View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Works

Pyro: A Python-based Versatile Programming Environment for Teaching Robotics

DOUGLAS BLANK
Bryn Mawr College

DEEPAK KUMAR
Bryn Mawr College

LISA MEEDEN
Swarthmore College

HOLLY YANCO
University of Massachusetts, Lowell

In this paper we describe a programming framework called Pyro which provides a set of abstractions that allows students to write platform-independent robot programs. This project is unique because of its focus on the pedagogical implications of teaching mobile robotics via a top-down approach. We describe the background of the project, novel abstractions created, its library of objects, and the many learning modules that have been created from which curricula for different types of courses can be drawn. Finally, we explore Pyro from the students' perspective in a case study.

Categories and Subject Descriptors: I.2 [**Artificial Intelligence**]: Robotics – *Autonomous vehicles*; K.3 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*; I.6 [**Simulation and Modeling**]: Simulation Support Systems - *Environments*

General Terms: Mobile robotics, education, robot abstractions, autonomous control, programming languages

Additional Key Words and Phrases: Computer science education, top-down instruction, platform-independent robotics control

1. INTRODUCTION

Not that long ago, robotics was a field of study relegated to well-funded engineering universities that built their own robots. Starting in the mid 1990's, simple, inexpensive robots (such as the Handyboard and LEGO Mindstorms) were introduced and their use proliferated in the classroom, first, in colleges and later even in middle schools. Currently, sophisticated robots with cameras, advanced sensors, and motors (such as Sony's robot dog, Aibo, and ActivMedia's Pioneer) are becoming financially accessible to undergraduate computer science, and even some courses in psychology.

Although sophisticated robotic platforms are now affordable, a large issue still remains: how do you teach students to use such robots? Unfortunately, each robot has its

own application programming interface (API), and, even worse, each specific type of sensor may have its own API. This situation is perhaps similar to the one in the early days of digital computers when every computer had a different architecture, a different assembly language, and even a different way of storing the most basic kinds of information.

The Pyro project was designed to answer the question of how to program sophisticated robots by serving as a high-level programming paradigm for a wide variety of robots and sensors. Pyro, which stands for Python Robotics, is a Python-based robotics programming environment that enables students and researchers to explore topics in robotics. Programming robot behaviors in Pyro is akin to programming in a high-level general-purpose programming language in that Pyro provides abstractions for low-level robot specific features much like the abstractions provided in high-level languages. Consequently, robot control programs written for a small robot (such as K-Team's hockey puck-sized, infrared-based Khepera robot) can be used, without any modifications, to control a much larger robot (such as ActivMedia's human-scale, laser-based PeopleBot). This represents an advance over previous robot programming methodologies in which robot programs were written for specific motor controllers, sensors, communications protocols and other low-level features.

Programming robot behaviors is carried out using the programming language, Python, which enables several additional pedagogical benefits. We have developed an extensive set of robot programming modules, modeling techniques, and learning materials that can be used in graduate and undergraduate curricula in a variety of ways. In the following sections we present an overview of the abstractions incorporated into Pyro that have made it possible to make robot programs portable across platforms. Next we explore several examples that illustrate the ease of use of Pyro in different modeling situations. Finally, we examine the role of Pyro in computer science curricula by presenting a detailed case study of its use in an artificial intelligence course.

2. OVERVIEW OF PYRO

The need for a project like Pyro grew out of our desire to teach mobile robotics in a coherent, abstract, robot-independent manner. For example, we wished to start with

simple "direct control" programs running on simple robots and incrementally take students on a tour of other control paradigms running on increasingly sophisticated robots. There are many freely available, feature-rich, real-world control systems which one can download, program and run. For example, Carnegie Mellon has made their Robot Navigation Toolkit (a.k.a. CARMEN) available as open source, and ActivMedia's ARIA is also freely available. However, all such programs that we encountered suffered from three separate problems.

Most existing robot control programs are designed to run on a single type of robot. At best, some of the robot control systems we found ran on a few types of robots, but even then the types had to be of similar size, shape, and abilities. Second, we wanted the control system to be something that could be studied, and changed, by the students. All of the existing systems we encountered were designed to be as efficient as possible, and were therefore filled with optimizations which obfuscated their overall design to the student. In addition, we were unable to find a system for which we could easily separate the "controller" from the rest of the system. For example, a control system based on occupancy grids might be intimately tied to a particular type of robot and laser scanner.

It should not be a surprise that existing systems suffered from these limitations because most of these projects were research explorations of a particular paradigm running on a particular robot. However, even if we could have found a series of programs to run on our robots, we would not have been able to incrementally make small changes to the controller to take us from one paradigm to another, nor would we have been able to mix parts of one paradigm with parts of another. However, there were two projects that did meet some of our requirements.

