Robotic Navigation with RFID Waypoints

Chris Tralie, REU Fellow

Princeton University, Computer Science

Adviser: Dr. Matthew Reynolds

Duke University, Electrical and Computer Engineering

Abstract

The purpose of this project was to explore occupancy grid construction and RFID heatmap generation of an unknown environment on a small, affordable robot with a laser scanner and RFID reader. The robot platform consisted of an “iRobot Create” robot with an attached netbook and Hokuyo Urglaser range scanner, along with a ThingMagic Mercury® 5e RFID reader and a webcam. PlayerStage, an open source robotics development environment, was used to communicate with the robot hardware and to run simulations. Then, a utility called “pmaptest” used SLAM (Simultaneous Localization and Mapping) techniques to build occupancy grids of the hallways in the Duke engineering building from logged laser scan and odometry readings. Finally, a driver was written in C to communicate with the RFID reader, and RFID tags were dispersed throughout the hallway. The robot was driven through the hallway with a program created to steer the robot towards the centroid of open space (as determined by the laser range scanner), and data was logged. The information from the laser and odometry logs was used to build the occupancy grid, and the information from the RFID reader was used to build heatmaps of all of the RFID tags seen on top of the occupancy grid.

To test the project for success, RFID tags were placed at regular intervals in a hallway, and the occupancy grids and heatmaps were constructed. Then, the distances between the centroids as determined by the program were compared to the actual interval length. In the first test, 5 tags were placed along the wall with 4 feet in between each tag, and the program determined them to be 3.77ft +- 1.38 ft apart. In the second test, 10 tags were placed 20 feet apart, and the program determined them to be 20.1ft +- 3.69 ft apart. Overall, the main goal of RFID tag localization was met. Further avenues for research include improving the SLAM occupancy grid software to better account for odometry drift and prevent the map from bending, and using Player’s built-in wavefront driver for global navigation with waypoints on an occupancy grid.

The source code repository for this project can be found at http://code.google.com/p/hospirfidbot/

I. Project Aim

The goal of this project is to create an autonomous robot that can drive around an unknown environment and create a 2D map, with the strength of every RFID tag that it sees recorded at each open space on an "occupancy grid," or map of the environment. After the map is created, the robot should then be able to use all of that information to reliably reach an arbitrary location in that environment chosen on the map. The primary application of this technology will be to map out a hospital and then to choose locations on the map to which the robot can deliver supplies. A "proof of concept" approach during my time here will be to set up RFID tags along a hallway and to make a best guess at their positions on the occupancy grid after driving the robot through.

II. Background and Basis for Project

Robotic navigation has been explored in a variety of contexts. Often, it may be useful to have a robot explore an environment that is treacherous for humans, such as a war zone or a vast desert, or an environment that is impossible for humans to reach, such as Mars. But robotic navigation is also increasingly relevant in many day-to-day applications as part of the artificial intelligence pipeline. Automating everyday tasks with robots can cut costs and divert human labor elsewhere. This project explores one such application, which is to have a robot deliver supplies to an arbitrary location within a hospital.

At a first glance, the problem does not seem too difficult. After all, the robot has a variety of sensors, including a very accurate laser, as well as an odometer, and it has control over motors that can propel it in any direction it wants. Once the robot has access to a map, it should be as simple as following the map by recording the distance traveled and degrees turned using the odometer. One major problem, however, is that the sensors are inherently noisy. This means that if the robot relied on odometry alone to predict its position on the map, the error would compound to unbearable degrees very quickly, due to slippage of the wheels and other factors. Even if the sensors were perfect, however, the environment can change slightly, so following the exact copy of one map would lead to problems. Therefore, more complicated navigation algorithms are needed to account for the stochastic nature of this problem. The core of this problem, known as "robot localization," is to make a best estimate of a robot's position on a map at each timestep. The algorithms that solve this problem usually maintain a "belief" (expected value) of the robot's position with a certain probability distribution, which is updated as the robot exercises controls on its environment and takes measurements. One of the most commonly used algorithms for this particular class of problems is the "particle filter."

Aside from noise in the sensors during navigation, another problem is that the robot could be deployed in a variety of environments, in which case it would need a new map every time. Also, as noted with the navigation, even a single environment could evolve over time. Hence, a map building step is also needed in this application if it is going to be in any way robust. The problem is to have a robot explore as much of the environment as possible without really knowing anything about it, and then to create an "occupancy grid," or a type of map where each cell is either open (meaning the robot can safely drive into it) or occupied. This is an extremely difficult contemporary problem called "simultaneous localization and mapping," or SLAM for short. Although it is allegedly a "solved problem," it sometimes takes a few years to implement. Thus, an added challenge is to get this working properly either with an open-source SLAM implementation or a watered-down implementation that can be done in a shorter timeframe.

One of the main aspects of this project will be to explore how RFID tags, which do not require power and can be dispersed widely throughout an environment, can be incorporated into the robotic exploration pipeline.

III. Procedures

A. Setting up the Software Development Environment

Our team decided to use an open source solution called "PlayerStage" as our robot interaction platform **[2] [9]**. Player is a high level hardware abstraction interface used to program robots. It has a set of "interfaces," such as a laser interface, an odometry interface, etc., which it links to drivers that represent the actual hardware. For instance, a Hokuyo urglaser and a Sicklaser, two different types of 2D laser scanners, may need different drivers, but they have the same interface in code. This greatly simplifies the configuration process, since hardware of the same class can easily be switched in and out without needing to change the code.

The second component is Stage, which is a robot simulation environment. Stage can be used to "fake" hardware devices to Player, so that test programs can be run in simulation before testing them in the real world. This is a useful debugging tool, since it allows the programmer to test new control programs in simulation and to detect errors more quickly with less damage potential to the actual robot.

PlayerStage runs under Linux, and robot control programs are written as clients to Player in C++. For organizational purposes, we created a subversion repository on Google code that houses all of our code. The steps for setting up the full software environment on a netbook and configuring the repository are as follows:

```
Operating System: Fresh install of Ubuntu 9.04 32-bit

Our software repository:
```
http://code.google.com/p/hospirfidbot/

```
1) Prepare a USB key with the Ubuntu installer on it by booting first from
the Ubuntu install cd, and then running the "usb-creator" program
```
https://help.ubuntu.com/community/Installation/FromUSBStick

2) When the netbook starts up, press F2 to access the BIOS. Under the "Boot" menu, disable "Boot Booster." This was a necessary step before replacing ram and putting on a new operating system

3) Save changes and shut down computer. Insert USB key with Ubuntu install. Power computer on and hold down "Esc" to boot from the USB key, then follow the install instructions

4) In Synaptic Package Manager under Settings->Repositories
Check off everything under Third Party Software. Then click "Reload" in the main GUI

Install the following packages in synaptic package manager:
*ssh (for administering computer remotely...will install both client and server and start server automatically)
*x11vnc (for launching x applications remotely)

subversion, kdesvn, cmake, autoconf, libltdl7-dev, libfltk1.1, libfltk1.1-dev,
libiceutil33,libavc1394-dev, libdc1394-22-dev,python-all, python-all-dev, python-opencv,g++, libgtk2.0-dev, libcv-dev,libstatgrab6, libstatgrab-dev,libpqxx-dev, libgnomecanvasmm-2.6-dev, libgsl0-dev,libPlayerxdr2-dev,libPlayerdrivers2-dev,libPlayercore2-dev,libPlayererror2-dev,libPlayerc2-dev,libPlayerc++2-dev,libgcl-dev,libglut3-dev,libstatgrab6,festival,festival-dev (for speech synthesis),imagemagick (Used later for DP_SLAM)

5) Go to Playerstage.sourceforge.net and download the most recent major releases of Player and stage. As of writing this tutorial, they are Player 2.1.2 and stage 2.1.1, released on January 15, 2009 and January 16, 2009, respectively

6) Open up the file "~/.bashrc" for editing
Add the following lines at the top:

```
export PATH=/usr/local/bin:$PATH
export CPATH=/usr/local/include:$CPATH
export LIBRARY_PATH=/usr/local/lib:/usr/local/lib/gearbox/:$LIBRARY_PATH
export
LD_LIBRARY_PATH=/usr/local/lib:/usr/local/lib/gearbox/:$LD_LIBRARY_PATH
export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig:$PKG_CONFIG_PATH
```

