A Python Software Library for Simulation of Mobile Robots

INESC-ID Technical Report 18/2013
Reference Number 9172

Rafael das Almas Ascensão
m8821@alunos.uevora.pt

June, 30, 2013
1 Introduction 2
  1.1 The Pololu 3pi ........................................ 2
  1.2 Simulation Platforms ................................. 3
     1.2.1 Gazebo ........................................... 3
     1.2.2 Morse ............................................ 3
     1.2.3 Webots ........................................... 3
     1.2.4 Player Project .................................... 4

2 Simulation Framework 5
  2.1 Framework Development ............................... 5
     2.1.1 The Agent Class .................................. 6
     2.1.2 The Environment Class .............................. 7
     2.1.3 Practical Usage Example ........................... 8
     2.1.4 Concerns and Limitations .......................... 8

3 Example using the Framework 9
  3.1 Simulation of a 3pi robot with sensors .............. 9
     3.1.1 Model ............................................ 9
     3.1.2 Kinetics .......................................... 10
     3.1.3 Dynamics ......................................... 10
     3.1.4 Sensors ........................................... 11
     3.1.5 Results .......................................... 12

4 Conclusion 17
Chapter 1

Introduction

Before implementing an application in real world robots, it is important to test and explore the application on a simulation platform so that mistakes can be detected and corrected, saving the developer time and resources of deploying versions that are doomed to fail. Currently, there are numerous robotics simulators that account many functions, such as using different kinds of robots, simulating multiple agents systems, different sensors, test the models in a 3D environment, account for the laws of physics and transfer the programs developed in the simulation environment to real robots.

1.1 The Pololu 3pi

The Pololu 3pi is a mobile robot with a circular shape and two independent wheels that are managed by two independent control units. With a 9.5 cm diameter it is also a relatively small robot.

On the inside, this robot has a ATmega168 processor, 20 MHz frequency, 32KB of flash memory, 2 KB of RAM, and 1 KB of EEPROM memory. It also includes infra-red sensors, a trimpot and four LEDs. However, users can add their own sensors to expand its functionality. It can be programed in C or C++ using the several libraries provided by Pololu. The robot can also be programed using the popular Arduino Platform.

These characteristics make this robot suitable for tasks like following a line or maze-solving, but its functionalities can be expanded by adding sensors (e.g. distance, temperature etc) [4].

There’s also the possibility to control the robot using the m3pi library from “mbed rapid prototyping platform” which uses an ARM 32-bit microcontroller in a Master-Slave configuration where the ARM microcontroller is responsible for the robot behaviour/program and the native AVR microcontroller is responsible for dealing with simple tasks such as driving the motors, lcd, etc.

This allows more processing power and more I/O lines. Thus, new and more complex features (e.g. Wireless capabilities with Xbee or Wixel modules) can be achieved that otherwise would be difficult or impossible to attain with the native microcontroller.

\[\text{http://www.arduino.cc/}\]
1.2 Simulation Platforms

This section is a quick review of some of the robot simulators available, where their main characteristics will be described.

1.2.1 Gazebo

Gazebo is a software for outdoors multi-robots simulation, it is capable of simulate articulated robots in complex and 3D realistic environments. With this software it is possible to build the robot model without programming, hence it provides models of some real robots (e.g. PR2, iRobot Create, etc.) as well as generic robots arms and grippers. Gazebo is known for its aptitude to simulate a population of robots and for its dynamics simulations, where realistic sensor feedback and object interactions are verified. The user has direct control over various parameters of the simulation and can also view and navigate through the simulation while it is running. [2]

1.2.2 Morse

Morse is a generic and versatil simulator for indoor and outdoor environments which includes control of lightning, multi-texture and physics through the Bullet library. This software has standard sensors (e.g. cameras, GPS, etc.), actuators (speed controllers, generic joint controllers, etc.) and robotic bases (e.g. PR2, generic four wheel vehicles), beyond that it is easy to add new features/robots hence both the environment and the robots can be built on an application or on a Python API. Using Bender the user can choose the level of reality associated with the simulation (e.g. movement can be choose over vision). [3]