The first of these is TeamBots [Balch, 2004]. TeamBots is written in Java, and, therefore, is object-oriented with the possibility of many appropriate abstractions for teaching. TeamBots was designed such that the hardware interfaces and controllers are independent of one another. Thus, one can write a control program without worrying about low-level details of the particular robot that it is controlling [Balch, 1998]. At the time TeamBots was written (1997--2000) the idea of using Java to control mobile robotics was quite revolutionary. However, the TeamBots authors argued that "the benefits of Java (correctness, ease of use, rapid development) far outweigh the negligible runtime overhead" [Balch, 2004]. We very much agree with this philosophy. In fact, we

wanted to take the philosophy of "ease-of-use over runtime considerations" even further. Although Java is arguably easier to use than, say, C or C++, it still has a large conceptual overhead.

The other project that met some of our requirements was the Player/Stage project [Gerkey, 2003] which was first released in 2001. Player/Stage is actually three separate projects: Player, which is an evolving set of client/server protocols for communicating with robots and simulators; Stage, a "low-fidelity", 2-D multi-robot simulator; and Gazebo, a "high-fidelity", 3-D simulator. Because the Player/Stage authors have their software running on so many different kinds of robots, they have developed many useful robot and sensor abstractions. Whereas TeamBots only supported two different kinds of robots (Probotic's Cye and the now defunct Nomadic Technologies Nomad 150), Player/Stage supports literally dozens of robots: from K-Team's Khepera to Segway's RMP (a customized version of their Human Transport). However, all of Player/Stage is written in the C language and is designed to operate as efficiently as possible. Although such efficiency is required by many control systems, such optimized code often obscures the high-level design concepts. As mentioned, we were very willing to trade runtime efficiency for ease-of-use (and ease-of-understanding). Player/Stage was not designed to be used by novice programmers.

In the end, we decided to build our own control system. We created a prototype using the extensible modeling language XML in combination with C++ [Blank, 1999]. Basically, the code looked like HTML with C++ code between the tags. Although this system had some nice qualities derived from its XML roots, it turned out to have all the complexities of XML and C++ combined, and was therefore difficult for students to learn and debug. For example, even syntax errors could be hard to track down because there were two levels of parsing (one at the XML level, and another at the C++ level). In addition, like many of the other available control systems, we became bogged down in low-level interface issues and never reached the point of implementing more than one control paradigm.

Having learned from this prototype, we decided to try again, but this time the primary focus was on the usability from the student perspective. We found that the language Python met many of our goals. To our surprise, we also found that Python had recently been used for solving real-world complex programming problems. For example,

[Prechelt, 2000] found in some specific searching and string-processing tests that Python was better than Java in terms of run-time and memory consumption, and not much worse than C or C++ in some situations. In this incarnation, we set out with the following goals:

- be easy for beginning students to use
- provide a modern object-oriented programming paradigm
- run on several robot platforms and simulators
- allow the exact same program to control different kinds of robots
- allow exploration of many different robot control paradigms and methodologies
- *scale up conceptually* by remaining useful as users gain expertise
- be extendible
- allow for creation of modern-looking visualizations
- be available freely for study, use, and further development

Python is an ideal language for implementing these goals. In fact, Python itself is driven by similar ideals. For example, Python supports many different programming paradigms without making strong commitments to any. Although one can write complete Python programs without ever defining a function, one can also use functions as first-class objects. As the user grows in sophistication, so can the language. Figure 1 shows three paradigms for printing "Hello world!": a simple, direct method; a method using functions; and an object-oriented method. One can see that more complex paradigms build on syntax and concepts from simpler paradigms. Java, on the other hand, does not allow students to start with simple expressions and statements and slowly increase the level of concepts and syntax, but rather forces the user to embrace the full object-oriented methodology from square one.

```
print 'Hello world!'        => Hello world!

def helloWorld():
    print 'Hello world!'
helloWorld()                => Hello world!

class HelloWorld:
    def greet(self):
        print 'Hello world!'
obj = HelloWorld()
obj.greet()                 => Hello world!
```

Figure 1. Python scales up pedagogically. Three different paradigms for greeting the world: direct, functional, and object-oriented.

Pyro is composed of a set of Python classes that encapsulates lower-level details. Figure 2 provides a schematic of the Pyro architecture. Users write robot control programs using a single application programming interface (API). The API is implemented as an object-oriented hierarchy that provides an abstraction layer on top of all the vendor-supplied robot-specific API's. For example, in Figure 2, all the robot-specific API's have been abstracted into the class `pyro.robot`. In addition, other abstractions and services are available in the *Pyro Library*. The libraries help simplify robot-specific features and provide insulation from the lowest level details of the hardware or simulation environments.

