Simultaneous Localization and Mapping Implementation for Navigation of an Autonomous Robot

By

MATTHEW STEPHEN FELDMAN

A THESIS PRESENTED TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF BACHELOR OF SCIENCE IN ELECTRICAL ENGINEERING WITH HONORARY SUMMA CUM LAUDE

UNIVERSITY OF FLORIDA

Fall 2014

To my family, who supported me in everything I did.

ACKNOWLEDGMENTS

I thank Dr. Scott Banks for his mentorship and support throughout my undergraduate education. Furthermore, all of my professors at the University of Florida helped to drive me towards success and becoming a better engineer. I also thank Christine Moore and Joshua Novick, the Computer Science undergraduates who played a critical role in this project by collaborating on the software and algorithm development.

TABLE OF CONTENTS

page

TABLE OF FIGURES

Page

Abstract of Thesis Presented to the Department of Electrical and Computer Engineering of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science with Honorary Summa Cum Laude

MAPPING AND LOCALIZATION IMPLEMENTATION FOR NAVIGATION OF AN AUTONOMOUS ROBOT

By

Matthew Stephen Feldman

December 2014

Chair: John G. Harris
Major: Electrical Engineering

This thesis studies the feasibility of using a vision-based implementation of Simultaneous Localization and Mapping (SLAM) for the navigation of an autonomous robot. Specifically, we outline the most important steps in using a forward-facing camera on an autonomous mobile robot to acquire the robot's position in space while accumulating a composite map of the field around it. These steps consist of perspective-transforming the input image to get a bird's eye estimate of the camera's image, detecting features of interest on this transformed image, identifying features that match on the new image and a full map of the course, and finally stitching the images together with minimal error.

The OpenCV library provides much of the support to perform all of these tasks with Python, but still presented a few shortcomings that required various algorithms to be custom-designed. Furthermore, not all of the build-in functions were compatible with other functions used in this project, so a significant amount of work was put in to smoothing the seams and building the interfaces to let the individual units communicate with each other. Random Sample Consensus (RANSAC) is one such algorithm that was re-implemented in order to better fit the needs of the University of Florida's IEEE autonomous robot for which this software project was intended.

CHAPTER 1
INTRODUCTION

All of the code and sample images relevant to this report can be found in the Python section of the University of Florida's IEEE 2015 Hardware team's github, https://github.com/ufieeehw/IEEE2015.

The Machine Intelligence Laboratory (MIL) at the University of Florida will be competing in the SoutheastCon 2015 IEEE Autonomous Robot Competition. This competition has "the intention of recreating the classical American road trip" by specifying that an autonomous robot must successfully navigate a white line and play these four classic road trip games[1]:

- Play Simon Says
- Draw "IEEE" on an etch and sketch
- Rotate one face of a Rubiks Cube
- Pick up a single playing card

Figure 1 shows a sample course layout for the competitions. The rules specify that the team of engineers has one minute to set the robot in the starting square and align each of the four props as desired in their respective squares. The robot then has five minutes to complete each of the four tasks and cross the finish line. The environment will be very noisy and contain various flashing lights and disturbances from spectators that the robots must be robust against.

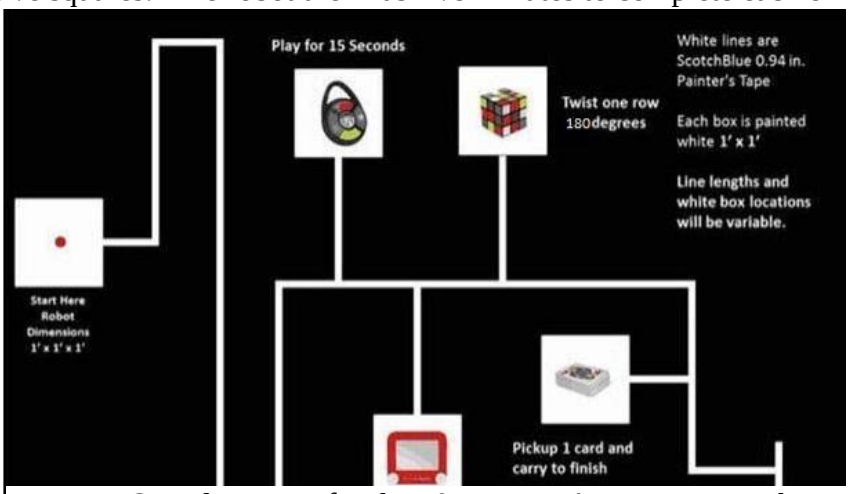

Figure 1: Sample course for the 2015 IEEE Autonomous Robot competition

Figure 2 shows the University of Florida's vehicle for this competition. The vehicle consists of four omnidirectional wheels, a four-axis robotic arm, two parallel grippers and a vacuum cup at the end effector, two video cameras, and various other sensors. To perform the high-level computations, Robot Operating System (ROS) is run on top of an Intel NUC x86_64 Mini PC loaded with Ubuntu 14.04.

