2D Simultaneous Localization And Mapping

Peter Bailey

Matthew Beckler

Richard Hoglund

John Saxton

Abstract

A common challenge for autonomous robots is the Simultaneous Localization and Mapping (SLAM) problem: given an unknown environment, can the robot simultaneously generate a map of its surroundings and locate itself in this map? In this project, a solution to the SLAM problem was implemented on a Pioneer 1 robot equipped with a SICK laser scanner. Extended Kalman filtering was used to continuously estimate the robot's position within the map and the associated covariances. Although landmark updates were not fully implemented, heading updates were performed using a structural compass. Despite lacking landmark updates, our solution produced a reasonably accurate map of the indoor environment.

Introduction:

Autonomous robots are becoming increasingly ubiquitous. Most autonomous robots require a map of their surroundings and an estimate of their location within this map. It is also required that this map be generated autonomously. This problem is known as Simultaneous Localization and Mapping (SLAM). The solution to the SLAM problem employed in this project has three main steps: propagation, compass update, and landmark update. The propagation step uses a kinematic motion model with Kalman Filtering to predict the state and covariance of the robot in the next time-step. Next, laser scan data is used to detect the walls around the robot, which can be used to improve the estimate of the robot heading. Finally, landmarks are detected and an Extended Kalman Filter is employed to further improve our state estimate.

Literature Review:

Leonard and Durrant-Whyte[1] showed that estimates of landmark positions taken by the same robot are correlated with each other. This fact can be used to improve landmark position estimates. Smith and Cheeseman[2] established groundwork in using noisy measurements to estimate relationships and covariances between objects observed from multiple positions. The relationships between positions could result from the movement of a robot, estimated by odometry.   Thrun et. Al [3] introduced probabilistic localization and mapping methods sufficient to provide a reasonably accurate solution to indoor SLAM. This work, combined with the Extended Kalman Filter (EKF), is the basis for this project.

The Environment:

This particular solution to the SLAM problem is designed to map indoor environments. Specifically, the robot is designed to map individual floors of the Electrical Engineering and Computer Science building. As this is an indoor environment, we can assume that walls are generally at right angles to each other. This makes a structural compass feasible. There are a number of doorways, characterized by very small corners with a door surface parallel to the wall surface. It is also assumed that the environment is static; the walls will not change positions, doorways will (hopefully) remain closed during the course of our experimental trials, and people will not be walking around the hallways interfering with the laser scanner.

Software Architecture

The robot application code is written in the C and C++ programming languages, and is based on the ARIA robotics interface. The ARIA API provides a number of convenient objects and methods that greatly reduce the amount of low-level programming that is required to get our robot up and running. As a result, we are able to focus more on the high-level system design and algorithm implementation.

To assist us in the matrix mathematics, we have enlisted the help of a matrix implementation called CwMtx. It worked acceptably, although we had to extend the class to add a few new class methods. One of these new methods, the *printer()* method, would print out the matrix to standard output in rows and columns, for easier debugging. Another method we added was the *slice(int rowStart, int rowEnd, int colStart, int colEnd)* method, which was used to extract a rectangular section of an existing matrix. This was particularly useful when constructing the new covariance matrix after adding a new ghost pose. A final pair of methods, *inserthoriz(Matrix &mat, int whichCol)*, and *insertvert(Matrix &mat, int whichRow)*, were used to paste a matrix into another matrix, which was used notability in adding new landmarks and ghost poses to the state vector. We did encounter some difficulties with the matrix library; There were difficult problems to track down when trying to store the result of a matrix multiplication back into the multiplicand, such as in A = A * v. We had to resort to using temporary matrices in these situations. Another source of frustration was the fact that the matrices were never initialized to zero (or any other sane value) on construction, and were simply left as un-initialized allocated memory.

We based our robotics code on the SickWander example application distributed with the ARIA framework. It uses a multi-threaded layout to enable the robot to use a handful of ArAction objects to provide a simple control system. We extended upon this by adding an ArActionGoal that will direct the robot towards a specified pose in the global coordinate system. This target pose is updated by the SickNavigation thread, which will be explained in a later section.

The main code file of our system has very little to do. It initializes the robot and the Sick laser, initializes the shared data, and starts the sub-application threads running. Because we are using a multi-threaded approach, we need a method of sharing data between the threads, and have utilized a relatively simple approach to data sharing and protection, that of storing the data in a centralized location, all protected by a common mutex. Each thread that is spun up is passed an instance of our *argStruct*, which contains pointers to the shared data, as well as a pointer to the mutex. It is the thread's responsibility to obtain a lock on the mutex before reading or writing the shared data.

We have four threads that make up our SLAM system. The first, the *sickNavigation* thread, uses the laser data to look for open areas to travel towards. It looks for regions of laser data with infinite distance to drive towards, but it prefers to turn to the right, producing the desired clockwise travel path. More details will be provided about the navigation thread later in this document.

The second thread that is started is the propagate thread. It runs 10 times per second, and is responsible for querying the ARIA system for the robot's instantaneous linear and rotational velocity, and computing the state and covariance propagation from these values. It also is responsible for periodically saving a copy of the current pose and laser data snapshot, a concept we call a “ghost pose”, which will be explained in more detail later on.

The third thread is the *getFeatures* thread, that frequently queries the Sick laser scanner for the current laser data, and tries to identify landmark features from the data. It uses the shared data system to notify the EKF update functions about the presence of new landmarks. The residuals and covariances of detected lines are also calculated here, for use by the update thread in the structural compass implementation.

The fourth and final thread that is started is the update thread. The structural compass resides in this thread, and the standard landmark-feature-based update function also resides in this thread. The shared data is monitored for flags set by the *getFeatures* thread that indicate the presence of a good line for the structural compass, or solid landmark features for the standard update step.

More details will be provided about all of these threads and their accomplishments in later sections of this document. We start first with the most basic of features, the *sickNavigation* thread that drives the robot around the hallways of the building.

Navigation

The navigation method used in this project was designed with three goals in mind: travel in a closed loop, avoid obstacles, and avoid entering doorways. The robot used laser range-finder data and ultrasonic range-finder data to achieve these goals.

Traveling in a closed loop was accomplished by consistently taking right turns when possible. By examining the laser data, the navigation algorithm was able to detect continuous open spaces and direct the robot toward the midpoint of the right-most such area.

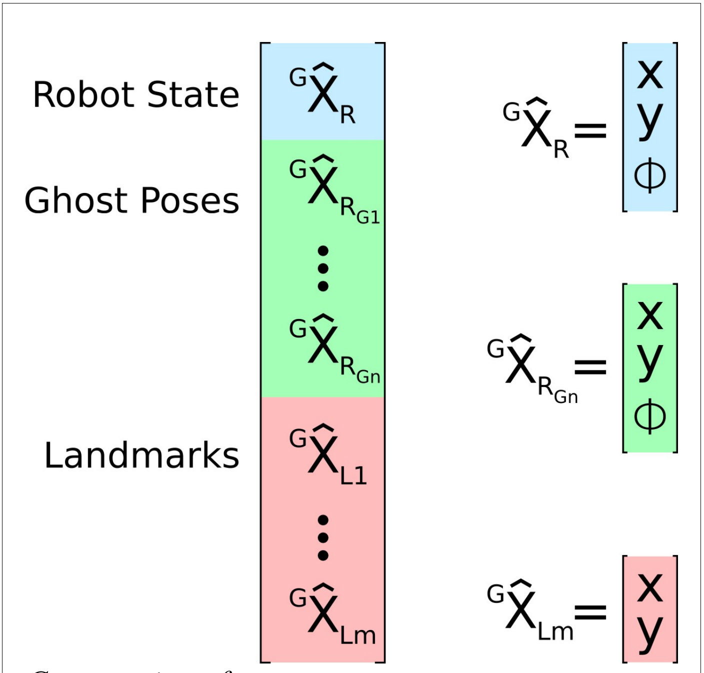
The goal of avoiding obstacles was met by a simple strategy. If the robot came too close to an object, the robot would slow down and turn until the object was no longer in the path of the robot. In practice, this allowed the robot to avoid damaging itself by colliding with unexpected interferences.

If the robot were to enter an office or lab through a doorway, it may not be able to exit the room without operator intervention. Thus, it was important for the robot to avoid entering doors. When the navigation code detected an open area from the laser data, the robot would only be instructed to move toward the area if its cross-section was larger than a doorway.

