Simultaneous Localization and Mapping (SLAM) in a single camera system

by

Isaac Tallack

Submitted in Partial Fulfilment of the Requirement for the

Degree of Master of Science in Communication and Information Engineering

Supervised by

Dr Thomas Popham

School of Engineering
University of Warwick
19/08/2020

Abstract

This paper discusses a way of solving the Simultaneous Localization and Mapping (SLAM) problem using only a single camera system. While this is challenging because of limited amounts of information available from a single camera, it provides benefits in terms of cost, weight, and speed of processing. The paper starts by breaking the problem down into three topics: target/video tracking, camera calibration/mapping and probabilistic filtering. For each of these topics, a suitable method is decided upon and an algorithm implemented in Python. The algorithms are tested on a combination of dataset videos as well as simulated scenarios and the quality of the algorithms discussed. Issues and challenges for each section are explored and discussed, and the algorithms iterated on to create a system that is effective under difficult scenarios. The result is a high-quality video tracking algorithm and SLAM system that works well using simulated data. Reasonable extensions to the work are suggested to develop upon the findings.

Contents

List of Figures

List of Code Listings

List of Tables

Acknowledgments

Thank you to my supervisor, Dr Thomas Popham, for his immense enthusiasm, support, and passion towards this project, and who has helped to shape my interests in engineering further. Thank you too, to my friends and family for their interest and encouragement throughout.

I hereby certify that this dissertation has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for a higher degree.

1 Introduction

1.1 The Problem

Simultaneous Localisation and Mapping (SLAM) is a computing approach to solving a problem where a system needs to determine where it is, fully autonomously. Specifically, localisation refers to determining the system position within an environment. Mapping refers to defining the environment around the system. The challenge of SLAM is performing both of these functions simultaneously based on one set of data being fed to the system.

SLAM has a wide range of uses and is important to progress towards fully autonomous systems, allowing them to have a sort of "awareness". In the transport sector, SLAM is useful in self-driving cars: being able to localise the position of the car as well as build up static (buildings, utility poles etc.) and non-static (other cars, pedestrians etc.) maps enables the car to follow set routes while avoiding dangers. SLAM can also be useful in understanding dangerous environments where it would be unsafe for humans to enter. For example, X. Fan et. al. (2019) discusses a SLAM approach for autonomous firefighting systems to enter environments with dangers such as high temperatures, lack of oxygen and thick smoke [1]. The algorithm can map the terrain, navigate through the terrain, and locate the source of the fire. In some situations, it may be advantageous to use a SLAM system to explore an unknown environment to plan a system of approach for human entry. SLAM can often be a relatively compute-efficient algorithm, meaning it can be implemented on lightweight systems like drones to improve positional accuracy and crash avoidance.

1.2 Project Specification

This project aims to create a robust algorithm to carry out SLAM on a video feed recorded by a single camera. This involves the following three sections: camera tracking to identify and track keypoints throughout the video; camera mapping to convert keypoint pixel locations into useable position data; probabilistic filtering to determine the most likely camera position and keypoint locations. Each of the three sections will be tackled separately and then combined at the end.

Camera tracking and camera mapping can be tested directly on the video data. To assess performance of these sections, video sequences with tracking points overlaid can be visually verified. To build the probabilistic filtering, synthetic data and simulations will be used to gradually build up the algorithm in a manner where complexity and randomness can be carefully controlled. This section can be assessed by plotting graphs and creating animations

to see how the filter does in comparison to the ground truth synthetic data. With all the sections combined, the algorithm can be tested on several video sequences and the results compared to the ground truth provided.

2 Literature Review

Four main sections of literature must be reviewed. Firstly, a suitable dataset must be chosen that provides sufficient data for testing and evaluation. Secondly, a video tracking algorithm must be chosen critically to select keypoints from video sequences and keep track of them throughout. Thirdly, a camera mapping approach must be selected to convert the feature pixel positions into real-world coordinates. Finally, a probabilistic filtering method must be chosen to make sense of incoming data and perform the camera localisation and world mapping.

2.1 Dataset Selection

A few requirements were drafted for selecting a suitable dataset for testing the SLAM algorithm. Most importantly, the dataset must provide video sequences recorded from a camera with calibration images (typically of a checkboard pattern) to allow for estimation of camera parameters. Secondly, a record of the ground truth camera position is required for evaluation of the SLAM model prediction against the actual position. Preferably, the dataset will provide a variety of scenarios with differing challenges (for example low-light, camera obstructions) so the robustness of the algorithm can be tested. Additionally, having some other sensor data recorded would be a bonus as testing the strength of the algorithm with camera-only measurements versus camera plus other measurements (e.g. accelerometer, gyroscope) could provide an extension to the project.

2.1.1 The Zurich Urban Micro Aerial Vehicle Dataset

The first dataset that was considered was the Zurich Urban Micro Aerial Vehicle Dataset [2]. This dataset provides video sequences from a camera mounted on a “Micro Aerial Vehicle (MAV)” flying at low altitude in a circular loop around the city of Zurich. In addition to high-resolution video, the dataset also provides GPS, inertial measurement unit (IMU), accelerometer, gyroscopic and barometric readings. Importantly, ground truth camera position data and calibration images are also provided. While this dataset meets most of the criteria set out, the lack of multiple scenarios with varying difficulties for SLAM and slow download speed on the host website (over 28GB taking over 18 hours to download) made it a second choice.

2.1.2 ADVIO: An Authentic Dataset for Visual-Inertial Odometry

The second dataset that was considered was the 'ADVIO' dataset [3]. This dataset provides video sequences recorded by a smartphone from a pedestrian walking. The dataset provides ground truth data on the movement of the camera as well as readings from the smartphone accelerometer, gyroscope, magnetometer, and barometric pressure gauge. In addition to these measurements, camera tracks from Apple ARKit, Google Tango and Google ARCore allow for comparison against some of the leading commercial visual-inertial tracking software. Regarding different scenarios with varying difficulties: this dataset provides 23 different sequences. Amongst the sequences are four different locations (both indoor and outdoor), changes in altitude (by stairs, escalator, or elevator), four different levels of people in the scene (none, low, medium or high) and sequences containing vehicles or no vehicles. Each sequence can be downloaded separately depending on what the algorithm is to be tested against. Based on the wide range of data and ease-of-use, this dataset will be used primarily.

2.2 Point Tracking Methods/Algorithms

2.2.1 Scale-Invariant Feature Transform (SIFT)

SIFT is an image matching algorithm that was designed to reliably match multiple features in a specific scene taken at different angles or viewpoints. The original paper, 'Distinctive Image Features from Scale-Invariant Keypoints' by D.G Lowe, describes the method and its advantages over other image matching algorithms [4]. The main benefit of the SIFT algorithm is that the rotation or scale of the scene in matching images does not matter and the algorithm is also fairly robust against differences in illumination and changes in 3D perspective. These properties of the algorithm are thanks to four key stages that are used to select and describe each feature: scale-space extrema detection, keypoint localization, orientation assignment and keypoint descriptors. A description and explanation of each step can be seen below [5][6].

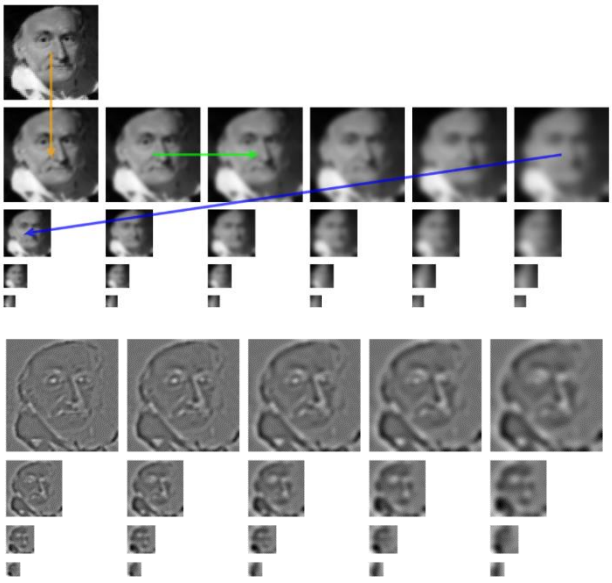

Figure 1: Gaussian (top) and difference-of-Gaussian (bottom) pyramids [5]

Scale-space extrema detection: a Gaussian pyramid of images is created by iteratively blurring and then down-sampling the input image. The purpose of this is to ensure general structures are picked up if the scale of the scene decreases. If features were only determined at full resolution, then if only small details were selected, these would no longer be visible if the scene scale were decreased. Having made a Gaussian pyramid of images, a difference-of-Gaussian pyramid is made. The difference of Gaussian images are a good approximation of the Laplacian of Gaussian image filter that is used for edge detection. This is done by subtracting the pixel values of the next blurred image to the right [7] (as can be seen in the bottom image of figure 1). This allows for a way of determining maxima and minima areas in the image. Specific keypoints are then found by choosing pixels in these difference-of-Gaussian images that are greater than or less than their surrounding 8 pixels as shown in figure 2.

Figure 2: Maxima and minima detection [5]

Keypoint localization: in this stage, the nature of the keypoints are observed to reject keypoints that are not strong enough. Specifically, keypoints that have low contrast or lie on an edge are discarded. Points with low contrast are not preferred as they are sensitive to noise. Edges are avoided - while it is easy to locate an edge in one direction (perpendicular to the edge), it is often hard to determine where the keypoint is along the edge (low contrast parallel to the edge). These are discarded by comparing the gradients along the keypoints in different directions.

Orientation assignment: for each keypoint a 'reference orientation' is calculated. This describes the direction in which the majority of the surrounding pixels point to based on the gradient change between them. This is done by calculating the difference between pixels in various directions split into a set number of 'bins' containing a range of angles. An example of this can be seen in figure 3. At this stage, keypoints can be discarded if there is no clear direction to the gradient or if the keypoint is too close to the edge to examine surrounding pixels.

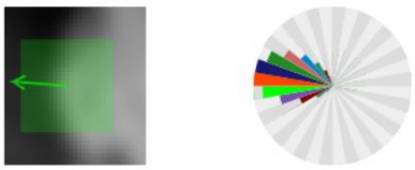

Figure 3: Orientation assignment [5]

Keypoint descriptors: finally, descriptors are used to describe the characteristics of each keypoint for later matching. To make the algorithm non-reliant on rotation, the descriptor is calculated relative to the 'reference orientation' calculated previously.

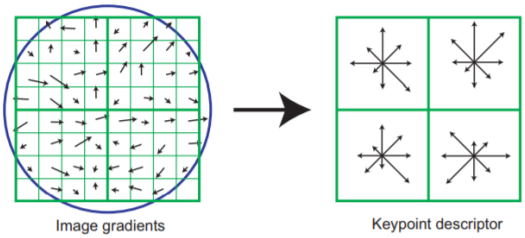

Figure 4: Keypoint descriptor [8]

Also, the descriptors that define each point only describe the relationship between gradients in a different direction and not the actual gradient strength which allows the algorithm to match images with different lighting. The image in figure 4 shows how the gradients are analysed in the area around the keypoint and the descriptor array can be calculated by describing the makeup of gradients in a subregion.

Another technique mentioned by Lowe is the 'ratio test' which is used to filter out keypoints based on their quality and usefulness. This is done by looking at the closest and second-closest match for a specific keypoint. If the Euclidean (straight-line) distance between the keypoint of interest and the closest match is sufficiently bigger than the distance between the keypoint of interest and the second-closest match then the keypoint is good. This is because only one of the keypoints is actually the correct match, so if it is easily distinguishable and different to any other keypoint then it is extremely useful and less likely to be mismatched. A ratio is used as it does not rely on absolute differences but instead proportional differences making it robust for different scenarios. The multiplier of the ratio can be adjusted to either give fewer keypoints of higher quality and reliability or a greater number of keypoints with lower quality and proneness to error.

C. Yang et. al. (2016) modifies the traditional SIFT algorithm to make it better suited to video tracking [9]. The main modification is creating a 'searching window' based on where the keypoint of interest was in the previous frame. It is assumed that as a video, consecutive frames are likely to be quite similar so processing time and accuracy can be improved by not searching for the keypoint in the whole image but in a localised area instead. The results show the tracking of a car in a video sequence. The car, despite the background constantly changing and even a fence occluding it in one frame, is robustly tracked throughout.

2.2.2 Speeded-Up Robust Features (SURF)

The SURF algorithm was first published seven years after the first SIFT paper to create a faster, more computationally efficient alternative which is better suited to real-time applications. SURF is based on the work of the SIFT algorithm although makes some alterations for an alternative approach. The three main steps of the SURF algorithm are novel detection, description, and matching. A description and explanation of each step can be seen below [10][11].

Novel detection: The SURF algorithm uses a different approach for approximating the Laplacian of Gaussian by using box filters (the average of surrounding pixels) and integral

images. A Hessian matrix is used to calculate the gradients of the Laplacian of Gaussian approximation and is also efficient. While SIFT requires subtracting each Gaussian derivative image from the original one, which takes $O(n^2)$ time, the SURF approach takes O(1) time.

Description: Similar to SIFT, the SURF algorithm then determines the 'reference orientation' of the keypoint. It does so slightly differently, by sliding a 60° window over the Haar-wavelet response of the keypoint. When the direction of maximum sum is found, this is denoted the 'reference orientation'. To describe each keypoint, the wavelet response in both the horizontal and vertical directions are calculated with the region of interest rotated based on the reference orientation. Each descriptor in SURF is length 64 as opposed to length 128 in SIFT which explains some of the speed increase.

Matching: matching is based on the Euclidean distance between descriptors therefore the smaller descriptors proposed by SURF has a significant increase in speed while still boasting enough descriptors to maintain accuracy.

Mistry, Dr & Banerjee, Asim (2017) compared the SURF algorithm to SIFT. They found that SURF outperformed SIFT in terms of robustness to rotation, blurring, warping, RGB noise [12]. However, SIFT performed better with scale changes and neither algorithm performed better than the other in scene illumination changes. In terms of time complexity, they found that SURF was 3 times faster than SIFT.

2.2.3 Shi-Tomasi Corner Detector

This algorithm is used to find 'good' points that are easily identifiable and therefore easy to track. 'Good' points are usually corners that have high contrast. Edges are not so good as, while there is high contrast, in the parallel direction it is hard to distinguish a specific area along the edge.

In the Shi and Tomasi (1994) paper, they propose a method of using the eigenvalues of the matrix of intensities in a given window in varying directions to determine whether a window contains an edge, a corner or neither [13].

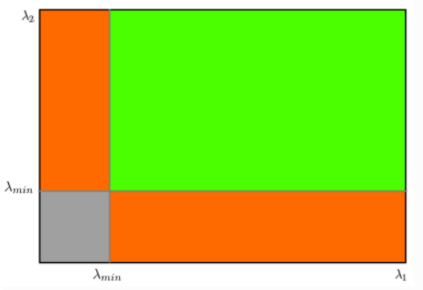

Figure 5: Shi-Tomasi corner thresholds [14]

Figure 5 shows when a window will be defined as a corner (in green), when it will be defined as an edge (in orange) or when it is neither (in grey). For a point to be described as a corner both eigenvalues must meet a certain threshold [14].

2.2.4 Lucas-Kanade Method for Optical Flow (KLT)

The Lucas-Kanade Method for Optical Flow [15] is a method of video tracking first published in 1981. The method can track a feature through many regular consecutive frames by estimating its motion. Optical flow compares frames to estimate motion between them and therefore similar frames are preferred. A demonstration of this can be seen in figure 6.

The method works on the principle that the intensity of a pixel will be the same in each time step, such that an equation for the intensity of a specific pixel is given by the following where *t* is time, *x* and *y* are the pixel coordinates, I_1 and I_2 are the first and second frame respectively.

$$I_1(x, y, t) = I_2(x + \Delta x, y + \Delta y, t + \Delta t) \tag{1}$$

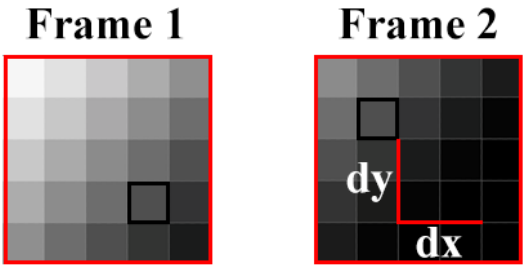

Figure 6: Optical flow frame comparison

Taking the Taylor expansion of this equation gives the optical flow equation as seen in equation 2.

$$\frac{\partial I(w)}{\partial x}u + \frac{\partial I(w)}{\partial y}v = -\frac{\partial I(w)}{\partial t} \tag{2}$$

This is saying that the summation of the gradient of the small area around a pixel centred at w, multiplied by the velocity, in both the x-direction and y-direction (*u* and *v*) is equal to the negative of the total change in intensity with respect to time. The Taylor expansion with removed higher-order terms treats the area around the centre pixel as if it is a simple gradient and that the window is moving in the direction of the gradient. Here there are two unknowns (*u* and *v*) but only one equation and therefore it impossible to solve from this alone. However, by assuming the area around this pixel moves in the same direction and magnitude it is possible to generate more of these equations that will share the same unknown variables [16].

For a specific window, further windows are created around the point and a set of these optical flow equations are created for each window in that 'neighbourhood'.

These set of equations can be expressed in matrix form as seen in equation 4.

$$A \cdot v = b \tag{3}$$

$$\begin{bmatrix} \frac{\partial I(w_1)}{\partial x} & \frac{\partial I(w_2)}{\partial y} \\ \frac{\partial I(w_2)}{\partial x} & \frac{\partial I(w_2)}{\partial x} \\ \vdots & \vdots \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \frac{\partial I(p_2)}{\partial t} \\ \frac{\partial I(p_2)}{\partial t} \\ \vdots \end{bmatrix} \tag{4}$$

To solve this matrix equation for *u* and *v*, with so many known variables that is now an over-constrained problem, the Lucas-Kanade method proposes using a least-squares approach so that there is the least sum error in all the equations in the system – the best approximate solution [17].

$$A^T A v = A^T b \tag{5}$$

$$\begin{bmatrix} u \\ v \end{bmatrix} = (A^T A)^{-1} A^T b \tag{6}$$

To address the issue that the Lucas-Kanade optical flow method only works for small movement between frames, the OpenCV library uses pyramids of different scales of images whilst maintaining the same pixel window size. This means that at small resolutions fast or big movements will move fewer pixels than they would at full resolution [18].

2.2.5 Comparison of video tracking methods

Newman, P. and Ho, K. (2005) discuss one of the challenges of SLAM: 'loop closure' [19]. Loop closure is the task of determining whether an area has already been passed through by the camera when coming back a second time. According to the paper: "Reliable loop closing is both essential and hard. It is without doubt one of the greatest impediments to long term, robust SLAM". The paper outlines how to achieve loop closure, a database of image-feature descriptors must be stored. In the previous section, three methods of video tracking were discussed: SIFT, SURF and KLT. Both the SIFT and SURF algorithm describe and match points based on descriptor arrays: each feature point is uniquely identified. On the other hand, the KLT tracker just uses motion models to keep track of points and hence cannot locate a point without having seen the previous frame. Therefore, both SIFT and SURF would have an advantage in performing loop closure. However, SIFT and SURF are primarily image matchers and, while the base methods can be adapted for video tracking (as seen in C. Yang et. al. (2016) [20]), they require additional coding work. The KLT tracker is designed

specifically for video tracking and therefore has no additional overhead. For this project, where time is a constraint, the KLT tracker will be used for its ease-of-use.

2.3 Camera Geometry and Mapping

Being a visual-only SLAM problem, the information available is a 2D image which is a representation of the 3D environment. Some way of interpreting the 2D image as real-world points in the environment must be considered. This is not easy as there is a loss of dimension in the image. The most simplistic model is the pinhole camera model. The pinhole camera model refers to a small hole in which light from the environment is projected through onto a 2D imaging plane some distance behind the pinhole.

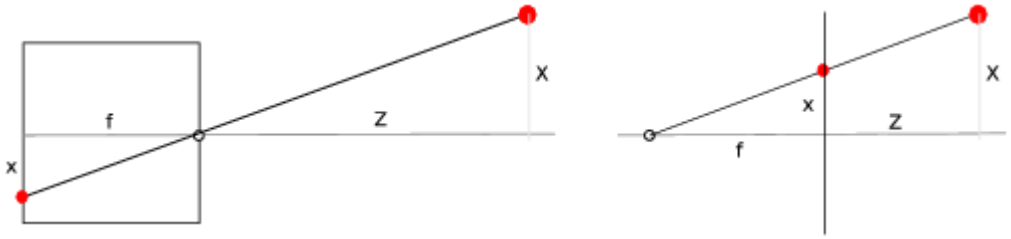

Figure 7: Pinhole camera model [21]

Figure 7 shows a visual representation of the pinhole camera. On the left, X is the real-world point, the pinhole is at the origin and x is the projection. On the right is a virtual representation where the pinhole is still at the origin, but the projection plane is placed in-between the object and the pinhole. In both representations the focal length of the camera (distance between the projection and the pinhole) is *f*.

In the camera pinhole model, there are two sets of parameters: extrinsic parameters and intrinsic parameters. The extrinsic parameters refer to the camera as an object and specify the position of the camera and the camera orientation. The intrinsic camera parameters refer to the imaging properties of the camera such as focal length and image distortion.

In the simpler direction of going from a real-world coordinate (X, Y, Z) to a projection in the 2D plane (x, y) the following equations can be derived as seen in 7 and 8.

$$\frac{x}{f} = \frac{X}{Z}, \qquad \frac{y}{f} = \frac{Y}{Z} \tag{7}$$

$$x = f\frac{X}{Z}, \qquad y = f\frac{Y}{Z} \tag{8}$$

As the gradient of projection is constant throughout (see the diagram), the mapping either side of the pinhole can be related. As this is a projection of a 3D world onto a 2D plane, the depth, Z, is lost.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \frac{f}{Z} \begin{bmatrix} X \\ Y \\ Z/f \end{bmatrix} \tag{9}$$

The projected image position (x, y) does not change if the ratio between *f/Z* changes, therefore, the camera matrix, C, (the 4 x 4 matrix) can be derived as simply:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{10}$$

However, while this is the simplest model, other intrinsic parameters of the camera should be included. Firstly, as this model is aiming to model a camera producing a digital image and not just a projection, the spatial coordinates of (x, y) should be measured as sampled pixels and not distances. Hence, a scaling factor is multiplied by each of the focal distances in the x and y directions (s_x and s_y respectively).

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x f & 0 & 0 & 0 \\ 0 & s_y f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{11}$$

Secondly, it must be considered that the imaging sensor of the camera may not be perfectly central to the pinhole and hence an offset must be added to the pixel locations in both directions (x_0 and y_0 respectively). Additionally, a 'skew' parameter, α, can be added to correct for when the projected image is sheared and the (x, y) directions are not at right-angles.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x f & \alpha & x_0 & 0 \\ 0 & s_y f & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{12}$$

This completes the intrinsic matrix of the camera which describes how the camera converts light entering the pinhole to a digital image.

$$M_i = \begin{bmatrix} s_x f & \alpha & x_0 & 0 \\ 0 & s_y f & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{13}$$

Regarding the extrinsic parameters – how the camera is positioned in the environment; another matrix can be created. Two different extrinsic parameters are considered: rotation and translation.

The rotation matrix, R, is a 3 x 3 matrix that describes how each point in the real-world translates to the camera image. The translation matrix, T, is a 3 x 1 vector that describes how far real-world point has moved in the camera image. The extrinsic parameter matrix is as follows:

$$M_e = \begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R & T \\ \mathbf{0}^\top & 1 \end{bmatrix} \tag{14}$$

By multiplying the intrinsic and extrinsic matrices together the final camera matrix (3 x 4) can be produced [21][22].

$$M = \begin{bmatrix} s_x f & \alpha & x_0 & 0 \\ 0 & s_y f & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{15}$$

Camera calibration is done to estimate both the intrinsic and extrinsic camera properties by taking multiple pictures of a known pattern (often a checkboard) at different angles with the camera position fixed and using linear-least squares error to estimate the parameters that best fit the images.

The obvious way of going from a 2D image back to the 3D world is just to invert the camera matrix as seen in equation 16.

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = M^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{16}$$

However, another dimension cannot be determined so all the coordinates will be in relation to the depth, Z, and in the form:

$$X = \frac{xZ}{f}, \qquad Y = \frac{yZ}{f}, \qquad Z = Z \tag{17}$$

So, there will be an infinite line of depths on which the real-world point will fall.

