A Graph-based Approach to Solving SLAM: Simultaneous Localization and Mapping

Isaac Dulin
Advised by Matt Zucker

May 2016
# Contents

1 Introduction
   1.1 The Problem ................................................................. 4
   1.2 Motivation ................................................................. 4
   1.3 Related Work ............................................................... 4

2 Implementation
   2.1 Hardware ................................................................. 5
   2.2 Motion Model ............................................................... 5
   2.3 Measurement Model ....................................................... 6
   2.4 SLAM Algorithm ............................................................ 6

3 Results
   3.1 2D Simulation ............................................................. 8
   3.2 Real World ................................................................. 9

4 Conclusion
   4.1 Future Work ............................................................... 11

Appendices
   A Model Module ............................................................. 13
   B Code Repository ........................................................... 15
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Results of Accumulated Error</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Differential Drive Mechanics</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>TurtleBot</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Kinect Camera</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>Identifying Naturally Occurring Landmarks</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>ApriTag Detection</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>Constraint Graph</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>2D Simulation With and Without SLAM</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>Constraint Disagreement Pre- and Post-SLAM</td>
<td>10</td>
</tr>
</tbody>
</table>
Abstract

The ability to map a previously unseen environment is often considered a prerequisite for any truly autonomous robot. Since sensors and motors are often imprecise, it is necessary to find a way to identify and correct error as it accumulates. This problem is often referred to as simultaneous localization and mapping (SLAM), and it has been an active area of research in the field of robotics for the last 20 years. The goal of this project was to implement GraphSLAM [7], a solution to the SLAM problem. This algorithm adjusts the robot positions and the map in order to minimize disagreement between measurements. Ultimately, the algorithm was successfully implemented; however, a small issue with the camera offset still needs to be corrected before the algorithm can be applied to real-world scenarios.
1 Introduction

The ability to map an environment on the fly is often considered a prerequisite for any truly autonomous robot. When the robot's own pose is uncertain due to error-prone motors, mapping becomes even more challenging. This problem is referred to as the SLAM problem, which stands for simultaneous localization and mapping. In this section, the SLAM problem will be more formally defined, the motivation behind the problem will be discussed further, and some related works will be outlined.

1.1 The Problem

The SLAM problem consists of using data from imperfect sensors and imperfect motors to determine a robot's own sequence of positions while simultaneously generating a map of the environment. The problem has two inputs: actions and measurements. Every time the robot moves, an action, $u$, is generated and appended to the list of actions, $\{u_0, ..., u_i\}$. Each action represents the transformation from the previous pose to the current pose. Every time the robot takes a measurement, $z$, the data is added to the list of measurements, $\{z_0, ..., z_i\}$. Unfortunately, these initial estimates are insufficient, and will often result in maps that look like the left-hand image in Figure 1. Since error accumulates as the robot moves, small errors can propagate through the system to generate very inaccurate maps. The goal of SLAM is to take the list of actions and the list of measurements and use them to produce a better estimation of the pose sequence and map, one that looks more like the right-hand image of Figure 1.

1.2 Motivation

A robust solution to the SLAM problem is required for many applications of autonomous robots. Some examples include self-driving cars, mapping new terrain and exploration, search and rescue, and even simple applications like self-controlled vacuum cleaners. Any autonomous robot expected to navigate an unfamiliar environment requires the ability to map the environment on the fly. Because of its importance, SLAM has been an active area of research in the field of robotics for the last twenty years. The goal of this project was to implement an already existing solution to SLAM in order to better understand the problem and approaches to solving it.

1.3 Related Work

In general, SLAM solutions fall into two categories. The first maintains a current state that is updated by subsequent actions and measurements. Each state contains a current position as well as a map. Many of these methods employ particle filters [6] [8] to keep track of different possible states and their relative likelihood. The second method estimates the full pose sequence and map from the entire list of actions and measurements. These methods often rely on a least-square minimization to find the pose sequence and map that causes the lowest disagreement between measurements. One of these algorithms, GraphSLAM [9] [7] was implemented in this project.