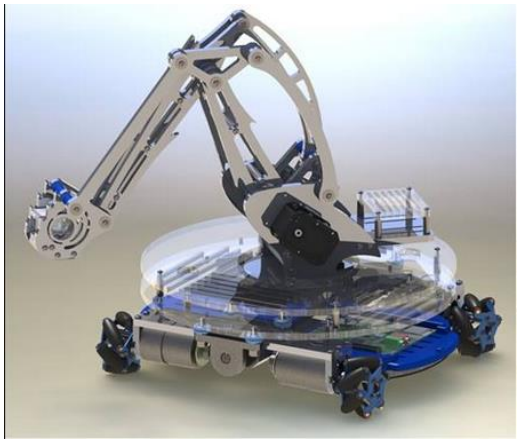

Figure 2: University of Florida's vehicle designed for the 2015 IEEE Autonomous Robot Competition

Rather than following the white lines on the course to arrive at each of the four tasks, the team decided not to follow them at all and drive straight towards the various destinations. The amount points deducted from driving off of the lines will be more than offset by the amount of points gained by completing the course more quickly. In order to assist the vehicle in determining its location and planning its path, a forward facing camera was decided to provide visual information about the course. These images are the basis for implementing Simultaneous Localization and Mapping (SLAM). At a later point in time, odometer, GPS, and compass telemetry will be fused with the data provided by the SLAM node using Kalman filtering to improve the accuracy of the robot's location.

Python was chosen instead of C++ for this project. The IEEE robot is a very large project requiring the collaboration of dozens of engineering students. Python provides a more intuitive and simple platform for students with little programming experience to get started right away on our project. In order to keep all of the code consistent, we decided to have every piece of software written in Python and adhering to PEP8 standards for comments, format, and style.

The OpenCV library in Python provides a wide variety of very convenient functions for working with images.

CHAPTER 2
TOP-LEVEL APPROACH

Figure 3 shows the general approach in the SLAM algorithm that was written for MIL's vehicle. After an initialization procedure that sets up the vehicle's starting location and blank map of the field, a series of methods fuses each new image of the course with the entire set of previous images. First, a perspective transform of the camera's image is performed in order to generate a bird's eye view of the features in the image. Then, the software runs feature detection algorithms on this image to extract an array of key points with their respective set of descriptors. Next, this array of key points is tested against a larger set of key points already detected in the full map to statistically determine which points appear in both the new image and the old map. Finally, the affine transform that rotates and translates the new image onto its appropriate location on the full map is computed and applied to update the map.

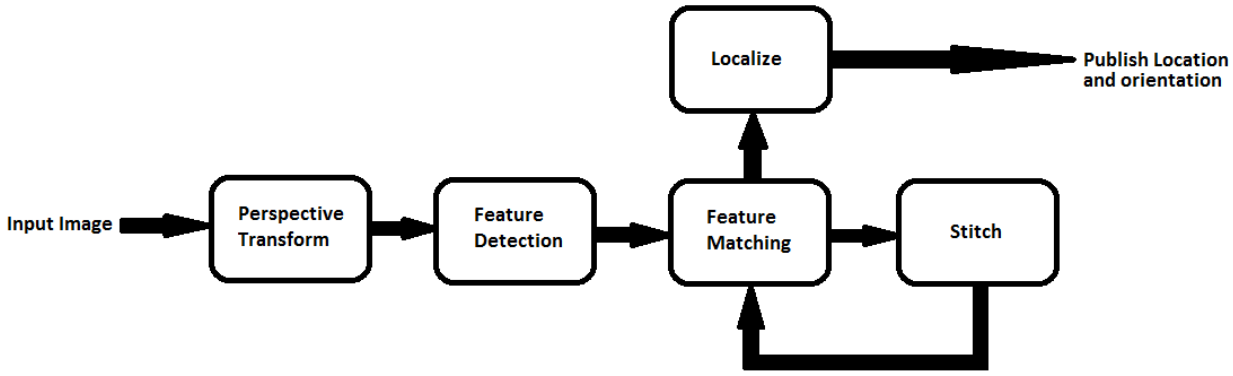

Figure 3: General SLAM approach

The full OpenCV library contains all of the functions and classes required to perform each of the main methods show above, such as warpPerspective, getPerspectiveTransform, ORB,

BFMatcher, findHomography, and SIFT/SURF. While some of these built-in methods were used in this project, others were replaced with custom methods or modified for various reasons.

