A Contemporary Variation on a Classic Robotics and Declarative Programming Problem

Revision 0.9

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

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

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

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

Dr. Marek Perkowski
Electrical and Computer Engineering
Portland State University
Portland, OR
h8mp@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. Here, we detail how Prolog can be integrated into Python in order to conduct a current variation of this experiment. Prolog is used as a representation for the robot's world knowledge and as a means to determine what the robot needs to do next. Python is used to query Prolog. A vision system using OpenCV and Aruco markers determines where relevant objects (e.g. the soda container, the ramp, the box, etc.) are located. Python uses this information to query a Prolog program, which provides the direction (such as left, right, backwards, forwards, and so on) the robot should take in order to acquire the goal object. In this case, the goal is a tall soda container. Python sockets were used for sending commands to the robot.**

***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. 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 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).

```
In [26]: runfile('C:/Users/etcyl/.spyde
-- PUZZLE --
/-----------------------------------\
|   | 6 |   | 1 |   | 4 |   | 5 |   |
|   |   | 8 | 3 |   | 5 | 6 |   |   |
| 2 |   |   |   |   |   |   |   | 1 |
| 8 |   |   | 4 |   | 7 |   |   | 6 |
|   |   | 6 |   |   |   | 3 |   |   |
| 7 |   |   | 9 |   | 1 |   |   | 4 |
| 5 |   |   |   |   |   |   |   | 2 |
|   |   | 7 | 2 |   | 6 | 9 |   |   |
|   | 4 |   | 5 |   | 8 |   | 7 |   |
\-----------------------------------/
 -- SOLUTION --
/-----------------------------------\
| 9 | 6 | 3 | 1 | 7 | 4 | 2 | 5 | 8 |
| 1 | 7 | 8 | 3 | 2 | 5 | 6 | 4 | 9 |
| 2 | 5 | 4 | 6 | 8 | 9 | 7 | 3 | 1 |
| 8 | 2 | 1 | 4 | 3 | 7 | 5 | 9 | 6 |
| 4 | 9 | 6 | 8 | 5 | 2 | 3 | 1 | 7 |
| 7 | 3 | 5 | 9 | 6 | 1 | 8 | 2 | 4 |
| 5 | 8 | 9 | 7 | 1 | 3 | 4 | 6 | 2 |
| 3 | 1 | 7 | 2 | 4 | 6 | 9 | 8 | 5 |
| 6 | 4 | 2 | 5 | 9 | 8 | 1 | 7 | 3 |
\-----------------------------------/

In [27]:
```

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 2 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. 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.

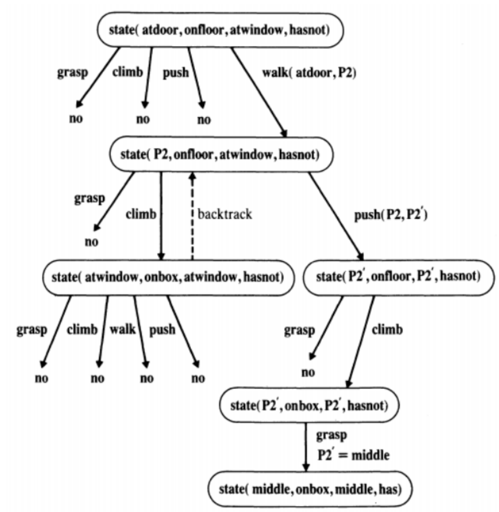

Fig. 5: Original Prolog search chart from Chapter 2 in Bratkos book, [2]

III. Donkey Car and Claw Robot: Object-Oriented Programming

There is one robot used for this project. It is called the Claw Robot and is shown below in Fig 6. The robot is mobile and uses a RaspberryPi 2. The software for this robot was written in Python and facilitates a relatively simple OOP class interface. Ultimately, this means that commands are easily performed in functions such as turn left, turn right, etc.