2 Implementation

In this section the details of the implementation will be discussed. Any SLAM implementation must include measurement and motion models that are specific to the hardware used. The measurement and motion models also impact the algorithm's implementation. The details of the hardware, the motion model, the measurement model, and the algorithm will all be discussed in the following subsections.
2.1 Hardware

The robot used was a TurtleBot, pictured in Figure 2. The TurtleBot operates using a differential drive motor configuration, which means that it has two parallel wheels that can rotate independently of one another (see Figure 3). This configuration allows for instantaneous turning and forward motion, but not instantaneous, side-to-side motion. To go directly left or right, it would first have to turn ninety degrees and then proceed straight. For more information on the TurtleBot or to view specifications, see their website [4].

![Figure 3: Differential Drive Mechanics](image)

The sensor used in this project was a Kinect camera, pictured in Figure 4. While the Kinect has an infrared depth sensor, only the basic image camera was used for this project. For more information on the details of the camera, see their specifications page [2].

The Kinect and the TurtleBot can communicate with each other and with a computer using ROS (Robot Operating System), a suite of packages and tools to create a flexible framework for controlling robots. All of the code written for this project was built on top of ROS and utilizes its basic communication and control channels. For more information on ROS, you can read their documentation page [3].

2.2 Motion Model

The motion model generates the robot’s initial guess of the new position after an action. This function is defined as $x_t = f(x_{t-1}, u)$ where $x_t$ is the robot's new position, $x_{t-1}$ is the robot's previous position, and $u$ is the action command that was given to the robot. If the robot were perfect, then its actions and its current position would always be able to predict the new position; however, since motors are imperfect and wheels can slip, this estimate is usually a little off. In practice, $x_t = f(x_{t-1}, u) + \mathcal{N}(0, \sigma)$ where $\mathcal{N}(0, \sigma)$ is a Gaussian distribution with a mean of 0 and some standard deviation $\sigma$. The goal of SLAM is to use measurements to correct for this error that accumulates during motion.

This model can be improved upon slightly by using odometry. Odometry is a measurement of the wheel rotations to determine what motion occurred. This is better than relying on the action that was initially sent to the wheels. Error can be introduced in the processor, the signal that was sent to the motor, the motor itself, and any wheel slippage. When odometry is used to measure how far the wheels turned, the only error that is introduced is from wheel slippage or errors in the odometry. This is still a poor estimate.
of motion, but it is substantially better than dead reckoning. In this implementation the cumulative motion measured by the odometer is reported as the action, \( u \).

In the code, the motion model is encoded in the function `motion_model()` which exists in the `model.py` file. Repeated calls of this function generate the robot’s initial estimate of the pose sequence.

### 2.3 Measurement Model

Different SLAM algorithms require different types of measurements. Some use point clouds or depth sensor data, while others, such as GraphSLAM, visually identify *landmarks* that can be easily recognized from different positions. These landmarks can exist in the scene naturally or can be added to an environment specifically for the robot. An example of naturally occurring landmarks is shown in Figure 5 where sharp corners are identified and used to map the area.

For this implementation, AprilTags [1], shown in Figure 6, are used as the landmarks. These tags are similar to QR codes and can be easily identified by the camera. Since the tag’s size is known, the full orientation of the tag can be determined from the image.

The *measurement model* determines what measurement the robot finds given its current position and the landmark(s) that it can see. The function is defined as

\[
z = f(x_i, l)
\]

where \( z \) is the robot’s measurement, \( x_i \) is its current position, and \( l \) is the position of the landmark. Again, some error is introduced in this measurement, and in practice

\[
z = f(x_i, l) + N(0, \sigma)
\]

where \( N(0, \sigma) \) is a Gaussian distribution with a mean of 0 and some standard deviation \( \sigma \). For this implementation, \( z \) is a set of four \((u, v)\) pixel locations corresponding to the corners of the AprilTag, \( l \). When the robot views multiple AprilTags at the same time, multiple \( z \) measurements are generated. Since the true size of the AprilTag is known (12 cm x 12 cm), the position with respect to the camera can be calculated. The landmark’s position in the world can then be calculated from the robot’s position in the world. These poses have three degrees of rotational freedom as well as three degrees of translational freedom. They can be represented as a \((\theta_x, \theta_y, \theta_z, x, y, z)\) tuple. In the actual implementation, the rotational elements are represented by a quaternion [5] instead of a rotation vector.

