An Analysis of SIFT Object Recognition with an Emphasis on Landmark Detection

by

Benjamin Charles Ross

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

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

Author .....
Department of Electrical Engineering and Computer Science
July 1, 2004

Certified by .....
Trevor J. Darrell
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

An Analysis of SIFT Object Recognition with an Emphasis on Landmark Detection

by

Benjamin Charles Ross

Submitted to the Department of Electrical Engineering and Computer Science
on July 1, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

In this thesis, I explore the realm of feature-based object recognition applied to landmark detection. I have built a system using SIFT object recognition and Locality-Sensitive Hashing to quickly and accurately detect landmarks with accuracies ranging from 85-95%. I have also compared PCA-SIFT, a newly developed feature descriptor, to SIFT, and have found that SIFT outperforms it on my particular data set. In addition, I have performed a relatively extensive empirical comparison between Locality-Sensitive Hashing and Best-Bin First, two approximate nearest neighbor searches, finding that Locality-Sensitive Hashing in general performs the best.

Thesis Supervisor: Trevor J. Darrell
Title: Associate Professor

Acknowledgments

To my parents and friends, for always being there for me. To Brian, Naveen, and Ellie, for spending late nights with me while I worked on my thesis. To my thesis advisor, Trevor Darrell, for his endless support and help. To Greg and Kristen for their assistance with LSH and BBF. To Mike, Louis-Philipe, Kate, Mario, Neal, Ali, and Tom, for dealing with my numerous questions. To Valerie and Jess for graciously allowing me to use their photos for use in my landmark recognition system. David Lowe for his assistance with the implementation of SIFT. To Yan Ke and Rahul Sukthankar for their assistance with PCA-SIFT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The raw computational power of the computer has increased tremendously over the past few years. For example, the number of transistors in the average processor has consistently doubled every 2 years, resulting in CPU speeds that were unheard of even just a couple years ago. One of the major benefits of this increase in computational power has been a steady rise in the number of computer vision applications. Computer vision problems once thought impossible to solve in any reasonable amount of time have become more and more feasible.

Perhaps one of the most important examples of such problems is efficient, accurate object recognition. That is, given an image of a scene, the computer must return all objects that appear in that scene. Furthermore, if no objects from the scene are "known" to the computer, it must not detect any objects.

There are a wide variety of approaches to this problem. Some of them attempt to extract a *global* description of the entire image, and utilize this description as a means of matching the particular object. This is a useful approach when a model of the background behind the object is known, so that it can be thus subtracted from the image, making detection less prone to errors.

To illustrate this global approach, take the problem of determining whether a face exists within an image. A global method would determine if the image in its entirety implies that a face exists within it. One might, for example, have a template of a face, and attempt to match that entire template to the image.

Another way to approach the problem is to extract local salient regions, or *features*, on

the model image, and determine if enough of these appear on a given query image. More specifically, something known as a *feature extractor* is used to extract points on the image which are likely to be salient across various camera viewpoints. The image is then sampled around every feature, forming *feature descriptors*. On the query image, the same operation is performed, and each feature descriptor extracted from it is matched to the closest feature descriptor on the model images. If enough features are found that they form a consistent representation of an object, a match is declared. This method is known as *Feature-Based Object Recognition.*

In our example of determining whether a face exists within an image, the feature based approach would be to extract various salient features on the image, such as a nose, a set of eyes, ears and a mouth. Then, it would match those features to a model of a face, all the while ensuring that the features are in their correct orientations.

One might immediately assert that the local feature-based recognition model is better, because by dividing the image into smaller regions, detection can be made more robust to changes in pose. Furthermore, if there is no available background model, global detection becomes even worse, because the entire background in the image can become incorporated into the global feature for the image.

Yet there are a few advantages to a global model of object recognition, especially if some of the background can be subtracted from the image. Most importantly, the feature-based model relies on there being a large number of extractable features on the object in question. For example, applying feature-based object recognition to the task of recognizing an apple would not be very fruitful, as there are very few salient features on an apple that could be extracted. In such a case, feature-based object recognition often fails, and often a global recognition approach would produce better results.

Nevertheless, this paper is devoted to the study of the feature-based approach to object recognition. In a small application, I have found that it is indeed the fact that objects with very few distinguishable features are difficult to recognize using a feature-based object recognition model. I have built an application that recognizes relatively salient objects quickly with accuracies of 85%. I concentrated on the detection of buildings and landmarks, however, this approach can easily be extended to recognizing other objects. In this particular application, I eventually use SIFT feature extraction and object recognition, developed by David Lowe [13] [11], although my initial recognition applications utilized other methods.

In the past, there has been a great deal of research in feature-based object recognition, yet the results were very limited in the number of objects it could recognize, and any large image deformation would produce extremely poor results. More often, a global method of object recognition was used, and feature extractors were mostly applied to the problem of tracking. Tracking involves extracting features from an image, then using those features as points to track using optical flow [7] [3]. Only recently, with the work of Schmid and Mohr [16], and Lowe [13], has feature-based object recognition been able to handle the large image deformations necessary for accurate and extensible object recognition.

Once it was realized that feature-based object recognition could be applied to large image deformations, there have been a tremendous number of interesting applications. Perhaps the most interesting application is Video Google [18]. Utilizing the same type of metrics used in text-based searching, Video Google attempts to create an efficient image-based search of a video sequence. Thus, a user can search for the occurrence of a particular object within a scene, and obtain results in descending order of their score. Rather than simply performing a brute force search of the object throughout every image sequence (which could take a tremendous amount of time), they employ a method of object localization in each frame by extracting regions based on maximizing intensity gradient isotropy. Thus a set of objects are obtained from each frame a priori, and searching can be done much more efficiently.

My thesis will concentrate on highlighting the four major stages of feature-based object recognition, all the while introducing other methods of approaching each one. These four stages are as follows:

1. **Feature Extraction**: Salient features are extracted from the image in question.

2. **Feature Description**: A local sampling around each feature, known as a *feature descriptor*, is obtained. It is often then represented in some fashion, for example by histogramming the values of the pixels in the region.

3. **Feature Matching**: For each feature in the query image, its descriptor is used to find its nearest-neighbor match among all stored features from all the objects in memory. This is often the most time-intensive step, and thus a faster, approximate nearest-neighbor matching algorithm is often used.

4. **Hypothesis Testing**: This step determines which of the matches are reasonable, and declares whether portions of the query image represent objects stored in memory. This

is normally done by finding an approximate affine transformation which transforms each feature on the model image into those on the query image.

The second chapter of this thesis will give a broad overview of both the first and second stages of feature-based object recognition, and will discuss my initial approaches when I first began work on this thesis. In this chapter, I also motivate the need for a scale-space representation of an image. As SIFT is deeply rooted in scale-space theory, this leads us into chapter three, where I introduce SIFT feature extraction, and thus begin again in detail with the first stage listed above. In chapter four, I describe the second stage by introducing the SIFT feature descriptor. In this chapter, I also compare SIFT's descriptor to a recently developed feature descriptor known as PCA-SIFT.

Chapter five then tackles the third stage of object recognition, feature matching, by describing two prominent approaches to approximate nearest neighbor matching, *Best Bin First Trees* (BBF), the method taken by David Lowe in his SIFT paper, and *Locality-Sensitive Hashing* (LSH). In chapter six, I describe hypothesis testing, the fourth and final stage of feature-based object recognition. I also discuss methods of maintaining a server of objects that can be queried to recognize images supplied by a user. In chapter seven, I show the results of an application that uses the optimal combination of SIFT and the other methods introduced in this thesis to perform landmark detection. Finally, I conclude with chapter eight, which summarizes my thesis and discusses possible future work in the field.

Chapter 2

Initial Approaches to Feature Extraction and Descriptors

This chapter discusses my initial approaches to feature-based object recognition, as well as a brief overview of feature extraction methods and feature descriptors in general. It will also motivate the need for a scale-space image representation that will improve the quality of image recognition by detecting features along multiple scales.

2.1 Feature Extraction

The goal of feature extraction is to determine salient points, or *features* on an image. These features are used to match against features in other images to perform object recognition (as discussed earlier). For any feature-based object recognition task, we wish that our features have the following properties:

1. **Salient:** Salient features will be those that are the most invariant to noise and changes in orientation. For example, in choosing features for a building, an ornament on the doorway would perhaps be a better than the doorway itself.

2. **Scale-invariant:** This means that the same features will be found regardless of how we might have scaled the image. Thus, the distance from the object to the camera should not cause a change in the features produced.

3. **Rotation-invariant:** If we rotate the image, we should find the same features. Note that this is different from rotating the camera when observing an object. We simply

mean here that if the 2-dimensional image of our object is rotated, then we should find the same features.

4. **Illumination-invariant:** We should find the same features regardless of the lighting conditions. This is a notoriously challenging problem in computer vision. A particularly difficult situation occurs when an object has a specular highlight (see figure 2-1[1]).

5. **Affine-invariant:** This effectively means that if we perform an affine transformation on the image of the object, the same features will be found. In other words, we can sheer, rotate, scale, and/or translate the image without any loss of the original features. We would like our feature extractor to be affine invariant because any reasonably small camera orientation change can be modeled as an affine transformation. (See Figure 2-2).

Figure 2-1: An example of an object with a specular highlight. Most feature-based object recognition methods will produce features on or around the specular highlight. Thus, if the object or lighting source is moved, the specular highlight moves, and features near or on the specular highlight will not match consistently with the features of the original object.

We will use these properties throughout the paper when discussing feature extraction methods, especially when determining whether SIFT is a reasonable feature extraction method.

My initial experiments used a feature extraction algorithm known as "Good Features to Track", developed by Jianbo Shi and Carlo Tomasi [17]. As the name of the algorithm implies, their approach was to find points on the image that are good for tracking purposes.

[1] Image from http://accad.osu.edu/~hcaprett/images/ac752_B_process/specular_highlight.jpg

(a) (b) (c)

Figure 2-2: Affine transformations used to simulate camera rotations: in (c), an affine transformation is applied to (a) to appear like (b).

More specifically, their algorithm outputs a set of salient points so that if the camera is moved, many of the same points will be produced. The general application of Good Features to Track is to determine salient features on an initial image, then track those features over time using optical flow. This is of course better than tracking every single point, since not every point at some frame I will be apparent at some later frame J.

