SLAM for Cyclopes: A MonoSLAM Tutorial

Anuj Gargava, Ayush Chowdhary, Manuel Soto, Muyang Wang, Phillip B Ring
University of Michigan, Ann Arbor, MI

***Abstract*—We seek to solve the general SLAM problem by implementing the Monocular-based SLAM approach and model pioneered by Davison, et al. [1]. In our methods, a slight deviation from the original paper is made; instead of warping image patches, the SURF algorithm runs on image patches to generate an adequate feature descriptor and a particle filter is used for initial depth estimation of the feature points. We conclude that the algorithm works quite well for small, rigid environments. As the scale of the problem increases, however, accuracy decreases due to the high-dimensionality of the state space as well as the difficulty of properly tuning hyper-parameters such as thresholds for data association.**

I. Introduction

The Problem of Simultaneous Localization and Mapping (SLAM) is one of the central problems in the field of Robotics. Much of the early days of SLAM research was based in using lidar, radar, sonar, and stereo-paired cameras for observing the world. Davison, et al [1] proposed a novel approach over a decade ago, using a single, monocular camera. Dubbed MonoSLAM, the paper involves an EKF-SLAM based approach with image patches as features. Some components of the traditional EKF SLAM algorithm are modified in order to adapt to the problem being addressed. Methods used and variations made are explained in further detail below.

II. Fundamentals of Monocular SLAM

MonoSLAM is essentially the SLAM problem using a single camera as a sensor for the system. This comes with a few unique challenges. One of the biggest challenges seen in the implemention of MonoSLAM is determining a detected feature's location in $\mathbb{R}^3$. To solve this problem, we implemented methods such as inverse depth parametrization and particle filters which are explained in greater detail below. Another unique challenge is how to describe the motion model for a freely moving camera in $\mathbb{R}^3$.

A. State Representation

The state vector is characterized by the stacked state estimates of the camera x_v and the landmarks l_i. The camera state vector x_v comprises of a metric $\mathbb{R}^3$ position vector r^W, orientation quaternion q^{RW}, velocity vector v^W, and angular velocity vector ω^R relative to a fixed world frame W and robot frame R carried by the camera (13 parameters):

$$x = (r^W \ q^{RW} \ v^W \ \omega^R)^T \tag{1}$$

The 3D locations of the features m_i are represented using Inverse Depth Parameterization [2] by a six dimensional vector as shown in Equation 2. The l_i vector encodes the ray from the first camera position from which the feature was observed by x_i, y_i, z_i, the camera optical center, and θ_i, ϕ_i azimuth and elevation (in the world frame) defining unit directional vector $m(\theta_i, \phi_i)$ as depicted in Figure 1. The point's depth along the ray d_i is encoded by its inverse $\rho_i = 1/d_i$. The 3D world coordinate of the feature can be computed using Equation 3 and 4.

$$l_i = (x_i \ y_i \ z_i \ \theta_i \ \phi_i \ \rho_i)^T \tag{2}$$

$$\begin{pmatrix} X_i \\ Y_i \\ Z_i \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} + \frac{1}{\rho_i} m(\theta_i, \phi_i) \tag{3}$$

$$m(\theta_i, \phi_i) = (cos\phi_i sin\theta_i, \ -sin\phi_i, \ cos\phi_i cos\theta_i)^T \tag{4}$$

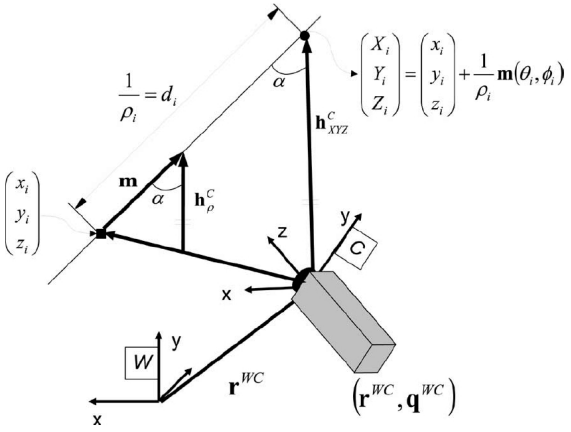

Fig. 1. Inverse Depth Parameterization

B. Motion Model and Prediction

The motion model for a freely-moving camera in $\mathbb{R}^3$ at first seems difficult to generate. The Davison paper [1] assumes a constant velocity, constant angular velocity model. This means that both control inputs are effectively zero. In this method, a smooth trajectory for the camera is also assumed, with relatively small accelerations. The Bayesian prediction and update steps occur during every image frame, which, at 30 Hz, leads to a near negligible motion in a small-scale setting.

