Python Robotics: An Environment for Exploring Robotics Beyond LEGOs

Douglas Blank
Computer Science
Bryn Mawr College
Bryn Mawr, PA 19010
dblank@cs.brynmawr.edu

Lisa Meeden
Computer Science
Swarthmore College
Swarthmore, PA 19081
meeden@cs.swarthmore.edu

Deepak Kumar
Computer Science
Bryn Mawr College
Bryn Mawr, PA 19010
dkumar@cs.brynmawr.edu

Abstract

This paper describes Pyro, a robotics programming environment designed to allow inexperienced undergraduates to explore topics in advanced robotics. Pyro, which stands for Python Robotics, runs on a number of advanced robotics platforms. In addition, programs in Pyro can abstract away low-level details such that individual programs can work unchanged across very different robotics hardware. Results of using Pyro in an undergraduate course are discussed.

Categories & Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education - *Computer Science Education.*

General Terms

Design, Human Factors, Languages

Keywords:

Pedagogy, Robotics, Python

1 Introduction

The use of robots in the undergraduate curriculum has grown tremendously in the last few years [9, 11, 7, 2, 5, 12, 4, 6]. The availability of low cost, easy-to-use products (such as the LEGO Mindstorms, and Fred Martin's Handyboard [8]) has even led to a wide use of robots in middle and high school curricula. Although this equipment has been of enormous help in the introduction of robotics to new students, many of the topics addressed must be necessarily limited due to the simplistic nature of the hardware. For example, more sophisticated artificial intelligence and robotics topics such as vision, mapping, and planning cannot be fully addressed.

There are now many, medium-cost advanced robotics platforms on the market, for example Probotics' Cye, ActivMedia's Pioneer2 and AmigoBOT, and K-Team's Khepera to mention just a few. These robots often allow the optional use of cameras, sonar, and even laser rangefinders. Unfortunately, these more advanced robot platforms cater mostly to research-oriented users and are often inaccessible to undergraduates. In addition, there is not a unifying interface between these robots: each one comes with its own (often proprietary) development tools and each is substantially different from the others (for example, implemented in Java, C++, some other scripting language). If one did invest in learning to use one robot platform, probably none of the code, and possibly little of the knowledge, would transfer to a different platform.

In this paper, we describe a project that addresses the above situation. We are creating a set of tools that make up the next generation teaching and research-level robot laboratory. In developing these tools, we want to ensure that research-level robotics hardware and methodologies are accessible to computer science faculty who may not have robotics experience or whose robotics experience was limited to Handyboard-type, LEGO-based robots. The resulting system, called Pyro, was designed with the following goals: the system should be easy for beginning students to use, provide a modern object-oriented programming paradigm, run on several platforms, allow exploration of many different robot control paradigms and methodologies, remain useful as users gain expertise, be extendable, allow for the creation

SIGCSE '03, February 19-23, Reno Nevada, USA.

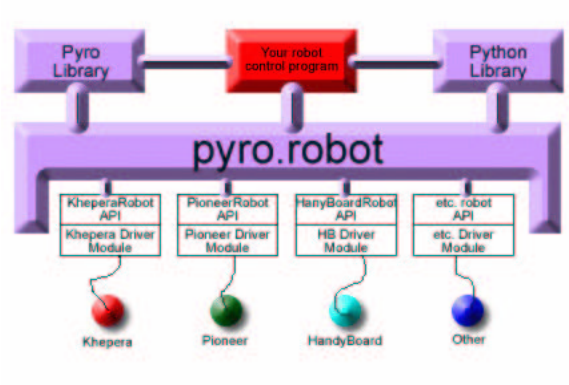

Figure 1: Pyro Architecture

of modern-looking visualizations, and be distributed as open source.

In what follows, we first present an overview of the system architecture, followed by a description of the modeling methodologies currently incorporated, programming examples, and how it has been used in our curriculum.

2 Pyro: Python Robotics

Pyro stands for Python Robotics. As mentioned, one of the goals of the Pyro project is to provide a programming environment that can be used for experimenting with various types of intelligent controllers on several robot platforms and simulators. Currently, the robots supported include the Pioneer family (Pioneer2, AmigoBOTs, etc.) and the Khepera family (Khepera and Khepera 2 robots). Additionally, there are simulators available for both of these types of robots that Pyro can connect onto, and control, as well.

Although it is important to be able to control very different kinds of robots from a single application programming interface (API), a more important goal was that individual programs should be able to control very different kinds of robots. That is, a single program should run on a 75 pound Pioneer2AT with, for example, laser and sonar sensors, and that same program should also run *unchanged* on a 2 inch tall Khepera with infrared sensors. By developing the right level and types of abstractions, Pyro largely succeeds in this goal. Examples will be discussed below.