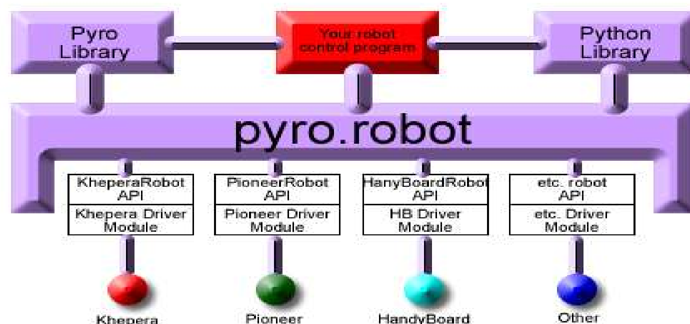

Figure 2. Pyro Architecture

Currently, Pyro supports K-Team's Kheperas, ActivMedia's Pioneer-class robots (including PeopleBot and AmigoBot robots), Player/Stage-based robots (including

Evolution's ER1 and many others), the Handyboard, RWI's Mobility-based B21R, and simulators for all of these. Currently, many other robots are also being ported to Pyro, including Sony's Aibo, K-Team's inexpensive Hemisson, and the Robocup Soccer Server Simulator.

A user's control program for a robot is called a *brain*. Each brain is written by extending the library classes similar to the way a Java programmer writes Java programs. This allows a robot programmer to concentrate mainly on the behavior-level details of the robot. Since the control program is written in Python, the standard *Python Library* is also available for use. Also, because the brain base class is also Python code, new control paradigms can be easily added and studied. Before we go any further, we would like to present a simple example of a robot control program.

2.1 A First Look

In this section we present a simple obstacle avoidance behavior to demonstrate the unified framework that Pyro provides for using the same control program across many different robot platforms. This type of simple controller is an example of "direct" (or "stateless") control. Direct control is normally the first control method introduced to students learning robotics. In this simple form of control, sensor values are used to directly affect motor outputs. The top five lines of Figure 3 show pseudocode that represents a very simple algorithm for avoiding obstacles.

The program shown in the lower portion of Figure 3 implements the pseudocode algorithm using the abstractions in the libraries. The program, written in an object-oriented style, creates a class called `Avoid` which inherits from a Pyro class called `Brain` (Figure 3, line 2). Every Pyro brain is expected to have a `step` method (line 3) that is executed on every control cycle which occur about 10 times a second. The brain shown will cause the robot to continually wander and avoid obstacles until the program is terminated.

It is not important to understand all the details of the Pyro implementation, but the reader should notice that the entire control program is independent of the kind of robot and the kind of range sensor being used. The program will avoid obstacles when they are within the `safeDistance` of 1 *robot unit* (discussed below) of the robot's front left or front right range sensors (lines 6 and 9, respectively), regardless of the kind of robot.

Lines 14 and 15 show the details of Pyro's automatic initialization mechanism. Such lines will be left out in subsequent examples.

```
#   if approaching an obstacle on the left side
#      turn right
#   else if approaching an obstacle on the right side
#      turn left
#   else go forward

 1  from pyro.brain import Brain
 2  class Avoid(Brain):
 3      def step(self):
 4          safeDistance = 1 # in Robot Units
 5          #if approaching an obstacle on the left side, turn right
 6          if min(self.get('robot/range/front-left/value')) < safeDistance:
 7              self.robot.move(0,-0.3)
 8          #else if approaching an obstacle on the right side, turn left
 9          elif min(self.get('robot/range/front-right/value')) < safeDistance:
10              self.robot.move(0,0.3)
11          #else go forward
12          else:
13              robot.move(0.5, 0)
14  def INIT(engine):
15     return Avoid('Avoid', engine)
```

Figure 3. An obstacle avoidance program, in pseudocode and in Pyro

3. A DESIGN AND DEVELOPMENT PERSPECTIVE

Most of the Pyro framework is written in Python. As one can see, Python is an easy-to-read scripting language that looks very similar to pseudocode. It also integrates easily with C and C++ code which makes it possible to quickly incorporate existing code. The C/C++ interface also facilitates the inclusion of very expensive routines (like vision programs) at lower levels for faster runtime efficiency. Also, we are able to "wrap" programs written in C and C++ (such as Player/Stage) so that they are instantly, and natively, available in Python.

One of the key ideas underlying the design of Pyro is the use of abstractions that make the writing of basic robot behaviors independent of the type, size, weight, and shape of a robot. Consider writing a robot controller for obstacle avoidance that would work on a 24-inch diameter, 50-pound Pioneer3 robot as well as on a 2.5-inch diameter, 3-ounce Khepera. The following key abstractions were essential in achieving our design goals:

1. **Range Sensors:** Regardless of the kind of hardware used, IR, sonar, or laser, these sensors are categorized as *range* sensors. Sensors that provide range information can thus be abstracted and used in a control program.