The motion model and camera state vector are defined below in Equation 5. **r** is defined as the camera's pose in the world, **q** is defined as the unit quaternion centered at the camera pose relative to the world, and **v** and ω are the linear and

angular velocity of the camera. The control inputs, or impulse accelerations, are defined below as $\mathbf{V}$ and Ω in Equation 6.

$$\begin{pmatrix} r_t^W \\ q_t^{WR} \\ v_t^W \\ w_t^R \end{pmatrix} = f_v(x_{t-1}, u_t) = \begin{pmatrix} r_{t-1}^W + (v_{t-1}^W + V_t^W)\Delta t \\ q_{t-1}^{WR} \times q((\omega_{t-1}^R + \Omega_t^R)\Delta t) \\ v_{t-1}^W + V_t^W \\ \omega_{t-1}^R + \Omega_t^R \end{pmatrix} \tag{5}$$

$$u = \begin{pmatrix} V^W \\ \Omega^W \end{pmatrix} = \begin{pmatrix} a^W \Delta t \\ \alpha^W \Delta t \end{pmatrix} \tag{6}$$

The covariances of our motion model and motion noise are trivially defined in the same way as the EKF-based approach for non-linear models. Equations for the covariances are defined below.

$$\Sigma_{t+1} = \frac{\partial f_v}{\partial x} \Sigma_t \frac{\partial f_v}{\partial x}^T + \frac{\partial f_v}{\partial u} M \frac{\partial f_v}{\partial u}^T \tag{7}$$

$$M = \begin{pmatrix} \sigma_a^2 & 0 \\ 0 & \sigma_\alpha^2 \end{pmatrix} \tag{8}$$

C. Measurement Update

1) Measurement Model: In MonoSLAM, the measurement model corresponds to mapping the 3D landmarks location to the 2D pixel coordinate in the image frame using both the camera projection model and the radial distortion model as shown in Figure 2. For a given camera pose and landmark location in the map, the projective camera model returns the projection of landmarks in 2D image frame. The radial distortion model is applied to handle the distortion by the camera lens.

$$z = h(x; m) + \delta \tag{9}$$

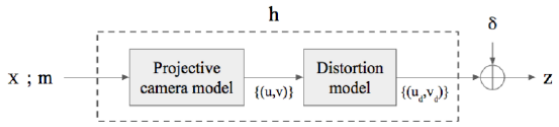

Fig. 2. Measurement Model

2) Measurement Prediction: The expected measurements $\hat{z}$ are computed by using the measurement model explained above and the predicted camera pose $\bar{\mu}$. This is simply to project landmarks in the map onto the image plane determined by the new camera pose. After predicting the measurements, the new frame is loaded and the observation is made.

3) EKF Update: The landmarks in the new frame are matched with the map using the SURF feature descriptor which is discussed in detail in the next section. The matched landmarks are used for the correction step for which the standard EKF SLAM framework is used. We compute the innovation (i.e. difference between the new measurement z and predicted measurement $\hat{z}$) to quantify how the estimates x and Σ should be corrected.

$$K_t = \bar{\Sigma}_t {H_t}^T (H_t \bar{\Sigma}_t {H_t}^T + Q_t)^{-1} \tag{10}$$

$$\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t)) \tag{11}$$

$$\Sigma_t = (I - K_t H_t)\bar{\Sigma}_t \tag{12}$$

D. Feature Matching and Data Association

The goal of the feature matching step is to obtain the measurement z by observing the landmarks in the map. It is noted that, corresponding to every landmark present in the map, there is an image template stored from the frame during which it was initialized. To obtain the measurement z, the search for the best matched template is performed only in the local region around the predicted measurement $\hat{z}$ in order to reduce the computational cost.

When a new frame is observed, the feature matching step aims to find the most likely pixel location corresponding to a particular landmark in that new frame. The most likely pixel location is determined as the one with the maximum correlation with the initial template of the landmark. There are two main challenges that need to be taken care of in order to use this method for feature matching: 1) Due the the camera movement, the appearance of a particular landmark changes over time and 2) Computing the correlation for each of the pixel around the predicted measurement $\hat{z}$ is cumbersome.

In order to overcome the above mentioned problems, we suggest a few methods: 1) correlation is computed only for salient pixel points around $\hat{z}$ and 2) image template for a particular landmark is computed based on "Speeded up robust features" (SURF) [3]. The correlation between two image templates is computed using the SURF descriptor values instead of raw pixel values.

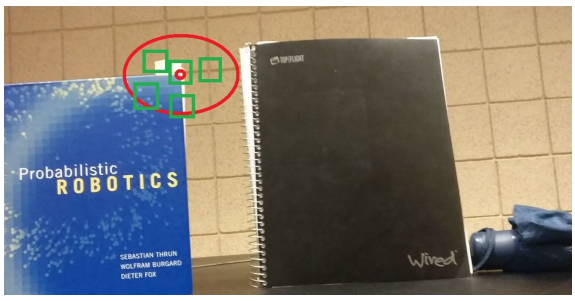

Fig. 3. Feature Matching

As computing the correlation for each pixel is inefficient, we focus only on salient points in the local region around the

predicted measurement $\hat{z}$. As shown in Figure 3, the red ellipse represent the 95% confidence region around $\hat{z}$. The approach of Davison et al generates image templates for each of the pixels inside this 95% confidence region and computes the correlation between each of them. In our implementation, we detect salient points inside the ellipse using corner detection techniques suggested in Shi and Tomasi [4] and generate SURF descriptors only around those points, shown by green boxes in Figure 3. This approach usually yields 10-100 times less the number of patches for each ellipse while preserving the saliency of the templates.

E. Map Management

After processing an image frame and associating landmark features, the landmarks in the map are evaluated on their consistency and significance. A finite state machine detailing this process is included in Appendix B, Figure 15.

