How Platform-Independent is Pyro?

T. Fossum and **J. Snow**[*]
{fossumtv,snow91}@potsdam.edu
Department of Computer Science
SUNY Potsdam
Potsdam NY

Abstract

Pyro is a Python-based software environment for robot control that is designed to minimize the necessity of programmers "having to worry about the low-level details of the underlying hardware." Pyro supports multiple robot platforms, but until recently, most of them were prohibitively expensive for departments with large interest in robotics but with low budgets. This paper describes our experiences at adding support for lower-cost robots in the Pyro environment.

Introduction

The Pyro project has received some well-deserved attention (and praise) in the AI education community by making it possible to use a single language (Python) "to program many different robots, allowing code to be shared across platforms as well as allowing students to experiment with different robots while learning a single language and environment."

Until recently, the principal robots supported by Pyro include the Pioneer family (Pioneer, Pioneer2, PeopleBot robots), the Khepera family (Khepera, Khepera 2 and Hemisson robots), and the AIBO. The initial cost for these robots range from about $400 for the Hemisson to about $2000 for the Khepera and significantly more for the Pioneer. The AIBOs are no longer in production. For small computer science college programs whose students want to engage in robot-related educational activities, these costs can be prohibitive. Adding peripheral sensors can drive the costs even higher.

In this project, we chose two "low cost" robot platforms – the IntelliBrain-Bot from RidgeSoft and the Roomba Red with RooTooth from RoboDynamics. The base units for these robots have movement and sensing capabilities that are minimally useful for carrying out robot-related activities.

In the remainder of this paper, we outline the Pyro infrastructure, describe how we integrated these two robot platforms into the Pyro infrastructure, and give a short summary and critique of Pyro.

[*]Supported in part by NSF grant DUE 0230030

Pyro

Pyro is intended to be as hardware independent as possible. For example, the Pyro command to move a robot forward at full speed would be coded as

```
robot.move(1,0)
```

This command would result in forward-moving behavior independent of the particular robot being controlled. (Of course, this command would make sense only if the robot in question had the capability of forward motion.)

Similarly, the Pyro command to rotate the robot counter-clockwise at full speed would be coded as

```
robot.move(0,1)
```

Sensor values that represent distances come in units including *hardware*, *raw*, and *robot*. Suppose a robot "front" sensor is to be used to determine the distance between the robot and an obstacle directly in front of the robot. Assume the sensor returns an 8-bit value to the controller, and that the 8-bit value is proportional to the distance between the robot and the obstacle. The hardware value is the actual 8-bit value the controller sees, the raw value is this value converted into centimeters (say), and the robot value is presented in "robot units" – where one robot unit is the physical diameter of the robot.

Light sensor values in Pyro do not have any predefined standard for units. Consequently, light sensor values typically are reported in the same hardware units as obtained directly by the controller from the light sensor port, possibly converted so that larger values correspond to brighter lights. This will generally not be a problem if a robot's goal, for example, is to follow a light source, since all that's needed is the relative intensity of the readings to find a maximum.

Consider the Hemisson robot, which has eight distance (range) sensors arranged clockwise around the robot: one left sensor (0), four front-facing sensors (1, 2, 3, and 4), one right sensor (5), and two back-facing sensors (6 and 7). The sensors are clustered by the robot software into groups: for example, the "front" group consists of sensors 2 and 3, the "front-left" group consists of sensors 0 and 1, and so forth. The Hemisson robot has 15 such range sensor clusters defined in Pyro.

The array of values returned by a sensor group can be retrieved as in the following command:

```
robot.range["front"]
```

This returns the array of values corresponding to the two frontmost sensors (2 and 3). The max or min of this array can be used for control purposes. Similarly,

```
robot.range["front-left"]
```

returns the array of values of sensors 0 and 1.

