BreezySLAM: A Simple, efficient, cross-platform Python package for Simultaneous Localization and Mapping

Suraj Bajracharya

Professor Levy

CSCI 493

April 28, 2014

Abstract

BreezySLAM is a simple, efficient, multiplatform, open-source Python library for Simultaneous Localization and Mapping. By using Python C extensions to wrap existing implementations of existing SLAM algorithms, BreezySLAM provides a Python API for SLAM that runs nearly as fast as the original C code. By making a SLAM API available in Python, students and other interested users will be able to get their hands on SLAM very quickly and efficiently. BreezySLAM has been tested with a number of robots in V-REP Simulator, as well as on a Neato XV-11, with promising results.

Acknowledgements

This work was supported in part by a Commonwealth Research Commercialization Fund grant from the Center for Innovative Technology (CRCF #MF14F-011-MS).

Professor Levy and I would also like to thank Professor Lambert for letting us borrow the *Breezy* name.

Last but not the least, I would like to thank Washington and Lee University to have given me this opportunity to explore and work on SLAM as my Honors Thesis.

Table of Contents

Background

What is SLAM?

Simultaneous Localization and Mapping, or SLAM (Dissanayake et al. 2001) is an algorithm to map an enclosed environment, in which a robot must simultaneously learn the layout of the environment and determine its position in the environment. SLAM is especially useful in non-human reachable situations like search-and-rescue inside a collapsed building.

A map is used to plan an action or navigate an environment. However, a map being used at a time is not the always the same as the map when it was created. SLAM runs into this problem too. The initial map, m_1, would show what the robot sees when at its initial position, p_1. But when the position of the robot changes, lets say, to p_2, the map the robot sees now is different than what the previous map showed. It is actually the same environment, but not for the robot at its new position, p_2. To incorporate this new map, m_2, drawn by the robot in its new position with map m_1, one would need to keep track of the movement of the robot, hence, the localization.

Localization is the process of determining the position of a robot in a map. Needless to say, a map is needed for localization. Positions p_1 and p_2 needs to be placed on a map m to make sense of the positions. Without the map, the positions would merely be two points on an image.

One might think of SLAM as a chicken-and-egg problem. Which came first: the chicken or the egg? In SLAM, what should be implemented first: localization or

mapping? A good map is required for localization whereas an accurate position is required to build a map. Therefore, a simultaneous process of both localization and mapping is required.

Fig. 1(a): Localization of the robot at points p_1 and p_2 on a map, m

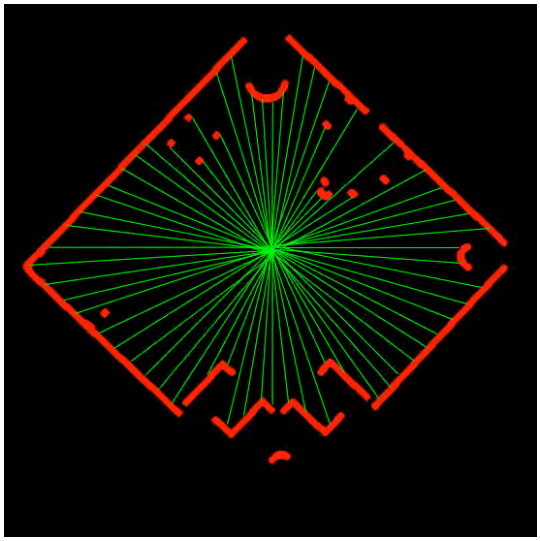


Fig. 1(b): A robot mapping its environment using a laser scan (green lines). This scan is more consistent with p_1 than with p_2. Note that the map here is less complete than the map in 1(a)

Figure 1(a) above shows localization of a robot. The robot is first shown at position p_1 at the center of a map, *m*, of a square room. The same picture also shows the same robot at position p_2 on the map.

Figure 1(b) above shows mapping. The map of the room shown in this picture is corresponding to position p_1 of Figure 1(a). Note that the laser scan lines are just an example and does not show the entire 360 degree scan.

In SLAM, we need to simultaneously localize a robot on a map of the environment, while mapping. Fortunately, we often have odometry (such as wheel rotation sensors) that would help us keep track of robot motions.

SLAM consists of multiple parts: data reading, data mapping, robot state, state update and map update. Different SLAM algorithms have different approaches to each of these parts. I would also like to note that SLAM itself is not an algorithm, but is, rather, an idea, which can be implemented in different approaches. SLAM also comes in 2D or 3D. We will be considering only 2D SLAM in this paper.

Probabilistic Robotics and Monte Carlo Localization

Probabilistic Robotics (Thrun et al. 2005) is a field of robotics that involves robots in environments subject to uncertainty. In SLAM, a robot does not know where exactly its position is in a map. By using recently developed algorithms, it is possible to keep track of possible positions and the probability of the robot being each of these positions. Probabilistic is used for localization of the robot in a map. A popular method for doing this is Monte Carlo localization (Doucet et al. 2001).

Monte Carlo localization is an algorithm that approximates a robot's position on a map using "particles", where each particle is a hypothesis for the position and bearing of the robot. It starts with assuming the robot can be anywhere. As the robot moves, the particles are shifted accordingly. These particles are also resampled every time the robot senses anything. At the end of the algorithm, the particles converge to the actual position of the robot.

2(a) 2(b) 2(c)

Fig. 2: Monte Carlo localization at work. In the 2(a), the robot assumes it can be anywhere possible in the map. The particles are resampled every time the robot moves, hence making a reasonable probability of being on each position. In 2(b), we see the particles converging to only a few different areas as compared to 2(a). These resampled particles still have a high probability of being the robot's actual position. As the algorithm progresses, the particles are converged to a single location as in figure 2(c), where the robot is now certain of its position on the map.

Kalman Filter

Another popular approach to localization is the Kalman Filter (Kalman 1960). It is an algorithm that uses a series of measurements over time containing noises and inaccuracies and produces estimates of unknown variables that are more precise than when based on single measurements.

In SLAM, the robot does not know its environment. But the robot has a laser sensor that maps the environment and also has odometry (e.g. wheel-rotation sensors) describing how it moves in the environment. These data are generated continuously. Using Kalman Filter with the map and motion inputs as input variables, the best estimate of the position and bearing of the robot is known.

Being a recursive process, Kalman Filter takes in the most recent map as an input variable, updates the map, and uses this new map as an input variable for the next iteration. It also takes in the most recent state of the robot (position and bearing),

updates the state using the given motion inputs, and gives out the best estimate of the new state (position on the new map and the bearing). Kalman Filter performs this process repeatedly every second, with different positions of the robot, and hence, drawing out different variations of the map for every iteration.