Do the same thing for /etc/bash.bashrc

Then relaunch the terminal

7) Extract the Player-2.1.2.tar.gz file and cd into it
Type "./configure –enable-rtkgui"
(the rtkgui is needed for the AMCL debugging GUI)

Then type "sudo make install"

8) Next extract the stage-2.1.1.tar.gz file, and CD into it
type "./configure"
type "sudo make install"

```
9) Next, check out the code from our google code repository:
http://code.google.com/p/hospirfidbot

10) Next, create a file called "kdesvn.sh" in the directory
~/.gnome2/nautilus-scripts/kdesvn.sh

that contains the code
#!/bin/sh/
kdesvn $1

This will allow for easy commits and updates to/from subversion
```

B. Building the robot platform

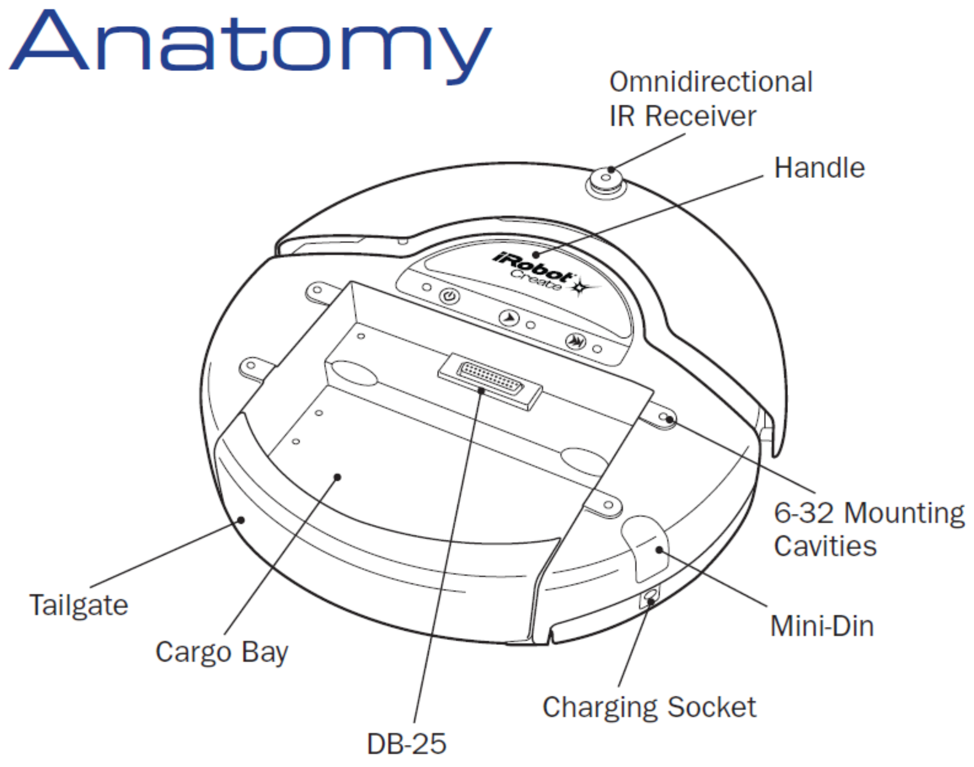


Figure 1: The iRobot Create Anatomy (Courtesy of the iRobot Create Open Interface Manual [5])

The base of the robot platform is the "iRobot Create," which is a version of the "Roomba" automatic vacuum cleaner, a circular robot with a bumper and IR sensors that drives around the room by itself and vacuums. The iRobot Create is simply a programmable Roomba without the vacuum. We also purchased a Hokuyo URG-04LX Laser to scan in a 270 degree field of view around the robot (actually, only the 180 degrees in front of the robot are needed). This laser requires a half amp at 5 Volts, but the Create outputs at around 16 Volts. We purchased a voltage converter to step that voltage down so we could power the urglaser from the Create for convenience. This required soldering together a plug to go into the base, where there are pins that connect to the iRobot's power source (labeled DB-25 in Figure 1).

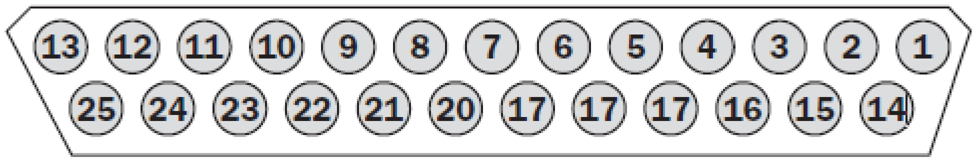


Figure 2: The female end of the plug on the iRobot Create; Pin 10 is used for the battery's voltage, pin 14 is used for ground (Courtesy of the iRobot Create Open Interface Manual [5])

Next we added an RFID reader, which was able to take the straight 16 Volts from the iRobot Create's battery. We connected a large, white antenna to the RFID reader, which is connected behind the urglaser and in front of the netbook. It is placed on top of spacers so that it clears the Hokuyo.

The next step was to figure out where to execute the robot control programs. The navigation part of the pipeline with the particle filter is an expensive operation, as is the data logging of 181 laser scans 5 times a second. Hence, it is important to have as much processing power as possible; however, we do not want to sacrifice portability and the compact nature of the robot. The best solution to this tradeoff was to purchase an Asus netbook with a 1.66Ghz processor, 2 GB of RAM, and a ~9 hour battery life. I built a platform on top of the iRobot Create to hold the netbook. I velcroed the netbook to the top of this platform with industrial velcro, and I screwed in the Hokuyo Urglaser at the front of the platform.

We also added a USB webcam to the front of the platform that can be used to capture still pictures as the robot is creating the map. Everything connects to a USB hub [**figure 3]** that is plugged into the side of the netbook. The purpose of this is to minimize the amount of plugging and unplugging when the netbook is connected/disconnected from the platform (only the hub needs to be connected). Also, I set up device aliases for the iRobot Create, the laser, the RFID reader, and the webcam using *udev* **[1]**. Udev has the system automatically create symbolic links to known devices (for example, devices with a specific vendor or ID) when they are found. This allowed me to automatically assign device strings to everything, regardless of when and where it was plugged in. For example, the RFID reader is always assigned **/dev/robot/rfidreader** (everything is under /dev/robot/*). This makes it easier to write configuration files that tell my programs where the devices are, since I know their assigned links will not change (compared to before, where, for instance, the RFID reader could be /dev/ttyUSB0 or /dev/ttyUSB1 depending on what order I plugged it in, meaning I would need to change the configuration file each time).