Scale-Invariant Feature Transform[2] (SIFT) and its streamlined counterpart, Speeded-Up Robust Features[3] (SURF), are two very robust feature detection techniques that would have made the code very trivial and straightforward. However, the version of OpenCV that came with ROS does not include these functions. OpenCV was originally written in C++, and it was not until later that developers wrote a Python wrapper for it. The minor decrease in speed was justified by the portability and simplicity of Python, allowing programmers to focus on the algorithms and not be weighed down by the quirks of C++. This version of the Python wrapper for OpenCV does not support SURF or SIFT. Furthermore, these are proprietary techniques and cannot be used for free on this robot.

FindHomography is another image stitching function that is built into OpenCV that takes a list of matched features between two images and computes the perspective transform between them. Feature matching is a very noisy and unreliable technique that often contains many erroneous matches, but findHomography uses optimization techniques to filter out the outliers and maximize the accuracy of the transform. Unfortunately, this function often failed when undergoing tests because it has extra degrees of freedom that can otherwise be constrained using *a-priori* knowledge about the input data. This allowed it to converge on optima that are physically impossible and clearly unintuitive to a human observer.

CHAPTER 3
PERSPECTIVE TRANSFORM

The first method described in Figure 3 is the "perspective transform." Equation 1 shows the concept of a perspective transform, T:

$$T: V \rightarrow W \qquad \text{Equation 1}$$

where T is a 3x3 matrix, V = $\mathbb{R}^2$ represents 2D Cartesian coordinates on an input image with 1 as the third element, and W = $\mathbb{R}^3$ represents 2D homogeneous coordinates of the transformed image.

In order to understand the mathematics behind the perspective transform, it is important to define the geometry that projects points in 3D space to a 2D image plane[4]. The derivation of a camera's intrinsic and extrinsic matrices will be outlined with a pinhole-camera approximation, and then the perspective transform will be built on top of this framework.

Figure 4 shows how points in 3D Cartesian space get mapped to 2D coordinates on a pinhole camera's image plane. If the image plane is a distance f from the origin, then we can compute u and v by using triangle similarities:

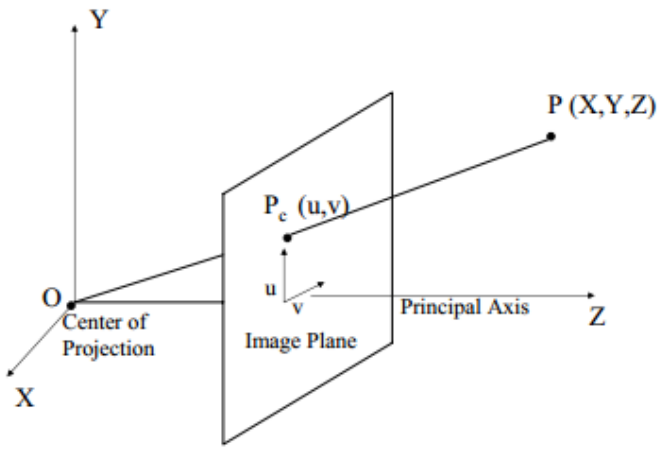

Figure 4: Diagram showing 3D cartesian space and its relationship to homogeneous coordinates.[4]

$$\frac{f}{Z} = \frac{u}{X} = \frac{v}{Y} \qquad \text{Equation 2}$$

This rearranges to:

$$u = \frac{fX}{Z}, \quad v = \frac{fY}{Z} \qquad \text{Equation 3}$$

For this reason, it is best to represent coordinates (u, v) in the image plane as homogeneous coordinates, (wu, wv, w), so that the above system can be represented with the following transform:

$$P_c = \begin{bmatrix} u' \\ v' \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \qquad \text{Equation 4}$$

If the principal axis, as shown in Figure 4, does not pass through the camera's origin, or center of projection, then it is necessary to offset u and v by constant factors of t_u and t_v, respectively. Furthermore, the skew of the image can be represented by including a Y-dependent offset for the u coordinate. In homogeneous coordinates, the resulting transform equation is:

$$P_c = \begin{bmatrix} u' \\ v' \\ w \end{bmatrix} = \begin{bmatrix} f & s & t_u \\ 0 & f & t_v \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = K * P \qquad \text{Equation 5}$$

This 3x3 matrix, K, only depends on the intrinsic properties of the camera, as the origin of the system is assumed to be the origin of the 3D space. However, in this application, the robot will be constantly moving through 3D space and therefore the relative location of the image plane will be changing. Therefore, it is necessary to introduce another 3x4 transform matrix, E, that takes into account the extrinsic properties of the camera that result from the relationship between the sensor and the external world. The E matrix is formed by first translating the 3D space so that the camera origin coincides with the origin of the 3D space, and then rotating about this point so that the image plane is perpendicular to the principal axis. If we let T be a 1x3

matrix representing the x, y, and z offsets and R be the 3x3 full rotation matrix, we can represent this transform as:

$$P = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = [R|RT] * \begin{bmatrix} X' \\ Y' \\ Z' \\ 1 \end{bmatrix} = E * P' \quad \text{Equation 6}$$

