

# EECS 442: Computer Vision

## Simultaneous Localization and Mapping

### Final Report

Thomas Cohn  
University of Michigan  
cohnt@umich.edu

Nicholas Konovalenko  
University of Michigan  
nkono@umich.edu

Neil Gurnani  
University of Michigan  
ngur@umich.edu

James Doredla  
University of Michigan  
doredlaj@umich.edu

## 1. Introduction

Simultaneous localization and mapping (SLAM) is a famous problem in robotics, where a computer system attempts to construct a map of its environment, while simultaneously maintaining knowledge of its position within that map. In order to construct this map, the agent needs some way of observing its surroundings – most modern solutions to SLAM rely on obtaining 3D data from laser ranging (LIDAR) sensors or depth cameras. But these sorts of sensors are expensive and somewhat uncommon.

SLAM has a vast number of real world practical applications across a growing variety of fields and disciplines. To really understand why SLAM is important and can be used in real world applications to aid humans, let's take a look at a simple example. According to [1], "Consider a home robot vacuum. Without SLAM, it will just move randomly within a room and may not be able to clean the entire floor surface. In addition, this approach uses excessive power, so the battery will run out more quickly. On the other hand, robots with SLAM can use information such as the number of wheel revolutions and data from cameras and other imaging sensors to determine the amount of movement needed. This is called localization. The robot can also simultaneously use the camera and other sensors to create a map of the obstacles in its surroundings and avoid cleaning the same area twice. This is called mapping." As we can see from this example, SLAM can prove useful in mapping unknown environments, which can be applied to various different fields.

In terms of related works we are building on, both [2] and [3] contain more probabilistic implementations of SLAM, where an emphasis is placed on handling uncertainty. Our approach built off [4], which strikes a balance and happy medium. Our approach which was based more on a de-

terministic model relying on features extracted, allowed us to handle more points on the map we were working with. Our main trade off stemmed from the fact that although we could handle more features, we weren't able to most effectively handle uncertainty within our results.

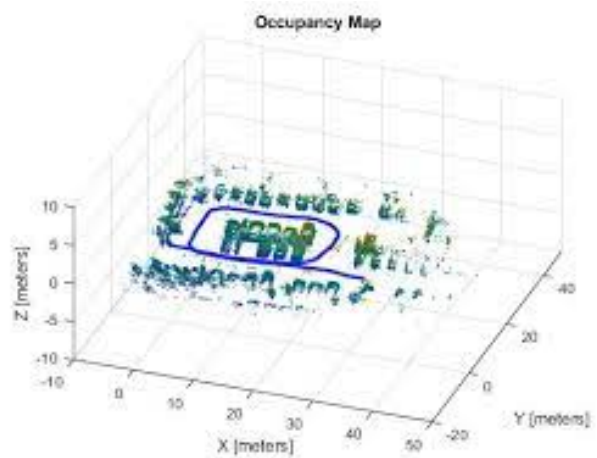


Figure 1. Reproduced from [1], SLAM with 3D Lidar Occupancy MAP

In the scope of our project, our goal is to produce an effective SLAM system using only a single, ordinary (2D) camera for sensory input. This type of camera is ubiquitous in our modern, digital world, so our SLAM algorithm could easily be applied to almost any mobile device.

We were able to obtain the data we needed for this project from online SLAM datasets from the ETH3D website. Through using the video sequences from ETH3D, we will be able to both qualitatively and quantitatively analyze the quality of our results.

## 2. Approach

The main principle of our approach to simultaneous localization and mapping is parallax. When viewing a scene from different perspectives, the apparent position of objects in the scene will change. Objects nearer to the camera will appear to move much more than objects further away. (TODO: Parallax figure.) By measuring the relative motion of multiple keypoints in the image, we are able to estimate the motion of the camera. Once we have the relative transformation between two different images, we can triangulate the location of these keypoints in three dimensional space, to build a sparse map of our surroundings. We then iteratively update our map with new camera images by solving the Perspective- $n$ -Point problem, where we match observed keypoints in the image space to their corresponding map points in three dimensional space.