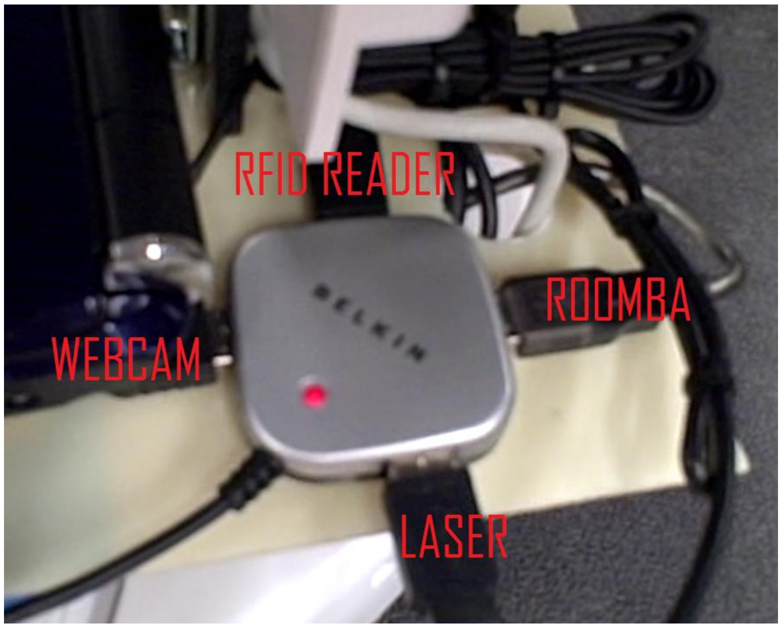


Figure 3: 4 Port USB hub

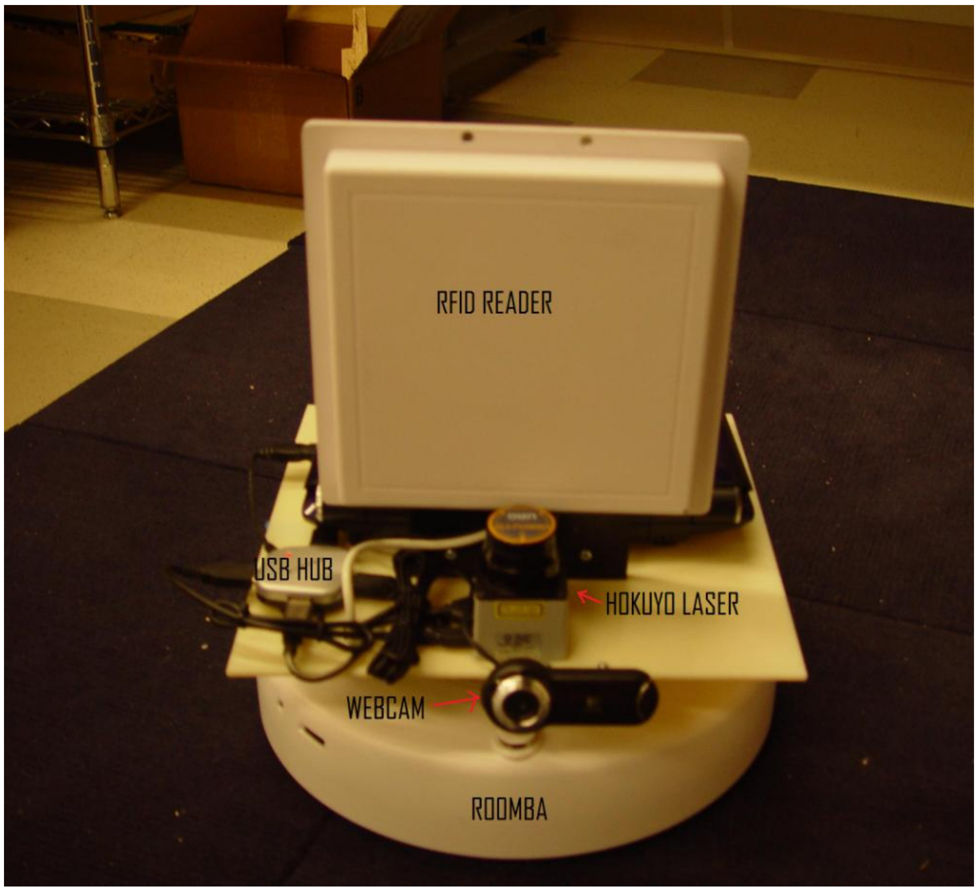


Figure 4: The fully assembled platform

NOTE: Most of the development is done on the netbook using a remote desktop (VNC). This also allows us to control the robot through the wireless network when we need to manually override certain settings during navigation

C. Writing an RFID Driver for Player in C

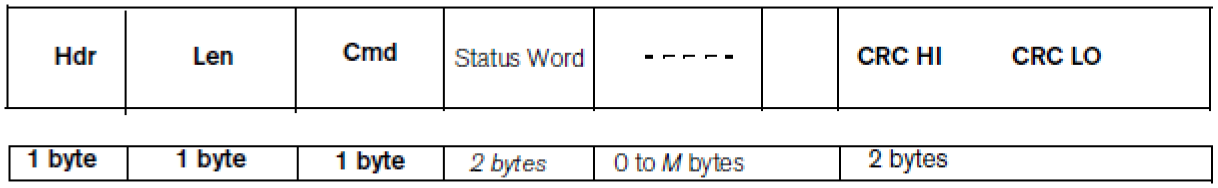
The iRobot Create's odometry data and the Hokuyo urglaser scans have drivers that are already written in Player. However, there is not currently a driver written for Player that can communicate with the specific RFID reader that we are using. The first attempt was to use a driver written in Python that can continually query the RFID reader with a certain timeout and return back the hex IDs of the tags seen along with their strength. In isolation, this driver performs its job adequately. However, when the RFID driver is paired with the Player interface, the two seem to interfere with each other. If I start the RFID driver first, Player will fail to start, so I won't be able to interact at all with the laser or odometry. If, on the other hand, I start Player first and attempt to start the RFID driver, Player will freeze and spit out lots of strange hex data. After a couple of days of experimenting, I decided that it was going to be necessary to write my own driver in C, since this problem is something out of the scope of this project that has to do with Python and C not both having proper access to USB devices.

I started with a simple prototype of a Player driver provided in an example directory in the Player installation. This example showed me where to initialize the hardware and where to put the main driver loop. I then had to figure out how to create a connection to the RFID reader and fill in the skeleton code. The RFID reader is connected to the netbook via USB like everything else, so I decided to use the **termios [9]** library in C to initiate a serial connection with the device with one stop bit, no parity, and an 8 bit strip size. I then had to consult the "Mercury ® 5e and M5e-Compact Devloper's Guide" to learn the communications protocol for the device. The basic structure for sending data to the RFID reader is as follows:

Hdr	Len	Cmd		- - - - -		CRC Hi	CRC LO
1 byte	1 byte	1 byte	0 to *N* bytes			2 bytes	

[7] Figure 5: Structure for sending a request/message to the reader

The first byte should always be **0xFF**, the second byte is the length of the "data" section, specifying that the data section is **N** bytes long. After that is the *opcode* for the specific command that the reader should execute. The data section can provide additional parameters for each command. Lastly is two bytes specifying a CRC checksum to ensure that the data was properly received (the checksum is generated on the client side, and the reader creates its own to compare with it to check for the validity of data transfer). After the request is packaged together and sent off byte by byte to the reader, the reader will send back a response. The basic structure for receiving a response is as follows:



Hdr	Len	Cmd	Status Word	- - - - -		CRC HI CRC LO
1 byte	1 byte	1 byte	*2 bytes*	0 to *M* bytes		2 bytes

[7] Figure 6: Structure of the response messages from the reader

The header and the length are the same, and *cmd* gives the opcode of the last command that the reader executed (for verification purposes). The status word is a 2-byte word that indicates the success of the last command. If the last command executed successfully, the status word should be **0x0000**. Otherwise, it will be some error code. And once again, there is a data section and a checksum at the end. When I am reading back a response from the reader, I usually give it some time to get all into the buffer, so I loop until I have (1 + 1 + 1 + 2 + M + 2) bytes of data read, where M is the length of the data section. That is, based on how long the data section is (given by the second byte I get back), I can determine the entire length of the message, so I know how much I need to read in before I break to the next section.

Once I have methods in place for communicating with the driver, I can begin to fill in actual message envelopes to control the reader and receive data back from it (NOTE: All of the operations below are carried out using commands specified in the manual [**7**]. For brevity, I will omit the specific opcodes and parameters). The first thing I do is connect to the reader at 230400 baud. If this fails, it probably means that the RFID reader has just been restarted, and it is sitting in a bootloader waiting for a message to boot. In this case, I need to reconnect at 9600 baud (the default rate), change the baud rate to 230400, reconnect at 230400 baud, and tell the reader to boot the firmware. At this point, I set the antenna ports on the reader, set the read power to 3000, set the communications protocol to "Gen 2" (required by the manual), and set the region to US. The reader is now ready to query the environment to see what tags are visible.

In the main driver loop, I send a command to the reader called "read tag multiple," with a timeout (specified in milliseconds). This timeout gives the reader a chance to look around in the environment and find tags that are "visible" (strong enough signal). I am using a timeout of 50 currently. This means that I actually have to pause the control flow of the driver to wait for the reader to catch up. Once I have paused the control flow for at least 50 milliseconds, I can read back the response message, which will contain all of the tags seen in the data section. Each tag has its own hex id and a strength, which are written to a logfile with a timestamp.