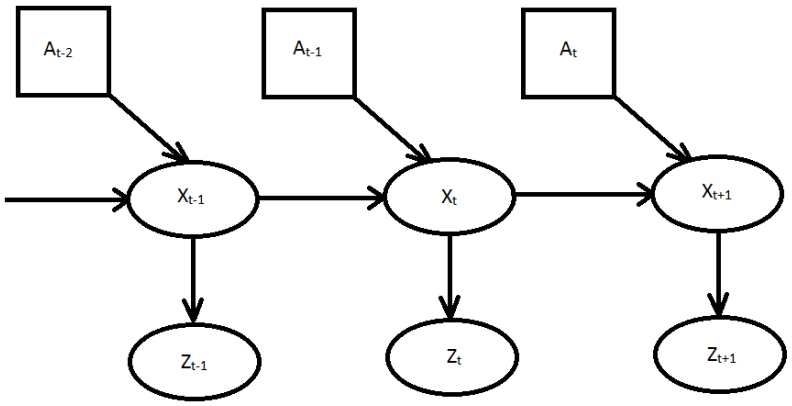


Fig. 3: A Dynamic Bayes net (Russell and Norvig 2009, Fig 25.5)

Figure 3 above shows a Dynamic Bayes net (Murphy 2002), a recently developed generalization of the Kalman filter. At time *t,* the current state, X_t of the robot is determined by the action, A_{t-1}, and the state, X_{t-1}, at time *t-1*, to give an observation Z_t. It then uses the current state and action to determine the likeliest next state and observation.

Choosing a SLAM Algorithm

The main constraints in choosing a SLAM algorithm to implement were:

(1) The need to be able to run code on a System-on-a-Chip (SoC) / Computer-On-Module (COM), supporting computation onboard the robot's own

hardware. This is important in environments where communication between the robot and a remote computer is unavailable or actively jammed as in our grant-funded work on miniature aerial vehicles with Advanced Aerials, Inc.

(2) The desire to program in Python.

Why Python?

Python, conceived in early 1980's, is one of the most powerful, widely used high-level programming languages today (Venners 2003). It highly focuses on readability. By using Python C extensions to wrap code around C, code speed can also be made up to par with native C implementations. We will be seeing more of this later in the paper.

Making SLAM available in Python has two major advantages. First, it makes SLAM available to a broad variety of researchers, students, educators, and hobbyists. Second, Python can easily be run Computer-on-Module (COM) platforms like Raspberry Pi and Gumstix Overo.

Previous Work

V-REP

Virtual Robot Experimentation Platform (V-REP) is a robot simulator developed by Coppelia Robotics, Switzerland (http://www.coppeliarobotics.com/). It is a powerful, open-source simulator, provided free of cost for educational use. It has many built-in robots and additional tools such as different types of sensors and configurable

environments, and it also allows the user to build their own robot or tools.

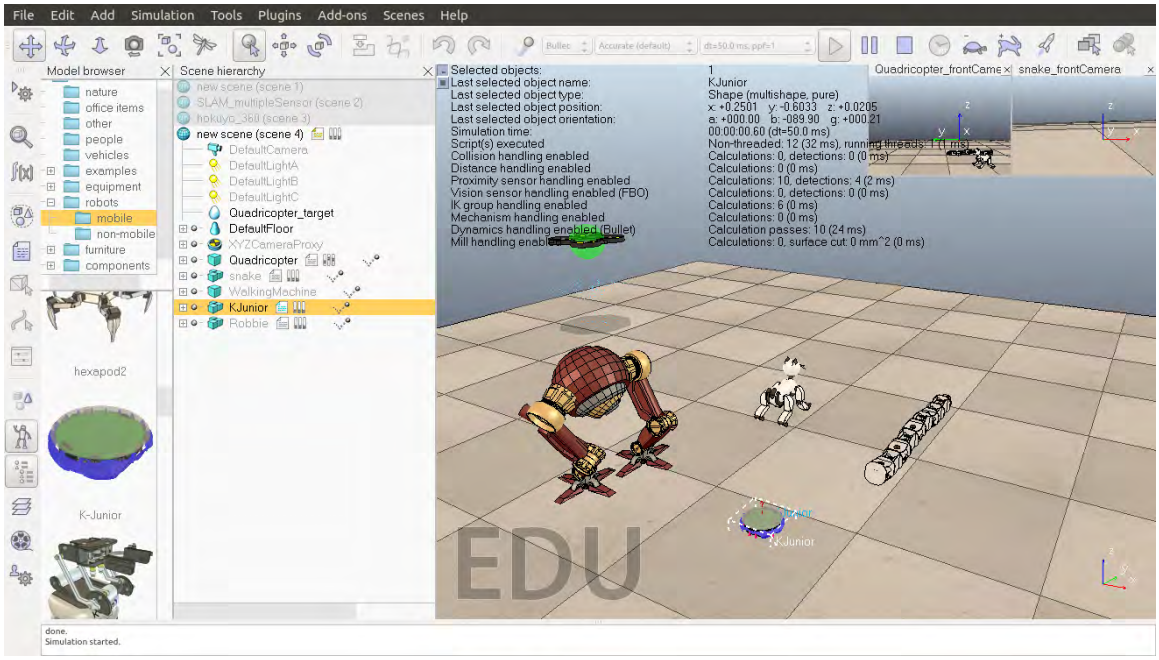


Fig. 4: V-REP, with different available built-in robots in a default environment

In winter of 2013, I worked on an independent study project with Professor Levy, where I first found out about SLAM. I used V-REP for the project. In the project, I worked on building a virtual robot on the simulator that moves around and maps out a virtual room (environment) I built into it. However, most of this time was spent understanding V-REP and the way it works. Codes were borrowed from existing robot models in the software and were written in Lua, V-REP's native language. During this project, we also did figure out V-REP's remote API, that allows the simulator to run on different other languages. Python is one of the many different languages that can be used for prototyping algorithms in V-REP. This also provided a test platform for BreezySLAM.

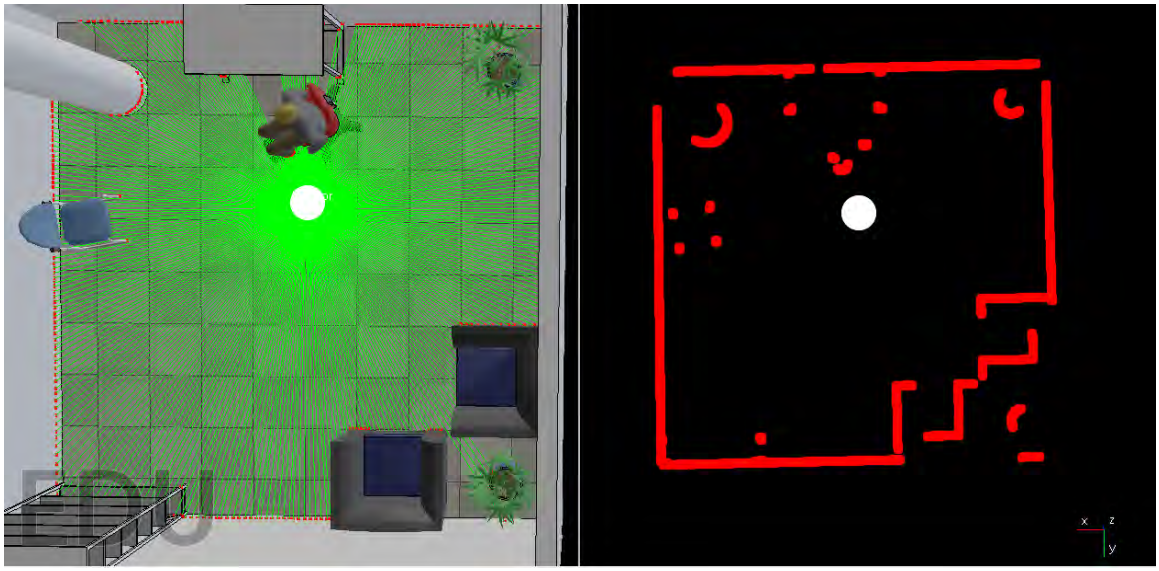


Fig. 5: V-REP performing SLAM using a K-Junior robot

The left figure above shows a K-Junior robot in an enclosed simulation environment. The K-Junior is equipped with a 360 degrees laser scanner (more on this later). The figure on the right is the mapping performed by the K-Junior as it moves around in the simulation environment.