The IntelliBrain, as we have configured it, has two front-facing range sensors, a front-left sensor (0) and a front-right sensor (1). We cluster both of these sensors into a "front" group, and let "front-left" refer to sensor 0, for example. The IntelliBrain robot has three such sensor clusters defined in Pyro.

Assuming that a given robot platform knows how to report values corresponding to sensor groups (or individual sensors) such as "front" and "front-left", algorithms for robot behavior can be developed that are platform-independent.

One clear difference between the Hemisson and IntelliBrain robots is that the Hemisson has a much richer array of sensors, including rear sensors. Consequently, any robot algorithm that relies on rear sensor values will not work with the IntelliBrain. This alone explains why Pyro cannot be truly hardware-independent.

Robots

A *robot* is a Pyro abstraction that allows Pyro to communicate with different robot platforms through a (relatively) common interface. Following the examples given above, a robot can be expected to carry out effector actions using method calls such as `move(1,0)` and retrieve sensor values using method calls such as `range["front"]` The software for a specific robot platform that implements a Pyro robot is also called a robot *driver*. In the remainder of the paper, we will use the term "robot" to refer to the software driver, and the term "robot platform" to refer to the robot controller device and associated sensors/effectors.

When a robot is requested to carry out a `move(t,r)` action, the values of both `t` and `r` must lie between -1 and 1. For the parameter `t`, a value of -1 means "full speed" backward, while a value of 1 means "full speed" forward. Similarly, for the parameter `r`, a value of -1 means "full speed" clockwise, while a value of 1 means "full speed" counter-clockwise. The term "full speed" is not defined in the Pyro documentation, although the intuitive operative definition means "as fast as the hardware will allow". Values between -1 and 1 should result in proportional behavior, so that `move(0.1,0.3)` would move the robot platform slowly forward and somewhat to the left (counter-clockwise).

Each robot must define an `update` method that reads the robot platform's hardware values by interrogating the controller's sensor ports, converts the hardware values (if required) into appropriate data units (such as raw or robot units) and packages these values into data structures (such as `range`) that Pyro can retrieve later.

Depending on the robot platform's hardware characteristics and serial communication speed, the `update` method may take some time to complete. An `update` method call typically will send one or more messages to the robot platform through the serial interface requesting that the controller retrieve the specified sensor values, and will wait for responses from the controller through the serial interface for return values. Pyro does not specify any protocols to govern the communication between Pyro and the robot platform; any such protocols are up to the robot driver to define and implement.

Brains

A *brain* is another Pyro abstraction that defines a specific robotic behavior such as following a line, seeking out a light source, or solving a maze.

In order to carry out a brain algorithm, communication between Pyro and a specific robot platform must have been established by loading the appropriate robot driver (see above).

Each brain implements the `step` method which is called repeatedly – once every 1/10 second, for example, depending on configuration. The `step` method interrogates the robot sensors by implicitly calling the robot's `update` method as described above. The `step` method then uses the information retrieved by the robot from the robot platform (such as interrogating the `range` values) to carry out robotic activities (such as calling the `move` method with appropriate parameters) as defined by the brain.

Notice that the "control loop" between the brain and the robot is also implicit, carried out through the repeated calls to `step` The rate at which these calls are made determines, in part, the responsiveness of the robot. If the calls to `step` occur too quickly, the robot may not be able to retrieve and package the sensor values successfully. If the calls occur too slowly, the robot may not be able to respond quickly enough to external events.

Pyro summary

Each robot platform must have a corresponding Pyro entity called a *robot* that communicates between Pyro and the robot platform through a serial interface. A Pyro robot must define an `update` method that retrieves and packages data from the robot platform sensors into Pyro data structures and that defines appropriate methods such as `move` to carry out out robot-specific actions.

Loading a Pyro robot driver results in initializing the robot data structures and establishing a serial communication channel between Pyro and the robot platform.