D. Data Collection and Logging

Once all of the drivers are properly configured, the next step is to collect the data from the odometer, laser scanner, and RFID reader and to log that information to a file continuously as the robot explores its environment. Player has logging capability for our odometer on the iRobot create and laser scanner built in, and they are logged in the following format:

<time> <host> <robot> <interface> <index> <type> <subtype> <data….>

The important fields above are the *time*, which is the Unix epoch time that the data was logged, the *robot* number, which is the port to which the robot is connected, the *interface*, which is a string that describes what the device is (either "laser" or "position2d" in this case), and the *index*, which specifies to which robot the device connects (a Player-assigned robot number). Player can deal with multiple robots at once by having different devices on different indexes, but it is important to keep the laser and the odometer both on the same robot (i.e. index 0 in the case of this program). Otherwise, the SLAM map builder will get confused and fail to build a proper map, since the laser and odometer will not appear to be on the same robot. Here is an example of a line with odometry data:

```
1245349416.451 16777343 6665 position2d 00 001 001 +01.155 -00.971 -1.736
```

Note that only the first three fields are important; they give the x position, the y position, and the yaw (rotational orientation), in that order.

Here is an example of laser data:

```
1245349397.275 16777343 6665 laser 00 001 001 0000 -1.5708 +1.5708
+0.01745329 +5.6000 0181 1.264  0 1.264  0 0.938  0 0.939...(truncated)
```

The first two fields in the data section give the leftmost scanning angle and the rightmost scanning angle of the laser. The third field gives the angular increment between the steps, the fourth field gives the maximum range of the laser (in meters), and the fifth line gives the number of samples that the laser took around the arc from the minimum angle to the maximum angle. In this application, the laser always takes 181 samples, starting at -90 degrees (-1.5708 radians) to 90 degrees (1.5708 radians), in 1 degree (0.01745329 radians) increments. Note that I actually had to down sample the laser slightly to get it into a format accepted by the SLAM client (the laser can actually read in finer increments and around a wider arc).

The last phase of the data logging is to log the data from the RFID reader simultaneously with the laser and odometry data, with the driver that I created in Player. I adopted the same logging scheme that the previous driver in Python used. Here is an example line from a log file created with this scheme:

```
1245886325.07 7 0000000000000000e001170 83 0000000000000000e00115e 86
0000000000000000e00114e 89 0000000000000000e001158 84
0000000000000000e001160 80 0000000000000000f001141 82
0000000000000000f001151 81
```

The first field is, once again, the Unix epoch time. The second field is the number of tags that were seen that timestep (7 in this case). Following the second field is a pair of data for each of the tags that was seen; first the hexadecimal ID of that tag as a string, followed by the strength of that tag.

E. Map building / Offline SLAM

Once the laser and odometry data are logged, an "occupancy grid," or a 2D map of which parts of the environment are likely open and which are obstructed, can be built using SLAM techniques. I am using a utility program wrapped with Player called **pmaptest** to construct the map **[4]**. A typical call to this program would look like this:

```
pmaptest --grid_scale 0.05 --laser_x 0.13 logfile.log
```

This instructs the map builder to create a bitmap with each pixel covering a 5cm x 5cm area of the map, using laser and odometry data from **logfile.log**. It also lets the map builder know that the laser is positioned 13cm from the center of the robot, so that the laser and odometry readings can be properly synchronized.

After experimenting with this utility, I discovered a few quirks that I needed to address. First of all, pmaptest uses different levels of gray on a spectrum to specify how sure it is that a cell is unoccupied, with darker grays indicating more certainty that a cell is occupied, and lighter grays indicating more certainty that a cell is unoccupied. The output from pmaptest is generally very bright (nearly white) along the path that the robot traversed **[figure 7]**. However, it is not white enough just outside of that path for the AMCL driver (used for localization in Player) to believe that they are valid points where the robot can rest. Hence, Player does a terrible job of localizing to that map since it perceives the map to be just a very thin line area (along that white line in the center). To correct for this, I created a filter to quantize more of the grays just outside of the actual path to white. I did this by first creating a grey level histogram of the original occupancy grid and picking out the peak. This peak will most likely be the background color. I then took the average of all pixels that do not have this grayscale level, and map all of the ones above that average to white (and preserve the rest of them). The right of **[figure 7]** shows the result of this feature.

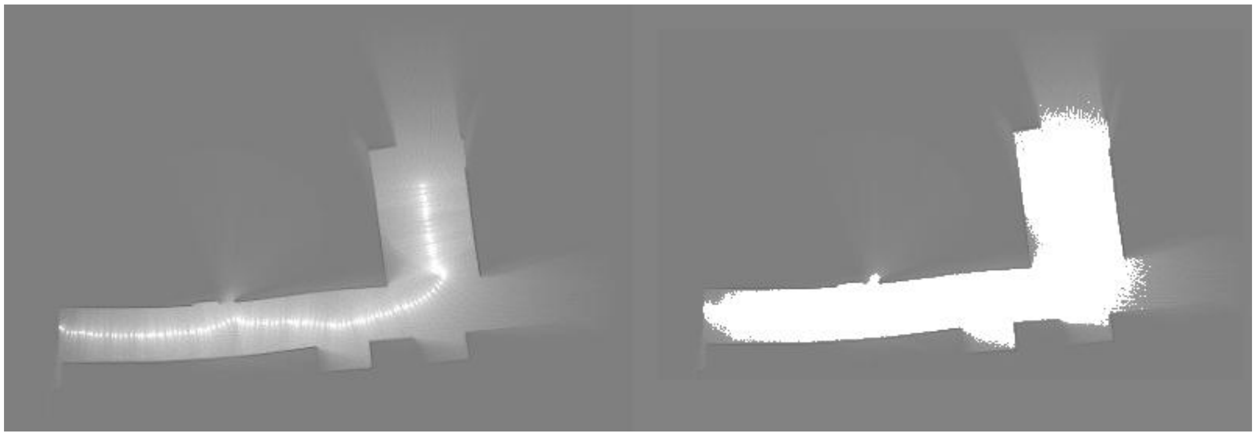


Figure 7: Left is output from pmaptest, Right is output from my filter

F. Heatmap Generation

Once the occupancy grid is built, we need to generate a "heatmap" for every RFID tag seen during the mapping session. The heatmap is a PGM (simple grayscale format) bitmap image with the same dimensions and resolution as the occupancy grid created with pmaptest, and it is drawn on top of the map so that the user can visually see where the tag was visible on the map. To do this, the data from the RFID tag logs and the updated odometry data need to be synchronized. The process is as follows:

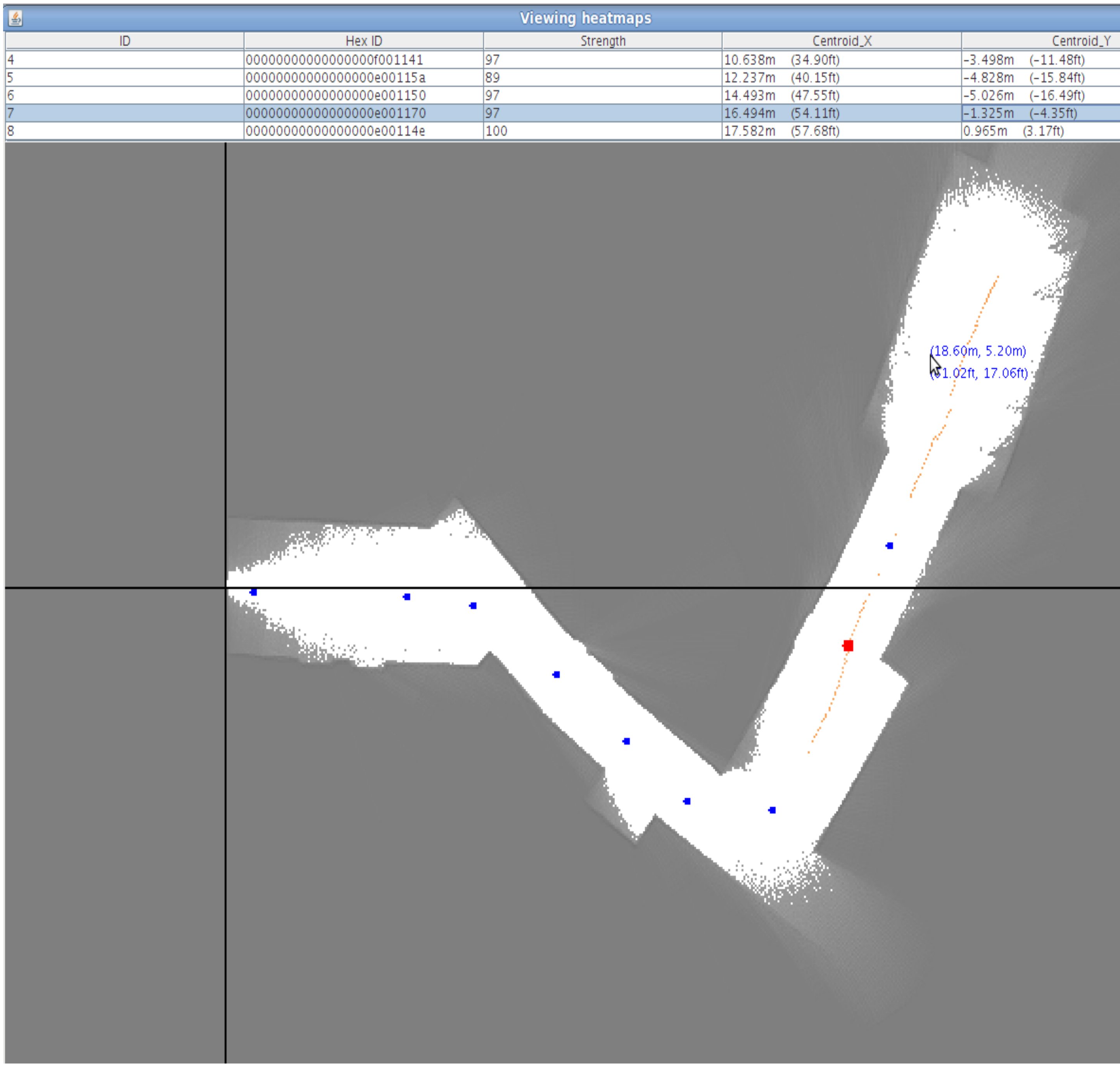
1) Replay the log file with the laser and odometry data, giving Player the occupancy grid and setting up an "AMCL". AMCL stands for "Adaptive Monte Carlo Localization," which is the localization process that is used to give laser-corrected odometry data on a map. This phase can replay the logfile with the raw laser and odometry data and output the corrected odometry data to a new logfile, usually called "localized.log."

2) After step 1, there now exists a new logfile that has the best guess of the position of the robot on the map. I then need to loop through that logfile and look back at the RFID logfile to discover which tags were seen at that time. That is, I have the Unix epoch time that each laser-corrected odometry reading was made, and I have the Unix time that all of the RFID tag readings were made. I find the closest time in the RFID log file to the time that's given in the updated odometry logfile, and I assume that those were the tags seen at that time at that position. For each tag, I look to see what the strength was, and I put a correspondingly bright or dark pixel on the heatmap at the best-guessed position based on the strength (lighter pixel means stronger reading at that position).

   NOTE: When I load the RFID logfile, it assigns an integer ID to all of the tags to make them easier to pick out than the extremely long 24-byte hex IDs. The integer IDs are assigned in the order that the tags were seen, with 0 being the integer ID of the first tag seen.

3) After I'm finished looping through the entire updated odometry logfile, I can write the heatmap for each tag out to a PGM grayscale file.

4) I wrote a program in Java that is used to view the heatmaps interactively, loading all of the heatmap files and the occupancy grid file. The program first displays the occupancy grid, along with a table above it that gives information about each RFID tag seen. The table gives the tags in the order that the tags were seen (numbered starting at tag 0), with the tag's hex ID, the tag's strength, and the "*centroid*" of the tag in meters. The centroid is the point at which the tag was perceived to be the strongest, at it is the program's best guess at the tag's actual location. Then, the program draws all of the centroids on top of the occupancy grids as blue dots, so the user can get an idea of where all of the tags were. Then, to highlight a particular tag, the user can select it on the table, and a red dot is drawn over that tag.


Viewing heatmaps

ID	Hex ID	Strength	Centroid_X	Centroid_Y
4	0000000000000000f001141	97	10.638m (34.90ft)	-3.498m (-11.48ft)
5	0000000000000000e00115a	89	12.237m (40.15ft)	-4.828m (-15.84ft)
6	0000000000000000e001150	97	14.493m (47.55ft)	-5.026m (-16.49ft)
7	0000000000000000e001170	97	16.494m (54.11ft)	-1.325m (-4.35ft)
8	0000000000000000e00114e	100	17.582m (57.68ft)	0.965m (3.17ft)

Figure 8: Heatmap viewer

G. Replaying webcam frames with localized odometry

While player is running and logging laser, odometry, and RFID data, I also have it logging frames from the webcam on its front to a logfile. Each line of the logfile contains the hex data for a JPEG image stored as a string, along with a system time that the frame was taken. I created a program to convert this logfile into a bunch of JPEG images. I then created a program to view these frames as a video alongside the map-corrected odometry data (the program's best guess at the pose of the robot at the time that the frame was taken).

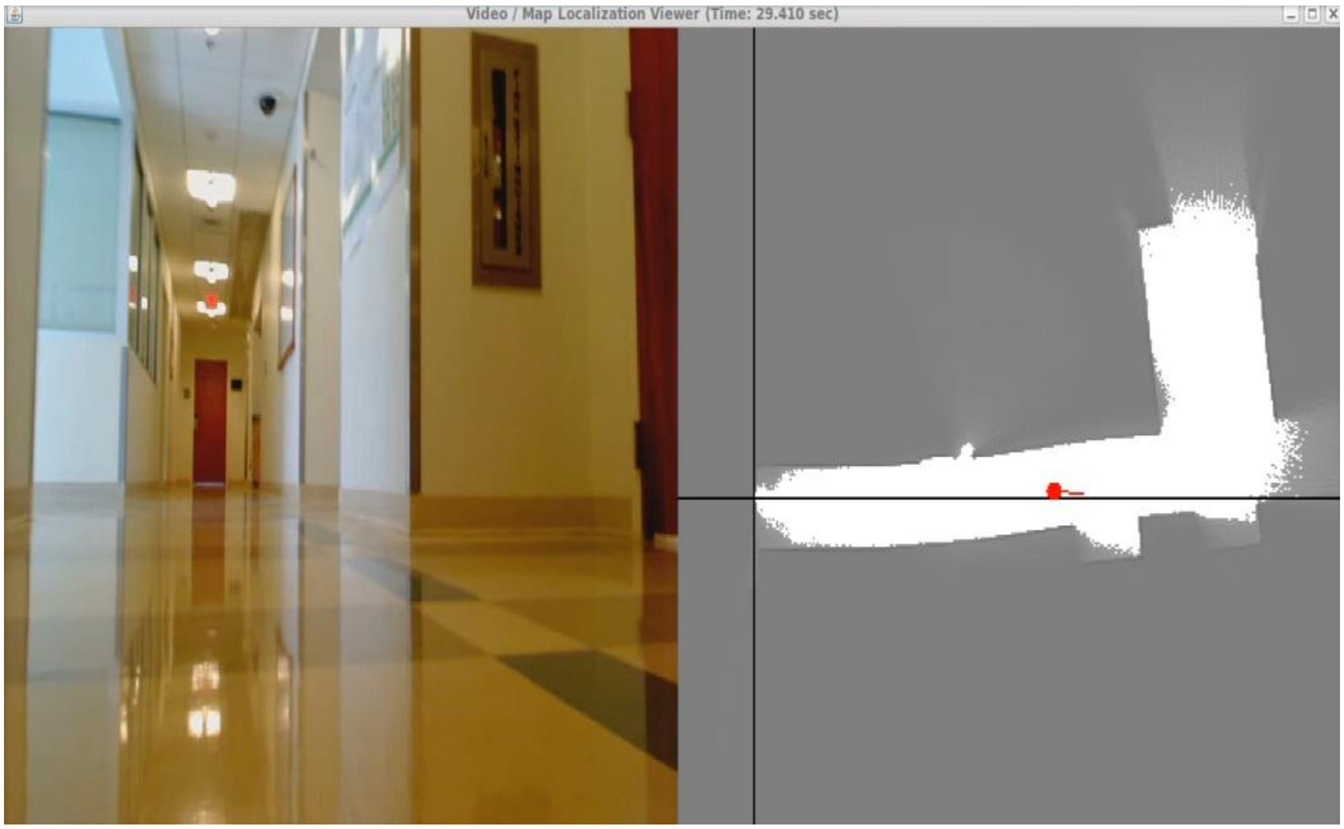


Figure 9: Replaying the video from a session alongside the program's best guess of the robot's pose

IV. Testing for Success

A. Testing Procedures

My main mission in this project was to combine known SLAM techniques with my own ad-hoc RFID tag localization process. Therefore, to test my work for success, I will focus on the RFID heatmap aspect of the project. To test this, I placed tags at regular intervals in hallways and drove my robot through. After this, I generated heatmaps from the logs and measured the distance between the centroids of each tag, comparing that to the interval I actually measured in the hallway. To make it easier to place the tags in regular intervals, I counted floor tiles between them, which are each 1 foot long.