Pyro also has the ability to define different styles of controllers. For example, the control system could be an artificial neural network (ANN), a subsumption architecture, a collection of fuzzy logic behaviors, or a symbolic planner. Any such program that controls the robot (physical or simulated) we refer to as a *brain*. Each brain is written in Python and usually involves extending existing class libraries (see Figure 1). The libraries help simplify robot-specific features and provide insulation from the lowest level details of the hardware. In fact, the abstraction provided uniformly accommodates the use of actual physical robots or their simulations even though vastly different sensors, drivers, motors, and communication protocols may be used underneath the abstraction layer. Consequently, a robot experimenter can concentrate on the behavior-level details of the robot.

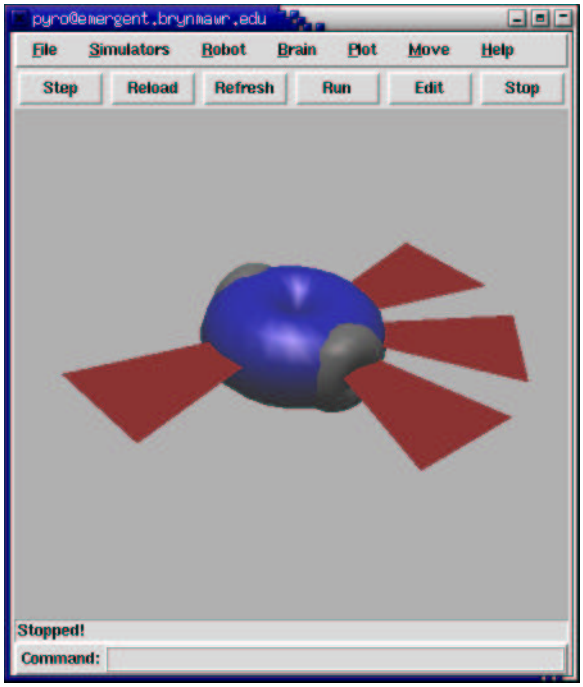

Figure 2: Dynamic 3-D visualization of a Khepera and its infrared sensors.

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 2 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.[1] Because Pyro is imple-

[1]Peter Norvig has recently been porting the example code from Russel and Norvig's " Artificial Intelligence: A Modern

mented 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 quite 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" and wasn't our first choice as a language in which to write advanced robotics programs.

Before developing Pyro, we first examined existing projects to see if any fit our constraints. There are many open sourced robotics programming environments available; however, most are committed to a particular control strategy. Separating the control strategy code from the rest of the system code seemed to require a major rewrite in all cases that we examined. However, TeamBots [1] is one open source project that satisfied many of our goals. TeamBots is written in Java, and, therefore, is object-oriented with many appropriate abstractions. However, because security is of such importance in Java, there are some additional burdens placed on the programmer at all levels of programming. For example, multiple inheritance must be implemented through single inheritance combined with interfaces. Although such limitations can be overcome in an introductory programming course, we did not want to have to address them in our introductory robotics courses.

We decided to build a prototype using the extensible modeling language XML in combination with C++ [3]. 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 introductory 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).

Having learned from the prototype, we decided to try again, but this time the focus was on the usability from the perspective of a new user. We found that the language Python meets 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, [10] 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++.

However, the question remained: Would Python be fast enough to use in a real-time robotics environment? Unfortunately, the only way to answer this question would be to build a system and try it. Now that Pyro exists, we have tested its speed performing with different types of brains, with different graphical outputs. Table 1 shows the resulting data. 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.

Pyro program and graphics	Updates/second
Bare brain with console	+10,000
Bare brain with OpenGL	+1,000
ANN with OpenGL	+200
Fuzzy logic with OpenGL	+20
Many ANNs + Vision + OpenGL	less than 1

Table 1: Timing data from running Pyro on a Dual Pentium 800 MHz Linux PC. OpenGL rendering was done in hardware on the graphics card.

However, Python, and therefore Pyro, doesn't fair as well with more complex brains. Trying a complex brain with visual processing, and OpenGL graphics slow the system down to less than one update per second. However, Python does allow the migration of code into C. We expect further improvements in the future, and expect Moore's Law to help.

2.1 Pyro Components

At this time, we have been working in Python for approximately a year. In that time, we have built the following components: ANN Back-propagation of error module; Self-organizing map module; Fuzzy logic, behavior-based brain module; Visual image processing library; OpenGL interface and renderer; High-level, abstract robot class; Generic brain class; Graphing module; Generic simulator.

Each of these modules is written in Python. As such, the modules and libraries can be used stand-alone, and interactively at the Python prompt.