Each robotic algorithm must be defined by a Pyro entity called a *brain*. A Pyro brain must define a `step` method that is called repeatedly by the Pyro system and that, after making an implicit call to the robot's `update` method, examines the robot sensor information and carries out appropriate robotic activity such as `move`

Loading a Pyro brain results in defining the behavior of the robot by means of the implicit `step` control loop. When the brain is *run* the `step` method is called repeatedly to carry out the desired robotic behavior.

IntelliBrain-Bot

RidgeSoft, a Pleasanton CA company, produces the IntelliBrain controller, a robotics controller based on the Atmel ATmega128. The IntelliBrain-Bot is a robot platform that uses the IntelliBrain controller mounted on the Boe-Bot chassis.

RidgeSoft displayed their IntelliBrain-Bot at the SIGCSE 2006 vendor exhibits in Houston. The first author of this paper obtained an evaluation kit from RidgeSoft, which was the basis for carrying out this aspect of the project.

IntelliBrain controller summary

The IntelliBrain controller has 132K RAM and 128K flash memory. Communication between a host PC and the controller is through a RS232 serial interface. The controller has a 16x2 liquid crystal display, a buzzer, thumbwheel, and two programmable LEDs.

The form factor of the IntelliBrain controller is about 65% smaller than that of the Handy Board – a robotics controller familiar to many robot enthusiasts. The computational speed and memory capacity of the IntelliBrain is roughly an order of magnitude greater than the Handy Board.

The IntelliBrain controller uses an on-board Java interpreter to execute bytecode loaded from a host computer. The interpreter is multi-threaded and supports several precompiled classes for communicating with devices such as drive motors and sensors.

IntelliBrain-Bot Robot Chassis

The IntelliBrain-Bot combines the IntelliBrain controller with the Boe-Bot^{TM}aluminum robot chassis. The metal platform is more rugged and less prone to spontaneous self-destruction than platforms based on Lego bricks.

The IntelliBrain-Bot includes the robot chassis, the IntelliBrain controller, a four AA-cell battery holder, and two infrared photoreflector sensors. The chassis is rectangular, about 13cm long and 9cm high (including the controller). The IntelliBrain-Bot comes bundled with RoboJDE development software.

The wheels on the chassis are driven by two servo motors which can be independently controlled by the IntelliBrain. The top forward speed of the chassis is about 8cm/sec.

We found ourselves frequently plugging and unplugging cables into/from the DB-9 connector on the controller board. Eventually, the connector will wear out or the traces on the board will break from stress. We considered attaching a serial pigtail to the DB-9 connector as a solution to this. The Handy Board solution – a simple RJ-11 snap connector – would have worked better.

RoboJDE

RoboJDE is a Windows^{TM}-based IDE that is specifically designed by RidgeSoft for robot programming in Java. RoboJDE includes a Java compiler designed specificlly for robot control, and includes special class libraries for hardware devices such as sensors and motors. The RoboJDE downloads bytecode files to the IntelliBrain through a serial interface. The bytecode files can either reside in RAM or in flash memory. RAM-loaded files disappear when the robot is powered down, whereas flash memory files are persistent. The on-board flash memory will degrade after a significant number of downloads. When used with Pryo, controller downloads are necessary only when adding new hardware interfaces.

IntelliBrain controller software

Since the IntelliBrain robot controller is programmable and has a hardware configuration similar to the Hemisson robot platform, we chose to implement its Pyro robot interface as a special case of the Hemisson robot.

IntelliBrain controller command structure The Hemisson robot platform is controlled through a serial interface by sending ASCII string commands through a serial interface to the robot platform and reading ASCII string results from the robot platform through the same serial interface.

For example, sending the command string `D,127,0` to the Hemisson robot platform through the serial interface will result in giving full forward power to the left motor and no power to the right motor. (All command strings are terminated by an ASCII carriage return character.) The Hemisson platform acknowledges the `D` command string by sending an ASCII response string of the form `d` back through the serial interface. (All return strings are terminated by an ASCII carriage return character.)