K-Junior Model

First released to the public in early 2010, V-REP proved to be a basis for my initial independent study project. “The Swiss army knife among robot simulators”, V-REP had a built-in robot model for K-Junior, which I believed was a very good robot to perform SLAM with. So I built a 360 degrees laser sensor on top of the built-in K-Junior model. Adding extra weight to the light robot with the LIDAR sensor caused the robot to initially topple over. Realizing that V-REP takes physics into account very seriously, I compared the K-Junior to the robots talked about in the literature paper.

I realized that these were much heavier and bigger in dimension than K-Junior. If I were to build one of these robot's model in V-REP from scratch, the same laser model I used on K-Junior would not be a problem if used on the Neato model because its weight would be negligible as compared to the base it is attached to, the robot. However, building a Neato model in V-REP from scratch would not have been ideal since we had to focus on SLAM. So, as a workaround, I made the weight of the laser components on top of the K-Junior model negligible in weight (0.0001 kg). This allowed the K-Junior model to function normally as it would without any components on top of it, but still have a laser sensor attached to it to perform scans for our SLAM.

Fig. 6(a): K-Junior model on V-REP

Fig. 6(b): My K-Junior model with a laser component

Remote API

V-REP also has six different programming approaches, including ROS (Robot OS) API and Remote API. ROS is a very good platform for robotics. In fact, ROS even has a laser-based SLAM node. However, ROS is not supported (at least not fully) on any other OS than Ubuntu. Since the goal of this project is to make SLAM available to anyone

interested, it would have to be supported on multiple platforms. So we chose to use V-REP's Remote API in BreezySLAM.

Remote API allows users to use their code written in different languages to work as either a server or a client for the simulation. Using V-REP's remote API, I was able to use the Python-using BreezySLAM to act as a client that continuously receives raw scan data from the simulator (server), performs the necessary SLAM implementations, and sends robot movements back to V-REP so that the robot could move on the simulator.

Server side code (written in Lua):

```
simSetStringSignal("points", simPackFloats(points))
```

Client side code (written in Python):

```
clientID = vrep.simxStart(address, port, True, True, /
          5000, 1)

vrep.simxStartSimulation(clientID, /
                         vrep.simx_opmode_oneshot_wait)

errorCode, signalVal = vrep.simxGetStringSignal \
                         (clientID, 'points', \
                         vrep.simx_opmode_oneshot_wait)

if errorCode == vrep.simx_error_noerror:
     points = vrep.simxUnpackFloats(signalVal)
…
…
…
vrep.simxStopSimulation(clientID, /
                         vrep.simx_opmode_oneshot_wait)
```

As you can see in the code snippets above, it takes only one line of Lua code for the server to send laser scan values from V-REP to the remote client. In the first snippet, the 360 different points the laser scanner detects are packed as floats and sent to the

client.

The second snippet, which is written entirely in Python, is the client code. The first line connects the client code to the V-REP server using the IP address and the port the server is located on. This returns a client ID, which is used in every V-REP method used thereafter in the client code, including starting the simulation (line 2), getting scan values from the server (line 3), or stopping the simulation (line 9). Unpacking the float points from the packed data sent by the server is performed with a single line of code in line 5.

SLAM Algorithm

Since various approaches to SLAM are already available, Professor Levy and I decided we would just work on a SLAM algorithm that is already written, but create a Python API for the implementation. Naturally, we searched for SLAM written in Python. To our surprise, we did not find any SLAM code in Python.

We came across OpenSLAM.org, where different authors have posted different implementations of SLAM in different programming languages, none of them being in Python. Many of them were also just demos of what the authors had done in their research, and none of them had a simple, intuitive API. So our first goal became writing an API to support on a simulator or an actual robot.

I specifically looked into two different SLAM algorithms from OpenSLAM.org: DP-SLAM (Eliazar and Parr 2003), which seems to be most popular in SLAM and TinySLAM

(a.k.a. CoreSLAM) (Steux and Hamzaoui 2010).

DP-SLAM

DP-SLAM, developed by Austin Eliazar and Ronald Parr from Duke University in 2003, uses particle filter to maintain a joint probability distribution over maps and robot positions. They used iRobot ATRV Jr. as their vehicle to perform SLAM.

DP-SLAM was the first SLAM algorithm I looked into. While it did have good C code, the README file stated that it "requires a LOT of memory – on the order of several GB." The reason DP-SLAM requires such a huge amount of memory is that it is a full version of SLAM: every hypothesis is a position and an orientation of the robot along with a map corresponding to that position. A naïve version of this scheme would mean multiple possible positions would have their own possible maps, with the memory storage taking number of position hypotheses times the number of points on a map. Although the DP-SLAM algorithm uses an innovative sharing mechanism to minimize the duplication of maps, the amount of memory required is still impractical for our constraints to fit on a SoC.

Fig. 7: iRobot ATRV Jr. (Source: DP-SLAM, http://www.cs.duke.edu/~parr/dpslam/)

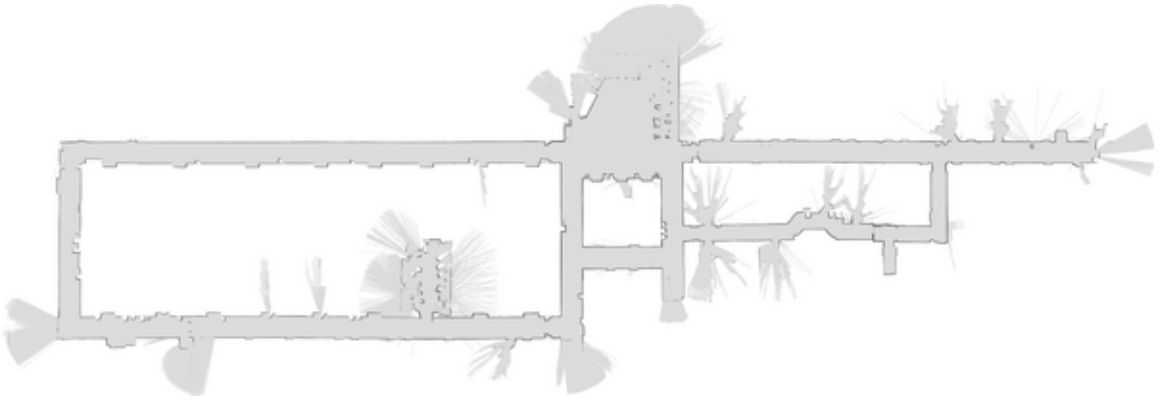


Fig. 8: DP-SLAM Map (Source: DP-SLAM, http://www.cs.duke.edu/~parr/dpslam/)
Note the high degree of accuracy, even in a long, narrow corridor

TinySLAM (CoreSLAM)

Putting DP-SLAM aside, I looked into TinySLAM, developed by Bruno Steux and Oussama El Hamzaoui in 2010. TinySLAM, as the name implies, has very little code. In fact, the algorithm itself is around 200 lines of C. Additional algorithmic and SIMD architecture optimizations were also possible (Hamzaoui and Steux 2010). Thus, we

decided to translate CoreSLAM into Python.