1) Feature Detection: In order to have sufficient information about the environment in the map, new corner points are detected using the eigenvalues of the gradient image whenever the total number of matched landmarks in every frame is less than a certain threshold. Initially, the 200 strongest salient points are detected in the image and one of them is picked at random. If no other feature points in the map lie around this point, this new feature point is initialized, otherwise the procedure is repeated with another point. This is to prevent two feature points from being very close to each other. The detection of corner points is shown in Figure 4 below.

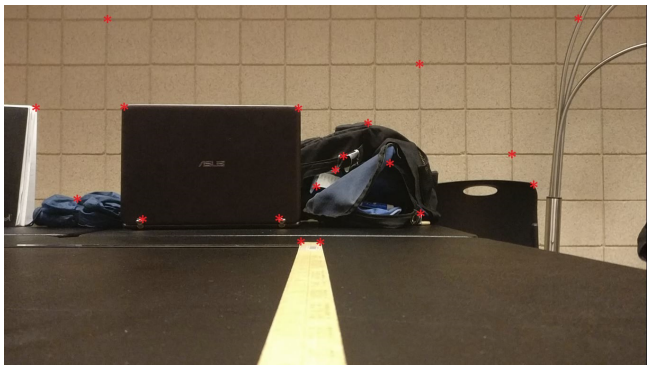

Fig. 4. Corner Detection

2) Feature Initialization: As shown in Equation 2 and Figure 1 previously, features are initialized and augmented to the state using inverse depth parametrization since inverse depth is a linear function. This makes it suitable for features at high parallax whose initial location may be highly uncertain. The last component of the landmark vector ρ, the inverse depth, is seen as a one-dimensional probability density over ρ and is therefore represented by a 1D particle distribution. A particle filter described later is implemented to solve for the distribution of ρ.

3) Feature Deletion: A landmark is eliminated from the map if it has not been associated to a feature in multiple successive image frames or, more specifically, if it meets the following criteria:

$$\frac{No.\ of\ Times\ Matched}{No.\ of\ Attempts\ to\ Match} < Threshold \qquad (13)$$

This occurs when either the camera's field of vision is no longer near the landmark or if it is a spurious landmark that was initialized with an inconsistent observation. A good starting threshold value is 0.5.

F. Depth Initialization - Particle filter

The feature detection procedure yields a landmark's orientation from the perspective of the camera. This, along with the estimate of the camera's location, will guarantee a single line where the landmark should be located. However, there is no way to determine the depth using information from a single frame. The simplest solution is to assign depth a constant initial value, and let the EKF correct this over time. The initial value may be hard to determine, and a poor initial value will yield large errors with respect to ground truth. In real world implementations, the lifetime of landmarks are not stable. Only a limited numbers of landmarks are maintained on a resource-tight embedded system, and those unmatched will be eliminated. Some spurious landmarks with extremely short lifetime of around 5 iteration will harm overall performance greatly.

Using a particle filter as a depth estimator was introduced to reduce initial uncertainty. Aware that the measurement error in depth is significantly larger than initial camera position and viewing direction, the error in initial camera position $[X_i, Y_i, Z_i]$ and viewing direction $m(\theta_i, \phi_i)$ are eliminated compared to depth parameter ρ in this project.

With this assumption, we store the camera position $[x_i, y_i, z_i]$ and direction θ, ϕ as constants during every iteration of the algorithm. Only the depth variable ρ is maintained as a state variable in the filter. A landmark will only be used in EKF update if the depth of the landmark converges to a proper range as shown in Figure 5

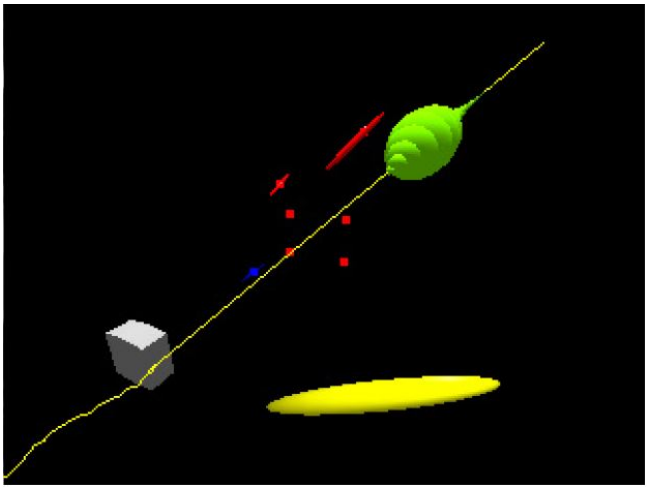

Fig. 5. Particle Filter

Algorithm 1 Particle filter initialization

function INITIALIZEPF(i, z_i)
▷ i: landmarkId
▷ z_i: measured 2D position for landmark i in image
$[x_i, y_i, z_i, \theta_i, \phi_i]$=RECOVERWORLDCOORDINATE($z_i$)
$State.P.\{x_i, y_i, z_i, \theta_i, \phi_i\} \leftarrow \{x_i, y_i, z_i, \theta_i, \phi_i\}$
$State.P.inverseDepth_i \leftarrow$ LINSPACE$(0.2 : 10; 100)$
$State.P.depthProb_i \leftarrow$ ONES(100,1)/100
$State.P.isValidlandmark_i \leftarrow 0$
end function

1) Initialization: We maintained a list of variables in State.P to maintain the particle filter:

- `featureInverseDepth`
- `featureProbMatrix`
- `rhoMean`
- `rhoVar`

The prior distribution of inverse depth is assumed to be uniformly distributed over 0.2 to 10, corresponding to real world depth from 0.1m to 5m:

$$\rho = \mathcal{U}(0.2, 10);$$

In our implementation, we divided the distribution into 100 particles.