### 2.1. Keypoint Detection and Matching

The first step in the approach for our project was to detect, extract and match features from the ETH3D dataset. In order to first extract the features, we used [5] ORB descriptions and OpenCV to create a Brute Force Feature Matcher.

This brute force feature matcher took the descriptor of one feature in first set and is matched with all other features in second set using the distance calculated. We used a distance threshold to isolate the closest matches that lied within the threshold.

Then, we had to filter the feature matches we obtained: we had to get rid of duplicate matches to ensure that each keypoint we had found matched up with exactly one other keypoint.

### 2.2. Map Initialization

In order to begin building a map of the camera's surroundings, and localizing the camera within it, we have to start with some ground truth information. As we initialize the map, it is important that we have a sufficient parallax to accurately estimate the camera motion and triangulate the points. Thus, we allow map initialization to "fail" if there's insufficient parallax; in this case, we simply get another image from the camera, and repeat until we've observed sufficient parallax.

The map initialization process is as follows. We set the first image to be the reference frame; all successive images will be determined in terms of this frame. We then attempt to initialize the map with frame two, three, and so on until the map is successfully initialized. Given some frame  $i > 1$ , we extract image keypoints from the frame, and match them with the descriptors from frame one. We then estimate the essential matrix, which gives the linear transformation from the camera pose in frame  $i$  to the camera pose in frame 1. (TODO: Diagram of essential matrix.)

This process gives us a result up to some unknown scaling factor. Normally, this scaling factor would be estimated with other sensors, such as an inertial measurement unit chip, or a wheel encoder on a camera-equipped robot. In our case, for ease of comparing with the provided ground truth camera poses, we define the initial scaling factor to be the ground truth distance between the two poses. (Alternatively, we could fix an arbitrary scale, and then transform the ground truth camera trajectory into our global coordinate system via scaling. But this simply amounts to much more complex code for an identical result.)

We compute the essential matrix using a RANSAC scheme, with the OpenCV library. This gives us a list of inlier and outlier matches, so we are able to discard false matches. Once we find the relative transformation between the two camera frames, we are able to triangulate the inlier matched keypoints in three dimensions. We use the DLT algorithm [6], as implemented by the OpenCV library. This is the point where we determine whether or not there is sufficient parallax to finish the map initialization process. We compute the vector from each triangulated keypoint to the two camera poses (in the global coordinate frame), and measure the angle. We then check for two criteria, as suggested by Feiyu Chen: there must be at least 40 points where the angle is at least  $3^\circ$ , and the average angle must be greater than  $5^\circ$ . If these criteria are satisfied, we add these frames and triangulated points to the map (with their descriptors), thus completing the initialization phase. If one of these criteria isn't satisfied we scrap the current frame, and attempt to finish the map initialization again with frame  $i + 1$ .

### 2.3. Camera Tracking

In the main, tracking phase of our SLAM project, we begin by trying to determine which map points should be visible from the camera's current pose. If we knew the camera's pose, we could project them directly using the pinhole camera model. We assume that the camera has not moved a large amount since the previous frame, which allows us to use the previous camera pose as an estimate for the current camera pose. We ignore any points which would appear behind the camera or outside of its field-of-view. We then match these map points with extracted points from the current frame, and then estimate the current frame's camera pose by solving the Perspective- $n$ -Point problem. We once again use a RANSAC scheme, as implemented by OpenCV.

We maintain a list of key frames in the constructed map; if the translation or rotation between the current frame and previous key frame is above a certain threshold, we consider the current frame to be a new key frame. We then follow a similar process to the map initialization, where we match keypoints between the current frame and the last key frame, and then triangulate them to find their positions in three di-

mensions.

We maintain a global version of the map, with all recorded points throughout the SLAM process. But for future point matching, we only consider a subset of the most recently observed map points. This ensures that the computational cost doesn't grow with time, as we're never matching to more than a certain number of features.