Unlike DP-SLAM, CoreSLAM stores a single position and a single map at a time. This makes CoreSLAM need much less memory than DP-SLAM, making it suitable for our needs. Using the most recent map and the new odometry, CoreSLAM guesses the best new position for the robot and updates the map accordingly.

The developers used a MinesRover as their robot. It is a homebrew, six-wheeled robot developed by Mines ParisTech and SAGEM DS, equipped with a Hokuyo URG04 laser scanner (Steux and Hamzaoui 2010).

Fig. 9: A MinesRover Robot

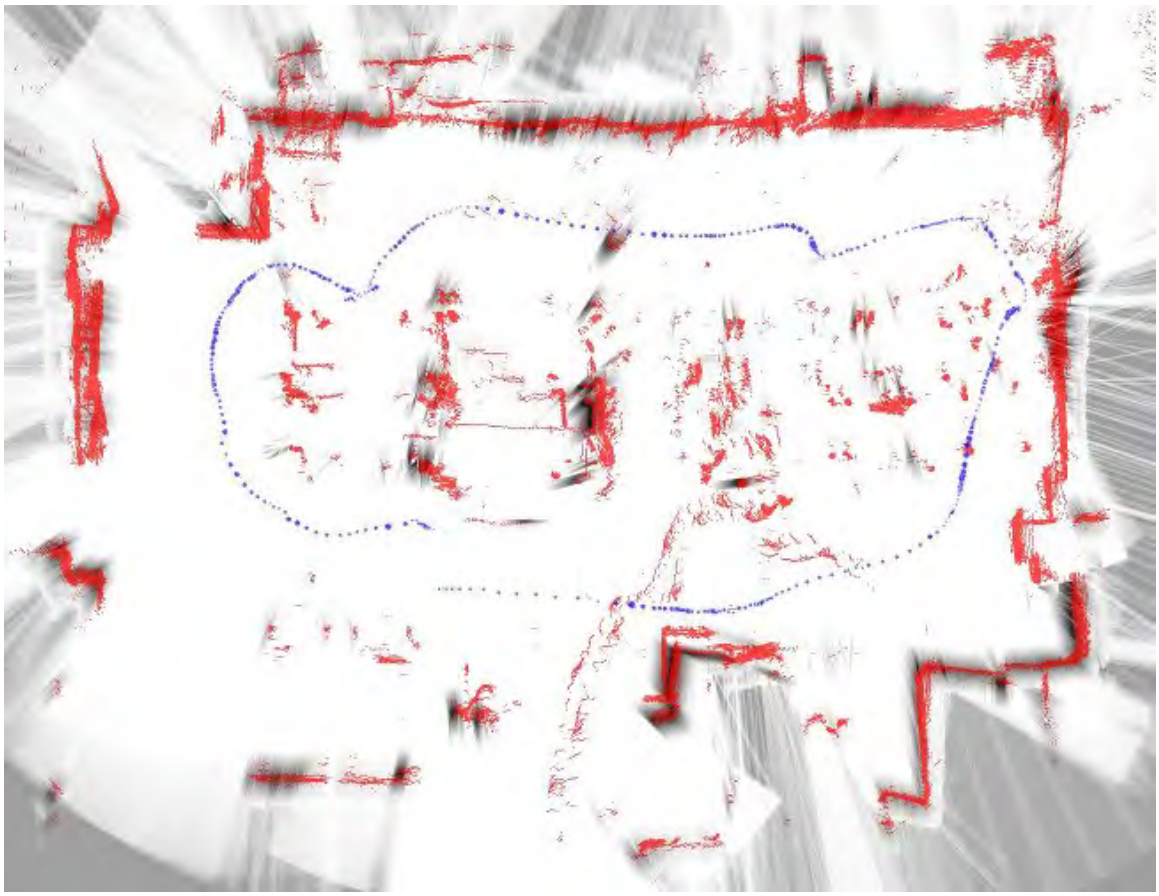


Fig. 10: TinySLAM Map (Source: TinySLAM, https://openslam.org/tinyslam.html)

In the above map, white color represents the areas the Hokuyo scanner has scanned and found to be free of obstacles. Red color represents the obstacles in the environment, and blue color represents trajectory for the moving MinesRover.

Python Translation for CoreSLAM

Instead of using the Simplified Wrapper and Interface Generator (SWIG) (http://www.swig.org/), Professor Levy translated CoreSLAM into Python line-by-line, so that we could thoroughly understand the details of the algorithm. With lack of

comments in the code, terse description in the papers, and some mistranslations of terminology, the papers and the code were a challenge to understand. So, even translating the code line-by-line was not fully helpful, though it did help us understand most of CoreSLAM. This approach also helped us in factoring various parts of the code in our translated version. In addition, the random number generator in CoreSLAM assumed only a 32-bit architecture and would not work on a 64-bit architecture.

BreezySLAM provides the pure Python classes: Robot (robot odometry-to-velocity method), Laser (laser parameters), and Odometry (measured at each instant). These are the classes that users will want to modify. The rest of the code is implemented as C extensions. As shown in the following code fragment, the API is extremely simple: a constructor that accepts Laser parameters and the size, in pixels, and scale of the map, in pixels per meter; an update method that takes the current scan and returns the new robot position and angular rotation (theta); and a method for retrieving the current map as a byte array.

```
laser = laserparams()
mapbytes = bytearray(800*800)
slam = CoreSLAM(laser, 800, 0.1)
while True:
    scan = readLaser()
    x, y, theta = slam.update(scan)
    slam.getmap(mapbytes)
```

If odometry is available, it can also be passed into the update method for improved accuracy. Bytearray for the map is pre-allocated before instantiating CoreSLAM.

Since Python is primarily interpreted rather than compiled, Python is bound to

run slower than C – especially for loop-intensive algorithms like SLAM. Testing Professor Levy's pure Python implementation of CoreSLAM against the original C version, we found that processing 100 scans took 50 seconds in pure Python and only 0.8 seconds in C (2.7 GHz Intel Core Duo iMac, OS 10.8.5 running Ubuntu 13.10 in VMware 6.0; five trials, $p < .000001$). In other words, the pure Python implementation was over 60 times slower than the original C implementation.