Davison, A.J et. al. (2007) discusses this issue specifically and explains how this issue is part of the reason why "there have been relatively few successful vision-only SLAM systems" [23]. In this study, a single image SLAM implementation is initialised by pointing it at a target where several known features are mapped with zero uncertainty. This is done so the system can start

with a specific map scale as well as having some points with depth to work from. From then on, a 3D line is drawn through each new tracking point from the estimated camera position to infinity such that the depth is unknown. However, the tracking point is not entered into the SLAM calculations straight away. Instead, the motion of it is examined over a short period with the sole intention of determining its depth. Once the depth guess is deemed good enough, the point is added into the SLAM calculations and can start contributing to the state estimates. While some useful data may be lost by not entering the target into the SLAM algorithm straight away, the limitations of a single camera SLAM system makes this a reasonable trade-off to prevent model divergence through inaccurate measurements.

2.4 Kalman Filter and Extended Kalman Filter (EKF)

The Kalman filter is used to estimate the state of a system based on a set of measurements that are often noisy and imprecise, with the key principle of combining measurements with different uncertainties to create an estimation better than any single measurement alone.

To implement the EKF in Python the 'FilterPy' library will be used which provides functions for a wide range of probabilistic filtering algorithms. The library has comprehensive documentation and is also written alongside the book 'Kalman and Bayesian Filters in Python' by Roger R. Labbe Jr, which provides an excellent introduction to Kalman filters in Python with many numerical examples and explanations [9].

2.4.1 Kalman Filter

The Kalman filter has two main steps in its operation: predict and update. The prediction step uses the state estimate in the last time step and knowledge of the system dynamics to predict the next step. The update function then takes this estimate and corrects it based on the measurements received [24]. The equation for the predict step can be seen in 18.

$$x_{k+1} = Fx_k + Bu_k + w_k \tag{18}$$

The *F* matrix is called the 'state transition' matrix and describes how the system state is expected to change compared to the previous state. For example, in a linear velocity model, the state transition matrix would increase the displacement state based on the velocity in the previous state and the time passed between states. The *B* matrix relates the input to the system to the next state. For example, to track the position of a car, if some torque had been applied to the wheels, the *B* matrix would describe how that torque affects the acceleration of the car. Finally, the w_k term is called the process noise and describes how there is some uncertainty to

the states. Without this uncertainty, the update step would not be needed as the prediction would describe the system in full. The equation for the update step is shown in 19.

$$y_k = Hx_k + v_k \tag{19}$$

The y_k term is the measurement taken. The H matrix describes how the state estimate, x_k, relates to the measurements. The v_k term is the measurement noise and describes how there is some uncertainty in the measurement. The update step works by calculating the difference between the measurement expected for the predicted state and the actual measurement received. Then, based on how certain either the prediction or measurement is, the filter chooses a point in the middle which is the new state estimate.

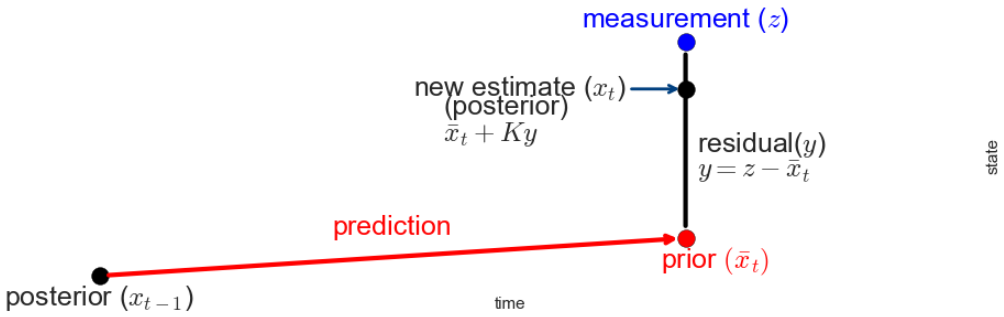

Figure 8:Kalman filter predict, update visualisation [9]

In figure 8, the posterior on the left shows the state in the previous time step. Then, the predict step is called which predicts the position in the next state (the prior) based on the state transition matrix and input control matrix. Having made the prediction, a measurement is received (z). The difference between the expected measurement at the prior and the actual measurement is calculated, called the residual. Finally, the new state prediction is made somewhere along the residual line, the distance along being determined by the Kalman gain. The Kalman gain denotes the ratio of uncertainty between the prediction and the measurement to weight one more highly [9]. This is the key attribute of the Kalman filter: combining multiple uncertain estimates to increase confidence in the overall estimate.

By examining both the predict and update equations, it can be seen that the Kalman filter only requires knowledge of one state in the past making the algorithm quick and memory efficient. However, it can also be seen that the state prediction matrices (F and B) and measurement state matrix (C) rely on linearly mapping inputs. In the real world, these relationships are often non-linear and thus an extension of the Kalman filter must be used.

2.4.2 Extended Kalman Filter

The Extended Kalman filter deals with non-linear models or non-linear measurements, or both. It achieves this by creating a linearized version of the model or measurement function for each time step at the current estimate and then performs the normal Kalman filter on this.

The equations for the predict and update steps can be seen in 20 and 21, respectively.

$$x_{k+1} = f(x_k, u_k) + w_k \tag{20}$$

$$y_k = h(x_k) + v_k \tag{21}$$

Notice how instead of multiplying the previous state by some matrix (as in the previous Kalman filter section), now to predict the next state, the previous state and control vector is transformed by some non-linear function, *f*. Process noise is added in the same way as before. In the update step, to determine the expected measurement for the given state, the state is transformed by the non-linear function *h*. The measurement noise, v_k, is added the same way as before [24].

The simple linear 'KalmanFilter' update function in FilterPy only requires the measurement and the function that calculates the expected measurement from the state. However, the 'ExtendedKalmanFilter' update function requires a further parameter which calculates the Jacobian matrix given a state. The Jacobian matrix is a matrix of partial derivatives of the functions *f* or *h* with respect to each variable in the state array. Essentially what this is doing is turning $h(x_k)$ into Hx_k at that specific point [9]. For example, a straight-line distance measurement function and plot (figure 9) can be seen below.

$$z = \sqrt{x^2 + y^2} \tag{22}$$

where

$$x = [0,5], \qquad y = 5$$

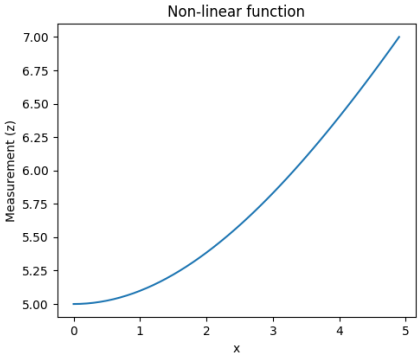

Figure 9: Non-linear function plot

Taking the partial derivatives of the measurement function, z, gives the Jacobian array. Then, as an example, to linearize about the point (2,5) gives the following array.

$$\begin{bmatrix} \frac{\partial z}{\partial x} & \frac{\partial z}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} \end{bmatrix} = [0.371 \quad 0.928] \tag{23}$$

This matrix now can be used as the *H* matrix (as seen in the linear example in section 2.3.1) and can be multiplied by a state estimate to give a good estimate of the measurement at that state. This Jacobian matrix is recalculated at every time step so it stays accurate about the current estimate. Plotting the linearized representation (in orange) for all the x and y combinations shows that this is a good estimate about (2,5) as shown in figure 10.

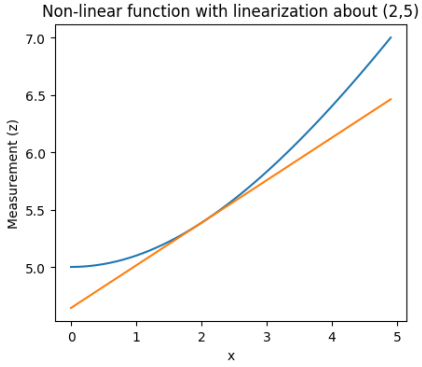

Figure 10: Non-linear function with linearization at (2,5)

3 Video Tracking

The video tracking component of SLAM is required to obtain useful information from frames of a digital video for use by the SLAM algorithm. While some sensor measurements used in SLAM such as radars, may be used without any processing, the richness and noisy nature of video require some condensing. In a SLAM problem, the key area of interest is the movement of the camera. Hence, examining how static objects move about in the video frame can give some indication of how the camera is moving.

3.1 Methodology

3.1.1 Scale-invariant feature transform (SIFT)

As a video is just a sequence of images, the logical first step in testing the SIFT algorithm was to test the matching capabilities between two images of the same subject taken at slightly different viewpoints.

The process for SIFT involves creating an array of keypoint locations and descriptors for the keypoints in each image and then using a matching algorithm to pair similar descriptors and match features between the two. The Python implementation can be seen in listing 1.

```
1.  sift = cv2.xfeatures2d.SIFT_create()
2.  kp_1, desc_1 = sift.detectAndCompute(img1,None)
3.  kp_2, desc_2 = sift.detectAndCompute(img2,None)
4.
5.  # Brute force matcher
6.  bf = cv2.BFMatcher()
7.  matches = bf.knnMatch(desc_1,desc_2, k=2)
8.
9.  # Apply ratio test to discard outliers and ensure proper point matching
10. good_matches = []
11. for m,n in matches:
12.     if m.distance < 0.60*n.distance:
13.         good_matches.append([m])
```

Listing 1: SIFT matching algorithm

The matching algorithm used in this example is a brute force, *k*-nearest neighbours matcher that for each descriptor finds the closest *k* matches based on the distance between the descriptors. The ratio test, as discussed in the literature review, is a common technique in SIFT feature matching and is incorporated in lines 12-14 and ensures that the closer of the two matches (in terms of Euclidean distance) returned by the *k*-nn matcher is sufficiently different to the next closest match.

A resize function was also implemented to reduce the resolution of the image. This resulted in improved speed of the algorithm and a reduction in the number of points which resulted in generally higher quality points and more robust tracking with fewer errors. SIFT on full 12MP resolution photos took far too long to process (around 50 seconds) which is unreasonable for video sequences. An example use of the resize function which takes an image and percentage scale factor (in this case 10% of the original size) and returns the resized image can be seen in listing 2.

```
1.  # Takes image and scales down
2.  def resize(img, scale):
3.      scale_percent = scale
4.      width = int(img.shape[1] * scale_percent / 100)
5.      height = int(img.shape[0] * scale_percent / 100)
6.      dim = (width, height)
7.
8.      scaled_img = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
9.      return scaled_img
10.
11. # Import images and resize
12. img1 = resize(cv2.imread('image_1.jpg'), 10)
13. img2 = resize(cv2.imread('image_2.jpg'), 10)
```

Listing 2: Image rescaling function

Finally, the two images and the matches between the images are displayed and the effectiveness of the algorithm can be visually assessed.

For use in video tracking the algorithm must be adapted to function on video sequences as opposed to a single pair of images. To do this, strong points are first selected by performing SIFT on two random frames of the video and the *n* strongest points are selected based on the strongest pairs (by Euclidean distance) between those two frames. Having selected the *n* strongest points in these frames these are tracked throughout the entire video.

For each frame in the video, a list of descriptors is obtained and for each initial tracking point, the closest match (by Euclidean distance) is found. The function to perform this matching can be seen in listing 3.

```
1. # Given a specific descriptor finds the best descriptor index from a list of de-
   scriptors
2. def findClosestMatch(desc, desc_list):
3.     min_dist = cv2.norm(desc, desc_list[0]); # Calculates baseline norm distance
4.     min_index = 0
5.     for i in range(len(desc_list)): # Tries to find a smaller norm distance
6.         if cv2.norm(desc, desc_list[i]) < min_dist:
7.             min_dist = cv2.norm(desc,desc_list[i]);
8.             min_index = i
9.     return min_index # Returns the index of closest match
```

Listing 3: Function to find the closest SIFT descriptor match

Then, for each frame, each descriptor can be found, and the coordinates saved to an array and/or marked onto the frame and the output video with tracked marker points displayed.

3.1.2 Speeded-Up Robust Features (SURF)

The SURF algorithm also detects and matches keypoints within images but is designed to do so faster than SIFT [25]. Implementing SURF in OpenCV is as simple as SIFT and requires the code changes seen in listing 4.

```
1. surf = cv2.xfeatures2d.SURF_create()
2. kp_1, desc_1 = surf.detectAndCompute(img1,None)
3. kp_2, desc_2 = surf.detectAndCompute(img2,None)
```

Listing 4: Implementing the SURF algorithm

For the video implementation, the same lines of code can be changed in the previous video tracking algorithm.

3.1.3 Lucas-Kanade Method for Optical Flow

The Lucas-Kanade Method for optical flow, an algorithm designed specifically for tracking movement of objects between consecutive frames, can also be implemented in Python through the OpenCV library. Initially, the coordinates of high-quality corner points (using the Shi-Tomasi corner point method) to track are determined by the 'goodFeaturesToTrack' function in OpenCV. The Lucas-Kanade optical flow algorithm is called by the lines of code in listing 5, where on line 3, the first and second parameters are the previous and current frames of video and the third parameter are the locations of the keypoints at that previous frame.

```
1. keypoints = cv2.goodFeaturesToTrack(frame_1, mask = None, **feature_params)
2.
3. nxt, status, error = cv2.calcOpticalFlowPyrLK(frame_1, frame_2, keypoints, None, **lk_pa
   rams)
```

Listing 5: Implementing the optical flow algorithm

This procedure is performed iteratively over all the frames of the videos and the new keypoint locations being updated from the 'nxt' variable returned by the optical flow algorithm.

However, to allow the algorithm to continue tracking when all the initially determined points go out of frame, an update is performed at every 20 frames which finds a new set of keypoints and adds them to the previous list of keypoints. However, to ensure points that were already being tracked retain continuity, a function is designed to combine the list of new and old tracking points based on proximal 'newness'. The threshold variable controls how distant a new point must be to be considered new. The 'combineTrackers' function can be seen in listing 6.

```
1.  def combineTrackers(oldPoints, newPoints, status): # Create new feature list based on
    old and new points
2.      threshold = 5
3.      extraFeatures = []
4.      for i in range(len(newPoints)):
5.          new = True # Assume point is new
6.          for j in range(len(oldPoints)): # Check proximty to all old points
7.              if status[j] == 1: # If point is still in frame
8.                  if math.hypot(newPoints[i][0]-oldPoints[j][0], newPoints[i][1]-
                                                          oldPoints[j][1]) < threshold:
9.                      new = False # If too close, then not new
10.         if new == True:
11.             extraFeatures += [newPoints[i].tolist()] # Otherwise add to key point list

12.     if len(extraFeatures) != 0:
13.         featureList = np.concatenate((oldPoints, np.asarray(extraFeatures)))
14.     else:
15.         featureList = oldPoints
16.     return featureList
```

Listing 6: Function to combine new and old unique tracking points

The result of this optical flow algorithm is an ever-growing list of past and present tracking points. The current location of each tracking point and the status of the tracking points (whether they are in-frame or out of frame) can be accessed and stored throughout the video sequence. To visualise the effectiveness of the tracking, the video sequence can be viewed with a dot at the pixel coordinates of the tracking points at each frame overlaid on the image. Each element of the feature array is assigned a unique colour to distinguish from other points and verify that the same point is sticking to the same area. Overall, the information obtained from this algorithm can be used to determine the motion of the camera throughout an extended move.

3.2 Results and Discussion

3.2.1 Scale-invariant feature transform (SIFT)

In the example where keypoints are to be matched between two images taken of the same subject at slightly different angles, the SIFT algorithm can be run. The original two images can be seen in figure 11 below.

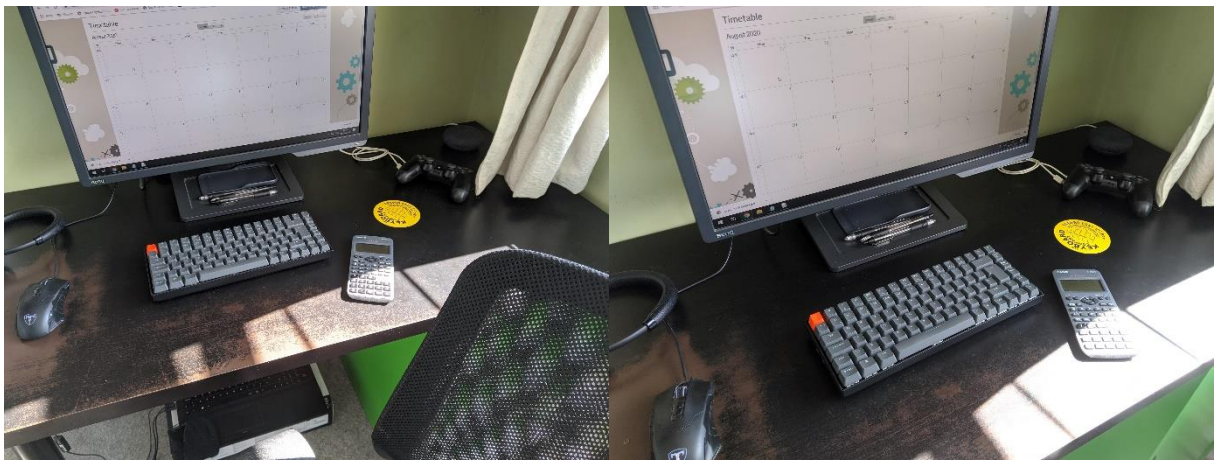

Figure 11: Two images of one scene at different angles

These images contain lots of high contrast areas and distinguishable points so is ideal for the SIFT algorithm which chooses high contrast areas as the keypoints. As discussed in the method, the images will be converted to greyscale and resized. The reason the images are converted to greyscale is to increase the speed of the algorithm by reducing the amount of data and also make the algorithm impartial to changes in colour between images. Furthermore, the resizing is done to speed up the algorithm as well as reduce specs of noise that may look high contrast but not appear in the same place in a different image. Another effect of reducing the resolution is also fewer keypoints, but from testing, these fewer points were usually of higher quality. An example of the SIFT algorithm on this pair of images at a scale of 15% (604px by 453px) can be seen in figure 12.

Figure 12: SIFT matching at 15% scale

A quick way to assess the effectiveness of this SIFT performance is to check the lines run roughly parallel – this works because the same keypoints shift position by roughly the same amount in each image. This SIFT algorithm performs fairly well – all the points on the monitor are correct matches, as are the point on the keyboard and the pens on the monitor stand. However, there are a few erroneous points, mainly in the sunlight/shadow area. Part of the shadow in the left-hand image is matched to the curtain in the right-hand image and another point in the shadow is incorrectly matched. However, these points are understandably mismatched: the points do look similar to the eye. Regarding processing time, to create SIFT keypoints and descriptors for both images and to match the keypoints took 0.231 seconds. Of the 40 matches here, there are only 3 mismatched points giving an accuracy of 92.5% for this image.

Reducing the resolution of the image further to 5% original scale (201px by 151px) produced an image seen in figure 13.

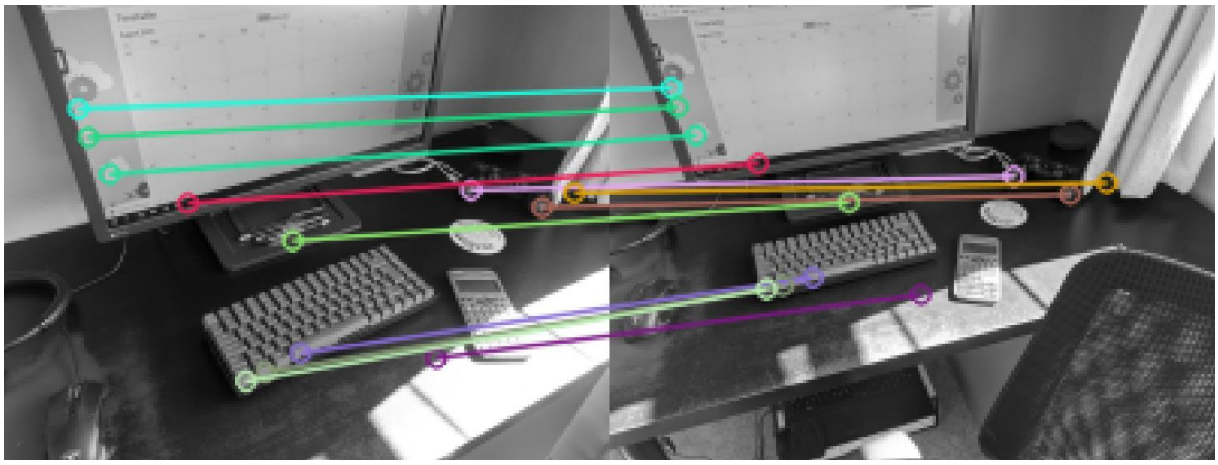

Figure 13: SIFT matching at 5% scale

While only 12 keypoints were matched here, they are generally of higher quality and there are no mismatched points in this image. Regarding processing time, the algorithm took 0.032 seconds to locate and match these keypoints between the images. For a real-time application with a video stream at 25 frames-per-second, for just the point tracking part, the algorithm must process each frame in 0.04 seconds.

For the video tracking adaptation of the algorithm, a simple video of a calculator on the desk will be used where the camera slowly moves left and right. The first and last frames are shown below in figure 14. These frames will be the reference frames when picking out the strongest *n* tracking points.

Figure 14: First and last frame of short video sequence

The strongest 15 matches will be picked from a brute-force descriptor match on these two frames and then these 15 points will be tracked throughout the entire video. The video will be output with each of the 15 points uniquely coloured and positioned on the frame where the best match occurs. For this specific experiment, the video will be converted to greyscale and resized to 10% (288px by 162px). Three frames from the video (the first frame, a middle frame, and the last frame) can be seen in figure 15.

Figure 15: Three frames of SIFT tracked video at 10% scale

While the mass of points appears to stick to the general area, the individual points shift around a large amount and often lose their true position while returning to it from time to time. This is clear in the image above: if you pick a single point in the first image and try to find it in the next two, in most cases it does not track perfectly. The full track of 192 frames took 47.84 seconds. As a visualisation, the position of a single tracking point in the video was plotted to view its path and the result is very jittery and can be seen in figure 16.

While it could be argued this is a difficult case because all the calculator buttons look very similar, repeating the algorithm on different videos produces the same problems. Also, for a real-world application, the video tracking algorithm will have to deal with repeated patterns and similar areas in the image such as leaves or walls.

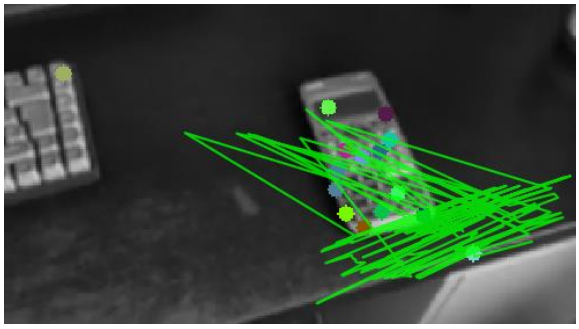

Figure 16: Path of one tracking point with SIFT (green)

3.2.2 Speeded-Up Robust Features (SURF)

The SURF algorithm produced extremely similar results to SIFT for the same images. At 15% scale, the SURF algorithm matches 51 points with slightly increased accuracy compared to SIFT from visual inspection. Regarding processing time, to create SURF keypoints and descriptors for both images and to match the keypoints took 0.370 seconds. This is somewhat slower than the SIFT equivalent. Despite finding 11 more tracking points the time per tracking point is still lower.

Figure 17: SURF matching at 15% scale

At 5% scale, the SURF algorithm finds and matches 15 tracking points in 0.023 seconds. This is marginally faster than the SIFT equivalent and produces more points. However, one point incorrectly matches an area of the calculator to the keyboard (see figure 18).

In the case of the video implementation, the SURF algorithm behaves similarly to the SIFT algorithm albeit considerably faster. The mass of points generally sticks to the intended area of interest but are very jittery and often lose their correct position before returning from time to time. The full track of 192 frames took 23.97 seconds.