We chose to adopt the same command/response structure for the IntelliBrain as the Hemisson. Our reason for this choice was to use the Hemisson Pyro robot driver code with as few changes as possible.

To interrogate the IntelliBrain's two proximity sensors, we send the string `N` to the controller and receive an answer string in the form `n,x0,x1` where `x0` and `x1` are the values of the front-left and front-right sensors. (The Hemisson has eight proximity sensors, so its return string would have eight return values instead of two.) The hardware values `x0` and `x1` we receive from the controller are in the range from 0 to 80 and are already scled to cm units. These values are returned unaltered in the `range` structure.

To interrogate the two light sensors, we send the string `O` to the controller and receive an answer string in the form `o,y0,y1` where `y0` and `y1` are the values from the two sensors. The hardware values retrieved from the controller are in the range from 0 to 255, with 255 corresponding to darkest and 0 corresponding to lightest. Since Pyro expects darker vlaues to be less than lighter values, we subtract the hardware values from the maximum value of 255 before these values are returned in the `light` structure.

IntelliBrain controller code We used the RoboJDE to write the Java code to be executed on the IntelliBrain controller. This amounted to writing a tiny interpreter that would read an incoming command line from the controller serial interface, parse the line to identify the command and its parameters, carry out the command actions, and sending a response string to the serial interface.

We tested our initial and subsequent versions of our IntelliBrain controller code using a terminal emulator on a host PC. On our Linux box, we used gnome-terminal. Since the IntelliBrain code uses human-readable ASCII strings for both input and output of command and respons information, we were able quickly to test the controller interface before attempting to make it work with Pyro.

Pyro IntelliBrain robot driver code

Since our original aim was to re-use as much of the Pyro Hemisson code as possible, our original code for both the IntelliBrain controller and the Pyro robot driver attempted to match the Hemisson hardware characteristics as closely as possible – for example, the maximum values for the motor speed parameters. As we gained experience with the IntelliBrain controller code and the corresponding Pyro robot driver, we simplified the IntelliBrain controller code so that it incorporated "native" IntelliBrain controller values instead of attempting to fudge Hemisson-like values, and we made correspondingly appropriate changes to the Pyro robot driver code. Still, the Pyro robot driver code for the IntelliBrain is remarkably similar to the original Hemisson code.

In all, our experiences with writing both the IntelliBrain controller code and the Pyro robot driver code was rewarding and successful.

Roomba

Roomba is a robotic vacuum cleaner developed by iRobot in Burlington, MA. More than two million Roomba vacuuming robots have been sold worldwide.

Unlike the IntelliBrain, the Roomba is not programmable at the controller level. However, the Roomba does provide a serial interface to its controller. Through this serial interface, commands can be sent to the controller to carry out various actions (move, turn on/off vacuum motor) and to inquire about the controller state.

Roomba robot chassis

The Roomba robot chassis is housed in a rugged circular plastic enclosure measuring 33cm in diameter and 8cm high. The drive wheels of the chassis are independently controlled by the Roomba, but the serial command interface does not allow control of the motors separately. The top forward speed of the chassis is about 50cm/sec.

The chassis also includes a vacuum motor and a "side brush" each of which can be turned on and off under serial command. An internal speaker can be programmed to play musical tones using MIDI note definitions and duration.

The Roomba has a number of built-in sensors that can be interrogated with serial commands. Two sensors are attached to the drive wheels and identify whether the wheels have "dropped" (for example, if the Roomba has been lifted off the floor or if a wheel has fallen over a step); another drop sensor is attached to the non-driving caster wheel. Two front bump sensors – left and right – determine if the Roomba has struck an obstacle during forward motion. Two "cliff" sensors – left and right – are used to determine if the Roomba has detected an falling edge such as a stair.

All of the Roomba sensors described above return binary results. There are no built-in sensors that can determine range or light.