2.2 Pyro Examples

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 follow 20 lines of Pyro code:

```
from pyro.brain import Brain
from time import *
from random import random, seed
class Wander(Brain):
   def step(self):
      safeDistance = 0.85 # in Robot Units
      l = self.getRobot().getSensorGroup('min', 'front-left')[1]
      r = self.getRobot().getSensorGroup('min', 'front-right')[1]
      f = self.getRobot().getSensorGroup('min', 'front')[1]
      if (f < safeDistance):
         if (random() < 0.5):
```

Approach." That will no doubt bolster Python's use in AI.

```
            self.getRobot().move(0, - random())
         else:
            self.getRobot().move(0, random())
      elif (l < safeDistance):
         self.getRobot().move(0,-random())
      elif (r < safeDistance):
         self.getRobot().move(0, random())
      else: # nothing blocked, go straight
         self.getRobot().move(0.2, 0)
```

This little program defines a brain called *Wander* that does just that. The program does indeed run on the suitcase-sized Pioneer2, and the hockey puck-sized Khepera. There are two mechanisms that allow this portability. First, all units returned from range sensors are given in "robot units." That is, the units are converted into values that are meaningful at the scale of each robot. For example, 1 Khepera unit is equal to about 60 mm, while 1 Pioneer unit is equal to about 2 feet. Secondly, we avoid referring to specific kinds or positions of sensors. For example, in the above example, we refer to the default range sensor 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, much of current advanced robotics problems can be handled in this manner.

Contrast the previous example with the following program that trains an artificial neural network to avoid obstacles:

```
from pyro.brain import Brain
from pyro.brain.conx import *
class NNBrain(Brain):
   def __init__(self, name, robot):
      Brain.__init__(self, name, robot)
      self.net = Network()
      self.net.addThreeLayers(self.getRobot().get('range', 'count'), 2, 2)
      self.maxvalue = self.getRobot().get('range', 'maxvalue')
   def scale(self, val):
      return (val / self.maxvalue)
   def step(self):
      ins = map(self.scale, self.getRobot().get('range', 'all'))
      self.net.setInputs([ ins ])
      if self.getRobot().getSensorGroup('min', 'front')[1] < 1:
         target_trans = 0.0
      elif self.getRobot().getSensorGroup('min', 'back')[1] < 1:
         target_trans = 1.0
      else:
         target_trans = 1.0
      if self.getRobot().getSensorGroup('min', 'left')[1] < 1:
         target_rotate = 0.0
      elif self.getRobot().getSensorGroup('min', 'right')[1] < 1:
         target_rotate = 1.0
      else:
         target_rotate = 0.5
      self.net.setOutputs([[target_trans, target_rotate]])
      self.net.sweep()
      trans = (self.net.getLayer('output').activation[0] - .5) / 2.0
      rotate = (self.net.getLayer('output').activation[1] - .5) / 2.0
      self.getRobot().move(trans, rotate)
```

Again, the code is quite short (30 lines) but packs in everything necessary to explore an example of on-line ANN learning on a robot.

Both of the previous examples showed direct reactive control. That is, the robot's movements were calculated on the spot, and directly sent to the motors. The final example shows that a brain is nothing more than a class, and it, too, can be changed into something more sophisticated. Consider the following:

```
from pyro.brain.fuzzy import *
from pyro.brain.behaviors import *
from pyro.brain.behaviors.core import *
import math, time
from random import random
class Avoid (Behavior):
    def init(self): # called when created
        self.Effects('translate', .3)
        self.Effects('rotate', .3)
    def direction(self, dir, dist):
        if dist < 1.0:
            if dir < 0.0:
                return 1.0 - dir
            else:
                return -1.0 - dir
        else:
            return 0.0
    def update(self):
        close_dist=self.getRobot().getSensorGroup('min','front-all')[1]
        close_angl=self.getRobot().getSensorGroup('min','front-all')[2]/
               math.pi
        self.IF(Fuzzy(0.0, 1.5) << close_dist, 'translate', 0.0)
        self.IF(Fuzzy(0.0, 1.5) >> close_dist, 'translate', .2)
        self.IF(Fuzzy(0.0, 1.5) << close_dist, 'rotate',
                self.direction(close_angl, close_dist))
        self.IF(Fuzzy(0.0, 1.5) >> close_dist, 'rotate', 0.0)
class state1 (State):
    def init(self):
        self.add(Avoid(1))
```

This brain is an example of a fuzzy logic behavior. Although this behavior has the ability to blend actions together into smoothly avoiding obstacles, it is only 28 lines long. In addition, the entire Behavior class that implements this algorithm is currently only 200 lines long, and is meant to be studied and modified by students.