By combining Equation 5 and Equation 6, we can generate the 3x4 matrix, C, that maps a vector in 3D space from a fixed coordinate system to its pixel coordinates, (wu, wv, w), when the camera is in any arbitrary location in the 3D space.

$$P_c = \begin{bmatrix} u' \\ v' \\ w \end{bmatrix} = K * E \begin{bmatrix} X' \\ Y' \\ Z' \\ 1 \end{bmatrix} = C * P' \quad \text{Equation 7}$$

This means that there are 12 unknowns in the camera calibration matrix, C. Now that C has been defined, we can this information to relate the image coordinates of features to their respective image coordinates when the camera is at a different location. When the camera takes an image from a different location, the R and T matrices are different. Let C' be the calibration matrix that computes the 2D homogeneous coordinates on this new image plane:

$$P_{oc} = \begin{bmatrix} {u_o}' \\ {v_o}' \\ w_o \end{bmatrix} = C' * P' = \begin{bmatrix} u' \\ v' \\ w \end{bmatrix} \quad \text{Equation 8}$$

Therefore, there must be a 3x3 matrix, H, that can convert C to C'. By applying H to elements in P_c, it is possible to compute their respective coordinates in an image taken from a different perspective, P_c' as shown in Equation 9.

$$P_{oc} = \begin{bmatrix} {u_o}' \\ {v_o}' \\ w_o \end{bmatrix} = C' * P' = H * C * P' = H * \begin{bmatrix} u' \\ v' \\ w \end{bmatrix} = H * P_c \quad \text{Equation 9}$$

It can be shown that the entry in the third row and third column of H is always 1. Therefore, the matrix, H, has 8 unknowns. However, for every pair of points P_{oc} and P_c, there is an extra unknown, w_o, introduced. Therefore, it is necessary to define four points on one image and their matches on the transformed image to fully solve for the perspective transform matrix, H.

Fortunately, OpenCV has already worked out the linear algebra to solve for the 8 unknowns in a perspective transform matrix with the function "getPerspectiveTransform." Once H is returned from this function, there is another function called "warpPerspective" that applies this transform to each of the points on an input image to create the transformed image. This function also linearly interpolates intermediate pixel values and shades out pixels outside of the original image's field of view. Figure 5 shows the result of transforming an image from a forward facing camera to a bird's eye view of a section of the course. These images were generated from a Gazebo simulation of the course.

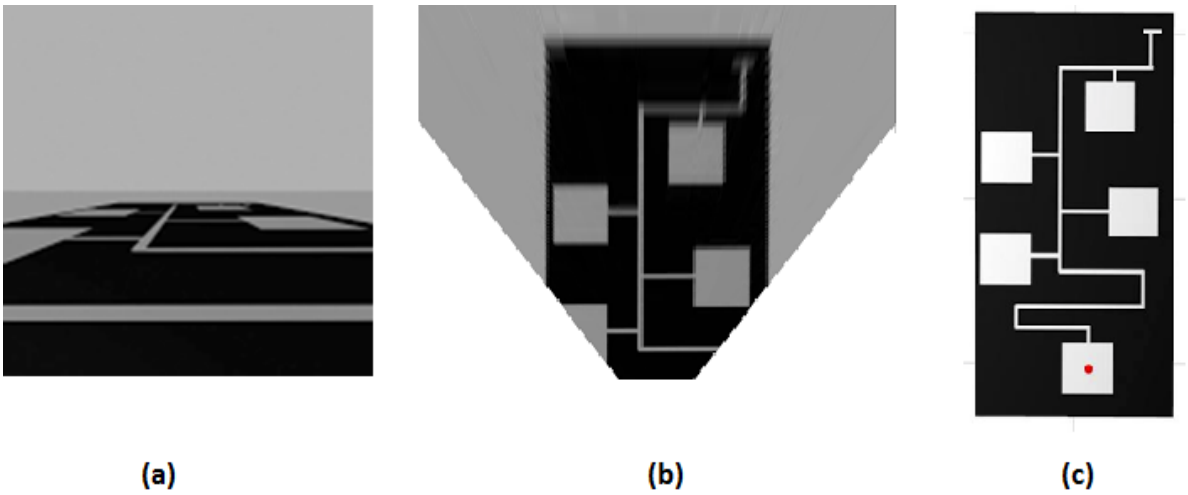

Figure 5: (a) Original image from forward-facing camera. (b) Bird's eye view of the same section by specifying that the four corners of the trapezoid become square. (c) Actual bird's eye view of the course.

CHAPTER 4
FEATURE DETECTION

Once an image has been transformed to represent a bird's eye view, features of interest must be located and processed in order to greatly simplify the image matching procedure. Corners are very important for image matching because they have a well-defined position and can be robustly detected[5]. They are the location where two edges intersect and can be detected by finding frames where the variance in pixel intensity in both the X and Y directions is large. For this project, we chose to use the broader category of "points of interest," as this includes any point with high intensity, end of an edge, or other unique features.