To automate the process, I wrote a simple local hallway navigation program to drive towards the center of the hallway. The program converts the laser range scans, which are in polar coordinates, into a rectangular grid of cells that are open or closed. Then, the program takes the centroid of all open (unoccupied) cells, and angles itself towards that location **[figure 10]**. This also has the advantage that when the robot gets to a turn in the hallway (as in [**figure 10**]), it will automatically execute that turn since there is much more open space in the direction that the hall turns. I make the robot slow down the more it has to turn, so it will cruise nicely when it's driving down the center of a vacant hallway and execute such turns without crashing. This program works extremely well and automates the data collection process quite nicely. However, I still need to override it in certain scenarios, such as where it takes a wrong turn at a T intersection. But this can be done easily through VNC using a utility called "playerv" that gives me a little joystick to control the motors (and then I return control to my program after I get past the problem spot).

NOTE: All tests are performed in the 3rd floor of the Fitzpatrick building.

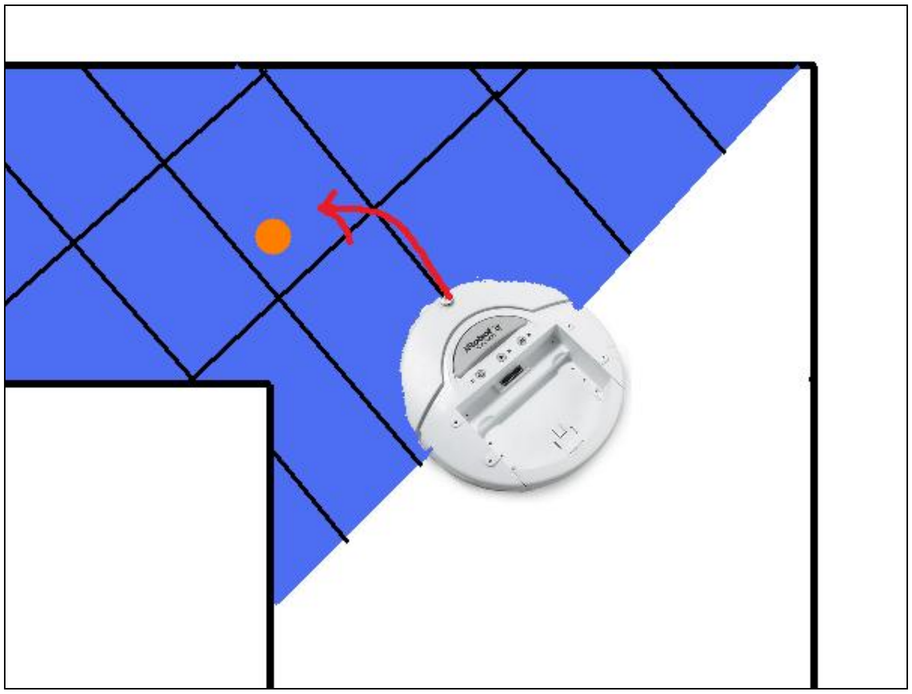
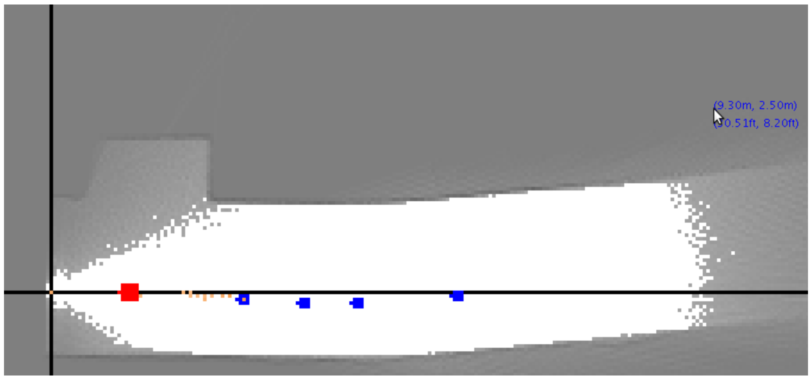


Figure 10: Depiction of the hallway drive program that drives towards the centroid of open space

B. Results

I. Test 1

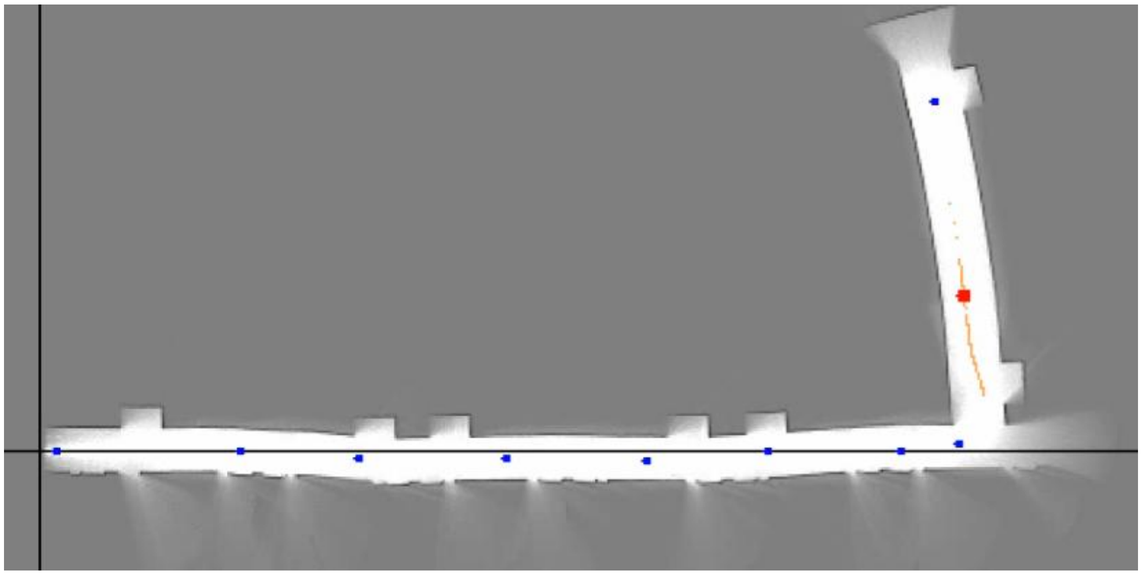
Scenario: Placed 5 tags four feet apart in a straight section of hallway. The tags were mostly seen in order, except the RFID reader sensed the third tag before the second tag.