Consistent with (Steux and Hamzaoui 2010), profiling the Python code revealed that the calls taking the most time were `distanceScanToMap` based on the current laser scan and map (used for determining the likelihood of a given robot position) and `mapLaserRay` (for integrating a new scan into the map). So, Professor Levy wrote C extensions for these two functions. This change provided a dramatic speedup, but still made Python take around 2.25 the runtime of the original C code. Further profiling revealed that most of the runtime was now being taken by the Monte-Carlo particle filter (Random-mutation Hill-climbing). Hence, Professor Levy translated this code into a C extension as well. This resulted in our Python code taking only 25% longer than the original C – i.e., much faster than real-time (10 Hz scan rate of data acquisition).

Finally, we attempted to replicate the favorable results that the CoreSLAM authors presented when comparing pure Monte Carlo scan-matching localization to scan matching plus odometry. Our initial failure to replicate these results turned out to have been caused by the original pseudo-random number generator, which assumed a 32-bit architecture. Professor Levy re-implemented this pseudo-random number generator (Marsaglia and Tsang 2000) to run on both 32- and 64-bit architectures, and

successfully reproduced the results from (Steux and Hamzaoui 2010).

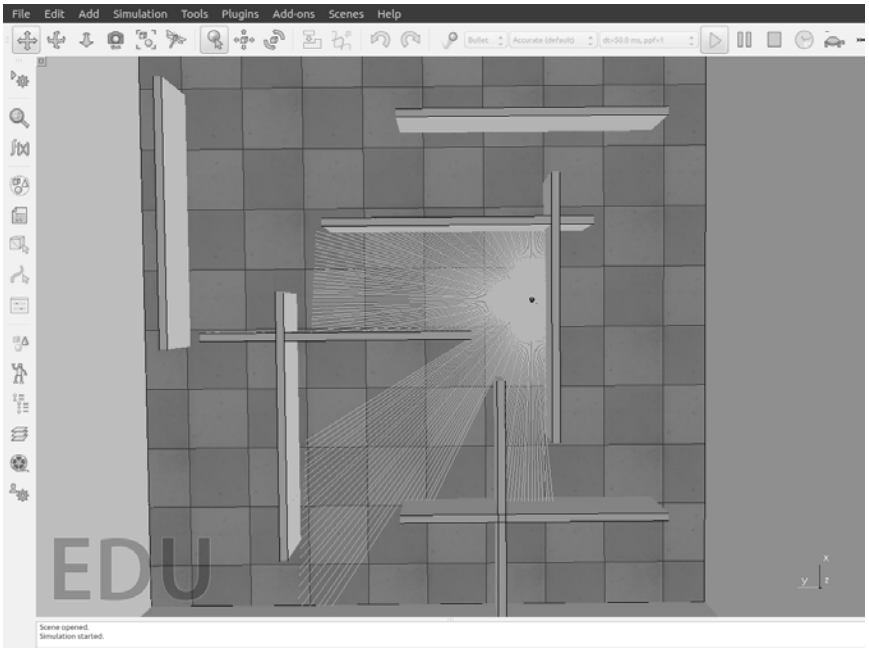


Fig. 11(a): V-REP with a K-Junior robot scanning the environment

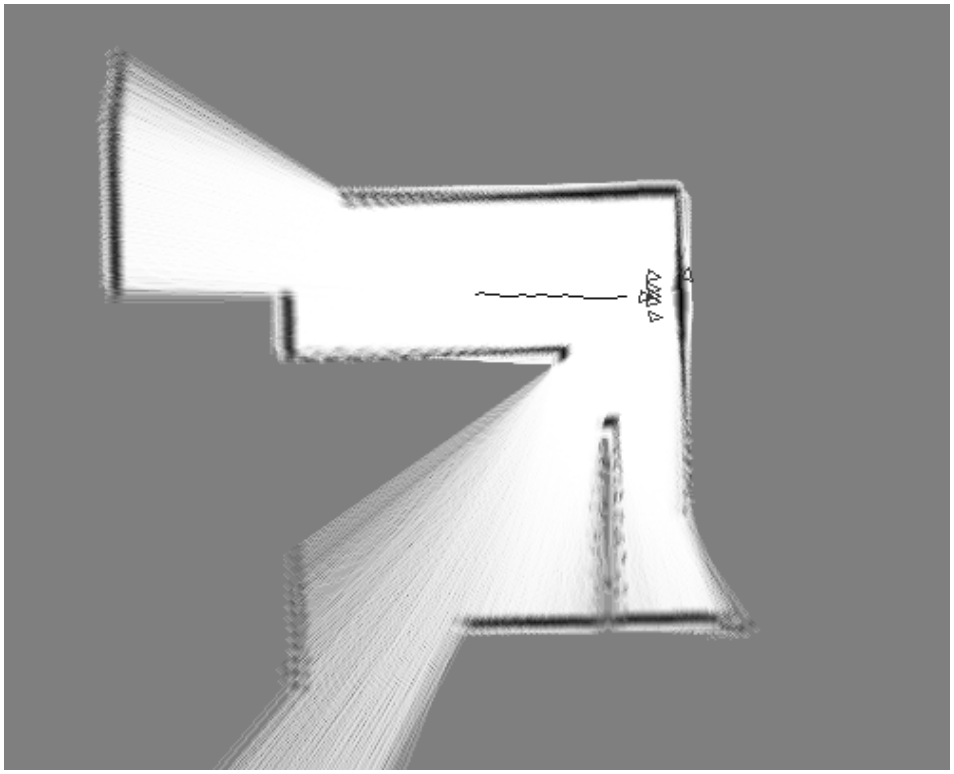


Fig. 11(b): A sample mapping obtained using BreezySLAM

Figure 11 shows sample results with a simulated wheeled robot using the popular V-REP simulator from Coppelia Robotics. The plotted trajectory shows that the robot had traveled a relatively short distance at the point at which the screenshot was taken. The map also shows that our CoreSLAM implementation supports retrieval of the current point-cloud (set of hypothesis about robot position and orientation) obtained through Monte Carlo localization.

Odometry

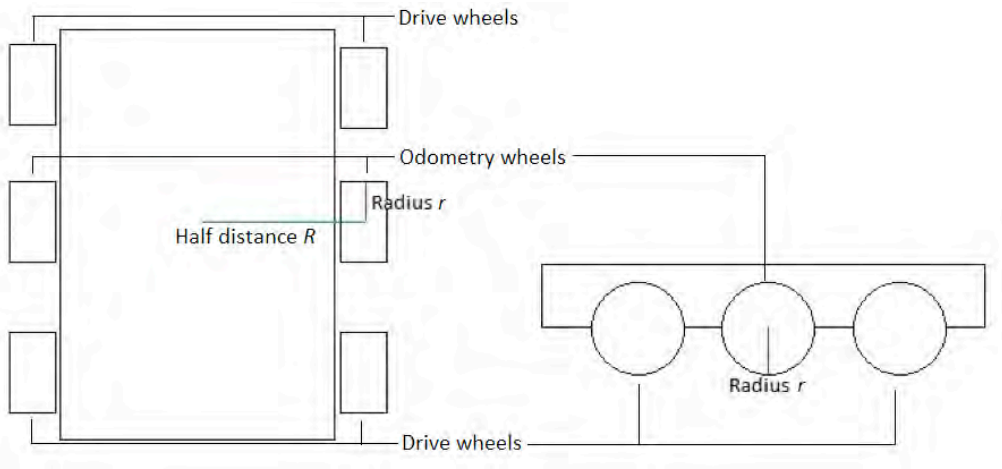


Fig. 12(a): MinesRover top-view (left) and side-view (right)