While SIFT and SURF are very efficient and comprehensive algorithms for doing exactly this, they were not available to us for this project. Therefore, it was decided to use the Oriented FAST and Rotated BRIEF (ORB) detector that was developed by Ethan Rublee in 2011[6]. It is based on the Features from Accelerated Segment Test (FAST) and Binary Robust Independent Elementary Features (BRIEF) algorithms.

FAST is a very fast technique for determining if a point of interest is actually a corner. Other techniques, such as the Harris Corner Detector, take into account 2D statistics and provide a score that describes a small square of pixels. The speed advantage of FAST comes from the way that it only considers the 16-pixel circumference of a circle draw around a pixel of interest. If there are enough contiguous pixels with "similar" intensities that are significantly different from the center pixel, then it decides that the pixel of interest is indeed a corner.

After key points of interest are located with FAST, it is the job of BRIEF to extract an array of descriptors for each of these key points. It is very difficult to match key points between two images without any information about the nature of the corner. BRIEF computes

information about each key point, such as orientation, range of pixel intensity, size, and radius of curvature.

The optional parameter, nFeatures, was set to its default value of 500. This parameter determines how many key points the detector will accumulate before dropping features of lower interest with newly detected features. Another parameter, WTA_K, was set to its default value of 2 and represents the number of points used by BRIEF to compute descriptor quantities.

As shown in Figure 6, there are many erroneous key points that are detected on an image that has just been perspective transformed. One source of error is that pixels that are far in the distance in the original image have their rectangular shapes exaggerated when the spaces between pixels are interpolated. Another source of error is that lines and features that lie on the edge of the original image appear to be corners when a black mask is applied over "unknown" space in the transformed image. The methods clean_image and _apply_roi were custom-

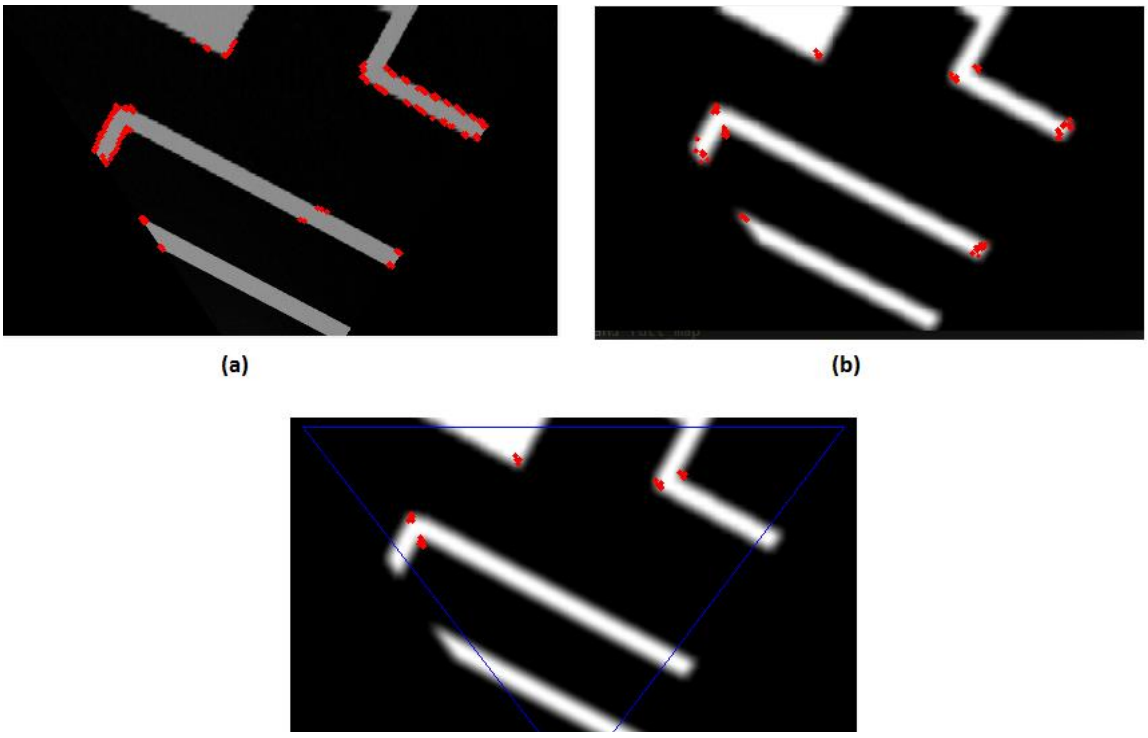

Figure 6: The red dots represent detected features. (a) Raw image sent through feature detection with pixelation corners. (b) Image after low-pass filtering and thresholding. (c) Resulting detected features after blue mask is applied.

designed to combat both of these issues by smoothing out pixilation corners, thresholding the pixel intensities, and masking out key points that lie suspiciously close to the border.