## 2.4. Visualization

Once we have the feature points and camera pose stored in a map object, we have all the necessary information in order to visualize the path the camera moves along. Using matplotlib, we are able to plot the feature points and camera's coordinate axes in 3D-space. To visualize the rotations the camera makes through each frame, we take the Hamilton product of the camera's coordinate axes with a rotation quaternion that is stored in every frame's camera pose. With the downloaded datasets we are also given the camera's ground truth positions for each frame, which we plotted against our camera's simulated SLAM path.

## 3. Results

When running our SLAM implementation with datasets downloaded from eth3d, we needed to analyze how correctly our implementation was behaving. Fortunately, the datasets downloaded from eth3d come with the ground truth camera trajectory for each image, thus allowing us to plot that against the trajectory of our SLAM implementation. We ran our implementation against several datasets, including Plant 1 in Figure 2, which has 70 images. We wanted to test datasets with both few and many pictures, as such we also ran our implementation on Table 3 mono in Figure 4 which contains 1180 images.

## 4. Conclusion

Simultaneous Localization and Mapping (SLAM) can be used for a computer system to construct a map of an unknown environment while simultaneously keeping track of the computer system's location within it.

We learned that SLAM is a powerful way for autonomous systems to know their relative location within an environment while formulating a map of the region as a whole. SLAM can be used in a variety of real world applications, some of which include autonomous vehicles and crop field robots.

## References

- [1] "What is slam (simultaneous localization and mapping) – matlab simulink." [Online]. Available: <https://www.mathworks.com/discovery/slam.html>

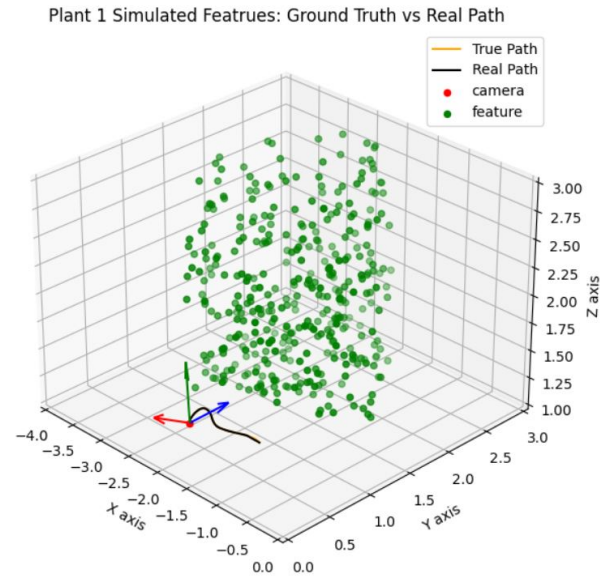


Figure 2. Plant 1 dataset, simulated trajectory plotted against ground truth trajectory