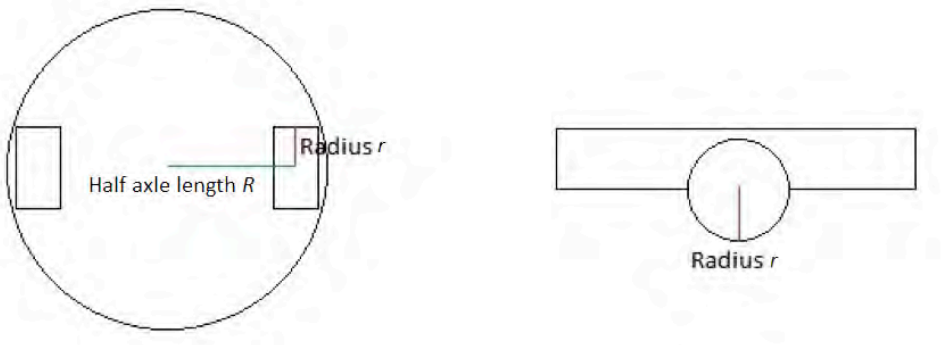


Fig. 12(b): K-Junior top-view (left) and side-view (right)

Figure 12(a) is a model of ParisTech's MinesRover robot, a schematic for the MinesRover robot in Figure 9. It has six identical wheels, each with radius *r*. Each wheel is equidistant from its counterpart wheel in the other side of the robot, with the distance between them, or the axle length, being *R*. The same goes for K-Junior, as shown in Figure 12(b): *R* is half the distance between its two wheels, whereas *r* is the radius of each of the identical wheels.

```
self.r = 0.077           # wheel radius
self.R = 0.165           # half axis length
self.inc = 2000          # counter inc per turn
self.ratio = 1.0         # left/right wheel ratio

self.m = self.r * pi / self.inc

def computeVelocities(self, odometry_prev, odometry_curr):

    dxy = self.m * \
         (odometry_curr.q1 – odometry_prev.q1 + \
         (odometry_curr.q2 – odometry_prev.q2) \
         * self.ratio)

    dtheta = (self.m * \
         ((odometry_curr.q2 – odometry_prev.q2) * \
              self.ratio – odometry_curr.q1 + \
              odometry_prev.q1) / self.R)
```

The above code snippet is the direct Python translation of a part of CoreSLAM. This particular snippet computes the forward velocity *dxy* and angular velocity *dtheta* of the MinesRover given the odometries of the wheels. These velocities can then be used to help estimate the robot's new position based on the time between measurements. It took me a long time to understand the above code. First of all, what does the big *R* stand for? What is "half axis length"? Then, what is *inc*? Do we really need the ratio? What is *m*?

After playing around with the values and thinking for a long time, Professor Levy figured out that the authors actually meant 'axle' rather than 'axis', thus the big *R* representing half the distance between the wheels. This is important because the wheels might be moving at different speed at a given time, but the distance between the wheels is the same, thus turning the robot. *Inc* refers to the number of ticks a wheel makes in one complete rotation, and *m* represents the distance the wheel moves per click. The values for the variables *r*, *R*, *inc*, and *ratio* above are for the MinesRover robot.

The `computeVelocities` method takes in the current odometry and the most recent previous odometry, and gives out the distance travelled by the MinesRover (dxy) and the angle it turns by (dtheta). *q1* and *q2* refers to the left and the right wheels.

I took the above code written for MinesRover, and wrote very similar code for K-Junior:

```
self.wheelRadiusMeters = 0.01525
self.halfAxleLengthMeters = 0.04

def computeVelocities(self, odometry_prev, odometry_curr):

     dxy = (odometry_curr.q1 – odometry_prev.q1 + \
          odometry_curr.q2 – odometry_prev.q2)

     dtheta = ((odometry_prev.q2 – odometry_curr.q2) - \
          (odometry_prev.q1 – odometry_curr.q1)) /     \
          self.halfAxleLengthMeters
```

Note that I eliminated *inc* as the wheels on K-Junior did not have clicks on them. Naturally, *m* is also eliminated. I also eliminated *ratio* because the robots we are testing on always have the same right and left wheel radii.

After making the changes above, I came to the remaining pieces of the puzzle. If

we look back at the code for how *m* was assigned, we see:

```
self.m = self.r * pi / self.inc
```

If *m* were to be the distance a wheel moves every time it makes a click, it would rather be the circumference of the wheel divided by the number of ticks on the wheel. But we do not see the circumference being calculated in the above snippet. Instead, we see half the circumference being calculated. Basically, the radius is supposed to be doubled to make sense out of the above snippet, without confusing the code reader:

```
self.m = 2 * self.r * pi / self.inc
```

With this, the calculation of dxy and dtheta would then be incorrect with the above formulas. Hence, computeVelocities turned out to be as follows, without using the variable *m*:

```
self.wheelRadiusMeters = 0.01525
self.halfAxleLengthMeters = 0.04

def computeVelocities(self, odometry_prev, odometry_curr):

    dxy = (odometry_curr.q1 – odometry_prev.q1 + \
         odometry_curr.q2 – odometry_prev.q2) / 2

    dtheta = ((odometry_prev.q2 – odometry_curr.q2) - \
         (odometry_prev.q1 – odometry_curr.q1)) /     \
         (2 * self.halfAxleLengthMeters)
```

Basically, dxy, the distance the robot travels in a second, is assigned to be the average of the distance covered by each of the wheels. Similarly, the angle the robot turns per second is calculated. As the wheels are at a fixed distance from each other at any given point, the angle the robot turns per time interval would be the distance

covered by a wheel minus the distance covered by the other wheel divided by the distance between the two wheels.

Using the above model, I created a `WheeledRobot()` class, which takes in the radius of the wheel and the half axle length of the robot a user is using, and calculates dxy and dtheta with the above formulas:

```
class WheeledRobot(object):
  '''
   An abstract class supporting ododmetry for wheeled robots. Your
   implementing class should provide the method:
     extractOdometry(self, timestamp, leftWheel, rightWheel) -->
       (timestampSeconds, leftWheelDegrees, rightWheelDegrees)
  '''

   def __init__(self, wheelRadiusMeters, \
     halfAxleLengthMeters):
      '''
       wheelRadiusMeters - radius of each odometry wheel, in
                           meters
       halfAxleLengthMeters - half the length of the axle between
                              the odometry wheels, in meters
      '''

     self.wheelRadiusMeters = wheelRadiusMeters
     self.halfAxleLengthMeters = halfAxleLengthMeters

     self.timestampSecondsPrev = None
     self.leftWheelDegreesPrev = None
     self.rightWheelDegreesPrev = None
```