2. **Robot Units:** Distance information provided by range sensors varies depending on the kind of sensors used. Some sensors provide specific range information, like distance to an obstacle in meters or millimeters. Others simply provide a numeric value where larger values correspond to open space and smaller values imply nearby obstacles. In our abstractions, in addition to the default units provided by the sensors, we have introduced a new measure, *a robot unit*: 1 robot unit is equivalent to the diameter of the robot being controlled.
3. **Sensor Groups:** Robot morphologies (shapes) vary from robot to robot. This also affects the way sensors, especially range sensors, are placed on a robot's body. Additionally, the number and positions of sensors present also varies from platform to platform. For example, a Pioneer3 has 16 sonar range sensors while a Khepera has 8 IR range sensors. In order to relieve a programmer from the burden of keeping track of the number and positions of sensors (and their unique numbering scheme), we have created *sensor groups*: *front*, *left*, *front-left*, etc. Thus, a programmer can simply query a robot to report its front-left sensors in robot units. The values reported will work effectively on any robot, of any size, with any kind of range sensor given appropriate coverage, yet will be scaled to the specific robot being used.
4. **Motion Control:** Regardless of the kind of drive mechanism available on a robot, from a programmer's perspective, a robot should be able to move forward, backward, turn, and/or perform a combination of these motions (like moving forward while turning left). We have created two motion control abstractions: **move (translate, rotate)** and **motors(leftpower, rightpower)**. The former abstracts movements in terms of turning and forward/backward changes. The later abstracts movements in terms of applying power to the left and right sides. This is designed to work even when a robot has a different wheel organization (such as multi-wheel, omni-directional abilities) or four legs (as with Aibo). As in the case of range sensor abstractions, the values given to these commands are independent of the specific values expected by the actual motor drivers. A programmer only specifies values in a range -1.0..1.0 (see examples below).

5. **Devices:** The abstractions presented above provide a basic, yet important functionality. We recognize that there can be several other devices that can be present on a robot: a gripper, a camera, etc. We have created a *device* abstraction to accommodate any new hardware or ad hoc programs that may be used in robot control. For example, a camera can be controlled by a device that enables access to the features of the camera. Further, students can explore vision processing by dynamically and interactively applying *filters*. Filters are modular image-processing functions that can be sequentially applied to camera images. All devices are accessed using the same uniform interface metaphor.

The above abstractions are similar to the abstractions one takes for granted in a high-level programming language: data types, I/O, etc. These abstractions help simplify individual robots into higher-level entities that are needed in order for generic behavior programming.

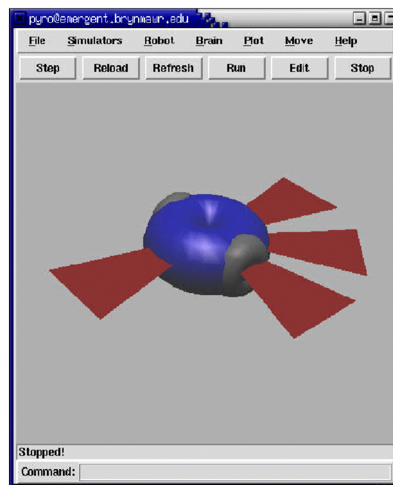

Figure 4. Dynamic 3-D visualization of a Khepera and its infrared sensors.

Pyro also provides facilities for the visualization of various aspects of a robot experiment. Users can easily extend the visualization facilities by providing additional Python code as needed in a particular experiment. For example, you can easily create a graph to plot some aspect of a brain, or sensor, with just a few lines of code. In addition,

Pyro can, through Python's OpenGL interface, generate real-time 3D views. Figure 4 shows a visualization of a Khepera robot and its infrared readings. In keeping with the spirit of the Pyro project, we created an abstract API so that 3D shapes can be drawn in this window without knowing anything about OpenGL.

The Python language has generated much interest in recent years as a vehicle for teaching introductory programming, object-oriented programming, and other topics in computer science. For example, Peter Norvig has recently begun porting the example code from Russell and Norvig's "Artificial Intelligence: A Modern Approach" [1995] into Python. This will no doubt bolster Python's use in AI. Because Pyro is implemented in Python, everything that applies to Python also applies to Pyro, both good and bad. Python appears to be a language that inexperienced undergraduates can pick up quickly. The language is object-oriented without any limitations on multiple-inheritance, and most objects are first-class. However, because Python is interpreted, it is generally considered a "scripting language". Is Python fast enough to use in a real-time robotics environment? We have tested Pyro's speed when executing different types of brains while varying the graphical outputs. Table 1 shows the resulting data from running Pyro under various loads on a Dual Pentium III 800 MHz PC. Experiments have shown that for doing very simple control, even with the OpenGL graphics enabled, the software was quite capable. In fact, most modern medium-cost robotics equipment can only handle about 10 updates per second, well within Pyro's typical performance. However, Python, and therefore Pyro, doesn't fair as well with more complex brains. Trying a complex brain with visual processing, and OpenGL graphics slows the system down to less than one update per second. Python does allow the migration of code into C. We expect further improvements in the future, and expect Moore's Law to help.

Pyro program and graphics	***Updates/second***
Bare brain with console	> 10000
Bare brain with OpenGL	> 1000
ANN with OpenGL	> 200
Fuzzy logic with OpenGL	> 20
Many ANNs + Vision + OpenGL	< 1

Table 1. Timing data from running Pyro on a Dual Pentium III 800 Mhz Linux PC. OpenGL rendering was done in hardware on the graphics card. ANN stands for Artificial Neural Network.