In the code, the measurement model is encoded in the function `meas_model` which exists in the `model.py` file. Measuring the disagreement among measurements is done by calling this function to see what the expected measurement would have been given some pose and some landmark. A good final configuration will have low disagreement between the expected measurement and the actual measurement.

### 2.4 SLAM Algorithm

The algorithm used in this implementation is called GraphSLAM [9]. The way this algorithm works is by treating every constraint as a link in a graph. Constraint links are one of two types:

1. Every action forms a constraint between a pose and the subsequent pose.
2. Every measurement taken forms a constraint between a landmark and the robot’s pose.

Each robot pose and landmark position forms a node of the graph; this graph structure can be seen in Figure 7. Constraints that conflict with one another can be measured as error, and running the algorithm minimizes this error by slightly moving the robot poses and landmark positions. A helpful analogy is to imagine that all of the nodes in this graph are connected by springs of varying length. The disagreement between measurements is the distance that the springs have to stretch or compress. Once the springs are all
released, the system will settle into a state with low overall displacement. Similarly, by adjusting the nodes, the overall disagreement can be reduced.

What follows is a detailed discussion of the code and how it relates to the pseudocode in Algorithm 1. The main functionality of the code is contained in the model.py file which can be seen in Appendix A. The pseudocode in Algorithm 1 is slightly modified from the pseudocode presented in the Grisetti paper [7], but it is largely the same. The algorithm begins by initializing naive guesses for the landmark positions and pose sequence. This initial guess \( x \) is represented as \( x \) in the pseudocode; in the actual code it consists of two data structures: a list containing the ordered robot poses and a dictionary mapping AprilTag IDs to their locations. \( x \) represents the robot's current guess for the pose sequence and map, and as the algorithm runs it will be iteratively updated. In line 2, \( G \) is initialized as the list of constraints which is really the list of actions combined with the list of measurements. For the purposes of the algorithm, both actions and measurements represent constraints put on the system so they are combined into one list of constraints. In line 3 a while loop is initialized to run the algorithm until iterative improvement approaches zero. The interior of this while loop, or one iteration of the algorithm, corresponds to the run_slam() function. Lines 6 and 7 simply define \( Q \) to be one of two vectors containing standard deviations for the actions or measurements. These values dictate how the error will be distributed between the map estimate and the pose sequence. High standard deviations in the action constraints’ \( Q \) vector coupled with low values in the measurement constraints’ \( R \) vector would cause the algorithm to heavily readjust the pose sequence to force the measurements to agree. In contrast, low values in the \( Q \) vector couple with high values in the \( R \) vector would cause the algorithm to readjust the landmark position to best fit the measurements while only slightly adjusting the pose sequence. Lines 8-10 represent a call to either the meas_error() or action_error() function, depending on whether the constraint in question is a measurement or an action. These functions return the error, \( \alpha_j \) as well as the two Jacobians of the error function \( A_j \) and \( B_j \). Next, the algorithm uses the result of the error function to fill in the information matrix \( H \). \( H \) is a large matrix that contains six rows and columns for each robot position and each landmark position. In the pseudocode notation, \( H_{ij} \) represents a 6x6 slice of \( H \) corresponding to the \( i \)th and \( j \)th pose or landmark. For action constraints, \( j \) will always be the pose immediately following \( i \) (in other words \( j = i + 1 \)). For measurement constraints, \( i \) will be the pose from which the measurement was taken and \( j \) will be the landmark that was seen (with the actual index value being \( j + \text{total number of poses} \)). In line 14 the \( b \) vector is similarly updated. In the code, all of these
updates are performed in the `fill_matrices()` function. In line 16, the initial node is fixed by adding the identity matrix, and then in line 17, the resulting least-squares minimization problem is solved. The result, $\Delta x$, is used to update the pose sequence and landmark positions in line 18. The resulting updated pose sequence and map, $x$, is returned by the `run_slam()` function.