Tag Order	Tag Hex ID	Max Strength	Centroid X	Centroid Y
0	00000000000000000e001144	100	1.129m (3.70ft)	-0.037m (-0.12ft)
1	00000000000000000f001141	94	3.598m (11.80ft)	-0.161m (-0.53ft)
2	00000000000000000e001170	100	2.744m (9.00ft)	-0.113m (-0.37ft)
3	00000000000000000e001150	96	4.337m (14.23ft)	-0.159m (-0.52ft)
4	00000000000000000f00114d	99	5.724m (18.78ft)	-0.058m (-0.19ft)

II. Test 2

Scenario: Placed 10 tags 20 feet apart; 7 which go along the x axis, 1 which lies at a corner, and 2 which go along the y axis. These tags were all seen in the correct order (with the leftmost tag seen first)

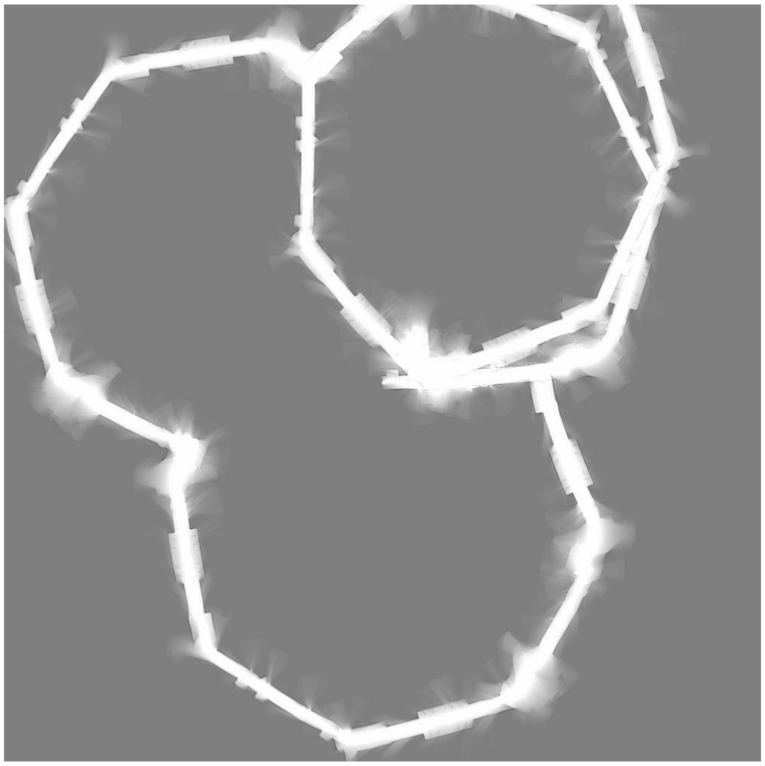


Tag Order	Tag Hex ID	Max Strength	Centroid X	Centroid Y
0	00000000000000000e00114c	102	0.783m (2.57ft)	-0.033m (-0.11ft)

1	00000000000000000f001141	94	8.363m (27.44ft)	-0.093m (-0.31ft)
2	00000000000000000f001151	105	13.26m (43.50ft)	-0.346m (-1.14ft)
3	00000000000000000e001170	103	19.309m (63.35ft)	-0.339m (-1.11ft)
4	00000000000000000e00114a	100	25.149m (82.51ft)	-0.445m (-1.46ft)
5	00000000000000000f00115f	92	30.109m (98.78ft)	-0.0060m (-0.02ft)
6	00000000000000000f001145	94	35.601m (116.80ft)	-0.048m (-0.16ft)
7	00000000000000000e001146	88	38.065m (124.89ft)	0.337m (1.11ft)
8	00000000000000000e00115e	96	38.245m (125.48ft)	6.455m (21.18ft)
9	00000000000000000e00114e	98	37.078m (121.65ft)	14.416m (47.30ft)

III. Test 3

Scenario: Placed 16 tags 10 tiles apart in a rectangular loop on the third floor Fitzpatrick building that goes through a glass walkway, and drove it around that loop 6 times.



C. Analysis

I. **Test 1**: The average distance between tags is **3.77ft +- 1.38 ft**. The standard deviation is quite large compared to the average here, but this is most likely because the tags are so close together. The tags could still be used for verification purposes during navigation even with that error.

II. **Test 2:** The average distance between tags is **20.1ft +- 3.69 ft**. These results turned out extremely well (the average is very close to 20ft, the actual distance between tags), there are just a few outliers causing the standard deviation to go up (24.9, 26.12, 16.06, and 16.27). Also note that the outliers come in pairs, since one tag that is detected too close to another will be detected too far from the tag on its other side. In other words, there are really only 2 tags out of 10 that had very noticeable localization error.

III. **Test 3:** The occupancy grid turned out so badly here that it wasn't even worth trying to create heatmaps for the tags. This highlights an unresolved problem with the map-building process; that of odometry drift. As previously mentioned, the odometry data is extremely unreliable, and errors with odometry will compound over time. In this case, the errors were so profound that right angles at the corner of hallways turned into obtuse angles in most cases, and the SLAM client was unable to close the loops.

V. Conclusions / Further Avenues for Research

The main purpose of this project was to be able to create RFID heatmaps while driving a robot through a hallway and collecting data with a laser range scanner, an odometer, and an RFID reader. This goal of the project was met. The ad-hoc approach of marking the heatmaps at the strongest location as an estimate of each tag's location worked surprisingly well. Though it is not perfect, it can certainly be used to help with global navigation and for robot localization. That is, the RFID tags can serve as a "sanity check" of sorts to check the work of the particle filter.

One aspect of the project that still needs attention is the map-building process. In the interest of time, I did not implement my own SLAM occupancy-grid builder, but instead relied on pmaptest [**4**]. The advantage of pmaptest over other SLAM clients is that it is directly compatible with Player; the logfiles of laser and odometry data that Player spits out can be fed directly to the utility without modification, and it will do its job. As advertised, it is supposed to give laser-stabalized odometry to reduce odometry drift that causes the map to bend. However, in practice, it does not do this nearly as well as advertised, as seen in example 3. Unfortunately, looking back at the web site, I realized that the program has not been updated since December 2004, and the documentation for that even is outdated. Therefore, for the future, it may be necessary to explore other options for doing SLAM. One option I just began to explore is

DP_SLAM, an occupancy grid builder created by Duke's own Ronald Parr **[8]**. I created a program to convert Player's logfiles to a format acceptable by DP_SLAM (*see Code Summary: player2dpslam.cpp*), and I began to experiment with it. It appears to do a much better job than pmaptest at correcting for odometry drift, but it is much slower. This is probably a good tradeoff in the long run, but it will need to be explored more.

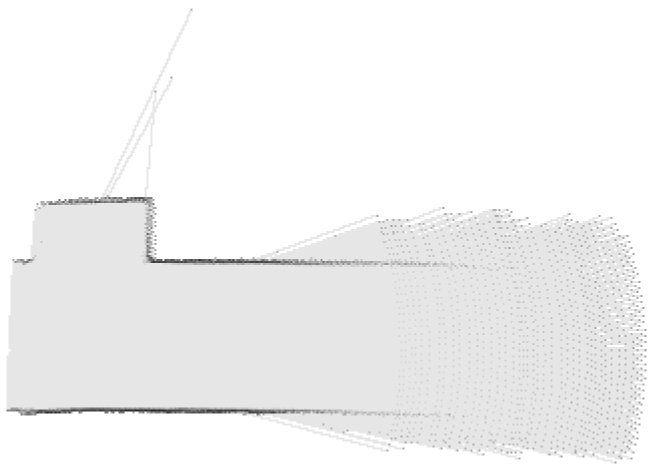


Figure 11: Partial output from DP_SLAM

Another further avenue for research is global navigation along an occupancy grid (in addition to my local hallway navigation). I also began exploring this, but ran out of time. I found a driver built into player called the "wavefront" driver, which is able to do global navigation and create waypoints if the user correctly estimates the robot's initial pose on the occupancy grid. I got this to work well in Stage, but it doesn't work on the actual robot yet; for some reason, it doesn't initialize properly. I am very close to getting this working, so it should be a viable option soon.

VI. Acknowledgements

Special thanks to:

- Matt Reynolds: My faculty adviser
- Travis Deyle: Visiting graduate student from Georgia Tech experienced with Player and RFID technology
- Stewart Thomas: My Duke graduate student mentor
- Taylor Gronka: Duke undergraduate who started before me and helped to figure out PlayerStage

- Martha Absher: REU Program coordinator, who chose me and made this all possible
- National Science Foundation: For providing funding for this endeavor

VII. Code Summary

Here is a summary of all the important code I have written for this project:

A. Primary Java and C Source Code Files

args.h: A helper file taken from the Player example codebase, that parses command line arguments that specify things like robot port number, etc.

