Pixel-Based Object Motion Detection and Tracking with a Moving Camera

by

Rishi Sundaresan

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author . . . .
Department of Electrical Engineering and Computer Science
May 12, 2020

Certified by . . . .
Jonathan P. How
Richard C. Maclaurin Professor of Aeronautics and Astronautics, MIT
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Pixel-Based Object Motion Detection and Tracking with a Moving Camera

by

Rishi Sundaresan

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Object motion detection and tracking is a major research focus in visual autonomous navigation. In the case of a moving camera, pixel-based approaches to detecting motion are difficult as the algorithm must be able to distinguish between pixel motion caused by ego motion and pixel motion caused by object motion in the environment. In this study, we propose a low-computation pixel-based method to detect and track moving objects in scenes with a moving camera. Our method calculates deviations in pixel values over a stream of RGB images and thresholds the deviations to identify motion. Most importantly, the method accounts for ego motion by using depth information and camera poses to determine the best pixel locations to compare across images. We experiment with two different methods of calculating deviation: standard deviation and Gaussian distribution percentile. Our proposed method is evaluated on a dataset from the AirSim Safari Environment and shows the ability to detect and track *only* the moving objects in scenes.

Thesis Supervisor: Jonathan P. How
Title: Richard C. Maclaurin Professor of Aeronautics and Astronautics, MIT

Acknowledgments

I would like to acknowledge Prof. Jonathan How and Mr. Michael Everett for their guidance throughout the project. They have been instrumental in my intellectual and personal growth. I would also like to acknowledge the Aerospace Controls Laboratory at MIT for their support and resources throughout the study. I also thank my mother and father, Latha and Sivakumar Sundaresan, and my brother Tejas Sundaresan, for their constant guidance throughout my studies.

I would also like to acknowledge the communities I have been a part of at MIT for their support over my degree. Zeta Beta Tau Fraternity and the MIT Bhangra team have been reliable support networks, aiding my studies in numerous ways. This work is supported by Ford Motor Company.

Contents

List of Figures

Chapter 1

Introduction

Autonomous systems including self-driving cars have started to gain popularity as recent advances in machine learning have furthered the capabilities of intelligent decision-making. These systems rely on a vast range of sensor suites to gain information about their environments. However, many of these sensors come at high cost. Most self-driving cars use LiDAR input as the primary means to map environments, and early models of LiDAR cost upwards of $75K [1]. In the application of ground robots in indoor and pedestrian-rich environments, scaling the production of the robots may be infeasible with such a large cost for sensors.

Relying on LiDAR or similar range-finding mechanisms also has other pitfalls: they are extremely sensitive to noise in the environment. Raindrops and fog can very easily introduce noisy close readings, throwing off mapping estimates, while cameras have the potential to gain a more semantic understanding of rain and fog [2].

For these, and other reasons, low-cost vision-based navigation systems are growing in popularity in the field of autonomy. These systems rely on input from an RGB(D) (or grayscale) camera as well as other low-cost sensors (i.e. inertial measurement units, wheel encoders) to perform localization, perception, and planning [3].

This research project is part of a larger study to build a fully-autonomous low-cost visual navigation stack. This system is meant to operate on a small ground robot in a pedestrian-

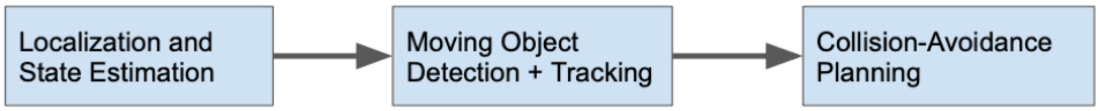

Localization and State Estimation
Moving Object Detection + Tracking
Collision-Avoidance Planning

Figure 1-1: This flow represents the three main modules of the vision navigation stack. The robot uses vision-based localization techniques to estimate its current state (heading, position, velocity, acceleration). Using its current state, the robot localizes moving pedestrians and objects in the environment and tracks them. Finally, using external object motion, the robot can path plan to avoid the moving objects and reach its goal.

rich campus environment (possibly indoors), with a driving speed of 1-2 m/s. With this application, the pipeline can be broken down into three main units that operate entirely on low-cost visual sensors (i.e. RGBD cameras) and IMU/wheel encoders, as detailed in Figure 1-1. The three units comprise the core of many autonomy stacks: determining state, observing moving objects in environment, and path planning to goal while avoiding those objects.

The vast majority of the project focuses on the second step of the pipeline, or more specifically, achieving moving object detection and tracking through low-compute pixel-based methods. This research project also experiments with achieving vision-based localization through applying VINS-Fusion, an extension of a vision-based state estimation system VINS-Mono [4].

The specific emphasis of this project is object movement detection and tracking *with a moving camera*. We define *movement detection* as identifying the moving object in a scene, and we define *tracking* as locating the moving object in the scene over multiple images and creating a trajectory. We denote the *static camera problem* as moving object detection and tracking with non-moving camera. We denote the *moving camera problem* as moving object detection and tracking with a moving camera, where the algorithm must be able to distinguish the object moving from all other motion in the pixel frame (see Section 3.1 for definition) caused by the camera's motion.

We will first introduce methods for measuring pixel deviation as a way of detecting motion in the context of the static camera problem (Section 3.2), and then show how we extend those methods to solve the moving camera problem (Section 4). For the moving camera problem, we will show the ability to detect and track object motion.

The main contribution of this thesis is proposing a method to accurately detect and track moving animals in an AirSim Safari Simulation Environment with a moving camera using low-computation pixel-based methods.

Now that we have introduced the problem and defined its scope, Chapter 2 will describe related work in the field, as well as related work from this research project in state estimation.

Chapter 2

Related Work

2.1 Related Work in Literature

This section will describe related work in literature on the three main steps of the visual navigation pipeline in Figure 1-1.

2.1.1 Vision-Based State Estimation

2.1.1.1 Monocular Vision

Monocular vision navigation uses a single camera to localize a moving robot in environments relative to the staring position. While it may be one of the most inexpensive solutions, monocular vision techniques have difficulty with providing position and distance measurements to objects, since without sophisticated deep learning methods it is difficult to measure depth without multiple locations of sensor input. Monocular methods can also easily be disturbed by sudden movements [5]. However, many monocular solutions do exist, including ORB-SLAM, which uses a FAST feature detector for corner/edge detection [6], bundle adjustment, and loop closure across frames to outperform other monocular SLAM methods including PTAM and LSD-SLAM [7, 8].

2.1.1.2 Stereo Vision

In stereo vision techniques, two cameras installed on the robot capture images in the environment. This system is usually more capable of estimating distance to objects from small differences where landmarks are located in the image plane between the cameras. We will be using a stereo vision system on our robot due to this ability.

2.1.2 Moving Object Detection

Deep-Learning based approaches can detect objects in images through object detectors [9]. Coupled with LiDAR or depth input, these approaches can track objects over time [10]. In pixel-based approaches, one of the major difficulties with moving object detection involves when there is ego motion with the camera (i.e. a moving camera). Stationary objects appear moving, and moving objects could appear stationary. Thus, it is vital to be able to subtract background motion purely caused by camera motion, a task that has many challenges. Toyama et. al. described a lot of these challenges, including foreground objects camouflaging with the background, low frame rates, and motion blur [11].

Previous methods for moving-camera object motion detection include classifying each pixel into three categories (planar background, parallax, or motion regions) by sequentially applying 2D planar homographies [12], developing non-panoramic background models [13], and classifying feature points into the foreground and background. [14]. Optical flow has also been used as a method to solve the moving camera problem in cases where the dense stereo and ego motion estimates are available [15]. Gaussian Models based on spatial distribution have also been used in attempting to subtract background pixel motion from images [16]. Filtering techniques including morphological filtering are commonly used to pre-process images before these algorithms are used [17]. These pixel-based methods all attempt to solve the difficult problem of removing background motion from a moving camera and only detecting actual object motion, and our method falls into this class of methods.

2.1.3 Collision-Avoidance Motion Planning

Path planning is a key module in navigation systems that uses estimated states and environment information to provide the robot path(s) to follow that usually optimize a cost function.

1. Collision Avoidance using Deep Reinforcement Learning (CADRL) frames collision avoidance as a sequential decision-making problem in an RL framework and then uses policy-based learning for path planning. An initial implementation of the algorithm performed well at human walking speed (1.2 m/s) on a small robot in an indoor pedestrian environment, but more accurate vehicle dynamics models are needed to apply it to faster speeds [18].

2. Diffusion Map algorithms learn the map's geometry to parameterize feasible paths in a memory efficient way [19]. The diffusion map algorithm currently employed by the lab efficiently computes cost-to-go for planners, but needs to be extended to handle dynamic environments.

These algorithms do not directly use visual input, but need to be modified to incorporate outputs of vision-based object tracking and state estimation systems.

2.2 Summary of Work in Vision-Based State Estimation

When applied in a real-life system, our pixel-based motion detection methods will rely on accurate state estimation from visual features. In this section we briefly summarize the portion of this research project that focuses on accurately determining ego pose from an RGB-infrared camera and IMU.

2.2.1 Method

We define the setup as follows: A RealSense D435i camera moves around a large open space, collecting infrared images and IMU data.

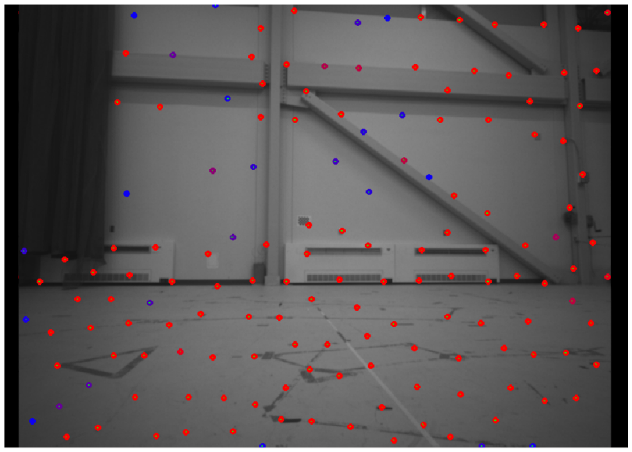


Figure 2-1: Example of infrared image that is an input to VINS. The blue dots indicate features that are moving (in the pixel frame) or new features (have not been detected before). The red dots indicate features that the algorithm believes to be stationary in the pixel frame. Note how many of the dots correspond with edges in the image

As inputs, we used the following method:

1. Timestamped infrared images (used rather than RGB to reduce lighting noise). Figure 2-1 shows an example of an infrared image, with corresponding red and blue dots that represent visual features tracked by the algorithm.

2. Timestamped IMU accelerometer data

Camera intrinsic and extrinsic parameters were determined from RealSense Documentation and Kalibr [20].

To estimate state, we then use the VINS-Fusion state estimation pipeline, that accepts these 2 inputs in the specified format, and outputs a predicted pose (orientation and location).