1.2.3 Webots

This software is described as providing a dynamic simulation where the user can define various parameters such as mass, center of mass or fluid dynamic. The robots and environment design can be made using Webots libraries, which have several robots (e.g. iRobot Create, Katana, Shrimp III, etc) and objects (e.g. walls, books, doors, etc.), alternatively the user can make the design in an external software, such as Blender, AutoCAD or SolidWorks. The user can also change some settings that influence the simulation’s performance (e.g. by re-defining each object’s collision bounds), and implement new sensors or actuators. While the simulation is running the user can interact with it, take screen-shots and record movies.

This application can run an unlimited number of robots and enables simulations of communication between robots accounting for the existing obstacles and distance between robots.

This application is able to simulate several robots and their communications. It accounts the robot’s position and orientation and the effects of distance and obstacles.

Webots’ API is available in 6 languages (C, C++, Java, Python, Mathlab and URBI) and can also link with external libraries. This application’s files have a readable format that allows the user to modify them. [1]
1.2.4 Player Project

The Player project aims to create free software for robot and sensor systems research. It includes the Player network server and the Stage robot simulator platform.

Player is an interface for robots and sensors over the IP network, allowing the use of any programming language to write the robot control programs. It is possible to have multiple concurrent connections providing several options for distributed and collaborative sensing and control. It also permits the use of several hardware and software, such as robots (e.g. Roomba, Mindstorms NXT, etc.), hardware (e.g. Geko201 - handheld GPS receiver; etc.), software (e.g. CMVision - blob-tracking software, Sphinx2 - speech-recognition software, etc.), algorithms and simulators (Stage and Gazebo, which is mentioned in section 1.2.1).

The Stage Simulator creates a 2D simulation environment with a population of multiple robots, sensors and objects and provides several sensor and robot models. With this simulator it is possible to research multi-agent autonomous systems. When Player and Stage are used together, Stage works as a Player’s “client”. Often, Stage can work as a C++ library, providing robot simulation inside other programs, which is useful when Player is not an option for the intent of the user/developer.
Chapter 2

Simulation Framework

2.1 Framework Development

The main objective of this framework is to have the ability to represent and simulate a generic Dynamical System.

A Dynamical System can usually be represented by two sets of functions: the kinematics equations and the dynamics equations. The former describes the rules of the motion, e.g. velocity of the object, the latter describes why such objects move the way they do, e.g. a force applied on an object will result in a certain type of motion.

However, in complex problems it is difficult to characterize the system as a whole, so in order accomplish this task one can divide the problem into several, simpler sub-problems in a divide-and-conquer manner.

One way to represent this is using a block diagram, where each block defines the behaviour of a part of the system and the interactions between this block and other components are made via input and output ports on each block.

The framework was developed based on this idea, allowing the user to create objects to represent blocks and define their interactions by using functions that “send” the output of a block to the input of another.

The simulation framework consists of two classes, the Agent class and Environment class. The first class is responsible for representing each block. It stores the state, the connections between blocks and references to the user-made functions that characterize the sensors/output. The second, keeps track of all blocks used in the simulation, controls the simulation and updates the blocks with the new values provided by the simulation process.

The language chosen to develop this application was Python and the science...
2.1.1 The Agent Class

The Agent Class is the part of the framework that represents the blocks and their interactions. Each block has a state, \( n \) input ports and \( m \) output ports. The input is fed values returned by the output functions which the input port is connected to.

In order to represent this information, the class was structured as presented below.

**Instance Variables.**

Each instance of the Class Agent has the following instance variables:

- **state**: A numpy array that stores the state of the block.
- **input**: A dictionary which maps the input ports from the block to the output function of the block it is connected to.
  
  \[
  \text{input}[\text{input_port}] = (\text{other_system}, \text{output_port})
  \]

  Where **input_port** and **output_port** are string labels that identify the input and output ports and **other_system** is a reference to another agent instance.