2) Update: We used 2D position of the image frame as a measurement to do the particle filter update. For each particle, we project its location to the image frame, and compare that with the measured value. The new probability distribution is be calculated using Bayes' rule, under the assumption that the error to the measured position is a normal distribution with zero mean and identity covariance matrix. A detailed algorithm is shown in Algorithm 2.

Algorithm 2 Particle filter Update

function UPDATEPF($L_{1,2,\ldots n}$, $z_{1,2,\ldots n}$)
▷ L_i: Location for landmarks i in world coordinate
▷ z_i: Measured 2D position for landmark i in image
for all landmarks L_i **do**
if $FeatureMatched(i)$ **then**
for all particle's inverse depth ρ_j **do**
$\hat{z}_i = [\hat{x}_i, \hat{y}_i]$ = PROJECTION(L_i)
$e = \|z_i - \hat{z}_i\|_2$
▷ update using Bayes rule
$P(\rho_i = \rho_i^j) * =$NORMPDF$(e, 0, \sigma)$
end for
NORMALIZE($\rho_i^{1,2,\ldots,100}$)
RESAMPLE(void)
end if
end for
end function

3) Integration with EKF update: In this project, the particle filter is only used to give an initial depth. Whenever a landmark is used in EKF update, the result from the particle filter will be ignored. Since an embedded system can only maintain a limited number of landmarks, we purge a landmark from EKF update if that landmark is not detected for a long time, and add a landmark if that landmark has a well converged depth value.

III. METHODS

A. Parameters and Calibration

1) Noise Parameters: Given that the motion model assumes constant linear and angular velocity for the camera, the camera's motion is propagated by the zero mean Gaussian noises of the control input. Therefore, we must set the variance parameters shown in Equation 8 to match the expected motion of the camera in the environment. Low values for M assumes smooth motions of small accelerations. A high M implies larger expected accelerations and displacements leading to larger increases in uncertainty at each time step. Therefore one would require many good measurements at each time step to further constrain the camera pose estimates.

2) Camera Calibration: Assuming a pinhole camera model, the camera parameters and intrinsic K matrix are obtained following the standard direct linear transformation (DLT) calibration methods with the image of a chessboard.

B. Generation of Data Set

Most publicly available datasets gathered for the purpose of testing MonoSLAM implementations either lack an explicit ground truth or specifications of the camera and parameters used. Therefore, we decided on generating our own datasets in a small restricted environment with simple motions and trajectory for which a ground truth is trivial. The setups for the two data sets are shown in Figures 6 and 7 below. The camera trajectory follows a straight line for the first data set and an upside-down U for the second data set. Rotations in both scenarios are limited to a minimum, therefore most of the camera trajectory is limited to two dimensions. These procedures are subject to some human error but this can be parameterized by the noise and thus accounted for. The datasets were taken on a commodity Android smartphone, with each frame downsampled by 3x from the original frame.

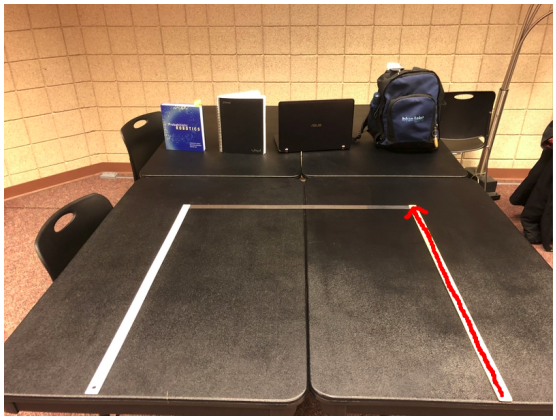

Fig. 6. Setup #1 for gathering of dataset

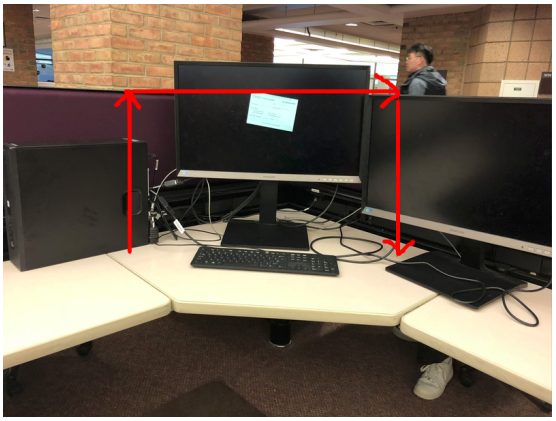

Fig. 7. Setup #2 for gathering of dataset

C. Implementation

Our implementation of MonoSLAM was developed using MATLAB; this choice was due to its familiar and easy-to-use debugging and visualization procedures. The Computer Vision Toolbox was used for identifying the corner points in an image patch and generating the feature descriptors for the landmarks.

