No other objects in scene.
  - Objective:
    - Clawbot to traverse scene and grasp Goal.
  - Design of System to accomplish objective:
    - Robot Motion controlled using Client/server setup for one robot.
    - Identify different objects using Aruco Markers.
    - Mapping Directions from robot to object using Prolog.
    - Writing master program using Python.
- **Phase 2: Climbing Ramp and Grasping Goal**
  - Initial conditions:
    - Clawbot on flat surface.
    - Goal hanging above or sitting atop box.
    - Ramp/box combination in fixed orientation with respect to rest of scene, but unfixed position.
  - Objectives:
    - Clawbot to traverse scene to orient itself with ramp.
    - Clawbot to traverse ramp/box combination and grasp Goal.
  - Additions to Design of System to accomplish objectives:
    - Mapping scene parameters, including location of box/ramp combination
    - Mapping Directions for robot to traverse scene using Prolog and Python
    - Updating master program using Python
- **Phase 3: Moving Box Below Goal, Climbing Ramp, and Grasping Goal**
  - Initial conditions:
    - Clawbot and DonkeyCar on flat surface.
    - Goal hanging in position not above box.
    - Ramp/box combination in unfixed orientation or position with respect to rest of scene.
  - Objectives:
    - DonkeyCar to traverse scene and push ramp/box combination to location and orientation such that box is positioned below goal.
    - Clawbot to traverse scene to orient itself with ramp.
    - Clawbot to traverse ramp/box combination and grasp Goal.
  - Additions to Design of System to accomplish objectives:
    - Modify scene mapping scheme to accommodate unfixed ramp orientation

We use a single wide-angle overhead camera above a scene with the robots, the objects, and the goal (the can for the clawbot to grasp).

In the Gridspace Appendix C image, eight ArUco markers and nine boxes drawn in black are visible. The locations of the ArUco markers in the corners of the scene define the four corners of the entire traversable area. The markers on the combined ramp/box object define the four corners of that object. From those two sets of four markers, we develop nine regions of the scene: top left, middle left, bottom left, etc. The centroids of each of the nine regions of the scene are marked with red dots.

VI. Results

A. Conclusions and Findings

The results of this experiment show at least one contemporary approach to solving the classic robotics problem of the banana and monkey. All designs, documents, and source code can be found on our GitHub: https://github.com/armaanhammer/ECE-510-Monkey-and-Banana. All video footage can be found here.

The Aruco system took significant development time. This seemed like the most ideal method to our team at the start of the project. However, it seems likely there are a number of ways OpenCV image detection and tracking could also be used as the vision component. It would be interesting to compare Aruco markers, with respect to performance and development time for an application, with alternative image detection and tracking in OpenCV (such as what Pineda's group did).

Prolog was also used somewhat naively in the sense that all Prolog programs were relatively short and simple. One idea to improve our work could be to use more sensors, for instance additional cameras angled differently, in order to drive the declarative aspect of the system fruitfully. Using more Aruco markers to identify extra objects could also fuel a more interesting Prolog system to reason about its environment.

B. Future Work

The DonkeyCar is an open source robot project used to explore autonomous vehicle design and algorithms [13]. The source code is available on github [14]. The main chassis

and components (battery, motor, servo, ESC) are built on an existing RC car (The Exceed Magnet). The main compute board is a Raspberry Pi 3 with a wide-angle Pi Camera for vision applications, and a PCA9685 16x12-bit PWM controller board connected to the Raspberry Pi I2C interface. The two PWM signals used to control the DonkeyCar are the Steering Servo (changes the angle of the front wheels) and the ESC input (drives the motor, which in turn drives all 4 wheels). These differences from the 4-wheeled claw robot design mean that the Donkey Car has much more torque available to travel at greater speeds or push a heavier load, but has much coarser controls making fine grained movements difficult.

The main DonkeyCar project includes a very complicated infrastructure for adding sensors, simulating the cars movements, and training machine learning models using Keras based on the camera inputs and other sensor values. For this project we simplified this code to create a similar interface to the 4-wheeled claw robot. The code for this implementation can be found here. Rather than having a left/right movement, there are only two movement functions: forward and reverse. Each includes a delay (how long to execute the command) and and angle. This means to turn to the left or right is a function of the turning radius of the car and would require a circling motion or multi-point turn.