2.1.1 Good Features To Track

Good Features To Track (GFT) [17] uses a simple affine model to represent an image J in terms of an image I:

$$J(\mathbf{Ax} + \mathbf{d}) = I(\mathbf{x}) .$$

A good tracking algorithm would have optimal values of $\mathbf{A}$ and $\mathbf{d}$, and thus minimize the dissimilarity ϵ taken over a window W surrounding a particular feature:

$$\epsilon = \int\int_W [J(\mathbf{Ax} + \mathbf{d}) - I(\mathbf{x})]^2 d\mathbf{x}$$

We wish to minimize this residual, and thus we differentiate with respect to $\mathbf{d}$ and set it equal to zero. Using the Taylor expansion $J(\mathbf{Ax} + \mathbf{d}) = J(\mathbf{x}) + \mathbf{g}^T(\mathbf{u})$, and [17], this yields the simple 2×2 system

$$\mathbf{Zd} = \mathbf{e},$$

where

$$\mathbf{Z} = \begin{bmatrix} g_x^2 & g_x g_y \\ g_x g_y & g_y^2 \end{bmatrix}$$

and

$$\mathbf{e} = \int\int_W [I(\mathbf{x}) - J(\mathbf{x})] \begin{bmatrix} g_x \\ g_y \end{bmatrix} d\mathbf{x}\,.$$

It turns out that if the eigenvalues of λ_1 and λ_2 of $\mathbf{Z}$ are sufficiently large, and also if λ_1 is sufficiently larger than λ_2, then the feature at $\mathbf{x}$ is a "good feature to track." The first condition very often implies the second, and thus, it is only required that

$$\min(\lambda_1, \lambda_2) > \lambda\,,$$

where λ is a predefined threshold.

2.1.2 Discussion

I found that GFT performed relatively poorly when applied to the task of feature-based object recognition. This is mainly due to its assumption that the deformations between adjacent frames are small [17]. In object recognition, it is often required that features be matched across relatively large image deformations. Thus, when the change in camera orientation was large, very few features maintained their original locations on the object (see figure 2-3).

Although it is possible to extend the formulation of GFT to model the feature window deformations, thereby allowing larger image deformations, [17] states that doing so would lead to further errors in the displacement parameters of each feature. Thus, using GFT for use in feature-based object recognition cannot possibly produce reasonable results.

Regardless, I continued to experiment with Good Features to Track, not to produce an application with any reasonable results, but simply to explore the realm of feature-based object recognition. This led me to experiment with *feature descriptors*, which are aptly named local descriptions of the image surrounding each feature.

(a) (b)

Figure 2-3: As expected, Good Features to Track produced relatively poor features for object recognition. These features were produced with a λ threshold of 0.001, and it was enforced that each feature be at least within 10 pixels of each other.

2.2 Feature Descriptors

Once we have a list of features, we must appropriately sample the image around it. This sampling of pixels, when represented in some fashion, is known as a *feature descriptor*, because it *describes* the local image properties of that feature.

My initial feature descriptor was a 2-dimensional Hue-Saturation (H-S) color histogram of the circular patches surrounding each feature. Matching was done by correlation of each histogram. The descriptor was also normalized for illumination invariance. This worked reasonably well, as the histogram model was relatively robust to noise. Furthermore, it has been shown that the hue and saturation values of an image are relatively invariant to illumination changes, albeit only to pure white light.

However, when more than a few objects were added to the database, the object matching began to fail miserably, and performance degraded to less than 10%. This makes sense, since although the H-S model is relatively robust to noise, it also allows for a great degree of false positives. This is due to the fact that a histogram of color values, by nature, under-describes an image patch. That is, if we simply rearrange the order of pixels in the image patch, the same histogram will be produced.

For this reason, I attempted to make the descriptor more specific by incorporating Hu Moments into the description of the local patch.

2.2.1 Hu Moments

Hu Moments [8] are a set of seven orthogonal moment invariants that are proven to be rotational-invariant, scale-invariant, and reflection-invariant. They have been used in the past in simple character recognition [20] and other simple feature-based recognition problems.

The set of 7 moments can be computed from the normalized centralized moment up to order three, and are shown below as I_1 through I_7:

$$I_1 = \eta_{20} + \eta_{02}$$

$$I_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$

$$I_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$$

$$I_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$

$$\begin{aligned} I_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta 03)^2] \\ &+ (3\eta_{21} - \eta 03)(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \end{aligned}$$

$$\begin{aligned} I_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta 03)^2 \\ &+ 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})] \end{aligned}$$

The 7th moment's sign is changed by reflection, and thus is not reflection invariant per se. Thus, for my use, this was the most important moment, since it is not reasonable to allow reflection in feature descriptors:

$$
\begin{aligned}
I_7 &= (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\
&+ (\eta_{30} - 3\eta_{12})(\eta_{12} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}
$$

These moments were combined with the H-S histogram to provide a relatively robust feature descriptor. Matching between descriptors was done by taking a weighted sum of the differences of each Hu moment, summed with a weighted correlation measure of the H-S histograms of the features in question. The optimal weights to use were determined experimentally.

2.2.2 Discussion

Unfortunately, this still proved to be inadequate. There are numerous issues with my approach. As mentioned earlier, using Good Features to Track to produce any reasonable object-recognition application is not very realistic. In addition, it has been shown that both Hu Moments and color histograms do not perform well compared to a host of other prevalent feature descriptors [14].

Eventually, this lead me to discover the paper SIFT [13] [11], which had claimed to produce significantly better results with feature-based object recognition. I introduce SIFT in chapter 3, but before discussing it, it would be worthwhile to discuss something known as *Scale-Space Theory*, as SIFT's implementation is deeply rooted in it.

2.3 Scale Space

Suppose we are at the edge of a forest, looking directly at it. (See Figure 2-4[2]) Initially, we might simply view this scene as a 2-dimensional wall, taking in the entire forest at once. We look closer, and begin to see individual trees. If we come closer, we see each individual leaf on the tree. At even closer inspection, we are able to see each vein on a leaf.

Each of these inspections occur at different scales. In viewing a scene, no one scale is more important than the other, and thus analysis of a scene should take all scales into account. In this light, we come to the notion of a scale-space representation of an image.

[2]Image taken from http://www.opl.ucsb.edu/grace/nzweb/pics/forest.jpg

Figure 2-4: A view of a forest that shows how scale space can be a useful parameter to add to a description of an image.

We add an additional parameter to our notion of an image, namely scale.

Thus, an image $I(x, y)$ will now include a scale value, and hence become $L(x, y, s)$. In our scale-space representation, $L(x, y, 0) := I(x, y)$. That is, at the finest scale, the scale-space representation will simply be the original image. Then, as the scale value is increased, details should disappear monotonically [10].

2.3.1 Gaussian Scale Space

What is the best way to formalize this concept of scale-space? What operation applied to the image will take us from a scale value of 0 to some arbitrary higher value? The answer came in 1984, when Koenderink described that, under a variety of assumptions, the scale-space representation of a two-dimensional signal must satisfy the *diffusion equation*:

$$\partial_t L = \frac{1}{2}\nabla^2 L = \frac{1}{2}(\partial_{xx} + \partial_{yy})L \text{ [10]}.$$

where $L(x, y, s)$ is the scale-space representation of the two-dimensional signal $I(x, y)$. The solution of the diffusion equation at an infinite domain is given by a convolution with the Gaussian kernel G.

$$G(x, y; t) = \frac{1}{2\pi t} e^{\frac{-(x^2+y^2)}{2t}}.$$

This implies that the most ideal scale-space operator is simply the Gaussian kernel. If we take an original image I and convolve it with a Gaussian kernel, i.e., blur it, the resulting image will be the image's description at a higher scale space level.

Intuitively, this makes sense: by blurring an image, we suppress fine-scale details on the image. If we blur it again, more details are suppressed. These operations represent the image's traversal through scale-space. Furthermore, we can simply let the imaginary "scale" value be the same as the covariance of the gaussian kernel used to blur the image. Refer to figure 2-6.

2.3.2 Pyramidal Gaussian Scale-Space

It has been shown beneficial not only to convolve the image with a Gaussian kernel, but also to halve the size of the image every k number of Gaussian blurs. This builds what is commonly known as a *pyramidal description* of the image.

The pyramid is defined as follows: at the i^{th} level, or *octave*, of the pyramid, we use the original image scaled by a factor of 2^{-i}. Within each octave, the image is blurred k times, in the same manner as described in subsection 2.3.1. See figure 2-5.

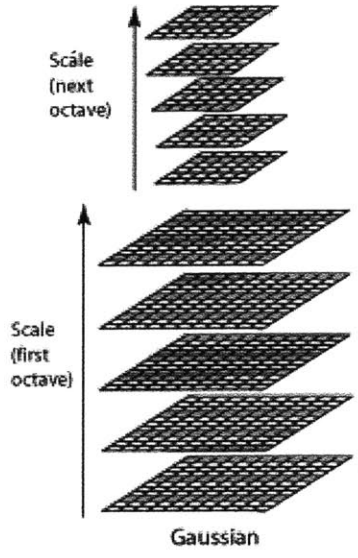


Figure 2-5: A 2-octave pyramidal description of an image, with 4 gaussian blurs per octave.

We wish for the effective "scale" to double each octave, and therefore, r, the ratio between the scale-value of each image is given by:

$$2 = r^k,$$

which, solving for r, becomes

$$r = 2^{1/k}.$$

We let the scale value of the initial image be σ_0, the initial covariance used to blur the first image in each octave. (This initial blurring may or may not be performed, depending on implementation). Then, the next scale values will be $2^{1/k}\sigma_0$, $2^{2/k}\sigma_0$, ..., $2^{(k-1)/k}\sigma_0$. In terms of r, this becomes $\sigma_0 r$, $\sigma_0 r^2$, ..., $\sigma_0 r^{k-1}$. Then, for the next octave, the first scale value is $2\sigma_0$, followed by $2 \cdot 2^{1/k}\sigma_0$, and so forth.

Note that the scale value is now no longer the same as the covariance level of the gaussian kernel used. It is only this way for the the first octave. For all subsequent octaves, the scale value doubles, while the covariance level starts at σ_0 at each octave and increases by a factor of r until it reaches $2^{(k-1)/k}\sigma_0$.

2.3.3 Applying Scale-Space to Feature Recognition

When we have a scale-space model of an image, features are no longer detected at the 2-dimensional coordinate (x, y) but at the 3-dimensional coordinate (x, y, s). When matching is performed between features on a query image and a model image, we thus match across all scales of a particular image. In this way, matching an image of a model from a great distance becomes easier, as we have already modeled that particular scale of the model in the scale-space representation of the image. Thus, by employing a scale-space representation, we make an attempt at ensuring the scale-invariant property of good feature-extractors mentioned earlier. The features for a particular image are not purely scale invariant, but rather a small subset of the features will likely contain the description of the image at the scale of the query image.