IV. Results

In our implementation of MonoSLAM, we came upon a major finding: SLAM in the real world is *hard*. It is especially difficult in MonoSLAM, where the algorithm relies on the camera as the lone sensing modality. There are dozens of parameters that must be tuned correctly, and data association is difficult in even well-structured scenarios.

A. Trajectory Estimation

The first step in testing our implementation was to run it against a basic straight line test - moving the smartphone forward one meter. This was performed on the first dataset, and the results are shown in Figure 8. The trajectory is quite linear, yet there is noticeable drift in direction (ground truth is 1 meter in the negative y direction). This drift is mostly due to poor data association and unstable data collection that conflicted with the assumptions of our motion model (zero velocity, zero angular velocity) and noise parameters.

Our experiments on the second dataset yielded comparable results, shown in Figures 9 and 10. In this test, the estimated trajectory is similar in shape to the ground truth, with an archetypal 'Upside-Down U' shape. The algorithm diverges a few frames in due to the difficulties inherent with feature-based data association, and thus there is a considerable level of error between our estimates and ground truth. The algorithm does its best to recover, and thus is allowed to approximate the 'U' shape.

B. Algorithm Heuristics

As noted in Section II-E, the map management system involves the deleting of landmarks in order to decrease temporal and spatial complexity. Figure 11 shows the number of landmarks held in the state for each frame being processed.

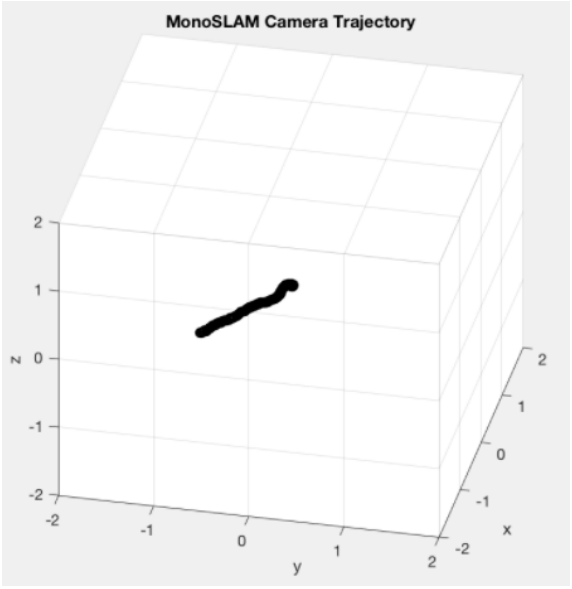

Fig. 8. Results of the Straight Line Test

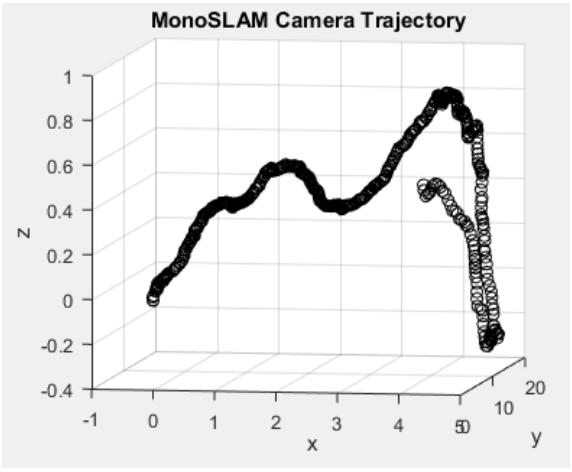

Fig. 9. Results of the Upside-Down U Test

Figure 12 yields the time, in milliseconds, of both prediction and update steps. It is interesting to note that the prediction times are mostly constant and fast, while the update times are more variable and slow, due to the map management process and fluctuation of the number of variables held in the state.

It is interesting to note that the average computation time per processed frame hovers around 108.1 milliseconds. Our datasets were collected at 30 Hz, and we downsampled the feed to 10 Hz in order to achieve 'real time' constraints. This yielded similar results in both cases, meaning that it can be run on real world embedded system under strict power and performance requirements.

C. Particle Filter Correctness Verification

This section verifies the correctness of the particle filter. Our tests are based on two criteria: rate of convergence and speed of convergence. Rate of convergence determines directly

