Robust and Unsupervised Interest Point Detection for Efficient Visual Odometry

Farhan Toddywala
Kristofer Pister, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-64
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-64.html

May 26, 2020

Acknowledgement

I am deeply grateful to my research advisor Professor Kristofer Pister for his enormous support; Kristofer has been extremely helpful in steering this research in the right direction. This thesis would not be possible at all without his continuous support. I would also express my deep thanks to Professor John DeNero for his time in serving as the second reader of this thesis. Finally I would like to thank Lydia Lee, Nathan Lambert and Johnny Wang for their help throughout this process.

Robust and Unsupervised Interest Point Detection for Efficient Visual Odometry

by Farhan Toddywala

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Kristofer Pister
Research Advisor

5/25/2020

(Date)

* * * * * * *

Professor John DeNero
Second Reader

5/23/2020

(Date)

Abstract

Robust and Unsupervised Interest Point Detection for Efficient Visual Odometry

by

Farhan Toddywala

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Kristofer Pister, Chair

Robotic exploration is a desirable goal for small-scale SWARM systems. From practical applications like rendezvous to pursuit evasion, the ability to map out an environment relative to one's own position is of vital importance to SWARMs. Simultaneous Localization and Mapping (SLAM) techniques are well suited to this problem. While traditional SLAM methods like Visual Odometry with the FAST interest point detector generally perform efficiently and well at these tasks, they are insufficient when dealing with the levels of noise we expect to encounter with low-resolution, millimeter-scale, grayscale cameras. Our approach seeks to address this problem through a combination of Control Feedback and Unsupervised Learning techniques. Benchmarks on the KITTI Odometry dataset shows significant gains in settings where images are grossly corrupted by Gaussian noise. Control Feedback techniques alone can provide similar performance to the noiseless setting in these situations; Unsupervised Learning techniques provide similar and sometimes even better performance than FAST while performing fewer instructions in the worst case. Through these techniques, we aim to bring robotic exploration at the scales we would expect for a SWARM system closer to reality.

Acknowledgements

I am deeply grateful to my research advisor Professor Kristofer Pister for his enormous support; Kristofer has been extremely helpful in steering this research in the right direction. This thesis would not be possible at all without his continuous support. I would also express my deep thanks to Professor John DeNero for his time in serving as the second reader of this thesis. Finally I would like to thank Lydia Lee, Nathan Lambert and Johnny Wang for their help throughout this process.

Contents

List of Figures

List of Tables

Acknowledgments

Chapter 1

Introduction

Interest Point Detection has a rich history in the field of Computer Vision. Historically, Interest Point Detection traces its roots back to corner detection, when algorithms were designed specifically to find corners in images and use them as stable image features. Generally speaking, the goal of interest point detection is to provide rich spatial information about the structure of the image in a way which is repeatable, mathematically founded, and stable under perturbations of the image. One of the most important interest point detectors to have been discovered is the Features from Accelerated Segments Test (FAST) [7]. This algorithm provides an extremely efficient way of detecting interest points (specifically corners) in images for tasks which require low-latency such as video processing or real-time feature tracking on robotic systems.

One of the most important applications of Interest Point Detection is the problem of pose estimation. Pose estimation provides a foundation for extrapolating 3 dimensional information from sets of 2 dimensional images. This enables systems to reconstruct the 3 dimensional positions of points within a set of images and determine the relative position and orientations of the cameras which took those images. The latter capability is one of the driving ideas behind Simultaneous Localization and Mapping [3] [1] [6] (SLAM) algorithms, which enable systems to track their environments and positions as they move. SLAM is of particular interest to small-scale, SWARMable robotic systems, in which a group of small robots work together to accomplish larger goals. The ability to explore an environment in an intelligent manner (intelligent exploration) enables more complex planning and coordination with these small robots. At the same time, adapting computer vision algorithms, which traditionally require significant processing power, to these small robots is a challenge. Memory is limited, processing power must be conserved, and cameras are likely to be low-resolution and noisy. This last point is of special importance to my research.

Simultaneously, the field of artificial intelligence has grown exponentially over the last two decades, with significant intersections with the field of Computer Vision. Past work has been done in adapting interest point detection using Machine Learning techniques in both

supervised and unsupervised learning. FAST in particular has a supervised learning extension to improve upon its already impressive efficiency. However, none of these approaches have yet taken into account a low-power system with noisy images, which is necessary in enabling SLAM algorithms for SWARM systems. FAST in particular has a particular weakness to image noise, as we will demonstrate later on. Unsupervised Learning provides a flexible manner of formulating problems from an optimization perspective, giving us control over what features we want to extract from an image and how much computation we want to put into making that calculation. When combined with control feedback techniques, the problem of image noise can be dealt with in an efficient manner with respect to interest point detection for pose estimation.

Having a Robust, SWARM-capable Visual Odometry algorithm has many real-world implications. For example, SWARM tasks like Pursuit Evasion and Rendezvous have been explored through the lens of deep reinforcement learning. These algorithms embed the positional information of each robot into a high dimensional space to formulate an optimal policy [5]. Thus, providing the capability for each robot to calculate a good estimate of its position has significant applications for the future of SWARM robotics.

Chapter 2

Related Work

2.1 FAST

Algorithm Description

FAST [7] is fundamentally a threshold based algorithm. To determine if a pixel is a corner, the algorithm marks the 16 pixels in a "circle" around the candidate pixel as significantly brighter, significantly darker or similar in intensity to the candidate pixel. This classification is determined by a threshold t, i.e. if the intensity p_i of a pixel in the circle centered on a pixel c satisfies $p_i - t > p_c$ then the pixel i is marked significantly brighter, and if $p_i + t < p_c$ then pixel i is marked significantly darker. Otherwise the pixel is marked as being similar in intensity to the candidate pixel. We then look for N contiguous pixels that are either significantly brighter or darker in the circle. $N = 12$ gives us a very fast way to reject non-corners. To avoid adjacent pixels being marked as corners, we use a technique called Non-Maximal Suppression, computing a score function (often a brightness metric such as $\sum_{i=1}^{16} p_i$), and throwing out potential corners adjacent to other potential corners with a higher score. See the figure below (2.1) for a visual aid for this algorithm.

This algorithm has a Machine Learning extension using a Decision Tree Classifier [8]. The algorithm marks each of the 16 surrounding pixels around a candidate as 1 if the pixel is significantly brighter, -1 if the pixel is significantly darker, and 0 otherwise. Then it treats these discrete embeddings as features, and tries to predict when a candidate will be marked as a corner by FAST. This is useful because it can theoretically reduce computation by having fewer than 16 levels in the decision tree. However, in practice, this algorithm is difficult to train because we have a severe imbalance of corners to non-corners; interest points typically take up no more than 1-5% of the pixels in an image (liberally speaking).

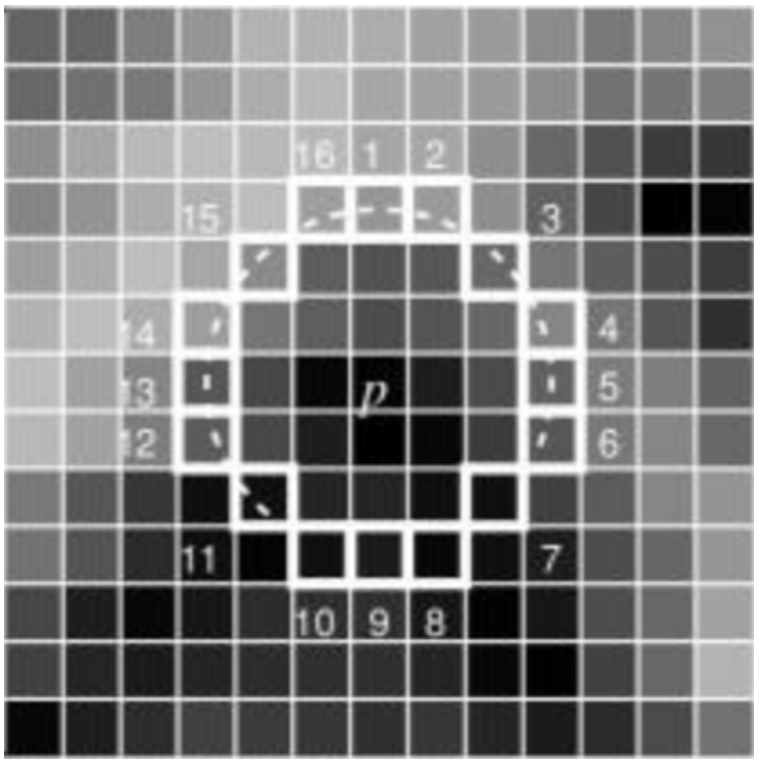

Figure 2.1: A visualization of the FAST algorithm from Rosten et al's original paper[7].

Noise

When noise is added to images, the number of interest points detected by FAST shoots up extremely quickly, which is problematic for any application using the interest point detector. Let us consider a toy example to demonstrate why this phenomenon occurs. Consider a grayscale image which displays a single color, with all pixels having an intensity of 127.

Consider a point p in the image. Clearly, this point in the noise free image cannot be considered a corner since there is no difference between any of the pixels. However, now add gaussian noise z to every pixel's intensity where $z \sim N(0, t)$, where t is the FAST threshold. Then, the probability that any pixel in the 16 pixel circle around p has an intensity no smaller than $127 - t$ is approximately 84%, and the probability that p's intensity is no greater than $127 - 2t$ is approximately 2.5%. Assuming $N = 12$ (i.e. we look for 12 contiguous pixels that are significantly brighter or darker than the center pixel in the surrounding circle), defining $X = k$ to be the a random variable representing the event where the longest chain of pixels with intensity greater than or equal to $127 - t$ is exactly k, we have

$$P(X \geq 12) = P(X = 16) + P(X = 15) + P(X = 14) + P(X = 13) + P(X = 12)$$

$$= 0.84^{16} + 16 \times 0.84^{15} \times 0.16 + 16 \times 0.84^{14} \times 0.16^2 + 16 \times 0.84^{13} \times 0.16^2 + 16 \times 0.84^{12} \times 0.16^2$$

$$\approx 37.7\%$$

Then, the probability that p is marked as an interest point is then

$$2 \times P(X \geq k) \times P(p \leq 127 - 2t)$$

$$\approx 0.377 \times 2 \times 0.025$$

$$= 1.89\%$$

The factor of 2 accounts for symmetry about 127. Then, if we have an image with dimensions of approximately 1200×375 as in our experiments detailed later on in this thesis, we expect approximately 8500 interest points in this new image. Note that this is a loose lower bound on the number of interest points we expect, as to get the true number we would have to approximate an integral over all possible combinations of p's noisy intensity and the intensities of the remainder of the pixels. We simulated this exact scenario and found that on average we got roughly 23000-25000 interest points depending on our choice of t. In summation, our addition of gaussian noise has brought us from 0 interest points in a completely uninteresting, information scarce image to almost 8500 in expectation at minimum! This demonstrates the issue noise brings: additive noise will add a large amount of points which convey little information about the 3D structure of the scene displayed in the image to our pose estimation calculation, resulting in poorer overall performance in trajectory estimation. Even in normal images, the above example demonstrates that patches of the image which are relatively uniform and uninteresting will eventually get marked with many corners as noise is added. The figures below (2.2, 2.3 and 2.4) demonstrate this issue with FAST.

2.2 Visual Odometry

Monocular Visual Odometry [10] (VO) is a SLAM algorithm which maps a set of images taken by a moving camera to a 3D trajectory. The algorithm works as follows. First, an interest point detector (like FAST) is used to generate a number of interest points (pixels) in an image. Next, a subsequent image is taken, and using an Optical Flow algorithm (in our case the Lucas-Kanade algorithm), we obtain an estimate of where those pixels are in the second image. This gives us 2 sets of interest points corresponding to the same "features" in both images. Next, using epipolar geometry, we are able to obtain estimates for the $SO(3)$ rotation matrix R and the R^3 translation vector t, where t represents the translation between the cameras assuming the global coordinate system is framed relative to the first image (we will need to adjust this t to get a global pose estimate). Note that these estimates only provide information about the directions of the positions and orientations of the cameras between both images; scale information cannot be inferred from Monocular Visual Odometry. This can be alleviated through inertial measurements (which can be fused with Visual Odometry to get Visual Inertial Odometry algorithms), or through adding a second camera (which must be spaced some distance apart from the first camera to get

useful distance measurements). For the sake of our experiments, we use the ground truth scale information and focus on improving cumulative VO pose estimations. If the scale of the motion between the two images is greater than some threshold (as we don't want the trajectory to change if the system is not moving), we can simply use the updates

$$p_t = p_{t-1} + s_t \times R_{t-1}t$$

$$R_t = R_{t-1}R$$

to estimate our position and orientation. Here p_t represents our position in 3D space at time t, R_t represents our orientation at time t, and s_t represents the scale factor (how far we moved in real world units) at time t (we use the ground truth for this). Note that $R_0 = I$ and $p_0 = 0$ unless we are given estimates a priori. On subsequent iterations, we use the corresponding points detected by the optical flow algorithm as our set of interest points. At each subsequent image, this number of interest points will drop (as points start to leave the frame or become obscured). Once the number of interest points we are tracking drops below some threshold, we re-run FAST to obtain a new set of interest points and continue. When noise is added to images, this last step often interferes with obtaining a good pose estimate, slowly throwing the trajectory more and more off course, as demonstrated below in 2.5 and 2.6.

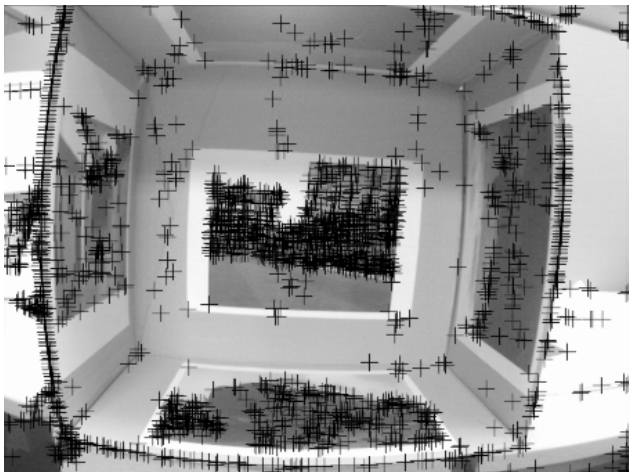

Figure 2.2: An example of corners detected by FAST on a grayscale image. The FAST threshold is set to 10. Black markers denote corners detected by FAST.

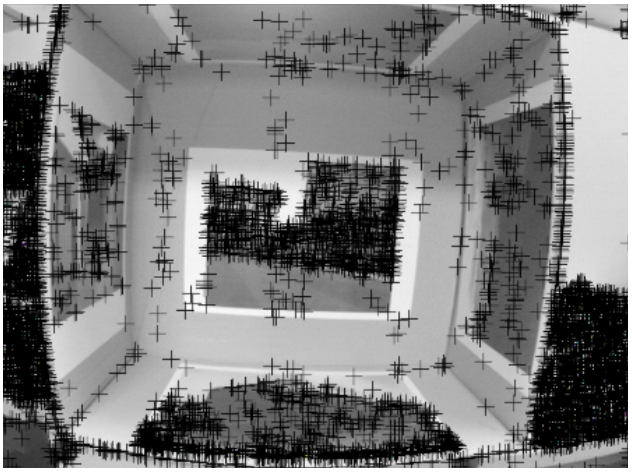

Figure 2.3: An example of corners detected by FAST on a grayscale image corrupted by 0-mean Gaussian noise with a standard deviation of 2 is added to the image. Pixels are rounded to the nearest 8-bit value after noise is added, and capped between $[0, 255]$ inclusive. The FAST threshold is set to 10. Black markers denote corners detected by FAST. Notice that the pixels on the left and right side with high brightness and somewhat uniform color have a massive spike in corners detected. This is partially due to the phenomenon described above, and made worse due to the fact that pixel values are capped to be no greater than 255.

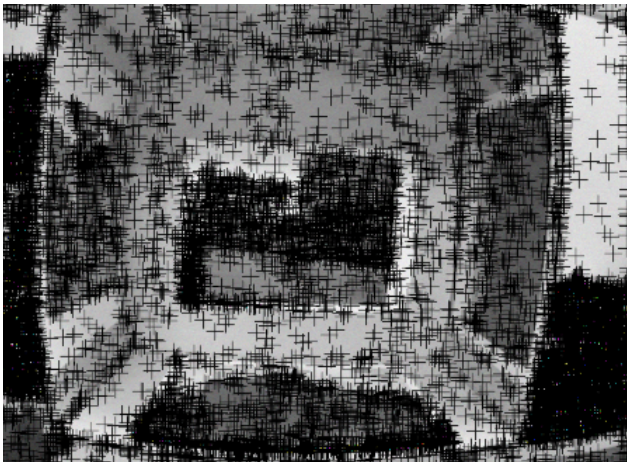

Figure 2.4: An example of corners detected by FAST on a grayscale image corrupted by 0-mean Gaussian noise with a standard deviation of 6 is added to the image. Pixels are rounded to the nearest 8-bit value after noise is added, and capped between $[0, 255]$ inclusive. The FAST threshold is set to 10. Black markers denote corners detected by FAST. The image has significantly more corners detected, many of which clump together. Non-maximal suppression can only mitigate adjacent pixels from being marked as corners, so the clustering effect of corner detections in uniformly colored image segments still occurs.

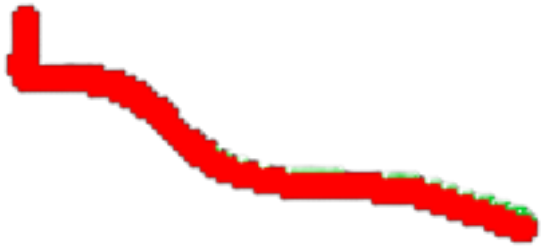

Figure 2.5: KITTI Sequence 3 trajectory using FAST when there is no image noise. In green is the predicted trajectory, and in red is the ground truth trajectory. The green trajectory and red trajectory overlap almost completely, indicating that at 0 noise, FAST provides a very accurate trajectory.

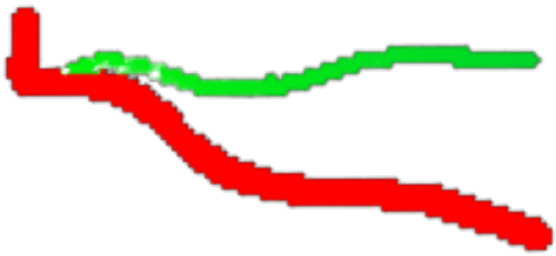

Figure 2.6: KITTI Sequence 3 trajectory using FAST with 0 mean Gaussian noise with a standard deviation of 70. In green is the predicted trajectory, and in red is the ground truth trajectory. The green trajectory and red trajectory stop overlapping very early on, indicating that FAST failed quickly under noise.

Chapter 3

Dynamic Thresholding

3.1 Control Feedback for Interest Point Detection

As we saw in the previous chapter, when the number of interest points detected by FAST increases dramatically under noise, the information conveyed by those interest points starts to become less and less useful. Since the primary issue is that a large number of points which are not at all corner-like start to resemble a corner when noise is added, the natural inclination can simply be to set the threshold of FAST higher. However, even at a fixed threshold, the number of interest points detected in a sequence of images fluctuates greatly, since we don't expect every image to have the same amount of corners. Noise makes this issue even worse, increasing the magnitude of these fluctuations. To improve FAST, we propose a method to dynamically regulate the number of interest points detected without increasing the spatial or computational complexity of the algorithm. Qualitatively, the automatic tuning sets a acceptable range of interest points. If FAST produces more than the upper limit of that range, we set the new threshold to be old threshold multiplied by a constant greater than 1 (rounding to the nearest integer); increasing the FAST threshold will reduce the number of interest points for that portion of the sequence of images. Similarly, if FAST produces fewer than the upper limit of that range, we set the new threshold to be the old threshold multiplied by a constant less than one. Reducing the FAST threshold will increase the number of corners detected. This algorithm can be thought of as a simple control feedback mechanism.

Overall, this mechanism may seem counterintuitive: we are changing a 1 parameter algorithm to a 5 parameter algorithm, where our parameters are the upper and lower limit of the number of interest points we desire, the increasing and decreasing rate at which we change the FAST threshold, and the starting FAST threshold. However, in practice, the choice of these hyperparameters is far more robust and less sensitive than the FAST threshold alone, and we have very good heuristics for them. Experimental observations and tuning suggestions are described below.

Another natural choice of dynamicly modifying the FAST threshold could be an additive

solution, where if the threshold is too high we decrease by a constant value and vice versa instead of reducing by a multiplicative factor. This works too, though for the purposes of our experiments we stuck with the multiplicative strategy because it allowed us to use the same algorithm when dealing with other threshold-based interest point detectors with a different scale of thresholds.

3.2 Hyperparameter selection

In practice, dynamic thresholding is primarily dependent on a good choice of a range of interest points. Some heuristics we can provide are that 1000-1300 to 2000-2200 worked well across many different sequences when the standard deviation of added gaussian noise was less than 60. Additionally, more gross levels of corruption beyond Gaussian noise with a standard deviation of 60 may require this range to be shifted up slightly to somewhere around 1500-2500. The performance increase is only marginal though, with trajectories looking more or less the same visually. A relatively wide range of acceptable interest point counts allows for the threshold to remain stable; changing an already good threshold too often can result in poor performance. The rates of change for the FAST threshold in dynamic thresholding are fairly robust given a good range of interest points, though it is important not to set them to be too large to avoid instability. 1.1 and 0.9 worked well in the experiments we performed, but not much difference was noticed when changed to 1.25 and 0.8. 1.5 and 0.67 provided some benefits on certain sequences, but caused the trajectory to degrade and go off course in several others due to dropping the threshold too low or increasing it too much during crucial frames (i.e. during turns). When dealing with a sequence or additive noise which changes the number of interest points between consecutive images at a very fast rate, these choices may need to be increased, though this is entirely dependent on the sequence and camera. All experiments were performed on the KITTI Odometry dataset. Finally, the choice of a base FAST threshold is the least important hyperparameter to tune, as it gets adjusted relatively quickly. However, on shorter sequences with gross corruptions, it may take longer for the images to re-run the FAST algorithm because there are so many interest points to track. In these conditions, picking a relatively large threshold (roughly 70 worked for us in these situations) can help the algorithm to converge to a relatively stable number of interest points quickly.

Chapter 4

Dynamic Thresholding Experiments

4.1 Experimental Setup

To evaluate the dynamic thresholding algorithm, we will run the visual odometry algorithm on the KITTI Odometry Dataset. This dataset contains roughly 1200×375 8-bit resolution greyscale images[4]. Note that although this sequence has both left and right images for binocular algorithms, we will only use the left (image 0) sequences for our monocular VO setup. We use sequences 3, 6 and 0 to represent easy, medium and hard trajectories which are also realistic in difficulty. The difficulty of a sequence can be defined by 3 characteristics: 1) How sharp/how prolonged turns are in the sequence 2) How far straight the sequence goes on after a difficult turn and 3) How many images the sequence contains. Sequence 3 can be considered easy as it has gradual turns over approximately 800 images. Sequence 6 has 2 full 180 degree turns over approximately 1100 images. Sequence 0 has many large (90+) degree turns over almost 4500 images.

For a baseline comparison, we fix the FAST threshold at 50 for our initial experiments, chosen by cross-validating across accuracy with multiple thresholds. While other smaller thresholds provide marginally better performance with no noise, they fail extremely quickly when even a small amount of noise is added to the images, so this gives FAST a fair chance against the dynamic thresholding algorithm. For Dynamic Thresholding, we pick a range of interest points from 1000 to 2000, and we pick rates of increase and decrease as 1.1 and 0.9 respectively. See Chapter 3 for more information on the choices of these parameters.

We model noise using independent identically distributed gaussian random variables for each pixel centered at 0 with varying standard deviation. After noise is added, pixels are rounded to the nearest integer between 0 and 255, and any pixels outside of this range are set to 0 or 255, whichever is closer. The goal of this experiment is to simulate on-chip variation of scale cameras which may be found on SWARM systems [2].

For our first set of experiments, we vary the standard deviation of the sampled additive Gaussian noise in the range of $[0, 60]$. Note that the noise used in our experiments is higher than most found in scale photography, but could account for other process and computation errors.

With image noise added, we run both FAST and the Dynamic Thresholding algorithms on each sequence. We measure the mean squared error between the predicted trajectory and the given ground truth for each sequence. To account for variation in between runs, we run the sequences 10 times each for every noise setting on both algorithms. This gives us a good estimate of the mean and variance of each algorithm when dealing with noisy images.

In a second set of experiments, we model a sequence in which images become dynamically corrupted as the robot moves–a varying noise level. Similar to a random walk, we update the Gaussian noise standard deviation by adding a random pixel noise shift $X \in \{-1, 0, 1\}$ (discrete uniform distribution) to the current noise standard deviation. The pixel noise is capped between 0 and some upper limit $L \in \{15, 30, 45\}$ to see how performance varies with different ranges of noise intensity during the sequence. This noise parameter starts off at 0 for all sequences. This type of noise is motivated by the observation that the number of interest points detected during a sequence tends to fluctuate depending on lighting conditions and during turns, so it may be helpful for an interest point detector to deal with situations where the clarity of the image changes as the sequence evolves.

4.2 Results and Observations

Visualizing the results proved to be an interesting task, as at relatively higher levels of image corruption, FAST failed completely, resulting in an extremely off-course trajectory. This, of course, shoots the MSE of the predicted trajectories up enormously. To make our results more interpretable, we display the ratio between the MSE of FAST and Dynamic Thresholding, where a ratio above 1 indicates that regular FAST gives a higher error in its trajectory than FAST with dynamic thresholding.

Dynamic Thresholding shows a clear improvement over the standard FAST algorithm in static noise (4.1) and a marginal improvement in dynamic noise (4.2) when noise levels become relatively high (see figures 4.3 and 4.4 for example trajectories). With static Gaussian noise with a standard deviation greater than 20, standard FAST trajectory MSE is on average 23.1, 82.9, and 4.7 times higher for sequences 0, 3 and 6 than the MSE of the dynamic thresholding trajectory. With dynamic Gaussian noise with an upper limit of 30 or greater, the standard FAST trajectory MSE is on average 7.3, 44.5, and 5.5 times higher for sequences 0, 3 and 6 than the MSE of the Dynamic Thresholding trajectory. Importantly, these results show that visual odometry can still be performed in extremely noise heavy situations, which will be required to translate pose estimation to microrobots using low-power, noisy cameras.

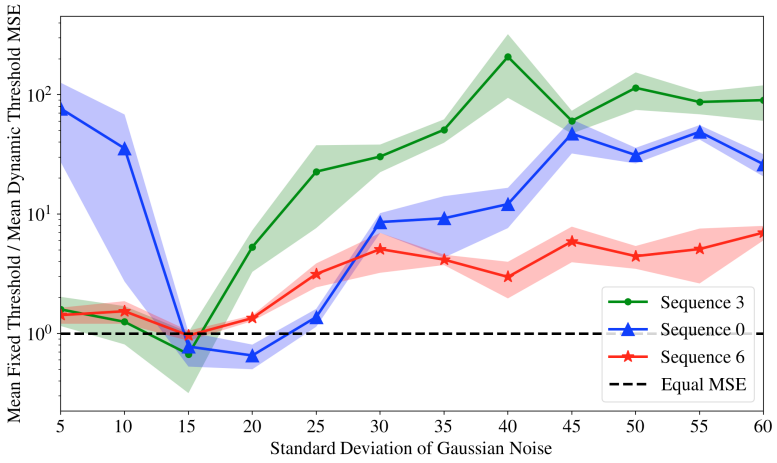

Figure 4.1: **Static Noise**: ratio of standard FAST error over average Dynamic Thresholding FAST MSE ($n = 10$), higher ratio is lower error, with static i.i.d. Gaussian noise of intensities$\in [5, 60]$ on KITTI sequences 0, 3, 6. The lower noise levels are less consistent, but still show an improvement with Dynamic Thresholding in FAST when the noise levels are constant. The dynamics thresholding shows a clear trend of improvement as the noise levels continue to increase beyond $\sigma = 25$. The bars around each line indicate the standard error of the observations.

The extreme performance difference on sequence 0 in the noiseless setting occurs because FAST had a few sequences where it missed a turn completely, resulting in a completely incorrect trajectory. This error is compounded due to sequence 0's extreme length, as mistakes tend to compound immensely in Visual Odometry. This issue is addressed in later experiments in Chapter 6.

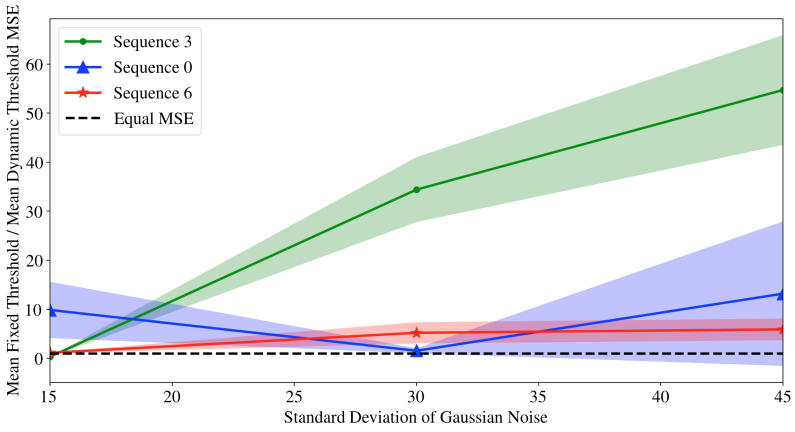

Figure 4.2: **Dynamic Noise**: ratio of standard FAST error over average Dynamic Thresholding trajectory MSE ($n = 10$), higher is better, with dynamically changing additive noise over KITTI sequences 0, 3, 6. The x-axis is the maximum noise level in the dynamic setting, L. There is a reduction in mapping error of up to 50x depending on the noise level and sequence. The bars around each line indicate the standard error of the observations.

Figure 4.3: Trajectory of FAST without dynamic thresholding on sequence 0 when 0 mean gaussian noise with a standard deviation of 60 is added to the images. In green is the predicted trajectory, and in red is the ground truth trajectory. FAST is able to closely follow the until about halfway, where it misses a turn and diverges.

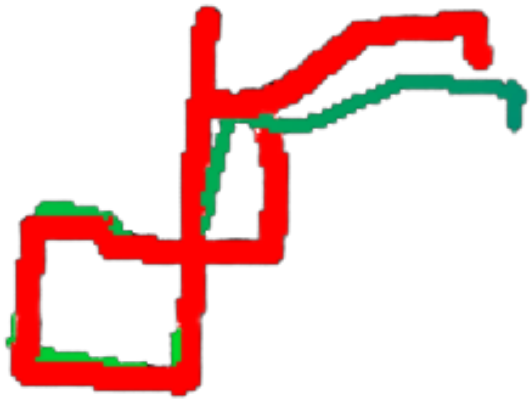

Figure 4.4: Trajectory of FAST with dynamic thresholding on sequence 0 when 0 mean gaussian noise with a standard deviation of 60 is added to the images. In green is the predicted trajectory, and in red is the ground truth trajectory. FAST is able to closely follow the trajectory longer with dynamic thresholding, beginning to diverge after about 80% of the trajectory has completed.

Chapter 5

Unsupervised Learning

In a similar manner to FAST's supervised learning extension, we seek to use unsupervised learning to find a more efficient algorithm for interest point detection which, when paired with the dynamic thresholding algorithm described in Chapter 3, will outperform FAST (also paired with dynamic thresholding).

5.1 Motivation

Unsupervised Learning approaches for interest point detection have been proposed [9], but none perform efficiently enough where they can be considered as a viable alternative to FAST. To do this, we require that the learned interest point detector to focus on a very small number of pixels around a candidate pixel. This leads us to formulate the detector as a sparse learning problem. We will consider a (possibly parametrized) score function f_θ which assigns every pixel a score in R, where the score corresponds to how "interesting" the point is. Ideally, we want the same point in two different (slightly moved and rotated) images to have the same score, since they are both displaying the same point in 3D space.

5.2 Formulation

We can define a set of candidate features around every candidate pixel (ignoring borders, since those pixels often get lost quickly during the optical flow step anyway), where a natural choice is the $n \times n$ block around that pixel. Denote the set of features for point i in 3D space in image 1 as x_1^i and in image 2 as x_2^i. Then, we wish to optimize the function

$$\min_{\theta,W} \lambda||W||_1 + \sum_{i=1}^{m}(f_\theta(W \odot x_1^i) - f_\theta(W \odot x_2^i))^2 + R(f_\theta(W \odot x_1^i)) + R(f_\theta(W \odot x_2^i))$$

Where m is the number of datapoints in our training set, R is a regularization function which can be used to center and control the distribution of pixel scores within an image, and W is a mask which selects which pixels are most important to consider around a candidate pixel.

Sparsity

In our objective function

$$\min_{\theta,W} \lambda||W||_1 + \sum_{i=1}^{m}(f_\theta(W \odot x_1^i) - f_\theta(W \odot x_2^i))^2 + R(f_\theta(W \odot x_1^i)) + R(f_\theta(W \odot x_2^i))$$

We use the L_1 norm to encourage sparsity. The reason we do this is that optimizing to reduce the number of nonzero entries in W would require the use of the L_0 pseudonorm. However, optimizing this norm is intractable. The L_1 norm is the convex hull of the L_0 pseudonorm, so it provides the best possible tractable replacement for the L_0 pseudonorm. Unfortunately, the L_1 norm is not continuously differentiable, so the objective into plugging into a gradient descent algorithm will not work. Moreover, even if a subgradient method is used, it will not necessarily encourage sparsity within a feasible number of iterations, requiring us to decide which weights to set to 0 based on a threshold. Therefore, we will use a proximal gradient descent method, which works by first differentiating our loss function (excluding the L_1 term) with respect to W, obtaining the gradient $\frac{dL}{dW}$. Once we run our gradient step on W, we apply the soft thresholding operator to W, which is defined as

$$soft_\lambda(x) = sign(x) \times \max(|x| - \lambda, 0)$$

To summarize, in order to encourage sparsity in the number of pixels we consider for each candidate interest point, we run the updates

$$Z_t = W_t - \alpha \times \nabla_W f$$

$$W_{t+1} = soft_\lambda(Z_t)$$

5.3 Data Generation

To generate training data, we need to gather pairs of points between pairs of images which represent the same point in 3D space. Once we have these points in both images, we can simply grab the $n \times n$ block of pixels around each point as our features. To do this we rely on epipolar geometry and a ground truth knowledge of the relative pose between each pair of images. If a point p in 3D space, where $p = (\, p_x \quad p_y \quad p_z \quad 1\,)$ is present in 2 images, then we have

$$p_1 = K\,(\, R_1 \quad -R_1C_1\,)\,p$$

$$p_2 = K\,(\, R_2 \quad -R_2C_2\,)\,p$$

where R_i represents the rotation matrix of camera i relative to the global coordinate frame, K represents the camera calibration matrix (which is the same for our purposes since we will be using a single moving camera), and C_i represents the center of the camera relative to

the global coordinate frame. To find p_2 given p_1, we must assume that p is visible to both cameras. Given this, we start by multiplying p_1 by $R_1^{-1}K^{-1} = R_1^T K^{-1}$. This gives us

$$R_1^T K^{-1} p_1 = (\, I \quad -C_2 \,)\, p$$

$$= (\, p_x \quad p_y \quad p_z \,) - C_1$$

Thus,

$$R_1^T K^{-1} p_1 + C_1 = (\, p_x \quad p_y \quad p_z \,)$$

Then,

$$R_1^T K^{-1} p_1 + C_1 - C_2 = (\, p_x \quad p_y \quad p_z \,) - C_2$$
$$= (\, I \quad -C_2 \,)\, p$$
$$= R_2^{-1} K^{-1} p_2$$

Therefore,

$$p_2 = K\, (\, R_2 \quad -R_2 C_2 \,)\, p$$
$$= K R_2\, (\, I \quad -C_2 \,)\, p$$
$$= K R_2 (R_1^T K^{-1} p_1 + C_1 - C_2)$$

Note that the KITTI Odometry dataset gives us a ground truth pose for each image of the form

$$(\, R_i \quad t_i \,)$$

Therefore, we replace C_1 with $-R_1^T t_1$ and C_2 with $-R_2^T t_2$ for an overall equation of

$$p_2 = K R_2 (R_1^T K^{-1} p_1 - R_1^T t_1 + R_2^T t_2)$$

$$= K(R_2(R_1^T K^{-1} p_1 - R_1^T t_1) + t_2)$$

Dividing by the third entry (to avoid floating point issues) and rounding p_2 to the nearest available pixel, we now have 2 pixels in 2 separate images which correspond to each other after moving. In order to simulate noisy conditions, we will add noise to our features in the following manner: first we sample a value s uniformly in $[0, 80]$, and add iid noise $z_1 \sim N(0, s)$ to p_1's block of features. Then we sample a second variable $t \sim N(s, 1)$ and add i.i.d. noise $z_2 \sim N(0, t)$ (rounding both features to be integers in $[0, 255]$). This will allow for us to train on noisy data so that our interest point function will not vary wildly when it encounters noise. Additionally, we make sure the noise added in between a pair of points is roughly, but not exactly the same to simulate dynamic conditions we may encounter like lighting changes.

5.4 SLIPD

Because we are aiming to minimize computation as much as possible, a natural choice of f is to apply the mask W to our features, and simply sum over them (note that in this case f is no longer parametrized and θ can be ignored). What we obtain is a Sparse Linear Interest Point Detector, or SLIPD. If we stack our features for each pair row-wise into matrices X_1 and X_2 respectively, we obtain the following as a part of our objective:

$$\sum_{i=1}^{m}(f_\theta(W \odot x_1^i) - f_\theta(W \odot x_2^i))^2 = ||X_1W - X_2W||_2^2$$

$$= ||(X_1 - X_2)W||_2^2$$

Ignoring the regularization term (for now), we obtain an objective

$$\min_W \lambda||W||_1 + ||(X_1 - X_2)W||_2^2$$

This resembles an singular vector problem with L_1 regularization! All thats left is a constraint on the norm L_2 norm of W. We replace the regularization function R with a unit norm constraint on W ($||W||_2^2 = 1$). This can be solved in several ways. We can obtain a basis using the k singular vectors corresponding to the k smallest singular values of $X_1 - X_2$ and find a sparse vector in that basis (using several Linear Programs), or we can use Projected Proximal Gradient Descent. We use the latter for convenience sake.

Preliminary Results

Preliminary Results for SLIPD were unsuccessful, with the algorithm performing worse than FAST (See figure below). What we took away from this is that f needs to impose some kind of nonlinearity, as a linear function is not expressive enough, our hyperparameters need tuning, and we should probably impose some constraints so that the thresholds we pick have a predictable behavior. This leads us to our more successful new formulation

5.5 Leaky-SLIPD

To address the issues faced by SLIPD we add a few modifications. Firstly, f now applies a nonlinearity of leaky-relu to $W \odot x$ before summing over all dimensions. Here we use 0.5 as our leaky relu parameter. Secondly, we impose a KL divergence penalty R to make the distribution of scores outputted by the function resemble a gaussian. Specifically, over every minibatch, we penalize the negative log probability of the interest point scores as if they were sampled from a unit gaussian. This will make the behavior of the thresholding more predictable. We found that this fixed many of the issues we encountered with SLIPD and provided comparable performance to FAST.

Computational Efficiency

For our experiments (see Chapter 6) we obtained a mask W with 8 nonzero parameters after training Leaky-SLIPD.

Sequential Comparison

First, we consider a comparison of the number of primitive operations (branches, bit shifts, additions and multiplications) necessary for both algorithms without specialized hardware.

For Leaky-SLIPD, consider a general mask which considers a $n \times n$ block of candidate pixels around each pixel, and obtains a mask with a sparsity of k. Then, obtaining a score for each pixel requires 2 steps. First, we center our data. While this would traditionally require dividing by 127.5 and subtracting 1 to get the data in $[-1, 1]$ for ease of training, this first division is computationally expensive. Thus, we replace the division by 127.5 with a division by 128. Our experiments do not show any significant performance differences (that is, outside the margin of error) between these preprocessing steps on any sequences. As an example, we show the differences between these two methods on KITTI Sequence 3 in figures 6.8 and 6.7 in Chapter 6. This requires 1 floating point multiplication (which can be implemented as a bit shift when dividing by 128) and 1 floating point addition operation per pixel. Then, for each pixel, we apply the mask W. This requires k floating point multiplications. Next, we check if any of the nonzero mask entries have a negative output output, and if so we multiply it by 0.5 (right shift by 1 bit), giving us k branching operations, k floating point multiplications, and up to k right shift operations. Finally, we sum over all these values, requiring $k - 1$ floating point additions. To determine if a point is interesting, we compare this final score to a pre-set threshold, requiring 1 branch statement. To run non-maximal suppression, we already have a set of interest point scores, so we only need to compare each possible corner to its adjacent pixels, requiring 8 branch statements. Thus, in total, Leaky-SLIPD requires 17 branch statements, $k - 1$ additions, k multiplications, and k right shift operations in the worst case. For an $M \times K$ image, Leaky-SLIPD can operate on all pixels in the $M - n \times K - n$ sub-image (to avoid edge cases). Thus, on an $M \times K$ image, Leaky-SLIPD requires

$$17(M - n)(K - n)$$

branch statements

$$(k - 1)(M - n)(K - n) + MK$$

additions

$$k(M - n)(K - n) + MK$$

multiplications and

$$k(M - n)(K - n)$$

right shift operations. Note that with vector instructions, we can solve for multiple corners at once (this applies to FAST too). For $k = 8$ and $n = 14$ as we have, this simplifies to

$$17(M - 14)(K - 14)$$

branch statements

$$7(M - 14)(K - 14) + MK$$

additions

$$8(M - 14)(K - 14) + MK$$

multiplications and

$$8(M - 14)(K - 14)$$

right shift operations.

Compare this to FAST. There are several ways to implement FAST, with various heuristics for improving performance. If the hyperparameter $N \geq 12$ (meaning we require 12 or more contiguous pixels in the surrounding circle to be significantly brighter or darker than the center pixel), a pixel can be rejected in very few branching operations by checking whether at least 3 out of pixels 1, 5, 9 and 13 in the circle surrounding a pixel are marked as bright or dark. However, this does not guarantee that any pixel which is not a corner will be rejected in 4 operations. Let us assume that we run this rejection test on every pixel, and then default to the regular FAST algorithm if it passes. Then, we first check whether pixels 1 and 9 are marked as either bright or dark. This takes 4 branch statements. Denote the proportion of pixels in the image which will be rejected at this step as q_1. If so, we check pixels 5 and 13 and see whether any group of 3 of those 4 pixels are marked as bright or dark. This will take 12 branch operations. Denote the proportion of pixels in the image which will be rejected at this step as q_2. Thus, the rejection test will take 4 branch statements to be reject in the first step, and up to 12 in the second step before we run FAST.

To run FAST, we must be able to label each pixel as significantly brighter, darker or in between as compared to the center pixel. Naively, this would require between 1 and 2 addition operations for each of the 16 pixels. However, we can reduce this down to 2 addition operations and store them in registers. First, for pixel p's intensity I_p and a threshold t, we can calculate $t_{bright} = I_p + t$ and $t_{dark} = I_p - t$. Then, we need 16 branch statements to determine which label each of the surrounding pixels gets. However, because we don't care about the order in which the contiguous bright or dark pixels occur, we may need to look at each of the 16 surrounding pixels more than once. Otherwise, we could naively begin counting a "significant" sequence from the middle rather than the beginning. The worst case scenario we could run into is that a sequence begins 1 pixel before our starting point and that the sequence is of exactly N pixels. In this case, we need to check $N - 1$ pixels twice to realize that p is a corner. Additionally, we need to keep track of how long a sequence gets. Assuming $N > 8$, which is a standard for FAST, we would need to run $N - 1$ additions to keep track of the length of the contiguous sequence. This gives us a total of $16 + N - 1$

branch statements and $2 + N - 1$ additions to determine if a pixel is a corner in the worst case. If a pixel is marked as a corner, (and we assume a proportion q_3 of them will do so) we also need to determine a corner "score" to run non-maximal suppression. Assuming that we use a simple corner score like the sum of the surrounding pixel intensities (a widely used choice), we will need 15 more addition operations. Then, to run non-maximal suppression, we need to compare each pixel to all adjacent pixels which are marked as corners, and only mark that pixel as a corner if it has the highest corner score as its neighbors. This will take 8 branch statements. Thus, in the worst case, FAST requires $23 + N$ branch statements and $16 + N$ additions. For an $M \times K$ image, FAST can operate on all pixels in the $M - 6 \times K - 6$ sub-image (to avoid edge cases). Incorporating our rejection test, on an $M \times K$ image, FAST requires

$$(M-6)(K-6)((1-q_1-q_2-q_3)(15+N+4+12)+4q_1+q_2(4+12)+q_3(23+N+4+12))$$

$$= (M-6)(K-6)((1-q_1-q_2-q_3)(31+N)+4q_1+16q_2+q_3(39+N))$$

branch statements and

$$(M-6)(K-6)((1-q_1-q_2-q_3)(1+N)+2(q_1+q_2))+q_3(16+N))$$

addition statements. For $N = 12$, which is commonly used, then we have

$$(M-6)(K-6)(43(1-q_1-q_2-q_3)+4q_1+16q_2+51q_3)$$

branch statements and

$$(M-6)(K-6)(13(1-q_1-q_2-q_3)+2(q_1+q_2))+28q_3)$$

additions.

While a direct comparison between the two algorithms requires values for q_1, q_2, q_3, it is clear that in the worst case FAST performs almost 3 times as many branch statements (using $N = 12$), and about double the addition statements. However, FAST doesn't need to perform any multiplications or right bit shifts, each of which are performed roughly as often as the additions in Leaky-SLIPD.

Parallel Comparision

Leaky-SLIPD's computations can be done in parallel using specialized hardware. First, when centering the pixels in $[-1, 1]$, all pixels have to run the same operation (divide by 127.5 and subtract 1). This can be parallelized in hardware. Assuming we have h_1 units of hardware to execute this operation, this step gives us a total of $\frac{MK}{h_1}$ additions and $\frac{MK}{h_1}$ multiplications (note: this is in terms of time taken to perform the operation, as the number of operations does not change). Until the summation at the end, all computations performed on the k pixels can be done using separate identical hardware to apply the mask and nonlinearity.

Assuming we have k of these hardware units to multiply by the mask, check whether the result is greater than 0 and if not perform a right shift on its bits, then this operation will take as much time as 1 multiplication, 1 branch statement and 1 bit shift for each pixel processed. Then, summing over the results can also be parallelized assuming k is a power of 2 (which it is in our experimental results). It will take $\log_2(k)$ steps of additions to get the final score. We then need 1 branch operation to determine if the pixel is greater than the predetermined threshold. Non-maximal suppression can also be performed using parallel hardware (we have no edge cases due to padding) since we simply need to check if the corner score of a pixel is greater than all of its neighbors, which will take the equivalent of 2 branch operations in total (1 to compute each individual comparison, and 1 to check the condition given every comparision). Thus, in total, with specialized hardware Leaky-SLIPD can take the equivalent of

$$\frac{MK}{h_1} + (M-n)(K-n)$$

multiplications

$$\frac{MK}{h_1} + (M-n)(K-n)\log_2(k)$$

additions

$$4(M-n)(K-n)$$

branch operations, and

$$(M-n)(K-n)$$

right bit shifts in terms of time with the right hardware. At any given time, the hardware will require $\max(h_1, k)$ multiplications, $\max(h_1, \log_2(k))$ additions, 1 right bit shift and 8 branch statements. For $k = 8$ and $n = 14$ as we have, this simplifies to

$$\frac{MK}{h_1} + (M-14)(K-14)$$

multiplications

$$\frac{MK}{h_1} + 3(M-14)(K-14)$$

additions

$$4(M-14)(K-14)$$

branch operations, and

$$(M-14)(K-14)$$

right bit shifts in terms of time. The hardware will require

$$\max(h_1, 8)$$

multiplications

$$\max(h_1, 3)$$

additions, 1 right bit shift and 8 branch statements

FAST's rejection test can be made much faster using optimized hardware. All combinations of pixels 1, 5, 9 and 13 can be checked at once after comparing to t_{dark} and t_{bright} (which themselves can be computed in parallel). This will take 3 steps of branching operations: 1 to compare to t_{dark} and t_{bright}, 1 to check each of the 4 combinations of 3 pixels, and 1 to check if any of these 4 combinations contains all bright or all dark pixels. Assume the proportion of pixels rejected by this step is $q_1 + q_2$. Next, optimizing for FAST in hardware can be done by noticing that if a sequence of N contiguous pixels contains all bright or all dark pixels, it must have a starting point (unless all the pixels in the circle are bright or dark). Thus, we can check all 32 options in parallel and ignore counting the length of each sequence. This can be done with the same set of branching operations for all 16 starting positions, which check if pixel 1 is bright, and if so is pixel 2 bright and so on (with a counterpart for dark pixels). This will take N branching operations. Once this is done, if the pixel is marked as a corner it will need a corner score (denote the probability of this happening as q_3. This will similarly take $\log_2(16) = 4$ additions, and 2 branch operations to actually run non-maximal suppression. Thus, in total, with specialized hardware, FAST can take the equivalent of

$$(M-6)(K-6)(3(q_1+q_2)+(3+N+2)q_3+(1-q_1-q_2-q_3)(3+N))$$

$$=(M-6)(K-6)(3(q_1+q_2)+(5+N)q_3+(1-q_1-q_2-q_3)(3+N))$$

branch operations and

$$(M-6)(K-6)((q_1+q_2+q_3)+(1+4)(1-q_1-q_2-q_3))=(M-6)(K-6)((q_1+q_2+q_3)+5(1-q_1-q_2-q_3))$$

additions. For $N = 12$, our number of branch statements simplifies to

$$(M-6)(K-6)(3(q_1+q_2)+17q_3+15(1-q_1-q_2-q_3))$$

At any given time, the hardware will require 32 branch operations, and 4 additions.

While again, a direct comparison between the two algorithms requires values for q_1, q_2, q_3, it is clear that in the worst case FAST performs approximately 4 times as many branch statements (using $N = 12$), and about 67% more addition statements. However, FAST doesn't need to perform any multiplications or right bit shifts; for each pixel these operations are only performed twice and once respectively in the worst case for Leaky-SLIPD. In terms of the amount of computation that needs to be done at once, Leaky-SLIPD will need to perform up to $\max(h_1, 8)$ multiplications at once (likely the most intensive step), and FAST will have to perform 32 branch statements at once.

Energy Analysis

To analyze the energy cost of Leaky-SLIPD we will assume a preprocessing step such that the algorithm boils down to mainly multiply-accumulate (MAC) operations. This step will, for each of the surrounding "feature" pixels, compare the sign of the centered pixel intensity

and its corresponding weight during each Leaky-SLIPD operation (for a given candidate interest point). Then, if the signs do not match the pixel's centered value will by divided by 2, amounting to a bit shift (this step will essentially perform the bit shift before applying the mask). We do nothing if the signs do match. Finally, the (possibly bit shifted) centered pixel values are multiplied by their mask value and summed over to get the final Leaky-SLIPD score for the candidate interest point. This operation will amount to k MAC operations per candidate interest point (where k is the sparsity of the mask). Then, for an $M \times K$ image where we consider an $n \times n$ block of candidate "feature" pixels, this translates to

$$k(M-n)(K-n)$$

MAC operations. If we choose to use a MAC operation to center each pixel instead of using a separate bit shift and addition to divide by 128 and subtract 1, we get

$$MK$$

extra MAC operations. Assuming each MAC costs 1 pJ, then for $k = 8$, $n = 14$ (as in our learned mask) and $M = K = 128$ this translates to

$$8(128-14)(128-14)(1 \times 10^{-12})$$

$$\approx 0.104 \mu J$$

As an energy cost for Leaky-SLIPD on the entire 128×128 image. Assuming we also use MAC's for the preprocessing steps, we get

$$(8(128-14)(128-14) + 128^2)(1 \times 10^{-12})$$

$$\approx 0.1204 \mu J$$

As an energy cost for Leaky-SLIPD on the entire 128×128 image. We do not provide a direct comparison to FAST here because FAST does not use MAC's in its implementation.

Chapter 6

Unsupervised Learning Experiments

6.1 Experimental Setup

To evaluate Leaky-SLIPD, we follow a very similar procedure as in our dynamic thresholding experiments. We use the same sequences from the KITTI Odometry Dataset.

First, we train Leaky-SLIPD as we described in chapter 4. To do this, we use $\lambda = 0.005$, a learning rate of 0.1 and 30 epochs. Additionally, we divide our pixel values by 127.5 and subtract 1 to center our data in the range $[-1, 1]$. Our results give us a mask W with 8 non-zero entries.

When using dynamic thresholding, both FAST and Leaky-SLIPD will use the same range of interest points of 1300-2100 (this is slightly higher than in previous experiments as we found that we got a closer average number of interest points detected between both algorithms; performance overall was comparable to 1000-2000). Both algorithms will use the rates of 1.1 and 0.9 to adjust their thresholds. Finally, Leaky-SLIPD will have a base threshold of 1.0 and FAST will start off at 70 (we increase this because it allows FAST and Leaky-SLIPD to start off at and converge to nearly the same number of interest points in roughly the same amount of time). We checked to make sure that no experiment had one algorithm detecting significantly more or fewer interest points than the other so that results were more directly attributable to the algorithms themselves. To avoid extremely large and hard to interpret MSE numbers from our previous experiments, we change our evaluation metrics slightly. Firstly, we truncate sequence 0 at 1500 images so its evaluation metric is more comparable to the other 2 sequences (as early errors will result in very high deviations later on even if the algorithm doesn't make any further mistakes) and to speed up testing. Secondly, we modify the evaluation metric from mean squared error to mean euclidean error (MEE), which can be related as $MEE = \sqrt{n \times MSE}$. To get the MEE ratio from our previous graphs, we would simply need to take the square root of the values on the y axis (since the factor related to the number of images cancels), which implies that this metric follows the same pattern as we saw earlier, albeit with somewhat smaller ratios. Together,

these steps will mitigate the penalty that an algorithm pays if it goes off course in a run, reducing the impact of outliers we saw in the previous experiment. We take the average of 10 runs for each algorithm for every level of noise and sequence.

We also provide a comparison here of both algorithms (Leaky-SLIPD and FAST) without dynamic thresholding. We use a a base threshold of 50 for FAST and 0.7 for Leaky-SLIPD since we found these to give roughly the same number of interest points and perform well.

We test here using the same static noise setup from our dynamic thresholding experiments, except this time we consider gaussian noise standard deviations in the range of $[0, 80]$. See 6.1, 6.2 and 6.3 below.

We modify our setup from earlier regarding dynamic noise slightly. First, we consider upper limits on the standard deviation of the gaussian noise in the range of $[20, 80]$. Next, we start off the noise standard deviation at half the upper limit for each test. The reason for this is so that the average noise experienced during each test during the beginning of the trajectory is different for different upper limits. The rest of the setup (minus changes already stated) are the same as in chapter 4. See 6.4, 6.5 and 6.6 below.

6.2 Results and Observations

Static Noise

When dynamic thresholding is not used, the static noise experiments show FAST and Leaky-SLIPD to be comparable in performance, having many overlaps. When dynamic thresholding is used, for all 3 sequences, Leaky-SLIPD outperforms FAST at 0 noise. For sequence 6, FAST consistently outperforms Leaky-SLIPD on average at all levels of noise except for 0. For sequence 3, Leaky-SLIPD generally outperforms FAST on average for lower noise levels (≤ 60 standard deviation), and the FAST marginally outperforms Leaky-SLIPD on average when noise levels exceed 80. On sequence 0, Leaky-SLIPD consistently outperforms FAST on average. Additionally, FAST has significant outliers where it completely fails at certain turns, resulting in extremely high MEE (causing the large standard errors we see in the graph). Overall, Leaky-SLIPD seems at least comparable to FAST, much better in some cases and somewhat worse in others. The sequence 6 issues are interesting in that both algorithms fail to properly make the 180 degree turns when noise is present, but FAST's failures are lesser in magnitude, perhaps indicating that Leaky-SLIPD has a marginal disadvantage when it comes to sharp, prolonged turns. However, the other 2 dataset show that Leaky-SLIPD handles both gradual turns and sharp turns (< 180 degrees) marginally better than FAST. Furthermore, these results are consistent with the dynamic thresholding results found earlier, with dynamic thresholding outperforming its static threshold counterpart signficantly (albeit with a less extreme difference due to the experimental design changes we made). Generally,

the performance increases found with dynamic thresholding begin to occur when the standard deviation of the gaussian noise is between 20 and 40. These gains are significant, as they typically provide similar, if performance as the normal interest point detectors did at 0 noise.

Dynamic Noise

The results for Dynamic noise were somewhat mixed. On Sequence 0, Leaky-SLIPD generally outperformed FAST on average until the noise upper limit was 80. At lower noise levels, SLIPD without dynamic thresholding provided the best performance. Leaky-SLIPD benefitted from dynamic thresholding when the upper limit of the noise exceeded 60, performing the best out of all 4 algorithms at the noise upper limit of 60. FAST here did not benefit from dynamic thresholding as consistently as much as in Chapter 4. This result is not as surprising, as we significantly cut down on the amount of sequence 0 we used to save time and avoid major outliers. Notably, FAST with dynamic thresholding provided the best performance at our highest level of noise.

On sequence 3, without dynamic thresholding, Leaky-SLIPD outperformed FAST on average across the board. However, both algorithms had increasing failure rates for higher upper limits on noise (i.e. trajectories with very high MEE), which is reflected by their standard error bars. When dynamic thresholding is used Leaky-SLIPD consistently outperformed FAST on average, though the performance difference is closer at higher noise levels. Standard Error's with dynamic thresholding did not increase dramatically as noise levels increased. This is consistent with findings in chapter 4, though the difference in performance between dynamic and static thresholds is not as extreme. This is likely due to changes in how we conducted our experiments (described above).

On Sequence 6, FAST generally outperformed Leaky-SLIPD on average. Dynamic thresholding provided FAST with significant performance advantages, as well as a lower variance between runs. Leaky-SLIPD benefitted from dynamic thresholding in the noise upper limit range of $[40, 60]$.

Takeaways

Overall, it seems that dynamic thresholding provides the best performance increases when roughly the same level of noise is added to images in a sequence on both algorithms, which is more along the lines of the noise we expect to encounter due to millimeter scale cameras. Additionally, Leaky-SLIPD provides some notable performance advantages over FAST on Sequence 0 and 3, but not on Sequence 6. This is due to the 180 degree turns encountered on sequence 6; when noise is relatively high, both algorithms overshoot the 180 degree turns, resulting in compounding error, but Leaky-SLIPD consistently overshoots by more than FAST. Thus, if a SWARM robot is navigating with very sharp and prolonged turns, FAST

may be more appropriate, and if it is navigating with smoother and less prolonged turns, Leaky-SLIPD may provide better performance.

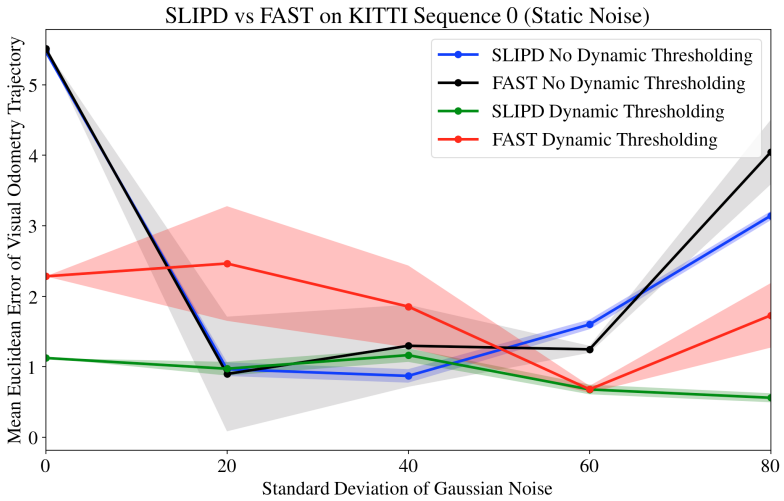

Figure 6.1: **Static Noise**: MEE of FAST vs Leaky-SLIPD, with and without dynamic thresholding as static noise standard deviation increases on KITTI sequence 0. Leaky-SLIPD with dynamic thresholding shows a clear advantage over FAST except at noise std = 60. It also has a consistently lowe standard error at each noise level. Without dynamic thresholding, FAST and SLIPD perform roughly the same. The bars around each line indicate the standard error of the MEE.

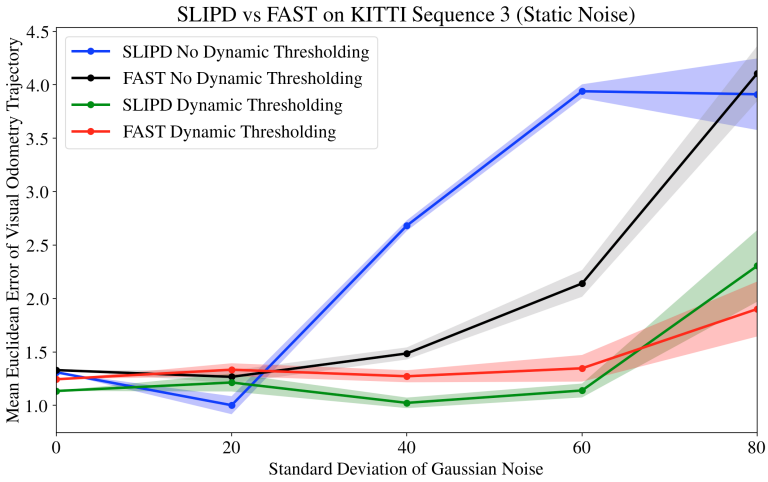

Figure 6.2: **Static Noise**: MEE of FAST vs Leaky-SLIPD, with and without dynamic thresholding as static noise standard deviation increases on KITTI sequence 3. Leaky-SLIPD with dynamic thresholding shows a clear advantage over FAST until noise std of 80. Without dynamic thresholding, Leaky-SLIPD performs a bit better at low noise levels, and worse at higher noise levels. The bars around each line indicate the standard error of the MEE.

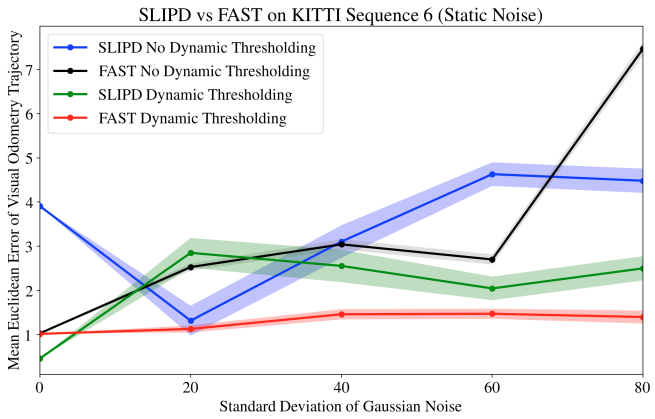

Figure 6.3: **Static Noise**: MEE of FAST vs Leaky-SLIPD, with and without dynamic thresholding as static noise standard deviation increases on KITTI sequence 6. FAST with dynamic thresholding shows an advantage over Leaky-SLIPD for all noise levels (though it loses at 0 noise). Without dynamic thresholding, the two algorithms perform roughly the same, alternating which is better at each noise level. The bars around each line indicate the standard error of the MEE.

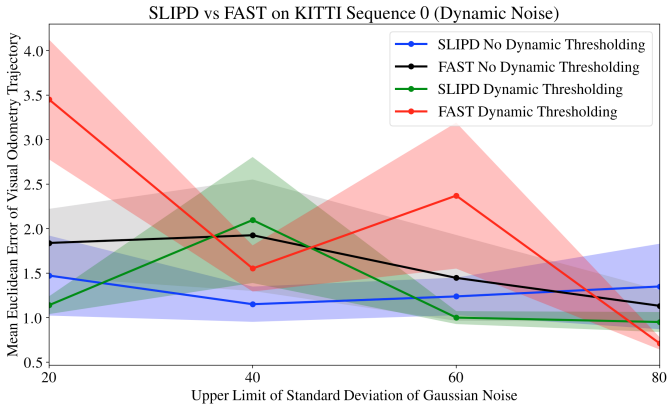

Figure 6.4: **Dynamic Noise**: MEE of FAST vs Leaky-SLIPD, with and without dynamic thresholding as dynamic noise standard deviation upper limit increases on KITTI sequence 0. Leaky-SLIPD with dynamic thresholding shows a marginal advantage over other algorithms when noise upper limit is at 20 and 60, and Leaky-SLIPD without dynamic thresholding performs best in between at a noise upper limit of 40. FAST with dynamic thresholding takes a slight advantage over Leaky-SLIPD with dynamic thresholding when the noise upper limit is 80. Standard errors are relatively high as all algorithms suffered from some failure cases with high trajectory errors (dynamic thresholding on SLIPD and FAST with and without dynamic thresholding suffered the worst in this respect).

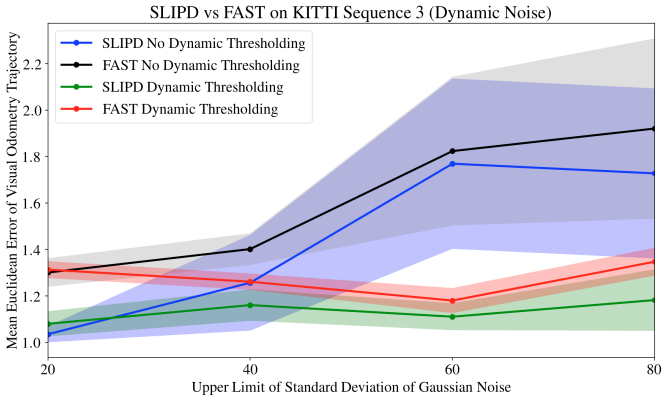

Figure 6.5: **Dynamic Noise**: MEE of FAST vs Leaky-SLIPD, with and without dynamic thresholding as dynamic noise standard deviation upper limit increases on KITTI sequence 3. Leaky-SLIPD with dynamic thresholding performs the best on average at all noise level upper limits except 20 (where regular Leaky-SLIPD marginally outperforms it on average), closely followed by FAST with dynamic thresholding. Leaky-SLIPD without dynamic thresholding marginally outperformed FAST (also without dynamic thresholding) on average, though both had high standard errors due to trajectory failures at higher noise levels.

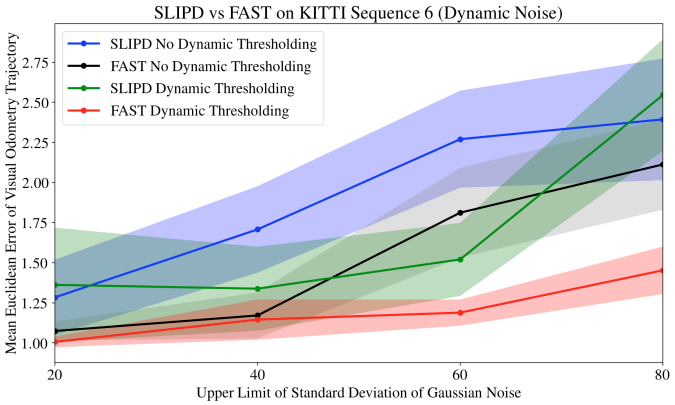

Figure 6.6: **Dynamic Noise**: MEE of FAST vs Leaky-SLIPD, with and without dynamic thresholding as dynamic noise standard deviation upper limit increases on KITTI sequence 6. FAST shows an advantage over Leaky-SLIPD for all noise levels. FAST With dynamic thresholding provides an advantage over vanilla FAST throughout. Dynamic thresholding improves Leaky-SLIPD in the noise upper limit range of $[40, 60]$, and is marginally worse at 20 and 80.

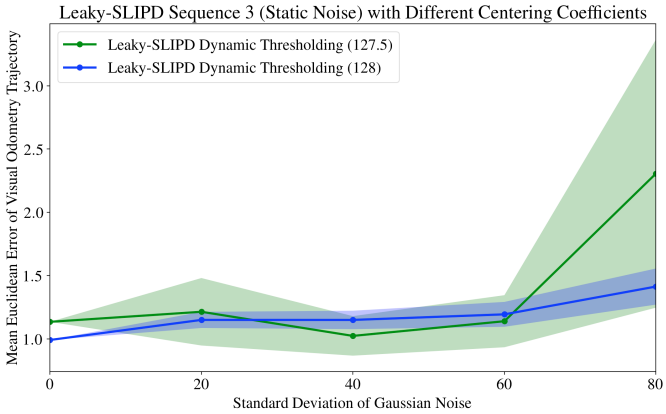

Figure 6.7: **Static Noise**: MEE of Leaky-SLIPD with dynamic thresholding with different centering coefficients when noise is static. In green is the performance of Leaky-SLIPD when all pixels are centered by dividing by 127.5 and subtracting 1. In blue is Leaky-SLIPD when all pixels are centered by dividing by 128 and subtracting 1. Notably, there is no significant performance gain or loss by changing this division constant, as the performance is nearly equal (with neither having a consistently lower mean error) and always within the margin of error (even at a noise level of 80, where the means somewhat differ).

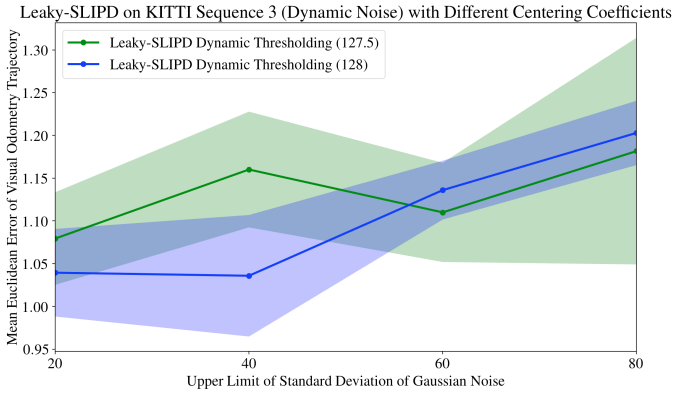

Figure 6.8: **Dynamic Noise**: MEE of Leaky-SLIPD with dynamic thresholding with different centering coefficients when noise is dynamic. In green is the performance of Leaky-SLIPD when all pixels are centered by dividing by 127.5 and subtracting 1. In blue is Leaky-SLIPD when all pixels are centered by dividing by 128 and subtracting 1. Notably, there is no significant performance gain or loss by changing this division constant, as the performance is nearly equal (with neither having a consistently lower mean error) and always within the margin of error.

Chapter 7

Conclusions

In this paper, we have presented a novel, highly efficient method for Unsupervised Interest Point Detection, as well as a method for regulating the aforementioned algorithm (as well as FAST) to be viable under extremely noisy settings, like the ones we would expect with millimeter scale cameras. Specifically, we observed and exploited the fact that a specific range of a number of interest points provides good and robust performance to come up with a dynamic thresholding algorithm which can provide great performance even at gross levels of image corruption. Additionally, we observed certain desirable properties of an interest point detector being used for the purposes of Visual Odometry to create an Unsupervised Interest Point Detection algorithm which can perform equally well or better than FAST in many circumstances. With these additions to the Visual Odometry algorithm, small-scale, SWARMable robotic exploration is significantly more viable.

Bibliography

[1] Tim Bailey and Hugh Durrant-Whyte. "Simultaneous localization and mapping (SLAM): Part II". In: *IEEE robotics & automation magazine* 13.3 (2006), pp. 108–117.

[2] Jonathan Choy. "A $10\mu J$/Frame $1mm^2$ 128×128 CMOS Active Image Sensor". MA thesis. University of California, Berkeley, 2003.

[3] Hugh Durrant-Whyte and Tim Bailey. "Simultaneous localization and mapping: part I". In: *IEEE robotics & automation magazine* 13.2 (2006), pp. 99–110.

[4] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012.

[5] Maximilian Hüttenrauch, Adrian Sosic, and Gerhard Neumann. "Deep Reinforcement Learning for Swarm Systems". In: *CoRR* abs/1807.06613 (2018). arXiv: 1807.06613. URL: http://arxiv.org/abs/1807.06613.

[6] Michael Montemerlo et al. "FastSLAM: A factored solution to the simultaneous localization and mapping problem". In: *Aaai/iaai* 593598 (2002).

[7] E. Rosten and T. Drummond. "Fusing points and lines for high performance tracking". In: *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*. Vol. 2. Oct. 2005, 1508–1515 Vol. 2. DOI: 10.1109/ICCV.2005.104.

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

[9] Nikolay Savinov et al. "Quad-networks: unsupervised learning to rank for interest point detection". In: *CoRR* abs/1611.07571 (2016). arXiv: 1611.07571. URL: http://arxiv.org/abs/1611.07571.

[10] Davide Scaramuzza and Friedrich Fraundorfer. "Visual odometry [tutorial]". In: *IEEE robotics & automation magazine* 18.4 (2011), pp. 80–92.