CHAPTER 5
FEATURE MATCHING, LOCALIZATION, AND STITCHING

While the previous two methods are extremely important to the SLAM process, the feature matching took the majority of the time and effort. OpenCV provides various esoteric built-in functions for matching features, such as Flann Feature Detection, for matching features detected by SIFT and SURF. However, because we are trying to match ORB descriptors, the only available option was the Brute Force Matcher.

The ultimate goal of feature matching is to determine the transform necessary to superimpose and stitch the new image on the full map to get a better representation of the field. After feature detection has been applied to the new image from the camera and the full map of the course, it is necessary to find the transformation matrix that will place the new image on the full map to get a more complete picture. Because every image used to generate the full map, as well as the new image, has already been perspective transformed to get a bird's eye view of the course from the same height, it will only require rotation and translation to stitch the images. These two processes can be handled by an affine transformation, as shown in Equation 10.

$$P_c' = \begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} * \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = A * P_c \qquad \text{Equation 10}$$

The affine matrix, A, has 6 unknowns. For every pair of points, two equations are generated. Therefore, it is only necessary to define three pairs of points to compute an affine transform.

The BFMatcher object allows for various algorithms to be used to match each point in a query set (new image) with its lowest-cost neighbor in a train set (full map) using the descriptor quantities. Because WTA_K was set to 2 in the ORB detector, the BFMatcher algorithm was set to NORM_HAMMING, as specified by the OpenCV documentation. The CrossCheck property was set to false. If true, then the *i*th key point in the query set is stored as a match for its lowest cost neighbor, the *j*th element, in the train set if and only if the best query-set match for the *j*th element is also the *i*th element. While setting this check to true may produce more reliable results, it is possible that little or no matches will be detected, hence resulting in a SLAM failure. Other statistical techniques, described later in this section, will be used to reject erroneous matches.

The BFMatcher returns an array of DMatch objects. A DMatch object contains three important properties: query index, train index, and the cost of the match. In order to run analysis on the integrity of the returned matches, we extract the pixel coordinates of the two matched key points by calling them by index in their key point arrays. The cost of the match can be used as a preliminary filter to decide which matches are "good."

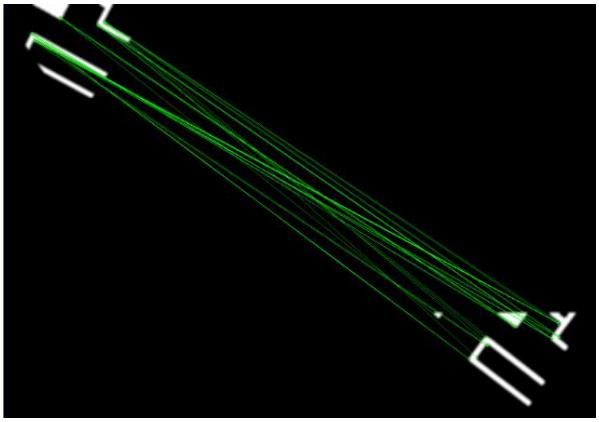

Figure 7: Matched features based on BFMatcher before filtering. Note how the green lines cross

The version of OpenCV used for this project did not contain the DrawMatches function, so it was custom-built for this project to assist with debugging. As shown in Figure 7, the matches that come directly from the BFMatcher cross each other and do not demonstrate a consistent transform to superimpose or stitch the images.

This is because the BFMatcher strictly uses the descriptors of key points to determine matches and does not consider their geographic locations.

FindHomography is a built-in OpenCV function that does exactly this. It takes a list of key points on the query image, an index-matched list of key points in the train image, and a pixel threshold. It uses Random-Sample Consensus (RANSAC) and returns a vector of Booleans that masks out key points in both of the arrays that are "outliers," along with a perspective transform matrix that best maps key points on the train image to their matches on the train image. It does this by applying the following algorithm[7]:

1. Choose four random matched pairs in the query and train arrays
2. Compute H, the perspective transform, using these four pairs
3. Project all points in the query array to their location in the train image using H
4. Compute point-to-point distance between these mapped points and their matches in the train image
5. Increment the accuracy count for each mapped point that falls within the pixel threshold
6. Update the "best H" if the accuracy count is greater than the previous "best accuracy count"
7. Return to step 1 and repeat N times

As previously explained, it is only necessary to apply a 2x3 affine matrix, rather than a 3x3 perspective transform matrix, to stitch the images. Unfortunately, there is no built in function