- **output**: A dictionary that contains references to the user made output functions that calculates each output. \( \text{output}[\text{output_label}] = f \)

- **update**: A reference to an user provided function that returns the derivative to be used in the simulation to update the state.

- **name**: A string with the name of the object.

- **input_state**: A list that stores the values on the input ports of the block at each iteration.

**Methods.**

The agent class has the following methods:

- **connect(input_port, other, output_port)**: The `connect` function connects the input and output from two blocks. \( A.\text{connect}(\ "a", \ B, \ "b" \) \) would connect the input “a” of block \( A \) to the output “b” of block \( B \).

- **set_update_function(f)**: Assigns the the function \( f \) as update function.

- **add_output_function(output_port, f)**: This function adds an output to the block and sets \( f \) as the function which computes that output.

  For instance, after adding the compass function with \( A.\text{add_output_function}(\ "compass", f \), calling \( A.\text{output}[\ "compass"]() \) will result on a call of the function \( f \).

  This method binds\(^1\) the function \( f \) to the object \( A \).

---

\(^1\)This is an example of a “monkey patch” which is a way to extend or modify the run-time code of dynamic languages without altering the original source code. [http://en.wikipedia.org/wiki/Monkey_patch](http://en.wikipedia.org/wiki/Monkey_patch)
Other methods exist but they are not intended for the user. They are used by the Environment class as explained next.

2.1.2 The Environment Class

The Environment class is the framework component responsible for running the simulation itself. It transforms the data represented by agent blocks in order to use the ODE methods from scipy to perform the simulation.

The scipy.integrate.ode solves an equation system:

\[ \dot{x}(t) = f(t, x) \] (2.1)

Since the agent’s behaviour is defined by its update function, and this function returns the derivative of the state, the integrate method can be used to compute the next state.

Mode of Operation.

In order to use the ODE methods from scipy, we need to retrieve the information from all Agent objects and aggregate it. The state needs to be a single, unidimensional numpy array and we need one function to define the whole system.

To achieve this, the function initial_values() constructs a state by making a numpy array that is the concatenation of the states from all Agent objects. In a similar fashion, the function that defines the whole system, update(), is a wrapper that calls each update function, and because this is a function of time \( t \), state \( x \) and input \( u \), the class appropriately passes the correct slice of the environment state, and computes the input by calling all output functions of the objects that it is connected to.

Instance Variables.

The only instance variable of an Environment object is a list of references to all Agents belonging to this Environment.

This list has multiple purposes, such as store references to all objects, allowing the simulation call the object’s methods; retrieve the agent’s state length and order in the list so as to compute where it lies on the environment state; allow the agents’ output functions to access another agent’s variables.

Methods.

update(): This is the differential function we need to solve. The result is build by calling the update function from each agent on the Environment’s list of agents.

initial_values(): This function builds the initial state from the states of the agents on environment’s list of agents

simulate(): This function returns a generator that returns a tuple \((t, y)\) where \( t \) is the timestamp of the current iteration and \( y \) is the current state of the simulation.

Since this method uses the Python’s yield keyword, it is possible to control and inspect the simulation at each step.
add_agent(agents): Adds each agent on the agents list to the Environment’s list of agents. This list keeps a reference to all agents included in the simulation.

update_child(): This method updates the state of the members of the simulation in order to be consistent with the current iteration.