As with the IntelliBrain, we found ourselves frequently plugging and unplugging the DIN serial connector pigtail into/from the Roomba. At one point, one of the pins on the male DIN connector bent, requiring a careful straightening of the pin in order to make the connector function again. A more rugged connector scheme would improve this situation.

Roomba controller command structure

The serial interface to the Roomba allows an external device to send commands to control the Roomba motors or to interrogate the states of Roomba sensors. All commands are byte sequences with one command opcode followed by zero or more data bytes. The command opcodes are in the range 128 to 143.

To enable the full command repertoire of the Roomba through the serial interface, the following bytes must be sent:

Opcode	**Command**
128	Start the command interface
130	Enable user control of the Roomba
132	Enable unrestricted (full) control

The Roomba drive motors are controlled by the Drive command with opcode 137, followed by four data bytes. The first two data bytes encode a 16-bit two's complement number (high-byte first) that represents the velocity of the Roomba in mm/sec, and the next two data bytes (similarly encoded) represents the radius of curvature that the Roomba should turn. A larger radius makes the Roomba drive straighter. If the radius is positive, the Roomba turns to the left, otherwise the Roomba turns to the right. For example, the following five-byte command

Opcode/Data	**Meaning**
137	Drive opcode
255	high byte of -200 in 2's complement
56	low byte of -200 in 2's complement
1	high byte of 500 in 2's complement
244	low byte of 500 in 2's complement

would drive the Roomba in reverse at a velocity of 200mm/s with a leftward turning radius of 500mm.

Except for the Sensor command (described below), the Roomba does not send any acknowledgement across the serial interface indicating the success or failure of the command.

Sensor and other internal state information can be retrieved from the Roomba through the serial interface by sending the Sensors opcode followed by one byte "packet code" that determines what sensor values should be returned. A packet code of zero will result in a byte strem of 26 values to be sent by the Roomba across the serial interface. The exact format of the return values is documented in the Roomba Serial Command Interface Specification.

For example, the following two-byte command

Opcode/Data	**Meaning**
142	Sensors opcode
0	Send all 26 bytes of sensor data

would result in the Roomba sending 26 bytes across the serial interface.

The first of the 26 returned bytes, for example, gives the state of the wheeldrops and bump sensors. Bits 2, 3, and 4 of the byte correspond to wheeldrops for the right and left drive wheels and the caser, respectively; bits 0 and 1 of the byte correspond to the right and left bump sensors, respectively. Each of these bits is interpreted with 1=true and 0=false.

Pyro Roomba robot driver code

Our robot driver code for the Roomba is dramatically different from the IntelliBrain code.

First, the IntelliBrain robot driver communicates with the controller through human-readable ASCII command sequences, and the controller always returns an ASCII response sequence for every command it receives. The Roomba robot driver communicates with the controller through byte-oriented opcodes and fixed-size data packets, often with data encoded in 2's complement binary format.

Second, the IntelliBrain robot driver has full control over both of the robot platform drive motors; calls to the `move` method must transform the *translate* and *rotate* values into appropriate independent motor actions. Aside from scaling issues, the Roomba robot driver need only convert the *rotate* component of a `move` method calls into an appropriate radius values, and this is a simple inverse relationship.

Third, the IntelliBrain robot driver can be commanded to retrieve range and light sensor data independently, and the values returned are proportional to what the sensor detects. The Roomba robot driver must retrieve all the sensor data at once, and the values returned are all bit values (0 or 1).

Since the 26-byte sensor data packet sent from the Roomba does not have any identifying start- or end-of-packet markers, we discarded any junk data read from the Roomba serial interface prior to sending the Sensor command, and then proceeded to read the next 26 bytes for further processing.

The primary Roomba sensors used for navigation around obstacles are the two front bump sensors. Since the values reported by the Sensor command return only bit values, any Pyro brain code that uses range-type values for measuring proximity will not work interchangeably with the Roomba.

Serial communication

While not part of the required Pyro infrastructure, we were interested in establishing wireless serial communication between the host PC runnig Pyro and our robot platforms.