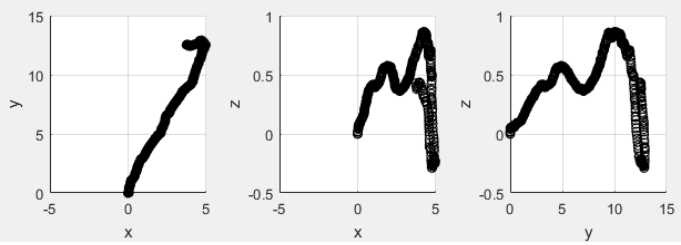

Fig. 10. Results of the Upside-Down U Test

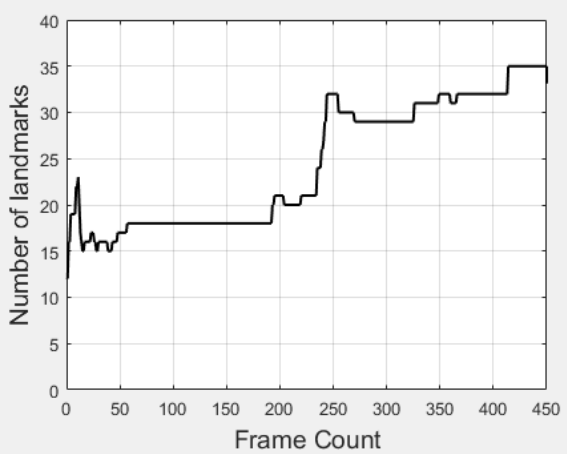

Fig. 11. Number of Landmarks in State at Each Frame

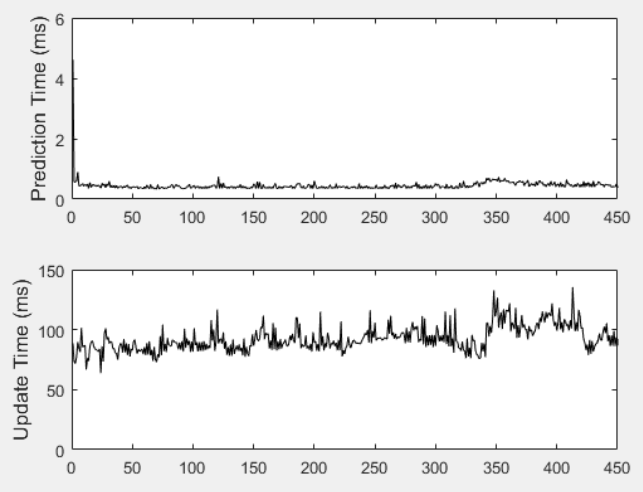

Fig. 12. Prediction/Update Times at Each Frame

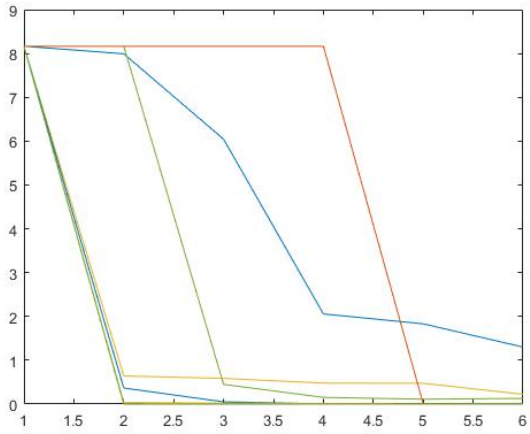

Fig. 13. variance of ρ for 12 particles over 6 iterations

whether a particle filter is suitable for the given scenario. If the filter fails to converge, then we should not use a particle filter for the purposes of this project.

The speed of convergence is also a very important characteristic of the particle filter. Not only is it a metric to determine efficiency of the filter, but for our implementation it will determine whether or not the filter will have time to converge. Since the entire MonoSLAM system is performed in real time, if the particle filter cannot converge during landmarks' lifetime, results will be useless. As mentioned in the map management section, some landmarks have a lifetime of around 5 frames. So if, for any feature, the filter takes more than 4 iterations to converge, it will be regarded as a failure.

In this section, we sampled through 12 landmarks in 6 iterations in our dataset to see the average rate and speed of convergence. The result is in Table I in Appendix C. Results are shown in Figure 13.

1) Rate of Convergence: As shown in the figure, 10 out of 12 particles successfully converged(variance less than 0.2) in 6 iterations. The system has an overall converge rate of 83%. Overall, rate of convergence is sufficient to work along with EKF.

2) Speed of Convergence: As shown in the figure, 7 out of 12 landmarks have their inverse depth converged to 0 (variance less than 10^{-6}) during the second iteration. and 8 out of 12 landmarks have a good result(variance less than 0.1) in three iterations. This means around 67% of captured landmarks can contribute to EKF SLAM for 2 iteration, which is good enough as an input to upper level SLAM.

3) Further Tests: Since all features/landmarks are generated automatically by a corner detector and matched by SURF, we were not able to track the ground-truth position of each landmark. So for now we cannot generate a goodness of fit test for the particle filter. Further work includes assigning specific landmarks to the particle filter and comparing estimation with ground-truth result.

V. Discussion

As seen in Section IV, the performance of Monocular SLAM highly depends on the environment and the visual information in the dataset. For the straight line trajectory in the first experiment, the model performs reasonably well within the expected error bounds. However, the results for the second dataset shows the sensitivity of the method to camera motion errors leading to incorrect data association.

It must be noted that the camera we used to craft our datasets with had custom auto-focusing and auto-exposure features baked into the firmware. This compromised the integrity of our datasets, as the focal parameters were constantly changing (we used MATLAB's camera calibration suite on a number of images, with different focal lengths). As well, SURF is not designed to be illumination invariant, and thus the captured descriptors were not as consistent frame-to-frame as they should have been. Without these factors in play, data association would likely have performed much better.

VI. Conclusion

In this paper we have implemented the titular MonoSLAM popularized by Davison, et al. Monocular vision-based SLAM is an ideal algorithm for low-cost and embedded platforms due to its inexpensive setup (only one camera), and has been used in many commercial products because of this. Specifically, the fields of Augmented Reality and Home Robotics have seen a boom due to its creation. The MonoSLAM paper came out over a decade ago, and since then, there have been much better solutions to Monocular-based SLAM. Particularly, ORB-SLAM, by Mur-Artal, et al [5] has yielded significant results.