AutoQuantize.cpp: Provides access to the function that will clean up the occupancy grid that pmaptest spits out and puts it into a format that the AMCL driver can better understand. It calls a function in pgm.cpp that takes the average of all of the cells in the occupancy grid (excluding the background cells), and maps all of the values above that average to 255. This makes the AMCL driver "more sure" that certain regions are unoccupied, leading to significantly better localization performance

comparators.h: A helper file for some of the other files that use STL maps, has some comparators for different templates

hallwaydrive.cc: A program that can navigate down a hallway using laser scans and sending commands to the motors. The program rasterizes the laser scans into a 2D rectangular grid of occupied or unoccupied cells. It then calculates the centroid of these cells and steers towards that. This makes it follow the center of the hallway.

heatmap.h, heatmap.cpp: This is a library used to create RFID heatmaps. The program is given a PGM image map file, along with that map's resolution (in meters),a logfile of odometry readings localized to that map (with system timestamps), and a logfile of RFID tags seen (also with sytem epoch timestaps). The program will create a heatmap of every RFID tag seen that aligns with the map by matching up the two logfiles; that is, at every localized position, it will find all of the RFID tags that were seen at the time that position was recorded (or the closest time to that time that exists in the RFID logfile)

logs/log2jpeg.cc: When Player logs camera data to a logfile, it puts a big long hex string of all of the JPEG data next to each timestamp. This program goes through the logfile and converts each hex string into an array of bytes, and then writes it out to a file for each JPEG image. In other words, it converts the logfile from the camera into an array of JPEG images in some folder, which can then be accessed by ViewVideo.java to play them back

makefile: An organized file to build all of the utilities that I've created

makeheatmaps.cpp: This is a program that wraps together a lot of utilities and outputs the heatmaps as greyscale images. This program takes a PGM image occupancy grid, a localized logfile of odometry data, and an RFID logfile.

pgm.h, pgm.cpp: A library I created that can load the simple PGM image greyscale format (without any comments in the header). This is the format that the pmaptest utility logs its occupancy grids to, so I decided to keep with that format and to write the heatmap data in this format.

PGMImage.java: A port of the PGM library to Java (that can only read, not write, PGM images), with added helper functions for drawing the axes to the occupancy grid and adding an alpha channel to the heatmaps (so that they're transparent where the tag was not seen). Having alpha channels and preloading an image buffer as such significantly speeds up performance (compared to drawing the images pixel by pixel each time), and it makes zooming in much easier (can rely on java's inherent image capability).

player2dpslam.cpp: A program that converts a player logfile with laser and odometry data into a format acceptable by DP_SLAM, a SLAM client written by Ronald Parr and Austin Eliazar (this is an alternative to pmaptest)

QuickRFIDView.cpp: This is a program that, given an RFID logfile, loads it using the rfid logfile library that I made (in rfid.cpp), and simply lists out all of the unique tags seen during that run in the order that they were seen. This is useful for checking to make sure that all of the tags that were expected to be seen during a testing run were actually picked up.

rfid.h, rfid.cpp: A library that loads all of the information from an RFID tag logfile into an organized class. It provides a function to help find the closest entry for a given time. The format of the log file is (Travis and I both use this format):

<unix system epoch time> <# of tags seen> <tag 1 hex id (string)> <tag 1 strength> .....

In addition to having Hex IDs, the program also assigns each RFID tag an integer ide, based on the order in which the tags were seen; that is, the first tag seen will have integer id 0, the second one seen will have integer id 1, and so on.

RFIDdriver.h, RFIDdriver.cpp: A driver in player that can communicate with the RFID readers that we're using. I modeled this off of Travis's RFID driver in Python (M5e.py). It first attempts to connect at 230400 baud, which will work if the RFID reader has already been initialized. If this is not the case, it will reconnect at 9600 baud, change the baud rate to 230400, and then tell the bootloader to boot. At this point, it's ready to query the tags by sending a "get tag multiple" command.

SimulateRFID.cpp: Before I had the actual RFID reader in place, I used this to fake RFID tag logfiles. This allowed me to start development of the heatmap software before all of the hardware was ready

speak.h, speak.cpp: Provides a command line wrapper to the Festival speech synthesis system; used with hallwaydrive so that the robot can say "please move out of the way" when someone is in the way (actually currently disabled because it was annoying)

ViewHeatmaps.java: Given a map and a set of heatmaps in the PGM grayscale format, along with a file that specifies the "centroids" of each RFID tag seen, create an interactive heatmap viewer. Have the capability to zoom in and out by scrolling and to change the point of focus by dragging the mouse. Draw a table on top of the GUI with all of the tags listed with their assigned integer ID and hex ID, along with the position of its centroid in meters and feet. Draw small blue dots on the occupancy grid for the centroid of each tag, and draw a larger red dot over the tag that is selected in the table.

ViewVideo.java: Program takes a logfile of localized odometry data, an occupancy grid, a folder with JPEG frames from a webcam, and a file that gives timestamps for all of those frames. It then advances through the frames in the left panel of the display. For each frame, find the system time it was taken, find the closest system time to that time in the localized odometry logfile, and draw that pose on an occupancy grid on the panel on the right of the display

B. Player Configuration Files

createbrain.cfg: The main configuration file for the platform that I have set up. It has drivers for the iRobot Create, the Hokuyo urglaser, the USB webcam, and the RFID reader. The file creates a logfile called "rawdata.log" in the logs directory that contains all of the laser and odometry data, it creates a logfile called "camera.log" in that directory with the frames of the webcam logged, and it creates a logfile called "rfidtags.log" with all of the RFID information. Note that there's a section in this file the down-samples the Hokuyo urglaser to be in the sicklaser format (181 bins from -90 to 90 degrees). This is required to use pmaptest

navigate.cfg: A configuration file that has access to all of the same hardware devices as createbrain.cfg, that is able to localize the robot to an occupancy grid and do global navigation with the help of Player's built-in AMCL (for localization), vfh (vector-field histogram avoidance for local navigation), and wavefront (for global navigation with waypoints). *This does not currently work, but it is a further avenue for exploration; it's very close to working.*

replay.cfg: This file takes a rawdata.log with laser and odometry data and a mapfile, and writes out a file localized.log with updated odometry data using the AMCL driver to align correct the odometry data to the map.

rfidtest.cfg: A configuration file for testing the RFID driver I wrote

simple.cfg: A configuration file used with stage to run simulations

VII. References

[1] Drake, Daniel. "Writing udev rules." *Reactivated.net - home of Daniel Drake (dsd)*. Web. 28 July 2009. <http://reactivated.net/writing_udev_rules.html>.

[2] Gerkey, Brian. "Mapping with the iRobot Roomba." *Artificial Intelligence Center @ SRI*. Web. 28 July 2009. <http://www.ai.sri.com/~gerkey/roomba/index.html>.

[3] Gerkey, Brian. "Player Manual." *Player Project*. Web. 28 July 2009. <http://playerstage.sourceforge.net/doc/Player-2.1.0/player/>.

[4] Howard, Andrew. "Simple map utilities: Simple mapping utilities (pmap)." *USC Robotics Research Lab*. Web. 28 July 2009. <http://robotics.usc.edu/~ahoward/pmap/index.html>.

[5] *IRobot(c) Create Open Interface*. IRobot Corporation, 2006. Print.

[6] "Java Platform SE 7 b59." *Developer Resources for Java Technology*. Web. 28 July 2009. <http://java.sun.com/javase/7/docs/api/>.

[7] *Mercury(R)5e and M5e-Compact Developer's Guide*. Cambridge, MA: ThingMagic, Inc, 2000-2008 (c). Print.

[8] Parr, Ronald. "DP-SLAM." *Computer Science - Duke University*. Web. 29 July 2009. <http://www.cs.duke.edu/~parr/dpslam/>.

[9] "Stage Manual." *Player Project*. Web. 28 July 2009. <http://playerstage.sourceforge.net/doc/stage-cvs/group__stage.html>.

[10] "Termios.h." *The Open Group - Making Standards Work*. Web. 28 July 2009. <http://opengroup.org/onlinepubs/007908775/xsh/termios.h.html>.

[11] Thrun, Sebastian. *Probabilistic robotics*. Cambridge, Mass: MIT, 2005. Print.

[12] "Writing a Player Plugin - RoboWiki." *The Penn State Robotics Club - Making the World Autonomous Since 2004*. Web. 28 July 2009. <http://www.psurobotics.org/wiki/index.php?title=Writing_a_Player_Plugin>.

[13] "Xvidcap | Open Source Alternative - osalt.com." *Find Open Source Alternatives to commercial software | Open Source Alternative - osalt.com*. Web. 28 July 2009. <http://www.osalt.com/xvidcap>.