### Algorithm 1: Graph SLAM

```python
1: $x$ ← initial pose and landmark guess
2: $C$ ← the list of constraints // constraints can be pose-to-pose or pose-to-measurement
3: while not converged do
4:   $H$ ← 0 $b$ ← 0 // fill both $H$ and $b$ with zeros
5:   for constraint$_{ij}$ in $C$ do
6:     if constraint$_{ij}$ is a pose-pose constraint then $\Omega$ ← $Q^{-1}$
7:     else // constraint$_{ij}$ is a pose-measurement constraint
8:       $e_{ij}$ ← result of the error function
9:       $A_{ij}$ ← $\frac{\partial e_{ij}}{\partial x_i}$ $B_{ij}$ ← $\frac{\partial e_{ij}}{\partial x_j}$
10:      // update the information matrix
11:      $H_{[i]}$ ← $A_{ij}^T \Omega A_{ij}$ $H_{[j]}$ ← $A_{ij}^T \Omega B_{ij}$
12:      $H_{[i]}$ ← $H_{[i]} + B_{ij}^T \Omega B_{ij}$ $H_{[j]}$ ← $H_{[j]} + B_{ij}^T \Omega B_{ij}$
13:      // update the b vector
14:      $b_{[i]}$ ← $A_{ij}^T \Omega e_{ij}$ $b_{[j]}$ ← $B_{ij}^T e_{ij}$
15:      $H_{[0]}$ ← $I$ // Fix the first pose by adding the identity matrix
16:      $\Delta x$ ← solve($H\Delta x = -b$) // solve using a Least Squares solver such as Levenberg-Marquardt
17:      $x$ ← $x + \Delta x$ // update the pose and landmarks
18:     return $x$
```

## 3 Results

### 3.1 2D Simulation

Before trying to implement the algorithm in three dimensions on the robot, a simpler two-dimensional simulation was created. The results were very successful, and can be seen in Figure 8. For this simplified simulation, the robot poses had three degrees of freedom and were represented by an $(x, y, \theta)$ tuple. The landmarks consisted of points in 2D with no orientation, $(x, y)$ pairs. In Figure 8 The red line represents the true robot path while the blue line represents the robot's estimation of its path. Similarly, the red dots indicate the true position of landmarks while the blue dots represent the robot's estimation of their position. On the left is the robot's uneducated, pre-slam guess. The error is particularly high on the right hand side of the image when the error has propagated through the system (the robot began at the bottom of the loop and proceeded clockwise). On the right the picture is remarkably better. After running SLAM, both the pose estimates and the landmark estimates match the ground truth almost perfectly.
3.2 Real World

Overall, the results indicate that the algorithm is working as expected; however, an issue with the camera offset is preventing the algorithm from working well on real data. This camera offset issue is discussed further at the end of this section and in section 4.1.

The final algorithm, as discussed in the implementation section, was run on data collected by the TurtleBot, and the results can be seen in Figure 9. In these images, the blue line represents the pose sequence and the green dots represent the landmark positions. In Figure 9a, the measurement constraints are overlaid as green lines while in Figure 9b the action constraints are overlaid as red lines. In both images, small sections are enlarged to better highlight the changes that occur.

Figure 9a shows how the algorithm reduces the disagreement between measurements of the same landmark. On the left, before the SLAM algorithm is run, the green lines representing the measurements are in severe disagreement. After the algorithm is run, the measurements should agree more closely with one another, and this is what happens. On the right, the green lines generally converge to their respective landmarks. As expected, some disagreement still exists; however, the measurements clearly agree more in the post-SLAM image than in the pre-SLAM image.
(a) Measurement constraints with pre-SLAM on the left and post-SLAM on the right.

(b) Action constraints with pre-SLAM on the left and post-SLAM on the right.

Figure 9: Constraint Disagreement Pre- and Post-SLAM
Figure 9b shows how the action constraints adjust when the algorithm is run. Since the initial naive estimate for the pose sequence is generated from the actions, the red lines should perfectly match the blue lines in the image on the left, and it’s clear that they do. After the SLAM algorithm is run, some of the error from the measurement constraints should be shifted to the poses, and this is what occurs in the image on the right. Small deviations in the pose sequence are necessary to create the radical improvements in measurement agreement that are seen in Figure 9a.

Overall, the algorithm behaves exactly as expected. The initial estimate contains zero disagreement between poses and actions, but contains high disagreement between different measurements. After SLAM is run, the overall error has decreased, and large disagreement between measurements has been transferred to small deviations between the poses and the actions. All of this is strong evidence that the algorithm works as expected.