4. EDUCATIONAL RESOURCE EXAMPLES

In addition to the abstractions and device definitions, the Pyro Library includes several modules that enable the exploration of robot control paradigms, robot learning, robot vision, localization and mapping, and multi-agent robotics. Within robot control paradigms there are several modules: direct/reactive/stateless control, behavior-based control, finite state machines, subsumption architectures, and fuzzy logic. The learning modules provide an extensive coverage of various kinds of artificial neural networks (ANNs): feedforward networks, recurrent networks, self-organizing maps, etc. Additionally we also have modules for evolutionary systems, including genetic algorithms, and genetic programming. The vision modules provide a library of the most commonly used filters and vision algorithms enabling students to concentrate on the uses of vision in robot control. The entire library is open source, well documented, and can be used by students to learn about the implementations of all the modules themselves. We have also provided tutorial level educational materials for all of the modules. This enables instructors to tailor the use of Pyro for many different curricular situations. As the project moves beyond the initial production phase, we expect to add many more modules. With increased use in the community we also expect contributed modules to be added to the library. In the remainder of this section, we provide a few more examples of robot control written using the available libraries. All of the examples presented are actual working Pyro programs.

```
from pyro.brain import Brain
from random import random
class Wander(Brain):
   def step(self):
      safeDistance = 0.85 # in Robot Units
      l = min(self.get('robot/range/front-left/value'))
      r = min(self.get('robot/range/front-right/value'))
      f = min(self.get('robot/range/front/value'))
      if (f < safeDistance):
         if (random() < 0.5):
            self.robot.move(0, - random())
         else:
            self.robot.move(0, random())
      elif (l < safeDistance):
         self.robot.move(0,-random())
      elif (r < safeDistance):
         self.robot.move(0, random())
      else: # nothing blocked, go straight
         self.robot.move(0.2, 0)
```

Figure 5. A wander program

As mentioned, we have designed the highest level robot class to make abstractions such that programs, when written appropriately, can run unchanged on a variety of platforms. For example, consider the Pyro code in Figure 5. This short program defines a brain called *Wander* that enables a robot to move about without bumping into objects. The program runs on the suitcase-sized Pioneer and, without any modifications, on the hockey puck-sized Khepera. As mentioned, there are two mechanisms that allow this portability. First, all units returned from any range sensors are, by default, given in robot units. For example, 1 Khepera unit is equal to about 2.5 inches, while 1 Pioneer unit is equal to about 2 feet. Secondly, we try to avoid referring to specific kinds or positions of sensors. For example, in the above example, we refer to the default range sensor values by names such as `front-left`. On the Pioneer this could be measured by three sonar sensors, while on the Khepera it could be measured by a single infrared sensor. Although these mechanisms have their limitations, many robotics problems can be handled in this manner.

```
from pyro.brain import Brain
from pyro.brain.conx import Network
class NNBrain(Brain):
   def setup(self):
      self.net = Network()
      self.net.addThreeLayers(self.get('robot/range/count'), 2, 2)
      self.maxvalue = self.get('robot/range/maxvalue')
   def scale(self, val):
      return (val / self.maxvalue)
   def teacher(self):
      safeDistance = 1.0
      if min(self.get('robot/range/front/value')) < safeDistance:
         trans = 0.0
      elif min(self.get('robot/range/back/value')) < safeDistance:
         trans = 1.0
      else:
         trans = 1.0
      if min(self.get('robot/range/left/value')) < safeDistance:
         rotate = 0.0
      elif min(self.get('robot/range/right/value')) < safeDistance:
         rotate = 1.0
      else:
         rotate = 0.5
      return trans, rotate
   def step(self):
      ins = map(self.scale, self.get('robot/range/all/value'))
      targets = self.teacher()
      self.net.step(input = ins, output = targets)
      trans = (self.net['output'].activation[0] - .5) * 2.0
      rotate = (self.net['output'].activation[1] - .5) * 2.0
      robot.move(trans, rotate)
```

Figure 6. A neural network controller

Contrast the wander program with the program in Figure 6 which trains an artificial neural network to avoid obstacles. Again, the code is quite short (about 30 lines) but includes everything necessary to explore an example of on-line ANN learning on a robot. The goal of this brain is to teach a neural network to go forward when it is not close to any obstacles, but to stop and turn away from obstacles that are within one robot unit. The network takes the current range values as inputs and produces translate and rotate movements as outputs. With learning turned on, the network learns to do what the teacher function tells it to do. Turn off learning after training and the network should approximate (and generalize) the teacher's rules.