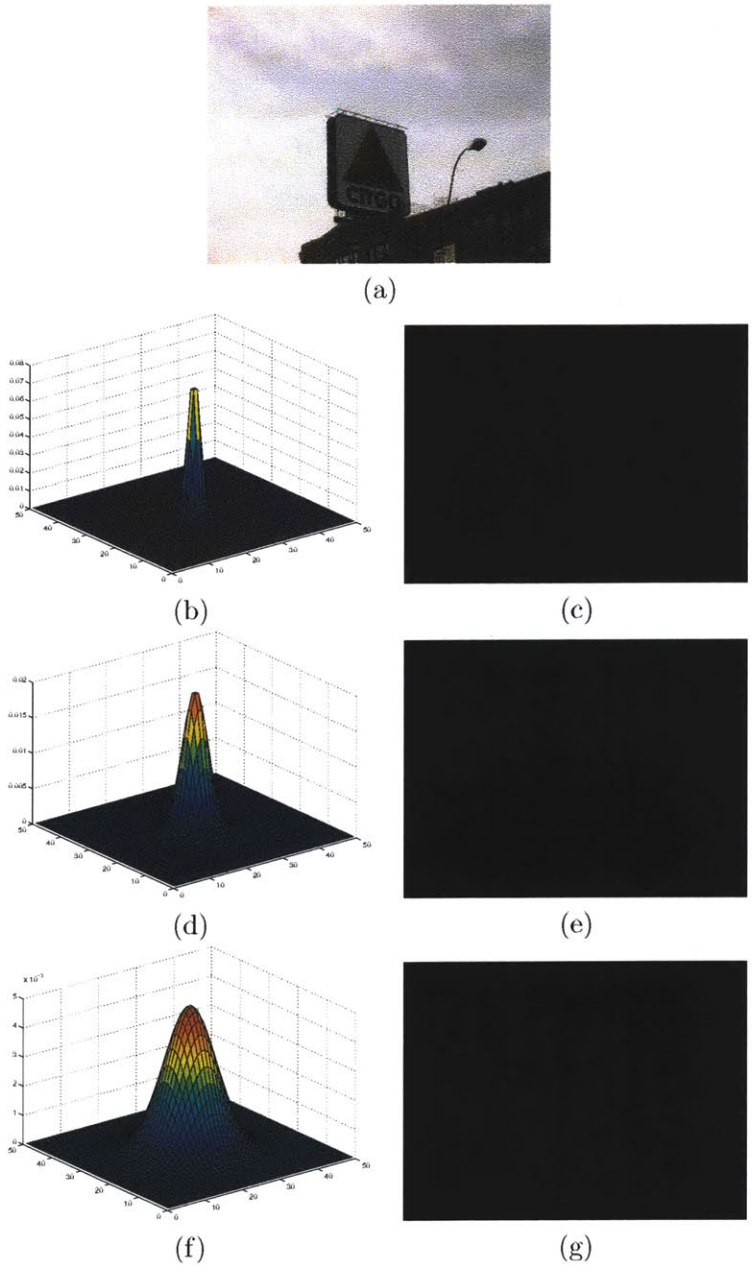


Figure 2-6: The citgo image blurred with Gaussian kernels of different variances. (a) shows the original citgo image. (b) shows an image of the first gaussian kernel, with variance $\sqrt{2}$. (c) is the result of blurring the citgo image Gaussian kernel with variance $\sqrt{2}$. (d)-(g) follow similarly with variance levels $2\sqrt{2}$ and $4\sqrt{2}$.

2.4 Summary and Motivation for SIFT

In this chapter I have discussed my initial approaches to feature-based object recognition, and explored various different feature descriptors. I have also described how a scale-space representation to an image can help to identify features at various scales so that feature extraction can become more scale-invariant. In the following chapter, I introduce and give an overview to SIFT, developed by David Lowe, which utilizes a scale-space representation to perform better feature extraction. Then, in chapter 4, I describe SIFT's feature descriptor, which has shown to perform better than Hu Moments and other methods which I had attempted earlier [14].

Chapter 3

SIFT Feature Extraction

As motivated in chapter 2, SIFT utilizes a scale-space representation of the image to extract features that are more invariant to scale transformations by finding features on an image at multiple scales. More formally, given an image $I(x, y, s)$, it finds all points (x_i, y_i, s_i) such that there is a salient feature on the image at location (x_i, y_i) and scale s_i.

There are, generally speaking, four stages in the SIFT algorithm:

1. **Scale-space peak selection:** The first stage of computation searches over all scales and image locations to identify interest points that are invariant to scale and orientation. It can be implemented efficiently by using a difference-of-gaussian function.

2. **Keypoint localization:** At each candidate location, a detailed model is fit to determine location, scale, and contrast. Keypoints are selected based on their stability.

3. **Orientation assignment:** One or more orientations are assigned to each keypoint based on local image properties.

4. **Keypoint Descriptor:** The local image gradients are measured at the scale in the region around each keypoint, and transformed into a representation that allows for local shape distortion and change in illumination. [13]

I discuss the first three stages in this section, saving the last for chapter 4, where I discuss both SIFT's descriptor and the newer PCA-SIFT descriptor.

3.1 Scale-space peak selection

As motivated in Chapter 2, we would like to provide a scale-space representation to the image so that we can select features across multiple scales. This will provide a much greater deal of scale-invariance when performing object recognition.

To review, we wish to describe an image I as

$$L(x, y, s) = G(x, y, \sigma) * I(x, y)$$

where $G(x, y, \sigma)$ is the 2-dimensional Gaussian distribution function:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

SIFT discretizes scale-space into a series of s values using a pyramidal gaussian scale-space model, as described in chapter 2. That is, it computes a series of sequentially blurred images. There are k blurred images in a particular octave and n octaves. There are thus nk images in total, representing n different scales and k different blurrings of the original image.

3.1.1 Difference of Gaussians

SIFT uses these images to produce a set of difference-of-gaussian (DOG) images. That is, at each octave, if there are k blurred images, then there will be $k-1$ difference-of-gaussian images. Each image is simply subtracted from the previous. (Refer to figure 3-1). The keypoints detected by SIFT are the extrema in scale-space within each octave of these $k-1$ difference-of-gaussians. Within a particular octave, a difference-of-gaussian function, $D(x, y, \sigma)$ is computed as follows:

$$\begin{aligned} D(x, y, \sigma) &= (G(x, y, r\sigma) - G(x, y, \sigma)) * I(x, y) \\ &= L(x, y, r\sigma) - L(x, y, \sigma) \end{aligned}$$

where $L(x, y, \sigma)$ is the original image I at a scale-space point σ within a particular

octave[1]. Also, r is the ratio between scales given in chapter 2, namely $r = 2^{1/k}$.

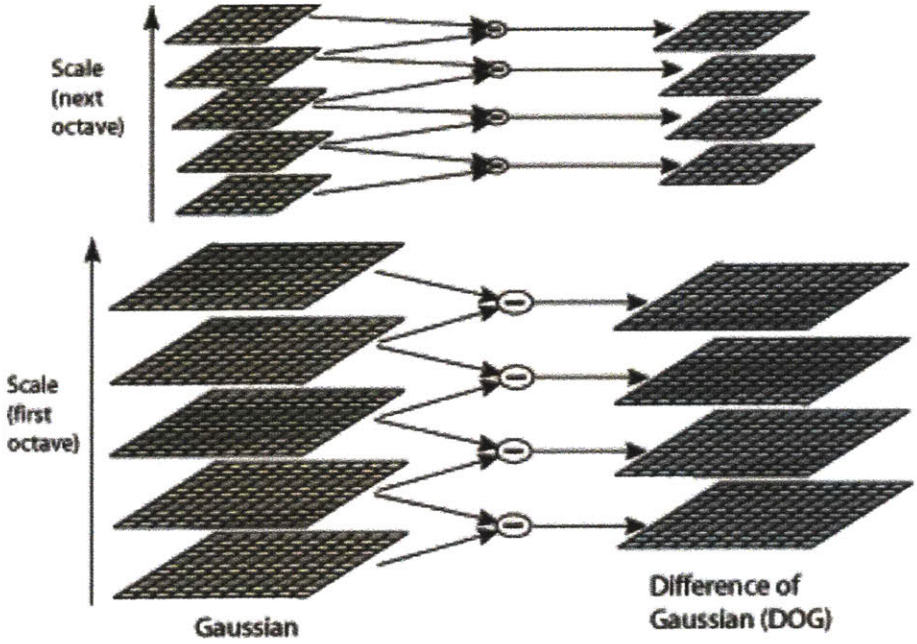


Figure 3-1: A view of how the difference-of-gaussian function is computed. At each scale, k images are computed, and from that $k-1$ difference-of-gaussians are computed [13].

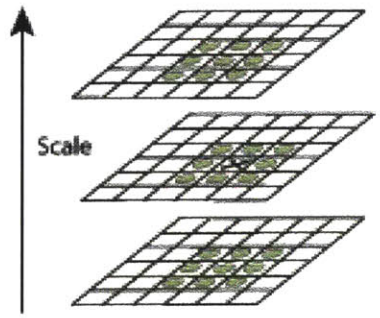


Figure 3-2: Detection of maxima and minima of DOG images is performed by comparing a pixel with its 26 neighbors.

The primary reason for choosing such a function is that it approximates the scale-normalized Laplacian of Gaussian ($\sigma^2\nabla^2 G$). It has been shown that the maxima and

[1]The notation here is a bit sloppy for the purposes of comprehension. When we say $L(x, y, \sigma)$, we mean to say $L(x, y, 2^i\sigma)$ where i is the current octave, because s, the scale value, is given by $s = 2^i\sigma$. Furthermore, $I(x, y)$ is the scaled version of the image for the particular octave i, and thus a better way to notate it would be $I(x, y; i)$, where $I(x, y; i)$ is simply I scaled by a factor of 2^{-1}).

minima of this particular function produce better, more stable image features than a host of other functions, such as the Hessian, gradient, and the Harris corner function [13].

We once again can use the diffusion equation to determine understand how a difference of gaussian function approximates the scale-normalized Laplacian of a Gaussian. We take the heat diffusion equation, (this time parameterized in terms of σ instead of $t = \sigma^2$):

$$\frac{\partial G}{\partial \sigma} = \sigma \nabla^2 G$$

Using the finite difference approximation to a derivative and the previous equation, we have:

$$\sigma \nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x, y, r\sigma) - G(x, y, \sigma)}{r\sigma - \sigma}$$

and therefore,

$$G(x, y, r\sigma) - G(x, y, \sigma) \approx (r - 1)\sigma^2 \nabla^2 G\,.$$

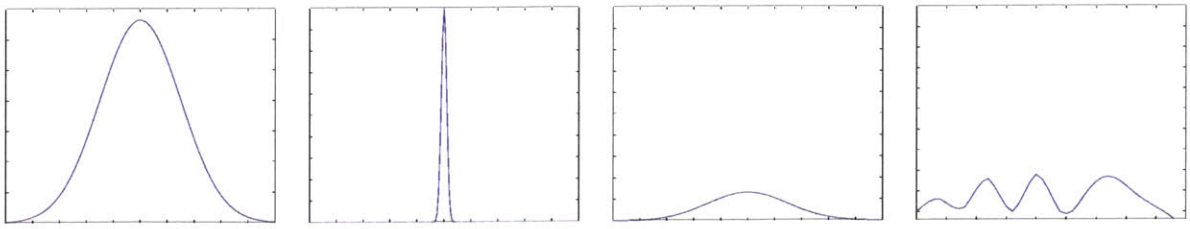
For a more detailed analysis, refer to [13].