### 2.1.3 Pratical Usage Example

- Create an agent instance for each agent.
- Set the object update function f with:
  ```python
  agent.set_update_function(f).
  ``
  The f function needs to have the following signature: \( f(self,t,x,u) \)
- Add the output functions with:
  ```python
  agent.add_output_function(output_f).
  ``
- Connect each input to the correspondent output with:
  ```
  agent1.connect('input', agent2, 'output').
  ``
- Create an environment instance \( \text{world} = \text{Environment}() \).
- Add all agents to the environment using \( \text{world.add_agents(agents)} \).
- Create the simulation generator object choosing the initial time, final time and step with \( \text{sim} = \text{world.simulate(t0=0, t=60, dt=.5)} \).
- Iterate over the sim object to retrieve the values of the simulation at each step with \( t,x = \text{sim.next()} \).

### 2.1.4 Concerns and Limitations

At time of writing, the framework has some limitations.

- The implementation doesn’t solve nor detects algebraic loops that may occur in the layout of agent blocks.
- The framework doesn’t support the creation of nested blocks.
- The framework doesn’t animate the result of the simulation, however, it’s possible to accomplish this by storing the values provided by the simulate() generator.
Chapter 3

Example using the Framework

3.1 Simulation of a 3pi robot with sensors

This example shows how to simulate multiple independent robots using the developed framework. Each robot is composed by two blocks, a “robot block” and a “controller block.”

The controller block represents the “brain of the robot”, it is the controller block who decides the target velocity of each wheel of the robot based on the information provided by the robot sensors (output ports of the robot block).

The robot block receives the target velocity of each wheel and then tries to reach that target.

Figure 3.1: Simulation with Robots using ultrasonic range finders

3.1.1 Model

Considering a circular robot with two wheels on opposite sides, radius $r$, $v_l$ and $v_r$ the velocity of the left and right wheels respectively, and $\theta$ the angle between

\footnote{The dotted arrows represent the agent accessing the environment class to compute the distance to the other agent.}
the $y$ axis and the perpendicular of the robot’s wheels imaginary axle. The angle increases in the clockwise direction, analogous to a compass heading.

Note that $v_l$ and $v_r$ aren’t the angular velocity but the velocity of the wheel on the $xy$ plane.

Figure 3.2 illustrates these considerations.

### 3.1.2 Kinetics

The differential equations (3.1) define the derivatives of $x$, $y$, and $\theta$.

The first two equations, (3.1a) and (3.1b), are the $x$ and $y$ components of the velocity vector $\vec{V}$ of the robot. The velocity of the robot can be obtained by averaging the velocity of each wheel.

The derivative of the angle can be obtained by averaging the difference between the velocity of the left wheel $v_l$ and the right wheel $v_r$, the rate of rotation is inversely proportional to the radius $r$, leading to the equation (3.1c).

\[
\begin{align*}
\dot{x} & = \frac{v_l + v_r}{2} \cos \theta \\
\dot{y} & = \frac{v_l + v_r}{2} \sin \theta \\
\dot{\theta} & = \frac{v_l - v_r}{2r}
\end{align*}
\]

### 3.1.3 Dynamics

The first order differential equations (3.2) define the behaviour of the robot when it receives a command. The target velocity is $v^*$, $m$ is the mass of the robot and $k_2$ is a constant that translates the command value to a velocity, assuming the conversion is linear. The role of the $k_1$ is to define how fast, (or how slow) the robot reaches the target velocity, as $k_1$ increases the robot gets
more responsive. This assumes that the wheels have a perfect grip, or in other words, they don’t skid.

\[ \dot{v}_l = \frac{k_1}{m} (k_2 v_l^* - v_l) \quad (3.2a) \]

\[ \dot{v}_r = \frac{k_1}{m} (k_2 v_r^* - v_r) \quad (3.2b) \]

### 3.1.4 Sensors

The sensors are responsible for giving information of the real world to the robot. Some sensors have a quick response so it can be considered they are providing the information in real time, while others are slow therefore this dynamic needs to be simulated by the platform.

In the following subsections the equations which define the behaviours of the compass, rate gyro and ultrasonic range finder are described.