Every Pyro brain may include the optional `setup` method for initialization; it is only called once when the brain is instantiated. In the `NNBrain`, the `setup` method is used to create an instance of the `Network` class which is a three-layer feedforward network where the size of the input layer is equal to the number of range sensors on the current robot being controlled. Each time the brain's `step` method is called, the robot's range sensors are checked and target values for translate and rotate are determined. Then the

range values are normalized using the `scale` method to prepare them as inputs for the neural network. Next, the network's `step` method is called to propagate the given inputs through the network, back-propagate the error based on the given targets, and update the weights. Finally the network's outputs are used to move the robot. In this way the robot is learning online to perform obstacle avoidance.

```
from pyro.geometry import distance
from pyro.brain.behaviors.fsm import State, FSMBrain
class edge(State):
    def onActivate(self):
        self.startX = self.get('robot/x')
        self.startY = self.get('robot/y')
    def update(self):
        x = self.get('robot/x')
        y = self.get('robot/y')
        dist = distance( self.startX, self.startY, x, y)
        if dist > 1.0:
            self.goto('turn')
        else:
            self.robot.move(.3, 0)
class turn(State):
    def onActivate(self):
        self.th = self.get('robot/th')
    def update(self):
        th = self.get('robot/th')
        if angleAdd(th, - self.th) > 90:
            self.goto('edge')
        else:
            self.robot.move(0, .2)
def INIT(engine):
    brain = FSMBrain(engine)
    brain.add(edge(1)) # 1 means initially active
    brain.add(turn())
    return brain
```

Figure 7. A finite state machine controller

The next example, shown in Figure 7, uses a finite state machine (FSM) to control a robot. A FSM brain is assumed to consist of a set of states. Each state has an `onActivate` method that is called when the state becomes active and an `update` method that is called on each brain step. A state can relinquish control by using the `goto` method to activate a new state. The programmer's job is to define an appropriate set of states to solve a given problem.

In this example, the goal is to control the robot so that it continually moves in a square. In this case, two states have been defined: one to control the robot while it traverses the edge of the square and a second to control the robot while it turns. Initially the edge state is activated. In the edge state, the starting position of the robot is saved and

compared to the current position. Once the robot has traveled the length of one robot unit, the edge state activates the turn state. In the turn state, the starting heading of the robot is saved and compared to the current heading. Once the robot has turned ninety degrees, the turn state re-activates the edge state. By repeating this sequence of states, the robot will travel in the desired square motion.

This section has provided a sample of the variety of robot control programs that can be explored from within Pyro. The next section outlines ways Pyro can be incorporated into existing courses in the curriculum.

5. IN THE CURRICULUM

Because Pyro allows students to immediately focus on the most abstract, top-down issues in autonomous control, we have been able to incorporate Pyro into a variety of courses. Many of these courses have been taught to students with little to no background in programming. Pyro has been incorporated in the undergraduate curriculum at Bryn Mawr College, Swarthmore College, and the University of Massachusetts Lowell (UML). Additionally, it has been used at at least ten other institutions. At UML, it has also been used in the graduate level courses. Specifically, Pyro has been incorporated into the following courses:

1. **Introduction to Artificial Intelligence**: A standard elective course in the computer science curriculum. This course is offered at Swarthmore College and Bryn Mawr College.
2. **Cognitive Science**: An elective in computer science and psychology. This course is offered at Bryn Mawr College.
3. **Emergence**: An elective course that studies emergent computation and emergent phenomena. Additional Python code has been developed to explore related topics, such as bird flocking behavior, and cellular automata. This course is offered at Bryn Mawr College.
4. **Androids: Design & Practice**: An upper-level elective on recent advances in robotics. This course is offered at Bryn Mawr College.
5. **Developmental Robotics**: Another upper-level elective on recent advances in robotics. This course is offered at Bryn Mawr College and Swarthmore College.

6. **Robotics II**: This is a second undergraduate course in Robotics at the University of Massachusetts Lowell (UML).
7. **Mobile Robotics**: This is a graduate-level course offered at UML.
8. **Senior Theses**: Students at several institutions have used Pyro as a part of their capstone projects.
9. **Summer Research**: Students at several institutions have used Pyro as a part of their summer research projects at the undergraduate and graduate levels. Additionally, some high school students have also used Pyro in their summer research projects.

It is clear that wherever in the curriculum robotics is used, Pyro can be used as a laboratory environment. In all of the above instances, students wrote several robot control programs for real and simulated robots. In most of these cases, students learned Pyro and robot programming by following the tutorial materials we have created. However, the kinds of exercises varied depending on the course and its focus.

The student projects from these courses span a large range of complexity. For example, in the cognitive science course, many students had never written a program before. However, they were easily able to take simple reactive brains, such as those shown above, tweak them, and ask observers their impression of the robot's behavior. On the other hand, advanced computer science students in the Developmental Robotics courses were able to perform research-level projects rather quickly. For example, students were able to write Pyro programs to co-evolve predator-prey controllers in a matter of days. Other examples included:

1. **Laser tag**: A group of students designed hardware to send and receive infrared signals, then wrote software to make the game-playing robots locate and target each other.
2. **Adding sensors to a research platform**: A student designed a circuit board to allow additional sensors to be added to a research robot using a serial port on the robot. The student also wrote control code for the robot. This project was used in the laser tag project above.
3. **Robot Slalom**: Several teams of students designed programs that used computer vision to find gates in a slalom course that ran down a hallway (and around corners).