3 Pyro in the Curriculum

As seen above, Pyro code is well-formatted (a Python requirement) and reminiscent of other languages' object-oriented syntax. But how would novice programmers find Pyro? To explore this issue, Pyro was used in the Spring 2002 semester in the Bryn Mawr College course "Androids: Design and Practice."

The class was composed of students from Swarthmore College, Haverford College, and Bryn Mawr College. The students' programming experience covered a wide range: from none to a lot. Although some had programming experience, none of the students had used Python prior to the class, but all of the students picked it up quickly.

The course covered basic robot navigation, obstacle avoidance, vision (including algorithms for blob detection, motion detection, color filtering, and color histograms), and tracking.

After using Pyro for the last two thirds of the semester, a questionnaire was given to them in order to explore their views. The results were positive on the use of Pyro; however, there was some confusion of understanding of the total system. For example, some students were unable to clearly delineate the boundaries of the simulator with the Pyro control system. Of course, the boundary is obvious when dealing with real robots. However, when everything is software, the separation is, apparently, not clear. This confusion is probably enhanced

because the simulators are actually started up from within the Pyro GUI. No doubt, this particular problem can be alleviated by limited use of the simulators.

Students also wished for more support in mapping abilities in Pyro. Although this isn't in the area of our research, the ability to explore localization, and mapping would make Pyro much more functional for high-level behaviors. Currently, goals to have the robot "go to room 232" are beyond the scope of what can be accomplished without high-level mapping abilities.

Overall, the students picked up Python easily, and quickly covered many advanced topics in artificial intelligence and robotics. Although there was evidence for some confusion, the majority of the important ideas were understood.

We are planning on using Pyro, or parts of Pyro, in other courses in our curricula, including the Introduction to Cognitive Science, Artificial Intelligence, Developmental Robotics, and Complexity Theory. In addition, we are planning to adapt these materials for use in other schools.

Pyro has become the central tool in our research toolbox. The ability to create complex visualizations on the fly, and change core components easily with Python's objects has benefits that far outweigh any loss in speed.

4 Summary

Pyro was designed to be a robotics API and a set of classes and libraries for exploring advanced robotics issues on a variety of hardware platforms. It was designed to allow inexperience undergraduate students to explore all levels of an artificial intelligence and robotics system, including everything under the hood. In addition, it is surprisingly fast enough to be used as our main research tool. Based on the success of Pyro in the classroom and laboratory so far, we are planning on expanding its use into other classes and projects.

Resources

Pyro is an open source, free software project. You can find the full source code and documentation at `http://emergent.brynmawr.edu/wiki/?Pyro`. This work is funded in part by NSF CCLI Grant DUE–0231363.

References

[1] Balch, T. *Behavioral Diversity in Learning Robot Teams*. PhD thesis, Georgia Institute of Technology, 1998.

[2] Beer, R. D., Chiel, H. J., and Drushel, R. F. Using Autonomous Robotics to Teach Science and Engineering. *Communications of the ACM* (June 1999).

[3] Blank, D. S., Hudson, J. H., Mashburn, B. C., and Roberts, E. A. The XRCL Project: The University of Arkansas' Entry into the AAAI 1999 Mobile Robot Competition. Tech. rep., University of Arkansas, 1999.

[4] Gallagher, J. C., and Perretta, S. WWW Autonomous Robotics: Enabling Wide Area Access to a Computer Engineering Practicum. *Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education 34*, 1 (2002), 13–17.

[5] Harlan, R. M., Levine, D. B., and McClarigan, S. The Khepera Robot and the kRobot Class: A Platform for Introducing Robotics in the Undergraduate Curriculum. *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education 33*, 1 (2001), 105–109.

[6] Klassner, F. A Case Study of LEGO Mindstorms Suitability for Artificial Intelligence and Robotics Courses at the College Level. *Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education 34*, 1 (2002), 8–12.

[7] Kumar, D., and Meeden, L. A Robot Laboratory for Teaching Artificial Intelligence. *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education 30*, 1 (1998).

[8] Martin, F. The handy board. World Wide Web, URL is http:// lcs.www.media.mit.edu / groups / el / Projects / handy-board /.

[9] Meeden, L. Using Robots As Introduction to Computer Science. In *Proceedings of the Ninth Florida Artificial Intelligence Research Symposium (FLAIRS)* (1996), J. H. Stewman, Ed., Florida AI Research Society, pp. 473–477.

[10] Prechelt, L. 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, 2000.

[11] Turner, C., Ford, K., Dobbs, S., and Suri, N. Robots in the classroom. In *Proceedings of the Ninth Florida Artificial Intelligence Research Symposium (FLAIRS)* (1996), J. H. Stewman, Ed., Florida AI Research Society, pp. 497–500.

[12] Wolz, U. Teaching Design and Project Management with LEGO RCX Robots. *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education 33*, 1 (2001), 95–99.