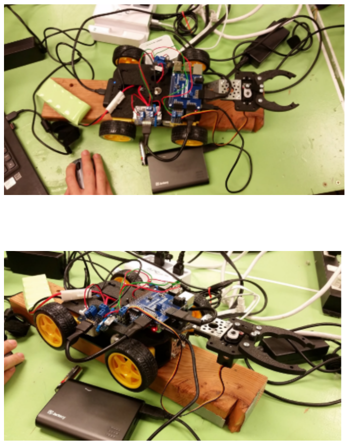

Fig. 6: Images of the claw robot.

IV. Aruco Markers in OpenCV using Python

Aruco markers provided a way to determine presence and orientation of identification (ID) markers. An image of Aruco markers used for the can and robot are shown in Fig. 7. In the top image, the OpenCV output of the angular vector from the robot to the can is shown. This vector was created in Python through the Aruco library. We followed the camera calibration setup described by the Aruco API [6]. After calibration, the Aruco markers could be printed and attached to objects of relevance. The center of each marker is calculated and then used along with the orientation of the marker to construct the vector shown in the bottom of Fig. 7.

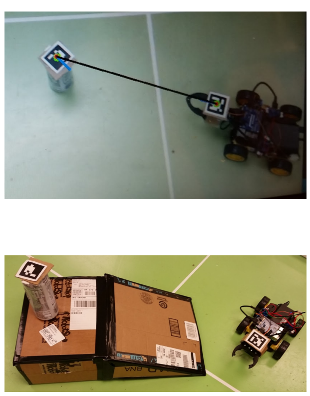

Fig. 7: Images of the angular vector shown in OpenCV (top) and the Aruco markers for the can and robot (bottom).

V. Design and Experiment

We attempt to re-create a variation of the original banana and monkey experiment by using the Donkey Car, 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. 9.

A. Project 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.

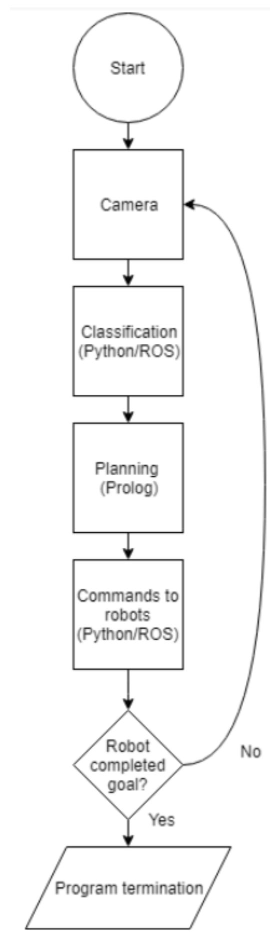

Fig. 8: High-level flowchart for the Python and Prolog system.

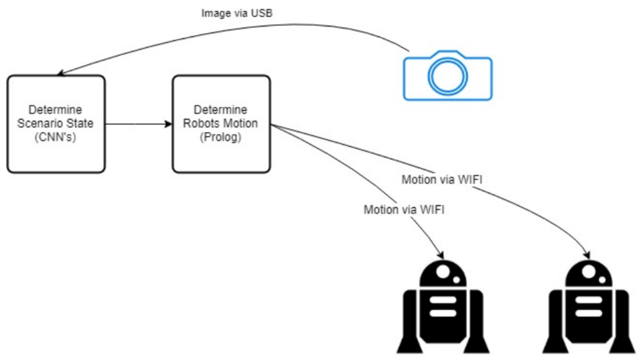

Fig. 9: Black-box for the Python and Prolog system.

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.

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

- Phase 1: getting the robot to specific locations on map using Python.
  - Robot Motion controlled using client-server setup for one robot.
  - Identify different objects using Aruco Markers.
  - Mapping Directions from robot to object using Python.
  - Writing the fully integrated program using Python.
- Phase 2: making the robot adjust objects using Prolog.
  - Identify orientation and (x, y) positions of objects.
  - Mapping directions to adjust objects to correct orientation.
  - Mapping directions from robot to object using Python.
  - Update master program to use new features.
- Phase 3: the robot adjusts and moves objects to reach goal locations on map.
  - Grabbing and moving objects.
  - Update master program algorithms for new features.

VI. Results and Conclusions

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.

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. Cambridge, 1985.
[5] R. A. Brooks, Intelligence without reason, 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].