2.2.2 Results

We evaluated the performance of VINS-Fusion in pose estimation with these two input streams, in a relatively open space, where most of the visual features were markings on the walls or floor. As shown in Figure 2-2, the camera was placed on a rolling chair and manually moved in the ground truth path shown in red (measured by markers). The estimated path from VINS-Fusion is shown in green. The algorithm tracked the trajectory with a max error

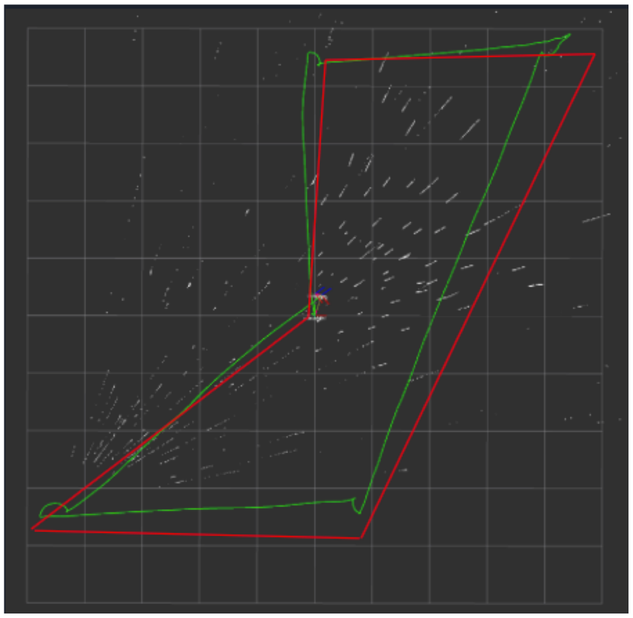


Figure 2-2: Results of trial of state estimation method. The Red path indicates the ground truth path, while the green path indicates the path estimated from VINS-Fusion. Areas where the estimation can be incorrect can be attributed mainly to propagation of previous error or lack of nearby visual features to detect. The max error (Euclidean distance between true point and estimated point) is 0.8 meters, while the ending error is 0.2 meters. This process was done without any loop closure logic.

margin of 0.8 meters. These results indicate that VINS-Fusion can provide valuable real-time information about state estimates in our indoor setting. However, much further testing is needed to evaluate the consistency and robustness of the algorithm before we have strong evidence that it can replace existing methods for localization. Based on our limited testing, we observed that VINS-Fusion requires a multitude of nearby objects to use as features, and performs poorly in open spaces without many nearby objects. We would need to see more accurate localization and feature tracking in open environments to be more confident in its use in environments similar to ours.

In this chapter, we have described related work in the field and described our experiment with state estimation using Vins-Fusion. Chapter 3 will detail background information in coordinate frames and pixel deviation that will be necessary to understand our proposed method of motion detection and tracking.

Chapter 3

Background

3.1 Coordinate Frame Definitions

This section defines the 3 coordinate frames used in this thesis. These frames are described in Figure 3-1 with the pinhole camera model.

1. Pixel Frame: 2D coordinate frame for a camera image. The u-axis represents the horizontal axis, and the v-axis represents the vertical axis. The origin is at the top left corner of the image. We denote a location in this frame as (u, v).

2. Global Frame: 3D Coordinate frame with origin set at specified point in the Simulation Environment. The X and Z axes cover the 2D plane parallel to the ground, and the Y axis spans the height of the environment. We will represent a point in the global frame with capital X,Y,Z: (X, Y, Z). In addition, when we detect and track object motion in this study, we are detecting the motion in the global frame (i.e. the object physically moved), even though we may represent the motion in other frames.

3. Local Camera Frame: 3D Coordinate frame with camera pose as origin. The z axis represents depth from camera ($+z$ points forward), the x axis corresponds to the axis horizontally across the camera left to right ($+x$ points right), and the y axis corresponds to vertical axis ($+y$ points down), as shown in Figure 3-1. We will represent a point

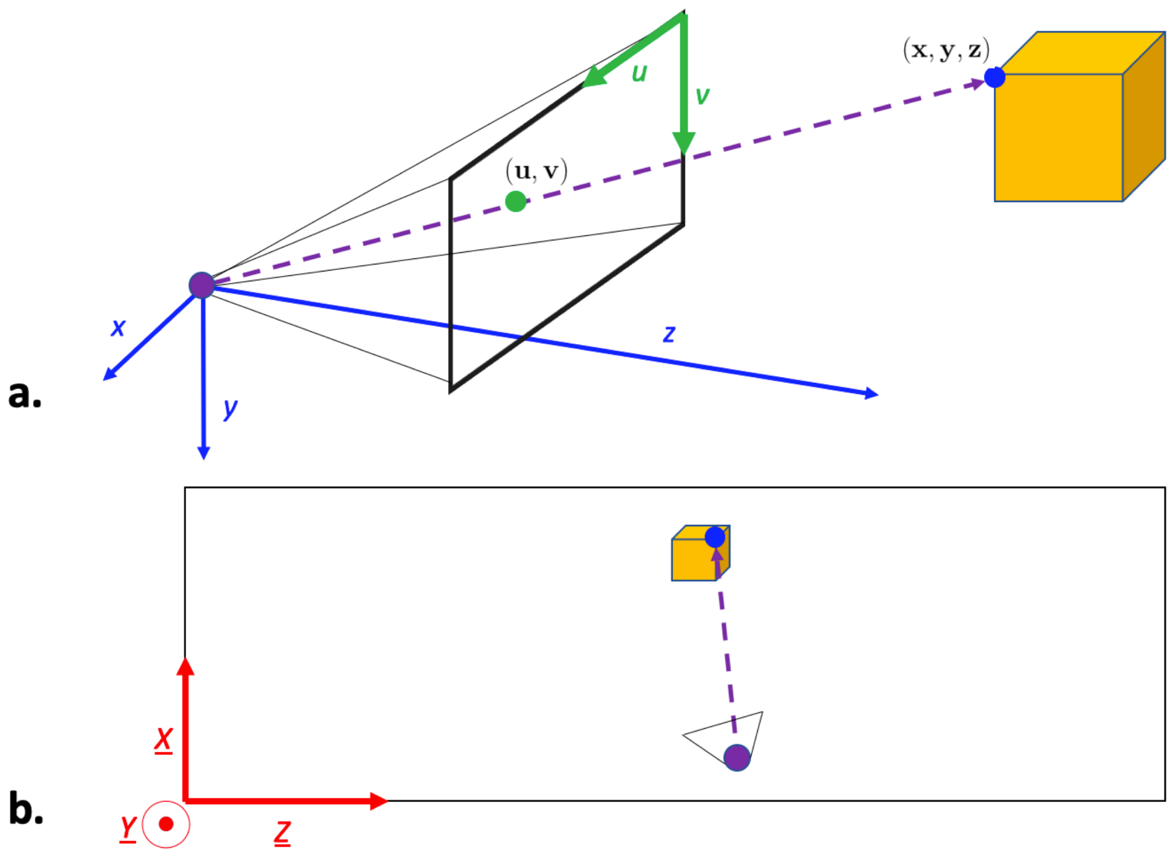


Figure 3-1: Relationship between the three coordinate frames used in this study. In (a), the black rectangle represents the pixel frame of the camera. The blue point (x, y, z) in the local camera frame shows up in the pixel frame (green) at (u, v). (b) is a top-down view of the global frame, which represents the entire AirSim environment. In (b), the camera (purple) and the corresponding projection line, as well as the gold box and the blue point are shown in the global frame (solid red). The axes of the global frame are represented by capital letters (underlined). The pixel frame is also shown in (b) as the black line that intersects the purple dotted projection line. For all 3 frames, the axes shown are the axes in the positive direction.

in the local camera frame with lowercase (x, y, z).

3.2 Pixel-Based Deviation for Stationary Camera Motion Detection

Our study will focus on pixel-based methods to detect motion. More specifically, we will track pixel values at a specific locations across a stream of images, and based on the distribution of those values, determine whether **deviation** (or motion) occurred at that specific pixel. In this section we describe this process and formulation for a stationary camera and describe the two ways we will measure deviation. In Chapter 4 we will describe how we extend these calculations to tackle the moving camera problem.

3.2.1 Formulation

We formulate the problem with the following input, output, and algorithm:

Input m RGB images (I_1 to I_m) (can be a moving window in a buffer) from the same stationary camera. Images are h pixels in height and w pixels in width.

Output 2D w x h array E, where $E[u_i, v_i] \in [-1, 0, 1]$, corresponding to whether positive deviation (+1), negative deviation (-1), or no deviation (0) was detected at pixel (u_i, v_i).

Algorithm The RGB images are first converted to grayscale (G_1 to G_m). The deviation algorithm compares pixels at the same location in the image across the m grayscale images and determines the extent of the pixel change. For a specific location (u_m, v_m), we can denote the deviation algorithm as:

$$E[u_m, v_m] = \mathit{deviation}(G_1[u_m, v_m], G_2[u_m, v_m], \ldots, G_m[u_m, v_m]) \tag{3.1}$$

Using a thresholding parameter θ, we then threshold $E[u, v]$

$$E[u_m, v_m] > \theta \rightarrow E[u_m, v_m] = 1 \tag{3.2}$$

$$E[u_m, v_m] < -\theta \rightarrow E[u_m, v_m] = -1 \tag{3.3}$$

$$|E[u_m, v_m]| < \theta \rightarrow E[u_m, v_m] = 0 \tag{3.4}$$

The idea behind this process is that motion should introduce pixel value deviations in a certain pixel location over a stream of images. If there is no object motion at (u_m, v_m), we expect $|E[u_m, v_m]| < \theta$. We will define our method of measuring deviation such that $E[u_m, v_m] = 1$ implies that pixel (u_m, v_m) is getting darker over the images, while $E[u_m, v_m] = -1$ implies that the pixel is getting lighter. The resulting 2D array E is denoted as a **deviation map**.

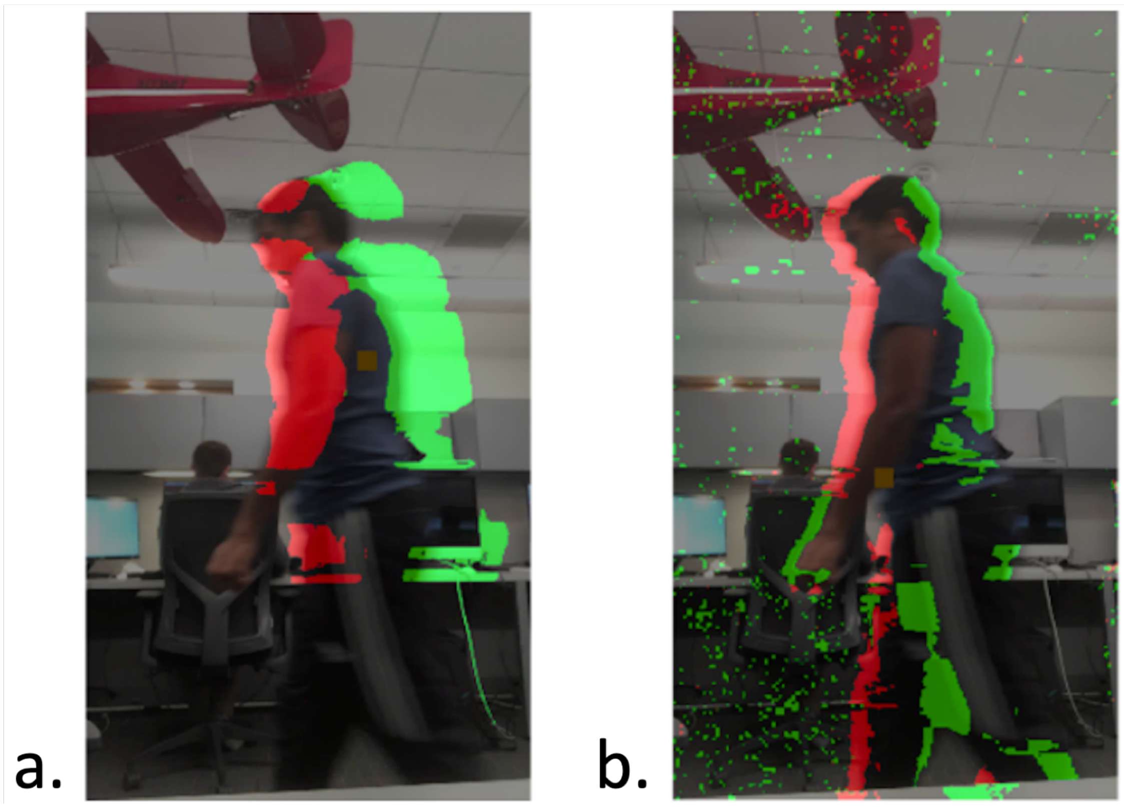


Figure 3-2: Example deviation maps for each of the two methods of measuring deviation. In this example, a person is walking from right to left across the pixel frame. Pixels with $E[u_m, v_m] = 1$ are colored green, pixels with $E[u_m, v_m] = -1$ are colored red. The red and green areas in this case provide a leading and trailing edge for the moving object. Map a. shows the result for the standard deviation, who's result is still able to identify the object but has less defined and thicker. Map b. shows the result for the Gaussian distribution percentile, which is able to identify the edges of the moving object but also detected deviations in the background.