3.1.2 Peak Detection

SIFT determines scale-space peaks functions over 3 difference-of-gaussian images. This is done by a simple comparison with each of its 26 neighbors (9 in the image at a higher scale, 9 in the image at a lower scale, and 8 surrounding pixels at the current scale). Refer to figure 3-2 for a view of how this is done. (For a more detailed explanation, refer to [13]).

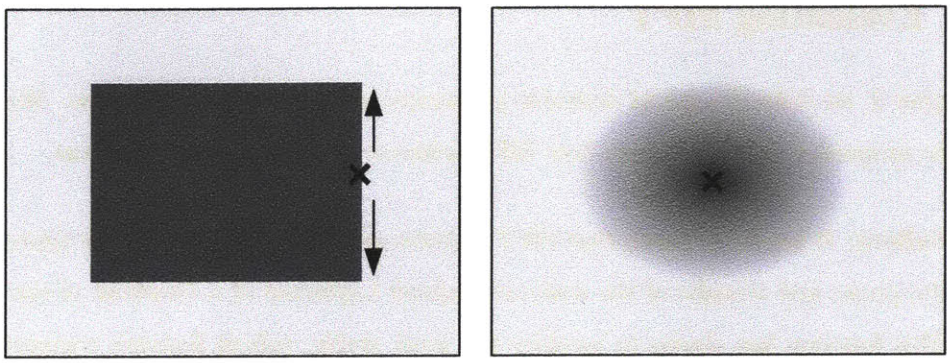
3.2 Keypoint Localization

At this point, each scale-space peak is checked to determine if it is a reasonable candidate for a keypoint. This check is important because some peaks might not have good contrast, and other peaks might not be consistent over the 26 sampled pixels. A reasonable peak is determined by modeling it as a 3-dimensional quadratic function, and is described in detail in [13]. Refer to figure 3-3 for examples of reasonable and unreasonable peaks.



(a) (b) (c) (d)

Figure 3-3: Selecting keypoints by determining whether the scale-space peak is reasonable. (a) shows an example of a reasonable peak in scale space, whereas (b), (c), and (d) show examples of unreasonable peaks.



(a) (b)

Figure 3-4: Example images that show why keypoints along edges are undesirable. (a) shows an undesirable keypoint located along an edge. Any noise could cause the keypoint to traverse the edge along the drawn arrows. (b), on the other hand, shows a much more reasonable keypoint, as it is stable in both directions.

3.2.1 Eliminating Edge Responses

It is also essential that a keypoint does not lie on an edge in the image. At an edge, the peak is well-defined in one direction, but not in the perpendicular direction. Thus, any noise in the image will potentially cause the keypoint to traverse the edge, resulting in poor matching (see figure 3-4). It is possible to determine which peaks are edges by computing the Hessian matrix at the location and scale of the keypoint. This is discussed in detail in [13].

3.3 Orientation Assignment

In addition to a position and a scale, each keypoint also has a orientation associated with it. This orientation is determined prior to creating the descriptor, and is done by creating a 36-bin orientation histogram covering a gaussian weighted circular region around each keypoint. The most dominant orientation is determined by finding the bin with the most points. Furthermore, if there exists bins within 80% of the largest bin, another keypoint is created at the same location with that particular orientation. Thus, it is possible for there to be multiple keypoints at the same position in the image, but with different orientations.

3.4 Evaluating SIFT

In chapter 2, we defined a list of desirable properties of good feature extractors. We return to these properties now to observe how SIFT achieves or does not achieve them.

1. **Salient:** It has been shown that SIFT features are indeed salient, as they approximate the peaks and troughs of the scale-normalized Laplacian of a Gradient of the image. This function has shown to produce the most stable, salient features compared to a host of other feature extraction methods.

2. **Scale-invariant:** SIFT features are not scale-invariant per se, but the scale-space representation of the image ensures that matching across various scales can be done more accurately.

3. **Rotation-invariant:** SIFT features are indeed rotation invariant. It is possible that some features will be lost due to the choice of sampling frequency, but at least the majority of features will remain in a consistent position as the image is rotated.

4. **Illumination-invariant:** The SIFT feature extractor runs on a set of gray-scale images. This in general reduces the positional movement of features when illumination is varied.

5. **Affine-invariant:** SIFT features are not affine invariant, yet the SIFT descriptor compensates for this by allowing for small changes in the position of the keypoint without a significant change in the descriptor. (refer to chapter 4).

3.5 Summary

Thus, we now have a list of keypoints selected at peaks in pyramidal scale-space using a difference-of-gaussian image representation. Each keypoint is defined by its position in the image, the scale value at which it was detected, and its canonical orientation. Some keypoints were rejected due to poor stability, and others were rejected if they lied along an edge. Furthermore, some keypoints have the same position in the image, yet different orientation, due to the fact that there might be more than one dominant orientation within a particular feature window. Once we have these keypoints, the next step is to perform a sampling around each to determine its local descriptor. This is the topic of the next chapter.

Chapter 4

SIFT Feature Descriptors

Once we have a list of features, we must appropriately sample the image around it. This sampling of pixels, when represented in some fashion, is known as a *feature descriptor*, because it *describes* the local image properties of that feature.

Although we have discussed a number of plausible feature descriptors in chapter 2, experiments (see [14]) have shown that the SIFT descriptor performs better than those presented in chapter 2, as well as a host of other feature descriptors, such as template correlation and steerable filters [4].

Thus, I chose to use the SIFT descriptor. However, I have also implemented a descriptor similar to SIFT, known as the PCA-SIFT descriptor (see [9]), since it claims to perform even better than SIFT's descriptor.

This chapter is devoted to discussion of both the SIFT and PCA-SIFT descriptor representations. The first section will discuss the SIFT descriptor, the second will discuss the PCA-SIFT descriptor, and the third will offer a comparison between the two.

Let us recall where we were once we finished chapter 3's description of SIFT: we have a list of keypoints, each found at a particular scale, orientation, and coordinate location. We now wish to obtain a descriptor representation for each keypoint.

We might consider simply sampling the region at the image corresponding to the feature's scale, and use it as a template for later matching by means of correlation. However, this will cause serious issues. Consider the situation when the template window is rotated by a few degrees, at which point the correlation metric will fail.

Thus, we consider rotating the entire descriptor using the orientation of the feature given

in chapter 3. This entails literally rotating the patch before sampling it into a descriptor vector.

We still have more issues to deal with: What if the location of the keypoint is slightly off? That is, consider a single pixel shift in the feature descriptor. It would then not match the original descriptor.

What is the solution to this problem? David Lowe proposed that instead of sampling actual pixel values, we sample the x and y gradients around the feature location[13] [11]. This method was inspired by a use of the gradient in biological image representations [2]. When gradient information is used, rather large positional shifts are allowed without a significant drop in matching accuracy.

Both PCA-SIFT and SIFT utilize this gradient sampling as a means of forming their feature descriptors. More specifically, both PCA-SIFT and SIFT share the following in common:

1. They sample a square region around the feature in the original gray-scale image. A square sampling region is used rather than the more conventional circular region for ease of implementation and performance gains.

2. They rotate the original sampling using the direction of the most significant gradient direction within the pixels sampled.

3. If the square patch size is of width w, they sample $(w-2) \times (w-2)$ x gradients, and the same number of y gradients. This gives a set of $2 \times (w-2) \times (w-2)$ values. In my implementation, the patch size was 41, and thus I had $2 \times 39 \times 39 = 3042$ x and y gradients.

At this point, SIFT and PCA-SIFT differ in their respective implementations, and thus I describe each in turn.

4.1 The SIFT Descriptor

SIFT takes the x and y gradients and computes the resulting set of magnitudes and orientations. Thus, we turn a 3042-dimensional vector of x and y gradients into a 3042-dimensional vector of magnitude and orientations. More specifically, each set of x and y gradients ($X_{x,y}$

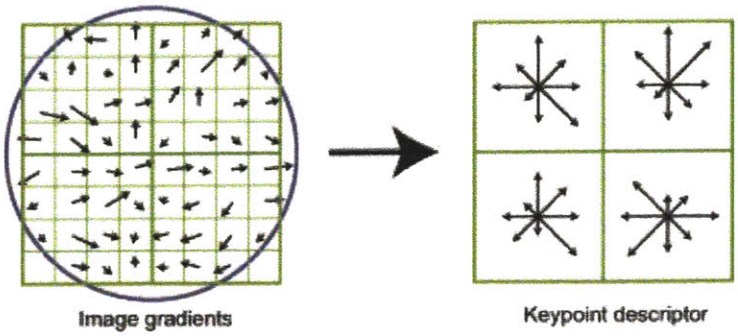


Figure 4-1: The left figure shows an image patch surrounding a given feature. At each sample point, the orientation and magnitude is determined, then weighted by a Gaussian window, as indicated by the overlayed circle. Each sample is accumulated into "orientation histograms". This is simply a histogram taken over all orientations within a particular region. The length of each arrow corresponds to the sum of the magnitudes of each sample in a particular orientation bin. This figure shows the sampling occurring over 2×2 regions, but in the implementation of SIFT, a 4×4 sampling is used. The number of orientation bins is 8. Thus, the total size of the descriptor is $4 \times 4 \times 8 = 128$ [13].

and $Y_{x,y}$) is used to compute the magnitude $m_{x,y}$, and orientation, $\theta_{x,y}$ by means of the following equation:

$$m_{x,y} = \sqrt{X_{x,y}^2 + Y_{x,y}^2}$$
$$\theta_{x,y} = \tan^{-1}(\frac{Y_{x,y}}{X_{x,y}})$$

Thus, we are left with a 3042-dimensional vector containing magnitudes and orientations of pixels surrounding each feature. SIFT reduces this into a significantly smaller vector by dividing the sampled region into a $k \times k$ region (see figure 4-1), and computing a histogram of gradient orientations within each. By histogramming the region in this way, the resulting descriptor is made more robust to image deformations.

Prior to placing each gradient in its appropriate histogram, a Gaussian weighting function with σ equal to one-half the width of the descriptor window is applied. The purpose of this is to avoid sudden changes in the descriptor with small changes in the position of the window. This is because the gaussian window function gives less weight to gradients near the edge of the window, and these are exactly the gradients that will be affected the most by changes in camera orientation.

To reduce the possibility of boundary effects, each gradient is placed into the 8 surrounding bins as well, each one weighted by its distance to that bin. That is, bilinear interpolation is used to determine the weight of each entry into the histogram.