VII. Acknowledgments

We would like to acknowledge Prof. Maani Ghaffari Jadidi for his support on this project.

VIII. Source Code

Our code is opened sourced on github under the GPLv3 license. The site may be found here: https://github.com/pringithub/monoSLAM

References

[1] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse, "Monoslam: Real-time single camera slam," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1052–1067, 2007.

[2] J. Civera, A. J. Davison, and J. M. Montiel, "Inverse depth parametrization for monocular slam," *IEEE transactions on robotics*, vol. 24, no. 5, pp. 932–945, 2008.

[3] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Speeded-up robust features (surf)," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346 – 359, 2008.

[4] J. Shi and C. Tomasi, "Good features to track," in *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, Jun 1994, pp. 593–600.

[5] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, "Orb-slam: a versatile and accurate monocular slam system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

APPENDIX A
FINITE STATE MACHINE OF MONOSLAM

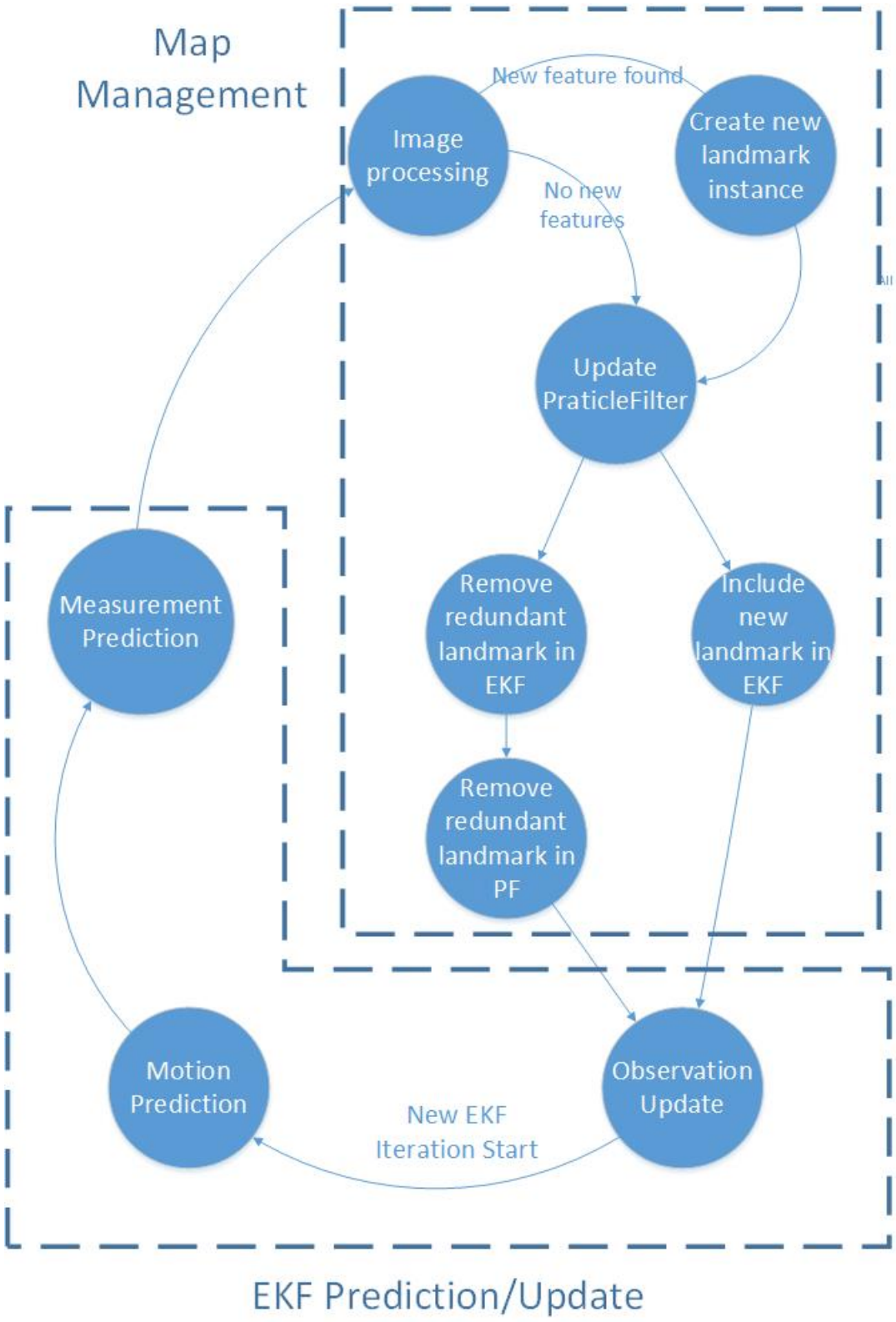