**Compass.**

The compass measures the intensity of the Earth’s magnetic field along each axis. Being relatively slow, it has its own dynamic that needs to be simulated. This is accomplished using a first order differential equation (3.3), where \( \Phi^{real} \) is the real magnetic field, \( \Phi \) is the value the measured by the sensor, and \( \eta \) is the noise the sensor reading is subject to. The \( k \) constant defines the sensor’s responsiveness. The greater the \( k \) the more responsive the sensor is.

\[ \dot{\Phi}_x = k (\Phi_{x}^{real} - \Phi_x) + \eta_x \quad (3.3a) \]

\[ \dot{\Phi}_y = k (\Phi_{y}^{real} - \Phi_y) + \eta_y \quad (3.3b) \]

**Rate Gyro.**

The rate gyro indicates the rate of change of an angle. It is generally fast, so its value can be calculated directly from the state variables.

\[ \omega = \frac{v_r - v_l}{2r} + \eta \quad (3.4a) \]

**Ultrasonic Range Finder.**

The ultrasonic range finder was one of the more complex sensors to simulate. This is mostly because it needs information about other agents to compute the returned value.

In this example, the robot checks an area to determine if there are obstacles/other agents and returns the distance to the nearest object. This area is defined by four points \( (p_1, p_2, p_3, p_4) \), calculated based on the robot’s position \( (x_0, y_0) \), its heading \( \theta \) and the sensor’s parameters: the field of view, where
\[ \delta = 0.5 \times \text{fov}, \text{ minimum range } r_{\text{min}} \text{ and maximum range } r_{\text{max}} \text{ as shown on equations (3.5).} \]

\[
\begin{align*}
p_{1_x} &= x_0 + r_{\text{max}} \cos \left( \frac{\pi}{2} - \theta + \delta \right), \\
p_{1_y} &= y_0 + r_{\text{max}} \sin \left( \frac{\pi}{2} - \theta + \delta \right) \\
p_{2_x} &= x_0 + r_{\text{min}} \cos \left( \frac{\pi}{2} - \theta - \delta \right), \\
p_{2_y} &= y_0 + r_{\text{min}} \sin \left( \frac{\pi}{2} - \theta - \delta \right) \\
p_{3_x} &= x_0 + r_{\text{min}} \cos \left( \frac{\pi}{2} - \theta - \delta \right), \\
p_{3_y} &= y_0 + r_{\text{min}} \sin \left( \frac{\pi}{2} - \theta - \delta \right) \\
p_{4_x} &= x_0 + r_{\text{max}} \cos \left( \frac{\pi}{2} - \theta - \delta \right), \\
p_{4_y} &= y_0 + r_{\text{max}} \sin \left( \frac{\pi}{2} - \theta - \delta \right)
\end{align*}
\]

Then, the poligon \((p_1, p_2, p_3, p_4)\) is created using the \texttt{matplotlib.path.Path} class and the check is made with the \texttt{Path.contains_point()} function. This function uses the \textit{Jordan curve theorem} to determine if the point lies inside the poligon\(^2\).

### 3.1.5 Results

This subsection shows the results of some simulations performed using this framework. The examples that follow are mainly to show the properties of the robot and sensors being simulated.

**Robot running in a circle.**

The objective of this example is to show the behaviour of the compass. As mentioned previously, the compass is slow and has its own dynamic that needs to be simulated. To demonstrate this, two compasses with different characteristics were added to the same robot. The robot moves in a circle which alters the experienced magnetic field along its axis.

The first compass has \(k = \frac{1}{67}\), making it very responsive, and the second has \(k = \frac{1}{37}\), resulting on a slower compass. The results are shown in figures 3.6 and 3.7 respectively.

\(^2\)Details about the method used can be found at \url{http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html}
The dashed line represents the real magnetic field experienced by the robot and the solid lines represent the data returned by the sensor. It is possible to see the effects of the $k_1$ parameter and the sensor’s noise.

**Two robots on a collision course.**

In this example, two robots are on a collision course. When they detect each other two scenarios are considered. In the first, both robots try to stop as soon as possible allowing us to see how the parameter $k_1$ affects responsiveness. In the second scenario, both robots follow a simple “turn-right” to avoid collision.

**Stop.** The purpose of this example is to show the reaction time of the robot given the $k_1$ parameter. It is possible to see that the robot gradually slows when the target velocity is set to 0 rather than stopping abruptly. In the example illustrated by figures 3.8 and 3.9, the red robot has $k_1 = 3$ and the blue robot has $k_1 = 4$.

The $k_1$ parameter affects both acceleration and deceleration, so a robot with higher $k_1$ will have better acceleration and deceleration.