In chapter 2, we mentioned that specular highlights can cause difficulties with object matching. Specifically, if there is a specular flare near a feature in a query image, and no specular flare in the template image, then that particular feature would not be matched. To attempt to compensate for this, Lowe notices that a specular highlights result in abnormally large gradient values. To compensate, once we have created our full feature vector, gradient magnitudes are cut off at an empirically determined value. Effectively, this "erases" the specular highlight from the feature descriptor. Of course, this cutoff will impact matching negatively in some cases, for example, if an object contains a feature that appears very much like a specular highlight. However, the implementation banks on the fact that this does not happen very often.

4.2 The PCA-SIFT Descriptor

Before we discuss how the PCA-SIFT descriptor works, let us take a moment to understand PCA, since PCA-SIFT is effectively PCA performed on each 3042-dimensional vector of x and y gradients.

4.2.1 Principal Component Analysis (PCA)

Principal Component Analysis, or PCA, is a method of reducing the dimensionality of a data set by projecting it onto a smaller coordinate space that maximizes the variance along each axis. Using the derivation outlined in [3], the coordinate axes of this coordinate space are simply the eigenvectors of the covariance matrix of the data. That is, if we construct a NxM matrix $\mathbf{A}$ where each N-dimensional mean-free data point is a row in the matrix, and there are M data points, then we need to find the eigenvectors of the covariance matrix $\mathbf{C}$:

$$\mathbf{C} = \frac{1}{M}\mathbf{A}\mathbf{A}^T$$

Thus, we decompose $\mathbf{C}$ using eigenvalue decomposition:

$$\mathbf{C} = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^{\mathbf{T}}$$

For PCA, recall that we wish to find the vector space such that the variance is maximized in each dimension. Conveniently, $\mathbf{\Sigma}_{i,i}$ corresponds to the variance when the data is projected onto the eigenvector located at the ith column of $\mathbf{V}$. Thus, if we desire a k-dimensional coordinate space, we simply need to find those eigenvectors whose corresponding eigenvalues are the k-largest. Then, we obtain the k-dimensional description of the data set by taking the original mean-free data matrix $\mathbf{A}$ and projecting it onto the eigenspace given by the those k eigenvectors.

4.2.2 Singular Value Decomposition

It is often the case that computing the covariance matrix $\mathbf{C}$ is too costly. Luckily, it turns out that we can simply compute the *Singular Value Decomposition* (SVD) of the original mean-free data matrix $\mathbf{A}$. From [19], the Singular Value Decomposition of any matrix $\mathbf{A}$ returns an orthonormal matrix $\mathbf{U}$, a diagonal matrix $\mathbf{S}$, and an orthonormal matrix $\mathbf{V}$ in the following form:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

The proof for why the SVD of the matrix $\mathbf{A}$ will produce the eigenvectors of its covariance matrix goes as follows:

$$\begin{aligned}
\mathbf{C} &= \tfrac{1}{M}\mathbf{A}\mathbf{A}^T \\
&= \tfrac{1}{M}\mathbf{U}\mathbf{S}\mathbf{V}^T(\mathbf{U}\mathbf{S}\mathbf{V}^T)^T && \text{(using the SVD decomposition } \mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T\text{)} \\
&= \tfrac{1}{M}\mathbf{U}\mathbf{S}\mathbf{V}^T\mathbf{V}\mathbf{S}^T\mathbf{U}^T \\
&= \tfrac{1}{M}\mathbf{U}\mathbf{S}\mathbf{S}^T\mathbf{U}^T && \text{(because } \mathbf{V} \text{ is orthonormal)} \\
&= \tfrac{1}{M}\mathbf{U}\mathbf{S}^2\mathbf{U}^T && \text{(because } \mathbf{S} \text{ is a diagonal matrix)} \\
&= \mathbf{U}\mathbf{K}\mathbf{U}^T
\end{aligned}$$

Thus, taking the SVD of $\mathbf{A}$ will produce the eigenvalue decomposition of $\mathbf{C}$ with a eigenvalue matrix that has been scaled by $\frac{1}{M}$ and squared. The columns of $\mathbf{U}$ are the

eigenvectors, and the diagonal elements of $\frac{1}{M}\mathbf{S}^2$ are the eigenvalues.

4.2.3 PCA Applied to SIFT

The PCA-SIFT descriptor is simply the application of PCA to the original 3042-dimensional vector of x and y gradients [9]. After the 3042-dimensional vectors for each keypoint for a set of training images are obtained, its eigenspace is computed. Then all keypoints extracted from further query images are projected onto this eigenspace. For a more complete discussion, refer to [9].

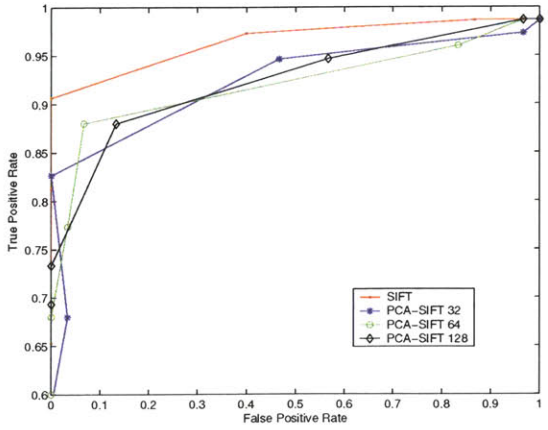
As the motivation behind PCA-SIFT is to create a descriptor that is both smaller and more accurate than the original SIFT descriptor, the value for the number of dimensions k should be less than or equal to 128, the original number of dimensions used in the SIFT descriptor.

4.3 Comparison

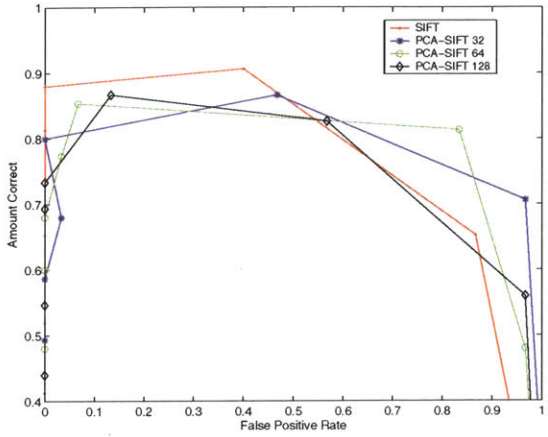
PCA-SIFT and SIFT were compared using an implementation of SIFT and my own implementation of PCA-SIFT. Tests were run on a data set of 15 models, each consisting of one training image with the background subtracted, and 5 test images. In addition, 30 images were used for determining PCA-SIFT's resilience to false positives (These 30 images were selected randomly from the false positive data set shown in appendix B). The hypothesis-checking model was used as described in chapter 6 of this thesis and [13], and naive (brute-force) nearest-neighbor matching was used. See appendix A for more details about the data set used.

Figure 4-2 shows a comparison of SIFT and PCA-SIFT. Unlike in [9], PCA-SIFT did not perform as well as SIFT. Figure 4-3 shows SIFT vs. PCA-SIFT when a Mahalanobis-like distance metric was used to attempt to improve the performance of PCA-SIFT. This improved the matching accuracy slightly, but still could not rival the SIFT descriptor for this particular data set.

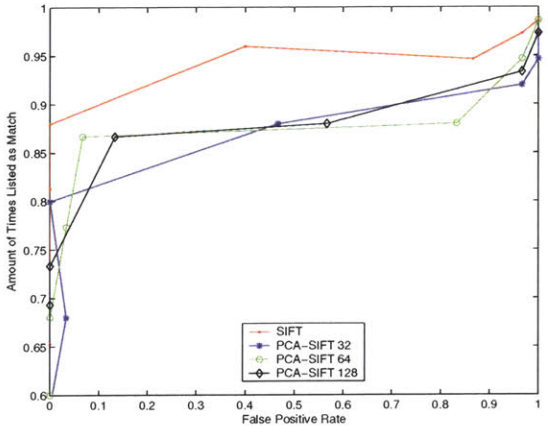
Thus the only possible conclusion, assuming that the results of the PCA-SIFT paper were correct, is that SIFT only performed better on this particular data set. For example, it is possible that PCA-SIFT would perform better on more uniform objects, because it would result in a sparser data set for the gradient descriptors.



(a)

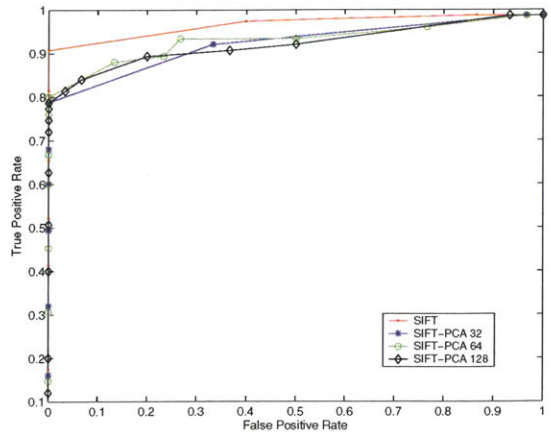


(b)

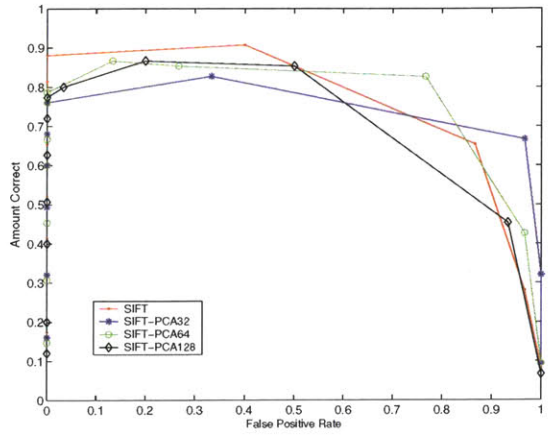


(c)

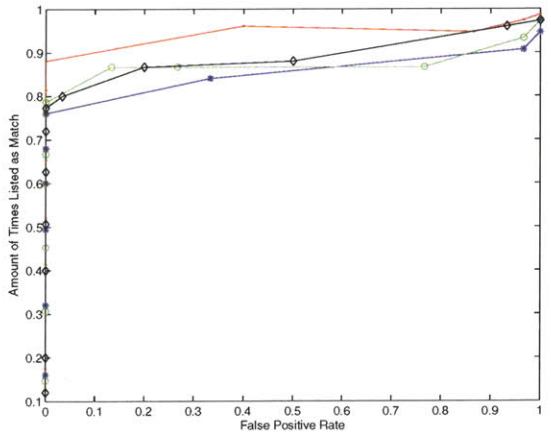
Figure 4-2: SIFT vs. PCA-SIFT on a relatively difficult data set of 15 models. (a) shows a comparison of the receiver-operating curve, (b) shows the amount correct and (c) shows the amount of times the correct match was returned.