Ghosts

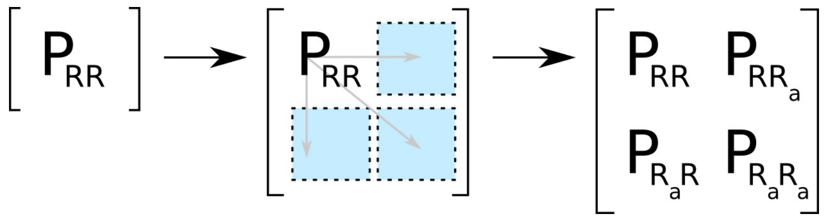
The creation of a quality map is an important part of the SLAM process. There are a few options pertaining to the creation of a suitable map, with varying degrees of accuracy and computational complexity. The easiest way would be to simply log all the laser data points at every time-step. This produces a large amount of data, but has a few issues. When the state estimate is not very accurate, simply recording the estimate when the data was captured is not good enough. As the robot observes the world, it can make better estimates of where it is, and where the landmarks are. If we don't apply this improved knowledge to our saved laser scans, then they will be very much misaligned when we go to plot them all. A better option is to simply plot the estimated position of all the landmarks from the state vector at the end. While this uses the EKF to the fullest potential to provide the most accurate estimates of the landmarks, all we can plot is the landmarks themselves. In our situation, the landmarks are the corners of the building's walls. There is no information available about where the walls are if we only plot the landmarks' estimated position.

A much improved way to produce a map is to combine these two approaches. We want to use updated position estimates, as well as the large amount of data available from the laser scans. We create what we call 'ghost' landmarks in the state vector. These ghosts are basically just invisible landmarks that are never matched with detected landmark observations. Since they are fully-qualified members of the state vector and covariance matrix, they benefit from the update step's reduction in uncertainty and covariance. When we store a ghost pose, we also save a copy of the current laser data snapshot. The position of this snapshot is not fixed to a particular pose estimate, but rather to a specific ghost pose, that is continually refined and improved through the course of the exploration.

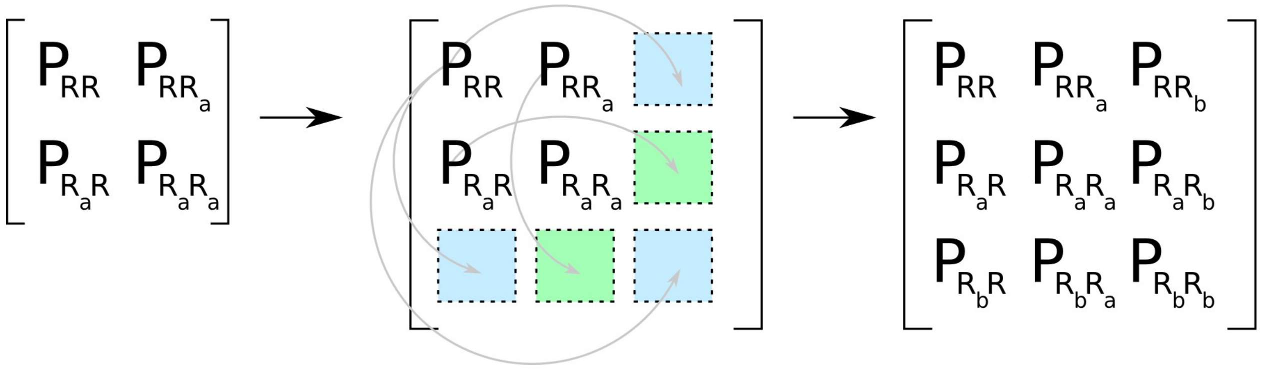


Construction of state vector

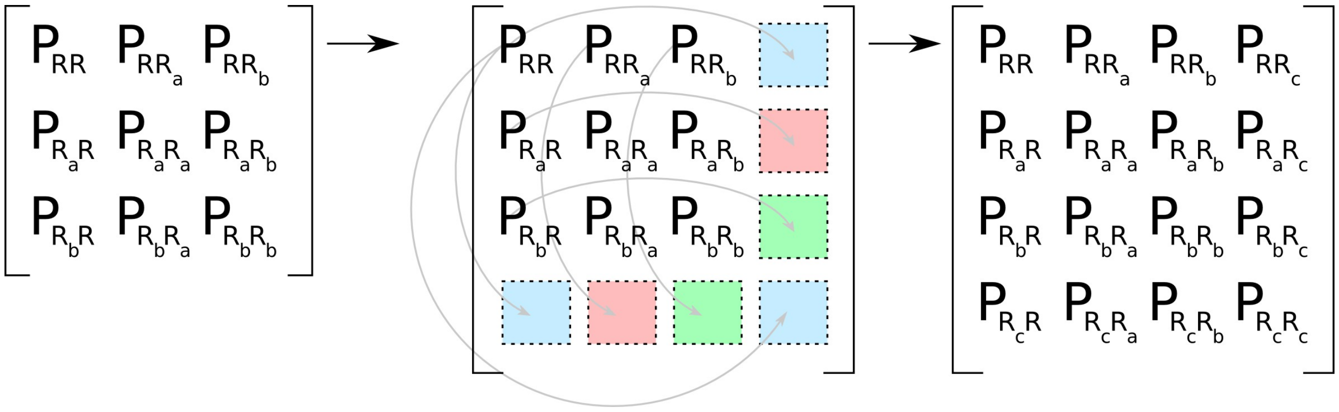
One small difference between ghost poses and true landmarks is the number of parameters that characterize each of these types. A ghost pose consists of an *x*, *y*, and *phi* parameter. A landmark consists of just an *x* and *y* coordinate pair, as our landmarks have no orientation. This has an effect on our propagation and update steps, as we need to be extra careful with the matrices. Another tricky area is determining how to change the state vector and covariance matrix when adding a new ghost pose. For the initial state, before we have seen any landmarks or stored any ghosts, the operation is simple, and is shown on the next page.



Here, P_{RR} is the covariance of the robot's state with itself. When we duplicate the state to create a ghost pose, we can simply copy P_{RR} to the other three positions in the new augmented covariance matrix. In this section, we will be using lowercase letters as subscripts to denote a ghost pose. Here is an illustration of adding a ghost pose where there is already a ghost pose present:

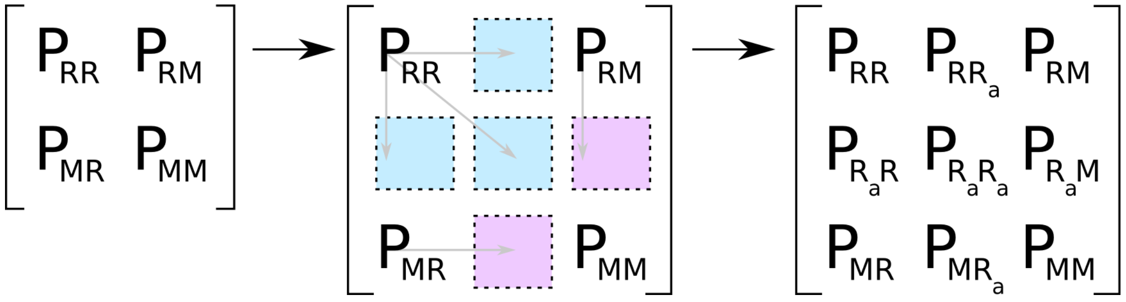

P_{RR} P_{RR_a}
P_{R_aR} $P_{R_aR_a}$
P_{RR} P_{RR_a}
P_{R_aR} $P_{R_aR_a}$
P_{RR} P_{RR_a} P_{RR_b}
P_{R_aR} $P_{R_aR_a}$ $P_{R_aR_b}$
P_{R_bR} $P_{R_bR_a}$ $P_{R_bR_b}$

As we can see, the P_{RR} sub-matrix is copied to the new blocks located at (3,3), (1,3) and (3,1). This is justified because, for example, the block matrix at (3,1) is the covariance between the third state and the first. Since, at this point in time, the third state (the new ghost pose) is exactly the same as the first state (the current actual state), we already know the covariance between these two, and it is P_{RR}. To calculate the other two blocks, the green blocks, we realize that P_{R_aR} is the covariance between R_a and R, which is the same as between R_a and the new R. If we extend this idea one more time, to adding a ghost pose when there are two existing ghost poses, we have fully generalized the case of adding a ghost pose where there are existing ghosts, and no landmarks present:



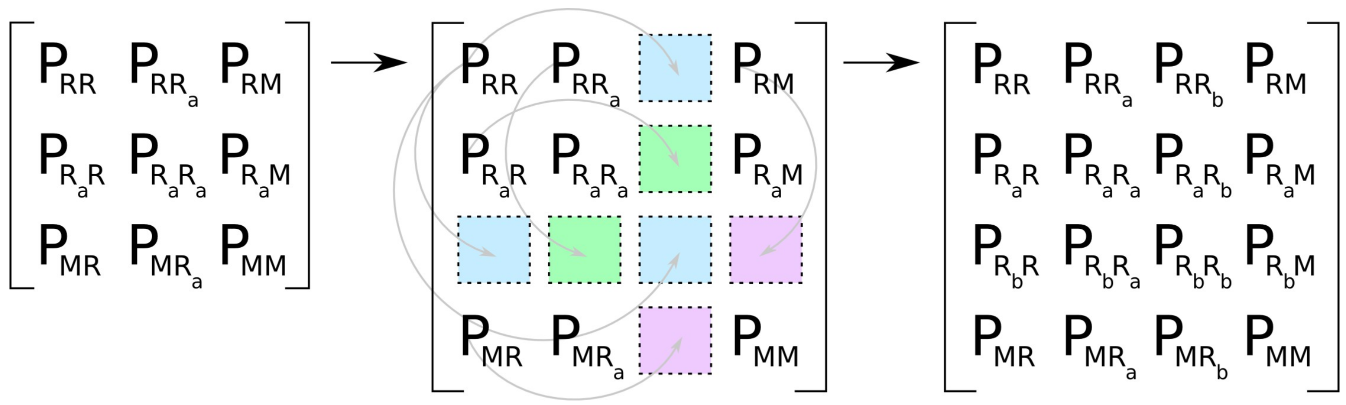
P_{RR} P_{RR_a} P_{RR_b}
P_{R_aR} $P_{R_aR_a}$ $P_{R_aR_b}$
P_{R_bR} $P_{R_bR_a}$ $P_{R_bR_b}$
P_{RR} P_{RR_a} P_{RR_b} P_{RR_c}
P_{R_aR} $P_{R_aR_a}$ $P_{R_aR_b}$ $P_{R_aR_c}$
P_{R_bR} $P_{R_bR_a}$ $P_{R_bR_b}$ $P_{R_bR_c}$
P_{R_cR} $P_{R_cR_a}$ $P_{R_cR_b}$ $P_{R_cR_c}$

To complicate matters, we also need to handle the case where there are landmarks present in our state vector and covariance matrix. The ghost poses need to fit between the current pose estimate and the landmark position estimates. In Matlab, this is relatively straightforward, but when using C++, it is much trickier to do the required matrix slicing and insertions. A few illustrations will help explain the situation. First again is the case of no existing ghost poses, with existing landmarks:

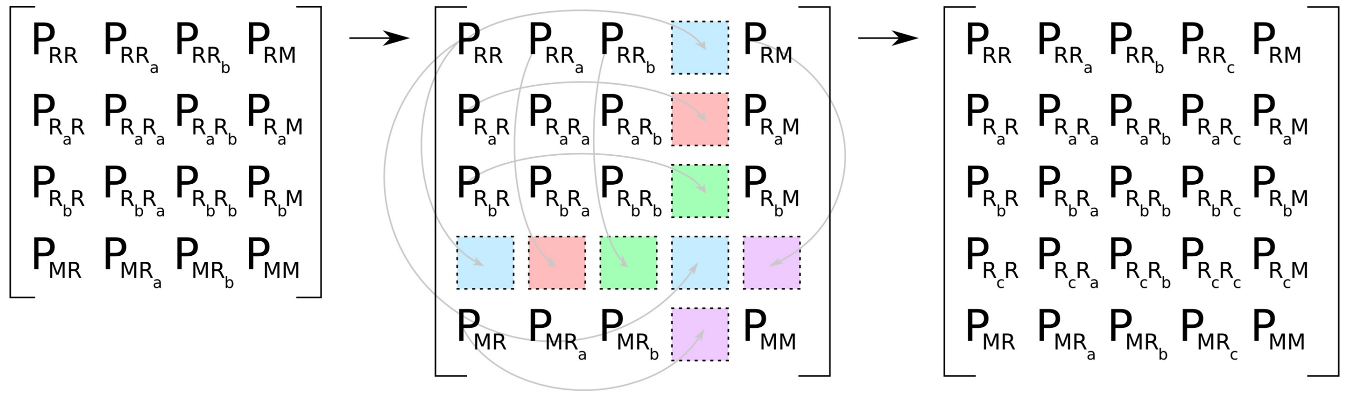


P_{RR} P_{RM}
P_{MR} P_{MM}
P_{RR} P_{RR_a} P_{RM}
P_{R_aR} $P_{R_aR_a}$ P_{R_aM}
P_{MR} P_{MR_a} P_{MM}

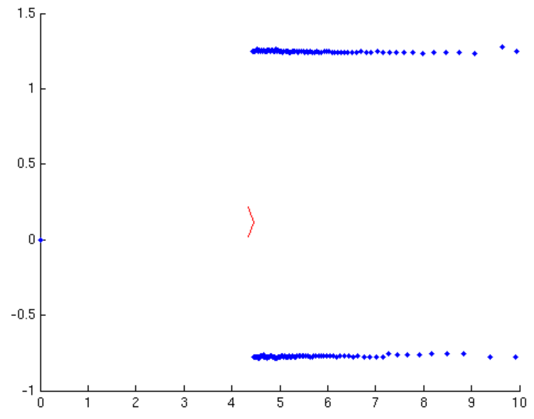
The blue block positions are simply copies of the P_{RR} block matrix as before. For the rest of these steps, the changes in the current pose and ghost sections are the same as above, the only interesting addition is how to deal with the map information. We can simply copy P_{MR} and P_{RM} to their new homes in the purple blocks. P_{MM} is simply copied over intact. Here are the extensions to one and two existing ghost poses:



Notice how P_{MR} and P_{RM} are copied over in much the same way as the previous example.

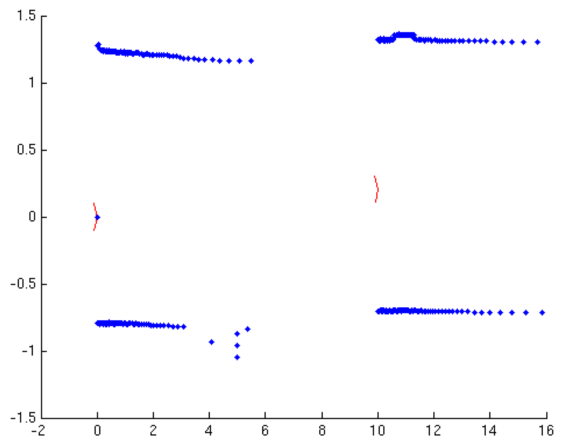


After implementing the addition of ghost poses to the system, a question soon arises. How often should we add a ghost pose? The answer to the question is based on a number of factors, including desired map resolution and the computing resources available on the robot. This figure shows what a single saved set of laser data looks like, with the robot drawn as a red arrow. The laser data points are drawn as blue dots on the plot. We can see that this data is from when the robot was facing directly down the hallway. We can even see how the laser data is more dense on the walls near the robot, where the laser beams are closer together, compared to the distance between laser readings further down the hall. We want to time the creation of ghost

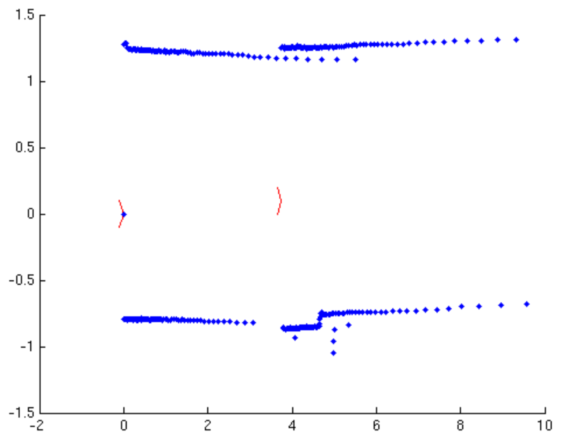


Plot of single laser data capture

poses to ensure that our laser data plots will overlap between successive ghost point snapshots. If we wait for too long between snapshots, we will get the type of image shown in the upper figure, where the laser data has gaps in it. There might be a hallway or doorway hiding in the gap in laser data. The data plot in Figure 3 was generated with 30 seconds between the creation of a new ghost pose. A much better value was found to be 15 seconds, which produced the second plot on this page in the lower figure, where the laser data is just barely starting to thin out when the next laser data is plotted. This is the value we chose to use for our real-world trials and experiments.

