Faster-than-SIFT Object Detection

Borja Peleato and Matt Jones
peleato@stanford.edu, mkjones@cs.stanford.edu

March 14, 2009

1 Introduction

As computers become more powerful and video processing schemes improve, the problem of efficiently organizing and searching through videos becomes increasingly important. While the question of searching for objects in preprocessed or indexed videos is an important one, so is the problem of searching for objects in realtime video that has not been indexed. This question of searching realtime video has many applications, especially in the field of robotics and autonomous vehicle navigation. It can be further broken down into problems of searching for known objects (e.g. the backs of cars or pedestrians), or searching for new objects on the fly (e.g. returning to a specific location). In the former case, an expensive training phase for the object recognizer is not particularly bad, as the training can happen offline. However, in the latter phase, online training and subsequent search (classification) must be fast.

One standard way to do object search is using SIFT descriptors to identify and match parts of query images and candidate images. However, this approach can be slow for classification, and could also be faster at training. Other methods optimize for classification, but at the detriment of training time. We propose the use of generic trees for realtime object search, and improve on the classification time taken by SIFT approximately by a factor of 5. Our approach also supports very fast training, taking no longer than it takes to search for an object in one candidate image.

This paper is organized as follows: Section 2 provides the background and an overview of the previous related work. Section 3 explains in detail our proposed scheme, before going into the analysis and results in section 4. Finally, section 5 reviews the main features of our method and gives some possible directions for future work.

2 Prior Work

Since David Lowe released his now widely known method for image description and matching (SIFT [4]) an increasing number of open problems have appeared in the image processing field. SIFT (and its more simplified version SURF by Bay [1]) is capable of reducing a set of images to a few descriptors and finding matches between them based only on those descriptors. Unfortunately, its

efficiency and viewpoint invariance is not enough for some applications, such as fast object-finding in videos.

This problem has been tackled by Sivic and Zisserman in [8], where they propose a "video google" scheme capable of searching for a certain object in a video sequence. The main difference with our current problem is that they try to find the objects even in very disfavorable conditions. They therefore need to deal with severe affine and perspective transformations and a wider range of scales. Once the keypoints have been found, the patches over which the descriptors will be computed undergo the "rectification" iterative algorithm proposed in [6]. However, this algorithm is very time-consuming, and needs to be independently run on every keypoint and patch. In consequence, this algorithm is not suitable for an online application. On the other hand, they used a very efficient approach for organizing and searching through their database: the vocabulary tree proposed by Nister and Stewenius in [7]. Our initial idea was to use this same approach to organize our database, but the number of descriptors that need to be matched to find a single object is small, so it ended up being more efficient to do pairwise comparison. For more discussion of performance, see Section 4.3.

More recently, Calonder, Lepetit and Fua published in [2] a new descriptor capable of solving this problem. Their original goal was to enable a robot to navigate while learning the surrounding keypoints in an online fashion. Their scheme is capable of efficiently learning a virtually unlimited number of new descriptors. It is based on a previous paper by Lepetit, Lagger and Fua [3] proposing a randomized tree classifier for feature matching. We implemented this approach and improved it to solve the same problem tackled by Sivic and Zisserman, namely object recognition in videos.

3 Approach

Our approach to the problem was similar to that in Calonder 08, with some optimizations. It can be broken down into three parts: extracting a base set of image patches from a set of arbitrary images, building and training a random tree classifier on that base set, then searching for an arbitrary query image in arbitrary video. However, understanding the use of the random tree classifier is essential to understanding the base set extraction and search, so we will discuss its design before that of the other two.

3.1 Random Tree Classifiers

3.1.1 Overview

A random tree classifier, in the general sense, takes an input of some sort, traverses a randomly-generated (binary, in our case) tree with it, and outputs the probabilities that that input is in each class that the tree was trained with. The tree traversal works as follows: the input starts at the root of the tree and is examined at each node by some function that makes a binary decision of left or right. If the function outputs left, the input travels to the left child of the current node, otherwise

it travels to the right child of the current node. Each of the leaf nodes represents the posterior probability that an input that reached that node is a member of each possible class, given that it reached that node. Therefore training and classifying are both very similar, and very simple.

Training consists simply of traversing the tree with many instances of each class, and keeping counts of the number of times an input of each class has reached a given leaf. After the training phase is over, these leaf histograms can be normalized such that they contain the posterior probabilities of each class, by changing the count in each class's bin to the percent of inputs that reached that node that were of that class. Storing the an additional scalar representing the total number of inputs that reached that leaf would then allow trivial online learning. Classifying then involves simply traversing the tree with an input and returning the posterior probabilities of each class associated with the leaf it ends up in.

The performance of random trees can be improved by combining many of them. This simple extension takes an input, classifies it with some number of random trees, and returns the average of the classification probabilities of each tree. We found that performance improved when using around 70 trees instead of a single tree. For the rest of the paper when we refer to the plural "random trees" this is what we mean.

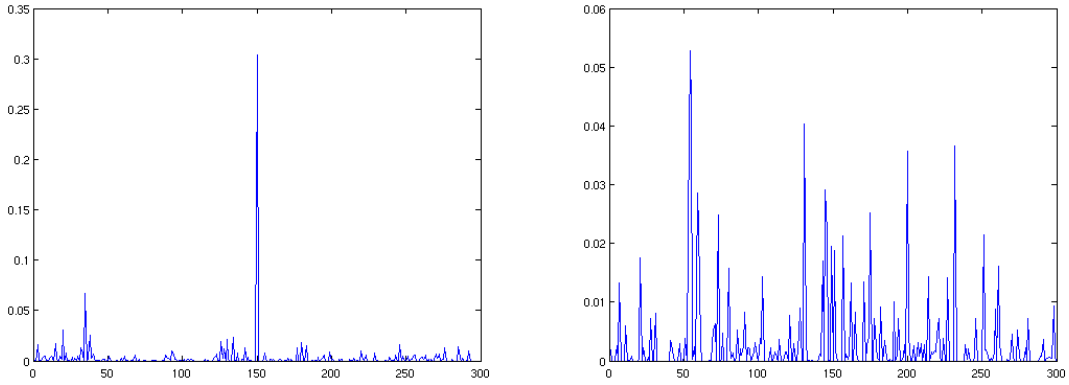
3.1.2 Use in Computer Vision

The classic use of random tree classifiers in computer vision comes from Lepetit 06. The input is an image patch of fixed size, and the classes correspond to image patches with which the tree was trained. Instances of each class can be generated easily by making synthetic (projective and affine) transformations of the image and taking patches around the transformed keypoint positions. The decision at each node is simply whether the pixel intensity at one consistent but randomly-chosen point in the given patch is greater than the pixel intensity at another consistent but randomly-chosen point in the patch. Trees are trained on synthetic transformations of the patches around each of the points that represent the classes. We expand on this implementation, as discussed in Section 3.4. In our experiments, training took on the order of minutes for 70 trees of depth 12.

3.2 Generic Trees

Calonder 08 discussed a way to build generic trees that can be used to identify new query objects without requiring expensive re-training. The generic trees they discuss work as follows. First, they train random trees on a "base set" of patches randomly chosen from a set of images arbitrarily related to the query images or the video. The fact that these base set patches are unrelated to query images or video is what makes these trees generic. Our selection of these base set patches was somewhat more advanced and is discussed further in Section 3.3.

Once the generic trees have been trained on the base set, signatures for the patches around keypoints in the query image and in candidate frames from the video can be extracted. Signatures in the context of generic tree classifiers are defined to be the response of the trees to patches from



(a) Response of generic trees to point in base set (b) Response of generic trees to point in query image

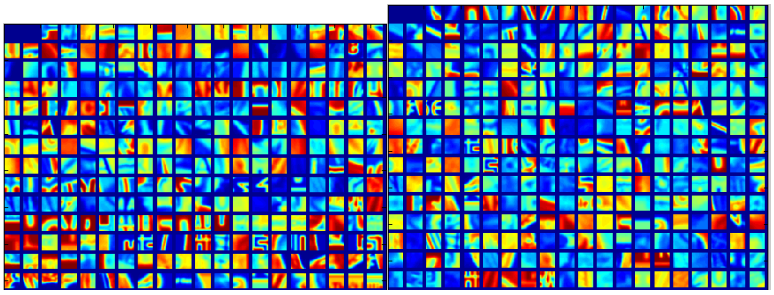
Figure 1: Examples of signatures from the base set and from a query image. As expected, when a point on which the trees were trained is passed through, there is a strong response for its class (150 in this case). When a point from a query image (unrelated to the base set) is passed through, it has multiple peaks.

images not in the base set. They generally have a few peaks and some lower-level noise that is filtered out. See Figure 1b for an example. The signatures generated by these generic trees are robust to transformations, as was the original classifier built from the tree. See Figure 5 for an example. Once signatures have been extracted from the query images and a candidate frame from the video, all that remains is finding matches, which can be done in a number of ways, as with SIFT features.

The key here is that the signatures for query or candidate image keypoints are relatively robust to transformations. As such, for example, a front-on query image such as the poster we used suffices for finding occluded or transformed versions of it in video.

3.3 Extracting a Base Set

In order to train our generic tree classifier, we need a diverse set of base images with which to train the trees initially. Calonder 08 suggested using landscape images and randomly choosing 300 keypoints with the only restriction being that they were at least 5 pixels apart. We started out using landscape images from around Stanford's Dish, but found that even after filtering out points more then 5 pixels apart, the patches were rather homogeneous. This homogeneity led to poor classifier performance on the base set in addition to poor generic tree performance. We then tried extracting points from a number of CD covers in addition to the landscape images, but still found some homogeneity. To enforce heterogeneity, we first extract many keypoints (750 worked well) from 30 images of CD covers and 10 landscape images, train random trees on them, and the remove patches which get poor responses (i.e. low spikes for their respective classes or many spikes



(a) Some base set patches before refining

(b) The base set patches after refining

Figure 2: Examples of the patches used for training before refining and after refining them. Note that there is increased diversity after refining them.

for classes that are not their own) from the random trees. For an example of keypoints before and after refining, see Figure 2.

3.4 Training the Classifier

As mentioned in Section 3.1.2, training the trees involves running many instances of each class through the trees. We create these instances by making projective and affine transformations of the base set images and then extracting the patches around the same keypoints in the transformations. We do this on the patch around each of the 300 keypoints in the base set, producing 37,500 transformed patches.

After generating the transformations, we train the trees with them. It is these transformations that give classical random tree classifiers their invariance to the same transformations we trained the tree with. In the case of generic trees, they also help give consistent responses to similar non-base set keypoints. As discussed in Section 3.1.1, training simply involves traversing the tree with each of the 37,500 patches and incrementing the appropriate class bin in the leaf it which it ends up in.

Though this training stage is relatively slow, it need only be done once for a generic tree. This is where one significant advantage of generic trees over classical random trees comes into play: with classical random trees, this stage would be necessary for every query image. With generic trees, it only needs to be done once. It is worth noting that the slowest part by far is extracting and transforming the base patches - the actual training of the tree takes up less than a quarter of the overall time. See Figure 8 for details.

There is one relatively simple but significant difference between the way we generate trees and the way Calonder 08 generated trees. They did work looking at how to generate an "optimal" tree that maximized the increase in information at each node, but found that it made training (specifically, tree generation) significantly longer, with little benefit for performance. We took this simplification a bit further, and rather than generating random comparison points for each node,

generate random comparison points for each *level* of the tree. i.e. all nodes at a given depth in a tree compare the same locations. This makes it much faster to compute which leaves a group of patches end up in - rather than running each through the tree individually, we can run all through the tree at once. While it does decrease the entropy of the tree, we can compensate by using more trees. In practice we were able to get similar performance with these simpler random trees, with much better speed. Were this to be implemented in C rather than the matrix-operation-optimized matlab, performance increases might not be as significant.

We experimented with the depth of trees and the number of trees to use, and found that 70 trees of depth 12 provided a good tradeoff between speed and performance. It is worth noting that if the number of leaf nodes (2^{depth}) exceeds the a few times the number of training patches, the probabilities will be so sparse as to be useless. There is also, of course, exponential blowup in the memory necessary to store the leaves with respect to the tree depth.

3.5 Searching the Video

Once the trees have been trained, we can begin searching the video for instances of the query image. The procedure for getting signatures is the same whether we are looking for those in a query image or in a video frame. First, we extract all the keypoints in the image using difference of gaussians with 6 different levels. This involves doing gaussian convolutions of different radii on an image, and choosing points that are local maxima in 3 space, within their own frame and the frame more convolved and less convolved than it. See Figure 3 for a visual explanation of this from [5]. Once keypoints have been extracted, we select the patches around each of them and generate the signatures for all patches.

Once we have signatures for the query image (only necessary to do once per query image), we can extract signatures in the same way for each frame image. At this point, we can treat the generic tree descriptors in the same way as SIFT descriptors and look for matches in some way. We chose to search for matches using cosine similarity, detecting a match when the second highest similarity for a given query point was less than 0.8 times the highest similarity. We experimented with a few methods of reducing false positives, including a lightweight geometry-checking algorithm and discarding locations with an absolute similarity below a certain threshold, but found better results without these enhancements.

4 Results

4.1 Setup

For testing our generic tree implementation, we searched for a poster in compressed episode snippets from the TV show *Friends*. We chose this query image because it came from a different source (we did not extract it from any frames in which we were searching), it had enough structure to be reasonably easily to get features out of, and appeared with different degrees of occlusion throughout

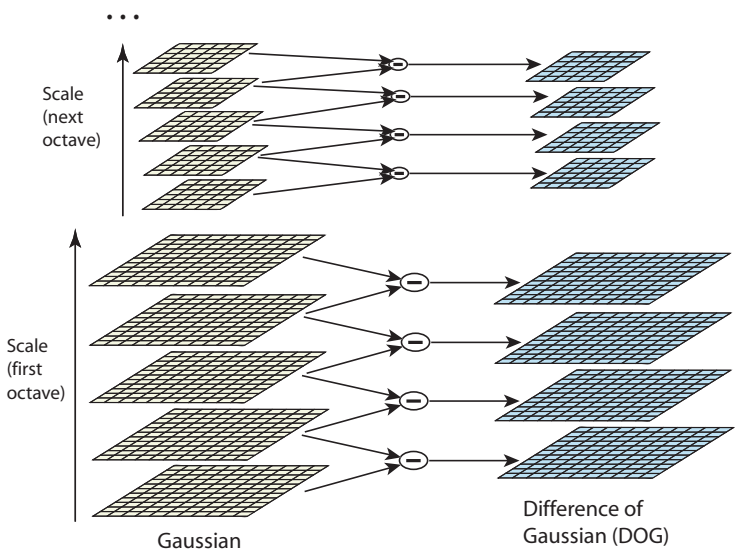


Figure 3: A visualization of the difference of gaussians technique used for keypoint detection in all images. Taken from [5].

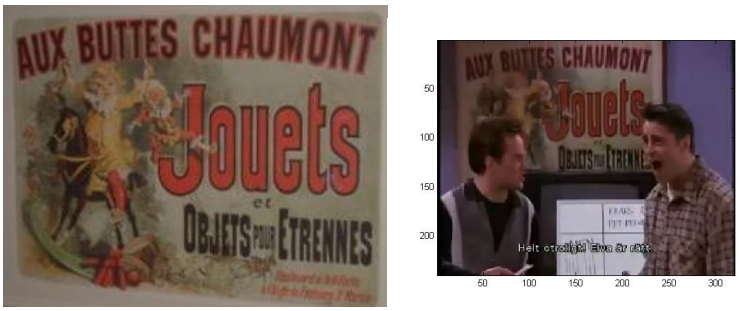
the clips we had. The query image can be seen in Figure 4a, and one example of a frame it shows up in can be seen in 4b.

4.2 Comparison

We tested our scheme against an open source SIFT implementation by Andrea Vedaldi available in [9]. The base of this implementation is in Matlab, but some of the most expensive functions have been implemented C and are called from matlab as mex code. Our implementation has all been done in Matlab, causing a clear disadvantage for us in terms of speed. On the other hand, we have tried to optimize our implementation for the problem under study, whereas the publicly available SIFT implementation was built for a more general framework. It could be argued that both implementations are therefore not 100% comparable, but we nevertheless plot them together here as a benchmark for comparison.

Our first step was to verify that the generic tree signatures proposed by Calonder et al. [2] were indeed invariant to affine transformations, rotations, etc. To check this, we designed an experiment comparing the responses of a patch under different deformations. Figure 5 shows the signatures obtained for a patch under different transformations. It is clear that the signatures remain relatively stable despite the large transformations that the patch is suffering. Note that patches are *not* from the base set - the trees were not trained on them.

Figure 6 shows the number of matches found by our scheme and the number found by SIFT for a short video clip. Looking carefully, we observe that there is correlation between the peaks. There are, however, some peaks and plateaus in our scheme which do not appear in SIFT. We have



(a) The query image

(b) A frame containing the query image

Figure 4: The query image and an example of a frame that contains it.

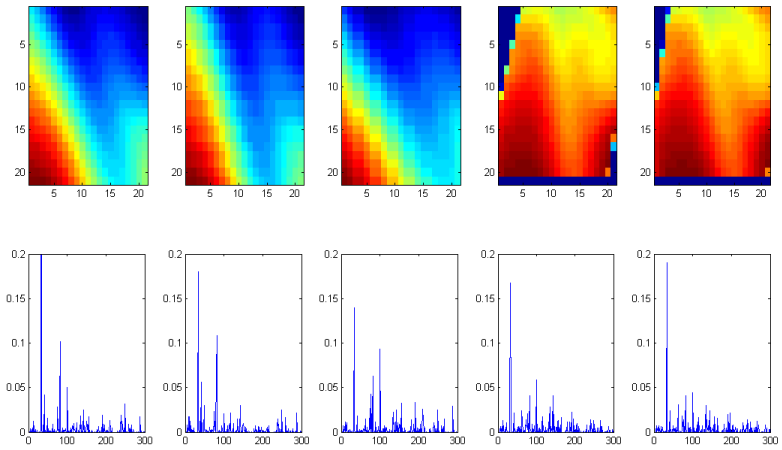


Figure 5: Signatures for a patch under different transformations.

observed that those generally correspond with sequences which are very rich in keypoints (such as a fence or a bookshelf in the background). We believe that with a some better tuning of the many parameters involved, in addition to more robust feature matching, we could reduce the rate of false positives.

4.3 Speed

Finally, our main goal was to improve the efficiency of SIFT. With some further work, we expect our algorithm to be able to run in real time while watching a video. Despite the fact that our implementation is not yet completely optimized, it turns out that it is already approximately 5 times faster than the reference SIFT implementation. Figure 7 compares the processing time per frame for both algorithms. Experiments were run on a Lenovo T61 laptop with a Intel T7700 2.4 GHz processor. We observe that we not only provide a better average time, but the variance with

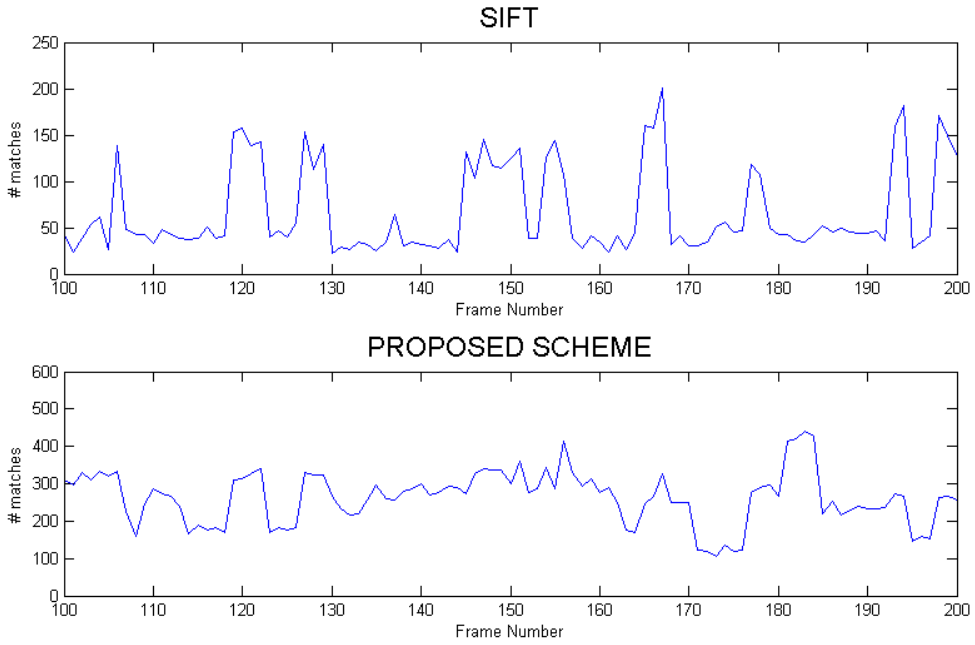


Figure 6: Number of matches between query image and given frame for SIFT and our generic tree implementation.

our method was also considerably lower than that with SIFT, which is a guarantee of performance stability regardless of the number of descriptors per frame or whether there is a match or not.

One advantage of working in Matlab is its profiling capabilities. The profiler made it possible to tune a few parts of our implementation, but also determine and report where the slow steps were. As discussed briefly earlier, the actual classification and training phases are very fast when compared to other operations that must always be done for similar types of descriptors.

Figure 8 was mentioned above and shows that the actual tree training part of the training phase is a small percentage of the total time. The other parts of that phase are somewhat specific to random trees, in contrast to the classification step. During classification, the time spent is dominated by actually extracting keypoints - something that is necessary for the majority of image search algorithms. As is shown in Figure 9, over 75% of the time classifying is spent extracting the keypoints. In fact, the actual tree traversal time - that which characterizes this type of search comprises only 13% of all time spent, including tree traversal and fetching of signatures. The fetching of signatures could likely be optimized to take about 25% as long if some of the underlying implementation was changed to store things in columns instead of rows, but we did not take the time to do this because classification time was so dominated by keypoint extraction. An optimized version of this step would get our algorithm most of the way to running in real time.