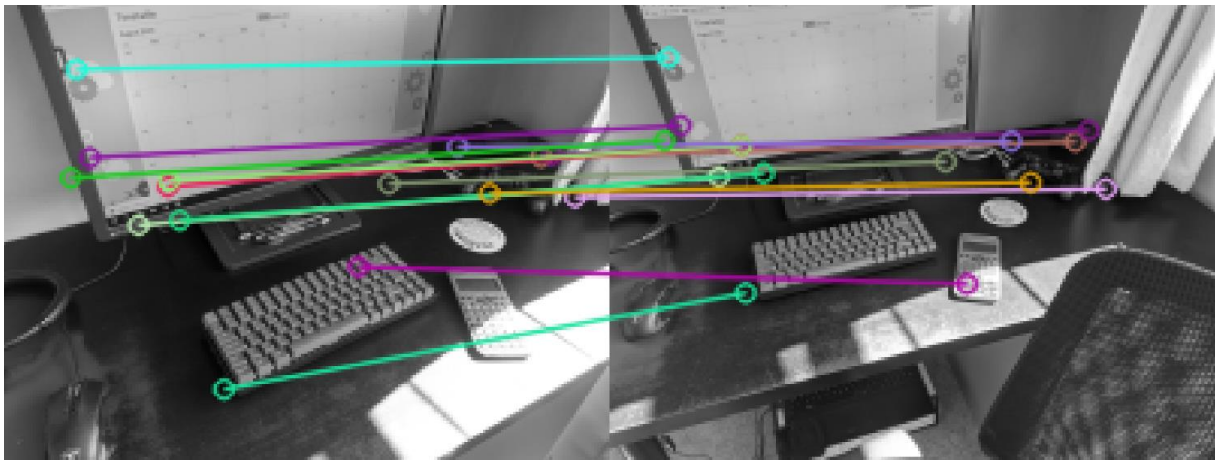

Figure 18: SURF matching at 5% scale

In total, for images, neither algorithm seems to have a clear advantage over the other. Both algorithms seem to have similar processing times and point accuracy. However, in terms of video tracking, the SURF algorithm is preferred due to the considerable advantage of processing time.

3.2.3 Lucas-Kanade Method for Optical Flow

The Lucas-Kanade Optical Flow algorithm is specifically designed for video feeds. It keeps track of movement between frames rather than re-searching the entire frame for the same point so, in theory, should improve on the shortcomings found in the SIFT and SURF approaches. Furthermore, the requirement to track motion through smaller areas may speed up the processing time of the algorithm – making it more suited to real-time applications.

Applying the Lucas-Kanade Optical Flow algorithm to the same video sequence shown prior (the calculator on the desk), the processing speed performance improvement is instantly noticeable. Using the video rescaled to 10% (288px by 162px) the optical flow algorithm took just 0.71 seconds for the entire 192 frames. This is over 67 times faster than SIFT and 33 times faster than SURF. Even at full resolution, the algorithm completed the entire track in just 8.85 seconds.

Not only is the processing speed required to track the video greatly improved, so is the quality of the tracking itself. The tracking points stay far more rigidly tracked to the correct keypoint.

Three frames from the video (the first frame, a middle frame, and the last frame) can be seen in figure 19 below.

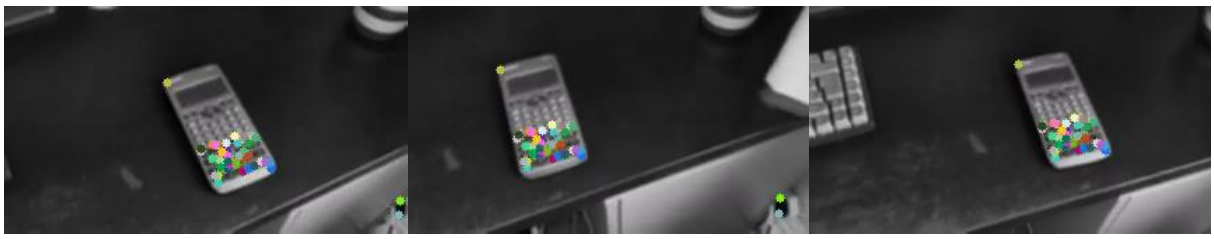

Figure 19: Three frames of optical flow tracked video at 10% scale

The optical flow algorithm tracks 47 points nearly flawlessly in this sequence. While increasing the resolution to full resolution (1920px by 1080px) increases the number of tracking points to 200, the added increase in processing time is not worth the little extra benefit. Tweaks can be made depending on the required application to the resolution of the frames as well as the parameters of the Shi-Tomashi corner detection and Lucas-Kanade algorithms to balance speed and quality.

However, one issue of this algorithm is the inability to create new tracking points if new areas of interest come into the frame or areas go out. This is apparent in the above figure where no new points are added to the keyboard that enters the left of the frame. To fix this, a function that finds new keypoints is run every 20 frames and the list of new points combined with the old points to update the list. The effect of this added function can be seen in the three frames in figure 20.

Figure 20: Three frames of optical flow tracked video at 10% scale with dynamic points

In the top-right corner of the second frame, a tracking point is added to the mug and in the final frame, many extra points are added to the keyboard which is a source of high-quality tracking points.

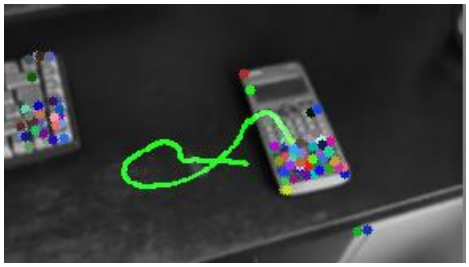

Figure 21: Path of one tracking point with optical flow (green)

To visualise the performance a little better, the movement of a single tracking point is drawn as a line. This can be seen in figure 21. Compared to the line track in the SIFT algorithm, this line is extremely smooth with no jitters and matches the movement of that

specific point on the calculator exactly. Also, this shows how the movement of a single point can be tracked throughout the whole video despite new points being added throughout the sequence. The same algorithm has been tested on other more complex video sequences and works equally well, as seen in figure 22.

Figure 22: Three frames of optical flow track with path line for one point (green)

In extended video sequences such as 'advio-10' in the ADVIO dataset [3] the optical flow algorithm is able to track points throughout the entire sequence. This involves tracking points of a target while it is in-frame and then tracking new points in an entirely new frame where none of the existing tracking points appear. Figure 23 shows three frames of the ADVIO 'advio-10' dataset, scaled at 75% with tracking points throughout. The green line shows the track of one point before it goes out of frame and then frames after with a new set of points.

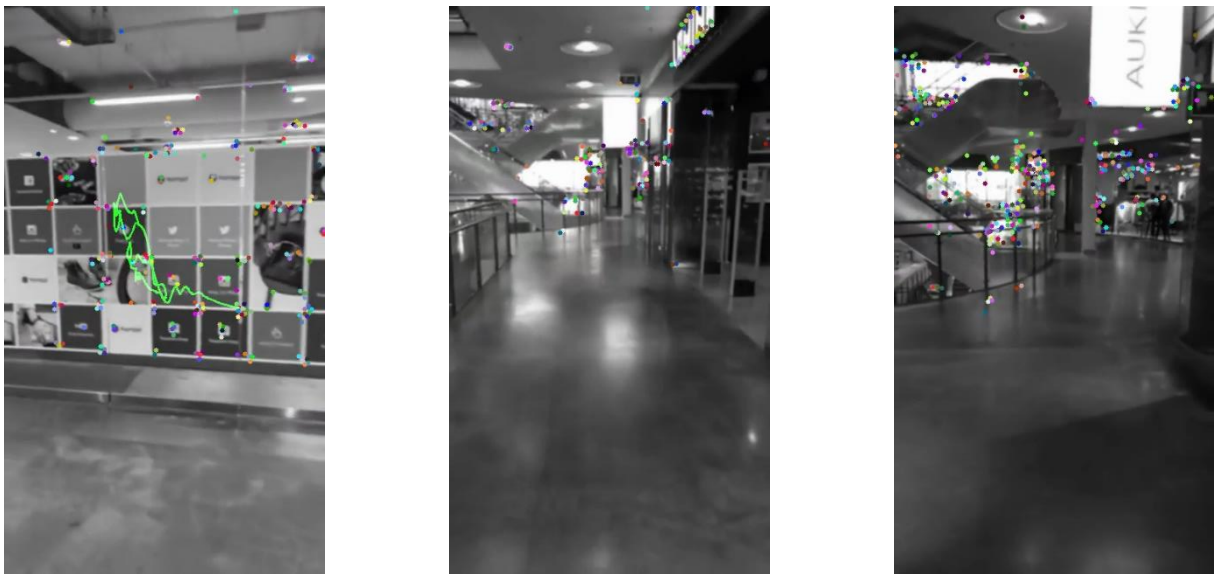

Figure 23: Optical flow track with dynamic points and path line on ADVIO-10 sequence [3]

The Python code for the optical flow tracker can be seen in Appendix 1.

4 Simulation and the Extended Kalman Filter (EKF)

4.1 Methodology

The Extended Kalman Filter (EKF) is the key component of a SLAM system and is tasked with taking uncertain measurements and observations from the camera and turning them into good estimates of the position of the camera. However, before diving straight into real-world datasets,

it makes sense to build up the algorithms on synthetic, computer-generated simulations where the complexity and uncertainty can be controlled. This allows repeatability and variables to be changed gradually without having to worry about external factors coming into play.

Synthetic data will be created in Python using simple iterative algorithms based on mathematical modelling of movement, and uncertainty/noise will be added by the 'NumPy' library to sample random numbers from the normal distribution with controlled mean and variance. Simulations and results will be visualised by using the 'Matplotlib' library to create graphs and animations. The 'FilterPy' Python library will be used for implementing the Extended Kalman filter.

4.1.1 Extended Kalman Filter (EKF) setup and operation

To set up the EKF in Python, first, the filter object must be initialised where *dim_x* and *dim_z* denote the number of state variables and the number of measurement variables, respectively.

```
1. from filterpy.kalman import ExtendedKalmanFilter
2.
3. ekf = ExtendedKalmanFilter(dim_x = dim_x, dim_z = dim_z)
```

Listing 7: Initialising the Kalman filter

Having initialised the filter, a state vector of size *dim_x, ekf.x* will be created which stores the current estimate of the state variables. This can be initialised by setting the values to nearby the true starting position.

$$\text{ekf.x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

Next, the state transition matrix, *ekf.F* can be set, which is an array of size *dim_x* x *dim_x* and relates the next state (x') to the previous state (x) based on the movement model.

$$\text{ekf.F} = \begin{bmatrix} a_{00} & a_{01} & \cdots \\ \vdots & \ddots & \\ a_{n0} & & a_{nn} \end{bmatrix}$$

$$\begin{bmatrix} x_1' \\ \vdots \\ x_n' \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & \cdots \\ \vdots & \ddots & \\ a_{n0} & & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \tag{24}$$

In terms of uncertainty, three arrays must be set: *ekf.R*, *ekf.Q* and *ekf.P*. The *ekf.R* matrix is of size *dim_z* x *dim_z* and describes the measurement noise. For uncorrelated measurements, the diagonals need to be set to the variance of each respective measurement and the off-diagonals set to zero. The *ekf.Q* array is of size *dim_x* x *dim_x* and describes the process noise in each

state. While many movement models have some correlation between the two states, the off-diagonals are ignored in the examples in favour of simplicity. The diagonals corresponding to each state must be set appropriately to the amount of process noise present in each state. The *ekf.P* array is of size *dim_x* x *dim_x* and describes the estimated accuracy of the starting state. Again, only the diagonal is concerned in this example and is set quite high to give the model room to adjust the state estimate.

$$\text{ekf.R} = \begin{bmatrix} \sigma_1^2 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_{\dim_z}^2 \end{bmatrix}, \text{ekf.Q} = \begin{bmatrix} \sigma_1^2 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_{\dim_x}^2 \end{bmatrix}, \text{ekf.P} = \begin{bmatrix} \sigma_1^2 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_{\dim_x}^2 \end{bmatrix}$$

The final two elements required for the EKF are the function that computes the Jacobian matrix at a given state and the function that computes the expected measurement vector for a given state. Both functions take the state vector as an input. The Jacobian function returns a *dim_z* x *dim_x* array of the partial derivative of each measurement function with respect to each state variable. The *h(x)* function returns a vector of size *dim_z*.

$$\text{Jacobian}(\bar{x}) = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots \\ \vdots & \ddots & \\ \frac{\partial h_{\dim_z}}{\partial x_1} & & \frac{\partial h_{\dim_z}}{\partial x_{\dim_x}} \end{bmatrix}, \text{H}(\bar{x}) = \begin{bmatrix} h_1(x) \\ \vdots \\ h_{\dim_x}(x) \end{bmatrix} \tag{25}$$

Now, after all this has been set-up for the model, the EKF can be run. At each time step, the simulation is called to update the true state variable. Then, the *ekf.update(…)* function can be called which takes the measurement at this time step (*z*), the Jacobian function (*XJacobian*) and the expected measurement function (*hx*) as parameters and in turn changes the state estimate, *ekf.x* to the best guess based on the measurements. Finally, the *ekf.predict()* function is called which predicts the next state from the previous state, based purely on the state transition matrix (*ekf.F*) described prior. This is repeated for as many time steps as is required. Further arrays can be used to keep records of the state estimates through time to graph and assess.

```
1.  # Change true position of model and return measurement
2.  z = update_model(dt)
3.
4.  # Update EKF with measurement, Jacobian function and function to find expected measure
    ment
5.  ekf.update(z, XJacobian, hx) # Update predictions with measurements
6.
7.  # Predict next state
8.  ekf.predict()
```

Listing 8: EKF predict, update cycle

4.1.2 Using the EKF to track a single target using radar

To get comfortable with using the Extended Kalman Filter a simple scenario is set up. This scenario is the same as seen in the Extended Kalman Filter section of Roger Labbe's book [9] but will be extended on further, later. A plane is flying overhead with a constant altitude and velocity. A radar at ground level receives distance measurements from the plane by firing directly at the plane. The plane can be described in three states: the x-position, the x-velocity, and the y-position (altitude). A diagram of the scenario can be seen in figure 24.

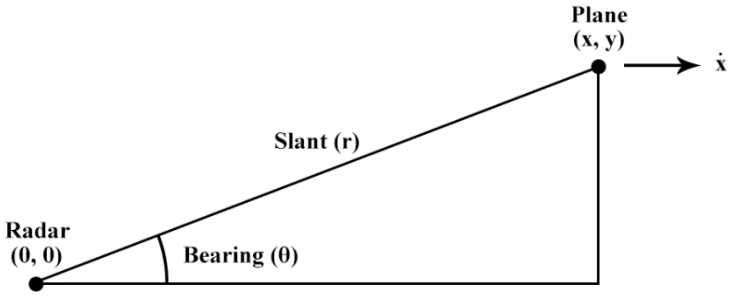

Figure 24: Simulation setup for EKF with single target radar

The update of the actual position of the plane is done through the function '*update_radar*' which takes the time period as an input which then updates the true position variables and returns the measurement for this step. The measurement that is taken is the straight-line distance between the radar and the plane. The straight-line distance can be calculated using the Pythagorean equation and the non-linear nature of the square root is why the extended Kalman filter is required. The Python implementation of the initialisation and update function can be seen in listing 9.

```
1.  from numpy.random import randn
2.  import math
3.
4.  # Initialise position
5.  x_pos = 0
6.  x_vel = 20
7.  y_pos = 100
8.
9.  def update_radar(dt):
10.     global x_pos, x_vel, y_pos
11.     # Add process noise
12.     x_vel = x_vel  + .1 * randn()
13.     y_pos = y_pos + .1 * randn()
14.     x_pos = x_pos + x_vel * dt
15.
16.     # Measurement noise
17.     err = x_pos * 0.05 * randn() # Create error
18.     distance = math.sqrt(y_pos**2 + x_pos**2) # Calculate measurement
19.
20.     return distance + err
```

Listing 9: Simulation model initialisation and updating

In this example the measurement noise is added proportionally to the distance the plane is from the radar – the further away, the more uncertainty there is. The process noise in the x-velocity and y-position is added based on scaled random samples from the normal distribution.

The true position of the plane and the initialisation of the state estimate can be seen below.

$$\bar{x} = \begin{bmatrix} \text{x position} \\ \text{x velocity} \\ \text{y position} \end{bmatrix} = \begin{bmatrix} 0 \\ 20 \\ 100 \end{bmatrix},\ \text{ekf.x} = \begin{bmatrix} \text{-10} \\ 22 \\ 90 \end{bmatrix}$$

From one time instance to the next, the state of the plane under a constant-velocity model can be described using the equations of motion seen in 26-28.

$$x' = x + \Delta t \dot{x} \tag{26}$$

$$\dot{x'} = \dot{x} \tag{27}$$

$$y' = y \tag{28}$$

This gives the following state transition matrix where the x-velocity and y-position remain constant throughout and the x-position changes based on the velocity and time period.

$$\text{ekf.F} = \begin{bmatrix} 1 & \Delta t & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The three uncertainty matrices, *ekf.R*, *ekf.Q* and *ekf.P* can now be initialised based on the measurement and process noise added prior. This simulation will run for 40 seconds and therefore the maximum distance in the x-direction will be 800m. Hence the variance in the measurement will be a maximum of 800*0.05 = 40m. In *ekf.Q*, the process noise, a good place to start is with the random noise scalar values, although some tweaking is recommended to improve the filter. Decreasing the uncertainty value means the filter is less sensitive to noise and gives a steadier prediction whereas higher uncertainty means the filtered result follows the true position more closely and is more prone to change. Finally, *ekf.P* can be set large to give the filter room for initial adjustment. The final values for these matrices can be seen below.

$$\text{ekf.R} = [40],\ \text{ekf.Q} = \begin{bmatrix} 10^{-7} & 0 & 0 \\ 0 & 10^{-3} & 0 \\ 0 & 0 & 10^{-3} \end{bmatrix},\ \text{ekf.P} = \begin{bmatrix} 150 & 0 & 0 \\ 0 & 150 & 0 \\ 0 & 0 & 150 \end{bmatrix}$$

Finally, the Jacobian matrix function and expected measurement function can be written. The partial derivatives can be either calculated by hand or by the '*SymPy*' Python library for more complex measurement equations. The equations for this matrix are shown in 29. The functions can also be seen in listing 10.

$$\mathrm{J}(\bar{x}) = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & 0 & \frac{y}{\sqrt{x^2+y^2}} \end{bmatrix} \tag{29}$$

```
def XJacobian(x):
    # At state x return Jacobian matrix
    x_pos = x[0]
    y_pos   = x[2]
    return array ([[x_pos/math.sqrt(x_pos**2 + y_pos**2),
                    0., y_pos/math.sqrt(x_pos**2 + y_pos**2)]])

def hx(x):
    x_pos = x[0]
    y_pos   = x[2]
    # Measurement expected for state x
    return (x_pos**2 + y_pos**2) ** 0.5
```

Listing 10: Jacobian and expected measurement functions

Finally, the plane position can be iteratively updated and the EKF called after each update as seen in listing 11.

```
dt = 0.05 # Time step

for i in range(int(20/dt)): # Number of update steps needed
    distance = update_radar(0.05) # Update plane position and take measurement

    # Update step with measurement, Jacobian and expected measurement functions
    rk.update(array([distance]), XJacobian, hx)

    rk.predict() # Predict next state based on state transition
```

Listing 11: Running EKF iteration

The results of this simulation can be seen in section 4.2.1.

Having implemented this simple example, the problem can be modified to operate on measurements of the angle to the plane instead of slant distance. The motion of the plane and process noise at the model update step is unchanged. The *update_radar* function will return an angle calculated by equation 30.

$$\theta = \tan^{-1}\left(\frac{y}{x}\right) + \text{error} \tag{30}$$

However, the measurement noise is now changed to a random sample from the normal distribution scaled by a constant 0.1 – irrespective of the distance to radar. As the measurement noise error has changed, the *ekf.R* matrix must be updated to reflect this change.

$$\text{ekf.R} = [0.1]$$

The expected measurement function, *hx*, is now the expected angle from the state passed to it and can be seen in equation 31.

$$\mathrm{H}(\bar{x}) = \left[\tan^{-1}\left(\frac{y}{x}\right)\right] \tag{31}$$

The new Jacobian matrix can be calculated as shown in equation 32.

$$\mathrm{J}(\bar{x}) = \left[\frac{-y}{x^2 + y^2} \quad 0 \quad \frac{x}{x^2 + y^2}\right] \tag{32}$$

Finally, the EKF can be run in the same way as before but will now update based on angle measurements instead. The results from these experiments can be seen in section 4.2.1.

Tracking the plane position with angle as the only measurement is rather tricky as shown in the analysis of this previous experiment. Hence, to improve the accuracy of the filter and extend this basic example of the EKF further, a second measurement for radial velocity (how fast the plane is moving away from the radar directly) will be added, as seen in figure 25. This measurement is possible for radar using the principles of Doppler shift and measuring the phase shift of the returning pulse. This measurement can be determined by equations 33 and 34.

$$\dot{x} = \text{radial velocity} \times \cos(\theta) \tag{33}$$

$$\text{radial velocity} = \frac{\dot{x}}{\cos(\theta)} \tag{34}$$

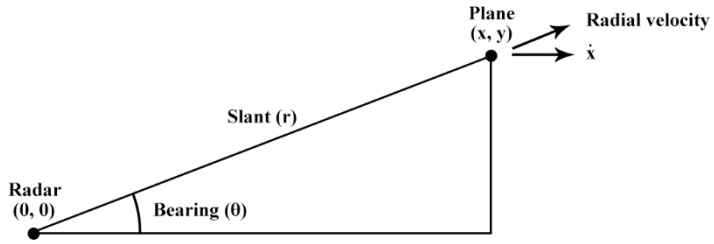

Figure 25: Simulation setup for EKF with single target radar with radial velocity

To prepare the EKF for this update a few changes must be made. Firstly, *dim_z* is now 2 as there are two measurements: angle and radial velocity.

The *update_radar* function now returns a 2 x 1 scalar of both the angle and radial velocity measurements at that time step. The added measurement noise for the radial velocity is a random sample from the normal distribution scaled by a constant 2 (irrespective of distance) and the measurement noise for the angle remains the same. The *ekf.R* matrix must be changed to reflect the uncertainty in both measurements.

$$\text{ekf.R} = \begin{bmatrix} 0.1 & 0 \\ 0 & 2 \end{bmatrix}$$

The expected measurement function, *hx*, returns a 2 x 1 scalar, the expected angle and radial velocity from the state passed to it and can be seen in equation 35.

$$\mathrm{H}(\bar{x}) = \begin{bmatrix} \tan^{-1}\left(\frac{y}{x}\right) \\ \frac{\dot{x}}{\cos\left(\tan^{-1}\left(\frac{y}{x}\right)\right)} \end{bmatrix} \tag{35}$$

With the additional measurement, the Jacobian function gains another row of partial derivatives relating to the second measurement as shown in equation 36.

$$\mathrm{J}(\bar{x}) = \begin{bmatrix} \frac{\partial h_1}{\partial x} & \frac{\partial h_1}{\partial \dot{x}} & \frac{\partial h_1}{\partial y} \\ \frac{\partial h_2}{\partial x} & \frac{\partial h_2}{\partial \dot{x}} & \frac{\partial h_2}{\partial y} \end{bmatrix} = \begin{bmatrix} -\frac{y}{x^2+y^2} & 0 & \frac{x}{x^2+y^2} \\ -\frac{\dot{x}y^2}{x^2\sqrt{x^2+y^2}} & \frac{\sqrt{x^2+y^2}}{x} & \frac{\dot{x}y}{x\sqrt{x^2+y^2}} \end{bmatrix} \tag{36}$$

Again, the EKF can be run in the same way as before but will now update based on both the angle and the radial velocity. The results from these experiments can be seen in section 4.2.1.

4.1.3 Using the EKF to track a camera moving through a field of targets

Having set up and experimented with a simple extended Kalman filter in the previous section, these principles can now be applied to a SLAM problem. In these simulations, a camera is moving with a fixed-velocity and gets measurements from numerous fixed targets as its position relative to them changes. A top-down view of this scenario for two targets can be seen in figure 26.