(*Continues on next page...*)

```
  def computeVelocities(self, timestamp, \
    leftWheelOdometry, rightWheelOdometry):
     '''
      Computes forward and angular velocities based on odometry.

      Parameters:
        timestamp - time stamp, in whatever units your robot uses
        leftWheelOdometry - odometry for left wheel, in whatever
                            units your robot uses
        rightWheelOdometry - odometry for right wheel, in
                             whatever units your robot uses

      Returns a tuple (dxyMeters, dthetaDegrees, dtSeconds):
        dxyMeters - forward distance traveled, in meters
        dthetaDegrees - change in angular position, in degrees
        dtSeconds - elapsed time since previous odometry, in
                    seconds
      '''

    dxyMeters = 0
    dthetaDegrees = 0
    dtSeconds = 0

    timestampSecondsCurr, leftWheelDegreesCurr,     \
      rightWheelDegreesCurr = self.extractOdometry( \
      timestamp, leftWheelOdometry, rightWheelOdometry)

    if self.timestampSecondsPrev != None:
      leftDiffDegrees = leftWheelDegreesCurr -   \
        self.leftWheelDegreesPrev
      rightDiffDegrees = rightWheelDegreesCurr - \
        self.rightWheelDegreesPrev

      dxyMeters = (math.radians(leftDiffDegrees) + \
        math.radians(rightDiffDegrees)) / 2
      dthetaDegrees = (rightDiffDegrees - \
        leftDiffDegrees) / (2 * self.halfAxleLengthMeters)
      dtSeconds = timestampSecondsCurr - \
        self.timestampSecondsPrev

    # Store current odometry for next time
    self.timestampSecondsPrev = timestampSecondsCurr
    self.leftWheelDegreesPrev = leftWheelDegreesCurr
    self.rightWheelDegreesPrev = rightWheelDegreesCurr

    # Return linear velocity, angular velocity, time difference
    return dxyMeters, dthetaDegrees, dtSeconds
```

Using a generic `WheeledRobot()` class to calculate dxy and dtheta, we can now create a model for any robot we are using by just passing in the radius of the wheels and half the distance between the wheels (half of axle length), both in meters. Thus, the K-Junior model is coded as:

```
class KJunior(WheeledRobot):
     def __init__(self):
          WheeledRobot.__init__(self, 0.01525, 0.04)
```

Similarly, a model for any other two-wheeled robot can be created easily just by changing the values for the variables representing the radius and the half axle length. For example, the model for Neato XV-11, the robot I will be using for real-time, real-environment simulation, would be as follows:

```
class XV11(WheeledRobot):
     def __init__(self):
          WheeledRobot.__init__(self, 0.047746,0.1425)
```

Odometry using K-Junior

The above formulas were used to keep track of K-Junior's odometry as it maps an environment and moves within it. However, the map was not correct. Seeing nothing wrong with the algorithm, Professor Levy and I tried to figure out what might be causing this problem. We soon found out that because K-Junior is a very small robot, and its wheels further smaller, the wheels were rotating much faster than the sampling rate, thus causing aliasing. The scans were being under sampled, which made the K-Junior appear to move backwards, thus giving an incorrect map.

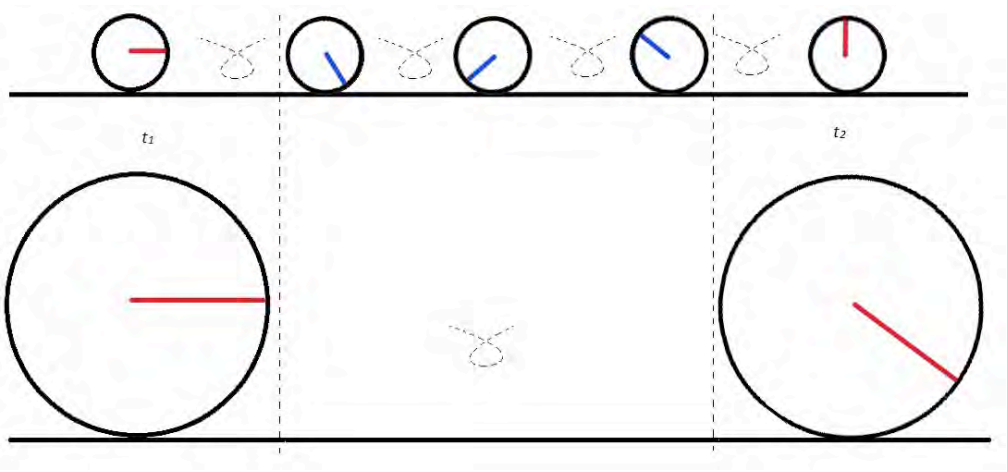


Fig. 13: Smaller wheel rotates multiple times over a short period of time causing aliasing. Bigger wheels do not rotate as many times within that same amount of time.

Let's say we have two different wheels, each of a different radius. The first wheel is small, with a circumference of only 1 cm, whereas the other wheel is 5 cm in circumference. When both the wheels are moving at the same velocity, let's say 5 cm per second, the smaller wheel makes almost a turn in a second whereas the bigger wheel turns very slightly. Figure 13 above describes the circular motion of these two wheels. Note that the smaller wheel almost makes a turn. However, since sampling takes place only at t_1 *and* $t2$, the first wheel's entire turn is not recorded, but only its orientation at time t_2, as shown above. So, it would look like the wheel moved a quarter turn backward rather than three quarters forward.

As I was working on a different simulated robot (dr-20) in V-REP, we decided we would rather return to working on a physical robot rather than the simulator, as we started running out of time. Though, I am certain that performing odometry based SLAM on a larger robot on V-REP would not be a problem. Thus, we started transporting our code to the robot, the XV-11.

Neato XV-11

Hardware

Fig. 14: Neato XV-11

Neato XV-11 is a robotic vacuum cleaner built by Neato Robotics (http://www.neatorobotics.com) with a 360 degrees LIDAR scanner attached to it (the bulge on the top right on the picture above). It has a wheel attached to either side of it, which are capable of rotating backward, allowing the Neato to move in both forward and backward directions. Moving one of the wheels in a slower speed than the other performs rotation. The manufacturer provides an API allowing a programmer to send commands to the XV-11 and receive data from it over the USB port.

NeatoPylot

NeatoPylot (http://home.wlu.edu/~levys/software/neatopylot/) is a XV-11 AutoPylot program written by Professor Levy that allows the user to drive the Neato

using a joystick. NeatoPylot also provides a server for LIDAR data for XV-11 and can be run on a Raspberry Pi or other computer-on-module mounted on the XV-11 (shown below). A client computer connects to the Pi using a Wi-Fi connection.

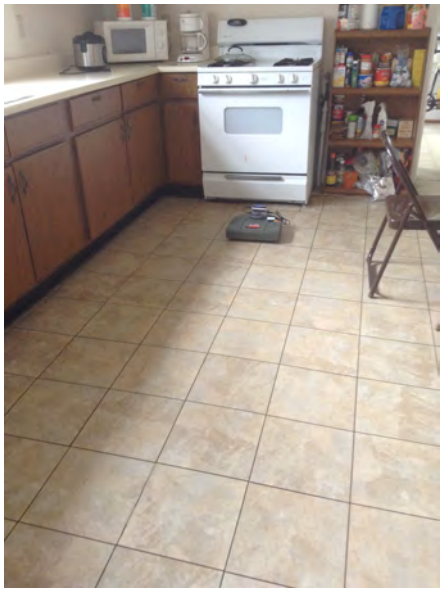
Fig. 15: Neato XV-11 with a Raspberry Pi (in the transparent box on top of the LIDAR component) and a battery pack (black device on top of the Pi). Wi-Fi dongle is on the left side, above the USB cable, connecting the Pi to the XV-11.