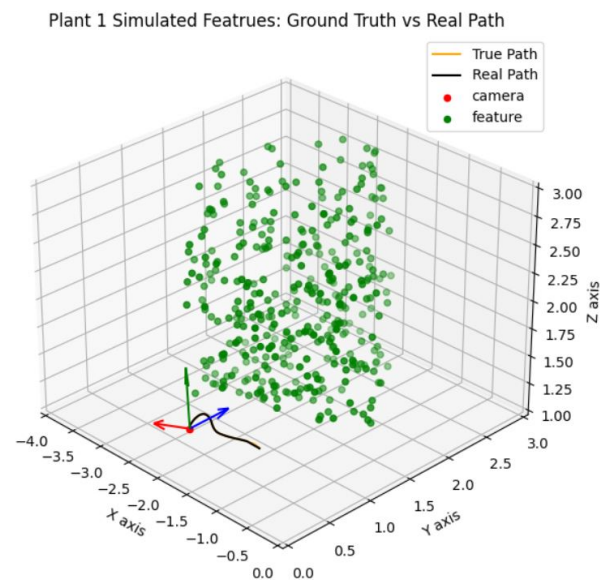


Figure 3. Plant 1 dataset, secondary view of Figure 2

- [2] "Orb-slam." [Online]. Available: <https://openslam-org.github.io/orbslam.html>
- [3] N. D. M. Andrew J. Davison, Ian D. Reid and O. Stasse, "Monoslam: Real-time single camera slam."
- [4]
- [5] "Opencv feature marching." [Online]. Available: [https://docs.opencv.org/master/d3/dc3/tutorial\\_py\\_matcher.html](https://docs.opencv.org/master/d3/dc3/tutorial_py_matcher.html)

Table 3 Simulated Features: Ground Truth vs Real Path

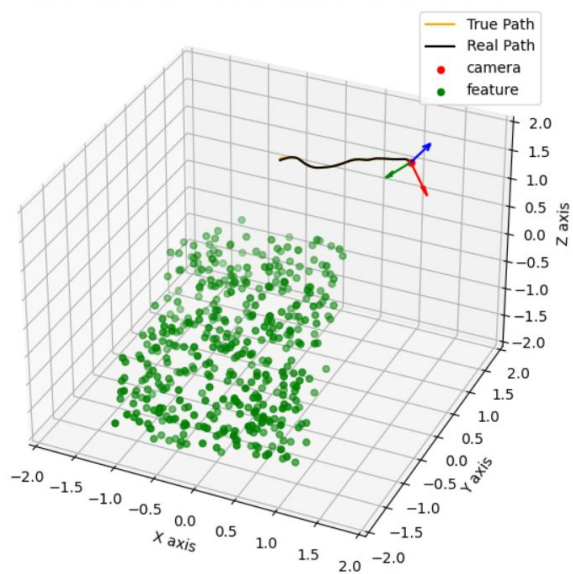


Figure 4. Table 3 mono dataset, simulated trajectory plotted against ground truth trajectory

Table 3 Simulated Features: Ground Truth vs Real Path

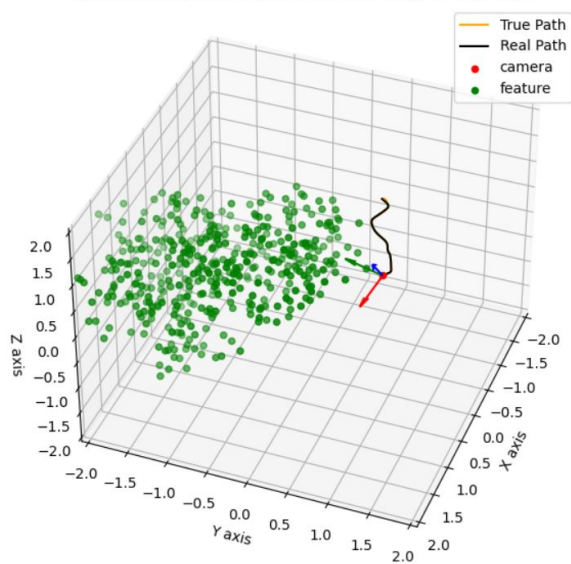


Figure 5. Table 3 mono dataset, secondary view of Figure 4

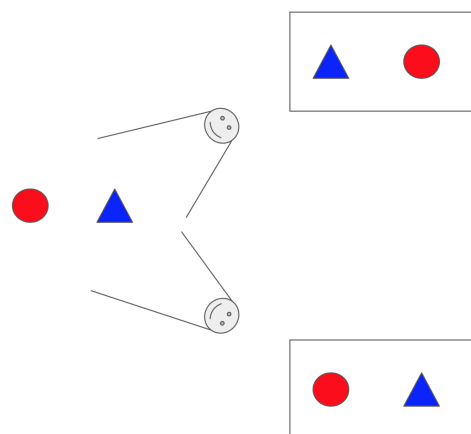


Figure 6. Parallax Figure

[6] A. Zisserman and R. Hartley, "Multiple view geometry in computer vision, 2003."