Time constraints have restricted DonkeyCar use in this project, but the work done to build the robot and design the interface could easily be extended to treat this robot as a secondary robot in the monkey/banana scenario. Also, the visual capabilities of the donkey car could allow for a visual exploration of the world rather than relying on a world view overhead camera.

APPENDIX A

TABLE I: Bill of Materials (BOM) for the Claw Robot.

Part	Quantity	Cost (USD)	Link
Raspberry Pi 3	2	39.99	amazon.com
Pmod HAT Adapter	1	14.99	store.digilentinc.com
DHB1 Pmod	2	16.99	store.digilentinc.com
CON3 Pmod	1	4.99	store.digilentinc.com
Runt Rover Junior	1	27.99	amazon.com
Motor Battery Pack	1	26.95	robotshop.com
Pi Battery Pack	1	39.95	adafruit.com
Gripper Claw	1	6.99	servocity.com
Standard Servo	1	11.49	servocity.com
Cables, nuts, bolts	NA	20.00	NA

APPENDIX B

```
sample_pl_to_py_test.py
1  from pyswip import Prolog
2
3  prolog = Prolog()
4
5  prolog.assertz('father(michael, john)')
6  prolog.assertz('father(michael, gina)')
7
8  result = list(prolog.query('father(michael, X)'))
9
10 print(result)
```

Fig. 13: Pyswip example 1.

```
(cv) sburkhar@sburkhar-SchoolLaptop:~/github/ECE-510-Monkey-and-Banana/Code/Pyswipl$ python sample_pl_to_py_test.
py
[{'X': 'john'}, {'X': 'gina'}]
(cv) sburkhar@sburkhar-SchoolLaptop:~/github/ECE-510-Monkey-and-Banana/Code/Pyswipl$
```

Fig. 14: Pyswip example 2.

APPENDIX C

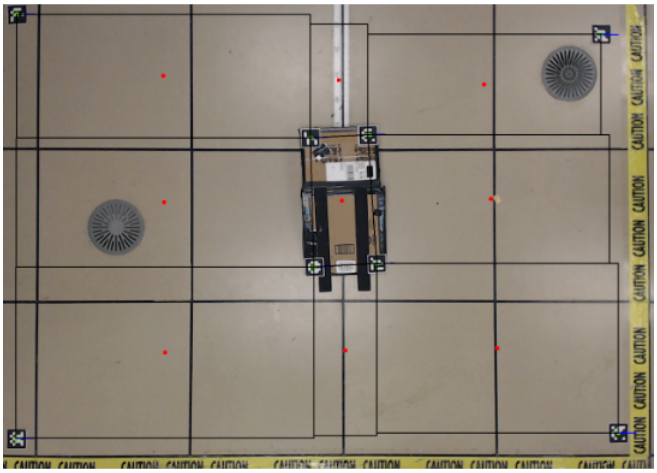

Fig. 15: Gridspace for Phase 1.

APPENDIX D

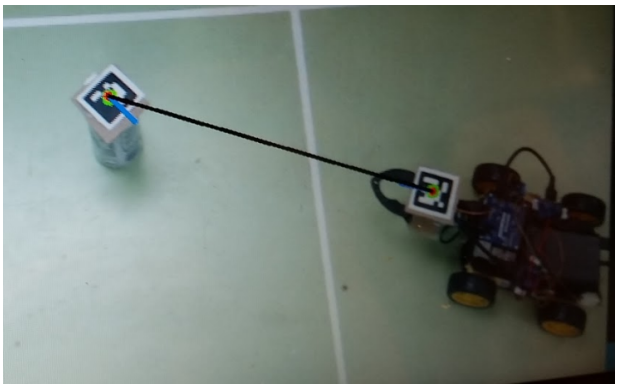

Fig. 16: Output showing the vector in OpenCV.