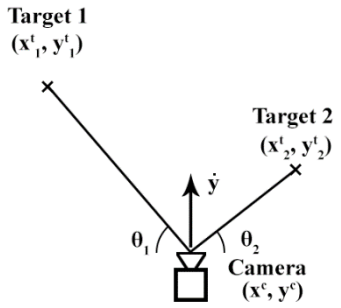

Figure 26: Simulation setup for camera moving past targets

For the first experiment, measurements of the Pythagorean distance between the camera and the two targets will be taken. As the distance must be figured out relative to the camera, the positions of the targets will be fixed and known – the functions that require calculation of the distance will be hard-coded for these specific targets.

$$\text{target\_1} = \begin{bmatrix} -5 \\ 10 \end{bmatrix},\ \text{target\_2} = \begin{bmatrix} 7 \\ 15 \end{bmatrix}$$

The function that updates the position of the camera for each time step does so by the following function, *update_camera*, shown in listing 12.

```
1.  def update_camera(dt):
2.      global x_pos, y_pos, y_vel, target_1, target_2
3.      # Process noise
4.      x_pos = x_pos  + .01 * randn()
5.      y_vel = y_vel + .01 * randn()
6.      y_pos = y_pos + y_vel * dt
7.
8.      # Measurement noise
9.      err_1 = abs(y_pos - target_1[0]) * .05 * randn()
10.     err_2 = abs(y_pos - target_2[0]) * .05 * randn()
11.
12.     range_1 = math.sqrt((x_pos - target_1[0])**2 +
13.                         (y_pos - target_1[1])**2)
14.
15.     range_2 = math.sqrt((x_pos - target_2[0])**2 +
16.                         (y_pos - target_2[1])**2)
17.
18.     return range_1 + err_1, range_2 + err_2
```

Listing 12: Camera update model

In this example, the process noise for both the x-position and y-velocity is a random number drawn from the normal distribution scaled by a factor of 0.01. The measurement noise for each target measurement is dependent on the distance of the camera from that target – the closer the target the more accurate the measurement. However, the randomness of the measurement noise comes again from a sample of the random distribution, this time scaled by 0.05. The maximum distance of the camera to a target never exceeds 40m in this experiment so the uncertainty variance of the measurement can be set as 2m. The uncertainty arrays *ekf.R*, *ekf.Q* and *ekf.P* can now be initialised.

$$\text{ekf.R} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix},\ \text{ekf.Q} = \begin{bmatrix} 10^{-4} & 0 & 0 \\ 0 & 10^{-7} & 0 \\ 0 & 0 & 10^{-4} \end{bmatrix},\ \text{ekf.P} = \begin{bmatrix} 150 & 0 & 0 \\ 0 & 150 & 0 \\ 0 & 0 & 150 \end{bmatrix}$$

The camera state is initialised, and the initial state estimate is set.

$$\bar{x} = \begin{bmatrix} \text{x position} \\ \text{y position} \\ \text{y velocity} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix},\ \text{ekf.x} = \begin{bmatrix} 5 \\ 7 \\ 2.5 \end{bmatrix}$$

The camera follows a constant-velocity model in the y-direction and therefore the equations of motion are as shown in 37-39.

$$x' = x \tag{37}$$

$$y' = y + \dot{y}\Delta t \tag{38}$$

$$\dot{y}' = \dot{y} \tag{39}$$

Hence, the state transition matrix can be derived.

$$\text{ekf.F} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

The expected measurement function, *hx*, returns a 2 x 1 scalar, corresponding to the distances from the two targets at state $\bar{x}$.

$$\mathrm{H}(\bar{x}) = \begin{bmatrix} \sqrt{(x - T_1^x)^2 + \left(y - T_1^y\right)^2} \\ \sqrt{(x - T_2^x)^2 + \left(y - T_2^y\right)^2} \end{bmatrix} = \begin{bmatrix} \sqrt{\left(x - (-5)\right)^2 + \left(y - (10)\right)^2} \\ \sqrt{\left(x - (7)\right)^2 + \left(y - (15)\right)^2} \end{bmatrix} \tag{40}$$

Therefore, the partial derivatives of these measurement equations with respect to each state variable make up the Jacobian matrix.

$$\mathrm{J}(\bar{x}) = \begin{bmatrix} \frac{\partial h_1}{\partial x} & \frac{\partial h_1}{\partial y} & \frac{\partial h_1}{\partial \dot{y}} \\ \frac{\partial h_2}{\partial x} & \frac{\partial h_2}{\partial y} & \frac{\partial h_2}{\partial \dot{y}} \end{bmatrix} = \begin{bmatrix} \frac{x - T_1^x}{\sqrt{(x - T_1^x)^2 + (y - y_1^t)^2}} & \frac{y - T_1^y}{\sqrt{(x - T_1^x)^2 + \left(y - T_1^y\right)^2}} & 0 \\ \frac{x - T_2^x}{\sqrt{(x - T_2^x)^2 + \left(y - T_2^y\right)^2}} & \frac{y - T_2^y}{\sqrt{(x - x_2^t)^2 + \left(y - T_2^y\right)^2}} & 0 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{x - (-5)}{\sqrt{\left(x - (-5)\right)^2 + \left(y - (10)\right)^2}} & \frac{y - (10)}{\sqrt{\left(x - (-5)\right)^2 + \left(y - (10)\right)^2}} & 0 \\ \frac{x - (7)}{\sqrt{\left(x - (7)\right)^2 + \left(y - (15)\right)^2}} & \frac{y - (15)}{\sqrt{\left(x - (7)\right)^2 + \left(y - (15)\right)^2}} & 0 \end{bmatrix} \tag{41}$$

In the same fashion as the previous experiments, the EKF can be run by iteratively updating the camera position and calling the update and predict steps of the filter with the measurements returned from the *camera_update* function. The results of the moving camera simulation with two targets can be seen in section 4.2.2.

Adding an additional target will provide the model with more information in the update step so should theoretically improve performance. To add a third target, the coordinates of the target are simply added to a new array (*target_3*) and the *update_camera* function is modified to return another camera measurement corresponding to this third target. Additionally, a third-row

is added to both the expected measurement function, *hx*, and the Jacobian function, *XJacobian.* Finally, *ekf.R* (the measurement uncertainty array) becomes an array of shape 3 x 3 where the third diagonal corresponds to the uncertainty in the new measurement – in this case also 2m. The performance graphs of this can be compared directly to the previous experiment to understand the effect of additional information on the performance of the filter.

To extend this example one step closer to a full SLAM problem some adaptations will be made. Firstly, the targets will become part of the state vector so they can be created randomly and dynamically and the Jacobian and expected measurement arrays will automatically update without being hardcoded for specific target positions. To control the randomness and for repeatability, the *numpy.random.seed* variable is set to a fixed number at the beginning of the experiment – thus the random numbers will be the same every time. To create the targets, a random set of *n* coordinate pairs (within a range) are created and stored in an array by the code in listing 13.

```
1.  from numpy.random import uniform
2.
3.  nT = 10 # Number of targets
4.
5.  # Initialise target positions
6.  targets = []
7.
8.  # For each required target get x and y position from uniform distribution
9.  for i in range(nT):
10.     targets.append([uniform(-10,10), uniform(-10, 100)])
```

Listing 13: Dynamic target creation

Then, the *update_camera* function will calculate the distances between all the targets iteratively to create the returned measurement array for the set number of targets (listing 14).

```
1.  def update_camera(dt):
2.      global x_pos, y_pos, y_vel, targets
3.      # Process noise
4.      x_pos = x_pos  + .1 * randn()
5.      y_vel = y_vel + .01 * randn()
6.      y_pos = y_pos + y_vel * dt
7.
8.      measurements = []
9.
10.     for T in targets: # For every target, calculate measurement
11.         err = abs(y_pos - T[0]) * .05 * randn() # Error dependant on distance
12.         rng = math.sqrt((x_pos - T[0])**2 +
13.                         (y_pos - T[1])**2)
14.         measurements.append([rng + err])
15.
16.     measurements = asarray(measurements)
17.
18.     return measurements
```

Listing 14: Camera update function with dynamic targets

To calculate *dim_x*, the number of states and *dim_z*, the number of measurements, the following calculations are used where *nT* is the number of specified targets. The number of states is the number of states describing the camera (x-position, y-position, y-velocity) plus two states per target (x-position, y-position). The number of measurements is equivalent to the number of targets (one measurement per target).

$$\text{dim\_x} = 3 + 2 * nT, dim_z = nT \tag{42}$$

The initial state array is initialised with the states describing the camera close to the true values but the states describing the target positions exactly at the target positions.

$$\bar{x} = \begin{bmatrix} \text{camera x position} \\ \text{camera y position} \\ \text{camera y velocity} \\ \text{target 1 x-coordinate} \\ \text{target 1 y-coordinate} \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0.97 \\ 68.7 \\ \vdots \end{bmatrix}, \text{ ekf.x} = \begin{bmatrix} 5 \\ 7 \\ 2.5 \\ 0.97 \\ 68.7 \\ \vdots \end{bmatrix}$$

In this simulation set-up, the positions of the targets are bound in the range [-10, -10] for the x-position and [10, 100] for the y-position, so a reasonable maximum distance a target could be from the camera is approximately 100m. Thus, the variance of the measurements can be defined as 0.05*100 = 5m. Because all the measurements are taken in the same way, the measurement uncertainty array, *ekf.R*, is simply a diagonal matrix of shape nT x nT (where nT is the number of targets).

$$\text{ekf.R} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & \ddots \end{bmatrix}$$

In this experiment, for the state process uncertainty array, *ekf.Q*, the uncertainty in the positions of the targets can be set to 0 as the state estimate is initialised exactly at the known target positions. The uncertainty in the other states describing the camera can be initialised based on the amount of process noise added in the *update_camera* function.

$$\text{ekf.Q} = \begin{bmatrix} 10^{-3} & 0 & 0 & 0 & 0 \\ 0 & 10^{-7} & 0 & 0 & 0 \\ 0 & 0 & 10^{-4} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \ddots \end{bmatrix}, \text{ ekf.P} = \begin{bmatrix} 150 & 0 & 0 \\ 0 & 150 & 0 \\ 0 & 0 & \ddots \end{bmatrix}$$

Since the number of states in the state array can change based on the number of targets created, the expected measurement and Jacobian matrix functions, *hx* and *XJacobian*, must be calculated

iteratively based on the state array. For the expected measurement function (*hx*) this means looping through each target in the state array and calculating its distance from the camera. The final result will be a vector of length nT. The array and code for this section can be seen in equation 43 and listing 15, respectively.

$$\mathrm{H}(\bar{x}) = \begin{bmatrix} \sqrt{(x - x_1^t)^2 + (y - y_1^t)^2} \\ \sqrt{(x - x_2^t)^2 + (y - y_2^t)^2} \\ \vdots \end{bmatrix} \tag{43}$$

```
# Measurement expected for state x
def hx(x):
    xp = float(x[0]) # x-position
    yp = float(x[1]) # y-position
    vp   = float(x[2]) # y-velocity

    nT = (len(x)-3)/2 # Number of targets
    out = []

    for i in range(int(nT)):
        tIndx = i*2+3 # Next target x-coordinate index in state array
        element = [math.sqrt((xp - float(x[tIndx]))**2 +
                             (yp - float(x[tIndx+1]))**2)]
        out.append(element)

    out = asarray(out)
    return out
```

Listing 15: Expected measurement function with dynamic targets

As for the Jacobian function, the array can be built up line-by-line. As the target positions are fixed in the state array and they should not move, the partial derivatives with respect to each target position can be set to 0. Hence, the Jacobian matrix can be calculated.

$$J(\bar{x}) = \begin{bmatrix} \frac{\partial h_1}{\partial x} & \frac{\partial h_1}{\partial y} & \frac{\partial h_1}{\partial \dot{y}} & \frac{\partial h_1}{\partial x_1^t} & \cdots \\ \frac{\partial h_2}{\partial x} & \frac{\partial h_2}{\partial y} & \frac{\partial h_2}{\partial \dot{y}} & \frac{\partial h_2}{\partial x_1^t} & \cdots \\ \frac{\partial h_3}{\partial x} & \frac{\partial h_3}{\partial y} & \frac{\partial h_3}{\partial \dot{y}} & \frac{\partial h_3}{\partial x_1^t} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$= \begin{bmatrix} \frac{x - x_1^t}{\sqrt{(x - x_1^t)^2 + (y - y_1^t)^2}} & \frac{y - y_1^t}{\sqrt{(x - x_1^t)^2 + (y - y_1^t)^2}} & 0 & 0 & \cdots \\ \frac{x - x_2^t}{\sqrt{(x - x_2^t)^2 + (y - y_2^t)^2}} & \frac{y - y_2^t}{\sqrt{(x - x_2^t)^2 + (y - y_2^t)^2}} & 0 & 0 & \cdots \\ \frac{x - x_3^t}{\sqrt{(x - x_3^t)^2 + (y - y_3^t)^2}} & \frac{y - y_3^t}{\sqrt{(x - x_3^t)^2 + (y - y_3^t)^2}} & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad (44)$$

Finally, the EKF can be run by iteratively updating the camera position, providing the measurements for each time step, and then performing the predict and update step of the filter, as shown in listing 16. The results of the camera move with varying number of targets can be seen in section 4.2.2.

```
1.  # At state x return Jacobian matrix
2.  def XJacobian(x):
3.      xp = float(x[0]) # x-position
4.      yp = float(x[1]) # y-position
5.      vp   = float(x[2]) # y-velocity
6.
7.      nS = len(x) # Number of states
8.      nT = (len(x)-3)/2 # Number of targets
9.      out = np.zeros((int(nT), nS)) # Jacobian output array shape
10.     for i in range(int(nT)):
11.         tIndx = i*2+3 # Next target x-coordinate index in state array
12.         element = np.zeros(nS)
13.         denom = math.sqrt((xp - float(x[tIndx]))**2 +(yp - float(x[tIndx+1]))**2)

14.         element[0:2] = array([[(xp - float(x[tIndx]))/denom,
15.                                (yp - float(x[tIndx+1]))/denom]])
16.         out[i] = element
17.
18.     return out
```

Listing 16: Jacobian function with dynamic targets

While the previous camera tracking simulations have involved measurements relating to the distance from the camera, this is not a measurement that can be obtained by a purely optical SLAM system. Therefore, in the next extension, the angle of each target with respect to the camera will be the measurement provided to the filter. This is a measurement that is more typical from an optical system. In addition to changing the measurements to angles, a bearing state will be added to the camera properties. An image of the simulation with bearing and angle visualisations can be seen in figure 27.

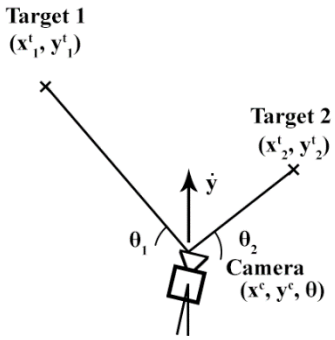

Figure 27: Simulation setup for camera moving past targets with bearing offset

To switch the measurement from an angle in the multiple targets experiment first the *update_camera* function must be changed – it must update the bearing measurement by adding process noise, calculate the angle between the camera and the target and add the bearing offset to the angle. The *update_camera* function can be seen in listing 17.

```
1.  def update_camera(dt):
2.      global x_pos, y_pos, y_vel, bearing, targets
3.      # Process noise
4.      x_pos = x_pos  + .1 * randn()
5.      y_vel = y_vel + .01 * randn()
6.      bearing = bearing + .01 * randn()
7.      y_pos = y_pos + y_vel * dt
8.
9.      measurements = []
10.     for T in targets:
11.         # Calculate angle between camera and each target with
12.         # bearing offset and fixed variance error
13.         err = .05 * randn()
14.         ang = math.atan2((T[1]- y_pos), (T[0] - x_pos)) + bearing
15.         measurements.append([ang + err])
16.
17.     measurements = asarray(measurements)
18.
19.     return measurements
```

Listing 17: Camera update function with bearing state

Next, the initial state array is created with the states describing the camera close to the true values but the states describing the target positions exactly at the target positions.

$$\bar{x} = \begin{bmatrix} \text{camera x position} \\ \text{camera y position} \\ \text{camera y velocity} \\ \text{camera bearing} \\ \text{target 1 x-coordinate} \\ \text{target 1 y-coordinate} \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0.4 \\ -222.4 \\ 522.4 \\ \vdots \end{bmatrix}, \text{ ekf.x} = \begin{bmatrix} 5 \\ 7 \\ 2.5 \\ 1.9 \\ -222.4 \\ 522.4 \\ \vdots \end{bmatrix}$$

The measurement error is fixed at a sample from the normal distribution with variance 0.05 (irrespective of distance or angle) therefore the uncertainty array, *ekf.R* for every measurement can be fixed at 0.05. The process noise array, *ekf.Q* can be set based on the amount of process noise added to each variable and tweaked until the required sensitivity is achieved. As always, *ekf.P* is set high for model flexibility.

$$\text{ekf.R} = \begin{bmatrix} 0.05 & 0 & 0 \\ 0 & 0.05 & 0 \\ 0 & 0 & \ddots \end{bmatrix}, \text{ ekf.Q} = \begin{bmatrix} 10^{-2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 10^{-7} & 0 & 0 & 0 & 0 \\ 0 & 0 & 10^{-4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 10^{-5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \ddots \end{bmatrix}$$

$$\text{ekf.P} = \begin{bmatrix} 1000 & 0 & 0 \\ 0 & 1000 & 0 \\ 0 & 0 & \ddots \end{bmatrix}$$

Next, the expected measurement function, *hx*, is updated to calculate angle measurements and accommodate the new bearing state.

$$\text{H}(\bar{x}) = \begin{bmatrix} \tan^{-1}\left(\dfrac{y_1^t - y}{x_1^t - x}\right) + \theta \\ \tan^{-1}\left(\dfrac{y_2^t - y}{x_2^t - x}\right) + \theta \\ \vdots \end{bmatrix} \tag{45}$$

The Jacobian matrix can then be built up line-by-line as before, with the partial derivatives of the *math.atan2* function being taken with respect to each state. Again, as the target locations are fixed and fully known, zeros can be placed in a space where the derivative is taken with respect to a target coordinate.

$$J(\bar{x}) = \begin{bmatrix} \frac{\partial h_1}{\partial x} & \frac{\partial h_1}{\partial y} & \frac{\partial h_1}{\partial \dot{y}} & \frac{\partial h_1}{\partial \theta} & \frac{\partial h_1}{\partial x_1^t} & \cdots \\ \frac{\partial h_2}{\partial x} & \frac{\partial h_2}{\partial y} & \frac{\partial h_2}{\partial \dot{y}} & \frac{\partial h_2}{\partial \theta} & \frac{\partial h_1}{\partial x_1^t} & \cdots \\ \frac{\partial h_3}{\partial x} & \frac{\partial h_3}{\partial y} & \frac{\partial h_3}{\partial \dot{y}} & \frac{\partial h_3}{\partial \theta} & \frac{\partial h_1}{\partial x_1^t} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$= \begin{bmatrix} \frac{y_1^t - y}{(x_1^t - x)^2 + (y_1^t - y)^2} & \frac{x_1^t - x}{(x_1^t - x)^2 + (y_1^t - y)^2} & 0 & 1 & 0 & \cdots \\ \frac{y_2^t - y}{(x_2^t - x)^2 + (y_2^t - y)^2} & \frac{y_2^t - y}{(x_2^t - x)^2 + (y_2^t - y)^2} & 0 & 1 & 0 & \cdots \\ \frac{y_3^t - y}{(x_3^t - x)^2 + (y_3^t - y)^2} & \frac{y_3^t - y}{(x_3^t - x)^2 + (y_3^t - y)^2} & 0 & 1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad (46)$$

As the measurements are based on angles now, there is a problem when the EKF tries to compute the difference between the expected measurement for a given state (*hx*(x)) and the actual measurement for that state (z). Normally this would just be calculated as z-*hx*(x) however an issue occurs on the border of the sign change. By pure subtraction, the difference between 179° and -179° is 358° whereas, in reality, these two angles differ by only 2°. This means the *ekf.update* step requires an additional function parameter to calculate the angular difference between two measurements. The *residual* function to do this can be seen in listing 18.

```
def residual(a, b):
    y = a - b # calculate subtraction residual
    y = y % (2 * np.pi)    # move to 0 -> 2*pi
    y[y > np.pi] -= 2 * np.pi # rescale to -pi -> +pi
    return y
```

Listing 18: Residual function

Again, the EKF can be run by updating the camera position at specified time intervals, updating the model by passing the measurements, Jacobian function, expected measurement function and residual function and finally predicting the next state based on the state transition matrix. Results for this experiment can be seen in section 4.2.2.

The final stage required for a complete 2D SLAM problem is to introduce the mapping element. In this simulation space, this involves initialising the positions of targets incorrectly and simultaneously localizing the camera whilst determining where the measurements are being taken from (the target positions).

The *update_camera* function is unchanged from the previous example. There is no process noise added to the targets as they are completely static and fixed. The initialisation of the state matrix is the same as before except a number sampled from the normal distribution scaled by 75 is added to the target positions making them highly inaccurate. The Python implementation of this can be seen in listing 19.

```
# Make incorrect guess for camera property states
ekf.x = array([x_pos + 5, y_pos + 7, y_vel + .5, bearing + 1.5])
# Append target positions with very large offset to initial state array
ekf.x = np.append(ekf.x, asarray(targets) + 75 * randn(nT, 2))
ekf.x = np.reshape(ekf.x, (1, 4 + 2 * nT)).T
```

Listing 19: EKF initialisation with uncertain targets

$$\bar{x} = \begin{bmatrix} \text{camera x position} \\ \text{camera y position} \\ \text{camera y velocity} \\ \text{camera bearing} \\ \text{target 1 x-coordinate} \\ \text{target 1 y-coordinate} \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0.4 \\ -222.4 \\ 522.4 \\ \vdots \end{bmatrix}, \text{ ekf.x} = \begin{bmatrix} 5 \\ 7 \\ 2.5 \\ 1.9 \\ -85.8 \\ 490.4 \\ \vdots \end{bmatrix}$$

The uncertainty in the measurements remains unchanged (0.05 across the diagonals). A small amount of process uncertainty is added to the target positions. While the targets have no process noise, this gives the model a little leeway to adjust the predictions throughout the run as opposed to perfectly guessing the first time.

$$\text{ekf.R} = \begin{bmatrix} 0.05 & 0 & 0 \\ 0 & 0.05 & 0 \\ 0 & 0 & \ddots \end{bmatrix}, \text{ ekf.Q} = \begin{bmatrix} 10^{-2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 10^{-7} & 0 & 0 & 0 & 0 \\ 0 & 0 & 10^{-4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 10^{-5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 10^{-7} & 0 \\ 0 & 0 & 0 & 0 & 0 & \ddots \end{bmatrix}$$

$$\text{ekf.P} = \begin{bmatrix} 1000 & 0 & 0 \\ 0 & 1000 & 0 \\ 0 & 0 & \ddots \end{bmatrix}$$

The expected measurement function remains unchanged as seen in equation 47.

$$\text{H}(\bar{x}) = \begin{bmatrix} \tan^{-1}\left(\dfrac{y_1^t - y}{x_1^t - x}\right) + \theta \\ \tan^{-1}\left(\dfrac{y_2^t - y}{x_2^t - x}\right) + \theta \\ \vdots \end{bmatrix} \tag{47}$$

The Jacobian function now requires the derivatives to be taken with respect to the target positions so that they can be linearized about the current state estimate and updated along with the camera parameters.

$$\mathrm{J}(\bar{x}) = \begin{bmatrix} \frac{\partial h_1}{\partial x} & \frac{\partial h_1}{\partial y} & \frac{\partial h_1}{\partial \dot{y}} & \frac{\partial h_1}{\partial \theta} & \frac{\partial h_1}{\partial x_1^t} & \cdots \\ \frac{\partial h_2}{\partial x} & \frac{\partial h_2}{\partial y} & \frac{\partial h_2}{\partial \dot{y}} & \frac{\partial h_2}{\partial \theta} & \frac{\partial h_1}{\partial x_1^t} & \cdots \\ \frac{\partial h_3}{\partial x} & \frac{\partial h_3}{\partial y} & \frac{\partial h_3}{\partial \dot{y}} & \frac{\partial h_3}{\partial \theta} & \frac{\partial h_1}{\partial x_1^t} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$= \begin{bmatrix} \frac{y_1^t - y}{(x_1^t - x)^2 + (y_1^t - y)^2} & \frac{x_1^t - x}{(x_1^t - x)^2 + (y_1^t - y)^2} & 0 & 1 & \frac{y - y_1^t}{(x_1^t - x)^2 + (y_1^t - y)^2} & \cdots \\ \frac{y_2^t - y}{(x_2^t - x)^2 + (y_2^t - y)^2} & \frac{y_2^t - y}{(x_2^t - x)^2 + (y_2^t - y)^2} & 0 & 1 & 0 & \cdots \\ \frac{y_3^t - y}{(x_3^t - x)^2 + (y_3^t - y)^2} & \frac{y_3^t - y}{(x_3^t - x)^2 + (y_3^t - y)^2} & 0 & 1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad (48)$$

This will produce a cascading effect in the matrix where the derivatives with respect to a target position will only be non-zero if that specific target is required to calculate that measurement. The Jacobian function can be seen in listing 20.

```
# At state x return Jacobian matrix
def XJacobian(x):
    xp = float(x[0]) # x-position
    yp = float(x[1]) # y-position
    vp   = float(x[2]) # y-velocity
    bear = float(x[3]) # bearing

    nS = len(x) # Number of states
    nT = (len(x)-4)/2 # Number of targets
    out = np.zeros((int(nT), nS)) # Jacobian output array shape
    for i in range(int(nT)):
        tIndx = i*2+4 # Next target x-coordinate in state array
        element = np.zeros(nS)
        # Calculate partial derivatives with respect to target
        targets_partial_deriv = array([(-float(x[tIndx+1]) + yp)/
                                       ((float(x[tIndx]) - xp)**2 + (float(x[tIndx+1])
                                       - yp)**2), (float(x[tIndx]) - xp)/
                                       ((float(x[tIndx]) - xp)**2 + (float(x[tIndx+1])
                                       - yp)**2)])
        element[tIndx:tIndx+2] = targets_partial_deriv

        element[0:4] = array ([-(-float(x[tIndx+1]) + yp)/
                                      ((float(x[tIndx])- xp)**2 + (float(x[tIndx+1])
                                       - yp)**2), -(float(x[tIndx]) - xp)/
                                      ((float(x[tIndx]) - xp)**2 + (float(x[tIndx+1])
                                       - yp)**2), 0, 1])
        out[i] = element
    return out
```

Listing 20: Jacobian function with uncertain targets

The same *residual* function is also required to resolve the sign change in the *math.atan2* function. As always, the camera is updated at set time intervals, the EKF filter is updated and the next state predicted. The final 2D SLAM results can be seen in section 4.2.2.