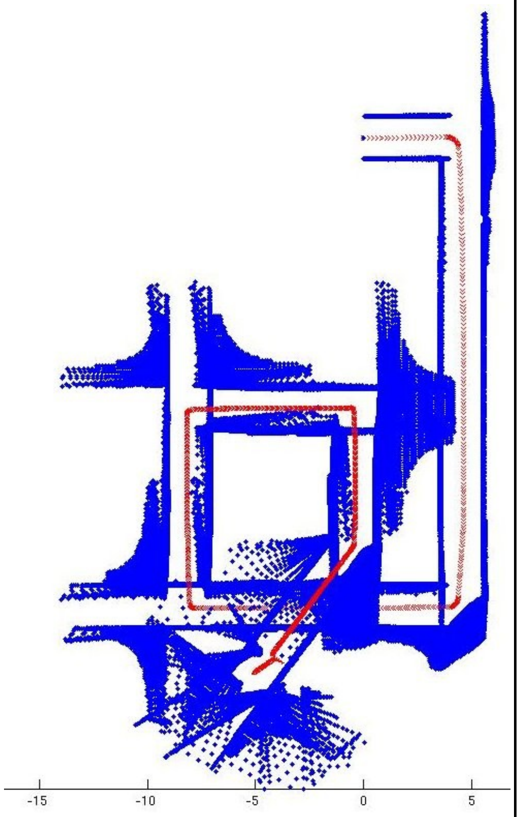


Ghost poses that are too far apart



Ghost poses that are a proper frequency

Regardless of all the optimizations of the propagation and update equations we performed, adding a large number of ghost poses to the system will definitely slow things down. We found in our experimentation that if we created ghost poses at a rate of 1 Hz, then the resulting slowdown in our propagation equations caused the propagation thread to propagate at a slower rate, effectively missing many of the details and magnitudes of the robot's motion. We saved a screen capture of a debugging data plot for one of these cases, and it is shown in the figure on the next page. It is easy to see how while the structural compass has kept the robot's heading accurate, the distance traveled is severely underestimated because of the reduced frequency of the propagation step caused by excessive processing delays from all the ghost poses in the state vector and covariance matrix.



Propagation error due to computational slowdown

Propagation

Propagation is the process of using the kinematic model of the robot in concert with odometric measurements to predict the future state of the robot and the corresponding uncertainty in that prediction. Although the state of the landmarks and ghosts are not propagated, the covariances are. However, since the covariance of the landmarks/ghosts relative to the landmarks/ghosts does not change, we can employ the following optimization:

$$\begin{bmatrix} P_{RR} & P_{RM} \\ P_{MR} & P_{MM} \end{bmatrix}_{k+1|k} = \begin{bmatrix} \phi_{R_{k|k}} P_{RR_{k|k}} \phi^T_{R_{k|k}} + G_{R_{k|k}} Q G^T_{R_{k|k}} & \phi_{R_{k|k}} P_{RM_{k|k}} \\ P_{MR_{k|k}} \phi^T_{R_{k|k}} & P_{MM_{k|k}} \end{bmatrix}$$

Initially, the propagation step was run at 20Hz. However, as the state vector got larger, the time it took to perform the propagation also increased. In fact, it eventually got to the point where there was not enough time to perform the propagation step in the time alloted. This led to a number of errors and caused the covariance matrix to increase to unreasonable values. As such, it was necessary to reduce the frequency of the propagation step to 10Hz, and take measures to ensure that the 10Hz rate was maintained, such as restricting the creation of ghost poses to every 15 seconds.

Update

Updates were broken up into two parts: structural compass updates and landmark updates. The structural compass update determined the bearing of the walls and exploited the nature of the environment (90 degree corners) to update the bearing of the robot. The landmark updates were somewhat more difficult. One of the first steps was the detection of interesting and reliable features. Once these were detected, the Mahalanobis distance was used to determine if this was a new landmark or a previously detected landmark. New landmarks are initialized using the standard equations provided in class. Previously detected landmarks are used for an EKF update. Additional information on the update step is provided in the following sections.

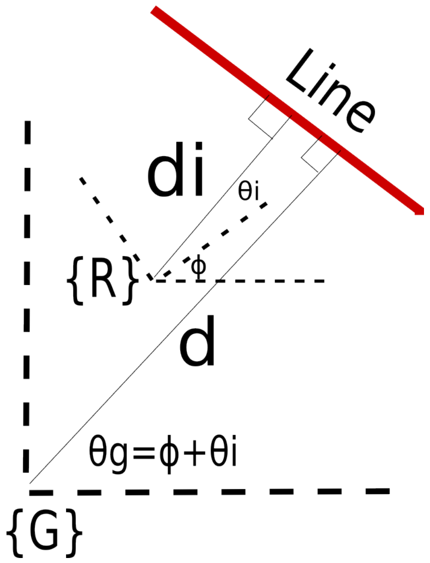
Structural Compass

In order to minimize the effects of the small angle assumptions made in the propagation and update steps, the heading of the robot must remain accurate. In order to ensure this accuracy, the robot utilizes a structural compass to correct its orientation errors.

The structural compass uses the lines found from the feature detection thread and assumes that these lines correspond to the walls of the building. The compass chooses the longest line and calculates the global orientation, θ_g.

$$\theta_g = \phi + \theta_R$$

where ϕ is the current orientation of the robot with respect the global frame and θ_R is the orientation of the line with respect to the robot. This is illustrated in the figure at right.



θ_g is expected to be within .09 radians of a multiple of pi/2. If it isn't, this means that the filter has failed or the line corresponds to a wall that is not a multiple of pi/2 radians.

Thus, if the θ_g is not within .09 radians of a multiple of pi/2, the second longest line will be used and so forth.

The next step is to map θ_g between zero and pi so that the residual can be calculated. This is accomplished using the following formula.

$$\theta_g = \theta_g - \pi * floor(\frac{\theta_g}{\pi})$$

Once θ_g has been properly mapped between zero and pi, there are one of three different residual possibilities. If θ_g is near zero, then the residual is θ_g minus zero. If θ_g is near pi/2 radians, then the residual is θ_g-pi/2. Finally, if θ_g is near pi, then the residual is θ_g minus pi.

Once the residual is calculated, the covariance of the measurement, θ_g, must be calculated. Since the lines are found using the Aria line finder method, Lidar data points must be matched to the line that was used to calculate the residual. This is done by calculating the perpendicular distance between the chosen line and all of the Lidar data points within the scan. The points that are within 10 millimeters of the line are considered to be points that belong to the line. Once all of the corresponding points have been found, the formulation of the state to measurement equation is done the in the following manner.

$$z_i = h(x_i) = d - d_i \cos(\theta_g - \theta_i)$$

z_i is an inferred measurement while d_i and θ_i are the polar coordinates of one of the points that have been associated with the line. Differentiating with respect to d and θ yeild the following H matrix.

$$H_i^T = \begin{bmatrix} 1 & d_i \sin(\theta_g - \theta_i) \end{bmatrix}$$

The covariance of the inferred measurement is found by differentiating the measurement function with respect to d_i and θ_i to obtain the relationship between the errors in d_i and θ_i and the inferred measurement.

$$\psi(i)^T = \begin{bmatrix} -\cos(\theta_g - \theta_i) & -d_i \sin(\theta_g - \theta_i) \end{bmatrix}$$

This matrix is then multiplied by R from the left and the right. R is the covariance of the range and bearing, which is characterized by the sick data sheet.

$$R_n(i) = \psi(i) R(i) \psi(i)^T$$

A least squares method is now employed to calculate the covariance of the state.

$$R_s = \left(\sum \left(H_i^T R_n(i)^{-1} H_i \right) \right)^{-1}$$

Since, this is a structural compass, the components that involve the distance standard deviation are not relevant. Thus, the $R_s(2,2)$ element is used for the covariance and is fed into the update portion of the structural compass along with residual that was calculated earlier. From now on the $R_s(2,2)$ element will be denoted by R_c.