Fig. 14. monoSLAM FSM

APPENDIX B
PARTICLE FILTER FINITE STATE MACHINE

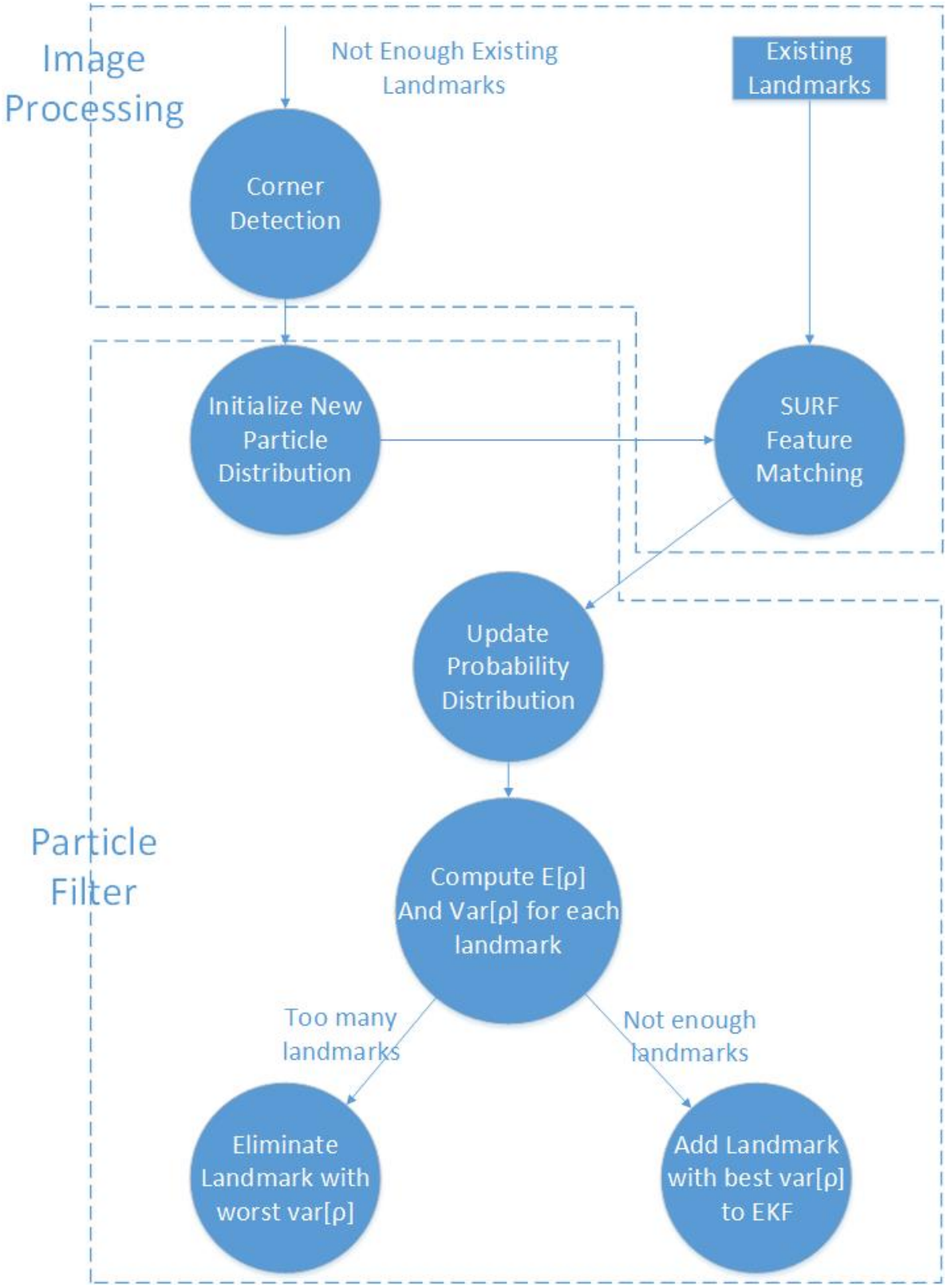

Fig. 15. Particle Filter FSM

APPENDIX C
PF VERIFICATION RESULT

TABLE I
INVERSE DEPTH VARIANCE(ρ) WITH 12 LANDMARKS IN 6 ITERATIONS

Iteration	1	2	3	4	5	6
L1	8.165016835	7.992859182	6.043733213	2.056774845	1.831506612	1.301357188
L2	8.165016835	0.000552664	0.000552664	0.000552664	1.93E-32	1.93E-32
L3	8.165016835	0.642563552	0.586586764	0.478877135	0.475270123	0.22421974
L4	8.165016835	1.93E-32	1.93E-32	1.93E-32	1.93E-32	1.93E-32
L5	8.165016835	8.165016835	0.449196944	0.153801186	0.111178475	0.129832832
L6	8.165016835	0.00676131	0.003061208	1.93E-32	1.93E-32	1.93E-32
L7	8.165016835	0.00176774	0.00048897	0.001681508	0.002308644	0.001625654
L8	8.165016835	0.366885343	0.051542741	0.007128773	0.002213594	0.000721206
L9	8.165016835	8.165016835	8.165016835	8.165016835	0.007426662	0.00401661
L10	8.165016835	0.037839819	0.020557322	0.013538299	0.010562342	0.00911307
L11	8.165016835	0.002127363	0.003543318	0.002088167	0.002328242	0.002414474
L12	8.165016835	0.004052866	0.00264181	0.001382639	0.001446332	0.001885328