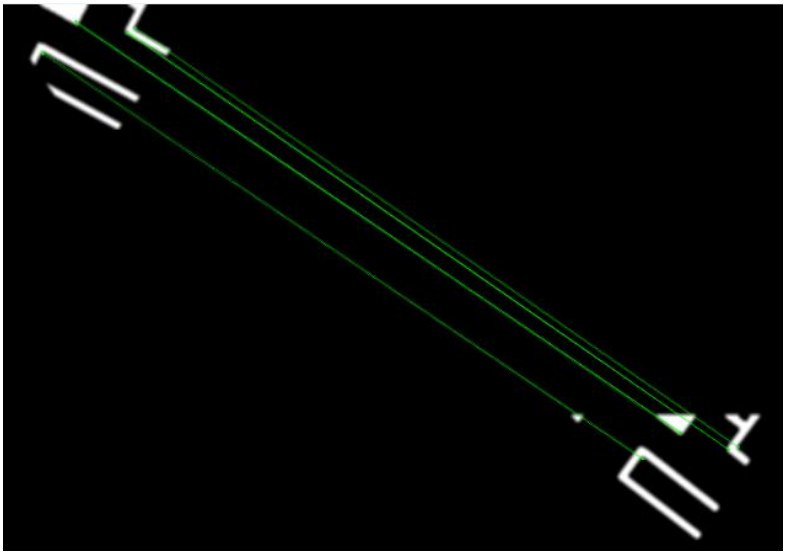

Figure 8: Matches remaining after applying RANSAC. Note how the green lines are all parallel.

that uses RANSAC to compute the best affine transform. Because findHomography works with three more degrees of freedom than necessary, it often converges on an incorrect optimum. It is capable of warping the image in a mathematically legal way that is not physically intuitive.

Therefore, the above algorithm was recreated to support RANSAC for computing the best affine matrix. After the affine transform, A, is computed and applied, more low-pass filtering is applied to the resulting composite image to prevent future iterations of SLAM from detecting corners on the seams. Furthermore, the findHomography algorithm was improved by providing support for decision-making in cases when the accuracy score ties the previous "best accuracy score." This is done by keeping track of the net accumulated distance of all points that fall within the threshold. Figure 9 shows the matches that are left over after running RANSAC, and Figure 9 shows the composite map when the images are stitched together.

Figure 9: Stitched composite image

Future work should seek to utilize compass, GPS, and odometer data to limit the degrees of freedom available to the affine matrix. Because the perspective transform already provides images from the same vantage point, there are really only three degrees of freedom, rather than

six, that need to be computed for the affine transform: x, y, and rotation. By applying this information to constrain the computation, it should be possible to further improve the RANSAC algorithm outlined above.

CHAPTER 6
CONCLUSION

Each of the algorithms outlined in this paper underwent unit testing. Figure 10 below shows the result of combining five camera images into one composite bird's eye image. Furthermore, a variety of images, shown in Appendix A, were used to simulate the input data streams for each of the major methods shown in Figure 3. The perspective transform, which will be hardcoded into the vehicle at the time of the competition, was carefully calculated based on an

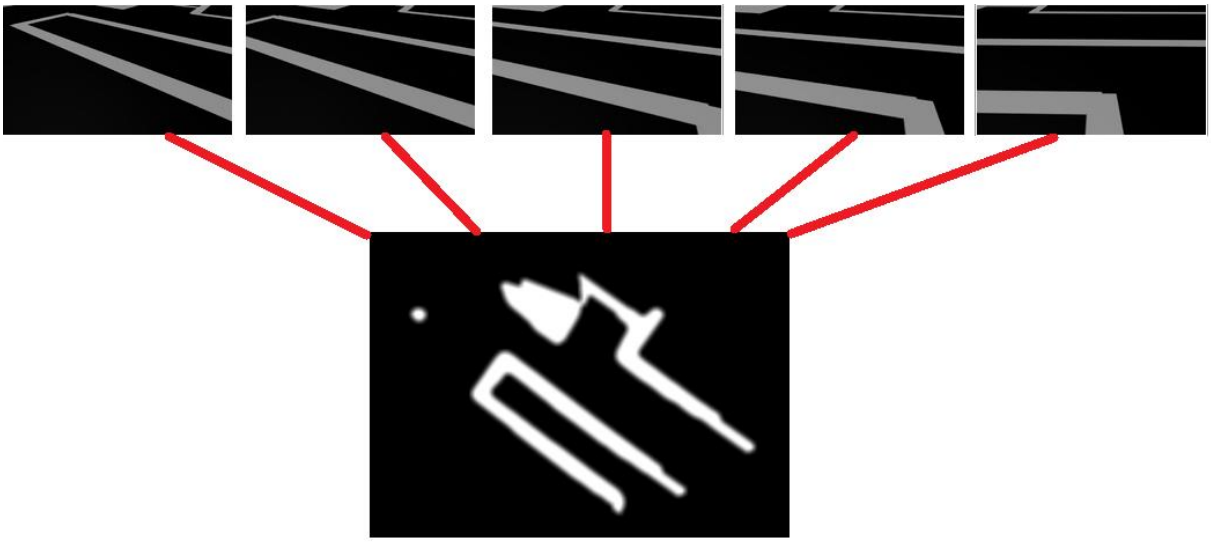

Figure 10: Composite image

image of a calibration square at various camera mounting angles. These angles were used to estimate the optimal mounting angle of the camera that gives enough information about the course to detect a fair amount of corners without the view extending so far that features in the distance are distorted beyond recognition.