4.2 Results and Discussion

4.2.1 Using the EKF to track a single target using radar

To begin, the radar track of the plane is done based off straight-line distance measurements while the state variables describing the plane state are the x-position (distance), x-velocity (velocity) and y-position (altitude). A plot of the three states over time can be seen in figure 28. In these graphs, the black line is the ground truth of each state as generated by the *update_radar* function and the red line is the predicted state from the EKF. From these graphs, it is clear that the filter is doing a good job at predicting the plane position: the x-position follows nearly perfectly, and the red line converges to the actual state quickly from the initial incorrect guess.

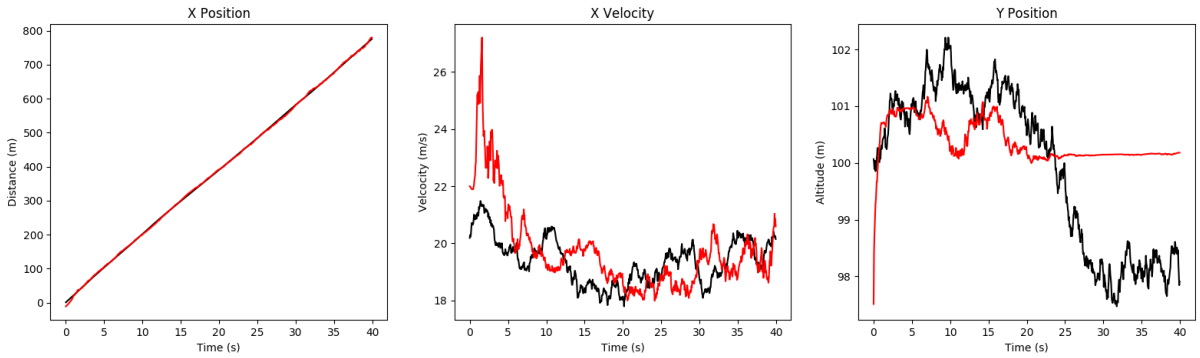

Figure 28: Plots of camera states over time for a single target with radar range measurements

Another way to look at the plane movement is through a plot showing the position of the target (the blue dot) and plotting the x-position against the y-position of the plane (the black and red lines are the ground truth and prediction, respectively) as can be seen in figure 28. In figure 29 the measurement from the radar of the straight-line distance to the plane can be seen. As the further away the plane gets, the greater the error in the measurement. A dotted line is used here to give some reference to the velocity of the ground truth and prediction by the spacing between dots.

By decreasing the process noise in the *ekf.Q* matrix, the sensitivity of the filter can be adjusted. Higher uncertainties result in more noisy filtered results closer to the ground truth whereas less uncertainty results in a more stable filter prediction. A comparison of the x-velocity (altitude) plots at uncertainties of 10^{-1}, 10^{-3} and 10^{-5} can be seen in figure 30.

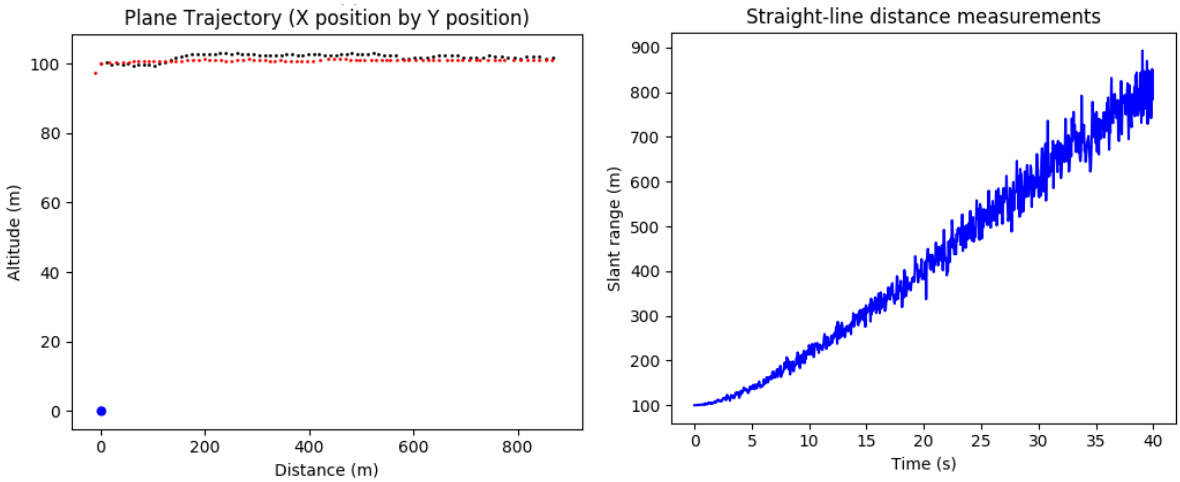

Figure 29: Plot of target position relative to radar and received range measurements

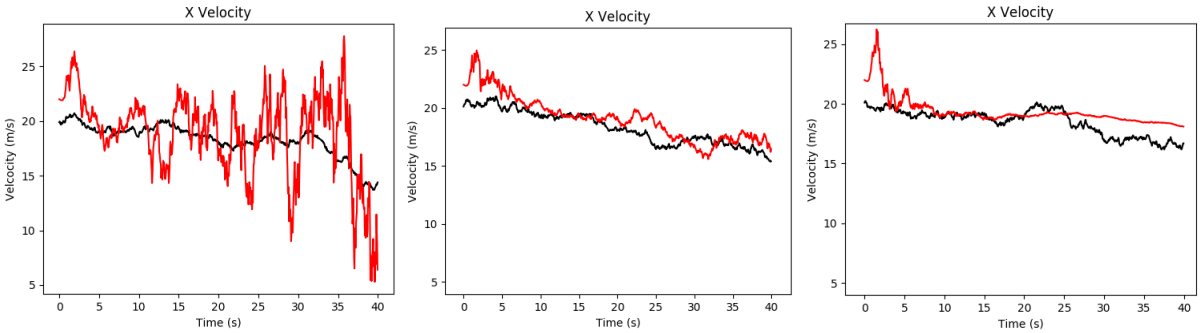

Figure 30: X-Velocity state plots over time for varying degrees of process noise uncertainty, left to right: 10^{-1}, 10^{-3}, 10^{-5}

A value of 10^{-3} was decided on as a balance of model stableness as well as flexibility to adjust to change.

The next experiment was to change the measurement from a straight-line range distance to an angle measurement. A plot of the three states over time can be seen in figure 31. In these graphs, the black line is the ground truth of each state as generated by the *update_radar* function and the red line is the predicted state from the EKF.

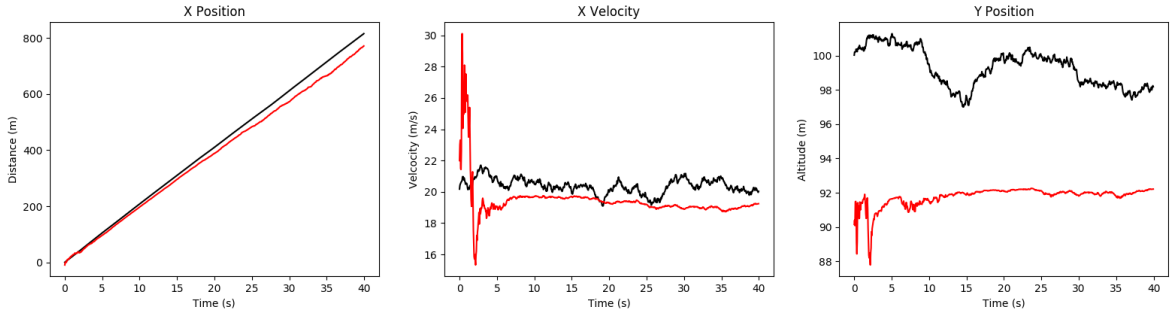

Figure 31: Plot of camera states over time for a single target with radar angle measurements

This filter performs worse than the one based on straight-line range measurements only. Both the x-velocity and y-position converge on the wrong values the x-position of the plane gets

further away from the correct position as time goes on. However, inspecting the plane trajectory and measurement graph in figure 32 give some indications as to why this is the case.

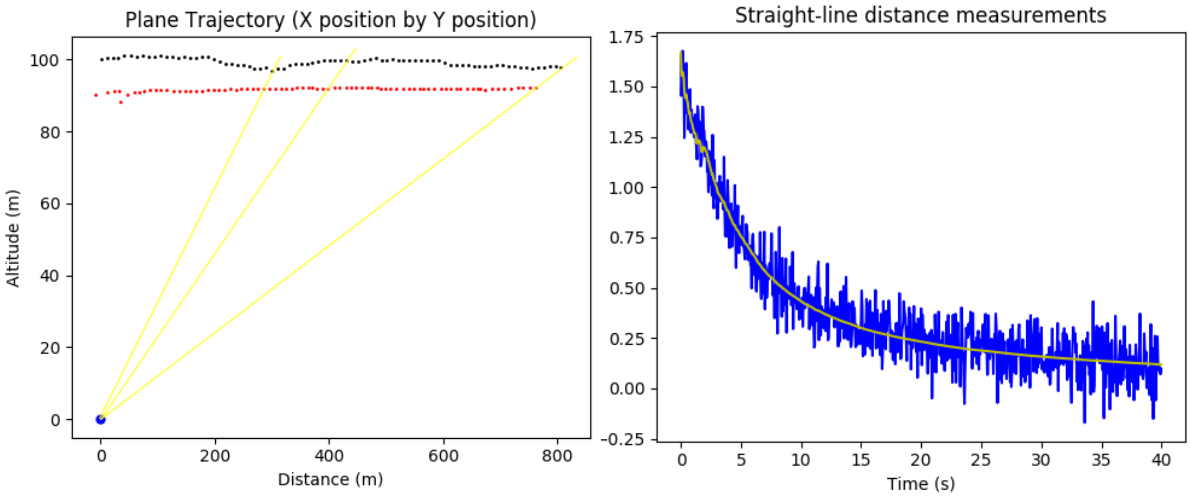

Figure 32: Plot of target position relative to radar and received angle measurements

In the left-hand graph, the yellow lines pass though the radar, the predicted point and the actual point at the same time-step. At the same time-step, the line passes through both points and therefore the same angle is being made between the radar and the prediction, and the radar and the actual position. Furthermore, the blue line in the right-hand graph shows the angle measurements from the radar and the yellow line shows the expected angle given the specific state at that time-step. This line passes nicely through the actual measurements and therefore indicates the predicted state fits with the actual measurement. So there appears to be a kind of 'alternative-solution' to the problem and a different velocity and altitude pair that adheres to the angle measurements. For a given angle measurement the plane could lie on an infinitely long line whereas for a given range measurement the plane can only lie on the circumference of a circle, as seen in figure 33. The solution to this problem is to get more measurements – these can be either different measurements or more angle measurements to give more reference points.

Figure 33: Comparison of challenge between range and angle measurements

To improve upon the angle measurement case, the radial velocity of the plane measured by the radar was added to the filter. This will test a two measurement filter as well as attempt to fix the previous issue. A plot of the three states over time can be seen in figure 34. Again, the black line is the ground truth of each state as generated by the *update_radar* function and the red line is the predicted state from the EKF.

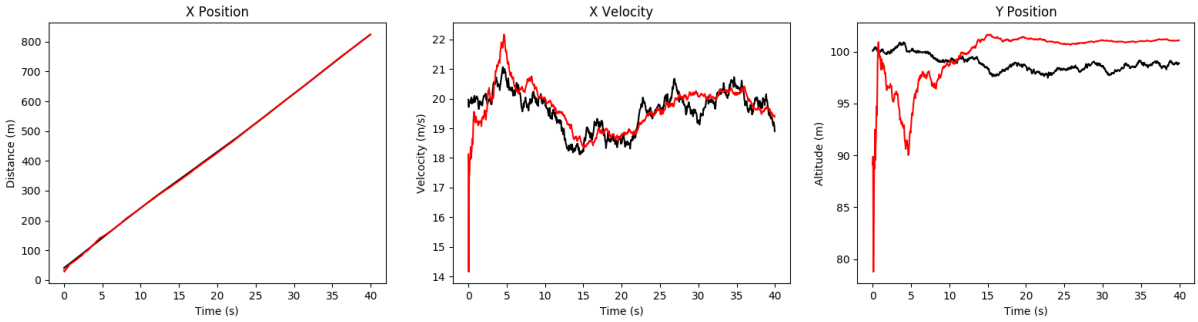

Figure 34: Plot of camera states over time for a single target with radar angle + radial velocity measurements

This performs considerably better than the angle only measurement filter and adding the radial velocity measurement improves the x-velocity guess greatly. This, in turn, improves the y-position estimate as it effectively removes a variable from the equation. Looking at the trajectory and measurement graphs in figure 35 also show this improved performance. The blue dot indicates the radar position in the left graph. In the central and right-hand graph, the blue line is the actual measurement and the yellow line is the expected measurement from the predicted state.

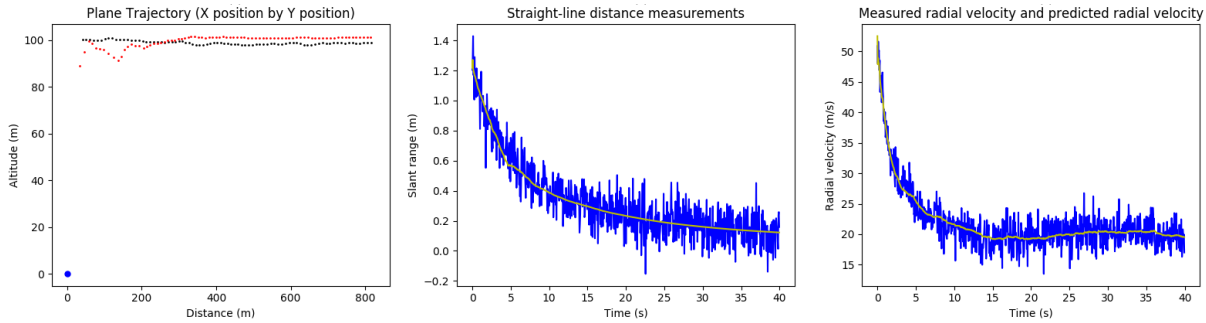

Figure 35: Plot of target position relative to radar and received angle plus radial velocity measurements

4.2.2 Using the EKF to track a camera moving through a field of targets

In the previous section, an EKF was used to track a single target (a plane) based on one or two measurements. The next experiments push this further towards the desired SLAM problem; a camera is moving with respect to multiple stationary targets around it.

To begin with, the camera is moving in a straight line, forwards (in the y-plane). This means the three states describing the camera are the x-position, y-position, and y-velocity. There are two static targets placed on either side but in front of, the camera. As a reminder, the simulation space appears as shown in figure 36.

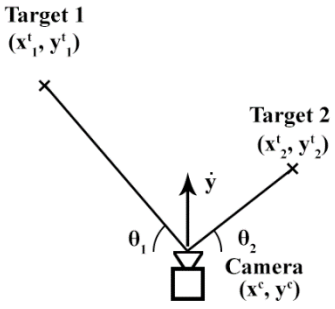

Figure 36: Simulation setup for camera moving past targets

The measurements that will be taken in this experiment are the straight-line distances from the camera to the two targets as calculated by the Pythagorean equation. The simulation is run for 20 seconds so as not to get too far away from the targets and so that the noise does not get too large. A plot of the three states over time can be seen in figure 37. Again, the black line is the ground truth of each state as generated by the *update_camera* function and the red line is the predicted state from the EKF.

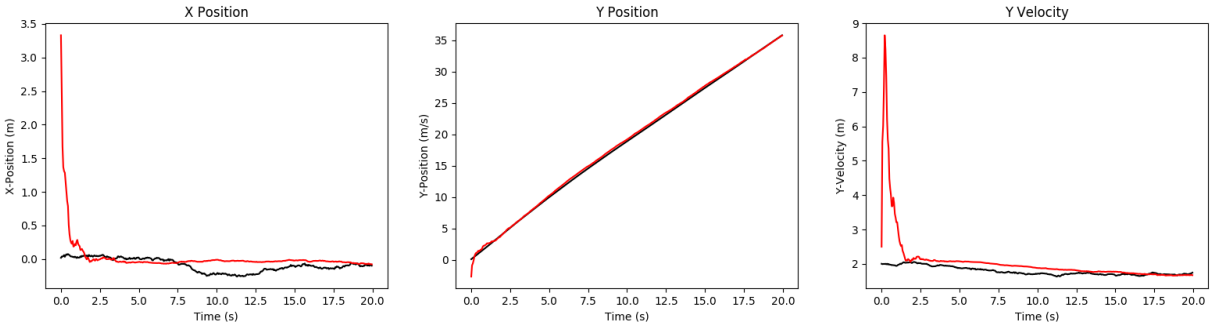

Figure 37: Plot of camera states over time for camera moving past two static targets

As can be seen on the graphs, the incorrect starting positions are corrected and updated right at the start meaning the prediction converges on the actual values quickly. Furthermore, this behaviour can be seen on the trajectory graph in figure 38.

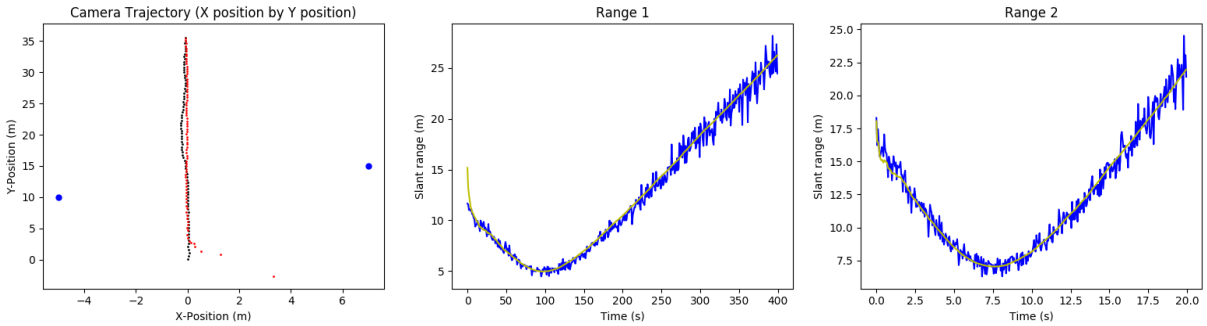

Figure 38: Plot of camera position relative to target positions and received range measurements

The prediction starts off way to the right and quickly corrects itself while maintaining the correct velocity to stay in line with the ground truth. The ‘Range 1’ and ‘Range 2’ plots show the actual measurement values in blue and the measurement expected for the predicted state in yellow. The noise in the measurement remains low at the beginning but grows very large near the end of the simulation as the camera gets further away. However, as the model has done most of the converging by this point, simply predicting the next state from the previous state becomes more viable and the measurements become less relied upon. If there were a drastic change in state at some point much further along, however, the simulation could benefit from a static

reference point further up to provide some measurements with less noise. Despite this, increasing the amount of process noise by a factor of 10 still produces a very high-quality track as seen in figure 39.

A further tracking point can be added to this model to add an extra piece of information and in theory, improve tracking. The camera states plot for three targets and three range measurements can be seen in figure 40.

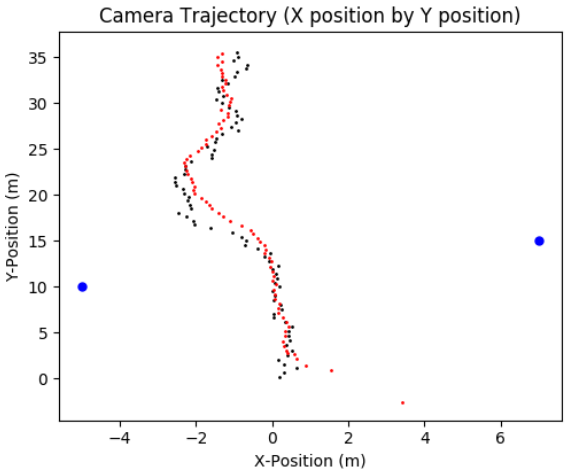

Figure 39: Plot of camera position relative to targets with increased process noise

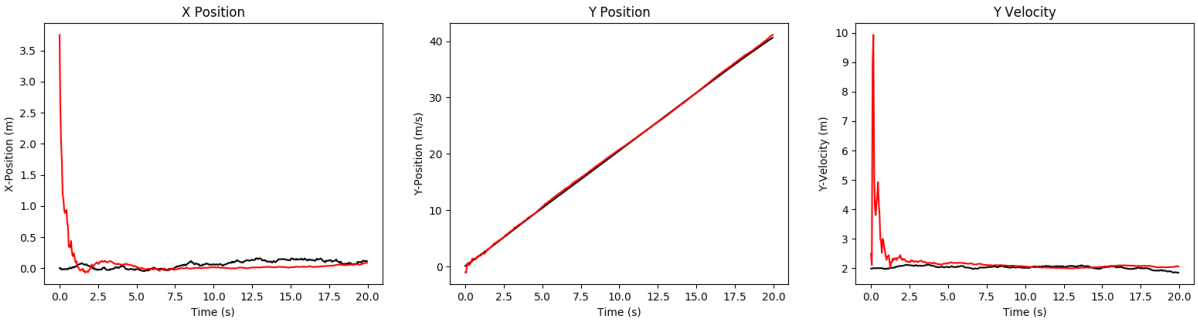

Figure 40: Plot of camera states over time for camera moving past three static targets

In these plots, the model performs similarly well to the case with two static measurement points indicating this is fairly sufficient for this simple scenario.

To extend this further, instead of fixing target points manually and calculating the relevant matrices by hand, any number of target positions will be randomly generated and added to the system state array (along with the camera states) for the relevant matrices to be calculated dynamically for any target combinations. This means different numbers of targets can be tested by changing only one variable.

In this example the noise will be increased by a factor of 10 (as previously seen in figure 40) as more targets mean better performance and hence, to stretch the capabilities of the model more noise is preferred. Below (in figures 41 and 42) are four trajectory graphs for four different number of targets: 2, 5, 15 and 40.