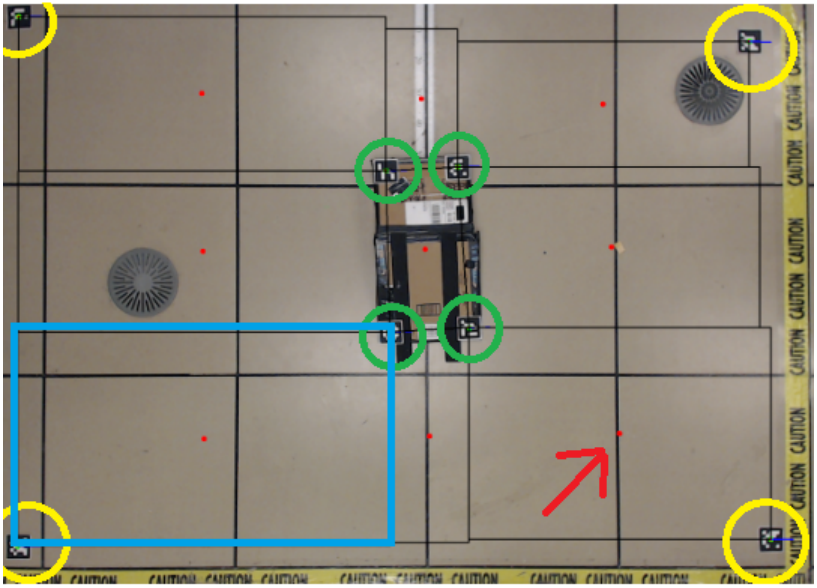

Fig. 17: Image of a generated ChArUco board.

ACKNOWLEDGMENT

Thank you Marek Perkowski for lab use and oversight of this project.

REFERENCES

[1] K. Foit, “Introduction to solving task-level programming problems in logic programming logic,” Journal of Achievements in Materials and Manufacturing Engineering, vol. 64, no. 2, pp. 7884, Jun. 2014.
[2] I. Bratko, Prolog programming for artificial intelligence. Harlow, England: Addison-Wesley, 2012.
[3] L. A. Pineda, C. Rascon, G. Fuentes, V. Estrada, A. Rodriguez, I. Meza, H. Ortega, M. Reyes, M. Pena, J. Duran, E. Campos, S. Chimal, and A. Orozco, The Golem Team, RoboCup Home 2014, pp. 17, 2014.
[4] R. A. Brooks, A robust layered control system for a mobile robot. MIT Cambridge, 1985.
[5] R. A. Brooks, Intelligence without reason, MIT Cambridge, 1991.
[6] OpenCV: Detection of ArUco Markers, OpenCV: Image Thresholding, 00-Dec-2015. [Online]. Available: https://docs.opencv.org/3.1.0. [Accessed: 11-Jun-2018].
[7] Pyswip GitHub. https://github.com/yuce/pyswip.
[8] SWI Prolog API. http://www.swi-prolog.org/pldoc/doc_for?object=manual
[9] https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq/
[10] https://reference.digilentinc.com/reference/add-ons/pmod-hat/start
[11] https://github.com/DesignSparkrs/DesignSpark.Pmod
[12] https://github.com/vivanbhalla/Dancing-Hexapod
[13] http://www.donkeycar.com/
[14] https://github.com/wroscoe/donkey
[15] S. Garrido-Jurado, R. Muoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marn-Jimnez. 2014. ”Automatic generation and detection of highly reliable fiducial markers under occlusion”. Pattern Recogn. 47, 6 (June 2014), 2280-2292. DOI=10.1016/j.patcog.2014.01.005
[16] P. Norvig and S. J. Russell, Artificial Intelligence: A Modern Approach (3rd Edition): Stuart Russell, Peter Norvig: 8601419506989: Amazon.com: Books. .
[17] F. C. N. Pereira and S. M. Shieber, Prolog and Natural-Language Analysis. Brookline, MA: Microtome Publishing, 1987.
[18] U. Endriss, Lecture Notes An Introduction to Prolog Programming, rep., Institute for Logic, Language and Computation, Universiteit van Amsterdam, 2016.