Now that the residual, r, and covariance, R_c, have been calculated, the structural update can be completed. Along with r and R_c, compass update algorithm takes in the state covariance, P, the state vector, X, which is comprised of the robot state vector, X_r, ghost states, X_g, and state map, X_m. The structure of the state and covariance are shown below. The outputs are $X_{(k+1)}$ and $P_{(k+1)}$.

$$X = \begin{bmatrix} X_R \\ X_G \\ X_M \end{bmatrix}$$

$$P_{(k)} = \begin{bmatrix} P_{RR} & P_{RG} & P_{RM} \\ P_{GR} & P_{G.G} & P_{GM} \\ P_{MR} & P_{MG} & P_{MM} \end{bmatrix}$$

The following measurement equation relates the states to the measurement.

$$z = \theta_g - \phi + n_\theta$$

Differentiating with respect the provides the H_R.

$$H_R = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix}$$

Since, the measurement model is independent of X_g and X_m, the Jacobian of the entire system is the following.

$$H=\begin{bmatrix} H_R & 0 & 0 \end{bmatrix}$$

The calculation of the covariance of the residual is rather easy because most of the entire state vector is zero and S is a scalar.

$$S=HPH^T+R_c=H_R P_{RR} H_R^T+R_c$$

The Kalman Gain is calculated in the following manner.

$$K=\begin{bmatrix} K_R \\ K_G \\ K_M \end{bmatrix}=\begin{bmatrix} P_{RR} H_R^T \\ P_{GR} H_R^T \\ P_{MR} H_R^T \end{bmatrix} S^{-1}$$

The next step is to calculate the state vector at time step k+1.

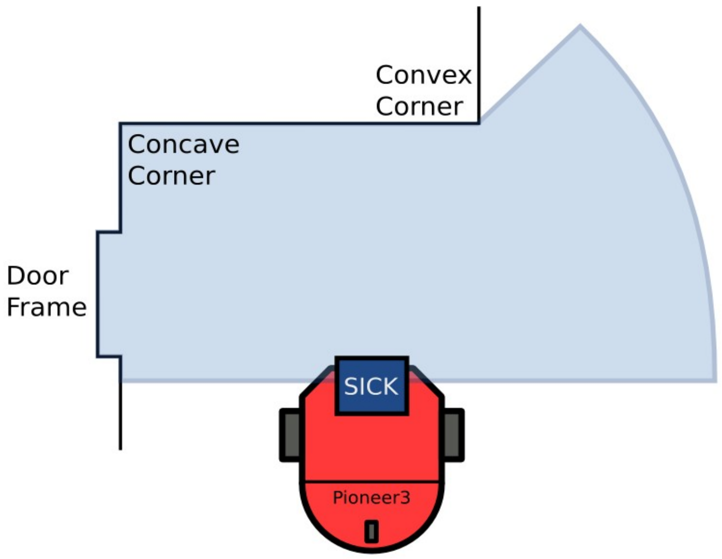
$$X_{(k+1)}=X_{(k)}+Kr$$

Once the state vector has been calculated, the covariance at time step k+1 is calculated.

$$P_{(k+1)}=P_{(k)}-P_{(k)} H^T S^{-1} H\, P_{(k)}=P_{(k)}-\frac{1}{S}\begin{bmatrix} P_{RR} H_R^T H_R P_{RR} & P_{RR} H_R^T H_R P_{RG} & P_{RR} H_R^T H_R P_{RM} \\ P_{GR} H_R^T H_R P_{RR} & P_{GR} H_R^T H_R P_{RG} & P_{GR} H_R^T H_R P_{RM} \\ P_{MR} H_R^T H_R P_{RR} & P_{MR} H_R^T H_R P_{RG} & P_{MR} H_R^T H_R P_{RM} \end{bmatrix}$$

Feature Detection

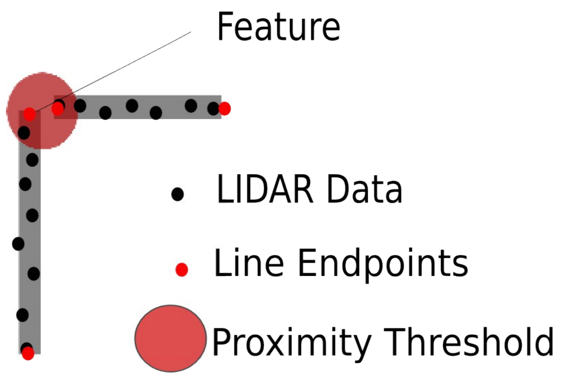
The feature detection thread locates three different types of features. These include concave corners, convex corners, and door frames. The figure to the right shows the different types of features the robot can detect. The feature detection thread takes in Lidar data and outputs an array of feature objects, which is a class that includes x and y position,



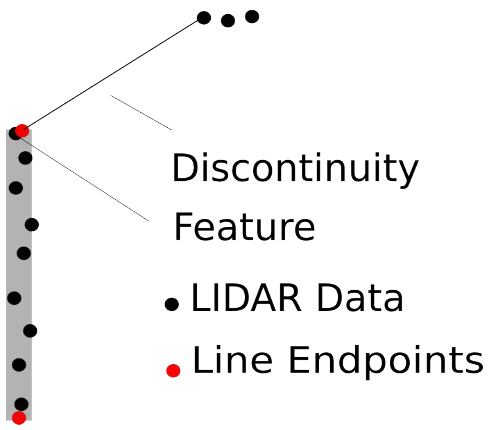
covariance, and a boolean that denotes whether a corner is convex or concave.

The first step of the feature detection thread is to find all of the lines that are associated with the Lidar data. In order to do this, the Aria line finder is invoked. The appropriate thresholds are set such that the minimum line length is one-third of a meter, and the minimum number of points that can make a line is 5. The next step is to find the corners associate with the Lidar data. Corners are found using two different methods.

The first method involves looking for perpendicular lines with nearby end points. The graphic to the right shows two perpendicular lines with endpoints that are within a proximity threshold from one another. This threshold is set at 200 millimeters. The feature itself is the Lidar scan point that is closest to both line end points.

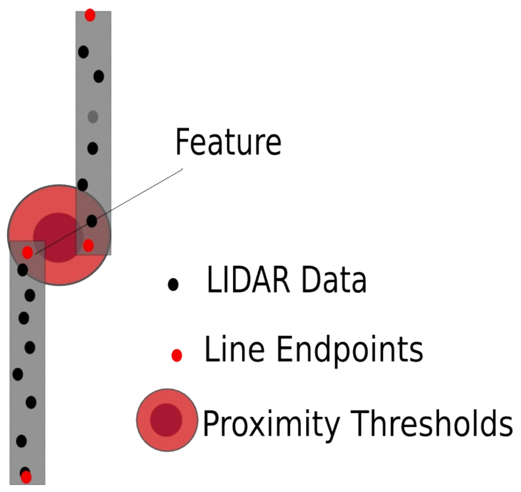


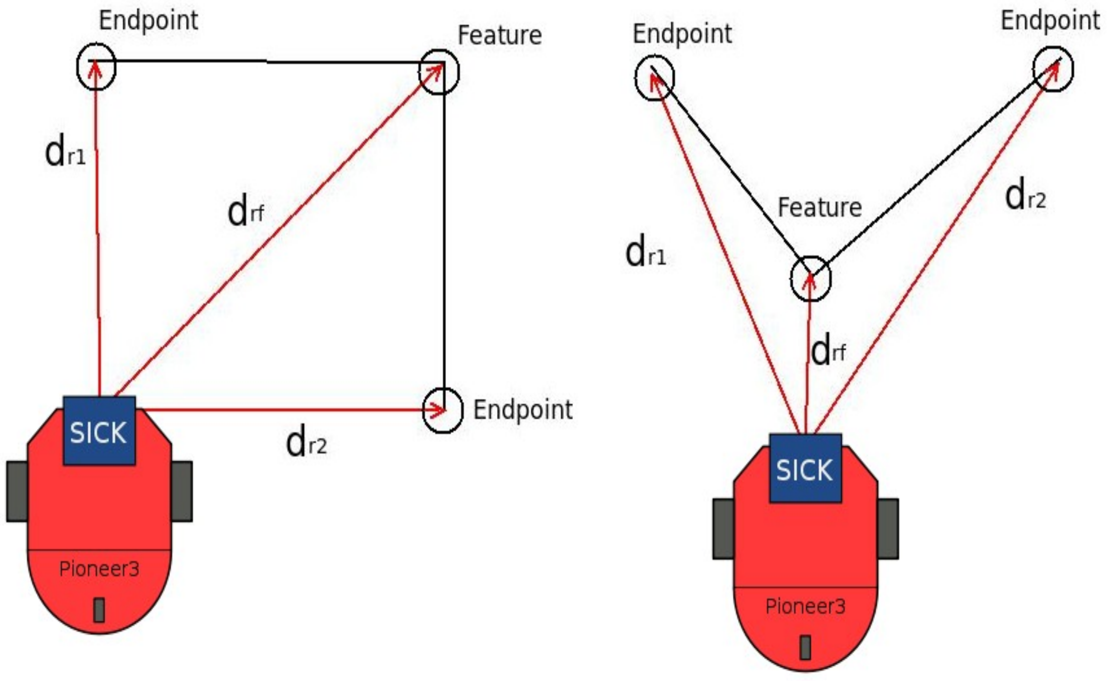
Often times, the robot is only able to detect one line that is associated with concave corners. In order to detect these corners the thread loops through all of the lines and looks for an endpoint that corresponds to a sharp discontinuity in Lidar data. The feature detected is the Lidar scan point that is closest to the endpoint of the line that corresponds to the gap in the data.



Door features are found by looking for parallel lines that have endpoints near one another. This is done in almost the exact same fashion as the perpendicular line corner finder described before. However, in order to avoid noisy feature extraction, a double sided threshold is used. If the line endpoints are separated by more than 75 millimeters but less then 150 millimeters, the feature detection thread will mark the Lidar scan point that is closest to the endpoint that is closest to the robot. An example of this is shown in the figure to the right.

In order to make the matching easier within the update thread, the features are marked with boolean flags indicating whether a feature is a door frame, convex corner, or concave corner. Upon detecting features with the lone endpoint to Lidar discontinuity method, the feature thread automatically assumes that this feature is a convex corner. If the feature is detected using the parallel line method, the feature is considered a door frame. However, if the feature is detected the using the perpendicular line method, more investigation is needed to see if the feature is a convex or concave corner. This is done by analyzing the distances between the robot and the line endpoints. The figure below shows the robot detecting a feature via the perpendicular line method in two different scenarios.





The first scenario on the left is the robot seeing a convex corner while the second scenario is the robot looking at a concave corner. The feature thread calculates d_{r1} and d_{r2}, which are the distances between robot and the endpoints of the lines that do not correspond to the feature. Then, d_{rf} is calculated which is the distance between the robot and the landmark. If d_{rf} is less than both d_{r1} and d_{r2}, then the feature is considered concave; Otherwise, it is considered convex.

Once the features have been extracted, the covariances must be calculated. From the data sheet we know the covariance of our features with respect to distance and bearing. This covariance will be called R, which is shown below. The Sick LMS 200 data sheet stated that the standard deviation of the distance measurement, σ_d, was 5 millimeters, while the standard deviation in bearing, σ_Θ, was .01 radians.

$$R=\begin{bmatrix}\sigma_d^2 & 0\\ 0 & \sigma_\theta^2\end{bmatrix}$$

In order to map the covariance from local polar coordinates to local Cartesian coordinates, we must multiply R by the following matrix from the left and right.

$$G=\begin{bmatrix}\cos(\theta) & -d\sin(\theta)\\ \sin(\theta) & d\cos(\theta)\end{bmatrix}$$

$$R_p=GRG^T$$

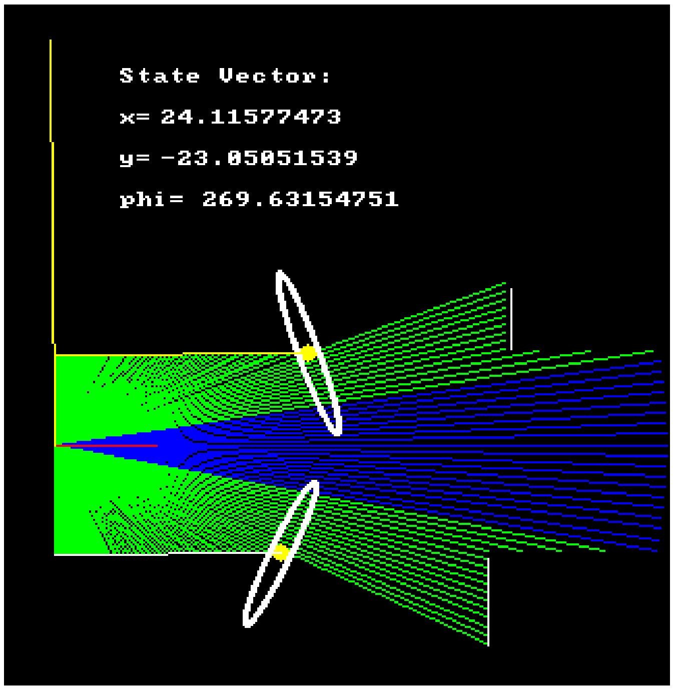
R_p along with the local Cartesian coordinates of the feature is then fed into the update thread.

Real-Time Plotting

An example of real-time visualization of laser data is shown below. The information in the visualization is as follows:

- Red = current destination vector
- Yellow = structural compass lines, convex landmarks

- Blue = obstacle-free areas
- Green = terminated range measurements
- White = landmark covariance ellipses, remaining detected lines

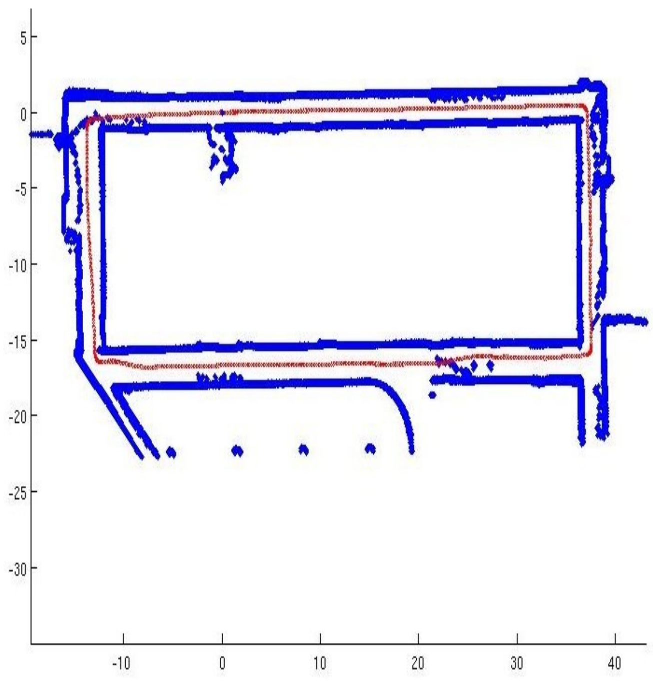


```
State Vector:
x= 24.11577473
y= -23.05051539
phi= 269.63154751
```

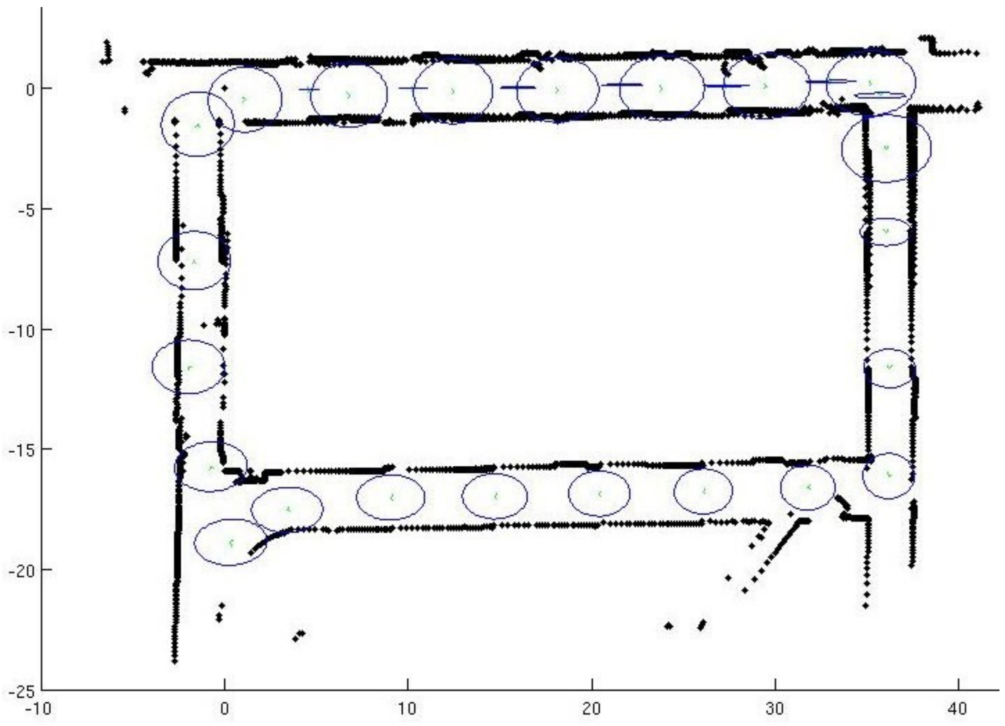
The real-time visualization was produced with the QuickCG graphics library. In the visualization, the robot X axis is always horizontal, and the Y axis is always vertical.

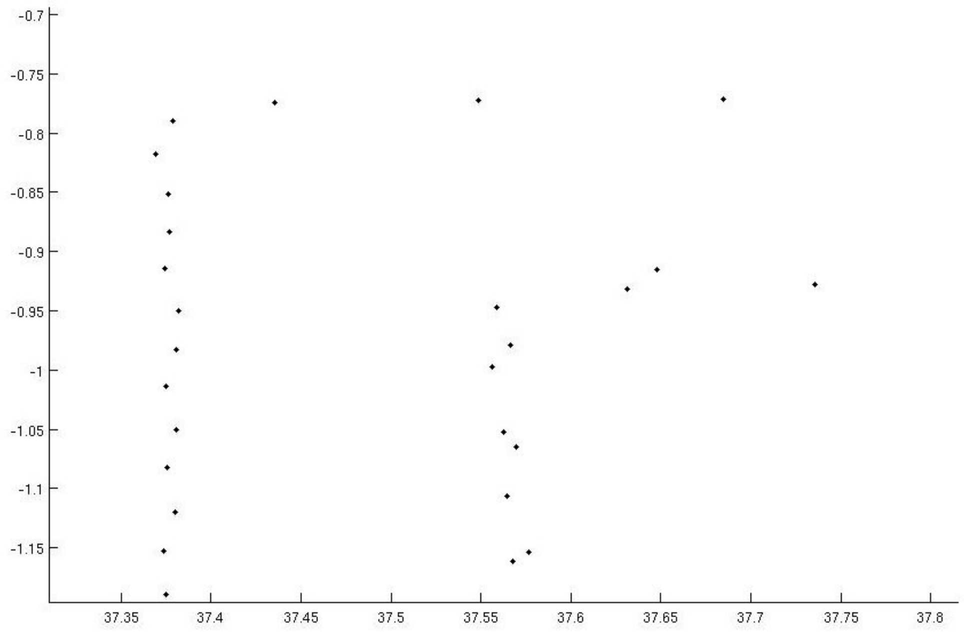
Map Rendering

This project recorded estimated robot poses and laser data “snapshots” at regular intervals. The robot pose estimates were corrected by means of a structural compass measurement, discussed previously. After the robot had finished moving in each experiment, the corrected estimates and their associated laser range-finder readings were written to disk in a human-readable format. In order to verify the correctness of the robot's pose estimates, a Matlab script was written to plot the saved laser data from its associated robot pose estimate. An example of a resulting plot is seen here, which represents the laser data measured by the robot during a single lap of a rectangular building section. The red trails correspond to the path of the robot, while blue dots correspond to detected obstacles, including walls, railings, and passers-by.



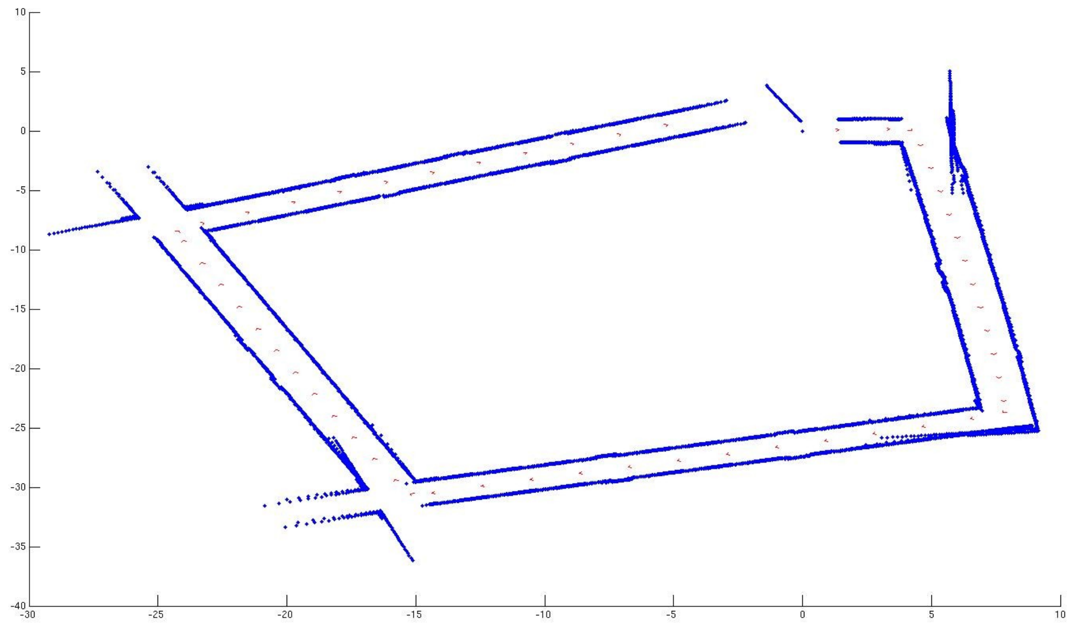
In another test, the robot moved one and a quarter loops around a rectangular section, resulting in the figure below. A close-up of one of the corners in the map that was measured twice is seen on the next page. After traveling approximately 130 meters, the error in the robots pose estimate was only ~20cm in the X and Y directions. This corresponds to an accumulated error of 0.15% of the total distance traveled.




Close-up of laser scan data

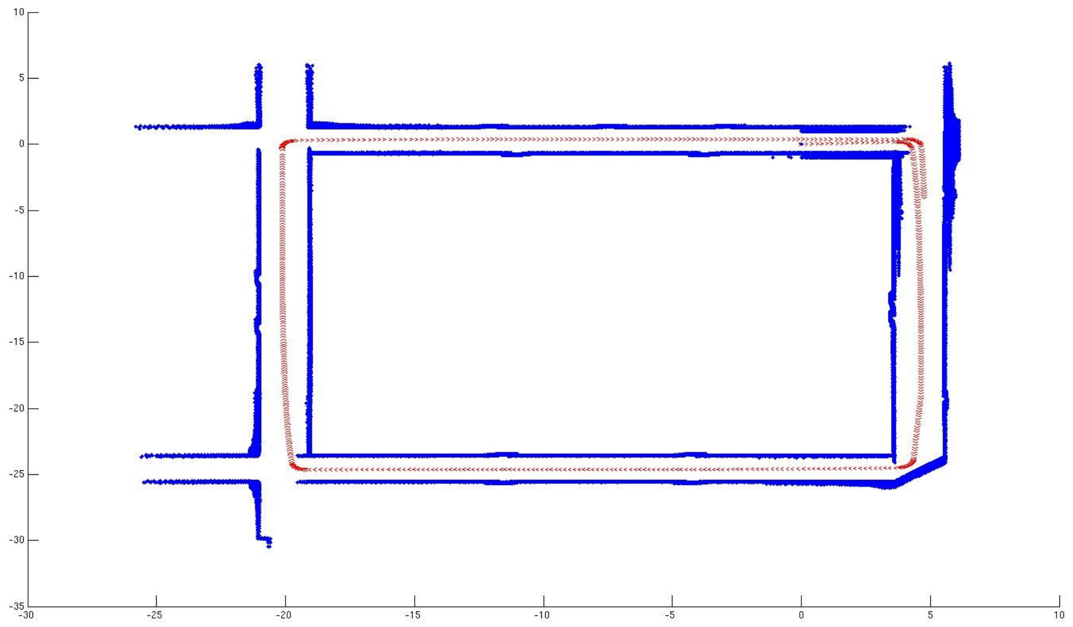
Results:

The EKF propagate function produced reasonable results. The map is shown below. Although the map somewhat resembles the actual environment, there is clearly a great deal of distortion caused by the inaccurate heading estimates.