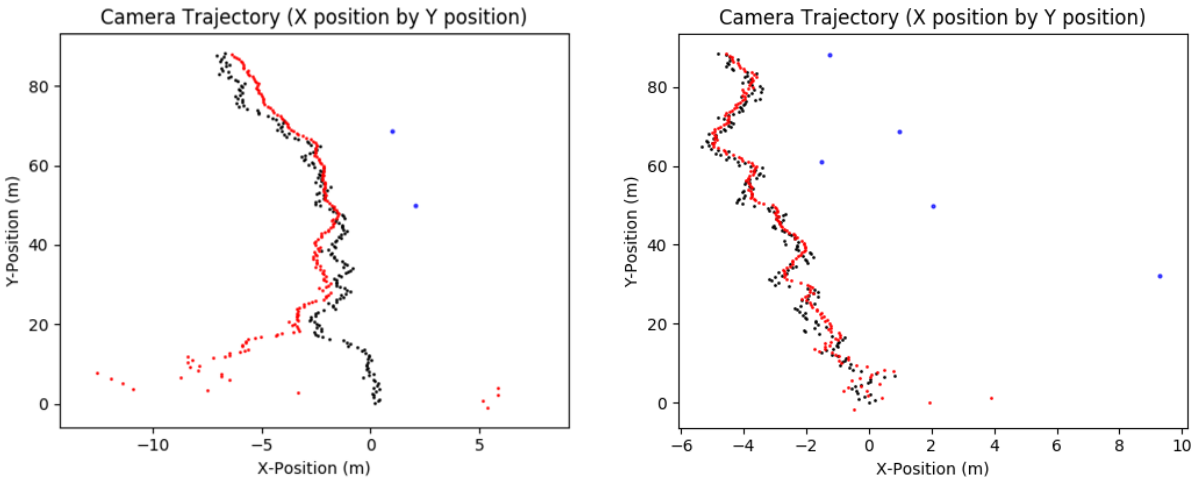

Figure 41: Plot of camera position relative to targets (left: 2 targets, right: 5 targets)

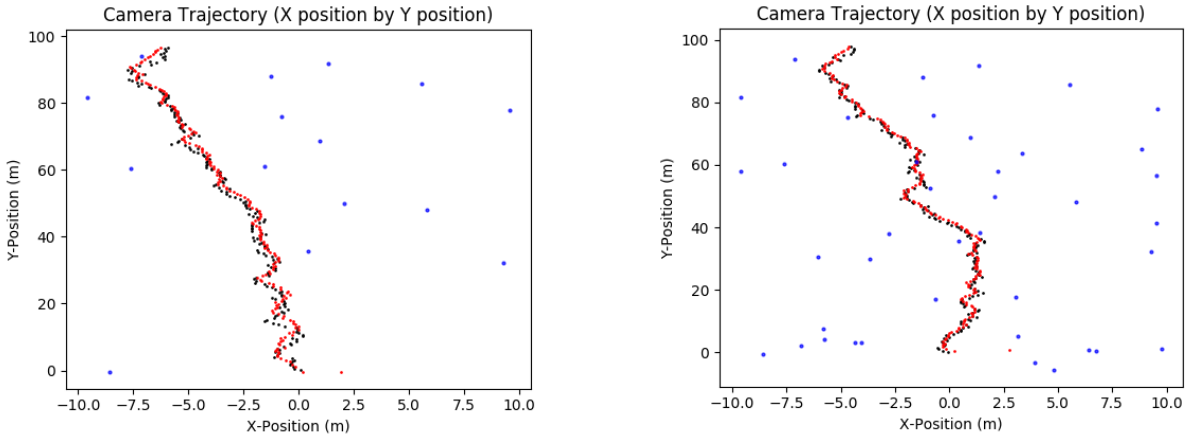

Figure 42: Plot of camera position relative to targets (left: 15 targets, right: 40 targets)

From these graphs, the effect of adding more targets is very clear. While the two targets with high noise track in the previous example (figure 40) was better than the one here, this is because the targets in the previous example were specifically placed. In this example, the targets are more similarly placed and therefore have a greater information overlap. However, increasing the number of targets to 5 improves the model massively – increasing convergence time and overall 'tightness'. At 15 and 40 targets the improvement is less drastic although still converges more quickly and stays slightly closer to the ground truth.

Next, as the defined SLAM problem here is one based on optical camera input, the range cannot be inferred so the angle of keypoints will be the primary measurement. The following experiments will work based on the angle of the target with respect to the camera. In addition to the measurement change, an extra camera defining state, the bearing, will be added to the state array. As a reminder, the new simulation scenario can be seen in figure 43.

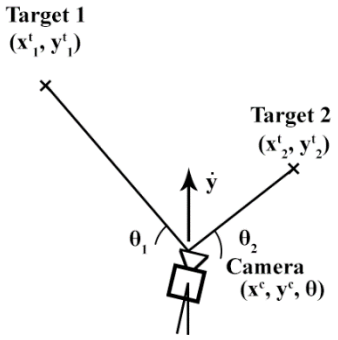

Figure 43: Simulation setup for camera moving past targets with bearing offset

In Python, the in-built *math* module provides two functions to calculate the inverse tangent of a side-length ratio: *math.atan* and *math.atan2*. The functions differ by the fact that the *math.atan* function only respects the sign of the ratio of y and x whereas *math.atan2* looks at the signs of both y and x separately. Therefore, this makes it impossible in some cases to tell which quadrant a point falls in, given an angle calculated by *math.atan*. To visualise the difference between the two functions, a circle plot of points was created, and the output of both functions plotted, as seen in figure 44.

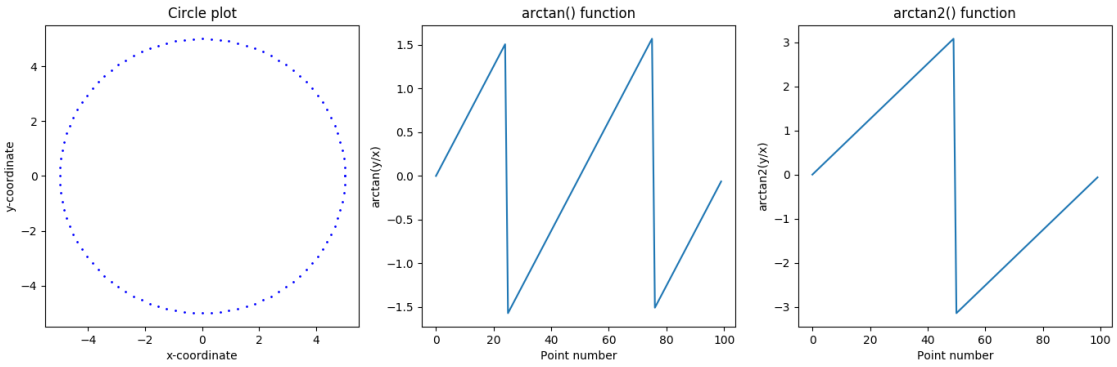

Figure 44: Plot of arctan() function (middle) and arctan2() function (right) for points arranged in a circle (left)

As can be seen in the plots, the *math.atan* function undergoes more sign changes and the same output can be found for a different set of input coordinates. On the other hand, *math.atan2* only undergoes one sign change (at coordinates [-5, 0]) and a unique result is produced for every set of coordinates. For SLAM, sudden sign changes are problematic and must be addressed, so for this reason, *math.atan2* will be used to calculate angles as it encounters fewer sign changes.

To see why the *math.atan* function is inferior to the *math.atan2* function for this problem, plots of the simulation with the *math.atan* function in place of *math.atan2* can be seen in figure 45. On the left is a simulation where the x-position process noise variance is 0.01 and on the right is a simulation where the x-position process noise variance is 0.1. Both simulations are getting measurements from 20 targets. Both plots are zoomed in to better examine what is happening.

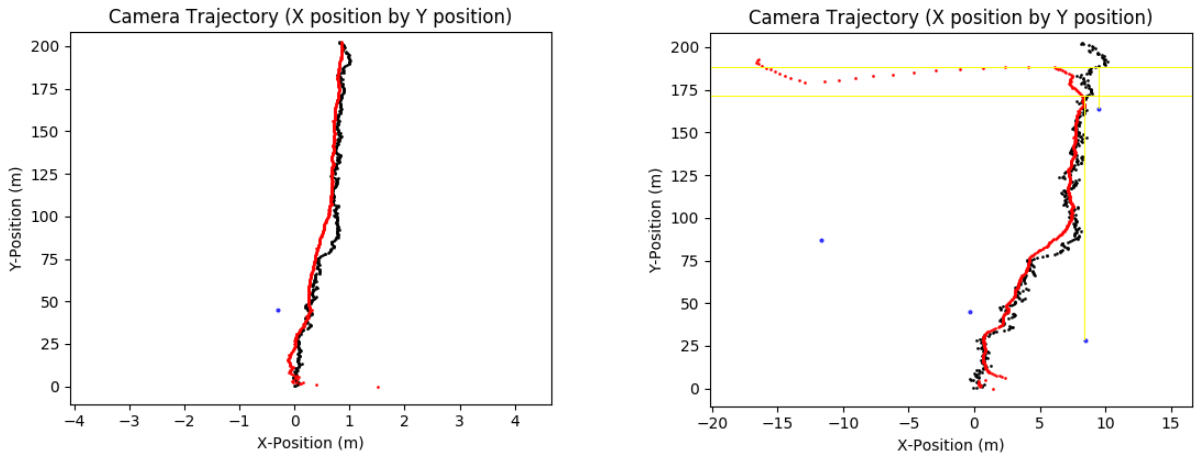

Figure 45: Display of limitations of the atan() function in an EKF

In the low x-position process noise example on the left, there is no issue with the track as no points come in-front of or behind the camera. However, when there is a little more noise and the camera passes in-front of or behind the targets the prediction goes wrong. The yellow lines on the right graph show the points where the predicted camera position passes in-front of the targets and cause a sign change in the measurement. These points coincide with the prediction deviating from the ground truth.

This same behaviour is also present for the *math.atan2* function but only when a target passes on the left-hand side of the camera. However, as discussed in section 3.2.2 the residual function corrects for this sign change and allows for target positions in any location. The behaviour of the same simulation with and without the residual function can be seen in figure 46.

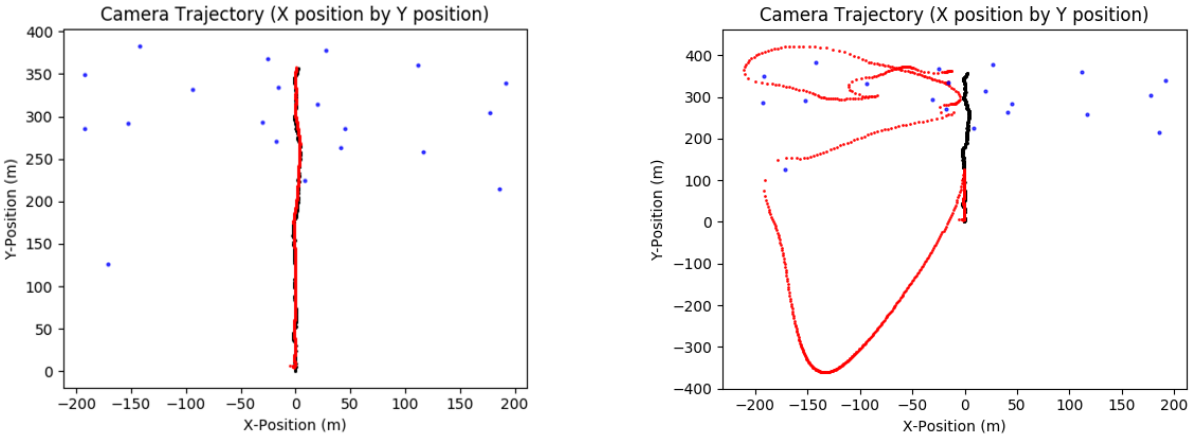

Figure 46: Visual explanation of the importance of the residual function

The example with the residual function creates a perfect track. Without the residual function, the track is fine until the point where one of the targets passes on the left-hand side of the camera where it spirals out of control.

With regards to the extra bearing measurement, with 20 tracking targets, the bearing estimate converges extremely quickly and tracks near perfectly, shown in figure 47.

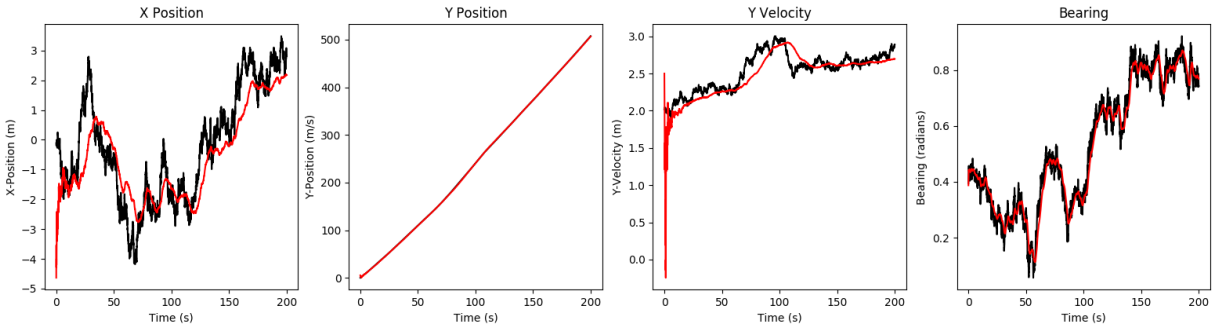

Figure 47: Plot of camera states over time (including bearing) with 20 tracking targets

This is to be expected as the bearing is just a constant that is added to every measurement.

The final addition to this simulation is adding some uncertainty to the positions of the static targets. This means the filter now has to determine every state in the state vector from an incorrect guess to localize the camera accurately as well as map out the environment (the positions of all the tracking points).

While the state graphs and trajectory graphs shown previously are still useful to examine the accuracy of the camera track, a way of assessing the accuracy of the target positions is now also required. To do this, the position error in each target in both the x-direction and y-direction is recorded. This is the difference between the position of the prediction and the actual position of the target. Ideally, these distances will converge on zero as the predictions become more accurate. In addition to these error graphs, animated plots will be used to show the path of the ground truth camera, the path of the camera prediction, the ground truth position of the targets and the prediction of each target. Using an animation provides a visualisation of the speed of convergence, the camera, and the prediction as well as when the measurement from each target becomes relevant to determining the camera state.

In the first example (random seed set to 0 and 40 targets created) the graphs of the states describing the camera position can be seen in figure 48 and the error plot in figure 49.

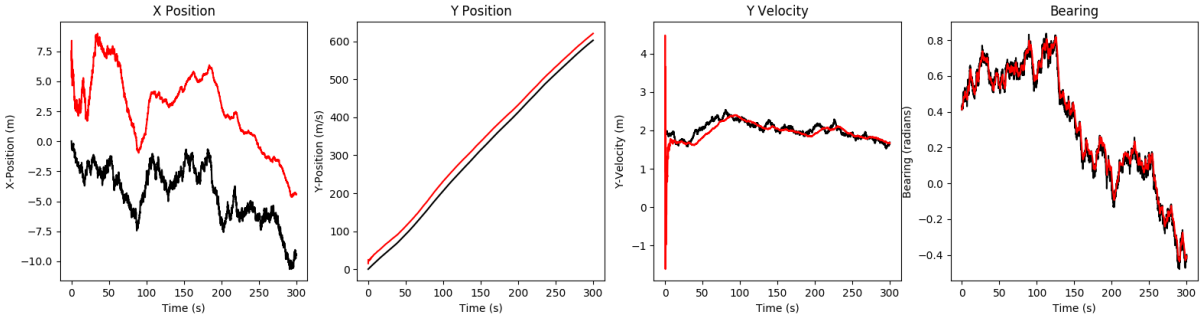

Figure 48: Plot of camera states with uncertain target positions (seed = 0, targets = 40)

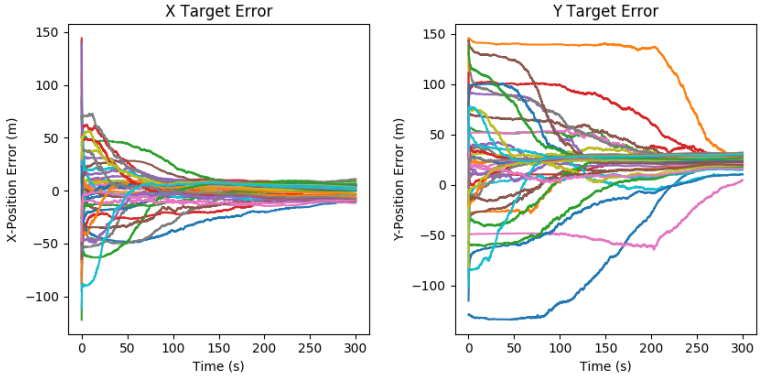

Figure 49: Error in x-position (left) and y-position (right) of targets (seed = 0, targets = 40)

From these graphs, it appears that the x-position and y-position of the camera are slightly out compared to the ground truth. However, both the x-position and y-position follow the same shape as the ground truth but are just offset. Looking at the x and y error graphs for the positions of the targets gives a little more insight into this.

While both the x target error and y target error do mainly appear to converge on a point, by the final time step the average error of the x-position is 0.14m and the average error of the y-position is 23.16m. For a perfect track, the error of the targets should average at 0 for both x and y. In the error graphs, the x error converges much more quickly whereas some of the y tracking points take longer to converge.

Looking at the animated plot tells the same story. In the animated graphs, the black line is the ground truth camera, the red line is the predicted camera position, the blue crosses are the ground truth target and the green dots are the predicted targets. The yellow lines link the respective ground truth and predicted targets. Three frames from the video taken at successive points can be seen in figure 50.

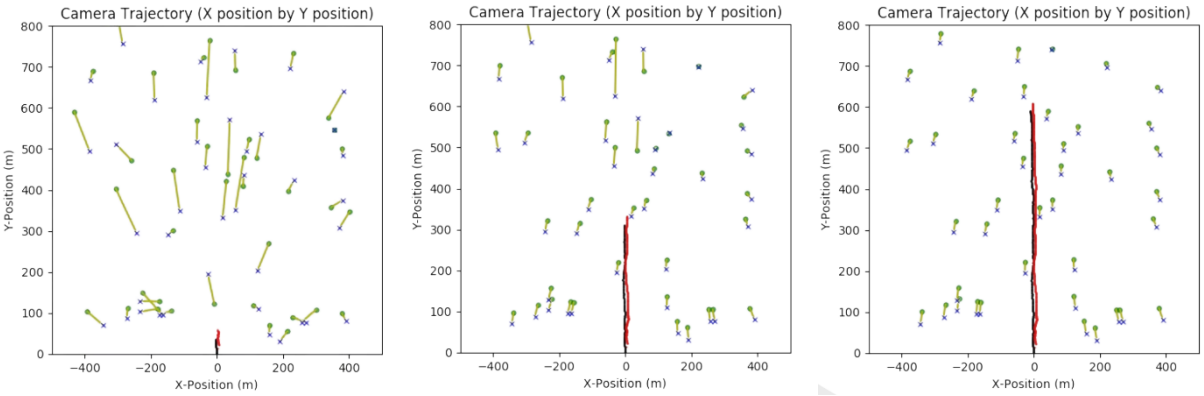

Figure 50: Animated sequence of camera and target convergence (seed = 0, targets = 40)

The speed of convergence in either direction can also be seen in the animation, as all the yellow lines point straight up quite quickly but the tracking points towards the top of the frame only converge in the y-direction as the camera gets closer.

In an early frame, the predicted camera position is initialised in front of the ground truth camera position and the target positions are initialised randomly away from their true positions. As the camera progresses the target predictions get closer to the ground truth although the camera prediction stays slightly in-front of the ground truth camera. By the final frame where the targets predictions have almost fully settled, the predicted targets are all slightly in front of the ground truth along with the camera. All the predictions are offset forwards from where they should be.

Looking at another simulation example (seed 123 with 17 targets), the graphs of the camera states can be seen in figure 51.

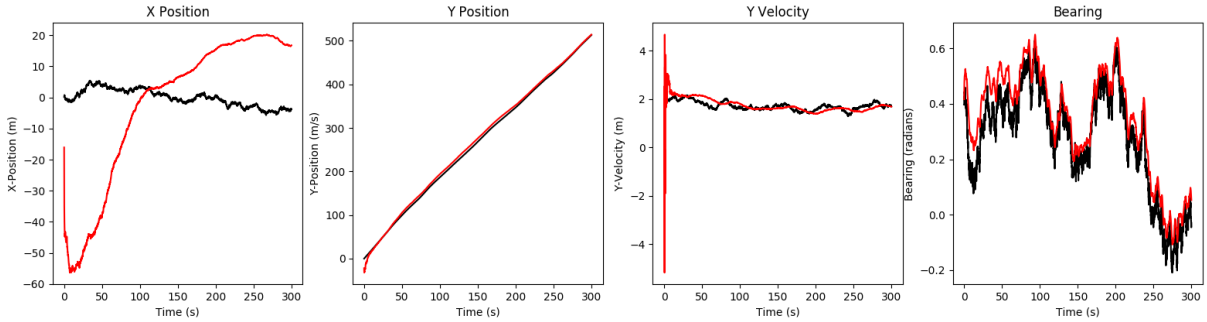

Figure 51: Plot of camera states with uncertain target positions (seed = 123, targets = 17)

Here, all the states predictions look good except for the x-position which initially sets itself at about -60m and increases throughout the whole simulation where it reaches about +20m. The ground truth track shows the camera closer to 0m the whole time. Again, the tracking point error graph (figure 52) shows signs of convergence but not centred about zero. The final time-step average x and y target error are 14.54m and 4.78m, respectively.

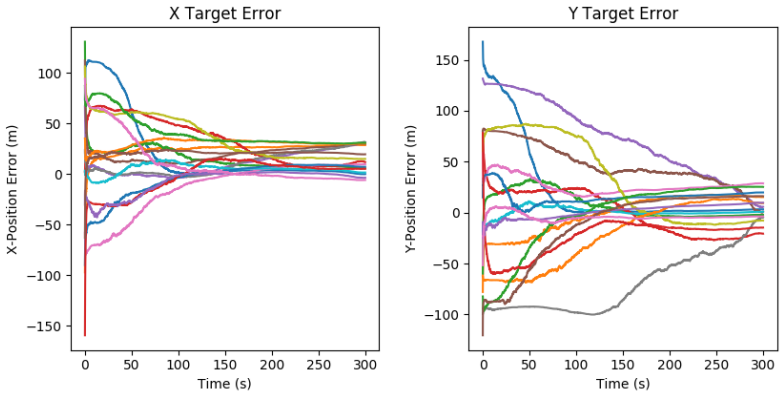

Figure 52: Error in x-position (left) and y-position (right) of targets (seed = 123, targets = 17)

Looking at the animated plot shows what is happening. Three frames from the video taken at successive points can be seen in figure 53.

In the beginning, the camera position prediction starts off to the left of the ground truth and the

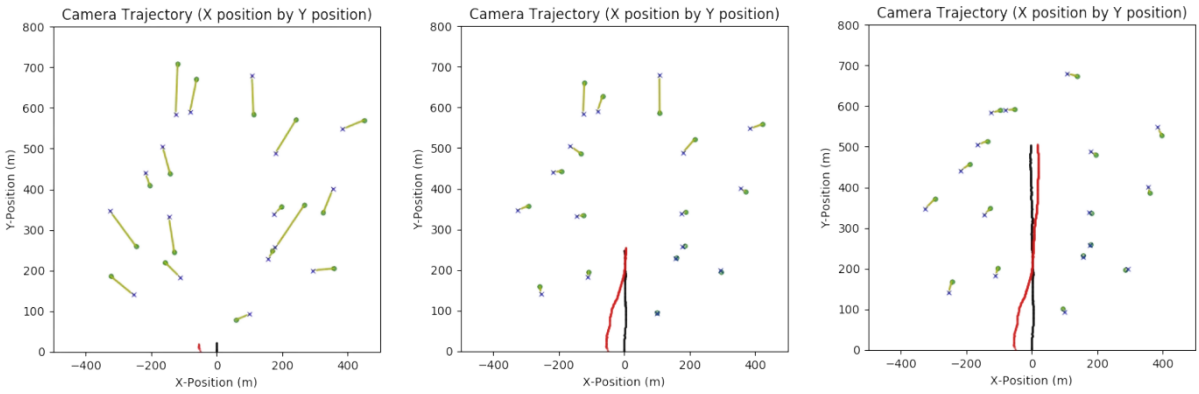

Figure 53: Animated sequence of camera and target convergence (seed = 123, targets = 17)

target predictions are randomly initialised. By the second frame, many of the points have converged and the camera prediction appears to synchronise up with the ground truth position. However, in the third frame the camera prediction overshoots the ground truth positions and continues to the right. The targets are offset in a circular ‘swirl’ pattern. The whole prediction model appears to be offset by a rotational factor.

From these two examples, it seems as though the initial corrective prediction of the camera is slightly offset in either the position or the rotation. While the angle measurements it is receiving still fulfil the mathematical equations describing it, it has found a sort of alternative solution to the problem. This makes sense as the model has no certainty in any of the states to begin with, so must rely on the initial conditions and the measurements received. Looking at a picture of the last frame in the prior example, overlaying a transparent pivoted version of all the tracking points and predicted camera track by about 5° fixes a lot of the misplaced points and the camera track looks as though it is on the right line. This overlaid image can be seen in figure 54.

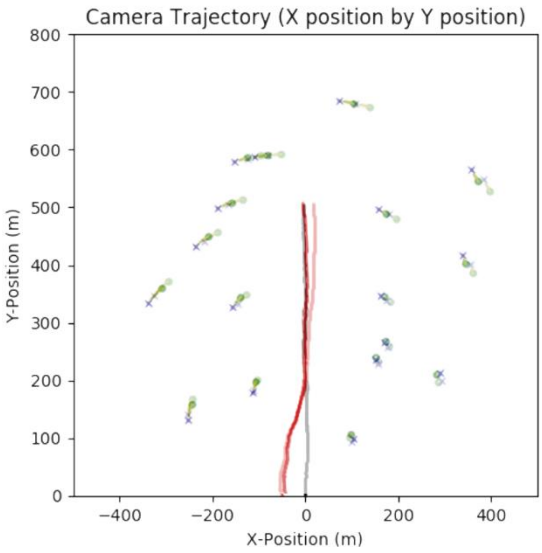

Figure 54: Transparent, rotated 'correction' of final frame of animated sequence (seed = 123, targets = 17)

To fix this problem, the camera must be calibrated to some known truths to be sure of its own state before trying to solve for all the other states as well. This is typically done at the beginning of a real-world SLAM problem [23]. To do this, a set of three points will be set up at the beginning of the simulation and initialised in the model at the exact location. The model is told there is zero uncertainty with these positions. Three points are required as the angle equation has three unknowns (x-position, y-position and bearing).