Neato-SLAM

As I suspected earlier with K-Junior's wheels being too small to help with the odometry, the XV-11 did, in fact, return fairly accurate values. The figures below show the result of running BreezySLAM on the XV-11:

Fig. 16(a): Neato XV-11 performing SLAM, starting position, p_1

Fig. 16(b): Neato XV-11 performing SLAM, position p_2, straight ahead of p_1



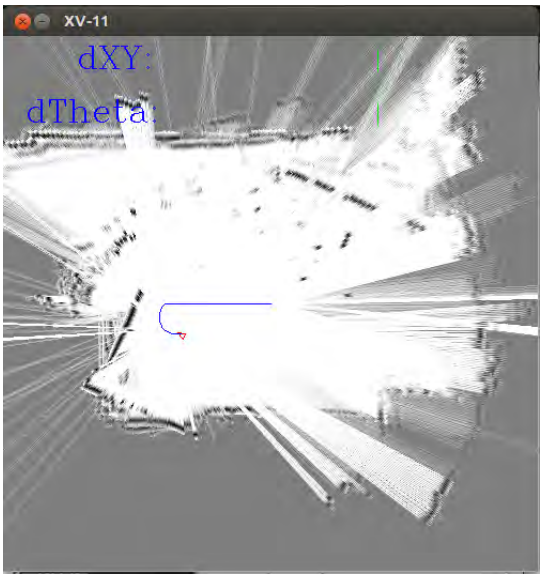
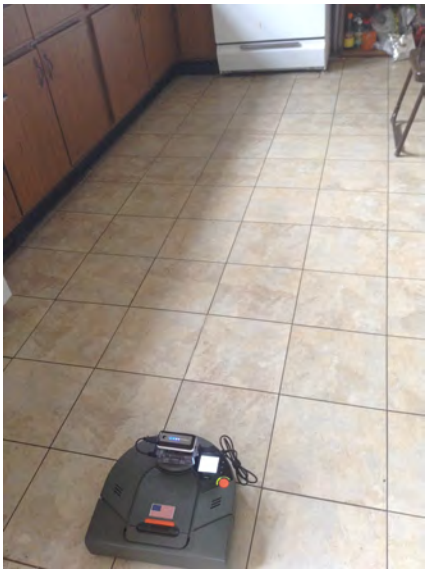


Fig. 16(c): Neato XV-11 performing SLAM, position p_3, rotated 180 degrees from p_2. Errors can be seen on this map (tilted map)



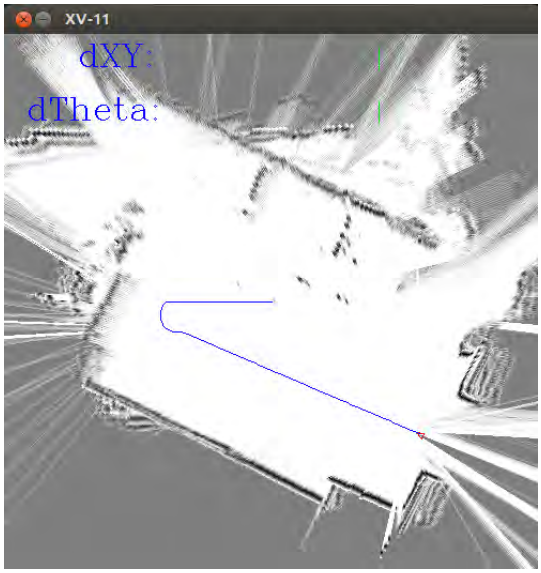


Fig. 16(d): Neato XV-11 performing SLAM, end position, p_4, straight ahead from p_3. Note the sharp trajectory (blue line) and sharp edges of the map (black lines).

Conclusion

Professor Levy and I have been able to successfully make BreezySLAM run with a simulator, V-REP and a real physical robot, the XV-11. With Professor Levy's translation of CoreSLAM into Python and my testing of his code on various platforms, we have been able to create a map that is fairly accurate, as you can see in Figure 16.

Currently, BreezySLAM is up online in the following link:

http://home.wlu.edu/~levys/ software/breezyslam/

As we hoped earlier, BreezySLAM has been made available to the public now. In fact, we already have at least two people using BreezySLAM as two people have contacted us regarding it.

Next Steps

The next step for anyone working on BreezySLAM will be to create wrappers for Java and Matlab, so that an even larger group of people will have access to BreezySLAM without having to go through the hassle of writing their own SLAM code or having to translate an existing algorithm into a different language. Professor Levy has already written the C++ version for BreezySLAM. Currently, he is also factoring out Scan and Map classes to be available as Python C classes rather than just the top-level algorithms.

We also hope to allow users to implement their own SLAM algorithm using BreezySLAM.

References

Dissanayake, G.; Newman, P.; Clark, S.; Durrant-Whyte, H.; and Csorba, M. A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions of Robotics and Automation* 17(3): 229–241, 2001.

Doucet, A., de Freitas, J.F.G., Gordon, N.J. *Sequential Monte Carlo Methods In Practice*. Springer Verlag, New York, 2001.

Eliazar, A., and Parr, R. Dp-slam: Fast, robust simultaneous localization and mapping without predetermined landmarks. In *in Proc. 18th Int. Joint Conf. on Artificial Intelligence (IJCAI-03)*, 1135–1142. Morgan Kaufmann, 2003.

Hamzaoui, O. E., and Steux, B. A fast scan matching for grid-based laser slam using streaming simd extensions. In *ICARCV*, 1986–1990. IEEE, 2010.

Hamzaoui, O.E., and Steux, B. Slam algorithm with parallel localization loops: Tinyslam 1.1. In *Automation and Logistics (ICAL), 2011 IEEE International Conference on*, 137–142, 2011.

Kalman, R. E., A New Approach to Linear Filtering and Prediction Problems. In *Journal of Basic Engineering* 82 (1): 35–45. doi:10.1115/1.3662552, 1960.

Marsaglia, G., and Tsang, W. The ziggurat method for generating random variables. *Journal of Statistical Software* 5(8): 1–7, 2000.

Murphy, K.P. *Dynamic bayesian networks: representation, inference and learning*. PhD thesis, UC Berkeley, Computer Science Division, July 2002.

Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach, Third Edition.* Prentice Hall, New Jersey, 2009.

Steux, B., and Hamzaoui, O. E. tinyslam: A slam algorithm in less than 200 lines c-language program. In *ICARCV*, 1975–1979, 2010.

Thrun, S., Fox, D., Burgard, W. *Probabilistic Robotics.* MIT Press, Cambridge, Massachussettes, 2005.

Venners, B. *The Making of Python*. Artima Developer, Artima, 2007.