4. **Pick Up the Trash**: Several teams of students designed programs using computer vision to find trash (styrofoam cups) and recycling (soda cans) and deliver the found items to the appropriate bins (trash can or recycling bin).
5. **Robot Tour Guide**: A group of undergraduate students created a robot that gives tours of the Park Science Building at Bryn Mawr College.

The last project listed above was done by three undergraduate students and was partially funded by a grant to them from the Computing Research Association's CREW program. In each of the above instances, we have carried out extensive evaluations on the impact of using Pyro in each course. Next, we present one such case study.

6. CASE STUDY: AN ARTIFICIAL INTELLIGENCE COURSE

One way to evaluate whether Pyro is successful at achieving its goal of providing undergraduates with an effective tool for exploring advanced robotics is to consider how well it can be integrated into an AI course. In this section we will look in detail at a particular AI course, and examine the level of sophistication of the robotics projects attempted by the students. We will also summarize student comments on using Python and Pyro.

At Swarthmore College, the AI course is typically taught every other year and is intended for Computer Science majors who have already taken a CS1 course in an imperative language and a CS2 course in an object-oriented language as prerequisites. The AI course was updated in Spring 2004 to incorporate Python and Pyro into every lab and project [Meeden, 2004]. This particular offering of the course had a machine learning focus covering game playing, neural networks, genetic algorithms, decision trees, reinforcement learning, and robotics. Students met twice a week for lecture and discussion and once a week for lab.

The labs were designed to introduce the students to the Python programming language, the tools available within Pyro, and the machine learning topics being covered in class. Most labs were relatively short in duration, typically lasting only a week. The projects were designed to allow the students to explore a machine learning topic in much more depth and lasted two to three weeks. For the first two projects, the students were

given at least one default option that they could implement, but were also encouraged to develop their own ideas into a project. For the final project, the students were expected to generate their own project proposal. All student-generated proposals required pre-approval by the instructor to ensure that they were feasible. Each project culminated in a four to six page paper describing the machine learning problem, the experimental design, and the results. The class included six labs and three projects. For the larger labs and projects, the students were allowed to work in teams of two or three.

The first project involved applying a neural network to a problem. The default option was to use a database of facial images, described by Mitchell in Chapter 4 of his machine learning textbook [Mitchell, 1997], to learn features such as pose or expression. The students used the Conx library (which is part of Pyro) for doing their neural network projects. Conx includes an implementation of back-propagation learning, allowing the students to focus on the data representation and training procedure.

The second project involved applying a genetic algorithm to a problem. The default option was to try to find solutions to the traveling salesman problem for particular countries in the world or to attempt one of the contests sponsored by the *Congress on Evolutionary Computation* which included growing virtual plants, predicting binary series, and creating art. The students used the Genetic Algorithm library (which is part of Pyro), allowing them to again focus on the representation of the problem, as well as creating a good fitness function. One of the interesting aspects of the traveling salesman problem is that many researchers have proposed special-purpose genetic operators to more quickly converge on good solutions. The students were asked to implement a new crossover and a new mutation operator from the literature.

The final project involved robot learning on a simulated Pioneer-style robot with sonar sensors, blob vision, and a gripper. For the final project, the majority of the students in the class chose a task in which the robot would be controlled by a neural network and the weights of the network would be evolved by a genetic algorithm. This combined all the tools that they had used in the previous two projects. In order to implement this learning method, the students had to do the following:

1. Design a learning environment and task for the robot.
2. Subclass Pyro's `Brain` class to create a neural network brain based on Pyro's `Network` class with task appropriate input values derived from sensors and output values to command the motors.
3. Subclass Pyro's `GA` class to create a task appropriate fitness function and stopping criteria for the evolutionary process. Include commands to save the best neural network weights found so far.
4. Create a testing program to instantiate a neural network from a file of saved weights and then evaluate the evolved behavior.

The most ambitious robot learning project from the class involved a three way game of tag in which each robot had a unique color: the red robot was chasing the blue robot, the blue robot was chasing the green robot, and the green robot was chasing the red robot. The neural network brain for each robot had the same structure, but the weights were evolved in a separate species of the genetic algorithm. The reason for this was to allow each robot to develop unique strategies.

Other robot learning projects from the class included having a robot gather colored pucks scattered randomly throughout the environment, having a robot navigate a PacMan-inspired maze while avoiding a predator robot, and having a robot trying to capture a puck from a protector robot.

Pyro's infrastructure allowed the students to focus on the most interesting aspects of the project, such as the environment, task, network architecture, and fitness function, without having to worry about the details of how the genetic algorithm and neural network were implemented. The abstractions provided within Pyro enabled the students to easily integrate a neural network with a genetic algorithm and thus to develop quite sophisticated robot learning projects in only three weeks time.