RoombaDevTools sells a package consisting of the Roomba Red (the lowest-end Roomba, yet still fully functional) and a BlueTooth serial interface that can communicate with standard Bluetooth serial dongles. This package is available for $229.

The IntelliBrain has a standard DB-9 connector for serial communication. Several commercial Bluetooth devices are available that serve as wireless RS232 communications channels. RidgeSoft recommends the AIRcable serial module, with a list price of $69. With a IntelliBrain-Bot package price of $358 for the base unit and a set of range and light sensors, the entire package – including the Bluetooth serial adapter, comes to $427. The Bluetooth device needs external power, which can be drawn from the IntelliBrain-Bot's battery pack or from a 9V battery through a 5V voltage regulator.

We purchased a Bluetooth serial adapter (different from the AIRcable) for use with the IntelliBrain controller. That adapter required hardware handshaking – which was not described in the adapter documentation – which the IntelliBrain controller does not provide. We added jumpers to the appropriate pints on the serial cable header to provide the various hardware handshaking signals, after which the Bluetooth serial adapter worked fine.

Conclusions

So is Pyro truly platform-independent?

Our experience writing Pyro robot drivers for the two robot platforms underscored the importance and value of implementing methods of an abstract interface to achieve a certain measure of "platform independence." Along the way, it gave us the opportunity to understand better the underlying hardware and how to report status information in a uniform way.

Yet differences in hardware are inescapable, and a robot brain that works with one robot platform may not work with another. For example, the Hemisson has a rich array of range sensors at the four compass points and at some points in between. The Roomba has two "range" sensors that are binary valued and only sense obstacles it bumps into when the robot is moving forward.

Here are several other observations we have made during our project:

- *Installation* – The Linux installation instructions on the Pyro web site (`http://www.pyrorobotics.org`) are tailored to the Red Hat distributions. There are many other popular Linux distributions, including Ubuntu and Debian, that the installation instructions do not mention. The installation page does not give details about how to install from one of the download directories mentioned at the top of the page. Moreover, once downloaded, the README file in the downloaded distributions simply points to online documentation.
- *Versions* – The distribution sites provide dozens of versions of Pyro. The following sites are listed on the installation page:

  ```
  http://pyrorobotics.org/tars
  http://pyrorobotics.org/download
  ```

  The line prededing this list says, "If you find that you need any files, please check here."

  The `tars` directory has distributions with names of the form `pyro-x.x.x.tgz`, with the highest number being `pyro-3.6.1.tgz`. In addition, there's `pyro-latest.tgz`, which (apparently) is version `3.6.2`. Normally, we would expect that `pyro-latest.tgz` would be a symbolic link to the latest version.

  The `download` directory has distributions with the names of the form `pyrobot-x.x.x.tgz`, with the highest number being `pyrobot-4.8.2.tgz`.

  What is the purpose of both the `tars` and `download` directories? The documentation gives no clue.

- *Brains* – There is no high-level documentation (such as a README) about what the the sample brains do.
- *Documentation* – The documentation does not make it clear whether `rotate(1)` moves the robot clockwise or counter-clockwise.
- *AIBOs* – Our initial interest in Pyro was prompted by our efforts to experiment with robotic behavior using our two AIBO robots. After months of searching, we could not find any reliable source for Sony Pink Memory Sticks, so we decided to pursue other robot platforms.
- *Documentation redux* – The Pyro documentaiton is best understood by programmers who already know Pyro. After completing the work on two robot drivers, we now have a deeper understanding of the Pyro structure, but better documentation and clearer examples would have helped considerably towards being more productive and meeting our project objectives earlier.

In conclusion, our work with Pyro was illuminating, challenging, and productive. We look forward to seeing a wider adoption of Pyro in the robotics community, especially in education. We hope that the results of our project – bringing two low-cost robot platforms under the Pyro umbrella – will help to see this happen.