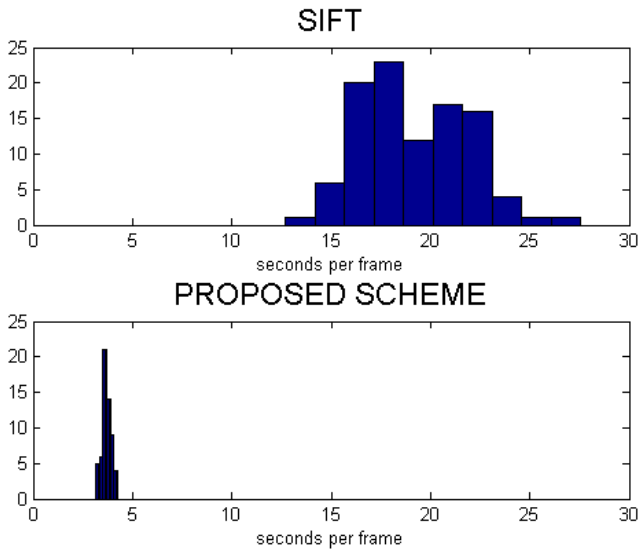


Figure 7: Seconds per frame for SIFT and our generic tree algorighm.

Action	Time	Percent of training time
Extract base patches	25.7 seconds	40%
Generate transformation	23.6 seconds	36 %
Train trees	6.9 seconds	11 %
Refine and retrain	8.5 seconds	13 %

Figure 8: How long each of the training steps took. As can be seen, the training time is dominated by extracting patches and generating transformations of them. The actual training of the tree represents only a small percent of the training time.

4.4 Online Resources

An example video of our results is available at http://www.stanford.edu/~mkjones/generic-tree.mp4. In it, frames identified by our algorithm as containing the query image are green while those not are black and white. We also hope to open-source our implementation of random tree classifiers and generic trees. The code will be available at http://www.stanford.edu~mkjones/ when sufficiently cleaned up.

5 Discussion

We were able to improve the reference SIFT algorithm's speed by 5 times with an implementation of a type of generic tree classifier, while also significantly decreasing the variance in time spent comparing images. This speed increase was unfortunately at the detriment of accuracy, as our algorithm ended up with more false positives than SIFT. We suspect further work tweaking parameters of

Action	Percent of classification time
Find DoG (not including finding gaussians)	41%
Find gaussians	34%
Extract leaf probabilities (given leaf indexes)	**10%**
Get patches (overhead)	4%
Traverse tree	**3%**
Refine keypoints	3%
Average tree outputs / overhead	2%
Other	5%

Figure 9: How long each of the classification steps took (ignores overhead like reading frames from disk). As can be seen, the classification is dominated by extracting keypoints and extracting the leaf nodes' probability distributions. The actual tree traversal accounts for less than 3% of the total CPU time. Approximate time per frame on the above-mentioned hardware was 3.5 seconds. On faster hardware, average time per frame was 1.1 seconds.

the algorithm could significantly reduce the false positive rate, and implementation of some parts (specifically feature detection) in C would improve speed considerably. Further work could look at specific applications to vehicle navigation, increasing accuracy by identifying common features of all possible query images and assuring they are present in the base set, pre-processing video as it comes in to recognize scenes and only search one frame per scene, using more effective and / or faster similarity mechanisms, or generating indices of images to they will be easily searchable in the future.

References

[1] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Lecture Notes in Computer Science*, 3951:404, 2006.

[2] M. Calonder, V. Lepetit, and P. Fua. Keypoint Signatures for Fast Learning and Recognition? In *Proceedings of the 10th European Conference on Computer Vision: Part I*, pages 58–71. Springer, 2008.

[3] V. Lepetit and P. Fua. Keypoint recognition using randomized trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(9):1465–1479, 2006.

[4] DG Lowe. Object recognition from local scale-invariant features. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, 1999.

[5] D.G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[6] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, 22(10):761–767, 2004.

[7] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *Proc. CVPR*, volume 5, 2006.

[8] J. Sivic and A. Zisserman. Video Google: A text retrieval approach to object matching in videos. In *Ninth IEEE International Conference on Computer Vision, 2003. Proceedings*, pages 1470–1477, 2003.

[9] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. http://www.vlfeat.org/, 2008.