The feature detection method was tested against a variety of images, such as random photographs of buildings, a snapshot of children playing soccer in an open field, and Picasso

paintings. The feature matching method and its affine transform output were tested using the outputs generated by feature detection. By continually detecting features, matching features, and comparing the results, these two methods were tuned and tandem until they were robust and provided consistent, accurate results.

While the algorithms individually underwent unit testing, this complete software package is only as good as its integration testing, and ultimately, its system testing when integrated with the entire ROS environment. While images from the Gazebo simulation were used to test the integrated system, it will be important to use real images to demonstrate that the software works. The real world will provide many unexpected issues and obstacles that may break the code in unimaginable ways.

Furthermore, odometer, GPS, compass, and other sensor data will need to be fused together in order to create increasingly accurate predictions of the vehicle's location. While no single ROS node will be able to pinpoint the vehicle's location with 100% accuracy, Kalman filtering must be used to weigh the various location approximations against their respective dependability. In order for this software to be written, it is necessary that the SLAM node described in this paper operate as a black box without creating any internal errors for the top-level software developer.

LIST OF REFERENCES

[1] Cabrera, Carlos. "Hardware Competition Southeastcon 2015." Memo. 4 Oct. 2014. Rev 3

[2] Lowe, David G. "Distinctive image features from scale-invariant keypoints."*International journal of computer vision* 60.2 (2004): 91-110.

[3] Bay, Herbert, et al. "Speeded-up robust features (SURF)." *Computer vision and image understanding* 110.3 (2008): 346-359.

[4] Bebis, George. Geometric Camera Parameters. N.p.: University of Nevada, n.d. PDF.

[5] C. Harris and M. Stephens (1988). "Proceedings of the 4th Alvey Vision Conference". pp. 147–151.

[6] Endres, Felix, et al. "An evaluation of the RGB-D SLAM system." Robotics and Automation (ICRA), 2012 IEEE International Conference on. IEEE, 2012.

[7] Hoiem, Derek. Image Stitching. N.p.: University of Illinois, Aug. 2011. PPT.

APPENDIX A: Unit Testing Images

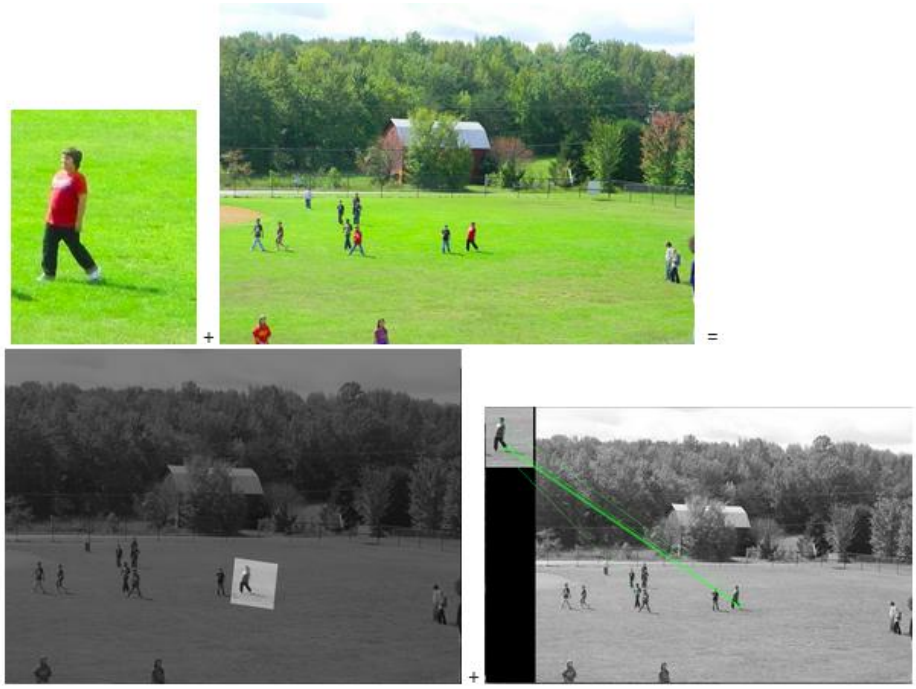

Figure 11: Image stitching performed on "guy in field"

Figure 12: Image stitching performed on Picasso-style Green Lantern

BIOGRAPHICAL SKETCH

Matthew Feldman is receiving his B.S. in electrical engineering from the University of Florida. His primary interests lie in neuromorphic computing, with secondary interests in machine intelligence and computer vision.