(a)



(b)



(c)

Figure 4-3: SIFT vs. PCA-SIFT on a relatively difficult data set of 15 models using a Mahalanobis-like distance function. (a) shows a comparison of the receiver-operating curve, (b) shows the amount correct and (c) shows the amount of times the correct match was returned.

Chapter 5

Descriptor Matching

In any feature-based object recognition system, the performance bottleneck will likely be the matching phase. We wish to find the closest match for some 750 feature points, each consisting of a d-dimensional descriptor, to a set of maybe 50,000 different features. In such a situation, a brute-force approach is simply not feasible. So, we devote this chapter to the discussion of *approximate* nearest neighbor solutions. These algorithms sacrifice some of the guaranteed precision of the brute-force approach for greater improvements in speed. We therefore wish to find the algorithm which will return the most accurate matches in the fastest time. I will concentrate on two nearest neighbor algorithms in this chapter: *Best Bin First* (BBF), and *Locality Sensitive Hashing* (LSH).

This chapter will first give a background of nearest neighbor searching, then move on to introduce the naive brute-force approach to nearest-neighbor matching. I will then discuss BBF and LSH in turn, and in the final section, provide an extensive empirical comparison between the two algorithms.

5.1 Background

Let us quickly take a brief overview of the two nearest-neighbor methods that will be discussed in this chapter:

1. *Best Bin First*, otherwise known as BBF, is the popular approximate nearest neighbor matching algorithm used by David Lowe in his SIFT paper [13] [1]. It creates a tree-based representation of the data so that only $O(\log n)$ scalar comparisons are needed to get to a region where the point "likely" exists. At this point, depending on the

degree of precision the user desires, further brute force nearest-neighbor searching is done in points surrounding that region.

2. *Locality-Sensitive Hashing*, otherwise known as LSH, is a hashing-based approach to approximate nearest neighbor matching. Its approach is exactly described in its title: use "locality-sensitive" hashing functions, (i.e., a set of functions H such that, for any $h \in H$, if $\mathbf{p}$ is close to $\mathbf{q}$, then $h(\mathbf{p}) = h(\mathbf{q})$). In this way, collisions will occur between points that are "neighbors" of each other, and thus queries simply take the form of searching for the hash bucket associated with the query point's hash key.

We should pause for a moment to define how our distances will be measured in this vector space. There are many reasonable ways to find the "distance" between two points in d-dimensional space. We will concentrate on perhaps the two most popular – the l_1 norm distance and the l_2 norm distance.

For two points $\mathbf{x}$ and $\mathbf{y}$, such that:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

the l_1 distance function is defined as:

$$d_{l_1}(\mathbf{x}, \mathbf{y}) = |\mathbf{x} - \mathbf{y}|_1 = \sum_{i=1}^{n} |x_i - y_i|$$

and the l_2 distance function is defined as:

$$d_{l_2}(\mathbf{x}, \mathbf{y}) = |\mathbf{x} - \mathbf{y}| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

[21]

The l_2 norm distance is the function most commonly thought of when describing a "distance" function, since it is the distance function for Euclidian space. In such a space, the l_1 norm distance would simply be an approximation. (By the triangle inequality, it would in fact always overestimate the l_2 norm distance).

Although LSH [6] uses the l_1 distance function and it is unclear whether BBF uses l_1 or l_2, we limit ourselves to using the l_2 distance function in each.

5.2 Naive Nearest Neighbor

The naive approach to nearest neighbor matching is to simply iterate through all points in the training set to determine the nearest neighbor. Although this is extremely slow in practice, a simple optimization can be used to add a moderate speed improvement.

This optimization is simply to cut off further processing of the distance function $d(p, q)$ if it is known that it cannot possibly produce the nearest neighbor. That is, for $d(p, q)$, we also pass in the distance to the best known nearest neighbor. Call this value $d_{BEST-KNOWN}$ (Thus $d(p, q)$ becomes $d(p, q, d_{BEST-KNOWN})$. As we are computing the distance between points p and q as defined earlier, we continuously check if the current distance, for the number of dimensions we have currently computed, has already surpassed the value $d_{BEST-KNOWN}$. If it has, then we exit; if not, we continue processing the distance.

This simple optimization has been used to cut processing time down by a moderate amount. Unfortunately, the optimization is quite data-dependant – even the order in which points are searched affects its performance. Nevertheless, I use this optimization in this chapter – whenever I discuss any "brute-force" processing later in this chapter, it is assumed that this optimization is being used.

5.3 Best Bin First

Best Bin First, otherwise known as BBF, is effectively an approximate form of the Kd-Tree search, (first described by Friedman, Bentley, and Finkel) [5]. Like Kd-Trees, it recursively splits the data perpendicular to the dimension of highest variance until each point is contained within a single "bin". This produces a balanced or very nearly balanced binary tree with depth $O(\log n)$, where n is the number of data points. For ease of discussion, I will describe the balanced-tree version, as it is simpler.

5.3.1 Tree Construction

The construction of a BBF tree is exactly the same as in Kd-trees. The data is recursively split by choosing a dimension to split the data along, and then choosing a pivot around

Name	*Description*
dim	The dimension along which the node splits.
split	The value at dimension dim which splits the data.
left	The subtree to the left of this node.
right	The subtree to the right of this node.
hasData	Whether this node references a particular point.
idx	The index of the point in d-dimensional space.

Table 5.1: The information contained in each Kd-tree node

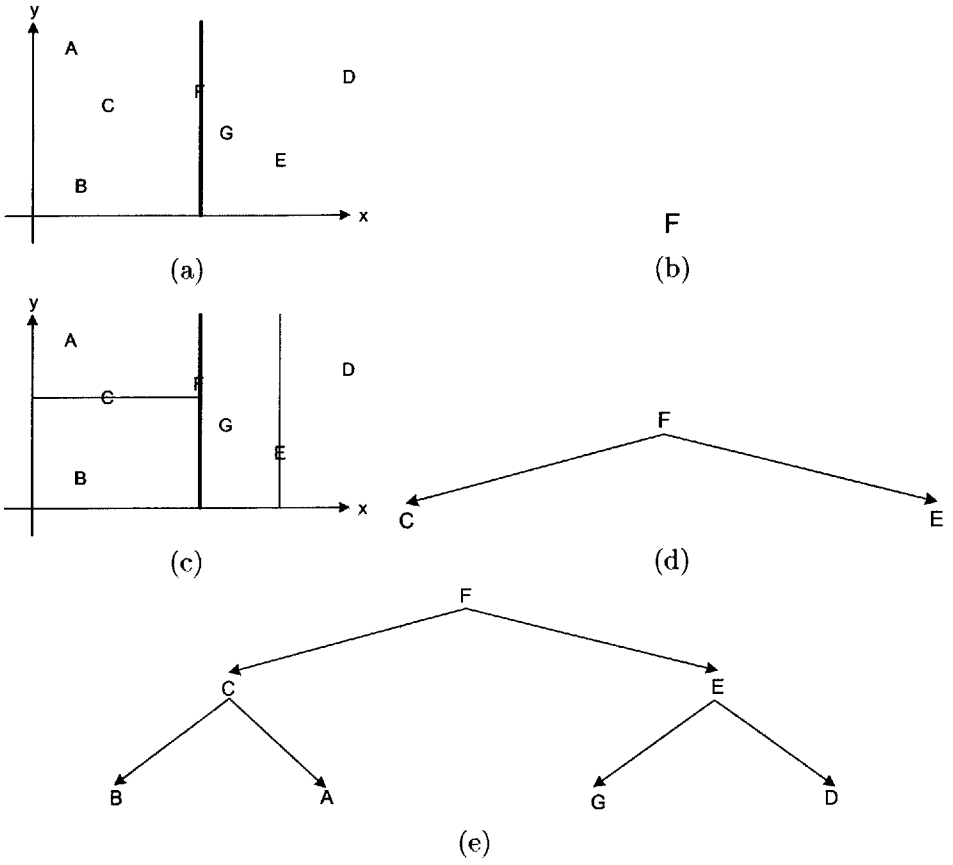


Figure 5-1: (a) The initial data in 2-dimensional space. We choose our first pivot as the median of the dimension with the highest variance, which in this case is centered around F along the dimension x. (b) Thus, the hasData and idx are set for the root node for the point F. (c) Once again, we choose our pivots to be the medians of the dimensions with highest variance. The dimension to the left of the root node is y, and to the right is x. In those dimensions, we choose the medians as C and E, respectively. (d) The updated Kd-Tree has data attached to the child nodes of F, which are thus C and E. (e) We are done with the construction of the Kd-Tree, and add the final leaf nodes.

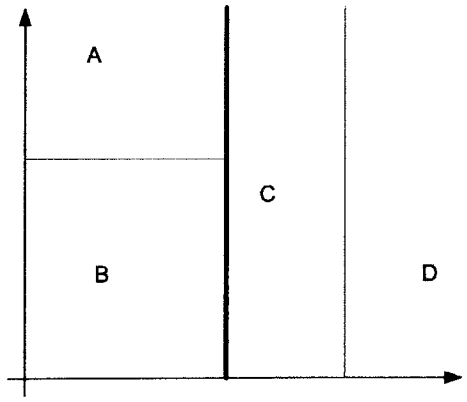


Figure 5-2: A Kd-Tree without any internal nodes that reference data points. If the number of nodes is an exact power of two, this will be the case. More precisely, during tree construction, if the amount of data remaining to be split at a given node is exactly divisible by 2, then the internal node at that split will not reference a particular data point. For example, in this case, at the root node, there are 4 data points to be split, and thus we split between the first two and the last two, and the root node does not reference any data point. Then, at the next recursive split, both child nodes have 2 data points remaining in them, and thus the internal nodes created at that level will not reference any data points.

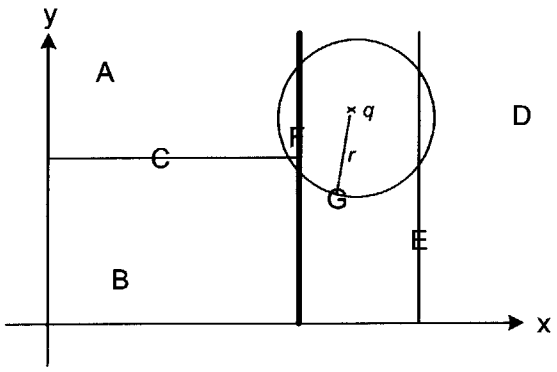


Figure 5-3: Depicts hypersphere intersection in a Kd-Tree. Note that since the hypersphere at q intersects the leaf bin A, it must also intersect its parent hyperrectangle, defined by the point C. Refer to figure 5-1e to understand more fully how this works.

which to split. There are a number of ways to choose a dimension to split, as well as a number of ways to choose a pivot. For the purpose of this discussion, however, we will describe the simplest approach: we choose a splitting dimension by finding the dimension along which the variance of the data is maximal, and we choose a pivot by finding the median along that dimension. Figure 5-1 shows the construction of a simple 2-dimensional Kd-Tree that utilizes this approach.