3.2.2 Methods of Measuring Deviation

3.2.2.1 Standard Deviation

One method of measuring deviation is based on sample standard deviation. Sample standard deviation will analyze the deviation over all of the pixel values, treating each pixel with equal weight. We define this method of measuring deviation as follows (denoting the calculation as an implementation of the *deviation* function):

$$E[u_m, v_m] = deviation(G_1[u_m, v_m], G_2[u_m, v_m], \ldots, G_m[u_m, v_m]) \tag{3.5}$$

$$= \sigma[u_m, v_m] \cdot \texttt{sgn}(G_m[u_m, v_m] - \mu[u_m, v_m]) \tag{3.6}$$

where

$$\mu[u_m, v_m] = \frac{1}{m}\sum_{i=1}^{m} G_i[u_m, v_m] \tag{3.7}$$

$$\sigma[u_m, v_m] = \frac{1}{\gamma} stdev(G_1[u_m, v_m], G_2[u_m, v_m], \ldots, G_m[u_m, v_m]) \tag{3.8}$$

and γ is a normalization parameter. Note that the sign term compares the last pixel value to the mean to determine whether the deviation is positive or negative. Figure 3-2a describes an example result of this algorithm on a stream of RGB images where a person is moving from right to left across the pixel frame.

3.2.2.2 Gaussian Distribution Percentile

The second way of measuring deviation we will use is the Gaussian Distribution Percentile. In this method, for pixel values at location (u_m, v_m) across m frames, we fit the first $m-1$ values to Gaussian distribution $N(\mu^*[u, v], \sigma^*[u, v])$. We then determine the percentile of the last value $G_m[u_m, v_m]$ in the distribution. In essence, we are determining how close $G_m[u_m, v_m]$ is to $G_1[u_m v_m], G_2[u_m v_m], \ldots, G_{m-1}[u_m v_m]$. This algorithm determines how close the last pixel compared is to the (normal) distribution of the rest of the pixel values. More specifically, we calculate the deviation as follows (denoting the calculation as an implementation of the *deviation* function)

$$E[u_m, v_m] = deviation(G_1[u_m, v_m], G_2[u_m, v_m], \ldots, G_m[u_m, v_m]) \tag{3.9}$$

$$= (2\Phi(z[u_m, v_m]) - 1) \cdot \texttt{sgn}(G_m[u_m, v_m] - \mu^*[u_m, v_m]) \tag{3.10}$$

$$z[u_m, v_m] = \frac{|G_m[u_m, v_m] - \mu^*[u_m, v_m]|}{\sigma^*[u_m, v_m] + k} \tag{3.11}$$

$$\mu^*[u_m, v_m] = \frac{1}{m}\sum_{i=1}^{m} G_i[u_m, v_m] \tag{3.12}$$

$$\sigma^*[u_m, v_m] = stdev(G_1[u_m, v_m], G_2[u_m, v_m], \ldots, G_{m-1}[v_m, v_m] \tag{3.13}$$

where $\mu^*[u_m, v_m]$ and $\sigma^*[u_m, v_m]$ are maximum likelihood estimates of the mean and variance parameters of the Gaussian distribution fitted to the pixel values. Note that the Φ function is the CDF of $N(0,1)$. Note also that the sign term determines whether the deviation will be positive or negative. The extra $k = 15$ term is added to the denominator of the z-score calculation to dampen the sensitivity of the algorithm to low standard deviations. We multiply $\Phi(z[u_m, v_m])$ by 2 and subtract 1 to scale the output between 0 and 1. Figure 3-2b describes an example result of this algorithm on a stream of RGB images where a person is moving from right to left across the pixel frame.

While distribution percentile measures how different the pixel in the last image is from the previous images, standard deviation treats each pixel over the images equally and calculates deviation over all the pixels. We expect standard deviation to perform better in cases where an object may move so quickly that it moves in and out of the pixel location through the images, since the last pixel may not differ much from the previous ones in this case. We expect distribution percentile to perform better in cases where the object just enters the pixel location at the last image: the overall standard deviation would be low, but the last pixel will be significantly different than the previous ones.

In this chapter, we introduced the three coordinate frames relevant to our study. We also introduced the problem formulation for pixel deviation, and described the two ways we measure deviation in our study. Chapter 4 describes our proposed method in detail for moving object detection and tracking.

Chapter 4

Method for Object Motion Detection and Tracking with a Moving Camera

In this section, we explain the main contribution of the research project, which is extending the algorithm described in Section 3.2 to detect a moving object with a *moving camera*. To do so, we will introduce two extra inputs to the process, depth images aligned and synchronized with the RGB images, and camera poses in the global frame for each image. These inputs are detailed in Section 4.1.1.

4.1 Problem Statement and Summary

Our method has the following inputs and outputs, and certain assumptions.

4.1.1 Input and Output

Input:

- n RGB images (I_1 to I_n) from the same moving camera. The images are h pixels in height and w pixels in width, and correspond to timesteps $1 \to n$
- n corresponding Depth images (D_1 to D_n). Images are h pixels in height and w pixels

in width. These images are synchronized and aligned with the RGB images

- n camera poses (P_1 to P_n) in the global frame corresponding to camera poses (4×4 transformation matrix between the global frame and local camera frame) for each RGB image .

- Camera intrinsics: Contains focal lengths in both dimensions and image center pixel in both dimensions. We assume a pinhole camera model with no distortion since the images are taken in a simulation environment.

We define A_i as all inputs corresponding to timestep i: $A_i = (I_i, D_i, P_i)$.

Output: For given window-size $m \ll n$, list of locations $k_m, k_{m+1}, \ldots, k_n$, where k_i is a 3D (x, y, z) location in the local camera frame at the last timestep n. This sequence of locations represents the motion of the object over the images, as observed in the local camera frame of timestep n. The reasoning for this specific representation of object motion is detailed in Section 4.5.

4.1.2 Assumptions

The problem formulation makes two main assumptions.

- There is only one moving object in each scene. We make this assumption to focus on the algorithm's ability to detect moving objects rather than the ability to cluster and separate moving objects from each other.

- The moving object is in all n images. We make this assumption to prevent extra filtering needed from input images.

4.1.3 Method Summary

1. For each sliding window of $A_{i-m+1} \ldots A_i$

   (a) Calculate Deviation Map E_i from window (Section 4.2)

(b) Calculate *centroid* of deviation (u_i^c, v_i^c) from the deviation map, which locates the object in the pixel frame (Section 4.3.1)

(c) Project the centroid into the global frame, to get the global location (X_i^c, Y_i^c, Z_i^c) of the moving object at time i. (Section 4.3.2)

2. Transform all global locations (X_i^c, Y_i^c, Z_i^c) from the sliding windows into the local camera frame of timestep n. The result is the tracked locations of the object over timesteps m to n in that local camera frame. The locations do not start at $t = 1$, since a buffer of m is needed to run the algorithm. (Section 4.4 and 4.5)

4.2 Moving Camera Pixel Deviation Algorithm

4.2.1 Problem Statement

Input: We will detect deviation for pixels from a window of m images (RGB and Depth) and corresponding camera poses. More specifically our inputs are

- A_1 ,. . ., A_m
- Camera intrinsics: focal lengths in both dimensions, image center pixel in both dimensions

Output: 2D w x h array E_m, where $E_m[u_m, v_m] \in [-1, 0, 1]$, corresponding to whether positive deviation (+1), negative deviation (-1), or no deviation (0) was detected at pixel (u_m, v_m).

We denote the input and output of the process as the following:

$$E_m = PixelDeviation(A_1, A_2 \ldots, A_m)$$

4.2.2 New Formulation for Moving Camera

The validity of the stationary camera pixel deviation algorithm for motion detection relies on the following statements. We compare pixels at a specific location (u_m, v_m) across images. If the object at (u_m, v_m) (i.e. object that pixel u_m, v_m is representing) is stationary, then the pixel value at u_m, v_m will be the same across all images, resulting in no deviation. If the object at (u_m, v_m) moved over the images, then the pixel value at (u_m, v_m) should be different across the images, resulting in deviation at (u_m, v_m). In other words, in all images, pixel (u_m, v_m) corresponds to the same global location.

These statements need to be modified in the moving camera problem. With a moving camera, if a stationary object is at $I_k[u_k, v_k]$, in the previous image it may not be at $I_{k-1}[u_k, v_k]$. Instead, it could be at a different location in the previous image, $I_{k-1}[u_{k-1}, v_{k-1}]$. We will now use the following notation: In the moving camera problem for m images, a stationary object at $I_m[u_m, v_m]$ would be at the following points in the previous images:

$$I_{m-1}[u_{m-1}, v_{m-1}], I_{m-2}[u_{m-2}, v_{m-2}], \ldots, I_1[u_1, v_1]$$

The pixel deviation formulation originally from Section 3.2.1 is modified to become:

$$E_m[u_m, v_m] = \mathit{deviation}(G_1[u_1, v_1], G_2[u_2, v_2], \ldots. G_m[u_m, v_m]) \tag{4.1}$$

This algorithm now relies on that statement that if the object at (u_m, v_m) is stationary, then $I_1[u_1, v_1]$, $I_2[u_2, v_2]$ $\ldots$, $I_m[u_m, v_m]$ will have similar pixel values. If the object moved, then the pixel values will be significantly different. For this statement to be accurate, $I_1[u_1, v_1]$, $I_2[u_2, v_2]$ $\ldots$, $I_m[u_m, v_m]$ must represent the same global location. Note that we also threshold E_m with a threshold θ as described in Section 3.2.1.

4.2.3 Determining Pixel Locations to Compare

This section will detail how we determine corresponding pixel locations (u_i, v_i) for images $i = 1 \ldots m-1$ from pixel location (u_m, v_m). We accomplish this by analyzing transforms between camera poses of consecutive images. We observe an object (assume stationary) at $I_m[u_m, v_m]$ with camera at pose P_m and determine where that object should be in the pixel frame of a camera with the previous pose P_{m-1}. More specifically, we project (u_m, v_m) into the local camera frame of image m, resulting in 3D point (x_m, y_m, z_m).

$$\begin{bmatrix} \frac{d_m}{f_u s} & 0 & \frac{-d_m c_u}{f_u s} & 0 \\ 0 & \frac{d_m}{f_v s} & \frac{-d_m c_v}{f_v s} & 0 \\ 0 & 0 & \frac{1}{s} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_m \\ v_m \\ d_m \\ 1 \end{bmatrix} = \begin{bmatrix} x_m \\ y_m \\ z_m \\ 1 \end{bmatrix} \tag{4.2}$$

where for notational simplicity, $d_m = D_m[u_m, v_m]$. f_u and f_v are the camera focal lengths in the u and v directions, respectively, and c_u and c_v are the camera center pixels in the u and v directions, respectively. Variable s is the scaling factor of the depth camera, such that $\frac{d_m}{s}$ is the true depth in meters to pixel (u_m, v_m).

We then transform (x_m, y_m, z_m) into a point (X_m, Y_m, Z_m) in the global frame using the global camera pose P_m

$$P_m \begin{bmatrix} x_m \\ y_m \\ z_m \\ 1 \end{bmatrix} = \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} \tag{4.3}$$

If the object at pixel (u_m, v_m) was stationary, then the object would also be at global point (X_m, Y_m, Z_m) at image $m-1$. We now need to find what pixel in image $m-1$ corresponds to the global point (X_m, Y_m, Z_m). To do so, we first transform (X_m, Y_m, Z_m) into the local camera frame of image $m-1$, using the global camera pose of image $m-1$ (P_{m-1})

$$(P_{m-1})^{-1} \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} = \begin{bmatrix} x_{m-1} \\ y_{m-1} \\ z_{m-1} \\ 1 \end{bmatrix} \tag{4.4}$$

Now we project $(x_{m-1}, y_{m-1}, z_{m-1})$ into the pixel frame of image $m-1$, resulting in (u_{m-1}, v_{m-1}),

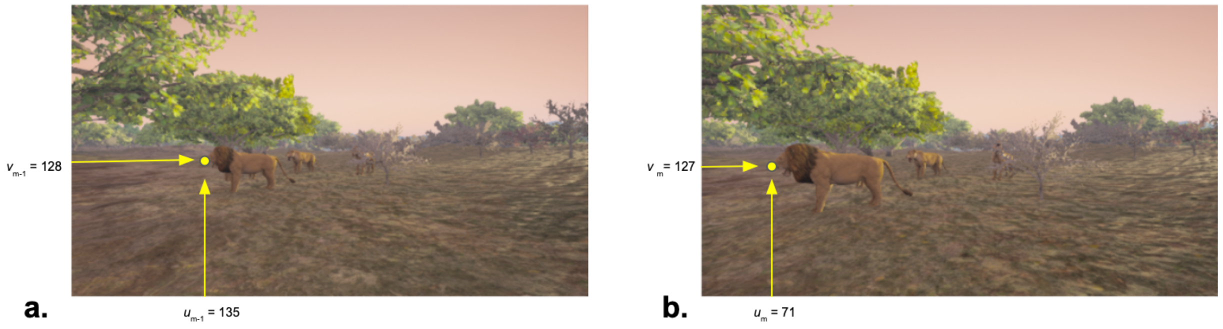


Figure 4-1: Example result of mapping of two pixel locations in RGB images m (b) and $m-1$ (a). In this example, we will track the pixel corresponding to the Lion's nose. The lion is stationary between these two pictures, but the camera is moving forward from (a) to (b). (b) shows the pixel location of the Lion's nose in image m. The algorithm determines the pixel location in image $m-1$ to compare it to, shown in (a). Both pixel locations over the two images correspond to the Lion's nose, showing the accuracy of the calculation. These images are taken from the AirSim African Safari Environment

which will be the pixel in image $m-1$ representing the global location (X_m, Y_m, Z_m)

$$\begin{bmatrix} u_{m-1} \\ v_{m-1} \\ 1 \end{bmatrix} = \frac{1}{z_{m-1}} \begin{bmatrix} f_u & 0 & c_u & 0 \\ 0 & f_v & c_v & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{m-1} \\ y_{m-1} \\ z_{m-1} \\ 1 \end{bmatrix} \tag{4.5}$$

We now have one pixel location from each of two camera images that represent the same point in global space: (u_{m-1}, v_{m-1}) and (u_m, v_m). If an object at (X_m, Y_m, Z_m) was stationary, with similar logic explained in Section 4.2.2, pixel values at $I_{m-1}[u_{m-1}, v_{m-1}]$ and $I_m[u_m, v_m]$ should be very similar.

In summary, we have taken a pixel location in image m (u_m, v_m) and determined the pixel location in image $m-1$ (u_{m-1}, v_{m-1}) to compare it with to calculate deviation. An example result of this process is shown in Figure 4-1.

We can extend this process to determine (u_{m-2}, v_{m-2}) from (u_{m-1}, v_{m-1}), and so on, until we calculate (u_1, v_1)

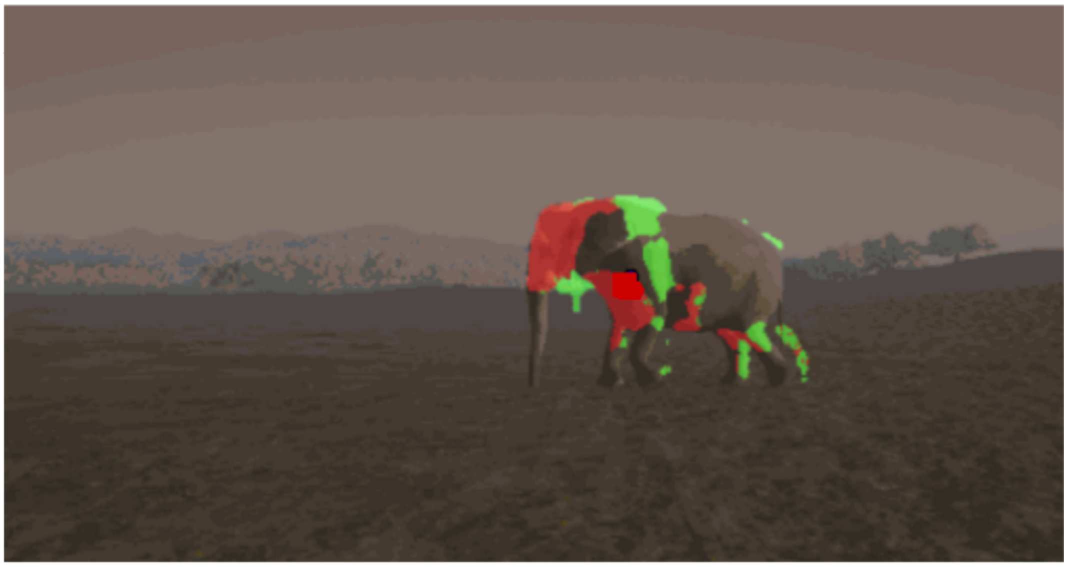


Figure 4-2: This is the deviation map E_m overlaid on the last image in the stream (I_m), with sample standard deviation for measuring deviation. The camera has moved forward over the m frames used to calculate the deviations. Pixels with $E_m[u_m, v_m] = 1$ are colored green, pixels with $E_m[u_m, v_m] = -1$ are colored red. The green and red areas in this case provide rough leading and trailing edges for the moving object, which is an elephant walking right to left across the screen. However, deviation is also calculated in the center of the moving object. Note that a few of the elephant's legs also have leading and trailing edges. The algorithm is able to subtract out background pixel changes due to camera motion and only elucidate the moving object.

4.3 Extracting Object Location from 1 Deviation Map

4.3.1 Determining Centroid of Deviation

We define E_m as a 2D w x h **deviation map** at image m, where $E_m[u_k, v_k]$ for each (u_k, v_k) in the image is calculated from the process in Section 4.2. If $|E_m[u_k, v_k]| = 1$, we say that the pixel (u_k, v_k) has deviation. An example of this deviation map is shown in Figure 4-2, with sample standard deviation for measuring deviation. Pixels with deviation are shown in red or green. In Figure 4-2, the green and red areas of deviation form a rough mask on the moving object. For any object translation, we rely on the presence of these colored areas in the image to create this rough mask.

We then utilize this mask to locate the moving object. Under the assumption that there is only 1 moving object in the image, we expect that pixels with deviation form an approximate mask for that object. Thus, we expect the centroid (u_m^c, v_m^c) of those pixels to be located on

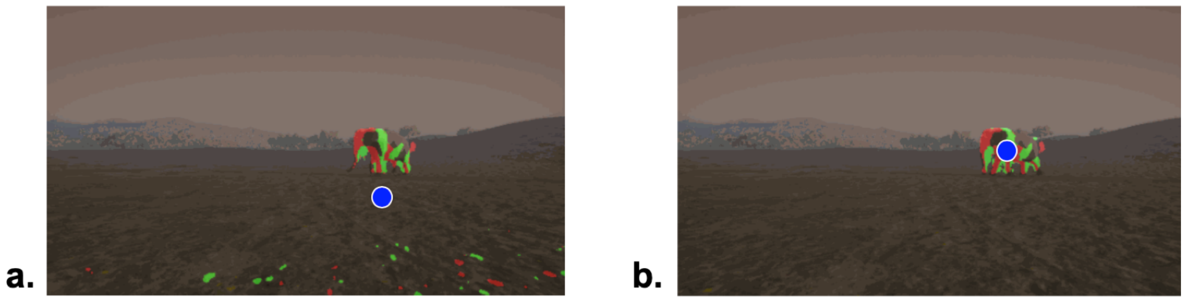


Figure 4-3: Example Comparison between algorithm for Stationary vs Moving Camera. (a) shows an example deviation map from assuming no change in camera pose (i.e. stationary pixel deviation algorithm). (b) shows the deviation map outputted by the moving pixel deviation algorithm that does take into account changing camera pose. There is significantly less background (more specifically in the terrain) deviation detected in (b). Centroids are shown as blue points. The centroid in (b) is on the moving object, while the centroid in (a) is not. The standard deviation method of measuring deviation was used in this example.

the object (noted as **centroid of deviation**):

$$U = \{u_m \forall u_m \in [0, w-1] : E_m[u_m, v_m] \in \{-1, 1\}\} \tag{4.6}$$

$$V = \{v_m \forall v_m \in [0, h-1] : E_m[u_m, v_m] \in \{-1, 1\}\} \tag{4.7}$$

$$u_m^c = \frac{1}{|U|} \sum_{u \in U} u \tag{4.8}$$

$$v_m^c = \frac{1}{|V|} \sum_{v \in V} v \tag{4.9}$$

In other words, these equations determine the mean u-value and v-value out of the pixels with deviation.

After incorporating camera poses and depth images to find the correct pixels to compare over frames, ideally the only pixels with deviation should be those that represent moving objects. Therefore, the centroid of deviation should lie on the object the vast majority of the time. Figure 4-3 shows an examples of deviation maps and corresponding centroids, with and without incorporating camera poses and depth images. In other words, it compares the results of using a stationary algorithm formulated in Section 3.2 versus using the algorithm we describe in this section (moving camera algorithm). The moving camera algorithm does a significantly better job of only identifying deviation on the moving elephant.

4.3.2 Projecting the Centroid to the Global Frame

We can then determine the global location of the object from projecting the centroid into the global frame, using a similar process as in Section 4.2.3. Again, let $d^c_m = D^c_m[u^c_m, v^c_m]$, then

$$\begin{bmatrix} \frac{d^c_m}{f_u s} & 0 & \frac{-d^c_m c_u}{f_u s} & 0 \\ 0 & \frac{d^c_m}{f_v s} & \frac{-d^c_m c_v}{f_v s} & 0 \\ 0 & 0 & \frac{1}{s} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_c \\ v_c \\ d^c_m \\ 1 \end{bmatrix} = \begin{bmatrix} x^c_m \\ y^c_m \\ z^c_m \\ 1 \end{bmatrix} \tag{4.10}$$

$$P_m \begin{bmatrix} x^c_m \\ y^c_m \\ z^c_m \\ 1 \end{bmatrix} = \begin{bmatrix} X^c_m \\ Y^c_m \\ Z^c_m \\ 1 \end{bmatrix} \tag{4.11}$$

Through the process described in equations 4.10 and 4.11, from a deviation map E_m and inputs $A_m = (I_m, D_m, P_m)$, we are able to determine the global location of the moving object at image m: (X^c_m, Y^c_m, Z^c_m). We will denote this process through the following function.

$$(X^c_m, Y^c_m, Z^c_m) = Locate(E_m, A_m)$$

4.4 Object Tracking from Multiple Deviation Maps

When we have an image stream of length $n > m$ for a single object, we can calculate multiple deviation maps and global locations for the object, one of each for each buffer of m images:

$$E_m = PixelDeviation(A_1, A_2 \ldots A_m) \quad \rightarrow \quad (X^c_m, Y^c_m, Z^c_m) = Locate(E_m, A_m) \tag{4.12}$$
$$E_{m+1} = PixelDeviation(A_2, A_2 \ldots A_{m+1}) \quad \rightarrow \quad (X^c_{m+1}, Y^c_{m+1}, Z^c_{m+1}) = Locate(E_{m+1}, A_{m+1})$$
$$\vdots$$
$$E_n = PixelDeviation(A_{n-m}, A_{n-m+1} \ldots A_n) \quad \rightarrow \quad (X^c_n, Y^c_n, Z^c_n) = Locate(E_n, A_n)$$

We can create a trajectory that the object took from the global locations. We denote this process as *CreateTrajectory*

$$\begin{bmatrix} (X^c_m, Y^c_m, Z^c_m) \\ (X^c_{m+1}, Y^c_{m+1}, Z^c_{m+1}) \\ \vdots \\ (X^c_n, Y^c_n, Z^c_n) \end{bmatrix} = CreateTrajectory((X^c_m, Y^c_m, Z^c_m), \ldots, (X^c_n, Y^c_n, Z^c_n)) \tag{4.13}$$

4.5 Determining Local Motion Vector

In the application of a camera on a robot, or an human observer with the camera, that has observed from A_1 up to A_n, the raw global locations resulting from $CreateTrajectory$ may not give useful information. What is more useful for planning applications is those locations in the *current local camera frame* of the robot, which would be the local camera frame at timestep n. Intuitively, the locations in the local camera frame would represent how the object motion looked relative to an stationary observer holding a camera at its current position. To calculate the local motion vector, we transform each of the global points into the local camera frame of time n as shown in Equation 4.14. To make the equation clearer, we denote the matrix output of $CreateTrajectory$ as B,

$$B = \begin{bmatrix} X^c_m & Y^c_m & Z^c_m & 1 \\ X^c_{m+1} & Y^c_{m+1} & Z^c_{m+1} & 1 \\ \vdots & & & \\ X^c_n & Y^c_n & Z^c_n & 1 \end{bmatrix} \tag{4.14}$$

and we calculate the motion vector in the local camera frame of time n as shown in Equation 4.15, denoting the process with the name $ConvertToLocalN$:

$$ConvertToLocalN(B) = B(P_n^{-1})^T = \begin{bmatrix} x_m^c & y_m^c & z_m^c & 1 \\ x_{m+1}^c & y_{m+1}^c & z_{m+1}^c & 1 \\ \vdots & & & \\ x_n^c & y_n^c & z_n^c & 1 \end{bmatrix} \tag{4.15}$$

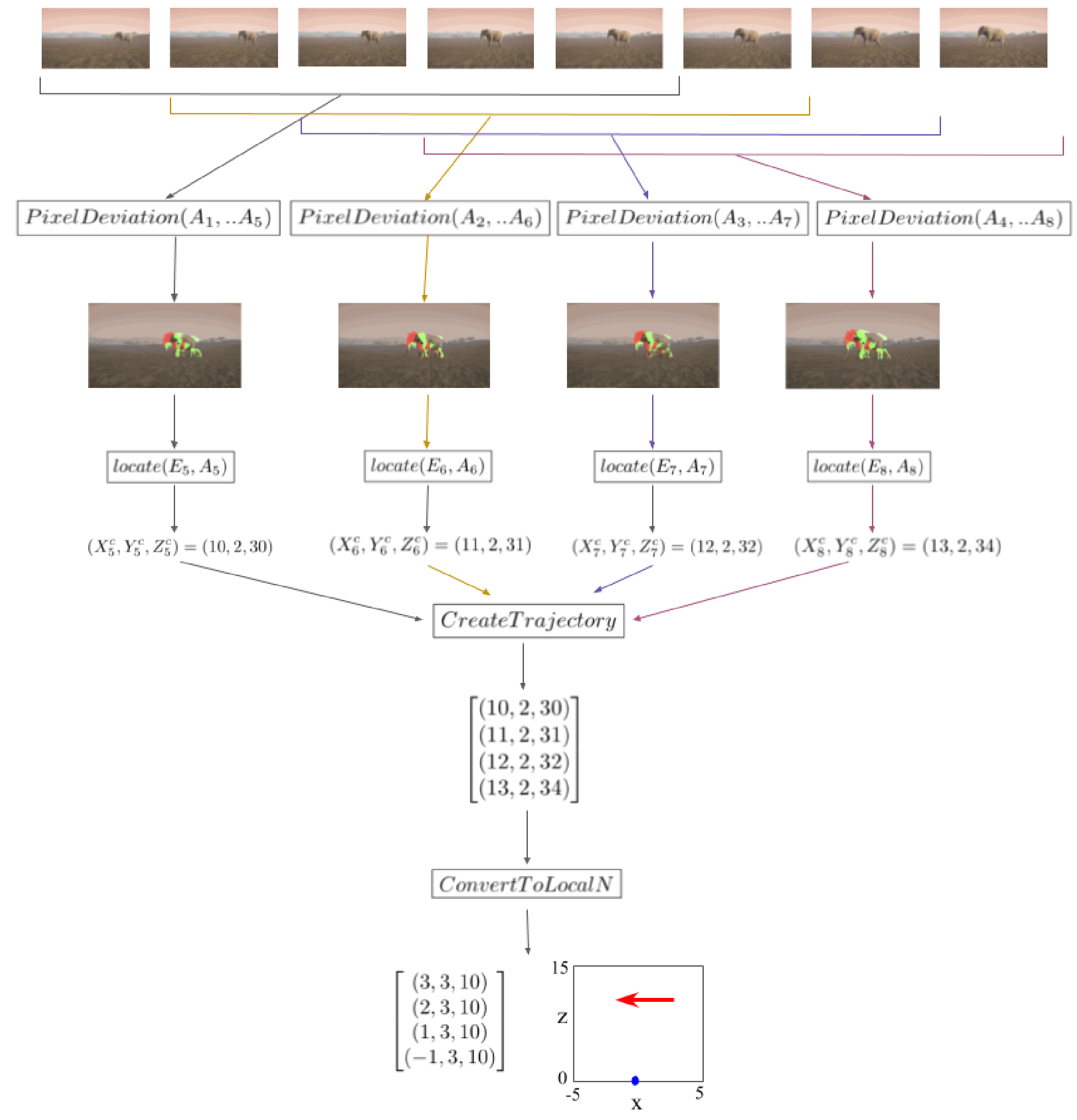
4.6 Summary and Example of Process

We have now described the entire object motion detection and tracking pipeline, having input $(A_1 \dots A_n)$, which is n RGB images, depth images, and camera poses, and outputs a list of (x, y, z) locations in the local camera frame of time n. Specified camera intrinsics are included in the input as well. We denote this entire process in Equation 4.16

$$\begin{bmatrix} x_m^c & y_m^c & z_m^c & 1 \\ x_{m+1}^c & y_{m+1}^c & z_{m+1}^c & 1 \\ \vdots & & & \\ x_n^c & y_n^c & z_n^c & 1 \end{bmatrix} = TrackObject(A_1, A_2, \dots A_n) \tag{4.16}$$

We also show an demo example of the entire process in Figure 4-4, and we detail pseudocode of the demo process in Algorithm 1

In this chapter, we described our proposed method of motion detection and tracking with a moving camera. We showed a demo example of the entire pipeline in Figure 4-4. In Chapter 5, we will create a dataset and evaluate the performance of our method on the dataset.



$PixelDeviation(A_1, ..A_5)$
$PixelDeviation(A_2, ..A_6)$
$PixelDeviation(A_3, ..A_7)$
$PixelDeviation(A_4, ..A_8)$
$locate(E_5, A_5)$
$locate(E_6, A_6)$
$locate(E_7, A_7)$
$locate(E_8, A_8)$
$(X_5^c, Y_5^c, Z_5^c) = (10, 2, 30)$
$(X_6^c, Y_6^c, Z_6^c) = (11, 2, 31)$
$(X_7^c, Y_7^c, Z_7^c) = (12, 2, 32)$
$(X_8^c, Y_8^c, Z_8^c) = (13, 2, 34)$
$CreateTrajectory$
$\begin{bmatrix}(10,2,30)\\(11,2,31)\\(12,2,32)\\(13,2,34)\end{bmatrix}$
$ConvertToLocalN$
$\begin{bmatrix}(3,3,10)\\(2,3,10)\\(1,3,10)\\(-1,3,10)\end{bmatrix}$
15
z
0
-5
x
5

Figure 4-4: Demo Example of the *TrackObject* Pipeline. In this example, as input we have $n = 8$ RGB images (and corresponding depth image, camera poses, and camera intrinsics, which are not shown). We use a window size of $m = 5$. We calculate the deviation map for each window of $m = 5$ inputs through the *PixelDeviation* method. In this example we use sample standard deviation as our method of measuring deviation. For each deviation map, we then locate the centroid in the global frame using the *locate* method. We combine the global locations into a trajectory using the *CreateTrajectory* method, and we transform those locations into the local camera frame of timestep n using the *ConvertToLocalN* method. In this example, the elephant is moving to the left in the pixel frame, and the resulting trajectory shows that by showing movement in the $-x$ direction. In the trajectory at the bottom, the blue dot is the origin in the local camera frame of time n (i.e. location of camera), and the red arrow is the vector from the first to last location in the resulting trajectory. The vector is plotted on the x-z plane, ignoring the y coordinate.

Algorithm 1 *TrackObject* Demo Example

```
$n \leftarrow 8$ {length of input}
$m \leftarrow 5$ {window size for calculating deviation}
$i \leftarrow 0$ {counter}
points $\leftarrow$ []
while $i + m < n$ do
    $E \leftarrow PixelDeviation(A_i \ldots A_{i+m})$
    $C \leftarrow locate(E, A_{i+m})$
    points.add($C$)
    $i \leftarrow i + 1$
end while
TrajGlobal $\leftarrow CreateTrajectory$(points)
Traj $\leftarrow ConverToLocalN$(TrajGlobal)
return Traj
```

Chapter 5

Results

5.1 Dataset

To evaluate our method, we needed to collect a dataset that has the following pieces of information

1. RGB images of moving objects (as input to the algorithm)

2. Accurate depth images synchronized and aligned with the RGB images (as input to the algorithm)

3. Accurate camera poses in the global frame for each RGB/depth image pair (as input to the algorithm)

4. Ground Truth Object Segmentations for each RGB image (for evaluation)

Since we wanted the inputs and segmentation to be as accurate as possible, we decided to collect a dataset in a simulation environment. Using AirSim with Unreal Engine in the African Safari, we collected a dataset of 110 examples, where each example is the sequence $A_1, A_2, \ldots, A_n$, and corresponding ground truth segmentation. The examples can overlap (i.e. A_i can be in multiple examples at different locations. Thus, after running *TrackObject* over each example (sequence of n inputs), we receive 110 trajectories to analyze performance with. In our collected dataset, $n = 10$ and the window size $m = 5$.

5.1.1 Restrictions

We apply the following restrictions to create a focused dataset for our task:

1. In each example there is only one moving object.
2. In this study we will also focus purely on forward camera translational motion, so in these examples, we restrict camera motion to forward motion (i.e. $+z$ motion in the local camera frame, with no camera movement in x or y directions).
3. The moving objects in the environment have negligible movement in the $\pm y$ direction in the local camera frame. The z axis in the local camera frame is always parallel to the ground, and the objects do not exhibit an elevation change. With this restriction, we only analyze the x and z coordinates of the output of *TrackObject*.

5.2 Quantities for Evaluating Performance

Taking as input $A_1 \ldots A_n$ and window size m, *TrackObject* outputs a list of object locations from time m to n in the local camera frame of timestep n.

We define the estimated **movement vector** v to be the displacement of x and z coordinates between the first and last points (at timestep m and timestep n).

$$v = [x_n^c - x_m^c, z_n^c - z_m^c]$$

5.2.1 Descriptions

We want to analyze the performance of *TrackObject* with the two goals of the method in mind: 1) Motion detection and 2) Motion Tracking. We will define detection error as a quantity to evaluate performance in motion detection, and will define 3 other quantities to evaluate performance in motion tracking.

1. Detection Error:
    - Fraction of the $n - m$ centroids are NOT on the moving object
    - This quantity evalutes how well *TrackObject* segments out the moving object
2. Angle Error:
    - Angle (in radians) between true movement vector and estimated movement vector
3. Magnitude Error:
    - True movement Vector: v_0
    - Estimated movement vector: v
    - Magnitude Error $= \left| \frac{|v| - |v_0|}{|v_0|} \right|$
4. Start Position Error:
    - Euclidean distance between estimated first position of vector (x_m^c, z_m^c) and true (x, z) position of object at timestep m

In a classification problem, a quantity analogous to detection error may not be used due to false positive and false negative rates influencing the quantity. In these cases precision/recall and Receiver-Operating Characteristic Area Under the Curve (ROC-AUC) provide better evaluations of the true performance of the systems. However, for detection error, these more sophisticated quantities are not needed, since the algorithm will be able to correctly detect the object only if it finds a correct centroid in the deviation map, a process that is not going to be influenced by false positive and negative rates as per normal classification problems.

5.2.2 Determining Ground Truth Motion

For detection error, we can easily determine whether a centroid is or is not on the moving object for a deviation map, since we have the corresponding ground truth segmentation. We just check that the centroid is inside the mask given by the segmentation.

To determine the ground truth motion vector v_0, we do the following:

1. Get the ground truth segmentation for RGB image m, corresponding to the first timestep in the trajectory.

2. For each pixel in the mask of the object, using A_m, project the pixel into the global frame and determine the global pose of each pixel (in an analogous process to Section 4.3.2)

3. For the 3D global poses of the pixels, determine the mean 3D global location.

4. Transform that mean global point into the local camera frame of timestep n. The result is the ground truth start point: S_{true}

5. repeat steps 1-4 for RGB image n, corresponding to the last timestep in the trajectory, to get the ground truth end point: Y_{true}

6. Determine the displacement between the two values in the $x - z$ plane ($S_{true}[x, z] - Y_{true}[x, z]$)

5.2.3 Example

To explain the quantities for measuring error more, we will calculate their values for the example shown in Figure 4-4, assuming that the ground truth movement vector (in the $x-z$ plane) is $(\Delta x_{true}, \Delta z_{true}) = (-4.5, 0.4)$ and the ground truth start position of the vector is $(x_{true}^{start}, z_{true}^{start}) = (3.2, 10.4)$. As shown in the 4 deviation maps in Figure 4-4, all 4 centroids of deviation (tiny blue dots on the deviation map) are on the elephant. Thus, detection error $= 0.0$.

In Figure 4-4, the estimated movement vector, is the displacement between the first and last values outputted by *ConvertToLocalN*: $(\Delta x_{est}, \Delta z_{est}) = (-4.0, 0.0)$. The estimate start position is just the x and z coordinates of the first value: $(x_{est}^{start}, z_{est}^{start}) = (3, 10)$

From these values, the angle error is the interior angle between the two vectors $(\Delta x_{true}, \Delta z_{true})$ and $(\Delta x_{est}, \Delta z_{est})$, which we calculate to be 0.08 rads. The start position error is

$$||(x_{true}^{start}, z_{true}^{start}) - (x_{est}^{start}, z_{est}^{start})|| = 0.44\text{m}$$

Finally, the magnitude error is:

$$\left| \frac{|(\Delta x_{true}, \Delta z_{true})| - |(\Delta x_{est}, \Delta z_{est})|}{|(\Delta x_{true}, \Delta z_{true})|} \right| = 0.11$$

5.3 Performance on Dataset

We evaluated the performance of *TrackObject* on our dataset for each of the two methods of measuring pixel deviation. The results are shown in Figure 5-1 which contains four graphs, one for each of the four quantities described in Section 5.2.1. For each quantity for evaluating performance, we show the performance of *TrackObject* for the 2 methods of measuring deviation (Standard Deviation, Distribution Percentile). For each method, we compare its performance with a baseline in which the algorithm assumes no camera motion (i.e. assumes all camera poses are the same). We show this comparison to emphasize that incorporating camera motion (and calculating the correct pixels to compare, rather than comparing the same pixel locations on the raw RGB images) is important for accurate object motion detection. In essence, the baseline performance represents running the stationary pixel-deviation algorithm from Section 3.2 and extracting trajectories from its output.

The detection errors of both algorithms are better than their baseline assume-no-camera-motion counterparts, stressing the importance of incorporating camera poses and camera motion to accurately detect the moving object. Looking at the solid blue and solid green bars, both methods of measuring deviation show a median detection error of at most 0.17, and standard deviation is the more accurate with a median of 0.0. Standard deviation also has the lower median angle error, start position error, and magnitude error.

We conducted 2-sample t-tests to compare the performances of the two methods of measuring

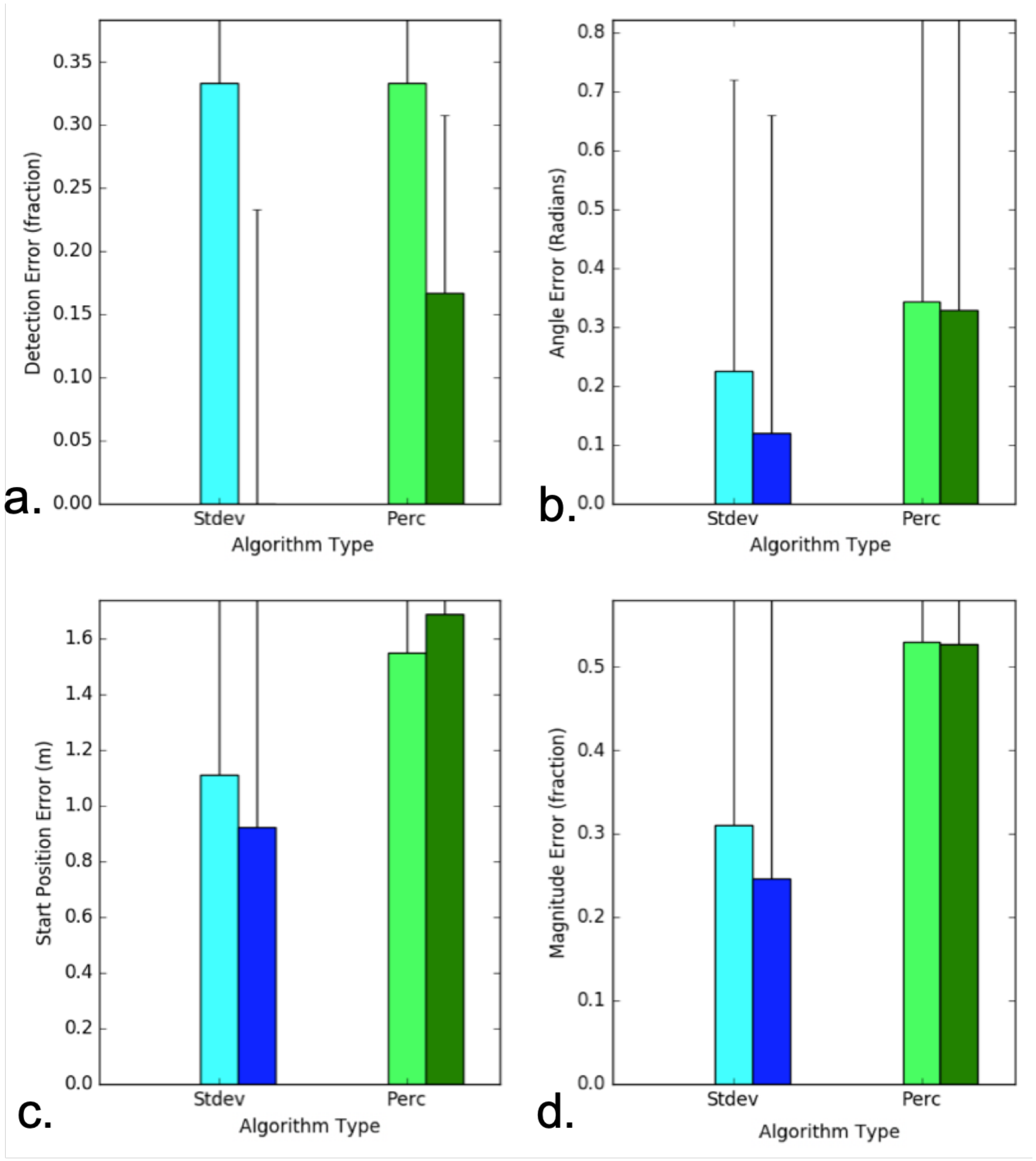


Figure 5-1: Performance of *TrackObject* for different methods of measuring deviation. Each graph represents performance evaluated with a different quantity – (a) shows detection error, (b) shows angle error, (c) shows magnitude error, and (d) shows start position error. For each graph, we compare the performances of *TrackObject* while varying the method of measuring deviation: Stdev=Sample Standard Deviation (Blue), Perc=Gaussian Distribution Percentile (Green).

The bar plots the median value (i.e. median detection error for graph a) across the 110 examples in the dataset, with error bars representing the 50 percent confidence interval of the median. We plot two values for each method of measuring deviation represented by side-by-side bars (i.e. pink and red). The lighter-colored bar on the left represents the performance of *TrackObject* when it assumes that the camera did not move (i.e. for each example, all camera poses are the same: $P_1 = P_2 =, \ldots, P_n$). The darker-colored bar on the right shows the performance for the regular algorithm using the real camera poses and transforms (i.e. the output of the *TrackObject* algorithm we described in Chapter 4). An optimal threshold θ was tuned for each of the two methods of measuring pixel deviation and used in the process.

deviation. For each of 4 performance quantities in 5.2.1, we compared the performance of both normal methods, not baselines. For the two methods, we used the mean and variance of the results to conduct a t-test to determine the probability that the difference in means is significant. There were no results where p-value < 0.05, indicating that there is no significant difference in performance between the two methods of measuring deviation.

The confidence intervals for the start position error and magnitude error quantities are larger by magnitudes than the corresponding median values. This observation can be attributed to the following: when a centroid is calculated from a deviation map, if the centroid is incorrect, it usually is a pixel location in the background of the image. When the global pose of the pixel location is calculated, the calculated location would be in the background of the image, which can be hundreds of meters away from the actual object. A location in the trajectory output by *TrackObject* that is incorrect to by this amount can introduce errors several magnitudes more than the true value itself in the case of start position error and magnitude error.

Due to this reasoning, in Figure 5-1 we primarily analyze detection error and angle error (Figure 5-1a and 5-1b) as a indication of performance, since error bars for the two quantities are small enough that trends can be inferred from the graphs.

Based on these observations, it is also important to get a sense of how close the estimated movement vector is to the true movement vector *when the moving object is correctly detected.* To evaluate this aspect, we filtered the results from our 110 examples to only include examples where the algorithm correctly detected the moving object in all $n - m$ deviation maps (i.e. examples where detection error is 0.0). For these examples, we determine how well the algorithm estimates the movement vector with the 3 other quantities, as shown in Figure 5-2. The 85% confidence intervals in Figure 5-2 are significantly more bounded than the intervals of Figure 5-1. For both methods of measuring deviation, when the moving object is detected by centroid correctly, we can be confident that the angle error does not exceed 0.6 rads and the start position error does not exceed 1.5m. For standard deviation, we can be confident that the magnitude error (proportion) does not exceed 0.4.

Overall, these results demonstrate that *TrackObject* is able to detect and track moving objects in our dataset, especially with the standard deviation method of measuring deviation. Although standard deviation may seem to perform better than Gaussian distribution percentile, based on Figure 5-1, large confidence intervals prevent that claim from having strong evidence.

To supplement these numerical results, we also provide qualitative results from the algorithm. More specifically, Figure 5-3 explains examples of accurate and inaccurate deviation maps calculated by our algorithm.

5.4 Robustness to Input Noise

The real world will not have the same level of accuracy and precision as the depth and camera pose estimates we receive in a simulation environment. Therefore, it is important to determine how adding error to our depth and camera pose estimates affects the performance of $TrackObject$. In this experiment, we will use our seemingly best performing method of measuring deviation: standard deviation. Note that we make this statement with some caution due to large error bars in our median results.

5.4.1 Adding Noise to Input

We will add noise to our inputs through the following processes.

5.4.1.1 Depth

For every depth value d in a depth image, we set it to a new depth value as $d_{new} = d(1 + \lambda)$ where scalar λ is sampled from $N(0, \sigma_d^2)$ (new sample for every depth value).

5.4.1.2 Pose

We will focus noise to the location of the camera (rather than the orientation), since our study deals with only translational movements. For every global camera location (X, Y, Z),

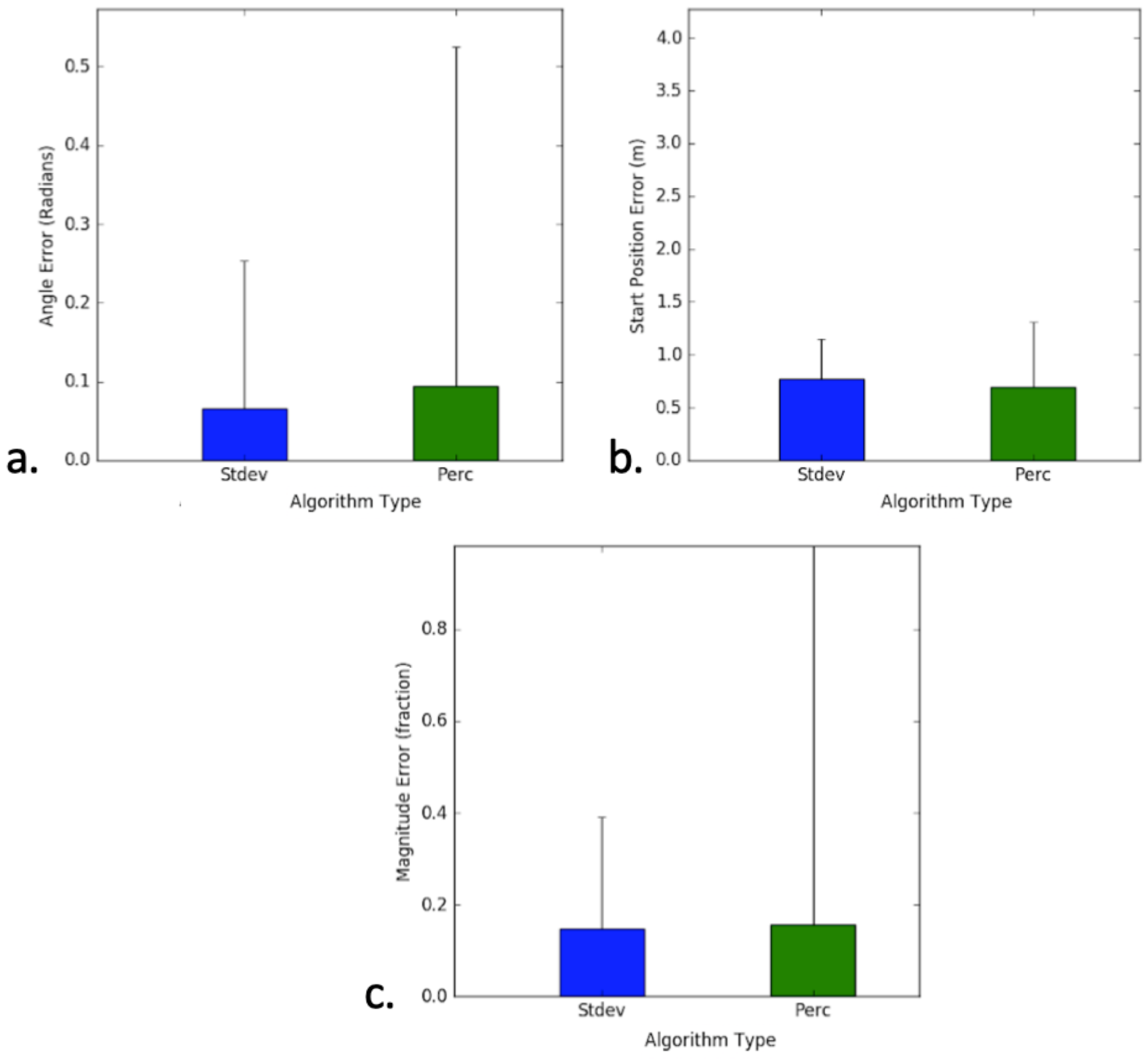


Figure 5-2: Performance of *TrackObject* for different methods of measuring deviation on examples with 0 detection error. (a) shows the angle error when using each of the two methods: Stdev=Sample Standard Deviation (Blue), Perc=Gaussian Distribution Percentile (Green). (b) shows start position error and (c) shows Magnitude Error, varied based on the method of measuring deviation used. Note that we do not show the stationary-algorithm baselines here. The height of each bar represents the median value across the examples, while the error bars represent an 85% confidence interval (within 1 z-score of the median)

we set the new location as follows:

$$X_{new} = X + \lambda, \quad Y_{new} = Y + \lambda, \quad Z_{new} = Z + \lambda$$

so the same λ is used for the three coordinates, and is sampled from $N(0, \sigma_p^2)$ (new sample for every pose).

5.4.2 Performance

5.4.2.1 Robustness to Depth Noise

Figure 5-4 shows the performance of *TrackObject* using standard deviation (for measuring deviation) using the 4 main quantities in Section 5.2.1 as we vary the noise parameter σ_d. From the graphs, it is not possible to tell if there is a significant overall increase or decrease in performance in the 4 quantities as we increase σ_d due to the large error bars associated with the performance values. We do not see a significant difference in performance with varying depth error, but due to the large error bars we are unable to discern the extent of the robustness.

5.4.2.2 Robustness to Pose Noise

Figure 5-5 shows the performance of *TrackObject* using standard deviation for measuring deviation evaluated by the 4 main quantities in Section 5.2.1 as we vary the noise parameter σ_p. Graphs 5-5a and 5-5b of detection error and angle error respectively show that increasing σ_p significantly hinders the performance of the system. Even $\sigma_p = 0.5$ results in the median detection error of 0.5 and angle error of 1.2 radians. Graphs 5-5c and 5-5d have error bars that are too large to make any conclusions from them specifically. Overall, these results show that the performance of *TrackObject* is sensitive to noise in camera pose estimations.

In this chapter, we analyzed the performance of our method on a dataset from the AirSim Africa Safari Environment, and we analyzed how robust our method is to depth and camera pose noise. In Chapter 6, we will summarize our study and describe possible future directions.

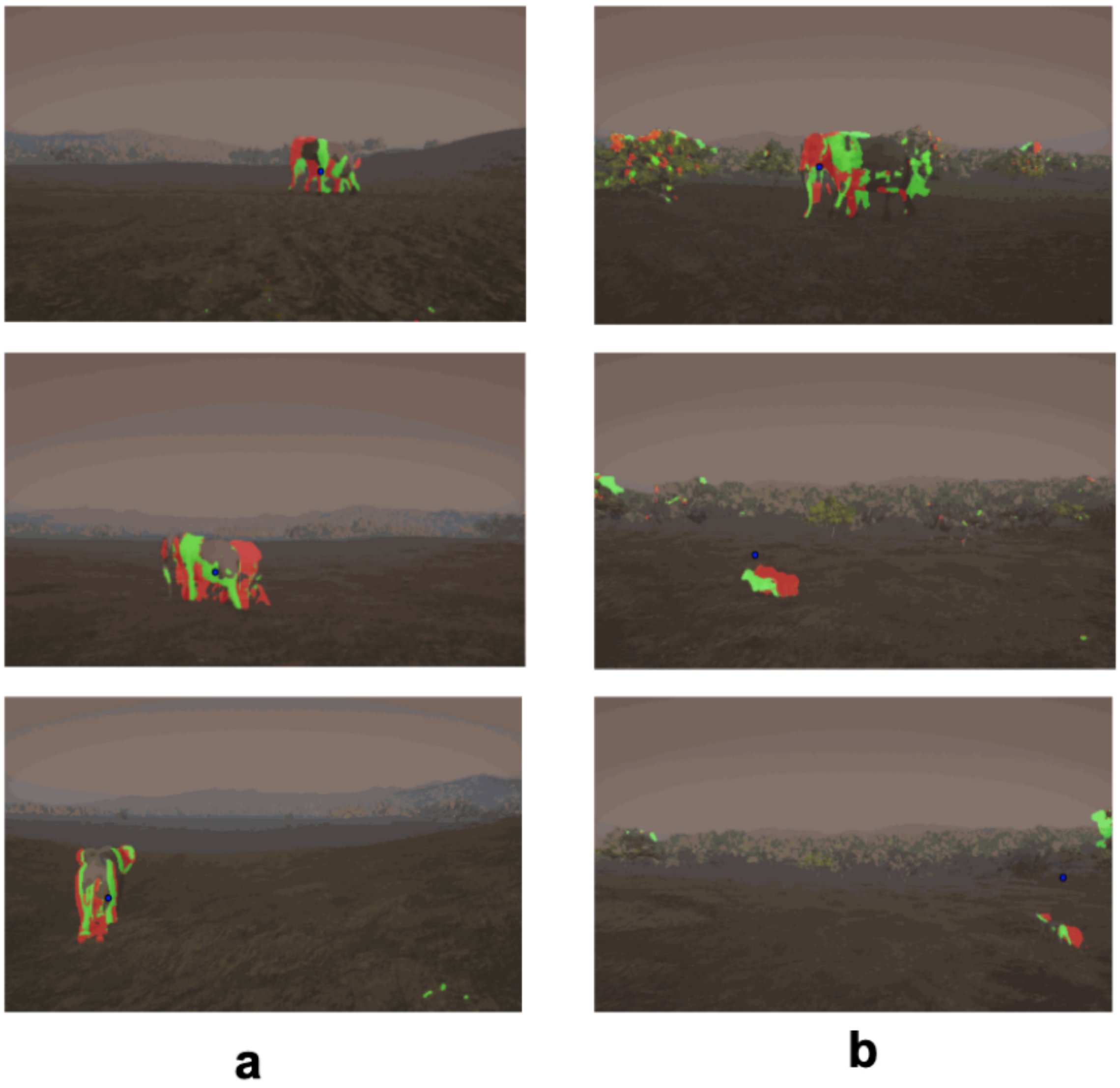


Figure 5-3: Accurate and Inaccurate Examples of Deviation Maps. (a) shows examples of deviation maps calculated from our dataset that accurately segment the moving object in the scene. (b) shows examples of deviation maps calculated from our dataset that do not accurately segment the moving object in the scene (and consequently the centroids would be incorrect). The standard deviation method of measuring deviation was used in these examples.

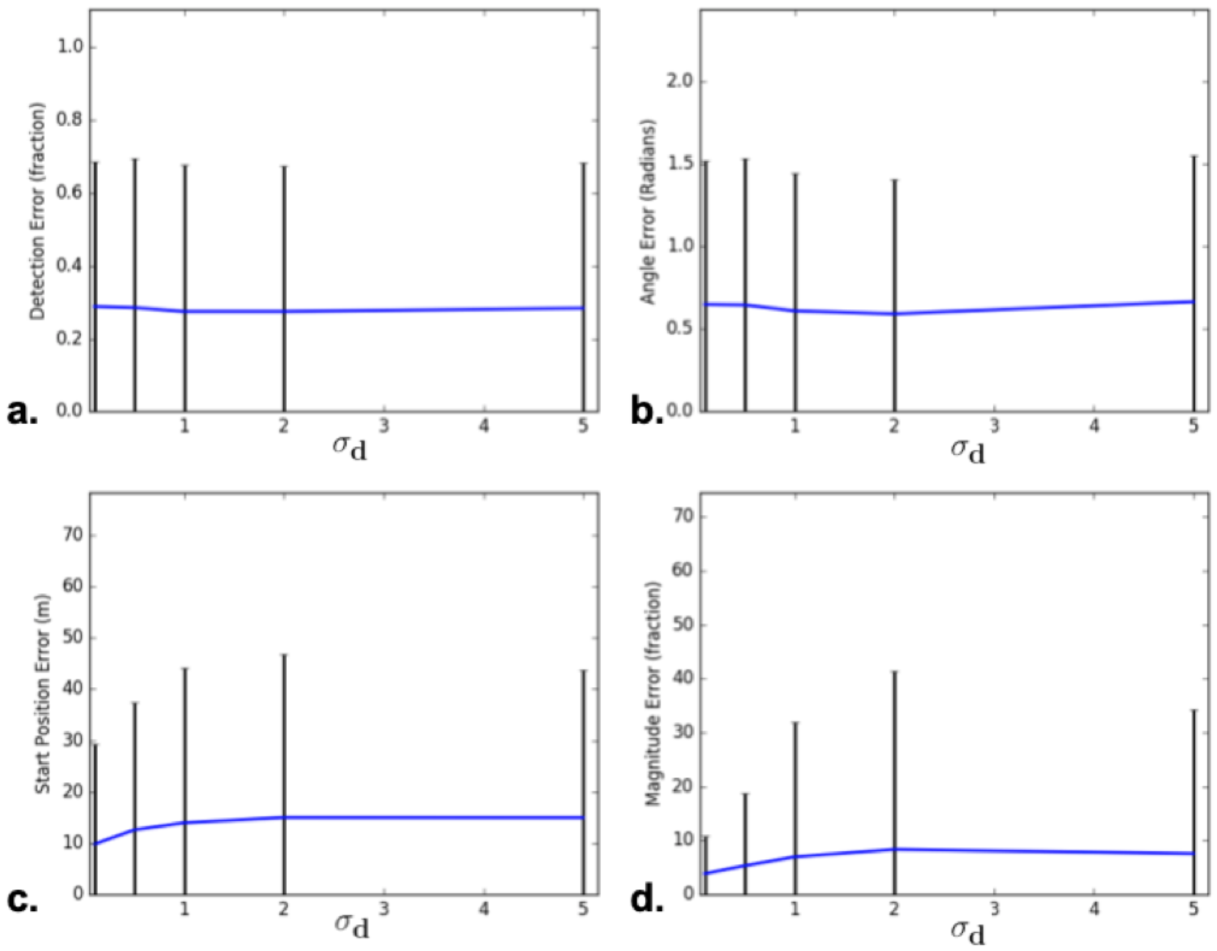


Figure 5-4: Performance of *TrackObject* using standard deviation for measuring deviation with varying depth noise. Graph a shows detection error over varying σ_d. Likewise, Graph b. shows angle error, Graph c. shows start position error, Graph d. shows Magnitude Error. Error bars represent +/- 1 z-score from the mean, and the values plotted are means. $\theta = 0.10$ is used for the deviation threshold.