Looking at the example with random seed 0 and 40 targets again (but now with calibration points), a graph of the states of the camera can be seen in figure 55.

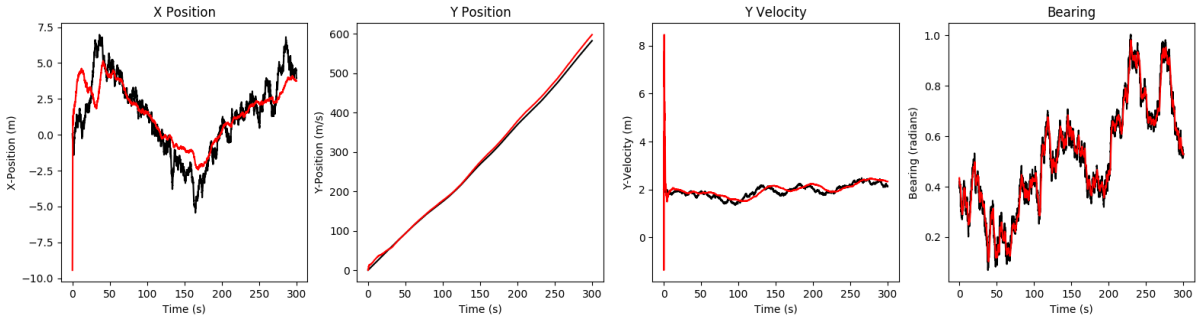

Figure 55: Plot of camera states with additional calibration targets (seed = 0, targets = 40)

Both the x-position and y-position now track much better after having been calibrated to their correct positions at the beginning. Looking at the frames of the animation in figure 56 show the same story.

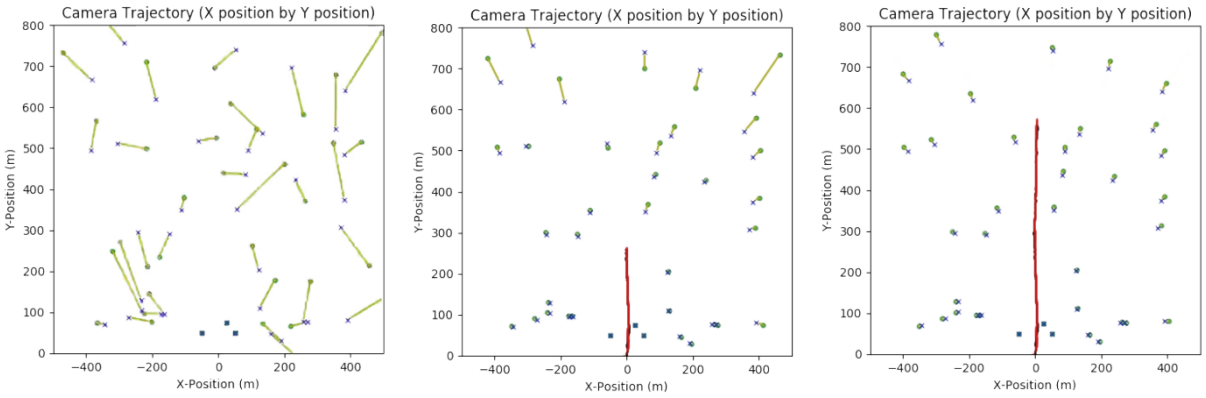

Figure 56: Animated sequence of camera and target convergence with additional calibration targets (seed = 0, targets = 40)

In the first frame, the three dots clustered around (0,0) are the calibration targets as the prediction is completely correct already as there is no uncertainty in these targets. As the camera makes its way through the field of targets the predicted positions converge quite accurately. By the final frame, most of the targets have finished converging and do not appear to display any sort of shift in one direction or rotation.

Looking at the example with random seed 123 and 17 targets again (but now with calibration points), a graph of the states of the camera can be seen in figure 57.

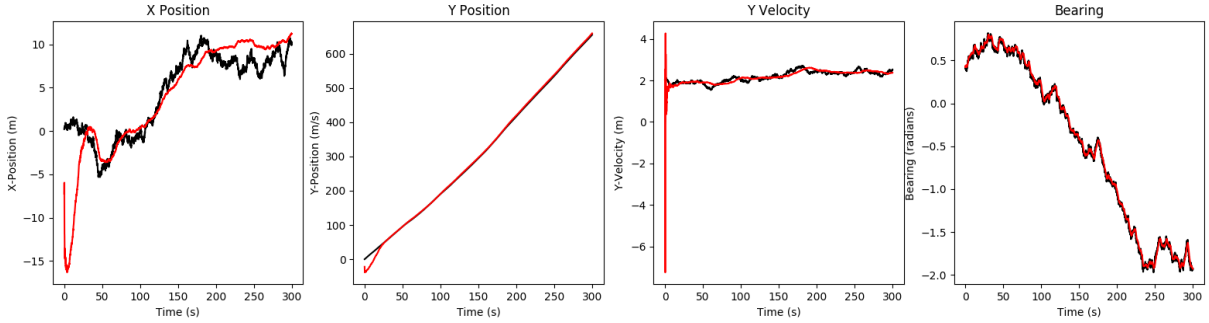

Figure 57: Plot of camera states with additional calibration targets (seed = 123, targets = 17)

The x-position track converges quickly onto the correct ground-truth path and continues to track effectively throughout. Looking at three frames from the animation in figure 58 confirms this.

In the first frame, the 3 calibration points clustered near (0,0) can be seen as they have no uncertainty in their prediction. This causes the camera prediction to quickly converge on the ground truth. Some of the other targets have already started to converge at this point. By the second frame, the camera prediction stays very close to the ground truth and the target position predictions converge quickly around the camera. By the final frame, all the target position

predictions are very close to the ground truth target positions and the camera prediction stays accurate and close to the ground truth.

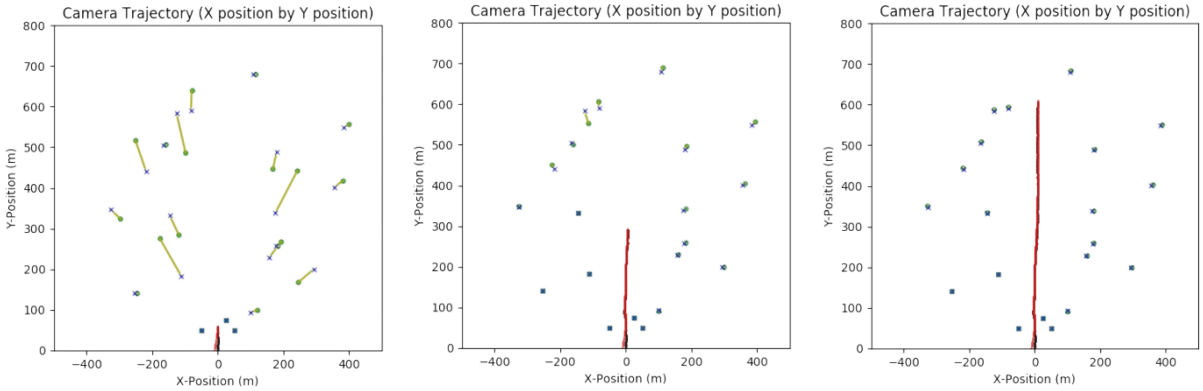

Figure 58: Animated sequence of camera and target convergence with additional calibration targets (seed = 123, targets = 17)

Looking at the covariance ellipses in figure 59 for one example (seed 123, 17 targets), for both no calibration and calibration shows how confident the model is with its prediction. On this graph, the black line is the camera ground truth and the green ellipses show at each point where the model has predicted the camera is and how certain its prediction is. The green ellipses show over what area the model is 99.7% sure the ground truth falls in. A large ellipse means the model is unsure and a small one means the model is highly certain.

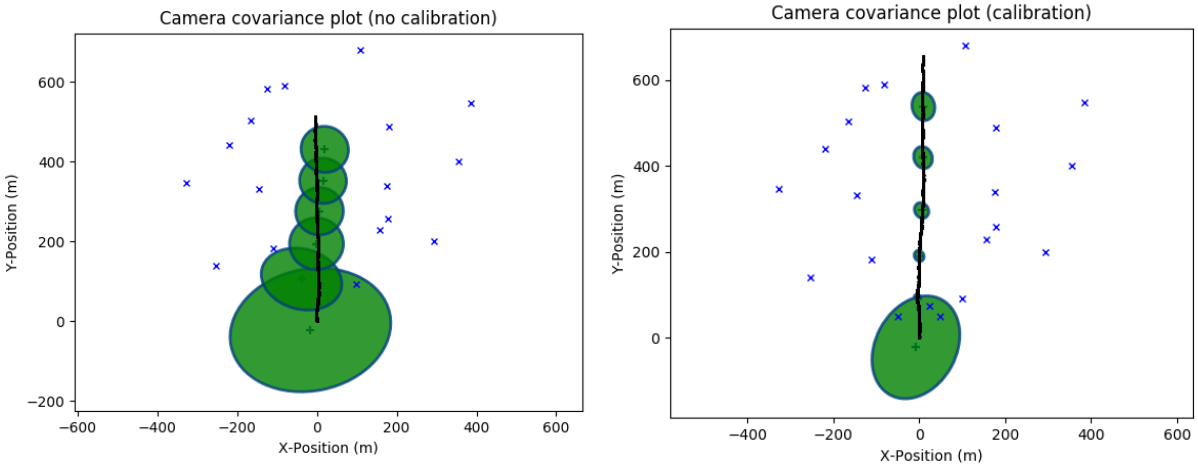

Figure 59: Covariance ellipses for camera at 99.7% certainty for no calibration versus calibration (seed = 123, targets = 17)

Looking at these graphs, the model is much more confident in its prediction having been given calibration points compared to when it has not. Interestingly, the plot with calibration shows the model is most confident having just received the known measurements from the calibration points and then the uncertainty increases slightly onwards as the camera enters new unknown

territory. However, the certainty is still far greater throughout the whole track compared to the model with no calibration.

The Python code for this 2D SLAM algorithm with calibration can be seen in Appendix 2.

A comparison of the average final state x and y target error for 5 different random seeds for 25 and 75 targets, with and without the calibration points can be seen in table 1.

Table 1: Comparison of no calibration versus calibration for a varying number of targets

		No calibration		Calibration	
Seed	No. targets	X Error (avg)	Y Error (avg)	X Error (avg)	Y Error (avg)
0	25	-39.87	-6.25	-2.93	-6.29
0	75	-4.44	-4.29	-3.79	-2.50
1	25	2.73	15.97	6.01	2.19
1	75	11.69	1.42	-2.30	0.89
2	25	-18.39	0.41	-12.78	4.60
2	75	1.14	1.43	9.77	1.90

So, the absolute average error in both x and y for no calibration is **9.00** and for calibration is **4.66**. So, calibrating the camera at the beginning of the camera move improves the accuracy of the target estimation and mapping and will subsequently improve the final predicted position of the camera. Also comparing the number of targets: the average absolute error in both x and y for 25 targets is **9.87** and for 75 targets is **3.80**. So, for these trials, increasing the number of targets increased the accuracy of the target mapping. However, it does not necessarily follow that more targets will always increase performance as Bekris, K.E. et. al. (2006) shows how the performance of SLAM with an EKF worsens in very dense scenes [26]. The paper describes how in more sparse scenes, the errors in the target positions do not affect the camera estimate so quickly as in dense scenes, allowing the positions to eventually converge.

The model is also quite robust for more erratic and noisier camera movement if the calibration is done. Increasing the x-position process noise by a factor of 10 results in a much more difficult camera path but the filter tracks it well. The camera state plot for this example (seed 11, 50 targets) can be seen in figure 60.

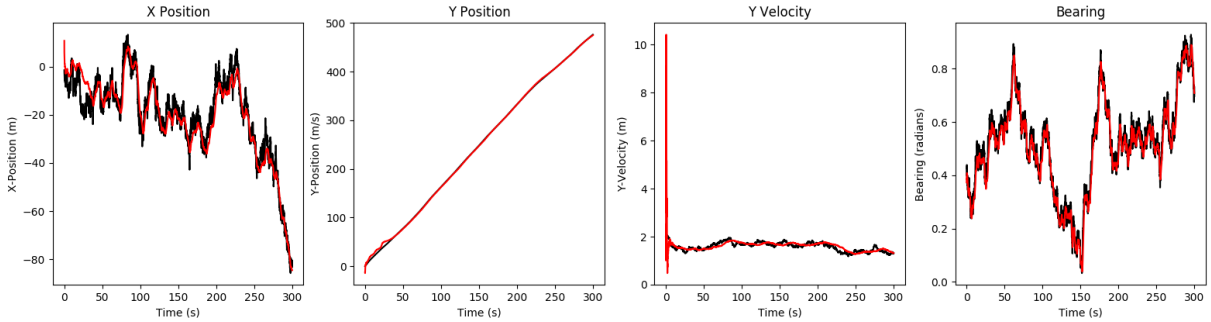

Figure 60: Plot of camera states with increased process noise (seed = 11, targets = 50)

Here all of the states track well despite the x-position moving around a lot more. The filter is performing well here as a lot of the very noisy bits in the ground truth are filtered out which gives a smoother path. Looking at the x and y target error in figure 61 shows good convergence which is centred around zero.

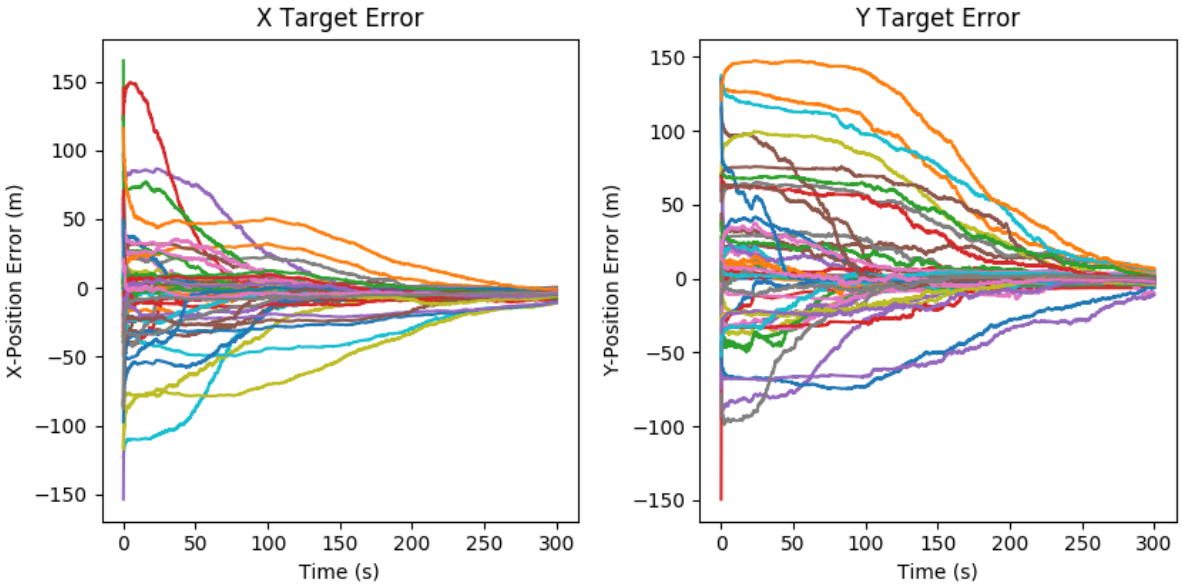

Figure 61: Error in x-position (left) and y-position (right) of targets (seed = 11, targets = 50)

Finally, looking at three frames of the animation in figure 62.

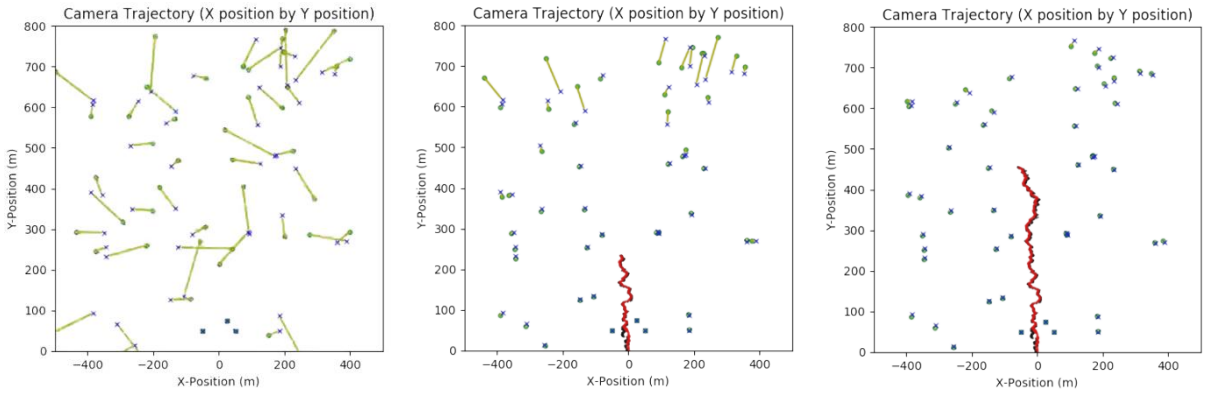

Figure 62: Animated sequence of camera and target convergence with increased process noise (seed = 11, targets = 50)

The target points begin initialised at random locations away from the true positions. The three set target points can be seen clustered around (0,0) with no uncertainty. The camera begins to move, and the prediction quickly locks onto the position and maintains the correct velocity. The target points begin to converge with the most relevant ones (the ones closest to the camera moving first). The points closest to the camera converge first as they are giving the most information through the largest angle changes. Finally, near the end of the animation, almost all the target points have converged, and the camera prediction remains accurate. The covariance ellipses in figure 63 show high confidence in the model about its prediction.

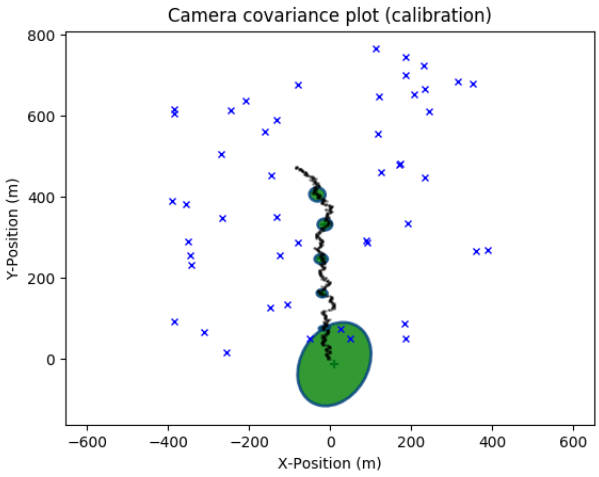

Figure 63: Covariance ellipses for camera at 99.7% certainty with increased process noise (seed = 11, targets = 50)

One problem that was encountered during experimentation was where one of the target predictions would pull away from their ground truths and converge towards a specific point. An example of this can be seen in the simulation with seed 0 with 50 targets. Looking at the state variables of the camera and the target position errors (figures 64 and 65), the filter has not tracked properly.

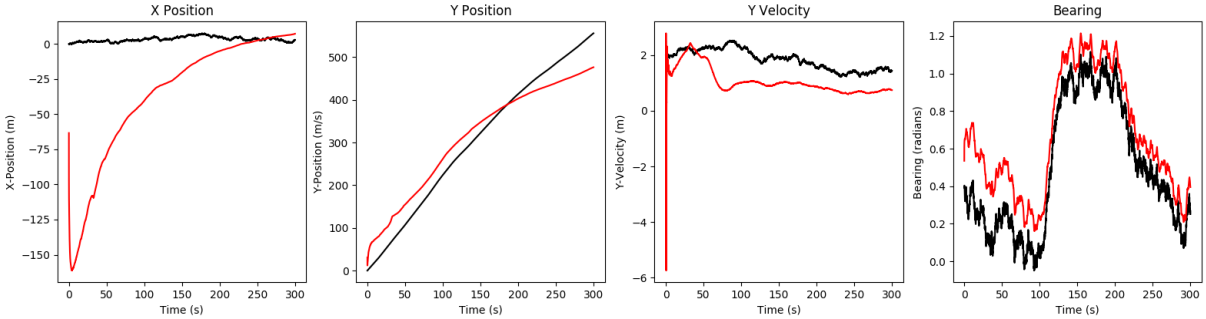

Figure 64: Plot of camera states when error has occurred (seed = 11, targets = 50)

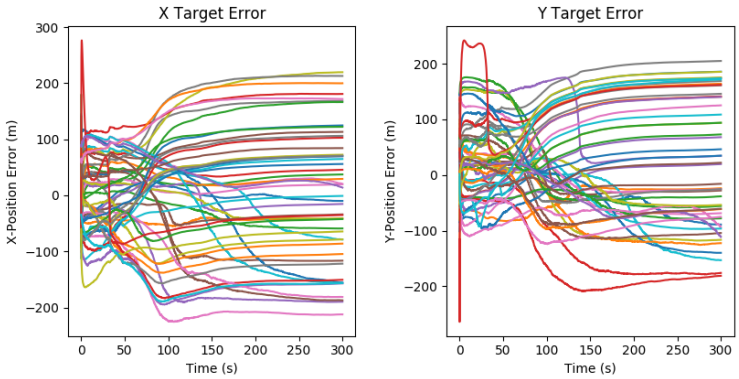

Figure 65:Error in x-position (left) and y-position (right) of targets when error has occurred (seed = 0, targets = 50)

However, looking at the animation of this simulation in figure 66, shows more clearly what is going on.

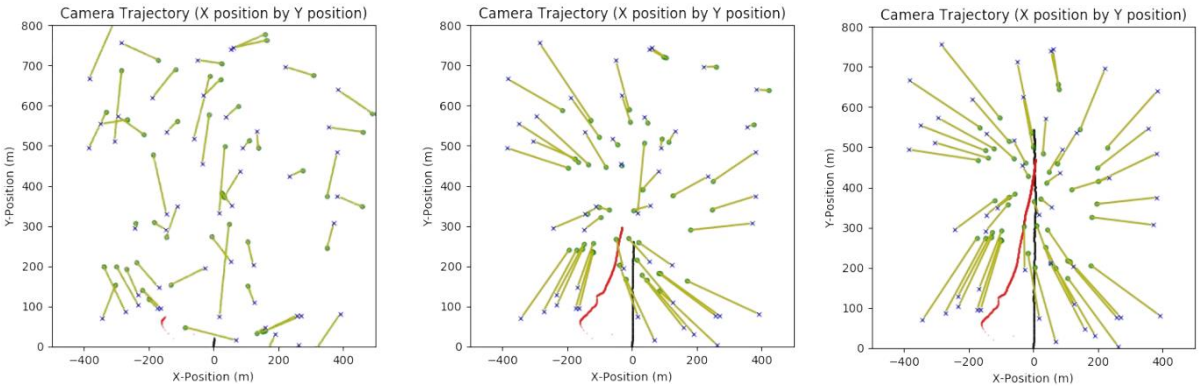

Figure 66: Animated sequence of camera and target convergence when error has occurred (seed = 0, targets = 50)

Initially, the prediction is quite far away from the ground truth and the targets are initialised far away from their ground truths. While the prediction seems to come towards the ground truth, clearly there is a problem as the target positions are getting further away from their correct positions as opposed to closer. By the final frame, all the target predictions point towards the middle and the camera prediction is lagging behind the ground truth camera significantly and not tracking properly.

Through experimentation, it seemed these scenarios only occurred when tracking points were placed too close to the camera. By sweeping out any target points that were in the middle channel (-50m to 50m) this problem never occurred. A reasonable explanation for this is that the EKF relies on each measurement having the same measurement noise throughout the simulation. However, uncertainty in either the camera position and/or the target position, if they are close, results in a much higher angular error than two points that are sufficiently far away. A demonstration of this can be seen in figure 67. Target points B and E have both only moved down by 1 but the angle change is much greater when the target is closer to the camera (points A and D).