Unfortunately, in practice, this implementation does not quite work. In a variety of informal trials where the pre- and post-SLAM maps were compared to a ground truth, the algorithm failed to converge on a true estimate. The reason for this is the camera offset. In the model, it is assumed that the camera and the robot share an identical pose; however, in practice, the kinect camera is located about 5 cm behind the center of the robot. The result is that while pure rotations of the robot are modeled as pure rotations of the camera, in reality the center of the robot rotates and swings the camera in an arc. This inconsistency between the model and reality causes the algorithm to perform poorly on real data taken by the robot.

4 Conclusion

The GraphSLAM algorithm was successfully implemented. By exploiting disagreement among different measurements, accurate mapping and localization can occur despite error-prone sensors and motors. The algorithm works as expected except for the camera-offset issue, which is easily correctable.

4.1 Future Work

As mentioned previously, the next step in this project is to correct for the camera offset. After that, there are a few interesting extensions that could be pursued.

First, adding a dense map from a range scanner is very easy once a sparse map has been generated. Simply overlaying data from the Kinect’s infrared depth scanner would produce a fairly accurate dense map of the room.

Second, the algorithm should be tested against a known ground truth. The three paths and maps (ground truth, pre-SLAM, and post-SLAM) can then be compared to see how much the SLAM algorithm improves the estimate. Ideally, there would be high disagreement between ground truth and the pre-SLAM estimate and low disagreement between ground truth and the post-SLAM estimate.
References


Appendices

A Model Module

```python
import sys
import quat_test as quat
import transformations
import numpy as np
import cv2

from helper_classes import *
from scipy.sparse.linalg import spsolve

def motion_model(x, u, diag_Q=np.zeros(6)):
    # x is robot-world xf orm
    # x is also the robot pose in the world frame
    # should just add error to the robot frame and then convert?
    # that seems sketchy but it'd be really clean
    action = u.copy()
    action += diag_Q * np.random.normal(size=6)
    new_xform = quat.xform-compose_twist(x, action)
    return new_xform

def motion_error(xa, u, xb):
    expected_pose = motion_model(xa, u)
    # calculate expected pose
    # calculate error and jacobians
    # WTF. Should be affine error of expected vs actual
    # so why do the jacobians work when its as us as?
    eij, Aij, Bij = quat.xform_error(expected_pose, xb, True)
    # calculate the extra jacobian piece
    R = quat.quaternion_matrix(quat.quaternion_from_rvec(u[3:]))[3, 3]
    dexp_du = np.zeros((6, 6))
    dexp_du[3, 3] = R
    dexp_du[3:, 3:] = np.dot(R.T, -quat.cross_matrix_3x3(u[3:]))
    Aij = np.dot(Aij, dexp_du)
    Bij = np.dot(Bij, dexp_du)
    return eij, Aij, Bij

def meas_model(x, l, tags, K, dist_coeffs, diag_R=0, ret_jacobians=False):
    # a is pose, a (q, tvec) xf orm
    # l is landmark, a (q, tvec) xf orm
    # l is in world coordinates
    # also returns the twist/dxform jacobians (if ret_jacobians)
    # get the twist difference and the jacobians (ret_jacobians=True)
    twist, JeA, JeB = quat.xform_error(x, l, True)
    # project the points onto the camera's sensor plane
    xerr = cv2.projectPoints(tags.reshape((-1, 1, 3)),
                              twist[:3], twist[3:],
                              K, dist_coeffs)
    xerr = xerr[0].flatten()
    # add some error to the xerr
    xerr += diag_R * np.random.normal(size=8)
    xrt = xerr[0:4]
    Iij = np.dot(Jrt, Jei)
    Iij = np.dot(Jrt, Jei)
```

if ret_jacobians:
    return msrmnt, Aij, Bij
else:
    return msrmnt

def meas_error(x, n, z, tags, K, dist_coeffs):
    # x is pose
    # n is map_estimate
    # z is msrmnt which is 8x1 (4 uv points)
    # get the expected msrmnt and the jacobians
    expected_msrmnt, Aij, Bij = meas_model(x, n, tags, K, dist_coeffs, ret_jacobians=True)
    # eij is the difference between expected and actual (u,v) coords
    eij = expected_msrmnt - z
    return eij, Aij, Bij