Although eighty percent of the students in the class had not used Python before and there was very little formal instruction given in the class on Python, students were enthusiastic about the language:

> "I think that I know enough languages (of a wide variety) that I was able to adapt to Python even with minimal instruction."
>
> "I really liked Python; it's a clean and easy yet powerful language."
>
> "I like the language. I think it is intuitive and easy to learn, and is appropriate for this course."
>
> "I like Python very much because it looks like executable pseudo code."
>
> "I think Python is great to use in higher-level CS classes like this because it allows for coding relatively complex programs quickly, compared to say C, and I always find I have more time for extra experimentation when using Python."

As demonstrated in the following comments, students also appreciated the abstractions provided by Pyro and liked having access to the source code:

> "Pyro took care of a lot of the repetitive, less interesting coding for us."
>
> "Pyro had good capabilities for programming real robots and implemented a lot of learning techniques that are useful in AI. Also it was nice being able to program generically for any robot."
>
> "I liked that the details were hidden, but I could go in and change things if needed."
>
> "I found that having the source available was very helpful on a number of occasions, especially since it's in Python and can be understood quickly."
>
> "Accessible source---I modified a lot of components for my experiments. The code was reasonably clean and straightforward to work with."

As is evident from above, Pyro enables students at all levels to do robotics projects that were only feasible in the past by research teams. This, we believe, is one of the biggest pay-offs of Pyro. It brings aspects of current research into the curriculum in an accessible, low cost manner.

7. CONCLUSIONS

The Pyro project is the latest incarnation of our attempts to make the teaching of autonomous mobile robots accessible to students and teachers alike. We have developed a variety of programs, examples, and tutorials for exploring robotics in a top-down

fashion, and we are continuing to add new curricular modules. Some of these modules are created by students in the classes, others by the authors, and some by faculty at other institutions who have adopted Pyro. Modules currently under development include multi-agent communication, reinforcement learning, logic, planning, and localization.

We believe that the current state-of-the-art in robot programming is analogous to the era of early digital computers when each manufacturer supported different architectures and programming languages. Regardless of whether a computer is connected to an ink-jet printer or a laser printer, a computer today is capable of printing on any printer device because device drivers are integrated into the system. Similarly, we ought to strive for integrated devices on robots. Our attempts at discovering useful abstractions are a first and promising step in this direction. We believe that discoveries of generic robot abstractions will, in the long run, lead to a much more widespread use of robots in education and will provide access to robots to an even wider range of students.

Our goal is to reduce the cost of learning to program robots by creating uniform conceptualizations that are independent of specific robot platforms and incorporate them into an already familiar programming paradigm. Conceptualizing uniform robot capabilities presents the biggest challenge: How can the same conceptualization apply to different robots with different capabilities and different programming API's? Our approach, which has been successful to date, has been shown to work on several robot platforms, from the most-expensive research-oriented robot, to the lowest-cost LEGO-based ones. We are striving for the "write-once/run-anywhere" idea: robot programs, once written, can be used to drive vastly different robots without making any changes in the code. This approach leads the students to concentrate more on the modeling of robot "brains" by allowing them to ignore the intricacies of specific robot hardware. More importantly, we hope that this will allow students to gradually move to more and more sophisticated sensors and controllers. In our experience, this more generalized framework has resulted in a better integration of robot-based laboratory exercises in the AI curriculum. In addition, our system is not only accessible to beginners, but is also usable as a research environment for our own robot-based modeling.

ACKNOWLEDGMENTS

Pyro source code, documentation and tutorials are available at `www.PyroRobotics.org`. This work is funded in part by NSF CCLI Grant DUE 0231363.

REFERENCES

BALCH, T. 2004. TeamBots website. The URL is www.teambots.org.

BALCH, T. 1998. *Behavioral diversity in learning robot teams*. Ph.D. thesis, Georgia Institute of Technology.

BLANK, D. S., HUDSON, J. H., MASHBURN, B. C., AND ROBERTS, E. A. 1999. *The XRCL Project: The University of Arkansas' Entry into the AAAI 1999 Mobile Robot Competition*. Tech. rep., University of Arkansas.

GERKEY, B., VAUGHAN, R., AND HOWARD, A. 2003. The Player/Stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics.* Coimbra, Portugal, 317-323.

MEEDEN, L. 2004. CS63 Artificial Intelligence, Spring 2004, Swarthmore College. The URL is http://www.cs.swarthmore.edu/meeden/cs63 /s04/cs63.html.

MITCHELL, T. M. 1997. *Machine Learning*. McGraw-Hill, Boston, MA.

MONDADA, R., FRANZI, E., AND IENNE, P. 1993. Mobile robot miniaturization: A tool for investigation in control algorithms. In *Proceedings of the Third International Symposium on Experimental Robots*. Kyoto, Japan.

MONTEMERLO, M., ROY, N., AND THRUN, S. *CARMEN: Carnegie Mellon Robot Navigation Toolkit.* The URL for CARMEN is http://www-2.cs.cmu.edu/carmen/.

PRECHELT, L. 2000. *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program*. Tech. rep., Universitat Karlsruhe, Fakultat fur Informatik, Germany.

RUSSELL, S. AND NORVIG, P. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ.