C# Implementation of SLAM Using the Microsoft Kinect

Richard Marron

Advisor: Dr. Jason Janet

4/18/2012

Abstract
A SLAM algorithm was developed in C# using the Microsoft Kinect and iRobot Create. Important features of this SLAM algorithm are that it successfully treats difficulties associated with moving humans and anomalous returns from reflective surfaces in the environment. This paper describes how the software interfaces with the Create, interfaces with the Kinect, detects spike landmarks, associates the landmarks, and implements an extended Kalman filter.

Introduction
Simultaneous localization and mapping (SLAM) is an algorithm that allows a mobile robot to form a map of an unknown environment and locate itself within this map. Such an algorithm is useful in any situation where a human wants to understand an environment but access to the environment is limited. Applications include ocean surveying, oil pipeline inspection, mine exploration, crime scene investigation and military operations. [1]

The problem of a SLAM algorithm, simply stated, is that in order to create a map of the environment, the robot's location must be known, and in order to know the robot's location, a map must be created. Several solutions to this problem exist; one common form is the following:

1) Move robot
2) Observe environment
3) Detect landmarks
4) Associate new landmarks with previously observed landmarks
5) Estimate the robot's motion from the distance between associated landmarks
6) Correct the new landmarks' position for the estimated robot's motion
7) Add corrected landmarks to the map and repeat

In order to implement such an algorithm three pieces of hardware are necessary: a mobile robot, a means to observe the environment and a computer. Of these essential pieces of hardware, the means to observe the environment is the most specific to SLAM. Observing the robot's environment can be implemented in several ways. Examples are sonar, monocular cameras, and laser range finders. Laser range finders provide the most accurate observations but have historically been the most expensive until the advent of the Microsoft Kinect. With the Microsoft Kinect it is now possible to get an array of depth data (640 x 480) that is accurate to the millimeter for $150. With this recent development, several open source SLAM algorithms have been created implementing the Kinect.

The motivation for this project stems from frustrations when using previously developed SLAM algorithms that use the Microsoft Kinect for environment observation. Often, these algorithms are poorly documented, computationally expensive, fail in environments with reflective surfaces, do not account for moving humans, require large image processing libraries and utilize unreliable libraries to connect with the Kinect. The goal for this project was - to rectify these shortcomings.

A well-documented SLAM algorithm will be created that (1) uses the Kinect for Windows SDK, (2) can operate in standard indoor environments with reflective surfaces and moving people, and (3) utilize no external visual processing libraries.

Design Architecture

With the above goal in mind, a SLAM algorithm was created in C#. The general architecture, seen in Figure 1, consists of three main parts: the Kinect, the IRobot Create and the SLAM algorithm. The SLAM algorithm recieves information describing the environemnt from the Kinect, combines this information with the robot's estimated position from dead reckoning and executes the algorithm. The computer then sends a motion command to the IRobot create and repeats the algorithm.

Main Algorithm

The main algorithm is executed in a constant loop and completes the 7 steps for SLAM outlined in the introduction. The first step is to move the robot. After the robot has moved to its new location, dead reckoning is used to estimate the distance and angle traveled by the robot giving an estimate of the robot's location.

Next, a 2D slice of data from the Kinect's depth information is taken and any interference from people or reflective surfaces is accounted for. This depth information is passed into an algorithm that locates protruding features known as spike landmarks. After all landmarks have been located, these locations are translated to account for the estimated robot motion from dead reckoning. For example, if a land mark is located at (50mm, 100mm, 150mm) with reference to the robot, but the robot has moved to (100mm, 100mm, 100mm) with reference to the origin, the landmark's position will now be (150mm, 200mm, 250mm).

Following this translation of landmarks, the Spike landmarks are associated with existing Spike landmarks by determining if any previously observed landmarks lay within a given radius of the newly found landmarks. If two landmarks are within a certain radius, then they are considered the same and are associated. The error between new landmarks and their associated existing landmark along with the estimated robot position are fed into an extended Kalman filter (EKF) to determine the robot's actual position. The newly found landmarks are then updated with the robot's actual position and the process is repeated.

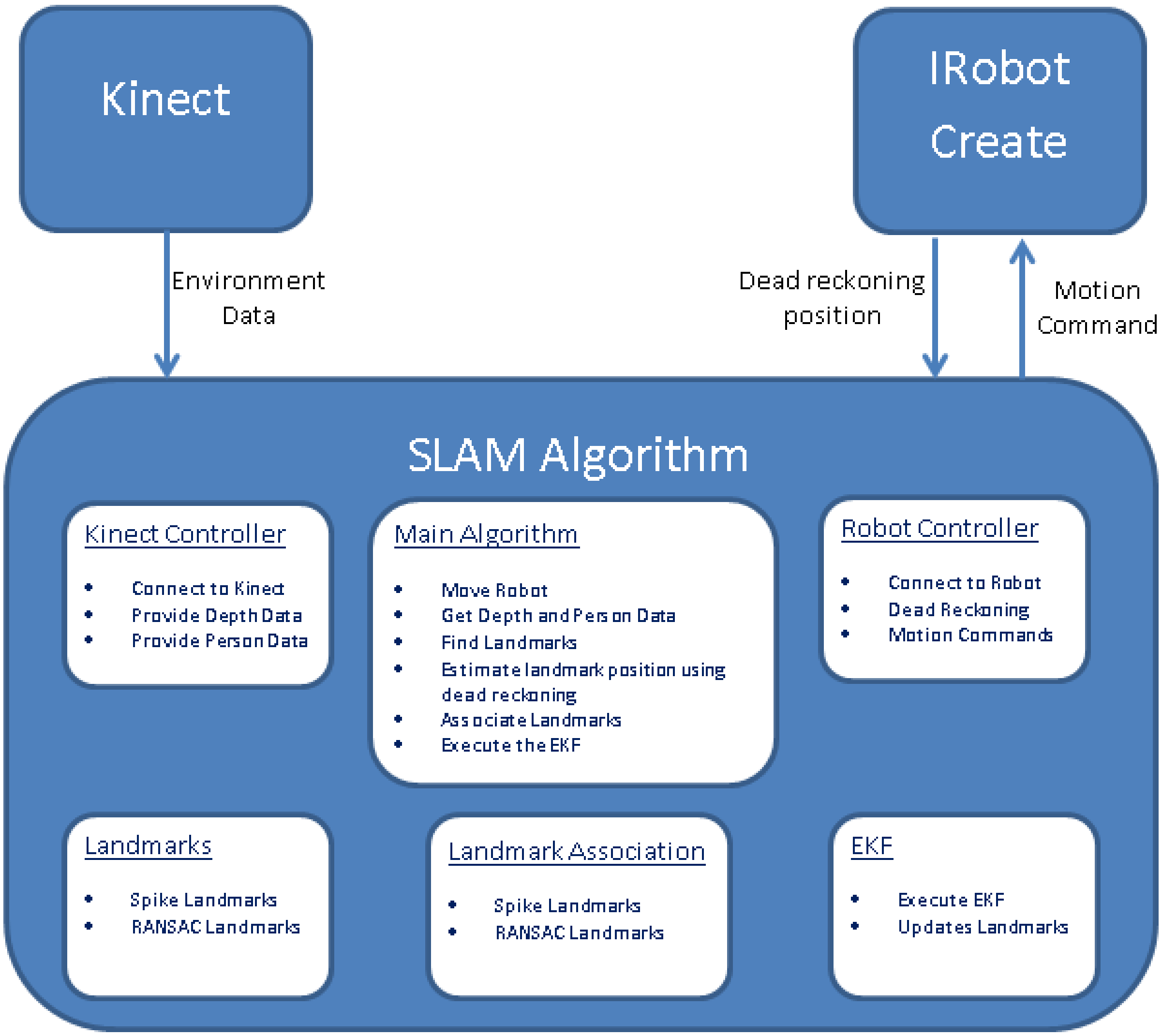

Figure 1: High level system architecture

Robot Controller

The robot driver requires four main methods: one to connect to the robot, one to move the robot forward, one to rotate the robot, and one to retrieve the robot's dead reckoning information. A slightly modified version Kevin Gabbert's CreateOI library [2] was used to communicate with the IRobot Create. This library uses serial communication to both talk with the robot and get sensor information from the robot. The serial codes sent to the robot can be found in the iRobot Create Open Interface document. [3]

Every time the sensor information is queried from the Create, values for distance traveled and angle rotated since the last sensor query are returned from the robot's wheel encoders. Previously, the CreateOI library was not storing this information, and as a result no aggregation of the distance and angle information was being computed. After some modification of the code, these values are now being aggregated thereby creating an estimation of the robot's position.

Kinect Controller

The Kinect Driver has three main methods: one to connect to the Kinect, one to get the location of people in the frame, and one to get the scaled depth information.

The method to connect to the Kinect uses the Kinect for Windows SDK to establish a connection to the Kinect. If there are any problems with this connection, the console window will report them. After establishing a connection, it enables the depth data and skeleton data streams. Following this step, the Kinect's motors are used to ensure the Kinect is parallel to the Robot. Finally, an event handler is set up to wait for the event that all of the Kinect frames are ready. When this event occurs, it means that the Kinect has received a depth data frame, a color image frame, and a skeleton frame simultaneously. In the case of this event, the Kinect driver stores this information in a series of arrays one for the depth image, one for the color image, and one for the skeleton points.

The method to receive the location of people in the frame uses the information stored in the depth array. Each pixel in the depth array encodes both the depth information and the player that the depth pixel belongs to. By applying a bitmask to this piece of data, one can extract the player information. The algorithm that Microsoft implements in its SDK returns the location of humans at 200 FPS. It is able to obtain such high FPS by using a randomized decision forest that was trained by tens of thousands of human images. [4]

When determining how the depth data should be fed to the SLAM algorithm four factors had to be considered: dealing with reflective data, dealing with human data, scaling the data, and cutting down the size of the data.

When using the Kinect for Windows SDK, any reflective surfaces will return a depth value of -1. This signal provides an easy way to distinguish "bad" data. While the current implementation of SLAM ignores this bad data, it is not removed because a future landmark detection algorithm may use this data in determining landmarks.

For the issue of human data, because human data is always a dynamic aspect of the depth information, it cannot be used as a landmark and is simply removed.

For the issue of data scaling, the raw depth data is returned as a pixel number and its associated depth value in millimeters. The issue comes when converting the pixel location to millimeters as it is not a trivial problem. Fortunately, the Kinect for Windows SDK provides a method for this conversion to millimeters.

Finally, for the issue of cutting down the amount of data, a horizontal slice of the depth information is taken. For example, from the array of 640x480, only the data points where the height is 200 may be used in the algorithm. Such a slice can be seen in Figure 2. By reducing the data, the SLAM problem simplifies from a 3D slam algorithm to a 2D algorithm, and the computational intensity is greatly reduced.

Figure 2: Example of taking a slice of the depth data at the red line

Landmark Detection

This implementation of SLAM uses a spike landmark detection system. Spike landmarks are identified by locating pixels that protrude from their surrounding pixels. These pixels often represent objects such as corners or chair legs. A spike landmark detection system was chosen because it is relatively simple and not computationally intensive at O(n). Additionally, when calibrated properly, it provides consistent landmarks independent of viewing angle. Pseudo-code for the algorithm is seen in Figure 3.

```
foreach pixel in dataSlice
{
        if the pixel is good data (not reflective data)
        {
                if the point is closer to the robot than its neighbours
                {
                        rightSlope = slope between the pixel and its closest right pixel of non-reflective data
                        leftSlope = slope between the pixel and its closest left pixel pixel of non-reflective data

                        slopeDiff = AbsoluteValue(rightSlope - leftSlope)

                        if slopeDiff > threshold value
                        {
                                Add pixel as a landmark
                        }

                }
        }

}
```

Figure 3: Pseudo-code of a spike landmark detection algorithm

Four interesting features of the above algorithm are the following. First, each pixel is iterated through once, making the algorithm O(n). This complexity is an improvement from many other common landmark extraction methods which often run O(n log(n)) at the quickest. [5]

Second, the algorithm ignores all bad data points, but if two points lay on each side of a section of bad data, it still treats these points as neighbors when looking for spikes. This precaution is taken because the Kinect often reads sharp surfaces as bad data. These bad data points are most likely a result of the IR lasers hitting the surface at a glancing angle significantly distorting the Kinect's laser array making it un-interpretable. These sharp surfaces are exactly the type of surface the algorithm is attempting to detect. If these points were ignored, many landmarks would go undetected.

Third, the algorithm ignores landmarks that are further from the Kinect than either of the landmarks neighboring pixels. This check is done to ensure the landmark being detected is the actual spike and not the surface behind the spike. A spike landmark often represents an object such as a pole or chair leg. If the algorithm were to detect the surface behind the landmark as the landmark itself, then the location of this landmark will be dependent on the viewing angle as it is a projection of the actual protruding object. This concept is illustrated in Figure 4.

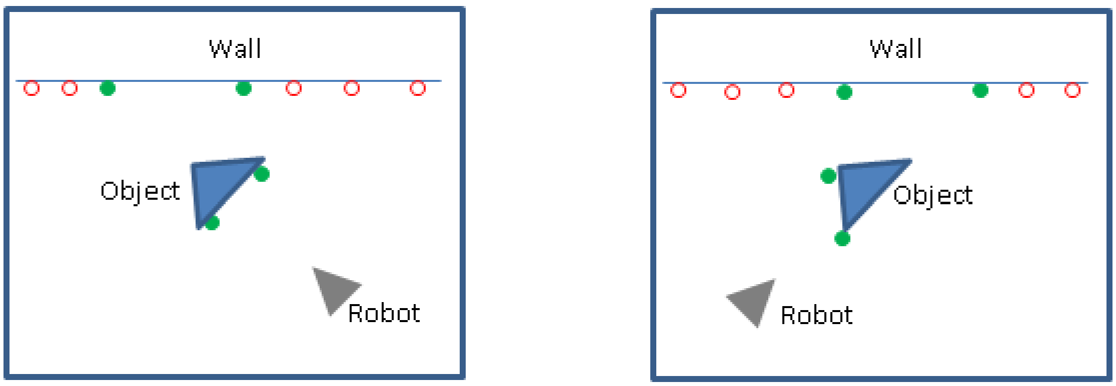

Figure 4: Example of why the protruding side of the spike landmark must be used. The red circles represent the pixels, and the green pixels represent detected landmarks. As it can be seen, the landmarks on the object remain consistent with the object's position, but the landmarks behind the object shift along the back surface.

The fourth interesting feature of this spike landmark detection algorithm is that it uses the difference in slope between the central pixel and both of its neighboring pixels as a means to detect spikes. If the algorithm used only slope values and not the difference in slope values to detect landmarks, then any point on a wall that is nearly 90 degrees to the robot would be considered a landmark. This potential mistake is a result of the slope between the central point and the neighboring points being large.

Once a landmark has been detected, it is translated to an estimation of the original SLAM reference frame. This translation is done by shifting the pixel by the robot's position as calculated by the previous iteration of SLAM combined with the data from robot's dead reckoning system.

Landmark Association

The next step of SLAM is to associate these newly found landmarks with previously observed landmarks that are believed to be the same. This step brings up one drawback of a simple spike landmarks detection algorithm. Because the only characteristic of the spike landmark is the position, only the position can be used to associate landmarks. Many times two different landmarks that are in close proximity will be associated in error.

The algorithm used to do this association simply iterates through each new landmark and determines if a landmark exists within a certain threshold radius of itself. If landmarks exist within this threshold radius, then the new landmark is associated with the closest previously existing landmark. This threshold radius is chosen based on two system variables. The first is the error of the dead reckoning system and the second is the error of the laser range finder. The error of the dead reckoning system is often a drifting error and is therefore dependent on the distance traveled by the robot. The error of the laser range finder is a constant error. This radius can be modeled by Equation 1 where r is the radius threshold; k is a constant value proportional to the dead reckoning translation error; dX is the distance traveled by the robot; m is a constant value proportional to the dead reckoning rotational error; $d\theta$ is the angle rotated by the robot, and l is the laser range finder error. In the system with the Create and Kinect, l is multiple orders of magnitude smaller than $k * dX + m * d\theta$ and is therefore ignored.

$$(1)\ r = k * dX + m * d\theta + l$$

Through experimentation, it was found that a value of k =.1 and m =.02 were appropriate for the specific Create robot used. Because r is able to expand indefinitely as the robot moves, the value was capped at .3m.

EKF

After the new landmarks have been associated with existing landmarks, the SLAM algorithm uses an extended Kalman filter (EKF) to correct the robot's position based on the error between associated landmarks. While this paper is not the proper location to derive the EKF, the specific version implemented follows very closely with that discussed in Søren Riisgaard and Morten Rufus Blas's *SLAM for Dummies.* [6]

One interesting modification of this SLAM algorithm is that the Kalman gain matrix, K, which determines how much the SLAM determined position should be trusted versus the dead reckoning position had to be significantly skewed toward the SLAM position to give accurate results. This modification is a testament to the Kinect's accuracy and the consistency of the landmark detection and association.

Results

The results from the SLAM algorithm show a strong proof of concept for the above described architecture and implementation.

Interpreting Kinect Data

For the interpretation of the Kinect data, the detection of humans and reflective surface were both successful. Figure 5 shows an example of manipulated depth data from the Kinect. In this figure, color represents the depth and white represents removed data. As it can be seen, both the human in the scene and the two reflective windows in the background have been detected and removed.

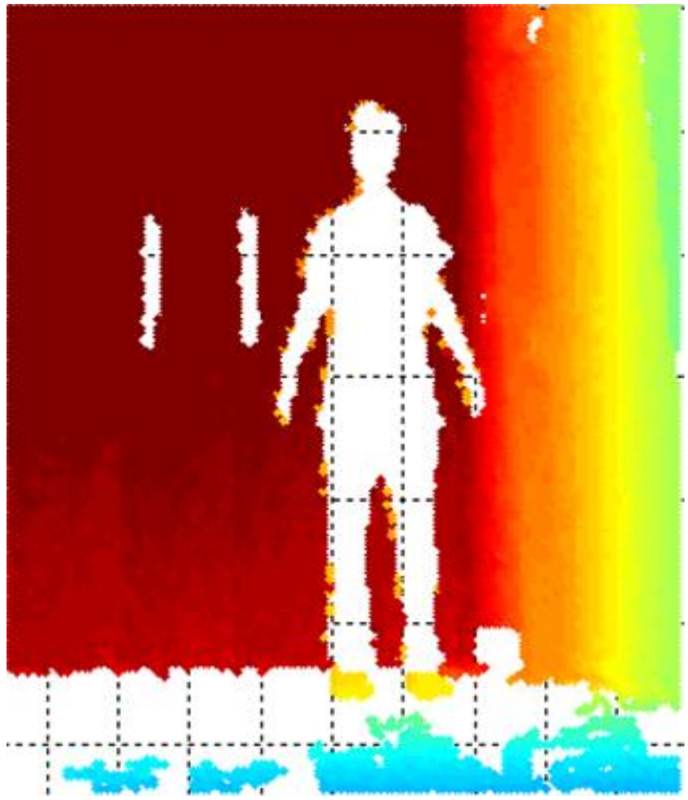

Figure 5: A graph illustrating the ability to remove humans and reflective surfaces from a scene. The color represents the depth.

Landmark Detection

A good spike landmark detection system is characterized by consistency. If the robot views the same landmark from different angles and positions, it should be re-identified as a landmark. In Figure 6, a series of observations are seen. Each observation is the robot viewing the same three spike landmarks from different angles and positions. The left landmark was detected in 5 of the 6 frames; the central landmark was detected in 5 of the 6 frames, and the right landmark was detected in 4 of the 6 frames.

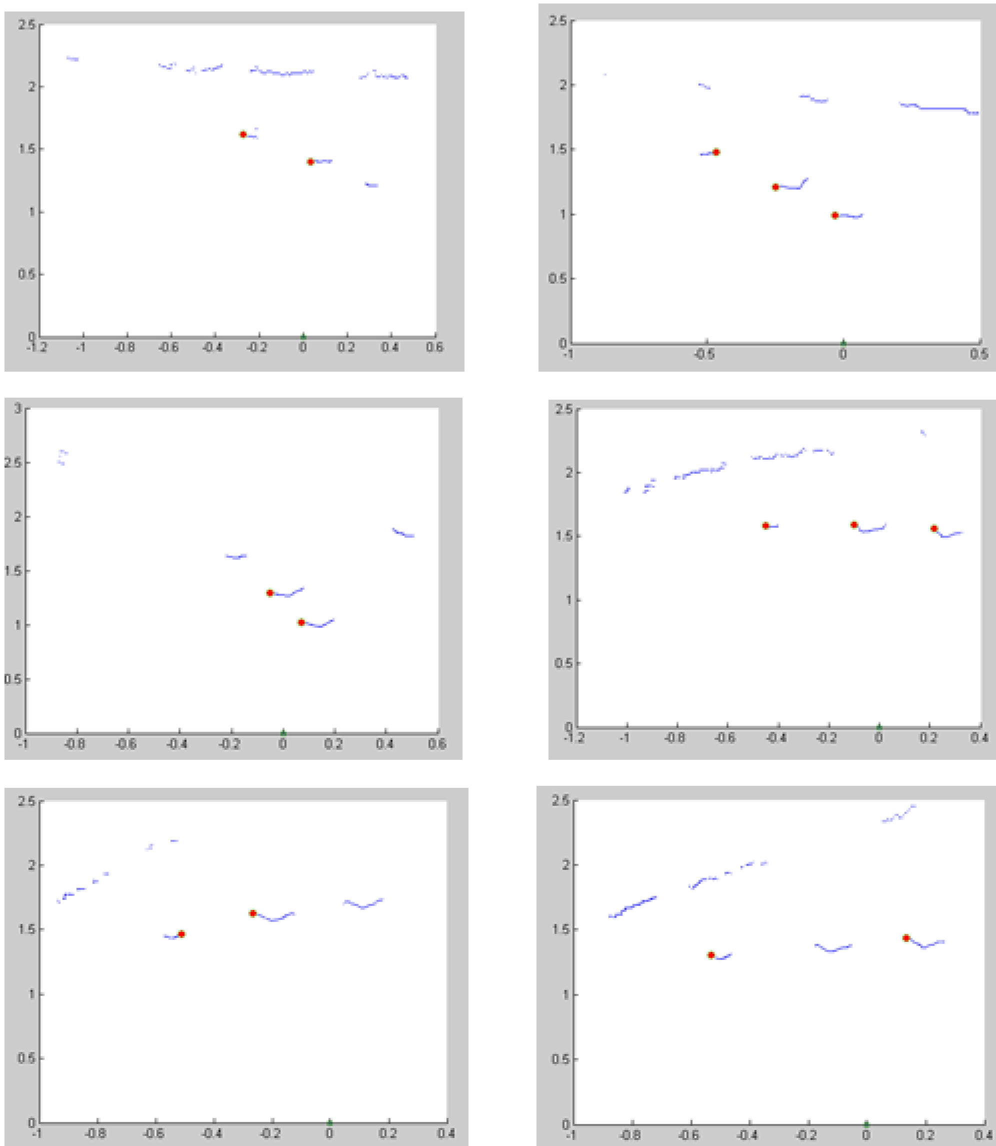

Figure 6: The same three protruding objects observed from different angles and positions. The blue lines represent observed surfaces and the red dots represent detected landmarks.

Localization

The localization aspect of SLAM was tested by moving the robot and recording the position as measured by dead reckoning, by SLAM and by measuring tape. These measurements were then compared. Figure 7 contains a graph of the robot's heading when rotated about its central axis.

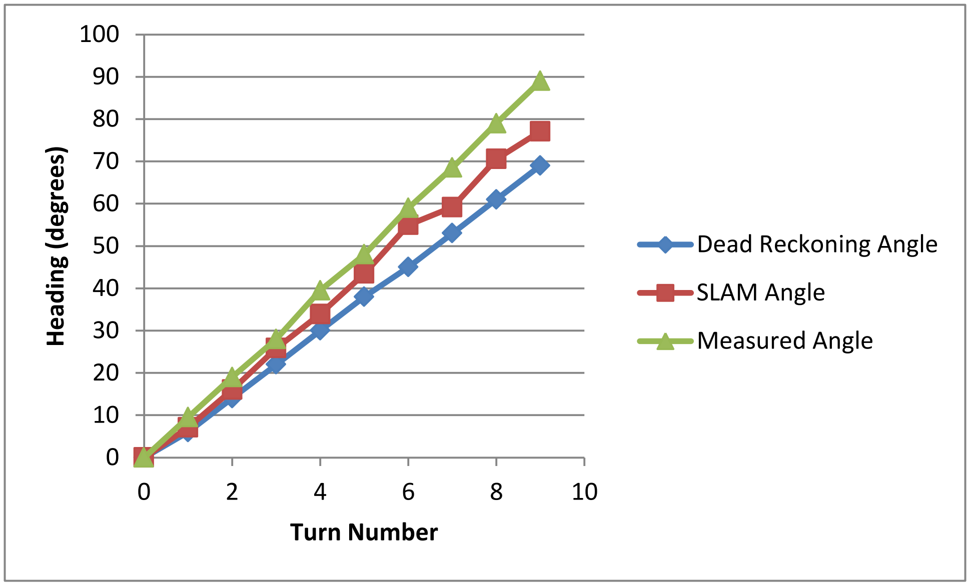

Figure 7: The robot's heading as it rotates about its axis.

The average %error between the dead reckoning position and the measured angle was 24.6% while the average percent error between the SLAM position and the measured angle was 12.9%. These results show significant improvement in localization from the SLAM algorithm.

The localization aspect of the SLAM implementation fails when robot is translated instead of rotated. There appears to be a significant, yet allusive, bug in the EKF that is causing this failure. Currently, the result of the SLAM algorithm when the robot is translated the dead reckoning position and therefore shows no improvement.

Mapping

The mapping aspect of the algorithm was tested by running the full algorithm and viewing the output. During this testing process, the robot was placed in an environment with protruding surfaces and the algorithm was started. A human also passed in front of the robot several times during the testing process. In Figure 8, the final map from the SLAM algorithm can be seen. The image shows a scene from a down-looking perspective. Each different color represents data from a different reading, and the circles represent detected landmarks. Each concentric circle around the landmark represents a new detection of that landmark. The gray dashed line shows an estimated map of the region from observed knowledge.

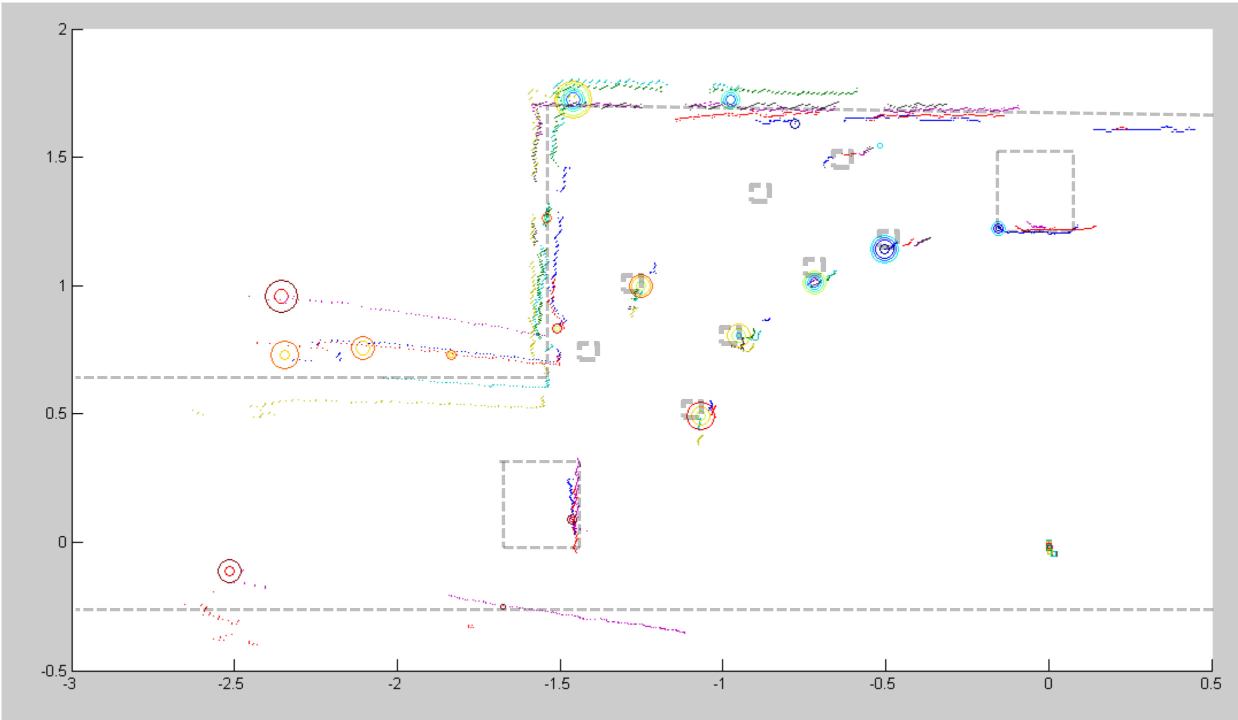

Figure 8: A map formed by the SLAM algorithm. Each color represents a different reading and the large circles represent landmarks. The gray lines represent an estimated map from observed knowledge.

As it can be seen, this map is not perfect; however it does give an understandable description of the room. There is an observable angular drift in the map that is a result of the 12.9% error seen between the SLAM angle and the measured angle.

When a similar test was run with robot translation included, the resulting map was interpretable and resembled a map created from dead reckoning information alone. This issue is due to the bug mentioned in the localization results section.

Conclusion

In conclusion, a SLAM algorithm was written in C# that uses the iRobot Create and Microsoft Kinect. This algorithm functions without the use of outside visual processing libraries (ignoring the visual processing capabilities of the Kinect) and is able to function in environments with humans and reflective surfaces.

Like all pieces of software, this implementation of SLAM still has some bugs to be fixed and features to be implemented. Remaining features to address include:

1) Fix the translation bug in the EKF
2) Incorporate a line landmark detection system
3) Add more sophisticated landmark association algorithm
4) Improve efficiency throughout

5) Aggregate SLAM from several slices of data
6) Produce a 3D map of the environment by stitching together each frame of Kinect data
7) Eliminate data from sections of the environment producing noise using the Kinect's microphone array. This improvement assumes noise is associated with movement.

SLAM algorithms currently play an important role in applications including ocean surveying, oil pipeline inspection, mine exploration, crime scene investigation and military operations. The future will certainly bring other important applications such as spacecraft and automobile navigation.

Works Cited

[1] A. Hogue, "Simultaneous Localization and Mapping - SLAM," 20 June 2005. [Online]. Available: www.cse.yorku.ca/~hogue/qual_slides.pdf. [Accessed 18 April 2012].

[2] K. Gabbert, "C# CreateOI Framework," 17 July 2009. [Online]. Available: http://sourceforge.net/projects/createoi/. [Accessed 18 April 2012].

[3] iRobot Corporation, "iRobot Create Open Interface," iRobot Corporation, Bedford, Mass, 2006.

[4] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman and A. Blake, "Real-Time Human Pose Recognition in Parts from Single Depth Images," 2011. [Online]. Available: http://research.microsoft.com/pubs/145347/BodyPartRecognition.pdf. [Accessed 18 April 2012].

[5] V. Nguyen, S. Gachter, A. Martinelli, N. Tomatis and R. Siegwart, "A comparison of line extraction algorithms using 2D range data for indoor mobile robots," 8 June 2007. [Online]. Available: http://www.springerlink.com/content/l36g683786h01px4/fulltext.pdf?MUD=MP. [Accessed 18 April 2012].

[6] S. Riisgaard and M. R. Blas, "SLAM for Dummies (A Tutorial Approach to Simultaneous Localization and Mapping)," [Online]. Available: http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf. [Accessed 18 April 2012].