Bachelorarbeit

Optimizing Feature Detection and Keypoint Distribution for Realtime SLAM

Ralph Gelnar

02.10.2019

Beginn: 02.04.2019

Abgabe: 02.10.2019

Anschrift:

Graphische Datenverarbeitung
Cauerstraße 11
91058 Erlangen
Germany

Tel: 09131/85-29919
Fax: 09131/85-29931
E-Mail: ralph.gelnar@fau.de

Optimizing Feature Detection and Keypoint Distribution for Realtime SLAM

Bachelorarbeit im Fach Informatik

vorgelegt von

Ralph Gelnar

geb. am 30.12.1987 in Erlangen

angefertigt am

Department Informatik
Lehrstuhl Graphische Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Darius Rückert

Betreuender Hochschullehrer: Prof. Dr. Marc Stamminger

Beginn der Arbeit: 02.04.2019

Abgabe der Arbeit: 02.10.2019

Contents

Chapter 1

Introduction

Corner detection is an important step in many vision-based computing tasks, such as structure from motion or simultaneous location and mapping (SLAM). In this thesis we want to examine an aspect of feature detection that has not commonly been getting much attention - the selection of the best keypoints to retain after they have been detected.
By optimizing how keypoints are selected we can improve our results without sacrificing much or any performance. Since frame-times that allow for realtime processing of live-video are desirable if not outright required, this computationally inexpensive part of the process warrants some examination.

1.1 Motivation

The variant of SLAM we are specifically taking a look at is sparse indirect SLAM. The system is sparse if only a small fraction of the pixels in the image are used in the mapping process. Indirect means that rather than mapping the pixels themselves we extract features from the image and then map using those features rather than the pixels directly.
In feature based SLAM, identification of these features (also called corners or keypoints) are a vital step in the finding a solution. An estimated trajectory of the camera that made the images is computed by finding points in the images that match onto the 3D map that was constructed using prior images. In order to find these matches, we first need to find distinctive points in the frame. Then we need to define what constitutes a match by having a method to describe each keypoint and afterwards look for one with a similar description in the next frame.
The computationally expensive part of this process is mostly the detection itself. After detection, we select which keypoints we want to use for actual matching. This step is necessary because of a few reasons. Achieving real-time performance is difficult to unfeasible if we have to compute the description for all detected keypoints, which often number in the tens of thousands. We also detect a greatly varying number of corners in different images, while a more stable number is beneficial to matching. In addition, often keypoints are detected in large clusters, which does not add much information to the mapping process. Experimentally

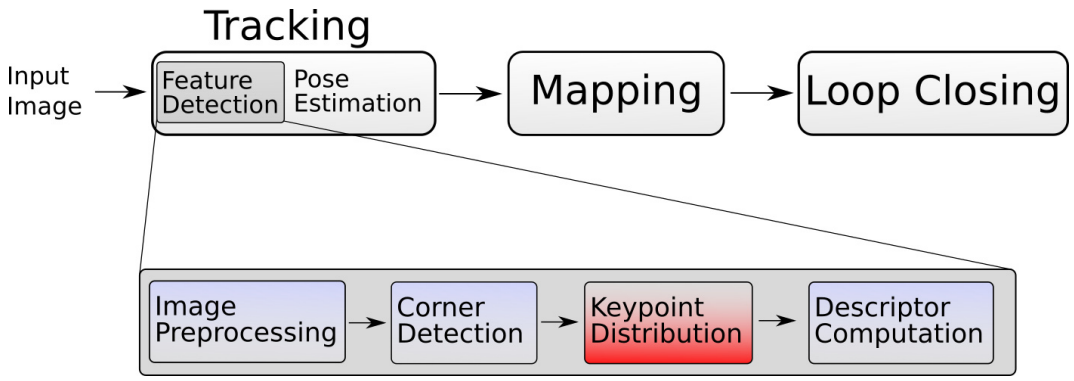

Figure 1.1: Illustration of the workflow of feature-based SLAM. The distribution of the keypoints that takes place as part of feature detection is the focus of this thesis.

we confirmed that the results greatly deteriorate in quality if we try to run our SLAM system while trying to match every detected keypoint.

For this purpose we replace the feature detection in ORB_SLAM2 and try a variety of different algorithms that select and distribute the detected features. The point at which distribution takes place in the pipeline is illustrated in figure 1.1. [1]

[1]The code for our feature detector as well as all distributions presented in chapter 3 is available online on github at https://github.com/darglein/saiga/tree/master/src/saiga/vision/orb.

Chapter 2

Related Work

Detection of corners and distinctive visual features has been the subject of much research. There have been numerous attempts to make detection immune to changes such as rotation, scaling, lighting, noise and others. Though only the works immediately relevant to the feature extractor implemented in this thesis are described in their own section, the Harris corner detector[7], one of the first of its kind, deserves mention for its relevance in the field. In the original version of ORB[13], a Harris corner measure was employed to calculate a cornerness score of the found keypoints.

2.1 SIFT

The Scale-Invariant Features Transform (SIFT)[8] uses repeatedly filtered and scaled images. Keypoints are detected in scale-space, by looking for scale-space extrema using difference-of-Gaussian convolved images. The image is filtered per scale with a Gaussian kernel repeatedly, and the difference-image between each iteration is saved. The image is then scaled down by a factor of 2 and the process repeated. A visualization can be seen in figure 2.1. Extrema are found by looking at immediate neighbours both locally on the same image, as well as at the points in the same respective 3×3 region in the adjacent scales, as seen in figure 2.2(both figures from [8]).
The descriptors of the keypoints are defined by gradients, which measure magnitude and orientation. A orientation histogram is built by measuring differences to neighbouring pixels of all pixels in a region around the keypoint. The resulting values are put into 8 orientation bins and the samples of 4×4 subregions are accumulated. This is done in a 16×16 region around the candidate pixel, resulting in a $4 \cdot 4 \cdot 8 = 128$ element vector per feature.
This detection method makes it possible to detect the same features even when they are nearer or further away and in image-space appear larger or smaller. SIFT also exhibits rotation invariance, as well as robustness to image noise and changes in illumination. It is however difficult to achieve the performance required by real-time SLAM with this system.

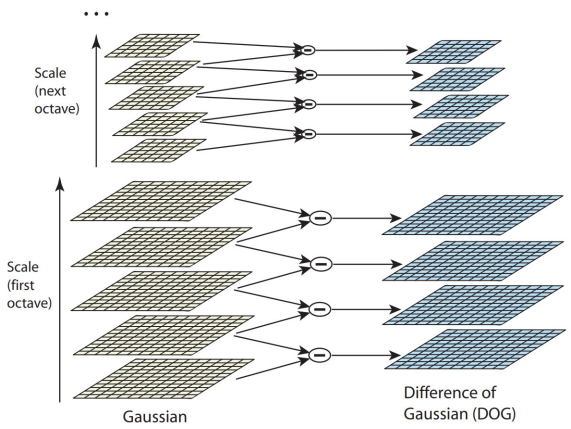

Figure 2.1: The Image is convolved with a Gaussian filter multiple times per octave to produce the DoG images.

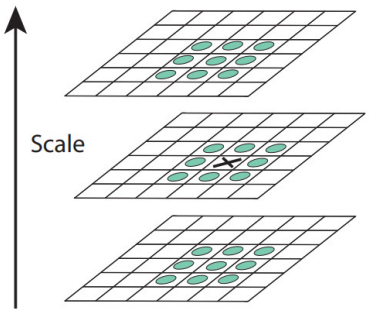

Figure 2.2: Scale-space extrema are detected by looking at all pixels in a 3×3 neighbourhood both at their scale as well as in adjacent scales

2.2 FAST

Features from accelerated Segment Test (FAST) [12] checks for corners per-pixel in the unmodified input image. A pixel is defined as a corner if there exists a continuous circle of n pixels around the pixel being examined that are all either brighter or darker by a certain threshold t than the central pixel, as illustrated in figure 2.3(image taken from [12]).

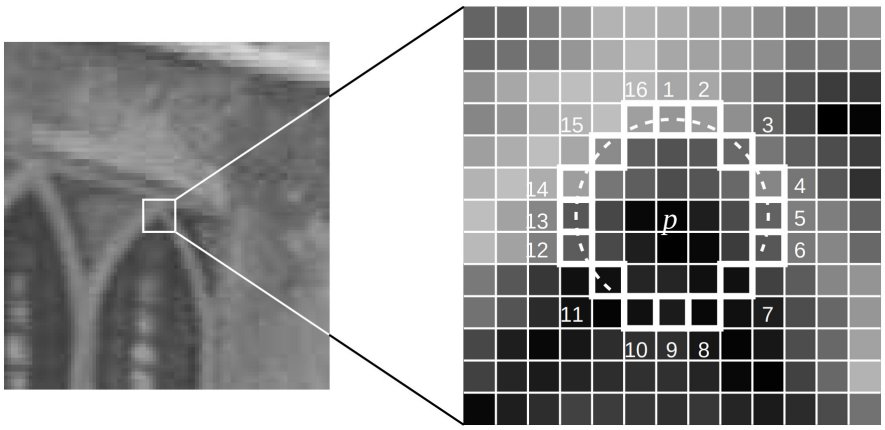

Figure 2.3: Pixel p is a corner if $\exists$ a set S of contiguous pixels in a circle around p with $\forall p_s \in S : I_{p_s} > I_p + t \vee I_{p_s} < I_p - t$.
The continuous pixels that might fulfill this condition (depending on the threshold) in the image are marked by a dotted line.

The circle is Bresenham-interpolated, and, while other sizes have been tried out, a circle of n=16 pixels has been established as the standard. Brief experimentation with different sizes at the beginning of our project have confirmed that n=16 does indeed provide the best results.

To minimize the necessary comparisons the pixels are checked in a way that allows one to exclude the pixel as a corner at the earliest possible moment. As soon as a contiguous circle of pixels that fulfill the required condition is no longer possible the candidate can safely be discarded.
Initially this early test examined the four pixels at positions 1, 9, 5 and 13, since if p were to qualify as a corner, at least three of these pixels would need to be either brighter than $I_p + t$ or darker than $I_p - t$. If any two of these pixels don't fulfill the condition, a contiguous circle of 9 pixels that fulfill the condition is not possible either.
This was then further optimized by machine learning. A decision tree was optimized by examining the entropy of each pixel in a training set of images, ultimately arriving at a fixed order to evaluate all pixels in and discard non-corners as early as possible. Only if this initial check is successful and a continuous circle of pixels either all brighter or darker is possible is the iteration over the circle performed. Now we examine whether there is an actual series of size $\frac{16}{2} + 1$ of contiguous brighter or darker pixels. This way pixels that do not qualify as a corner can consistently be disqualified within the first couple of comparisons, saving computation time.

A cornerness score is also calculated per pixel, via a score function S - for the score function used in our implementation, see chapter 3. This is necessary to perform non-maximal suppression, which avoids excessive cluttering of detected corners. Each corner is only added to the result if there is no neighbouring corner with a higher S.
FAST has some shortcomings, most notably that it is susceptible to image noise, and that it is not by itself invariant to either scale- or rational changes in the image.

2.3 BRIEF

The other important step after recognizing the features in an image is to define a description of a given feature and calculate it for every feature found. This is required so matching features can later be found in subsequent frames. So as to minimize matching time without compromising results a minimal yet discriminative descriptor is desired. Binary Robust Independent Elementary Features (BRIEF)[3] defines a descriptor for keypoints that is fast to compute and match, as well as small to store in memory.
A BRIEF descriptor is a 256-element bit-vector. Each bit is the result of a comparison of two pixels in a region around the described feature. For this purpose a set of 256 (a,b) location pairs are defined and the corresponding bit is set to 1 if $I(a) < I(b)$ and to 0 otherwise. The test locations are Gaussian-distributed with a mean of 0 and a standard deviation of $\frac{1}{25}S^2$ in a $S \times S$ patch around the keypoint, as seen in figure 2.4.
To render the descriptors less vulnerable to image noise, we convolve the image with a Gaussian filter. Each comparison operates with the exact values of two pixels, so noise could affect the result significantly. Since the descriptors can be matched via the Hamming distance of the bit-vectors, matches can be calculated very efficiently.

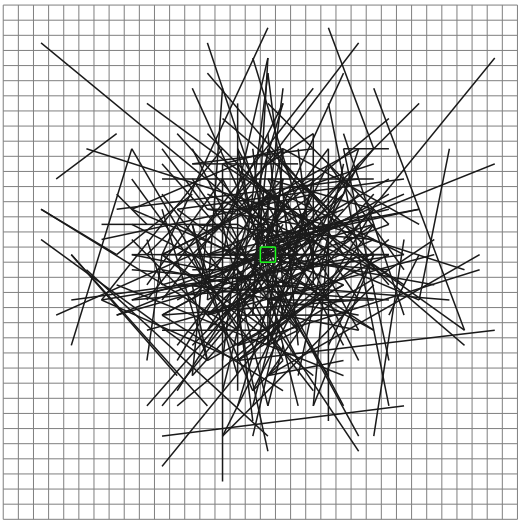

Figure 2.4: Example of the Gaussian$(0, \frac{1}{25} \cdot 32^2)$ distributed pixel comparison pairs used to calculate the BRIEF descriptors in a 32×32 patch. Each of the 256 straight lines marks a comparison of two pixels. The corresponding bit is set to 1 or 0, depending on which pixel is brighter. The candidate pixel is marked in green.

2.4 Oriented BRIEF and Rotated FAST (ORB)

The ORB feature-detector[13] combines the FAST keypoint-detection with BRIEF descriptors and extends them to make the features both rotation- and scale-invariant.
To achieve scale invariance a scale pyramid is constructed before any feature detection is performed. Unlike SIFT, the scale levels aren't octaves scaled down by a factor of 2 each level. We also don't convolve our images with multiple Gaussians. Instead we have only one scaled down image per level, which is scaled by a factor sf, which can be chosen freely, as seen in figure 2.5.
FAST is then run individually on each scaled image in the pyramid. In ORB, the radius of the FAST-circle is 9 pixels. To better distinguish the corners by their quality, a Harris[7] corner score is calculated by which the keypoints can be sorted.
Next we compute an orientation for each keypoint. For this purpose, we calculate the moments of the image patch around the keypoint using the intensity centroid as defined by Rosin[11]: $Centroid = \left(\frac{m_{10}}{m_{01}}, \frac{m_{00}}{m_{00}}\right)$ with the moments m_{pq} defined as:

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y)$$

Which gives us the orientation of the patch with $\Theta = atan2(m_{01}, m_{10})$.

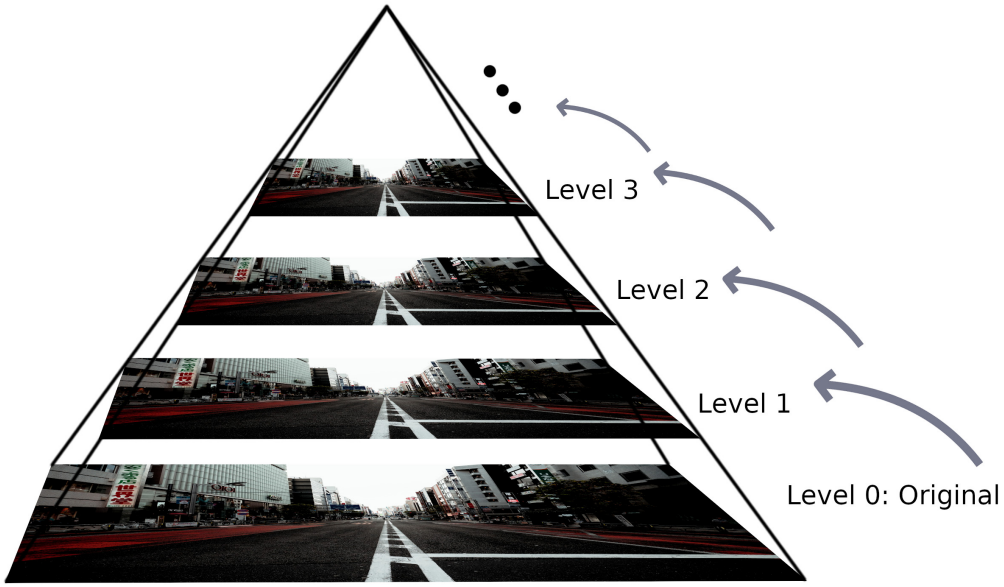

Figure 2.5: Visualization of the scale pyramid used in ORB. The image is scaled down by the same factor per level.

As we now have an angle for each keypoint we can then apply this orientation before calculating the descriptors. This effectively rotates the relevant patch into the same position regardless of the original orientation in the image.
The next step is to calculate the descriptors. The comparisons used for the BRIEF descriptors in ORB have been optimized by machine-learning them (seen in figure 2.6), as opposed to the Gaussian-distributed comparisons used originally.
The ORB feature detection and description is overall very robust to rotation and scaling, and decently invariant when it comes to changes in illumination and image noise. As it is very fast to compute and match, it lends itself to application in real-time SLAM.

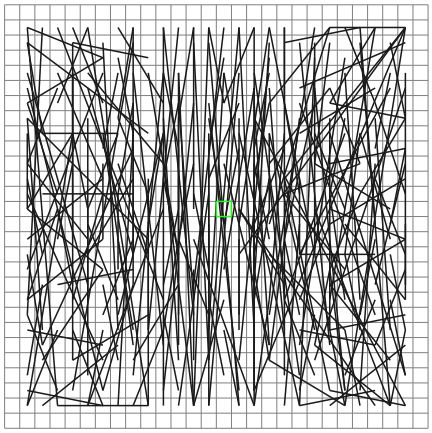

Figure 2.6: The machine-optimized BRIEF-comparisons used in ORB.

In our implementation of the ORB feature detector we have some slight additions and changes.
FAST is performed in a circle of 16 pixels around the candidate, as described in section 2.2, instead of the 9-sized circle used originally.
During FAST, the cornerness score of the keypoints is already computed, so the best features can be selected at this point and we can thus eliminate the large majority of them. This saves time on the computation of both the angle and the descriptor which only need to be performed on the features that will be part of the result.
Features are detected in overlapping patches roughly 30 $\times$ 30 in size. The threshold used during FAST-detection can then be adjusted to the alternative minimum value if no keypoints are found in a patch that is located within a homogeneous region in the image. This prevents areas of the image that contain less distinctive features from not having any keypoints detected at all if the initial threshold was set too high.
We also ensure that the number of features we retain per scale level is proportional to the size of the image. With N desired features in total, L levels in the scale pyramid and a scalefactor S, the desired features per level are set to $N_{level} = \left(\frac{1}{S}\right)^{level} \cdot N \cdot \frac{1-\frac{1}{S}}{1-(\frac{1}{S})^L}$.
This number is used in the distribution algorithms when the features are distributed per scale level.

2.5 ORB_SLAM2

In SLAM we are provided with a series of sensory inputs. From this input we want to compute an estimation of both a map of the unknown environment as well as of the respective location of our agent. Sensory input can come in a variety of different forms, such as laser-scans, odometry or tactile data. Almost always we are provided with a camera feed, which can be mono or stereo. In 2015 an open-source SLAM system that employed the ORB feature detector was released under the name ORB_SLAM2[10]. ORB_SLAM2 is built on ORB_SLAM[9], which only handled mono-images. On top of that ORB_SLAM2 is also capable of handling stereoscopic image feeds as well as RGB-D (monocular images with a depth map). The depth map is used by ORB_SLAM2 to generate a synthetic stereo coordinate. After that point the system is agnostic to whether actual stereo imagery or a monocular image plus a depth map were used as input.
The global map in ORB_SLAM2 is estimated using only keyframes. A normal frame is promoted to a keyframe based on a number of conditions:

- More than n_{min} frames have passed since last keyframe creation and the mapping thread is idle.
- More than n_{max} frames have passed since the last keyframe creation. The mapping thread is interrupted in this case.

- Few matches were found and therefore tracking is weak.
- The frame shares few tracked points with the reference keyframe (the keyframe that shares the most tracked points with the current frame).

An overview of ORB_SLAM2 can be seen in figure 2.7. Features detected during input pre-processing are mapped onto the internal 3D-map. With stereo and RGB-D input a map can be created on the first frame, using the depth information gained from the stereo coordinates. With every additional frame, matches between detected features and features in the map are located and the estimation of the camera position adjusted. Loop closure is performed if the system recognizes an area it has already been in. In such a case the whole trajectory is re-optimized based on this new information.

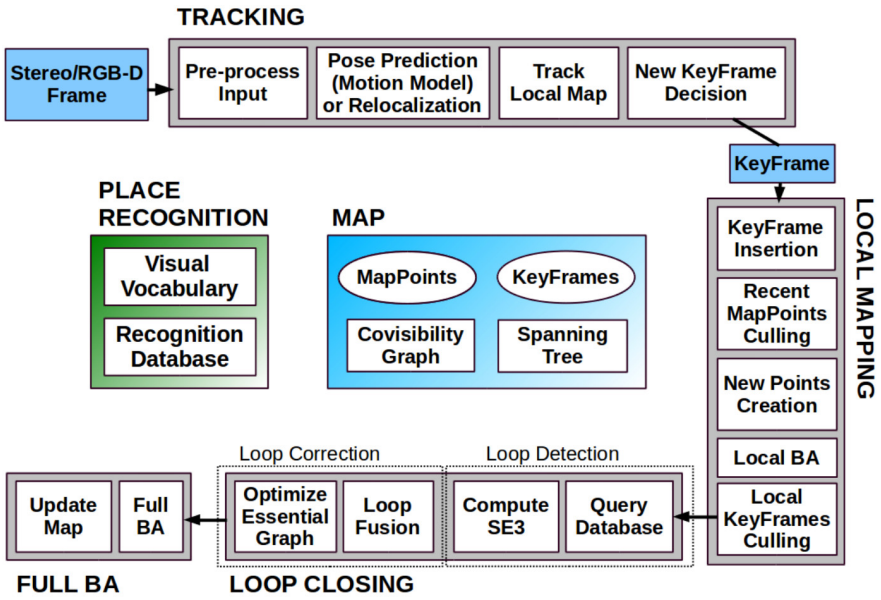

Figure 2.7: Overview of ORB_SLAM2's main threads. Keypoint detection takes place as part of input pre-processing, everything after it is based on the detected features. (Image taken from [10])

We replaced ORB_SLAM2's feature detector, adjusted the keyframe-decision and disabled loop closures (see section 5.1) but everything else remains in place.

2.6 ANMS-Keypoint-Distribution

There has also been prior work done to research efficient distribution of keypoints. In an article titled 'Efficient adaptive non-maximal suppression algorithms for homogeneous spatial keypoint distribution' [1] a few different methods to distribute keypoints evenly across the analyzed image are discussed. The proposed algorithms implement an iterative adaptive non-maximal suppression with slight variations in underlying datastructure.

In each iteration keypoints are added to the result in order of their cornerness score. Upon selection of a keypoint all other features in a region of interest around the selected one are eliminated. This is performed until all keypoints have been either added to the result or eliminated. If the size of the result is either larger or smaller than the desired number of keypoints $\pm$ a threshold th, the process is repeated with an adjusted region of interest. The various implementations will be discussed in more detail in later chapters, as well as a proposed improvement upon the best selection-method as per the authors.

Chapter 3

Distributing Keypoints

In order for meaningful selection to be able to take place we first need a way to assess the quality of any individual keypoint. We define a score function S and, by calculating the score of each recognized keypoint, we can order the keypoints by their respective S.
The question is how to define that score function. In ORB the score was originally calculated by applying a Harris corner measure. During testing however, we found that the Harris score did not provide any better results (slightly worse, in fact) than the one that was used by the function implemented in OpenCV[4] (and thus, ORB_SLAM2). With I being the intensity of a pixel, P the candidate and t the FAST-threshold, this function is defined as:

$$\max\left(\max\left(I_P - I_k, ..., I_P - I_{k+9}, t\right), |\min\left(I_P - I_k, ..., I_P - I_{k+9}, -t\right)|\right), k \in \{0, 2, ..., 16\}$$

So, within the given circle of n pixels around the central pixel p we look for the lowest difference within all contiguous sets of $\frac{16}{2} + 1$ pixels. Then, from all the lowest scores within those sets we select the highest score. This is performed twice, to cover both negative and positive differences.
Since only minimum-, maximum- and comparison-operations are required, the calculation is efficient and the results provide a very reasonable measure of estimating the cornerness of any given keypoint.
All keypoints in the example images were detected with an initial FAST-threshold of 20, a minimum threshold of 7 and over 4 scale levels. With these settings, around 30.000 detected features per megapixel were typical (depending on image domain), of which we want to keep 2000.
Important to keep in mind is that regardless of distribution method, all keypoints are selected per scale level and consolidated in the result. Selection per scale level is far more robust (at least in ORB_SLAM2) but brings with it more visible clustering in the final image, where the features of all scale levels are shown together, than if one were to distribute the final result scale-agnostically. How ANMS-distribution (discussed in section 3.4) looks if performed per scale vs on the final result can be seen in figure 3.1 and figure 3.2, respectively. In figure 3.3 all keypoints that were detected with the selected parameters are shown.

To make the examples more illustrative we distribute per level (like in figure 3.1), but only show the keypoints detected on the first of the four scale levels. With 2000 desired features in total, a scale factor of 1.1 and 4 scale levels we have $N = \left(\frac{1}{1.1}\right)^0 \cdot 2000 \cdot \frac{1-\frac{1}{1.1}}{1-\left(\frac{1}{1.1}\right)^4} \approx 574$ desired features for the first level.

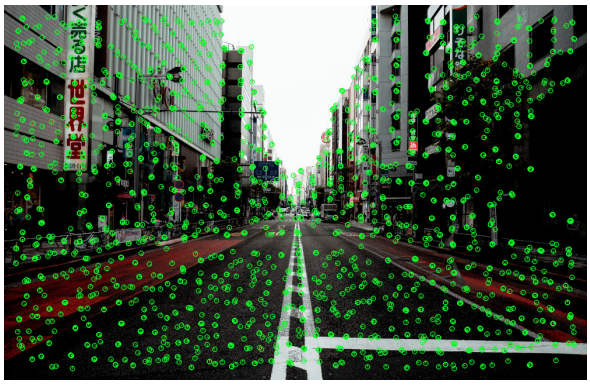

Figure 3.1: In this image the keypoints are distributed per scale level. The algorithm prevents clusters of keypoints from occurring on each scaled image. In the result the keypoints of all scale-levels are consolidated.

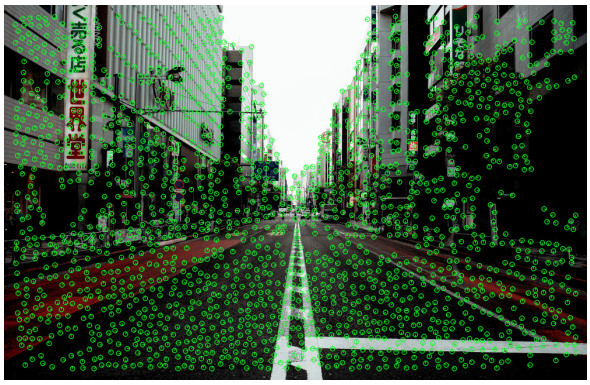

Figure 3.2: Here all detected keypoints were consolidated first and then distributed. They are spaced out much more evenly across the image, but a number of features per scale-level can no longer be guaranteed.

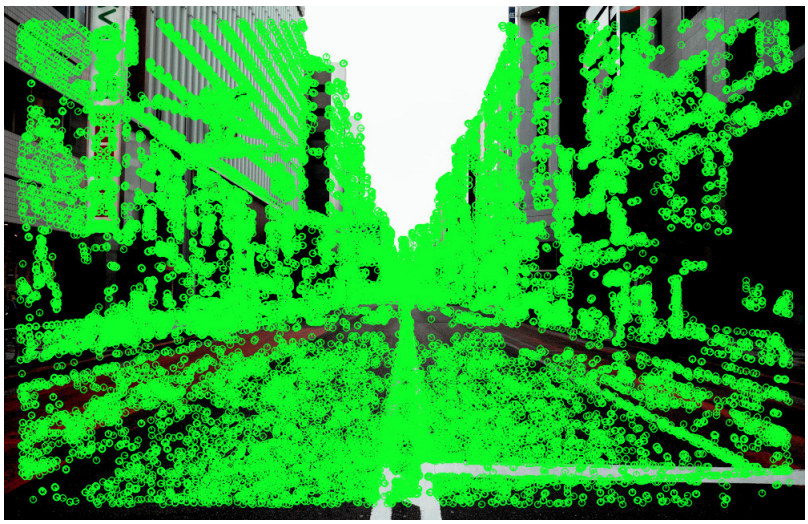

Figure 3.3: All detected keypoints in the image(1000 × 640). With 4 scale levels, a scale-factor of 1.1, an initial FAST threshold of 20 and a minimum threshold of 7, roughly 28,220 keypoints were detected.

If the number of keypoints found should for some reason be equal to or less than the number we want to keep all of them are kept. This step is implemented before any of the specific algorithms are called, which is why this exception is not specifically mentioned in the respective sections.

3.1 Top N

Top N is the simplest and, because of its simplicity, also the fastest selection method. If more than the desired number of N keypoints are found we add the best scoring N keypoints to the result. Since we keep all features better than the Nth one and order within the retained features does not matter, we can solve this by using an nth-element algorithm, thus achieving an average complexity of O(N).

In figure 3.4 we can see that the keypoints selected by this method cluster heavily in high-

Figure 3.4: Shown here are the keypoints retained by the Top N algorithm on the first level of the scale pyramid.

contrast areas. On the signs at the side of the road we have a large number of keypoints on the outline of the characters. The road itself only had features in the center selected, at the corners of the road surface markings. Clusters provide redundant information, while we have no information at all about the areas devoid of features.

3.2 Bucketing

The next simplest solution: the image is divided into gridcells of 80×80, a value which in our tests has empirically shown to provide good results. If the dimensions of the image don't allow for this size, we choose the number of columns or rows of the image instead, whichever is lower. The starting value can also be adjusted by the algorithm to prevent buckets at the edges from becoming too small.

Per gridcell we then select the best $\lfloor \frac{N}{|gridcells|} \rfloor$ features the same way as in top N. Because of the flooring, this ends up giving us slightly less features than the desired amount. This did not, however, negatively impact the results of this method.

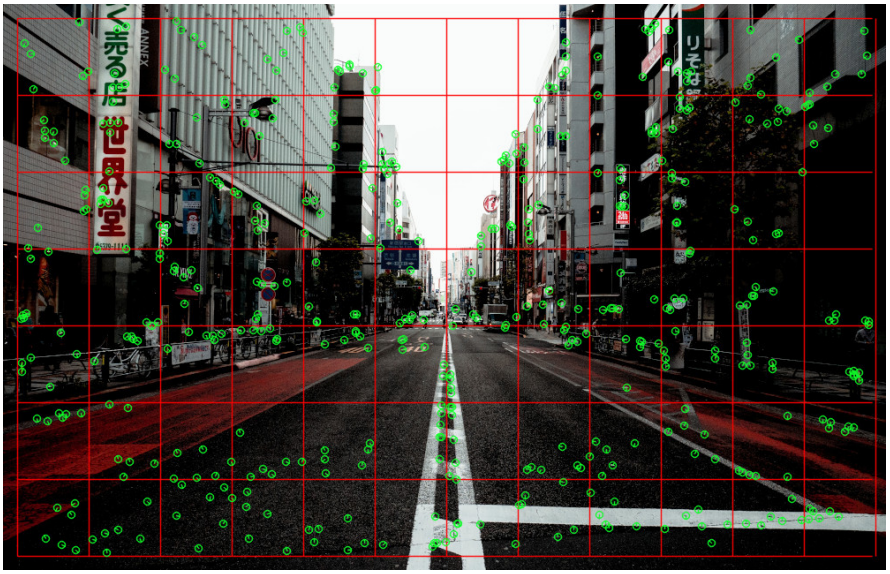

Figure 3.5: Keypoints distributed via bucketing. The gridcells of size 80×80 are marked with red lines.

In figure 3.5 keypoints selected by bucketing. Keypoints are much more evenly distributed across the image. As clustering within buckets is not prevented, there is still some empty space, such as in the middle of either side of the road. Minor clusters and empty sections aside, the features are very well spaced out across the image.

3.3 Quadtree

Distribution via Quadtree is the method initially implemented in ORB_SLAM2:
As we see in figure 3.6, in order to construct the quadtree, we first add all keypoints to the root node, which contains the whole image. The image is then divided into 4 cells of equal size. From line 7 onward we repeatedly subdivide every node, again into 4 equally sized cells. The subcells are added to the list containing all nodes, while the cell that was split is deleted. If a node only contains one keypoint it will not be divided. Once the total number of nodes equals or exceeds the number of desired keypoints N, we keep only the point with the highest score in each node and add it to the result.
In figure 3.7 we see the result of this method. Keypoints are well distributed over the image, with no major clusters or empty spaces.

```
input: detected features P, desired number of features N
nodelist L = ∅
output = ∅
root = P
L = L ∪ root
divide root into 4 nodes
while (|L| < N):
    for each node n_i in L:
        if (|n_i| > 1):
            subdivide n_i
            add subnodes n_i1, n_i2, n_i3, n_i4 to L
            delete n_i
        if (|L| >= N):
            for each node n_i in L:
                add best-scoring keypoint p ∈ n_i to output
            return ouput
```

Figure 3.6: Pseudocode for the quadtree distribution as implemented by ORB_SLAM2

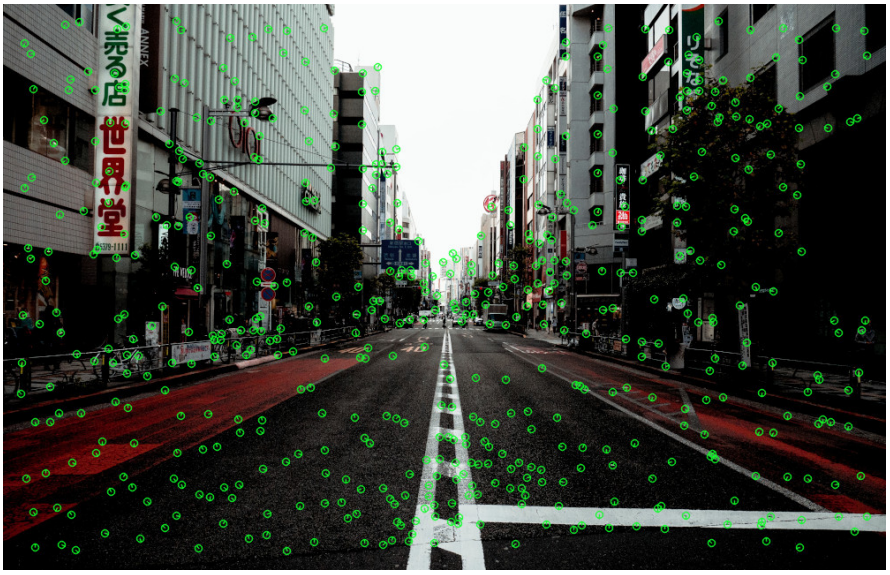

Figure 3.7: Keypoints as distributed by ORB_SLAM2, with a quadtree-based distribution algorithm.

3.4 Adaptive Nonmaximal Suppression

Distributing the detected features via adaptive non-maximal suppression requires them to be completely sorted by corner-score. The various implementations of this algorithm vary in the underlying datastructure used to find the neighbouring keypoints in the region of interest around the added keypoint, as well as the nature of the region of interest.

```
1  input: detected features P, desired number of features N
2  sort P by corner-score
3  while (true):
4      if (ROI update failed)
5          return output
6      mark all p_i in P as legal
7      output = ∅
8      for all p_i in P:
9          if (p_i legal):
10             add p_i to output
11             for all p_j in ROI around p_i:
12                 mark p_j as illegal
13     if ¬(N-ϵ < |P| < N+ϵ)
14         update ROI
15     else
16         return output
```

Figure 3.8: pseudocode for the adaptive non-maximal suppression algorithm

The pseudocode can be seen in figure 3.8. The main loop starts in line 3. It will be repeated until the size of the output is within $\pm\epsilon$ of N. Whenever we start over, all keypoints are marked as valid and the result of the last iteration is discarded (lines 6-7). We then look at all input-keypoints in descending order of their corner-score and add it if it still valid (lines 8-10). Whenever a keypoint is added, all neighbouring keypoints within their region of interest are marked as invalid (lines 11-12). If our output size is too small we decrease the size of the ROI and start over, if it is too large, we decrease it.

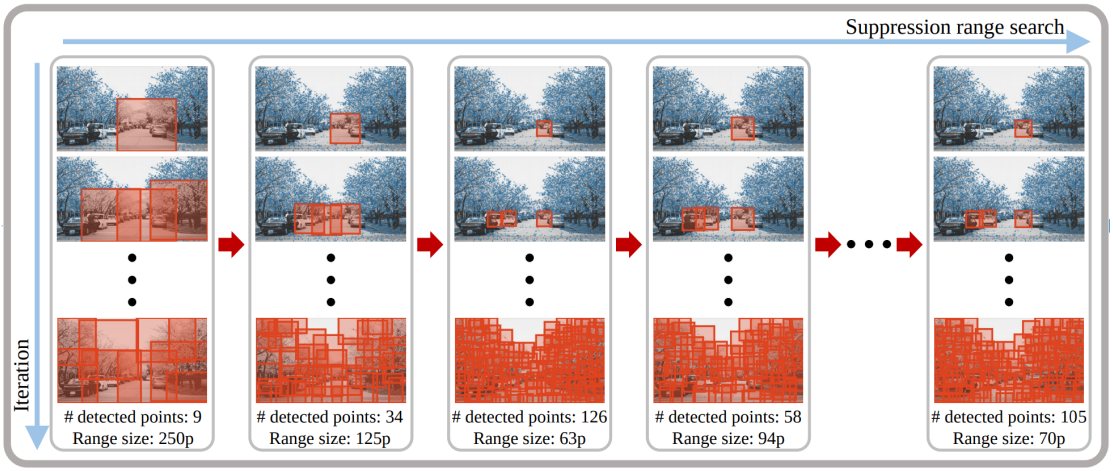

Figure 3.9: Visualization of the algorithm. Per iteration points are excluded in the ROI around the best point that is still legal, then the search range is adjusted and the process repeated until the size of the result is correct.

To zero in on the right size we have an upper and a lower bound saved. The search range r is set to $r = lower + \frac{upper-lower}{2}$ at the start of every iteration. To increase the ROI we set the lower bound to $r + 1$, to decrease it we set the upper bound to $r - 1$. This is the update of the ROI in line 14. If upper becomes less than lower or the other way around, the range can no longer be updated and we return the output as is (lines 4-5). If the range is the same as in the last iteration after the update the process is also aborted by the same check.
A visualization of the workflow of the algorithm, taken from [1], can be seen in figure 3.9.

One issue is that each iteration invalidates the previous result. The threshold of what is an acceptable amount of retained features is set to $N \pm 0.1N$, so ϵ in figure 3.8 is $0.1 \cdot N$. The whole process needs to be redone if we are outside of that number after a iteration, which could become costly depending on the number of iterations we need until we arrive at the correct search range. A solution to initialize the upper and lower bound to a value that reduces the number of passes dramatically (as presented in [1]) is discussed in chapter 5.
In figure 3.10 we see that with this selection method we get output keypoints that are very

Figure 3.10: Keypoints distributed via ANMS

evenly spaced out across the whole image. Clustering is made impossible by the exclusion of neighbours. Regardless of the datastructure we use to find the nearest neighbours the results are almost identical.

3.4.1 Kd Tree

The first variant of the adaptive non-maximal suppression algorithm is based on a 2-dimensional kd tree datastructure. A 2D kd tree is constructed by repeatedly splitting the input at the median point. Each split occurs along the ($depth \mod dimensionality$)-axis, so in our 2D tree we alternate between splits along the x- and y-axis. We used Nanoflann [2] for our implementation.

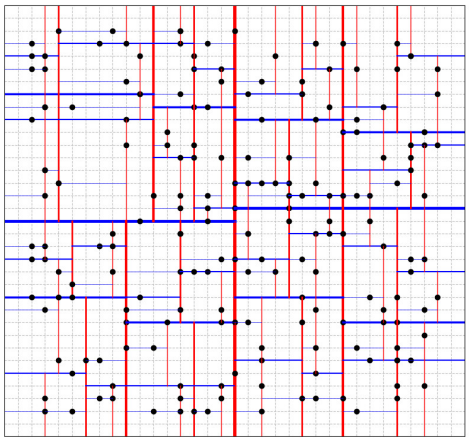

Figure 3.11: Example kd tree with 200 input points. Each red line represents a split along the x-axis, each blue one a split along the y-axis.

An example tree can be seen in figure 3.11. To exclude the keypoints around a chosen candidate nearest neighbour radius search is employed. Since the tree is constructed once per set of keypoints and we distribute per scale level, we actually need to construct multiple trees per analyzed image. Even though there are less features found in scaled down images, the time needed for construction and queries is not insignificant (see section 4.1).

3.4.2 Rangetree

Instead of a kd tree we can instead use a rangetree as the datastructure to store and find the keypoints in. A rangetree is a tree data structure that saves a 1D binary search tree in each node for each dimension. In our 2D case we therefore have a subtree in each node for the x- and the y-axis. As can be seen in figure 3.12 this gives us a square ROI in which points are excluded. The rangetree is even more expensive to build than the kd tree, but since the

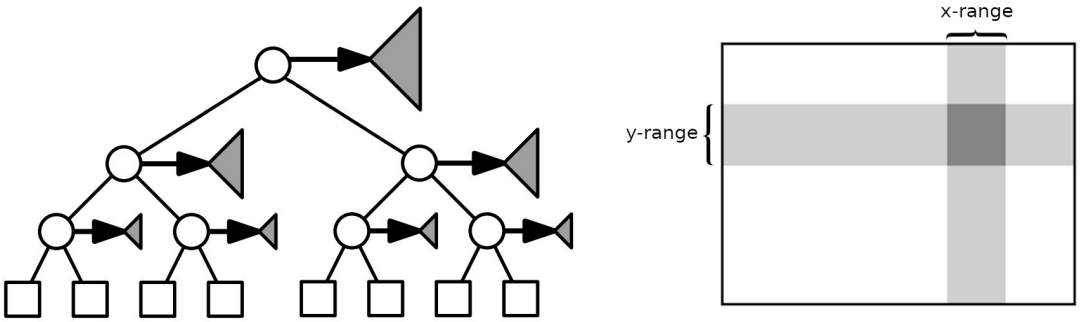

Figure 3.12: On the left we see a representation of the subtrees in each node. If the white tree is constructed along the x-axis, each gray triangle represents the respective subtree along the y-axis. On the right side we can see what a search in a ROI with this TDS would look like. First we search in one dimension, then within that subtree we search in the other dimension.

search is slightly faster it performs better than the kd tree overall, especially as input grows larger.

3.4.3 Suppression via Square Covering (SSC)

By sorting the keypoints into gridcells over which we can exclude the neighbours SSC gets rid of the expensive datastructures of the previous two implementations. Adjustment of the ROI comes in the form of changing the resolution of the grid the keypoints are placed in. Search for adjacent features to be excluded is then performed in a 5×5 grid around the candidate in the center.
This version is much faster than the previous ones and finding and excluding neighbours in this simpler way does not sacrifice any quality in matching. On the contrary, the average error of estimated trajectories calculated with SSC-distributed features was less than the expensive variants in basically all the tests we ran (see section 5.5).

3.5 Soft SSC

A potential issue of all the ANMS-methods described so far is that the features that are eliminated around the candidates may be high-scoring ones themselves. Even if those keypoints may not add a lot of information due to the closeness to another feature that will likely be matched in the next frame, having multiple matches in the same small region of the image can still add to the robustness of the matching process and thus to a better estimated trajectory. This is especially the case since it is not guaranteed that the retained feature that led to the exclusion of the others will be the highest scoring one of the same region in the next frame. Another feature might earn a higher corner-score due to slight changes in angles or lighting, which would then actually exclude the feature that was scoring the best the frame before.
As an extension of SSC, soft SSC is the method proposed by this thesis to solve this issue. Instead of excluding all neighbours in the ROI, only features with a score lower than $s + th_{SSC}$ are removed from consideration, where s is the score of the candidate and th_{SSC} a threshold-value.

The threshold depends heavily on the method used to calculate the score and the range of values that are possible. In this case, with the score-definition as described earlier we can get score values of $th_{FAST} \leq s \leq 255$, where th_{FAST} is the threshold used during the FAST feature detection. Typically the vast majority of values place in the lower parts of this range though, which means that a small threshold is sufficient to avoid the discarding of stable keypoints. With this score calculation we found that the threshold should not exceed a value of around 5, otherwise the condition of required number of features of $N \pm 0.1N$ can not be satisfied in many instances. A value of 3-4 however, depending on the environment of the dataset, provided results that were a slight improvement on regular SSC, with performance also being slightly faster.

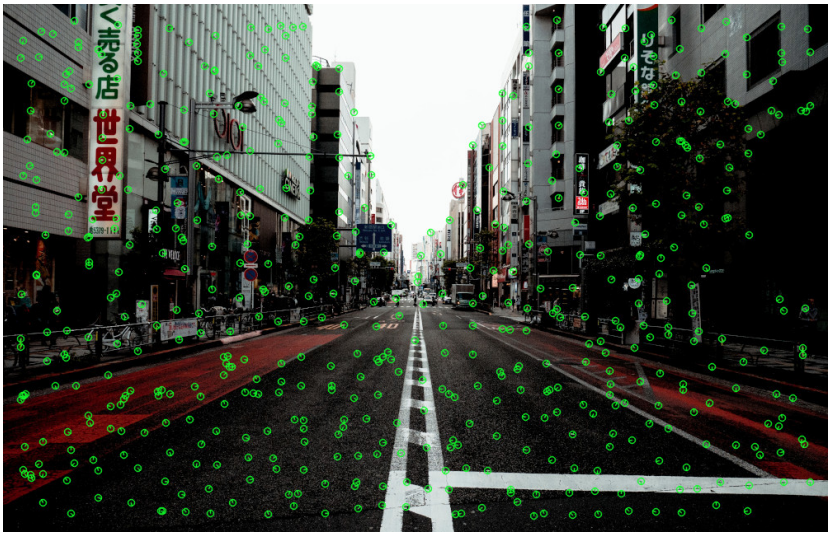

Figure 3.13: With soft SSC points in the ROI are only discarded if their score is worse by a threshold than the candidate. This results in slightly more visible clustering.

In figure 3.13 we see that the keypoints are not quite as homogeneously distributed. Since we can not guarantee that all keypoints we select in traditional ANMS will find matches the slight clustering is not necessarily redundant information. In figure 3.14 we have zoomed in on two small parts of the image where the difference between SSC and soft SSC is highlighted.

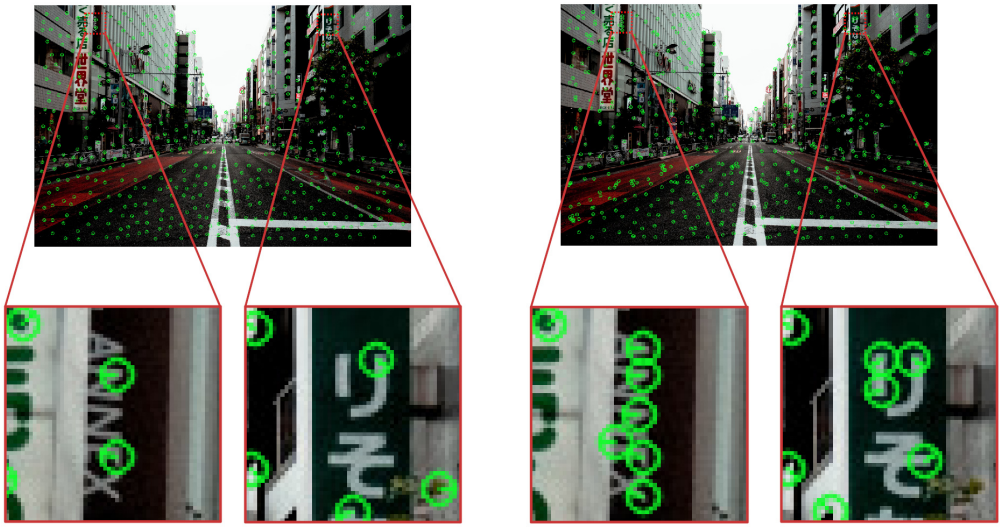

Figure 3.14: The keypoints on the left are distributed by SSC, the ones on the right by soft SSC. In the zoomed in sections we can see that soft SSC allows features to be closer together if they score highly.

Chapter 4

Performance

4.1 Time Complexity

Already shortly discussed in the previous section, we want to take a further look at the complexities of the datastructures that are employed for the neighbourhood-searches in the ANMS-algorithm.
All variants of the algorithm rely on a sorted input set of keypoints, so for this first step we start out with $O(M \log M)$, M being the size of the input set of keypoints. In the case of the kd- and rangetree methods, we also need to build the tree first. This comes down to a complexity of $O(M \log M)$ and $O\left(M \log^2 M\right)$, respectively. To see how expensive query times are, we then need to multiply the number of iterations we need in order to arrive at the correct search range r with how long it takes to work through one search range guess. For the iterations over the range we have $\log r_{ini}$, since we employ a binary search algorithm to find it. Per guess we need look through all M keypoints plus all the keypoints that are excluded around each one added to the result. Query time for a keypoint and k associated points in an ROI in a 2-dimensional rangetree is $O\left(\log^2 M + k\right)$. In a 2-dimensional kd tree the equivalent search is in $O\left(\sqrt{M} + k\right)$.
Thus we arrive at an overall query complexity of

$$O\left(\log r_{ini} \cdot \left(M + \log^2 M + \sum k\right)\right)$$

and

$$O\left(\log r_{ini} \cdot \left(M + \sqrt{M} + \sum k\right)\right)$$

for the range- and kd tree, respectively.
Overall, time-complexity wise, the kd tree performs the worst, especially with a lot of detected features, followed closely by the rangetree, though the latter does not scale quite as badly with a large input. SSC is much faster, and scales much better with a larger set of input features.

Time complexity			
Method	Sorting	Build	Query
Top N	-	-	$O(M)$
Bucketing	-	-	$O(2M + N)$
Quadtree	-	$O(M)$	$O(M + N)$
KD tree	$O(M \log M)$	$O\left(M \log M\right)$	$O\left(\log r_{ini} \cdot \left(M + \sqrt{M} + \sum k\right)\right)$
Rangetree	$O(M \log M)$	$O\left(M \log^2 M\right)$	$O\left(\log r_{ini} \cdot \left(M + \log^2 M + \sum k\right)\right)$
SSC	$O(M \log M)$	-	$O(r_{ini} \cdot (M + N))$

Table 4.1: Time complexities of the various methods. M is the size of the input, N the number of features we want to keep and k the number of keypoints in a ROI around a candidate keypoint. The rangetree is the slowest to build, while the kd tree has the longest query time.

In order to minimize the amount of times we have to guess the size of the ROI, it is advisable to find a good starting guess. This can be achieved by calculating the lower and the upper bound of possible sizes of the ROI and taking the average as the starting guess. The upper bound is the width of the exclusion region under the best-case assumption, so if all detected keypoints were perfectly distributed inside the image. This would provide us with a square side of

$$upper = -\frac{H + W + 2N - \sqrt{4W + 4N + 4HN + H^2 + W^2 - 2WH + 4WHN}}{(N - 1)}$$

where H is the height, W the width of the image and N the desired number of features. The lower bound then is the exact opposite, the worst-case scenario where all detected features are cluttered in a single region of the image with no space between them. We still want to have N points in the result; with P input points and N points to retrieve we have $P = N \cdot \left(lower^2\right)$. Solved for lower this gives us

$$lower = \sqrt{\frac{P}{N}}$$

Since we search in each direction, we only take half of a square side as our search range and end up with an initial search range of $r_{ini} = \frac{1}{2}\left(upper + lower\right)$. With this initialization, it was shown by the 6 authors in 2018[1] that the ANMS-methods only take around a third of the iterations, as opposed to without it, to arrive at a search-range that gives us the desired result. An overview of all complexities can be seen in table 4.1.

4.2 Storage Complexity

Storage complexity is a simpler issue, apart from the tree datastructure-based algorithms, everything is rather straightforward.
Top N can be done in-place, by partially sorting and the cutting the array off at the Nth

Space complexity	
Top N	$O(M)$
Bucketing	$O(2M)$
Quadtree	$O(2M + 2N)$
KD Tree	$O\left(2M + N + \max\left(k\right)\right)$
Rangetree	$O\left(M \log M + M + N + \max\left(k\right)\right)$
SSC	$O(2M + N)$

Table 4.2: Storage complexity of each algorithm. M is the size of the input, N the size of the output and k the number of keypoints in the ROI.

element, as such it only needs O(M) storage. If we bucket our keypoints, in addition to the input, we also need to store the keypoints (or their indices) per bucket, which is also M in total. Selection per bucket again can be done in-place, as well as placing the result in our return vector.
The Quadtree needs space for N nodes as well as M keypoints plus memory for the return vector. The kd- and range tree are the most expensive variants, as they need to temporarily save the results of the ROI queries in addition to the memory required for their tree data structures. The actual data structure for the kd tree is relatively inexpensive (input can be stored in O(M)), while the range tree also stores a sub-node per dimension in every node, making it the most storage-intensive ANMS-variant. All ANMS variants also need space for the vector which saves whether a keypoint has been excluded or not. On most hardware storage of features should be only of minor concern. A table displaying the complexities can be seen in table 4.2.

In addition to the real experiments in chapter 5 we also ran a short series of synthetic experiments. Random keypoints were added to the input-array in a 1024×768 grid. The number of keypoints in this input was raised steadily from 5000 to 30,000, in increments of 1000. Each distribution was then called to select the best 1000 features 100 times per increment on the same set of input features and the calculation times were averaged. The results can be seen in figure 4.1. The kd tree scales almost linearly at about 1ms per 1000 input features. While the range tree is more than twice as fast beyond input sizes of around 15,000 it is still much slower than the rest of the field.
This is generally consistent with the real experiments in chapter 5, with only one exception. Soft SSC performed slower than normal SSC with synthetic input, while in all real sequences the opposite was the case. This might be the case because of the unrealistic distribution of the cornerness score of the synthetic features, as those were evenly distributed within the possible range of $[threshold, 255]$. In real detection the vast majority of scores fall within the lower third of this range, thus allowing Soft SSC to arrive at the correct search range more quickly.

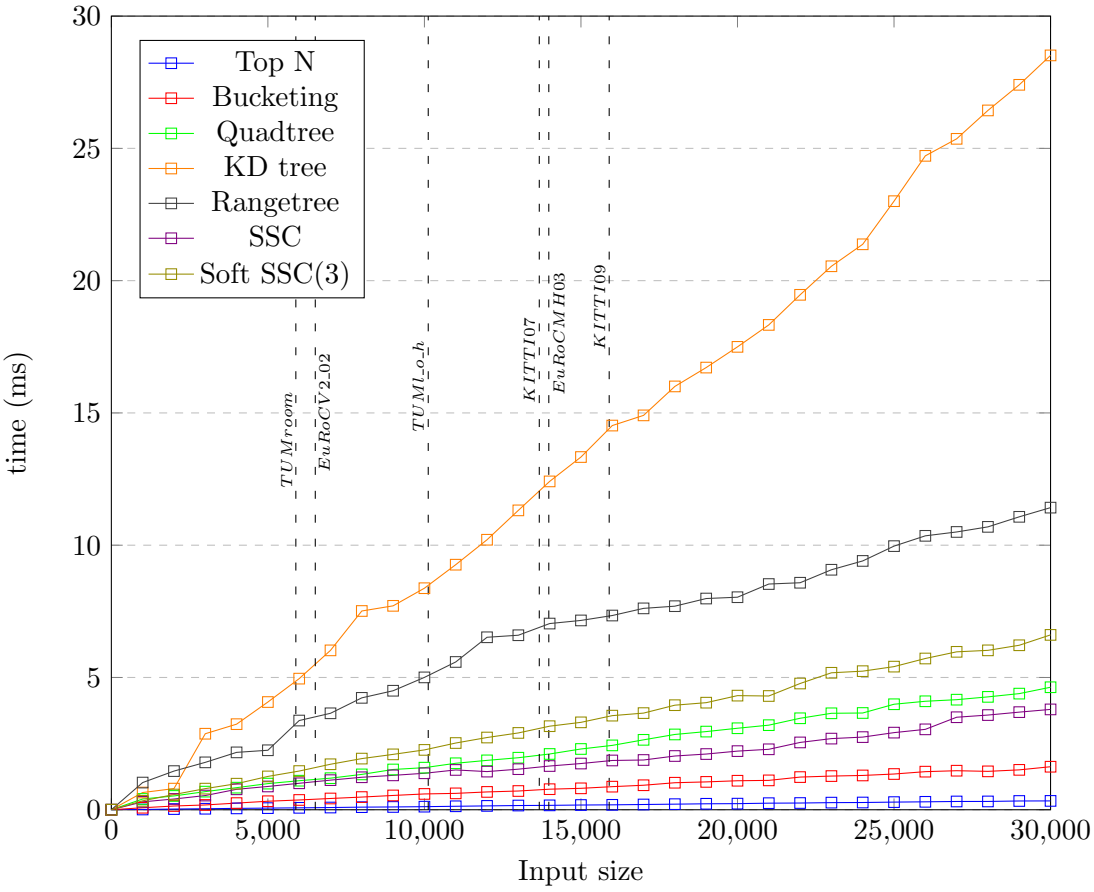

Figure 4.1: The TDS-based ANMS variants slow down noticeably with larger input. The dashed lines mark the average input size of the datasets used for the experiments in chapter 5. While that average mostly does not exceed a value of 15,000, there were regular outlier images in all sequences that had 30,000 or more keypoints detected.

Chapter 5

Evaluation

We give ORB_SLAM2 various video sequences from which an estimated trajectory is calculated. This is done for each of the distribution methods presented in 3. We then compare each result to the ground truth trajectory and evaluate which method gives us the best result, i.e. comes closest to the ground truth.

5.1 Methodology

In order to ensure that the feature detector (which includes the distribution) was the relevant factor in the quality of the results, a few of modifications of ORB_SLAM2 were necessary.
Firstly, loop detection and closure was completely disabled, as this feature would otherwise alter the calculated estimation of the trajectory once a loop was recognized. This would retroactively eliminate accumulated drift and muddy the results. Drift is exactly what we want to observe, since it is greatly affected by matched points (and thus distribution).
The other adjustment that was needed was to alter the condition under which a new keyframe is to be inserted. On keyframe-insertion the local neighbourhood of frames attached to that keyframe are optimized, but one of the conditions to insert a keyframe is that few matches were found. This could be an indication of bad keypoint selection, a circumstance which normally ORB_SLAM2 would try to rectify by keyframe-insertion. This correction of bad selection however, is unwanted in our case, as bad distributions might produce better results than they otherwise would thanks to this counteraction. This was changed to insert keyframes at a steady pace of 5 per second, regardless of any other factors. This way it can be ensured that the quality of the matching will be seen in the quality of the results as well.

5.2 Used datasets

We need datasets that have ground truth information available. To see how the distributions performed in different environments, three datasets were used.

The KITTI Vision Benchmark Suite [5] provides various video sequences filmed in stereo by a camera rig mounted on top of a car; ground truth was measured via a laserscanner and GPS. Content of the videos are drives around rural areas and occasionally highways in and around the German city of Karlsruhe. These image sequences were shot in 10Hz. Figure 5.1 shows the setup of the car, in figure 5.2 we see a shot from sequence 09.

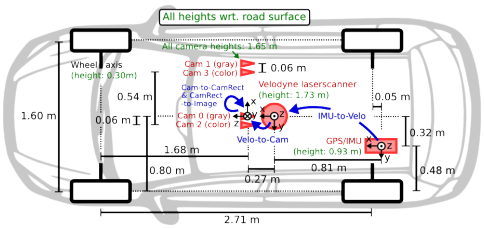

Figure 5.1: Setup of the KITTI Vision Benchmark Suite

Figure 5.2: Example image from KITTI sequence 09

The images included in the EuRoC MAV dataset were made using a Micro Aerial Vehicle (MAV) and are shot in 20Hz stereo. There are sequences of smaller and medium-sized environments available (rooms and a machine hall). The MAV is seen in figure 5.3, figure 5.4 shows the machine hall.

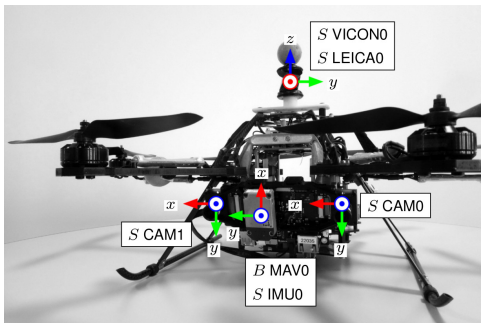

Figure 5.3: The Micro Aerial Vehicle used during data collection for the EuRoC dataset

Figure 5.4: The machine hall featured in EuRoC sequences MH01-MH05

Finally, a sequence from the RGBD-TUM dataset was also used, which does not provide stereo-images but instead RGB monocular images along with a depth map, which was made using a Kinect-system. The sequences are shot in 30Hz and mostly feature small, enclosed environments as their content. In figure 5.5 we see the sensor used to create the depth maps, figure 5.6 is an image from one of the two sequences used in our experiments.

Figure 5.5: Depth in the TUM dataset was mapped using a Kinect sensor.

Figure 5.6: Image from one of the two TUM sequences used: long_office_household

From each of those sets we chose two sequences to run the full tests on: KITTI 07 and 09, EuRoC MH4 and V2_02 and RGBD-TUM freiburg3_long_office_household as well as freiburg1_room.

5.3 Testing Environment

Per sequence and per distribution an estimated trajectory was calculated 100 times in order to minimize the effect of outliers and so that the sometimes minimal differences between methods could actually be attributed to the difference in distribution and not merely chance, as might be the case with a smaller amount of samples.

Parameters other than the distribution itself also were adjusted, sometimes slightly differently per dataset. In ORB_SLAM2, the factor by which the images in the scale pyramid were scaled down was set to 1.2, a value which in our testing performed consistently worse across all datasets than lower values. We arrived at a scale factor of 1.05 as a consistently well-performing factor. This smaller value provided much better results in the KITTI dataset specifically. The TUM and EuRoC sequences also performed better with images scaled down by 1.05 each, but the difference was much less dramatic. In table 5.1 we can see an overview of all parameters compared to the ones originally used by ORB_SLAM2.

Number of desired features was another inspected parameter. Generally, results tend to improve with more keypoints up until a point; once diminishing returns set in, soon after the results will actually get worse. We found that the optimal number scales roughly with resolution, specifically we arrived at around $N = p \cdot \frac{1}{3} \cdot 10^{-2}$, where p is the number of pixels in the image.
For the TUM sequences this gives us a value of $640 \cdot 480 \cdot \frac{1}{3} \cdot 10^{-2} \approx 1000$ features. EuRoC images, at a resolution of 752×480 work out to roughly 1200 features. The KITTI benchmarks, while the resolution of roughly 1225×370 varies very slightly in some sequences, give us around

Parameters						
	Dataset	N	Scale Factor	Scale Levels	$FAST_{ini}$	$FAST_{min}$
ORB_SLAM2	EuRoC	1200	1.2	8	20	7
	TUM	1000	1.2	8	20	7
	KITTI	2000	1.2	8	20	7
Ours	EuRoC	1200	1.05	8	20	7
	TUM	1000	1.05	8	20	7
	KITTI	1500	1.05	4	20	7

Table 5.1: Table detailing the parameters for the SLAM system.

1500 desired features per image.

Number of scale levels was set to 8 for both the TUM and the EuRoC dataset, while a value of 4 provided the best results for the KITTI sequences.
In the tests that were run to determine these values, all distributions across the board would perform either better or worse when a single parameter was changed, which indicates that the quality of the detected features itself was affected and the distribution methods were largely agnostic to these changes.
All tests were run on an Ryzen 2700X with 32GB of RAM.

5.4 Error Calculation

To calculate the error of the estimated trajectories, the camera motion estimated by ORB_SLAM2 is compared to the ground-truth trajectory. Since we want the relative pose error as opposed to the absolute one, accumulated differences up to the point of comparison are disregarded, and we only look at the difference in motion. In practice it does not seem to make much of a difference what measure of error is examined either way. Rotational or translational error, relative pose error or absolute trajectory error - they all align in every case that we examined. Which makes sense, of course, as an error in rotation results in a trajectory that translates into a wrong direction, and accumulated difference in motion over time results in a wrong trajectory as well.
The estimated trajectory consists of a series of poses $P_1, ..., P_N$. Each pose is defined by a translational and rotational coordinates as well as a timestamp. Given the ground truth poses $G_1, ..., G_N$, the error at time i with fixed time interval Δ is defined as:

$$error_i = \left(G_i^{-1} G_{i+\Delta}\right)^{-1} \left(P_i^{-1} P_{i+\Delta}\right).$$

From these error values we only take the translational component $error_t$, then compute the mean relative translational pose error: $MRPE = \frac{1}{N} \sum_{i=1}^{N} error_{t_i}$. From this point forward, all errors given are mean relative translational pose errors.
For the purpose of calculating the errors evo [6] was integrated into our evaluation pipeline,

which proved capable of handling any input trajectory of the three datasets and calculating the error against the ground truth trajectories. To arrive at the mean error over the 100 iterations per sequence, a small addition was necessary, as evo only handles individual trajectories.

5.5 Results

What became apparent during the testing process was that aside from the top N selection, all distribution methods performed very similarly, with the difference in mean relative pose error often not exceeding tenths of millimeters. These small gaps in relative error can however still make a meaningful difference over time. Within a relatively small amount of time the discrepancy in the trajectory can amount to a difference of multiple meters, even with very similar relative errors per pose. An example can be seen in 5.7.
After looking over the results of all real experiments a clear recommendation regarding the method of distribution in feature-based SLAM systems will be given.

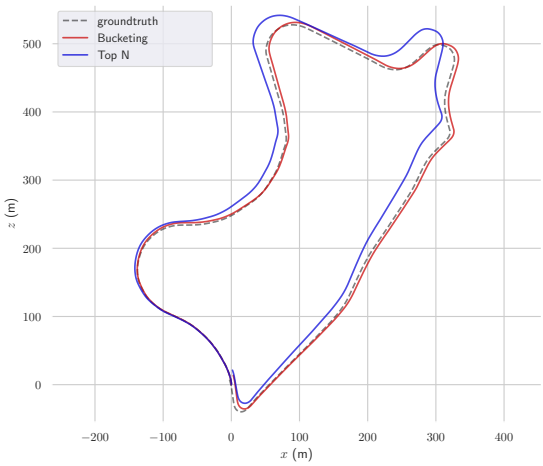

Figure 5.7: The blue example-trajectory was calculated using top N distributed keypoints, for the other trajectory the features were distributed by bucketing. So as to make the difference obvious we chose a case where the two distributions had a rather large discrepancy in error, about 3mm per pose. The difference in trajectory is clearly visible and the short sequence (KITTI 09) of about 2-3 minutes was enough to accumulate to a meaningful gap in absolute error: about 8m for bucketing versus circa 18m for top N, a 10m difference.

Bucketing performs extremely well in the MAV sequences (figures 5.8 and 5.9) and even top N does produce acceptable results. Initially it was our assumption that the good results of the bucketing method in these sequences was an outlier, as this performance does not match the results of [1], but it turns out that simply performing top N selection per fixed size gridcells is an extremely stable approach.
ORB_SLAM2's quadtree performs about on par with the ANMS approaches in the V2_02 sequence, which features the MAV flying around in a small room without much texture, but has a worse mean result than even top N in the machine hall sequence, though it is more stable with fewer outlier results than top N. The ANMS variants perform about equally, as would be expected, though Soft SSC does produce a slightly lower error than the original variants.

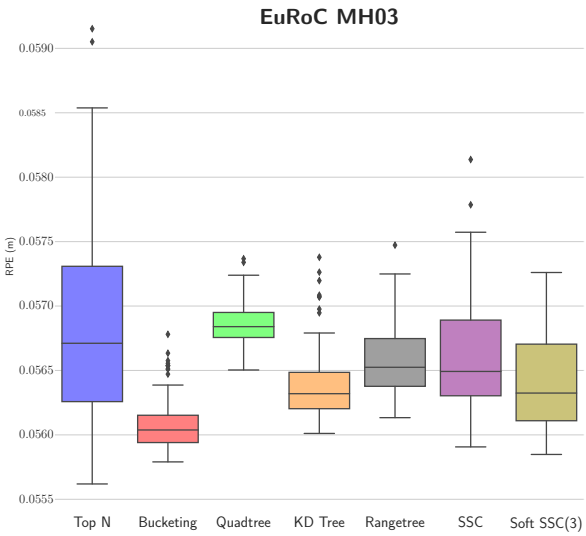

Figure 5.8: Boxplot of the mean error per distribution in the EuRoC MH03 sequence. Though there are a few outliers, bucketing produces the best results overall.

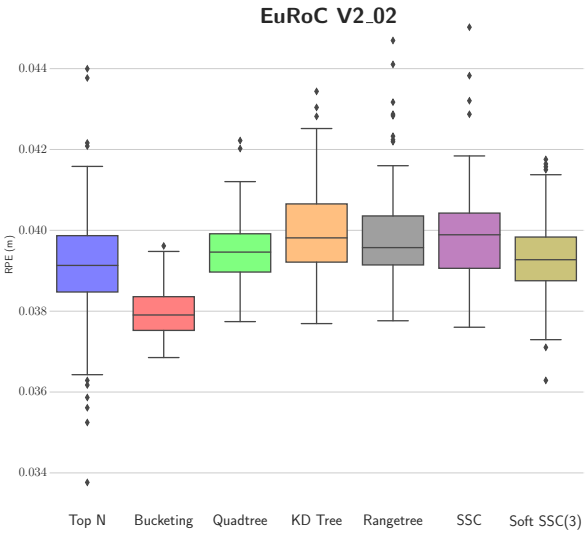

Figure 5.9: In this plot of the mean error per distribution in the EuRoC V2_02 sequence we again see bucketing perform the best. Notably, top N outperforms all methods that strive for homogeneous distribution.

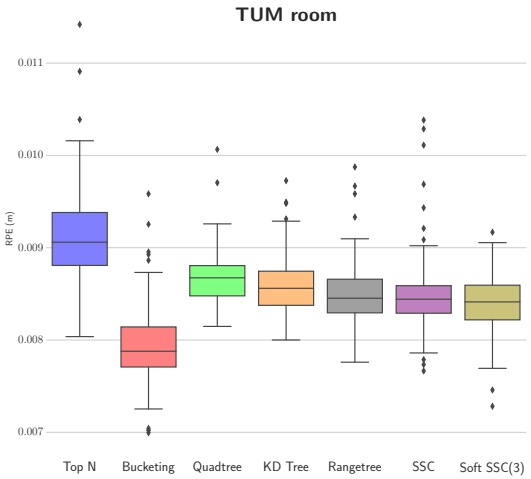

Figure 5.10: Mean error in TUM room. All methods are relatively close together, though bucketing does perform the best.

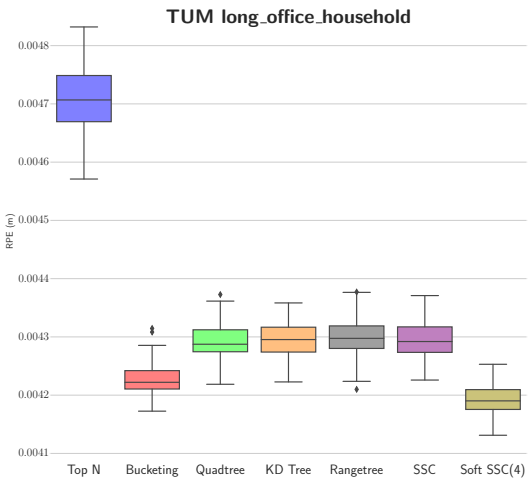

Figure 5.11: The second TUM sequences is one of the two sequences where soft SSC produces the smallest mean error. Bucketing is a close second.

The results for the RGBD-TUM sequences can be seen in figures 5.10 and 5.11. Bucketing seems to be the best solution for these sequences as well. Though outperformed by Soft SSC in one of the two instances, its consistent performance across sequences would give it the edge. Other than these two, and top N performing slightly worse than the rest, pretty much everything is very similar.

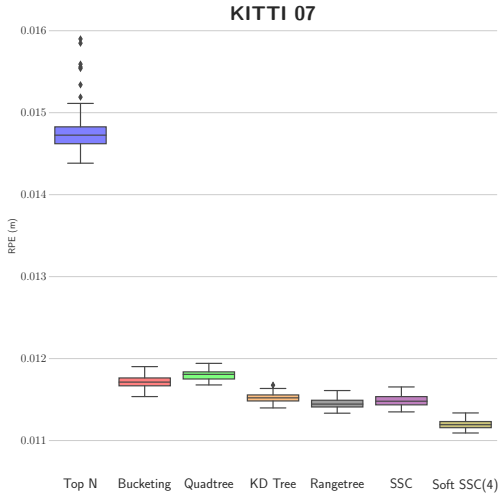

Figure 5.12: In both KITTI sequences there is a large gap between top N and the competition. Soft SSC clearly outperforms the competition here.

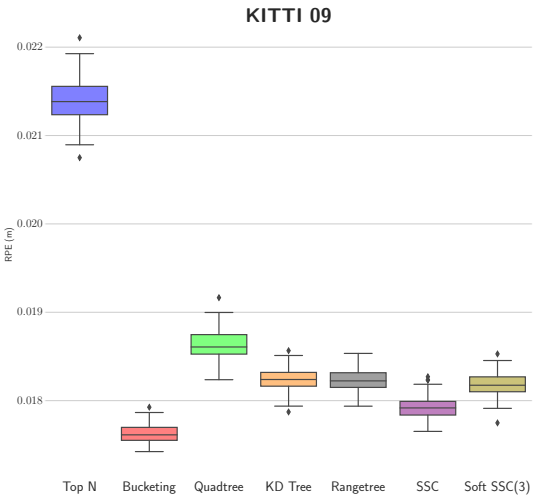

Figure 5.13: Unlike in the other KITTI sequence, bucketing produces the smallest error in KITTI 09.

Top N produced the worst result by far in the KITTI dataset. Figure 5.12 shows the results of the KITTI 07 sequence, the mean error of each distribution in KITTI 09 can be seen in figure 5.13. The often featured shrubbery and trees in the drives through rural areas add quite a bit of features that score highly, but will likely not be found again in subsequent frames, or might be erroneously matched with a pixel found in leafs in a different position. KITTI 07 was also one of two sequences that didn't feature the bucket approach as the clear frontrunner. Also notable is that in KITTI 09 Soft SSC was outperformed by its original variation for the first time.
The raw mean relative pose error values per distribution and sequence can be seen in table 5.2.

Mean relative pose error (m)						
	EuRoC MH3	EuRoC V2_02	TUM room	TUM l_o_h	KITTI 07	KITTI 09
Top N	0.0567882	0.0391117	0.00911439	0.00470948	0.0147764	0.0213971
Bucketing	0.0560878	0.037983	0.0079346	0.00422593	0.0117181	0.017625
Quadtree	0.0568514	0.0395121	0.00866964	0.00429142	0.0117981	0.0186259
Kd Tree	0.0563811	0.0399823	0.00859399	0.00429504	0.0115217	0.0182486
Rangetree	0.0565763	0.0398669	0.00850547	0.00429899	0.0114495	0.0182393
SSC	0.0566291	0.0398741	0.0084975	0.00429445	0.0114832	0.0179251
Soft SSC	0.0564171	0.0393469	0.0084004	0.00419235	0.0111961	0.01818

Table 5.2: Raw error values of distributions in all sequences.

Mean computation time for distribution only (ms)						
	EuRoC MH3	EuRoC V2_02	TUM room	TUM l_o_h	KITTI 07	KITTI 09
Top N	0.16	0.09	0.08	0.091	0.152	0.163
Bucketing	0.558	0.329	0.303	0.308	0.459	0.504
Quadtree	1.673	1.292	1.106	1.161	1.513	1.602
KD tree	8.862	4.292	3.488	3.756	6.614	8.573
Rangetree	7.407	5.298	4.581	5.124	7.091	6.9
SSC	2.057	1.537	1.302	1.333	1.541	1.694
Soft SSC	1.512	1.189	1.037	1.004	1.403	1.383

Table 5.3: Average times for distribution computation per sequence

In addition to the synthetic tests of the running times of the distribution methods we also looked at how long each distribution took to select the keypoints in the real experiments. The times are the average of the computation time across all images in a sequence and 10 iterations per sequence. They can be seen in table 5.3.
Soft SSC performs better than in the synthetic tests, but otherwise the results are more or less consistent with what one would expect after looking at the synthetic experiment. Top N and bucketing are very fast, while the tree based ANMS algorithms take a very long time. The rangetree performs worse when the average of the input is smaller, since building it takes longer, but overall both methods are not a good option in realtime SLAM.

Figure 5.14 shows the error of all sequences next to each other. Overall, also taking computation time into consideration, bucketing seems to be the clear winner. Even though the proposed improvement upon the ANMS approach, Soft SSC, performs better than the other ANMS methods and even slightly outperforms bucketing in two sequences, the error difference in the sequences that it does perform best in is minute.
The bucketing approach is computationally very inexpensive and produces consistently good to great results, which is why based on all the data we have accumulated across our experiments it would be the method we would recommend for use in feature-based SLAM systems. Soft SSC can be experimented with as an alternative approach in certain sequences, but the results are a bit less consistent. Bucketing features very low variance in its results, while Soft SSC produces outliers slightly more often. Soft SSC also requires the user to deal with the adjustment of the threshold parameter, which might be a possible issue with a different scoring function.
These results do not align with the findings of the work that originally proposed the ANMS-algorithm [1]. Their results, which were based on experiments using the KITTI dataset only, showed bucketing consistently behind all ANMS-variants.

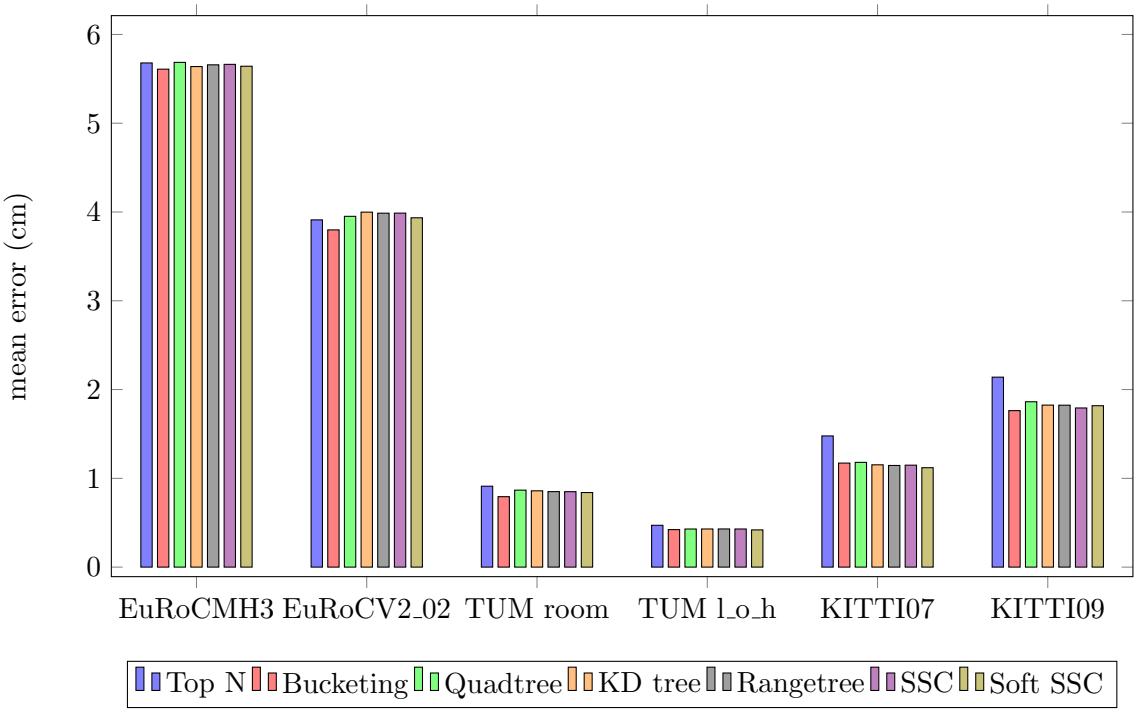

Figure 5.14: Overview of mean error per distribution and sequence. At this scale it is very clear that most results are very close together.

Chapter 6

Summary and Future Work

We have described how our feature detector was integrated into ORB_SLAM2 and examined how to select the best of the detected keypoints. For this purpose we presented various methods that would distribute the keypoints more or less evenly across the image.
Our own extension of the adaptive nonmaximal suppression algorithm presented in [1], Soft SSC, proved to outperform the other implementations of this algorithm in most cases.
Still, the experiments showed that one of the simplest approaches, bucketing, also potentially produces the best results (and did so in most of the test-sequences). Considering that it is also much less computationally expensive than the more complex methods our recommendation for future implementations of feature-based SLAM systems would be to distribute their features using this approach. To rule out overperformance of bucketing based on specific characteristics of the datasets involved one could run additional tests on other datasets in order to confirm the results.

Overall, the different approaches only produced very slight variations in relative pose error. For this reason, as well as the fact that even the simple methods performed very well, we would argue that future research in this direction is not the way to further improve upon results of SLAM.

Bibliography

[1] Oleksandr Bailo et al. "Efficient adaptive non-maximal suppression algorithms for homogeneous spatial keypoint distribution". In: *Pattern Recognition Letters* 106 (Feb. 2018). DOI: 10.1016/j.patrec.2018.02.020.

[2] Jose Luis Blanco and Pranjal Kumar Rai. *nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees*. https://github.com/jlblancoc/nanoflann. 2014.

[3] Michael Calonder et al. "BRIEF: Binary Robust Independent Elementary Features". In: *Proceedings of the 11th European Conference on Computer Vision: Part IV*. ECCV'10. Heraklion, Crete, Greece: Springer-Verlag, 2010, pp. 778–792. ISBN: 3-642-15560-X, 978-3-642-15560-4. URL: http://dl.acm.org/citation.cfm?id=1888089.1888148.

[4] OpenCV Foundation. *Open Source Computer Vision Library*. https://github.com/opencv/opencv. 2017.

[5] Andreas Geiger et al. "Vision meets Robotics: The KITTI Dataset". In: *International Journal of Robotics Research (IJRR)* (2013).

[6] Michael Grupp. *evo: Python package for the evaluation of odometry and SLAM.* https://github.com/MichaelGrupp/evo. 2017.

[7] C. Harris and M. Stephens. "A Combined Corner and Edge Detector". In: *Proceedings of the 4th Alvey Vision Conference*. 1988, pp. 147–151.

[8] David Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60 (Nov. 2004), pp. 91–110. DOI: 10.1023/B:VISI.0000029664.99615.94.

[9] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós. "ORB-SLAM: A Versatile and Accurate Monocular SLAM System". In: *IEEE Transactions on Robotics* 31.5 (Oct. 2015), pp. 1147–1163. DOI: 10.1109/TRO.2015.2463671.

[10] Raúl Mur-Artal and Juan D. Tardós. "ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras". In: *IEEE Transactions on Robotics* 33.5 (2017), pp. 1255–1262. DOI: 10.1109/TRO.2017.2705103.

[11] Paul L. Rosin. "Measuring Corner Properties". In: *Comput. Vis. Image Underst.* 73.2 (Feb. 1999), pp. 291–307. ISSN: 1077-3142. DOI: 10.1006/cviu.1998.0719. URL: http://dx.doi.org/10.1006/cviu.1998.0719.

[12] Edward Rosten and Tom Drummond. "Machine Learning for High-Speed Corner Detection". In: *Computer Vision – ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006. Proceedings, Part I.* Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443. ISBN: 978-3-540-33833-8. DOI: 10.1007/11744023_34. URL: https://doi.org/10.1007/11744023_34.

[13] E. Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. Nov. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht wird und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, den 02.10.2019

(Ralph Gelnar)