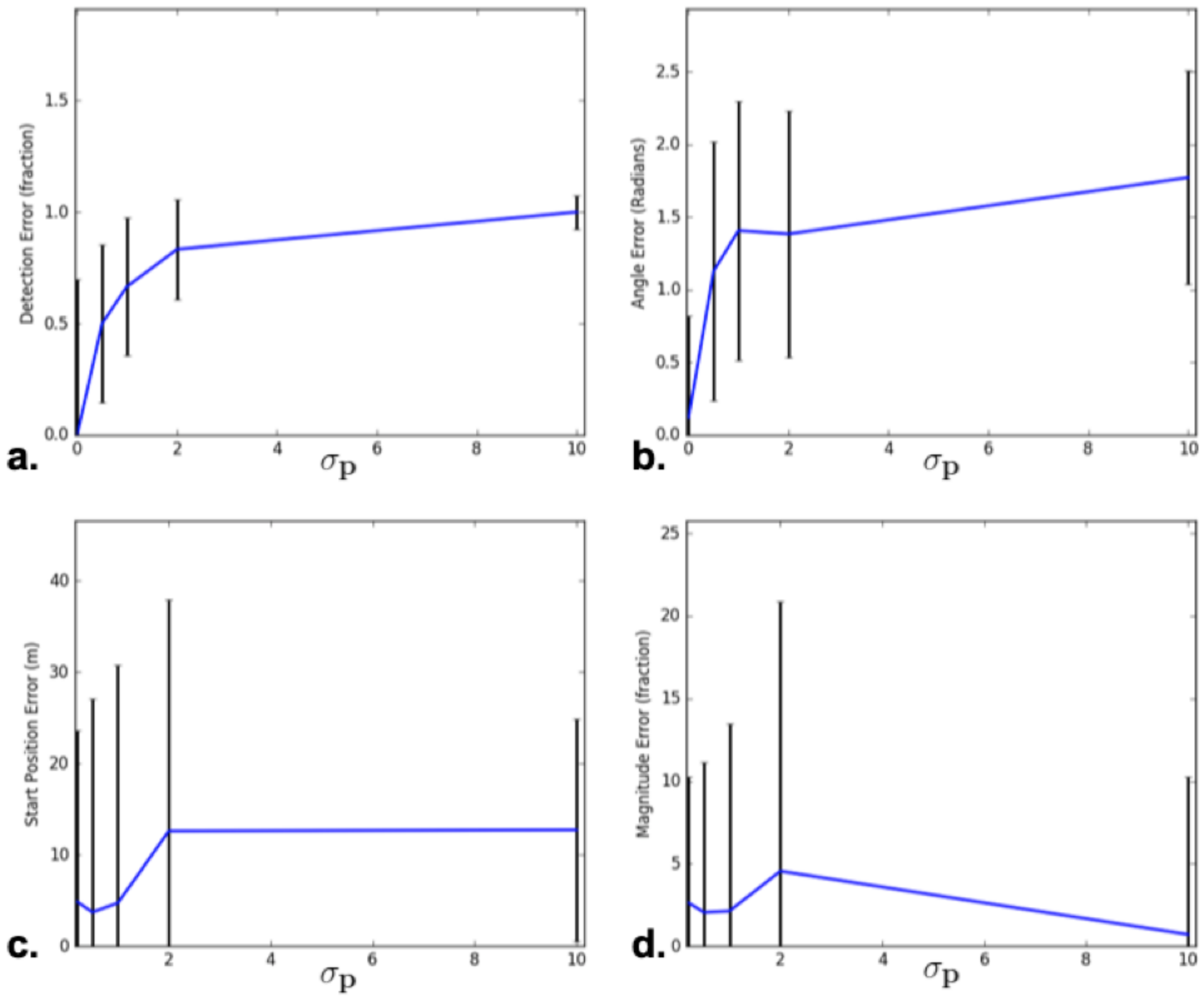


Figure 5-5: Performance of *TrackObject* using standard deviation for measuring deviation with varying pose noise. Graph a shows detection error over varying σ_p. Likewise, Graph b. shows angle error, Graph c. shows start position error, Graph d. shows Magnitude Error. Error bars represent ± 1 z-score from the median, and the values plotted are medians. $\theta = 0.10$ is used for threshold for deviation.

Chapter 6

Conclusions

6.1 Contributions

This thesis proposed a method of detecting and tracking moving objects with a moving camera using a low-computation pixel-deviation based algorithm. We demonstrated the method's ability to detect and track single animals in scenes from a simulation environment with translatory camera motion. Our best method has a median of 0.0 for moving object detection error, which indicates that over the scenes in the dataset, the median performance of our method is the case where the moving object is detected correctly in all images in the scene. For our estimated motion vector, our best method has medians of: 0.1 radians of angle error, 0.93 meters of start position error, and 0.25 magnitude error (proportion). There is significant noise in these 4 quantities. Although the standard deviation method of measuring deviation had the lowest calculated median errors, the confidence intervals of those quantities overlap with that of Gaussian distribution percentile, and a 2-sample t-test provides no significant difference in performance between the methods of measuring deviation. Therefore, no strong conclusion can be made comparing the performances of the methods of measuring deviation.

While it is unclear how sensitive our method is to error in depth input, our method's performance is significantly sensitive to error in given camera poses. This observation indicates that the system might have applications in scenarios where ground truth camera pose can be determined accurately but depth input may be significantly inaccurate.

Having pixel-based methods of motion detection and tracking can prevent real-time bottlenecks that can arise from using more sophisticated intelligent models, even with recent advances in machine learning infrastructure. This study shows that analyzing pixel values in images can provide valuable information about the semantics of the environment.

6.2 Restrictions to release in Future Work

Our study used some restrictions on our data to focus our project and evaluate our proposed pipeline. For use in more types of scenes, these restrictions need to be released:

- Our dataset only contains forward translatory camera motion
- For each example scene, there is only one moving object
- The moving object does not change elevation, and the camera's z-axis is parallel to the ground. Therefore there is no significant change in the objects global Y-coordinate or local camera y coordinate in a scene.

6.3 Directions of Future Work

There are many possible enhancements that can be made to the *TrackObject* pipeline, mainly when dealing with real-life input.

The method is currently sensitive to lighting conditions in the environment, since lighting conditions affect the RGB pixel values, and therefore the grayscale pixel values. Perhaps a better method of analyzing pixel values is to convert them from RGB to the HSV (Hue-Saturation-Value) scale. We can then compare pixel hues to determine deviation, a process that is independent of brightness disparities from lighting. Note that in some cases, lighting increases the pixel deviation caused by moving objects since the camera can see visible reflections of light from said objects. Thus, it is unclear whether switching to a hue comparison would enhance performance.

To incorporate being able to detect multiple moving objects in scenes, one can leverage clustering algorithms. For a given deviation map, one can cluster the pixels with deviation into k clusters, where k is an estimate of the number of moving objects in the scene. This method can encounter difficulties in images when moving objects are occluding one another, since it will become difficult to differentiate the pixels with deviation between the objects.

The motion detection and tracking method we propose based on the two methods of measuring pixel deviation shows promise as a possible future module for the second step in the visual navigation stack described in Figure 1-1. Using pixel deviation as a method of object detection and tracking can open up a wide range of possibilities in the autonomy space.

Bibliography

[1] Paden Tomasello, Sammy Sidhu, Anting Shen, Matthew W. Moskewicz, Nobie Redmon, Gayatri Joshi, Romi Phadte, Paras Jain, and Forrest Iandola. Dscnet: Replicating lidar point clouds with deep sensor cloning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[2] A Filgueira, H González-Jorge, S Lagüela, L Díaz-Vilariño, and P Arias. Quantifying the influence of rain in lidar performance. *Measurement*, 95:143–148, 2017.

[3] Li Jian and Li Xiao-min. Vision-based navigation and obstacle detection for uav. In *2011 International Conference on Electronics, Communications and Control (ICECC)*, pages 1771–1774. IEEE, 2011.

[4] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, 2018.

[5] Thomas Lu and Tien-Hsin Chao. A single-camera system captures high-resolution 3d images in one shot. *SPIE Newsroom*, 2006.

[6] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European conference on computer vision*, pages 430–443. Springer, 2006.

[7] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. A benchmark for the evaluation of rgb-d slam systems. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 573–580. IEEE, 2012.

[8] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163, 2015.

[9] Hyok Song, In Kyu Choi, Min Soo Ko, Jinwoo Bae, Sooyoung Kwak, and Jisang Yoo. Vulnerable pedestrian detection and tracking using deep learning. In *2018 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–2. IEEE, 2018.

[10] Andrés Michael Levering Hasfura. *Pedestrian detection and tracking for mobility on demand*. PhD thesis, Massachusetts Institute of Technology, 2016.

[11] Kentaro Toyama, John Krumm, Barry Brumitt, and Brian Meyers. Wallflower: Principles and practice of background maintenance. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 1, pages 255–261. IEEE, 1999.

[12] Chang Yuan, Gerard Medioni, Jinman Kang, and Isaac Cohen. Detecting motion regions in the presence of a strong parallax from a moving camera by multiview geometric constraints. *IEEE transactions on pattern analysis and machine intelligence*, 29(9):1627–1641, 2007.

[13] Soo Wan Kim, Kimin Yun, Kwang Moo Yi, Sun Jung Kim, and Jin Young Choi. Detection of moving objects with a moving camera using non-panoramic background model. *Machine vision and applications*, 24(5):1015–1028, 2013.

[14] Wu-Chih Hu, Chao-Ho Chen, Tsong-Yi Chen, Deng-Yuan Huang, and Zong-Che Wu. Moving object detection and tracking from video captured by moving camera. *Journal of Visual Communication and Image Representation*, 30:164–180, 2015.

[15] Ashit Talukder and Larry Matthies. Real-time detection of moving objects from moving vehicles using dense stereo and optical flow. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 4, pages 3718–3725. IEEE, 2004.

[16] Ying Ren, Chin-Seng Chua, and Yeong-Khing Ho. Motion detection with nonstationary background. *Machine Vision and Applications*, 13(5-6):332–343, 2003.

[17] Don Murray and Anup Basu. Motion tracking with an active camera. *IEEE transactions on pattern analysis and machine intelligence*, 16(5):449–459, 1994.

[18] Michael Everett, Yu Fan Chen, and Jonathan P How. Motion planning among dynamic, decision-making agents with deep reinforcement learning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3052–3059. IEEE, 2018.

[19] Yu Fan Chen, Shih-Yuan Liu, Miao Liu, Justin Miller, and Jonathan P How. Motion planning with diffusion maps. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1423–1430. IEEE, 2016.

[20] Joern Rehder, Janosch Nikolic, Thomas Schneider, Timo Hinzmann, and Roland Siegwart. Extending kalibr: Calibrating the extrinsics of multiple imus and of individual axes. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4304–4311. IEEE, 2016.