**Turn Right.** This example is a simple use case for the ultrasonic range finder sensor. On this example both robots follow a simple “turn right to avoid” protocol to avoid each other. When there is the possibility of collision, i.e. when one robot detects the other’s presence, it begins to turn right and as soon as the robot’s sensor stops detecting the other the robot will continue on a straight line. Figures 3.10 and 3.11 illustrate this scenario.
Following Robot with PID Control.

This example simulates a scenario where one robot follows another. The first robot, the leader, runs through the map without any sensor or communication devices. The task of the second robot is to follow the leader maintaining a predetermined distance between them. This was accomplished with the implementation of a PID controller tuned with the Ziegler–Nichols method.

Three different situations are explored in the following examples. In the first two the robots don’t make turns, they can only accelerate and decelerate on a straight line, in the third they are free to make turns.

Following In a Straight Line with constant velocity. The master robot, who’s target velocity remains constant, is pursued by a second robot who must maintain a fixed distance of five length units. When the simulation starts both robots are separated by 2.5 units. The results are illustrated in figures 3.12, 3.13, 3.14 and 3.15.

Following In a Straight Line with variable velocity. In order to simulate unexpected behaviour by the master robot during the pursuit, this examples shares all the characteristics of the previous one, however the master robot varies its target velocity during the simulation. The results are illustrated in figures 3.16, 3.17, 3.18 and 3.19.
Following a robot that curves. In this example the master robot runs with a constant target velocity and at a given instant \( t \) changes its direction.

The other robot is equipped with a modified ultrasonic range finder\(^3\) that gives the bearing to the other agent, \( \theta_{rel} \), in addition to the distance between them. Then, this distance is used by the PID controller to compute the velocity adjustment “adj” which is the adjustment the robot needs to make in order to maintain the predetermined distance.

Since the sensor gives the distance between the robots, the adjustment “adj” calculated by the PID controller is considered to be along this direction. Therefore the solution of the system of equations (3.6) is used to set the target velocity of the chasing robot wheels.

\[
\begin{align*}
\frac{v_l - v_r}{2} &= r \cdot \theta_{rel} \\
\frac{v_l + v_r}{2} &= \text{adj}
\end{align*}
\Rightarrow \begin{cases} 
  v_l = \theta_{rel} \cdot r + \text{adj} \\
  v_r = -\theta_{rel} \cdot r + \text{adj}
\end{cases} \tag{3.6}
\]

Using the pid controller and this method to set up the velocities of each wheel, the robot managed to follow and maintain the distance to the other robot successfully. The results of this simulation can be seen on figures 3.20, 3.21, 3.22 and 3.23.

\(^3\)Although a standard range finder doesn’t provide the relative angle, this could be calculated by attaching two range finders to the robot and finding the angle through parallax or by making a sweep, either by attaching the sensor to a rotating servo, or by changing the robot’s direction.
Chapter 4

Conclusion

This report covers the development of a framework, in Python, suitable for simulating the interaction between robots, along with actual test-cases that demonstrate its use and capabilities. During this project’s progress several obstacles were encountered, the most time-consuming of which was converging on a suitable architecture for the simulation framework.

Although the test-cases for this could have been implemented using one of the simulation platforms mentioned in section 1.2, it was decided that a generic problem-independent framework would be developed for this purpose.

Advantages of taking this approach included being able to easily extend the simulation environment with new functionality and modify or fine tune its behaviour, given that its inner workings were well understood. The downside of this decision was that a significant portion of time had to be allocated to developing this framework, along with the inevitable cost of debugging and maintaining the codebase.

Although this task was accomplished, the simulator still has room for improvement and presents an interesting challenge for future work.
Bibliography