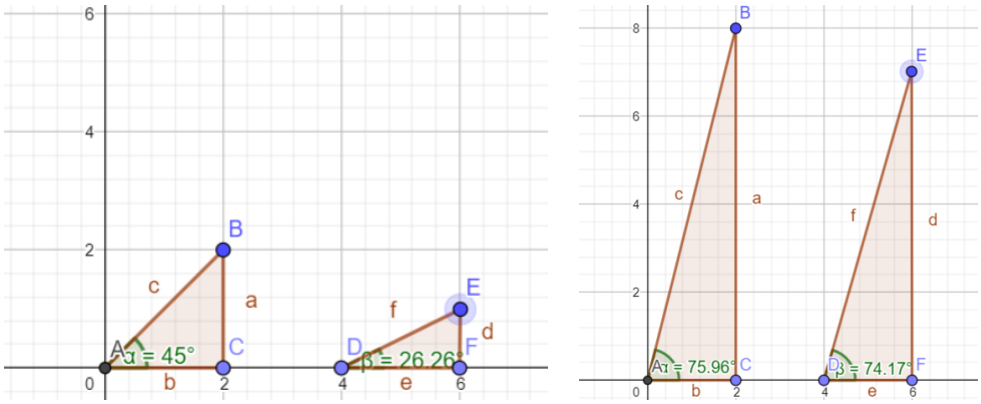

Figure 67: Demonstration of angle difference between near and far targets for equivalent shift distance

5 Conclusions

In conclusion, this study presented the SLAM problem with many potential real-world uses, broke down the problem into the key aspects and made good headway on all of them with many discoveries and conclusions drawn.

Regarding video and motion tracking, several approaches were researched, considered, and experimented with and, as a result, a robust and capable video tracking algorithm was developed. Some shortcomings of the prior literature were overcome (new targets created on-the-fly) while the limitations of the algorithm were understood (lack of feature saliency for loop-closing). The video tracking algorithm was tested on a few simple video recordings as well as a video from a SLAM dataset with successful results.

In terms of camera calibration, while, due to time constraints, an algorithm to determine camera parameters from calibration images, and then convert pixel coordinates to measurements was not created, the method and its limitations were thoroughly considered (using the pin-hole camera model projecting the 2D points along an infinite line in 3D space).

Finally, the process of state estimation in a SLAM system using an EKF was investigated through many 2D simulations on synthetic data. This meant the degree of uncertainty could be controlled to find the limits of the algorithm and understand why things went wrong when they did. The algorithm performed very well even with a high degree of uncertainty. Some interesting behaviour was observed regarding the type of measurement provided (range versus angle, atan versus atan2), and the use of graphs in conjunction with animation for evaluation made it easier to spot trends and patterns.

Because of time constraints, there was not an opportunity to bring all the parts together into one single SLAM solution, although the discoveries in each part provide significant findings to achieving the overall goal.

5.1 Recommendations for Further Work

Some suggestions for each aspect of the SLAM problem are given below:

5.1.1 Video Tracking

- Testing tracking performance by using measurements in SLAM algorithm: so far tracking was only assessed visually so testing it regarding the overall aim is a logical next step.

- Spend more time developing a SIFT or SURF approach for use on videos: while KLT works well, a SIFT or SURF approach would provide unique identification for specific features which can be beneficial in SLAM for loop-closing.

5.1.2 Camera Calibration

- Develop an algorithm for camera parameter estimation from calibration images: this is a necessary step for linking the camera tracking and EKF aspects of the problem and can be tested by comparing estimated parameters to actual parameters given in a dataset.
- Develop an algorithm for mapping 2D pixel coordinates to 3D space: again, this is important to give the EKF algorithm real-world measurements.

5.1.3 Probabilistic filtering

- Add more states to the camera allowing for more complex camera moves: adding velocity states in both x-direction and y-direction, as well as states for acceleration and angular velocity, would allow for more complex camera moves.
- Increase the dimensionality of the simulation: add a further dimension to the simulation to allow for movement in 3D space as would likely be found in a real-world camera move.
- Try a different probabilistic filter method from EKF: the EKF approach breaks down in dense spaces [26] so to address this, it could be compared to another filtering method like the UKF (Unscented Kalman Filter).
- Test filtering on measurements obtained through the video tracking algorithm: the overall project aim.
- Use additional measurements in the filter: adding additional sensors may provide information the camera cannot that improves performance.

It was not possible to add these extensions within the time available. With more time, a camera mapping algorithm, linking all aspects of the problem, would be created and then extended further. On reflection, SLAM is a complicated problem and many challenges arose throughout the development of the algorithms. While many of the challenges were overcome and important findings made towards achieving the project goal, creating a whole SLAM system from scratch (including camera tracking, camera mapping and probabilistic filtering) is very ambitious and would require more time.

6 References

1. X. Fan *et al*., 2019. "A Fire Protection Robot System Based on SLAM Localization and Fire Source Identification," *2019 IEEE 9th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, Beijing, China, pp. 555-560, doi: 10.1109/ICEIEC.2019.8784563.
2. Majdik, A.L., Till, C. and Scaramuzza, D., 2017. The Zurich urban micro aerial vehicle dataset. The International Journal of Robotics Research, 36(3), pp.269-273.
3. Cortés, S., Solin, A., Rahtu, E. and Kannala, J., 2018. ADVIO: An authentic dataset for visual-inertial odometry. In Proceedings of the European Conference on Computer Vision (ECCV) (pp. 419-434).
4. Lowe, D.G., 2004. Distinctive image features from scale-invariant keypoints. International journal of computer vision, 60(2), pp.91-110.
5. Weitz, Prof. Dr. E., 2016. SIFT - Scale-Invariant Feature Transform. [online] Weitz.de. Available at: http://weitz.de/sift/index.html [Accessed 7 August 2020].
6. Docs.opencv.org. 2020. Opencv: Introduction To SIFT (Scale-Invariant Feature Transform). [online] Available at: https://docs.opencv.org/4.4.0/da/df5/tutorial_py_sift_intro.html [Accessed 8 August 2020].
7. Bhalerao, A., n.d. CS413 Image and Video Analysis Lecture Series. Hierarchical Computations [online] Available at: https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs413/ia_-_5.pdf [Accessed 7 August 2020].
8. Ai.stanford.edu. n.d. Tutorial 2: Image Matching. [online] Available at: https://ai.stanford.edu/~syyeung/cvweb/tutorial2.html [Accessed 13 August 2020].
9. Labbe, Roger. 2014. Kalman and Bayesian Filters in Python. GitHub Repository. [online] Available at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python [Accessed 25th June 2020].
10. Bay, Herbert & Tuytelaars, Tinne & Van Gool, Luc, 2006. SURF: Speeded up robust features. Computer Vision-ECCV 2006. 3951. 404-417. 10.1007/11744023_32.
11. Docs.opencv.org. 2014. Introduction To SURF (Speeded-Up Robust Features) — Opencv 3.0.0-Dev Documentation. [online] Available at: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html [Accessed 8 August 2020].

12. Mistry, Dr & Banerjee, Asim, 2017. Comparison of Feature Detection and Matching Approaches: SIFT and SURF. GRD Journals- Global Research and Development Journal for Engineering. 2. 7-13.
13. Jianbo Shi and Tomasi, "Good features to track," 1994 1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, Seattle, WA, USA, pp. 593-600, doi: 10.1109/CVPR.1994.323794.
14. Mordvintsev, A. and K, A., 2013. Shi-Tomasi Corner Detector & Good Features To Track — Opencv-Python Tutorials 1 Documentation. [online] Opencv-python-tutroals.readthedocs.io. Available at: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_shi_tomasi/py_shi_tomasi.html [Accessed 8 August 2020].
15. Lucas, B.D. and Kanade, T., 1981. An iterative image registration technique with an application to stereo vision.
16. Kitani, K., n.d. Lucas-Kanade Optical Flow. Computer Vision 16-385, Carnegie Mellon University. [online] Available at: http://www.cs.cmu.edu/~16385/s15/lectures/Lecture21.pdf [Accessed 7 August 2020].
17. Rojas, R., 2010. Lucas-kanade in a nutshell. Freie Universit at Berlinn, Dept. of Computer Science, Tech. Rep. Available at: http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/Lucas-Kanade2.pdf
18. Mordvintsev, A. and K, A., 2013. Optical Flow — Opencv-Python Tutorials 1 Documentation. [online] Opencv-python-tutroals.readthedocs.io. Available at: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html [Accessed 8 August 2020].
19. Newman, P. and Ho, K., 2005, April. SLAM-loop closing with visually salient features. In proceedings of the 2005 IEEE International Conference on Robotics and Automation (pp. 635-642). IEEE.
20. C. Yang, L. Wanyu, Z. Yanli and L. Hong., 2016. "The research of video tracking based on improved SIFT algorithm," 2016 IEEE International Conference on Mechatronics and Automation, Harbin, pp. 1703-1707, doi: 10.1109/ICMA.2016.7558820.
21. van den Boomgaard, R., 2017. 1.2. The Pinhole Camera Matrix — Image Processing And Computer Vision 2.0 Documentation. [online] Staff.fnwi.uva.nl. Available at: https://staff.fnwi.uva.nl/r.vandenboomgaard/IPCV20172018/LectureNotes/CV/PinholeCamera/PinholeCamera.html [Accessed 10 August 2020].

22. Bhalerao, A., n.d. CS413 Image and Video Analysis Lecture Series. View Geometry and Camera Calibration [online] Available at: https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs413/va_-_3.pdf [Accessed 7 August 2020].
23. Davison, A.J., Reid, I.D., Molton, N.D. and Stasse, O., 2007. MonoSLAM: Real-time single camera SLAM. IEEE transactions on pattern analysis and machine intelligence, 29(6), pp.1052-1067.
24. Kim, Y. and Bang, H., 2019. Introduction to Kalman Filter and Its Applications. Introduction and Implementations of the Kalman Filter, IntechOpen, 2019. Chap. 2. doi: 10.5772/intechopen.80600. Available at: https://doi.org/10.5772/intechopen.80600
25. Bay, H., Tuytelaars, T. and Van Gool, L., 2006, May. Surf: Speeded up robust features. In European conference on computer vision (pp. 404-417). Springer, Berlin, Heidelberg.
26. Bekris, K.E., Glick, M. and Kavraki, L.E., 2006, May. Evaluation of algorithms for bearing-only SLAM. In Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006. (pp. 1937-1943). IEEE.

7 Appendices

7.1 Appendix 1 (*optical_flow.py*)

```
1.  import skvideo.io
2.  import cv2
3.  import numpy as np
4.  import math
5.  import time
6.
7.  # Create new feature list based on old and new points
8.  def combineTrackers(oldPoints, newPoints, status):
9.      threshold = 5 # Set proximity threshold
10.     extraFeatures = []
11.     for i in range(len(newPoints)):
12.         new = True # Assume point is new
13.         for j in range(len(oldPoints)): # Check proximity to all old points
14.             if status[j] == 1: # If point is still in frame
15.                 if math.hypot(newPoints[i][0]-oldPoints[j][0], newPoints[i][1]-
    oldPoints[j][1]) < threshold:
16.                     new = False # If too close, then not new
17.         if new == True:
18.             extraFeatures += [newPoints[i].tolist()] # Otherwise add to key point lis
    t
19.     if len(extraFeatures) != 0:
20.         featureList = np.concatenate((oldPoints, np.asarray(extraFeatures)))
21.     else:
22.         featureList = oldPoints
23.     return featureList
24.
25. video = skvideo.io.vread('video.mp4') # Import video
26. t0 = time.time() # Start timer
27. # Parameters for Shi-Tomasi corner detection
```

```
feature_params = dict(maxCorners = 200, qualityLevel = 0.3, minDistance = 2, blockSize = 7)
# Parameters for Lucas-Kanade optical flow
lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))
# Variable for color to draw optical flow track
colourList = []
# Create random colour list for points
for i in range(5000):
    colourList += [np.random.random(size=3) * 256]

video_bw = video[:, :, :, 0].astype('uint8') # Select one colour channel
prev_img = video_bw[0]
prev_feat = cv2.goodFeaturesToTrack(prev_img, mask = None, **feature_params)
mask = np.zeros_like(video[0])

writer = skvideo.io.FFmpegWriter("tracked_video.mp4") # Prepare to write video

for i in range(len(video)):
    frame = video[i]
    frame_bw = video_bw[i]

    # Run optical flow algorithm
    nxt, status, error = cv2.calcOpticalFlowPyrLK(prev_img, frame_bw, prev_feat, None, **lk_params)
    # Selects good feature points for previous position
    good_old = prev_feat[:, 0, :]
    # Selects good feature points for next position
    good_new = nxt[:, 0, :]

    for j in range(len(good_new)):
        new_coord = tuple(good_new[j])
        old_coord = tuple(good_old[j])
        mask = cv2.line(mask, new_coord, old_coord, (0,255,0), 2) # Draw tracking line in green
        frame = cv2.circle(frame, new_coord, 3, colourList[j], -1) # Draw tracking point

    # On every 20th frame find new tracking points and merge
    if (i%20 == 0) and (i!=0):
        new_trackers = cv2.goodFeaturesToTrack(prev_img, mask = None, **feature_params)
        good_new = combineTrackers(np.rint(good_new), new_trackers[:, 0, :], status)

        good_new = np.float32(good_new)

    output = cv2.add(frame, mask)
    cv2.imshow("sparse optical flow", output) # Show output video
    cv2.waitKey(1)

    prev_feat = good_new.reshape(-1, 1, 2)
    prev_img = frame_bw.copy()
    writer.writeFrame(output) # Save video frame
t1 = time.time()
print(t1-t0) # Print operation time
writer.close()
```

7.2 Appendix 2 (*2D_SLAM_sim.py*)

```
from filterpy.kalman import ExtendedKalmanFilter
from numpy import eye, array, asarray
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from numpy.random import randn, randint, uniform
import numpy.random
```

```
import numpy as np
import math
from filterpy.stats import plot_covariance_ellipse

seed = 0 # Set random seed
numpy.random.seed(seed)

nT = 50 # Ideal number of targets

# Initialise camera position
x_pos = 0
y_pos = 0
y_vel = 2
bearing = 0.4

# Initialise target positions
targets = []
for i in range(nT):
    point = [uniform(-400,400), uniform(0, 800)]
    if ((point[0] < -
75) or (point[0] > 75)): # Remove targets that are too close to camera path
        targets.append(point)

# Add in specific calibration targets
targets.insert(0,[50,50])
targets.insert(0,[-50,50])
targets.insert(0,[25,75])
nT = len(targets) # Update number of targets with removed targets

def XJacobian(x):
    # At state x return Jacobian matrix
    xp = float(x[0]) # x-position
    yp = float(x[1]) # y-position
    vp   = float(x[2]) # y-velocity
    bear = float(x[3]) # bearing

    nS = len(x) # Number of states
    nT = (len(x)-4)/2 # Number of targets
    out = np.zeros((int(nT), nS)) # Jacobian output array shape
    for i in range(int(nT)):
        tIndx = i*2+4 # First target element
        element = np.zeros(nS)

        # Calculate partial derivatives relevant to target positions
        targets_partial_deriv = array([(-float(x[tIndx+1]) + yp)/
                                       ((float(x[tIndx]) - xp)**2 + (float(x[tIndx+1])
                                       - yp)**2), (float(x[tIndx]) - xp)/
                                       ((float(x[tIndx]) - xp)**2 + (float(x[tIndx+1])
                                       - yp)**2)])

        element[tIndx:tIndx+2] = targets_partial_deriv

        # Calculate partial derivatives relevant to camera states
        element[0:4] = array ([-(-float(x[tIndx+1]) + yp)/
                                      ((float(x[tIndx])- xp)**2 + (float(x[tIndx+1])
                                       - yp)**2), -(float(x[tIndx]) - xp)/
                                      ((float(x[tIndx]) - xp)**2 + (float(x[tIndx+1])
                                       - yp)**2), 0, 1])
        out[i] = element
    return out

def hx(x):
    # Measurement expected for state x
    xp = float(x[0])
    yp = float(x[1])
    vp   = float(x[2])
```

```
72.         bear = float(x[3])
73.
74.         nT = (len(x)-4)/2 # Number of targets
75.         out = []
76.
77.         for i in range(int(nT)):
78.             tIndx = i*2+4 # First target element
79.             element = [math.atan2((float(x[tIndx+1] - yp)), (float(x[tIndx]) - xp))+bear]

80.             out.append(element)
81.         out = asarray(out)
82.         return out
83.
84.     def residual(a, b): # Fix residual for angles
85.         y = a - b # calculate subtraction residual
86.         y = y % (2 * np.pi)    # move to 0 -> 2*pi
87.         y[y > np.pi] -= 2 * np.pi # rescale to -pi -> +pi
88.         return y
89.
90.     def update_camera(dt): # Call to update camera and return measurements
91.         global x_pos, y_pos, y_vel, bearing, targets
92.         # Add process noise to each state
93.         x_pos = x_pos  + .1 * randn()
94.         y_vel = y_vel + .01 * randn()
95.         bearing = bearing + .01 * randn()
96.         y_pos = y_pos + y_vel * dt
97.
98.         measurements = []
99.         # Calculate measurement to each target with measurement error
100.        for T in targets:
101.            err = .05 * randn()
102.            ang = math.atan2((T[1]- y_pos), (T[0] - x_pos)) + bearing
103.            measurements.append([ang + err])
104.        measurements = asarray(measurements)
105.
106.        return measurements
107.
108.    dt = 0.05 # Time step
109.
110.    ekf = ExtendedKalmanFilter(dim_x = 4 + 2 * nT, dim_z = nT) # Create EKF
111.
112.    np_targets = asarray(targets) + 75 * randn(nT, 2) # Create target array with incorrec
        t guess
113.    np_targets[0:3] = array([[25,75], [-50,50], [50,50]]) # Add in calibration targets
114.    ekf.x = array([x_pos + 5, y_pos + 7, y_vel + .5, bearing + 1.5]) # Make incorrect gue
        ss for camera states
115.    ekf.x = np.append(ekf.x, np_targets) # Append camera states and target states
116.    ekf.x = np.reshape(ekf.x, (1, 4 + 2 * nT)).T
117.
118.    # Create state transition matrix
119.    ekf.F = eye(4 + 2 * nT)
120.    ekf.F[1,2] = dt
121.
122.    range_var = 0.05 # Uncertainty of angle measurements
123.    ekf.R = eye(nT) * range_var
124.    ekf.Q = np.eye(4 + 2 * nT) * 1.000e-7 # Target position process covariance
125.    ekf.Q[0,0], ekf.Q[1,1], ekf.Q[2,2], ekf.Q[3,3] = 1.000e-03, 1.0000e-06, 1.0000e-
        04, 1.0000e-5 # Process noise uncertainty
126.    ekf.P *= 1500 # Estimated accuracy of state estimate
127.    ekf.Q[4,4], ekf.Q[5,5], ekf.Q[6,6] = 1.0000e-10, 1.0000e-10, 1.0000e-
        10 # Calibration target uncertainties
128.    ekf.P[4,4], ekf.P[5,5], ekf.P[6,6] = 0, 0, 0 # Add in initial calibration target unce
        rtainties
129.
130.    # Arrays to track results
131.    cameraTrk, predictionTrk, updateCovarTrk = [], [], []
```

```
MAng, TargetX, TargetX_Error = [], [], []
for i in range(int(300/dt)): # Set run time
    z = update_camera(dt) # Update camera
    cameraTrk.append((x_pos, y_pos, y_vel, bearing))
    MAng.append(z[0][0])

    TargetX.append((ekf.x[4:]))
    TargetX_Error.append(ekf.x[4:].flatten()-asarray(targets).flatten())

    # Update EKF with measurement, Jacobian function and function to find expected measurement
    ekf.update(z, XJacobian, hx, residual = residual) # Update predictions with measurements

    predictionTrk.append((ekf.x[0], ekf.x[1], ekf.x[2], ekf.x[3]))
    updateCovarTrk.append(ekf.P)

    ekf.predict() # Predict next state


## PLOT IT
cameraTrk = asarray(cameraTrk)
predictionTrk = asarray(predictionTrk)
updateCovarTrk = asarray(updateCovarTrk)

MAng = asarray(MAng)
TargetX = asarray(TargetX)
TargetX_Error = asarray(TargetX_Error)

xaxis = np.arange(0, 300, 0.05)
titles = ['X Position', 'Y Position', 'Y Velocity']
ylabels = ['X-Position (m)', 'Y-Position (m/s)', 'Y-Velocity (m)', 'Bearing (radians)']
fig, axs = plt.subplots(2,4, figsize = (50,5))
for plot in range(3):
    axs[0, plot].set_title(titles[plot])
    axs[0, plot].set_xlabel('Time (s)')
    axs[0, plot].set_ylabel(ylabels[plot])
    axs[0, plot].plot(xaxis, cameraTrk[:,plot], 'k')
    axs[0, plot].plot(xaxis, predictionTrk[:,plot], 'r')

axs[0, 3].set_title('Bearing')
axs[0, 3].set_xlabel('Time (s)')
axs[0, 3].set_ylabel('Bearing (radians)')
axs[0, 3].plot(xaxis, cameraTrk[:,3], 'k')
axs[0, 3].plot(xaxis, predictionTrk[:,3], 'r')

axs[1,0].set_title('Camera Trajectory (X position by Y position)')
axs[1, 0].set_xlabel('X-Position (m)')
axs[1, 0].set_ylabel('Y-Position (m)')
axs[1,0].plot(cameraTrk[:,0], cameraTrk[:,1], 'ko', markersize=1, markevery=5, alpha = .9)
axs[1,0].plot(predictionTrk[:,0], predictionTrk[:,1], 'ro', markersize=1, markevery=5, alpha = .9)

for i in range(nT):
    axs[1,0].plot(targets[i][0], targets[i][1], 'bx', markersize = 4, alpha = .7)

axs[1, 1].set_title('X Target Error')
axs[1, 2].set_title('Y Target Error')
axs[1, 3].set_title('X Target Locations')
axs[1, 1].set_xlabel('Time (s)')
axs[1, 2].set_xlabel('Time (s)')
axs[1, 1].set_ylabel('X-Position Error (m)')
axs[1, 2].set_ylabel('Y-Position Error (m)')

```

```
for i in range(0, nT*2, 2):
    axs[1,1].plot(xaxis, TargetX_Error[:,i])
    axs[1,2].plot(xaxis, TargetX_Error[:,i+1])

## PLOT COVARIANCE ELLIPSES

cov_fig = plt.figure()
plt.plot(cameraTrk[:, 0], cameraTrk[:, 1], 'ko', markersize=1, markevery= 10, alpha = 0.3)
plt.title('Camera covariance plot (calibration)')
plt.xlabel('X-Position (m)')
plt.ylabel('Y-Position (m)')
for i in range(nT):
    plt.plot(targets[i][0], targets[i][1], 'bx', markersize = 4)
for i in range(0, predictionTrk.shape[0], 1000):
    plot_covariance_ellipse(
                        (predictionTrk[i, 0], predictionTrk[i, 1]), updateCovarTrk[i, 0:2, 0:2],
                        std=6, facecolor='g', alpha=0.8)

### ANIMATE SIMULATION

TargetX = np.reshape(TargetX, (TargetX.shape[0], nT, 2))[0::10]
cameraTrk = cameraTrk[0::10]
predictionTrk = predictionTrk[0::10]

fig_anim = plt.figure()
fig_anim.set_size_inches(5, 5, True)
ax_anim = plt.axes(xlim=(-500, 500), ylim=(0, 800))
ax_anim.set_title('Camera Trajectory (X position by Y position)')
ax_anim.set_xlabel('X-Position (m)')
ax_anim.set_ylabel('Y-Position (m)')

ground_truth, = ax_anim.plot(cameraTrk[0][0], cameraTrk[0][1], 'ko', markersize=1)
kalman_track, = ax_anim.plot(predictionTrk[0][0], predictionTrk[0][1], 'ro', markersize=1)
target_pred, = ax_anim.plot([], [], 'go', markersize = 4, alpha = .7)
lines = [plt.plot([], [], 'yo-', animated=True, markersize = 0)[0] for _ in range(nT)]
for i in range(nT):
    ax_anim.plot(targets[i][0], targets[i][1], 'bx', markersize = 4, alpha = .7)

def animate(i):
    ground_truth.set_alpha(0.4)
    kalman_track.set_alpha(0.4)
    ground_truth.set_data(cameraTrk[:i:,0], cameraTrk[:i:,1])
    kalman_track.set_data(predictionTrk[:i:,0], predictionTrk[:i:,1])
    target_pred.set_data(TargetX[i,:,0], TargetX[i,:,1])
    for j in range(nT):
        lines[j].set_data([TargetX[i,j,0], targets[j][0]], [TargetX[i,j,1], targets[j][1]])

anim = FuncAnimation(
    fig_anim, animate, interval=0, frames=cameraTrk.shape[0]-1)

path = 'D:\\Documents\\University\\MSc Project\\main\\Kalman\\videos\\' # Path to save animation video
anim.save(path + 'Seed {} ({} targets).mp4'.format(seed, nT), fps=60, bitrate = 1000)

plt.close()

print('Video Outputted.')
plt.show()
```