As is evident from figure 5-1, each node of the tree can contain a reference to a particular data point. All leaf nodes and some internal nodes reference a particular data point. Table 5.1 shows the information contained in each node of the Kd-Tree.

Of course it is not always the case that an internal node references a data point. If the number of points is exactly a power of 2, then, in fact, there will be no internal nodes that reference a data point. Figure 5-2 shows an example of this fact.

5.3.2 Tree Searching

To search a Kd-tree, a first approximation is made by determining the bin which contains the query point. This is a fast operation, taking $O(\log n)$ scalar comparisons. Although this is not the nearest neighbor, it is very often a reasonable approximation.

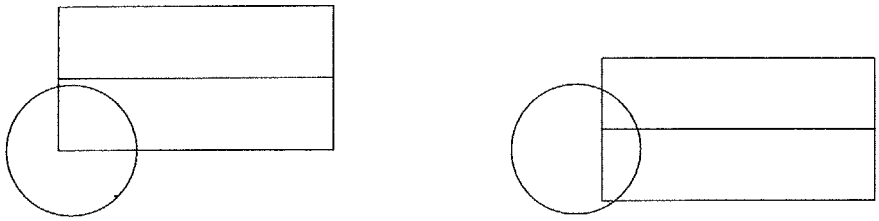
Let the approximate point found by this initial tree traversal be p_a. If the distance between our query point and p_a is r, then the true nearest neighbor distance must be less than or equal to r. Also, the bin containing the nearest neighbor must intersect a hypersphere of radius r centered at p_a. In addition, *all hyperrectangles defined by all parent nodes of that bin must also intersect that hypersphere.* This fact is illustrated in figure 5-3.

Thus, as we make our initial traversal of the tree, we simply maintain a queue of hyperrectangular regions that we need to search, defined by nodes of the Kd-tree. These nodes are exactly those that we did *not* select as we were making the initial traversal of the tree.

Then, we perform a breadth-first search down the tree starting from each node in the queue. As this is done, we prune off whole sections of the tree if the hyperrectangular regions cannot possibly contain the nearest-neighbor. This is done by computing the intersection between the query hypersphere and the hyperrectangle at each node.

An important observation about hypersphere-hyperrectangle intersections should be made that is useful for optimizing Kd-Tree searching: if a hypersphere intersects both of the child hyperrectangles of a given hyperrectangle, then and only then should we add

the internal node *idx* for that hyperrectangle (assuming that the *hasNode* entry is set for that particular node). This observation is made clear in figure 5-4.



(a) (b)

Figure 5-4: Note that any point lying on the splitting line in the hyperrectangle will never intersect the hypersphere unless it intersects both child hyperrectangles. Thus, there is no need to add the internal node of the parent hyperrectangle to the queue unless both child hyperrectangles are intersected. This observation holds true in higher-dimensional space as well. In (a), only one child hyperrectangle is intersected, and thus there is no possibility that a point lying on the splitting line could be closer than the query point. The only possibility would be if it is exactly the same distance to the query point, a case that we need not consider as we are only returning one nearest neighbor. In (b), both child hyperrectangles are intersected, and it is thus obvious that a point on the splitting line might be closer to the query point than the p_a, and thus we should add that point to the queue.

5.3.3 The BBF optimization

As it turns out, Best-Bin First adds a very simple optimization. This optimization is needed for higher-dimensional spaces, such as the 128 dimensional space that SIFT uses. For dimensions lower than 6 or 7, Kd-Trees adds a reasonable improvement, but for higher dimensional spaces, Kd-Trees quickly becomes slower than naive brute-force nearest neighbor. This fact is evident in figure 5-5.

The motivation for BBF is that some elements of the queue are more likely to hold the nearest neighbor than others. That is, the queue should be ordered in terms of the distance from q to the region corresponding to each element in the queue.

To create an approximate search, BBF simply terminates after it has searched a predefined number of points. The user defines this approximation parameter when the BBF search is called. If the user requests more points to be returned, then more elements of the queue are expanded, and the results will likely be more accurate.

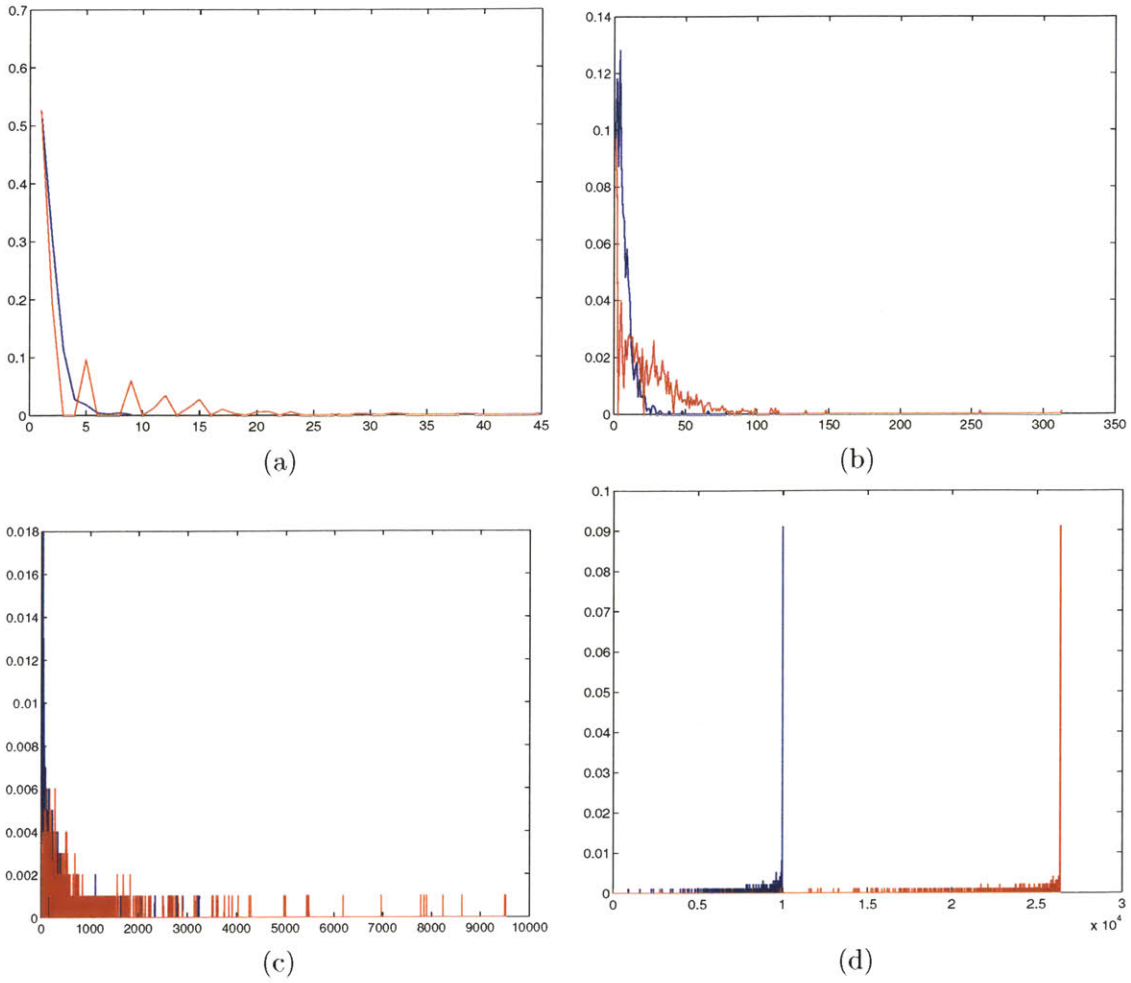


Figure 5-5: The PDFs of points searched using Kd-trees in terms of the number of distance computations and the number of points returned. The red lines are the PDFs for the number of distance function calls, whereas the blue lines are the PDFs for the number of points in the queue. (a) shows the PDF for 1-dimensional data, (b) shows the PDF for 2-dimensional data, (c) shows the PDF for 4-dimensional data, and (d) shows the PDF for 20-dimensional data. The training data was a mixture of 10 gaussians with unity covariance and randomly generated means from 0 to 10 in each dimension. The test was performed on 10 different dimensions, ranging by powers of 2 from 1 to 1024, and the query data was selected from a gaussian with mean 5 in every dimension and diagonal covariance matrix with value 5 on each diagonal. The x-axis is the number of distance function calls and the number of points placed on the queue. The y-axis is the probability that a particular point would utilize that particular number of distance function calls and would return that many points. Note that the PDF for the number of points returned in 20-dimensional data is almost exactly centered around the total number of training points. Also, note that the number of distance function calls required far exceeds the number required for a naive brute-force search.

5.4 Locality-Sensitive Hashing

Locality-Sensitive Hashing, or LSH, is another method of approximate nearest-neighbor search. LSH centers around a relatively simple notion: insert each point into a hash table such that if a query point is close to a point in that hash table, then they will be hashed to the same key. The hash function used to do this is also relatively simple. Every hash key is a bit vector, where each bit is 1 if the point at a given random dimension is greater or equal to a random threshold, and 0 if it is not.

More precisely, the hash key for a d-dimensional point $\mathbf{p} = [x_1\ x_2\ \ldots\ x_d]^T$, is given by the bit vector:

$$g(\mathbf{p}) = (f_{t_1}(x_{i_1}),\ f_{t_2}(x_{i_2}),\ \ldots,\ f_{t_{k_1}}(x_{i_{k_1}}))$$

where

$$f_t(x_i) = \begin{cases} 1 & \text{when } x_i \geq t \\ 0 & \text{otherwise} \end{cases}, \quad t = rand(0, C),$$

C is the maximum value over all dimensions and all points in the data set, and k_1 is the number of bits in the key. The values x_i are coordinates chosen at random with replacement from the point $\mathbf{p}$.

This formulation assumes that the data has the following important properties: first, it is positive, and secondly, it consists of integer values. This of course does not limit our implementation. For any real data, we can ensure the first property by appropriately shifting the data. The second property can be achieved by choosing a number of significant digits s and multiplying all data by 10^s, then converting all points to have integer values.

Since this formulation requires a hash table of size 2^{k_1}, easily too costly for reasonable values of k_1, a secondary hashing function is employed. The Level 2 hash function works by selecting bits at random *without* replacement from the level 1 hash key, $g(\mathbf{p})$, and using those bits to access the hash table. More specifically, we define $h(\cdot)$, the secondary hash function, as:

$$h(b) = (b_{i_1},\ b_{i_2},\ \ldots,\ b_{i_{k_2}}).$$

where b is the bit vector produced by the Level 1 hash function's application to $\mathbf{p}$, and b_i are various random bits of b with the constraint that $i_1 \neq i_2 \neq \ldots \neq i_{k_2}$. Refer to figure 5-6.