def get_map_est(x, z, tags, K, dist_coeffs):
    # x is pose
    # z is 8x1 msrmnt, a set of 4 (u,v) coords
    corners = np.array(z).reshape((4,2,1))
    __, rvec, tvec = cv2.solvePnP(tags, corners, K, dist_coeffs)
    map_est = (quat.quaternion_from_rvec(rvec.reshape(3)), tvec.reshape(3))
    map_est = quat.xform_compose(x, map_est)
    return map_est

def fill_matrices(i, j, Aij, Omega, Bij, ej, H, b):
    H[i, j] = H[i,j] + np.dot(Aij.T, np.dot(Omega, Aij))
    H[i, j] = H[i,j] + np.dot(Aij.T, np.dot(Omega, Bij))
    H[i, j] = H[i,j] + np.dot(Bij.T, np.dot(Omega, Aij))
    H[i, j] = H[i,j] + np.dot(Bij.T, np.dot(Omega, Bij))
    b[i] += np.dot(Aij.T, np.dot(Omege, ej)).reshape(6,1)
    b[j] += np.dot(Bij.T, np.dot(Omege, ej)).reshape(6,1)
    return H, b

def run_slam(poses, landmarks, actions, measurements, cam_info, max_iters=20):
    R = np.diag(np.array([1e-5, 1e-5, 1e-5, .005, .005, .005])**2)
    Q = np.diag(np.array([2e-10, 2e-10])**2)
    Hinv = np.linalg.inv(R)
    Ginv = np.linalg.inv(Q)
    nseed = copy
    Nseed = copy
    poses = list(poses)
    landmarks = landmarks.copy()
    actions = list(actions)
    measurements = list(measurements)\n    mark_ids = {}  
    for i, key in enumerate(landmarks.keys()):
        mark_ids[key] = i
    iterations = 0
    converged = False
    while not converged and iterations < max_iters:
        iterations += 1
        converged = False
        for i in range(len(poses)):
            for j in range(len(poses)):
                if i != j:
                    fill_matrices(i, j, Aij, Omega, Bij, ej, H, b)
```python
# Initialization N and b
N = len_matrix(len(poses), len(landmarks))
b = len_vector(len(poses), len(landmarks))

# Add all pose-pose constraints
for i in range(1, len(poses)):
eij, Aij, Bij = motion_error(poses[i-1], actions[i], poses[i])
H, b = fill_matrices(i-1, i, Aij, B, H, b)

# Add all pose-mem constraints
for meas in measurements:
    if meas[0] > len(poses):
        continue
eij, Aij, Bij = meas_error(poses[meas[0]], landmarks[meas[1]],
meas[2], cam_info[meas[4]], cam_info[meas[5]],
cam_info[meas[6]])
H, b = fill_matrices(meas[0], len(poses) + mark_idxs[meas[1]],
eij, Qinv, Bij, eij, H, b)

# fit the first point
H[0, 0] = 1e5 * np.eye(6)

# add damping
H = 1e5 * np.eye(H.shape[0])

# solve using a sparse solver (Levenberg-Marquardt)
dx = sp.solve(H.get_solvableView, -b)
print('dx norm:', np.linalg.norm(dx))

# Update poses
for i, pose in enumerate(poses):
    new_pose = quatcompose_twist(pose, dx[6*i:6*(i+1)])
    poses[i] = new_pose

# Update landmarks
for landmark in mark_idxs:
    index1 = len(poses)*6 + mark_idxs[landmark]*6
    index2 = len(poses)*6 + (mark_idxs[landmark]+1)*6
    new_mark = quatcompose_twist(landmarks[landmark],
dx[index1:index2])
    landmarks[landmark] = new_mark

# Check if we're converged (do this better probably)
dxsz = np.linalg.norm(dx)
iterations += 1
if djsx < 1e-1:
    converged = True

return poses, landmarks
```

### B  Code Repository

Contact at https://github.com/IsaacDulin/ for the full code repository. Other parts of the code include scripts for data collection and to create pre- and post-SLAM visualizations.