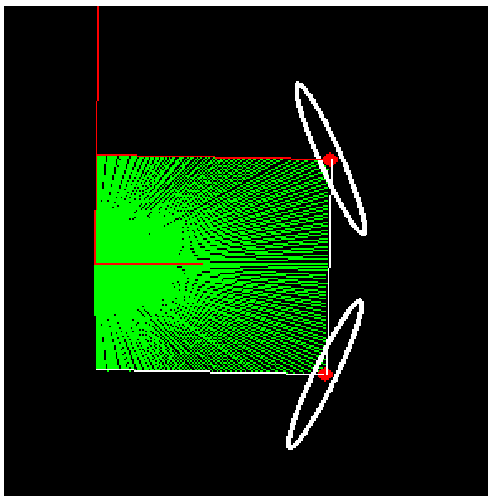


Map created using propagation only

The structural compass does an excellent job of eliminating this distortion. The map below closely resembles the actual environment. All corners are multiples of 90 degrees, which is to be expected in a well-constructed indoor environment.

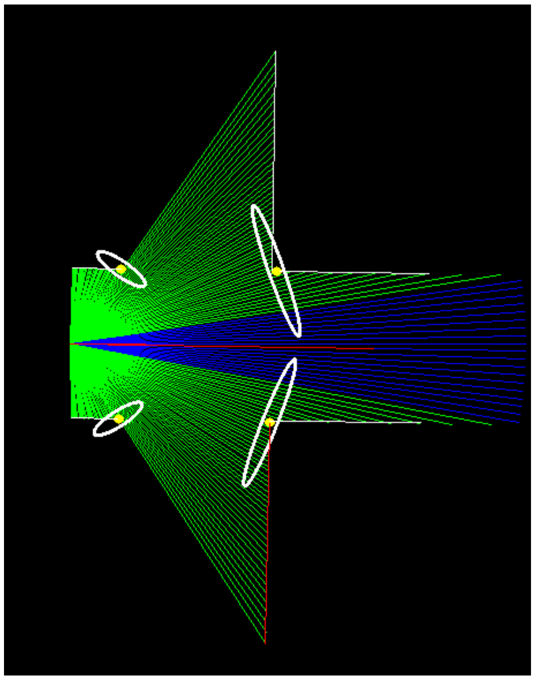


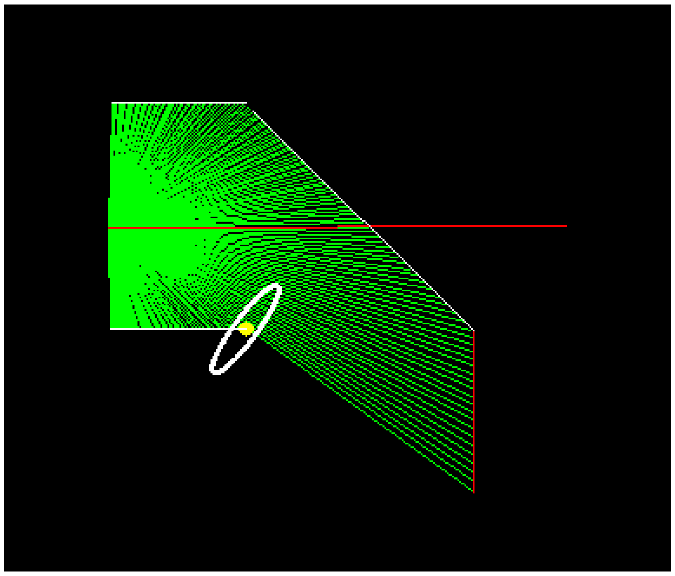
Map created using structural compass updates and propagation

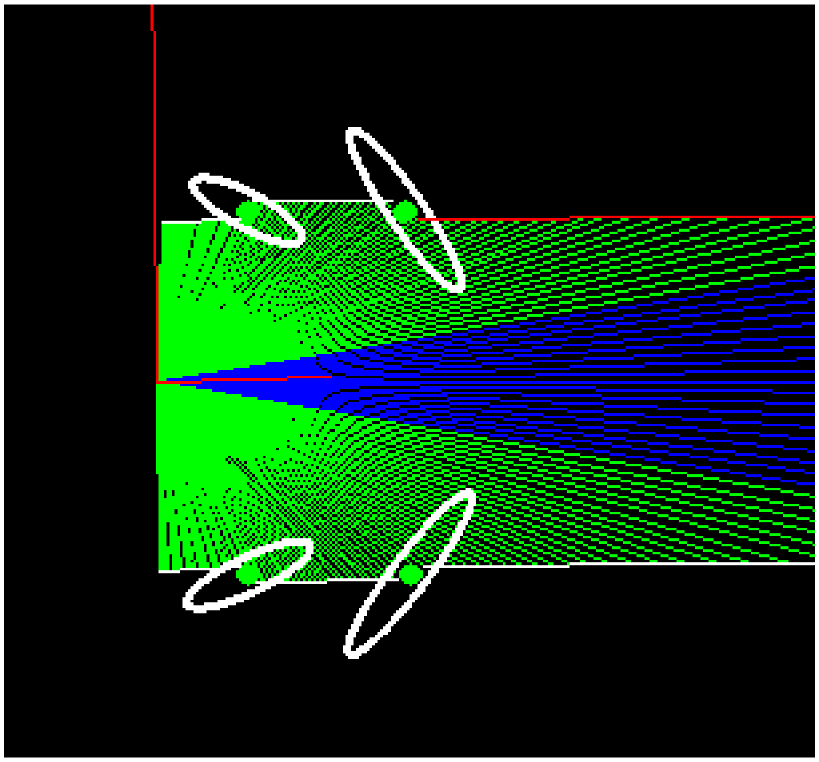


Concave Corners (red dots)

The feature detection does a reasonably good job of detecting features. It can detect both convex and concave corners, along with doors. However, from time to time our feature detection would pick up a spurious landmark. A possible cause of this is the highly reflective metal plates on the bottom of some of the doors.


Convex Corners (yellow dots)


The structural compass ignores walls that aren't at 90 degree angles


The feature detector can detect doors

Conclusion

The goal of this project was to create a robot capable of solving the 2D-SLAM problem. Although a true SLAM implementation did not fully come to fruition, the robot can be placed in an unknown indoor environment and generate a reasonably accurate map. Refined feature detection and a debugged EKF update step should, in theory, provide a working solution to the 2D-SLAM problem. Gamma values may need to be tweaked to ensure proper loop closure. Once the 2D-SLAM problem is solved with laser scan updates, it would be interesting to implement it with camera updates. A single camera can only obtain bearing measurements, and it is inherently noisier than a laser scanner, but the substantially lower cost makes this an option worth exploring.

Bibliography

[1] J. Leonard and H. Durrant-Whyte, "Simultaneous Map Building and Localization for an Autonomous Mobile Robot" IEEE IROS 1991, Nov. 3-5

[2] R. Smith and P. Cheeseman, "On the Representation and Estimation of Spatial Uncertainty" International Journal of Robotics Research, 1986

[3] S. Thrun, W. Burgard, and D. Fox, "A Probabilistic Approach to Concurrent Mapping and Localization for Mobile Robots" Autonomous Robots 5, 253–271, 1998

[4] S. Riisgaard and R. Blas, "SLAM For Dummies (A Tutorial Approach to Simultaneous Localization and Mapping)", http://ocw.mit.edu/NR/rdonlyres/Aeronautics-and-Astronautics/16-412JSpring-2005/9D8DB59F-24EC-4B75-BA7A-F0916BAB2440/0/1aslam_blas_repo.pdf