Thus, every point is hashed to the key $h(g(\mathbf{p}))$. There are exactly 2^{k_2} hash bins, and since multiple points may be hashed to the same level 2 key, each entry in the hash table points to a linked list of maximum size B. If we are attempting to insert a point into an entry already containing B elements, we simply do not insert it, because with high probability it will be inserted into some other hash table, assuming that l, the number of hash tables, is of a reasonable size.

As implied in the previous paragraph, there are a number of important parameters involved in LSH. It would be helpful to enumerate them and discuss each one in turn. (It is helpful to refer to figure 5-6 while reading this list.)

- k_1: the number of bits in the Level 1 hash key. By increasing this value while holding all other values fixed, the algorithm requires that results match a greater number of the hash functions g mentioned earlier. This in effect will improve speed but reduce accuracy, because we will return less points for brute-force matching.

- k_2: the number of bits used in the Level 2 hash key. The size of each hash table is thus $M = 2^{k_2}$. Recall that we needed a second-level function because the storage requirements for a table of size 2^{k_1} would be too great.

- l: the number of hash tables. Increasing this value will obviously decrease performance and increase accuracy, holding all other values fixed.

- B: the size of the Level 2 hash buckets. Increasing the size of B will result in degraded performance but improved accuracy.

- α: this parameter is known as the *memory utilization parameter*. It is related to the size of each hash table M, the bucket size B, and the total number of points n, by the following equation: $M = \alpha\frac{n}{B}$. α represents the ratio of the memory allocated for the index to the size of the data set. It is sometimes clearer in another form: $B = \alpha\frac{n}{M}$. In this way, by increasing α, we increase the length of our hash buckets, and thus our flexibility in storing more information within each index. In my specific implementation, I set a threshold on the minimum bucket size. Thus, B was given by

the equation

$$B = \max(\gamma, \alpha\frac{n}{M}) .$$

In my implementation, I used a value of γ equal to 3.

Figure 5-6 shows how a data point **p** is placed into the hash tables.

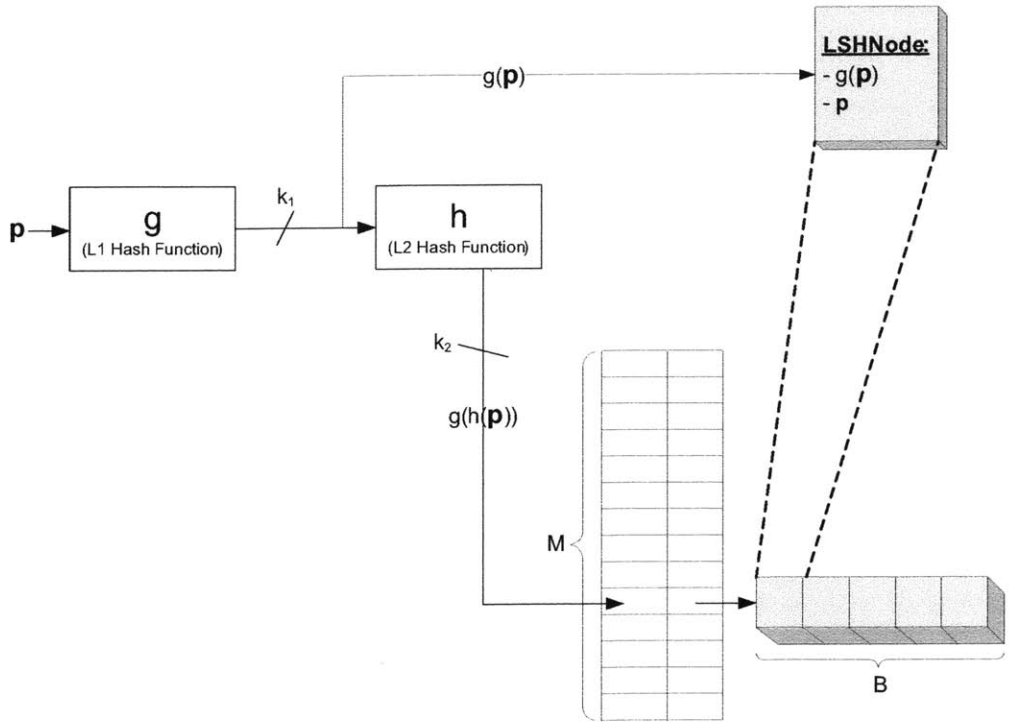


Figure 5-6: Inputting data into one of the hash tables in an LSH structure. The data point **p** is used to generate the Level 1 key, $g(\mathbf{p})$, which in turn is used to generate a Level 2 key, $h(g(\mathbf{p}))$, used to index into the hash table. The size of the hash table is $M = 2^{k_2}$, where k_2 is the number of bits in the L2 key. Each entry in the hash table points to a set of B buckets, each of which can contain an LSHNode structure. The LSHNode structure contains the L1 description of the **p**, and also a reference to the point itself. There are exactly l hash tables of this form, and thus l Level 1 and Level 2 hash functions.

5.5 Comparison

The purpose of this section is to compare BBF and LSH. I will first offer a non-empirical comparison, and then present the results of an empirical comparison of the two algorithms.

5.5.1 Non-Empirical Comparison

There are some important differences to be noted between LSH and BBF. They are listed as follows:

1. BBF returns an ordered list of points, ordered by the probability that the point is a nearest neighbor, whereas LSH simply returns an unordered list of points. The advantage to returning an ordered list is that we end up finding the nearest neighbor sooner, and thus the optimization given in section 5.2 will likely be more efficient. This property of BBF makes comparing the two algorithms difficult. However, in my comparisons, I found that it did not make a significant difference. That is, the closest nearest neighbor was always approximately in the middle of the list (as it was of course with LSH). Yet, this is a property of the data set used, and thus future comparisons between these two algorithms should be performed on various different types of data sets.

2. BBF's approximation parameter is given on entry to the search, whereas in LSH, it is given prior to inserting the data into the LSH structure. Furthermore, there is only one parameter to BBF, whereas there can be between 4 and 5 parameters to LSH.

3. BBF is an intricate and detailed algorithm, and requires a lot of thought when implementing it. LSH, on the other hand, is extremely simple, and thus optimizing the code, or even writing it entirely in assembly, is relatively feasible.

4. The optimal splitting strategy for BBF is somewhat data dependent, whereas there is no aspect of LSH which is data dependent. That is, there is no parameter to set in LSH which will optimize performance when using a particular data set. In this sense, BBF is somewhat more powerful, but only if the distribution of the data is known a priori.

5. LSH requires the data be integer-valued, whereas BBF does not have such a requirement. However, I have found that the process of transforming a vector into its integer representation for LSH does not add a significant amount of computation time to the algorithm.

As we can see, there are a number of advantages to both LSH and BBF. One might choose to use one or the other depending on the application.

5.5.2 Empirical Comparison

BBF and LSH were implemented in C++ using standard libraries on a 2.0 gigahertz machine running Windows 2000. They were both compiled using Visual Studio C++ 6.0. The training data was a mixture of 10 gaussians with unity covariance and randomly generated means from 0 to 10 in each dimension. The test was performed on 5 different dimensions, ranging by powers of 2 from 32 to 1024, and the query data was selected from a gaussian with mean 5 in every dimension and diagonal covariance matrix with value 5 on each diagonal.

Note that BBF calls a distance function when determing hypersphere intersections and that both LSH and BBF call a distance function when finally determining their approximate nearest neighbors. Using Intel's VTune Performance Analyzer, I noted that the dominating function in both implementations was the call to the distance function. Thus, the comparison will be done in how well the algorithms perform for a given number of distance function calls.

We define the error, E, of the approximate nearest neighbor search result, to be

$$E = \frac{1}{|Q|} \sum_{q \in Q} (\frac{d_a}{d} - 1)$$

where d_a is the approximate distance given by a particular algorithm, and d is the correct distance.

Figure 5-7 shows a comparison of the error as a function of the number of distance calls LSH and BBF make. Figure 5-8 shows a comparison in terms of milliseconds of my actual implementation. Note that it is indeed the case that we can judge the performance of these algorithms in terms of the number of times it calls the distance function.

A similar comparison test was performed on LSH and BBF using $1,000$ SIFT features extracted from a test set of 20 images, where each algorithm was trained from a set of 10 images totalling approximately $10,000$ keypoints. The results are shown in 5-9.

It should be noted that the number of distance calculations decreased when the number of dimensions increased. This seems counter-intuitive, but in fact is correct. The same amount of data is being spread over a greater area, and thus it will be clearer what the nearest-nearest neighbor actually is. Another way to think about this is that both LSH and BBF effectively split up the data space into axis-aligned regions. If there are more axes to choose from, but the same number of points, using such regions as a means of determining

Number of dimensions	Optimized Query Time (ms)
32	9.834
64	23.214
128	49.762
512	222.55
1024	480.982

Table 5.2: Query time of optimized nearest-neighbor for varying dimensions.

the nearest neighbor becomes more useful.

However, each distance call obviously takes a greater amount of time, and thus naive brute-force (using the optimization given in section 5.2), will be slower in milliseconds for larger dimensions (refer to table 5.2).

5.5.3 Discussion

As is evident from figure 5-7, LSH seems to perform better than BBF, assuming that a reasonably small amount of error is required. Surprisingly, if one does not require a great deal of precision in their calculations, and the dimensionality of the dataspace is large (approximately 512 dimensions or higher), these results imply that BBF would be the better choice. However, it is likely that this is simply an artifact of the data set used, since, as mentioned earlier, it is easier to separate the data using scalar comparisons when the number of dimensions increases and the number of data points remains the same. For smaller error, however, the results imply that LSH is the clear choice. One would likely need to verify this for different types of data sets, but it is my opinion that LSH would still perform better. Observing figure 5-8, it is likely worth the extra milliseconds to perform a naive nearest neighbor search when the number of dimensions is small (64 dimensions or lower). This should come as no surprise – as the number of dimensions grow, the distance calculations become more expensive (refer to table 5.2), and thus using an approximate method to avoid some of those distance calculations becomes more useful.

Another surprising aspect of these results is that BBF scaled reasonably well with higher dimensions in this particular data set. It was my intuition that as the number of dimensions increased, LSH would begin to far outperform BBF. This was not the case. Once again, this could be the result of the data set used because as the number of dimensions increases, the data becomes more separable. From 5-9b, it is clear that for my particular application

with SIFT features, LSH is the appropriate choice. It is, however, much less clear that this choice should be used as a general rule (observing figure 5-9a).

Yet, for any reasonable SIFT object recognition, it is necessary that the error be approximately less than 0.5. This was found experimentally with the tests of the system described in chapter seven. Thus, it is my opinion that, for object recognition, LSH is a faster, more useful algorithm than BBF.

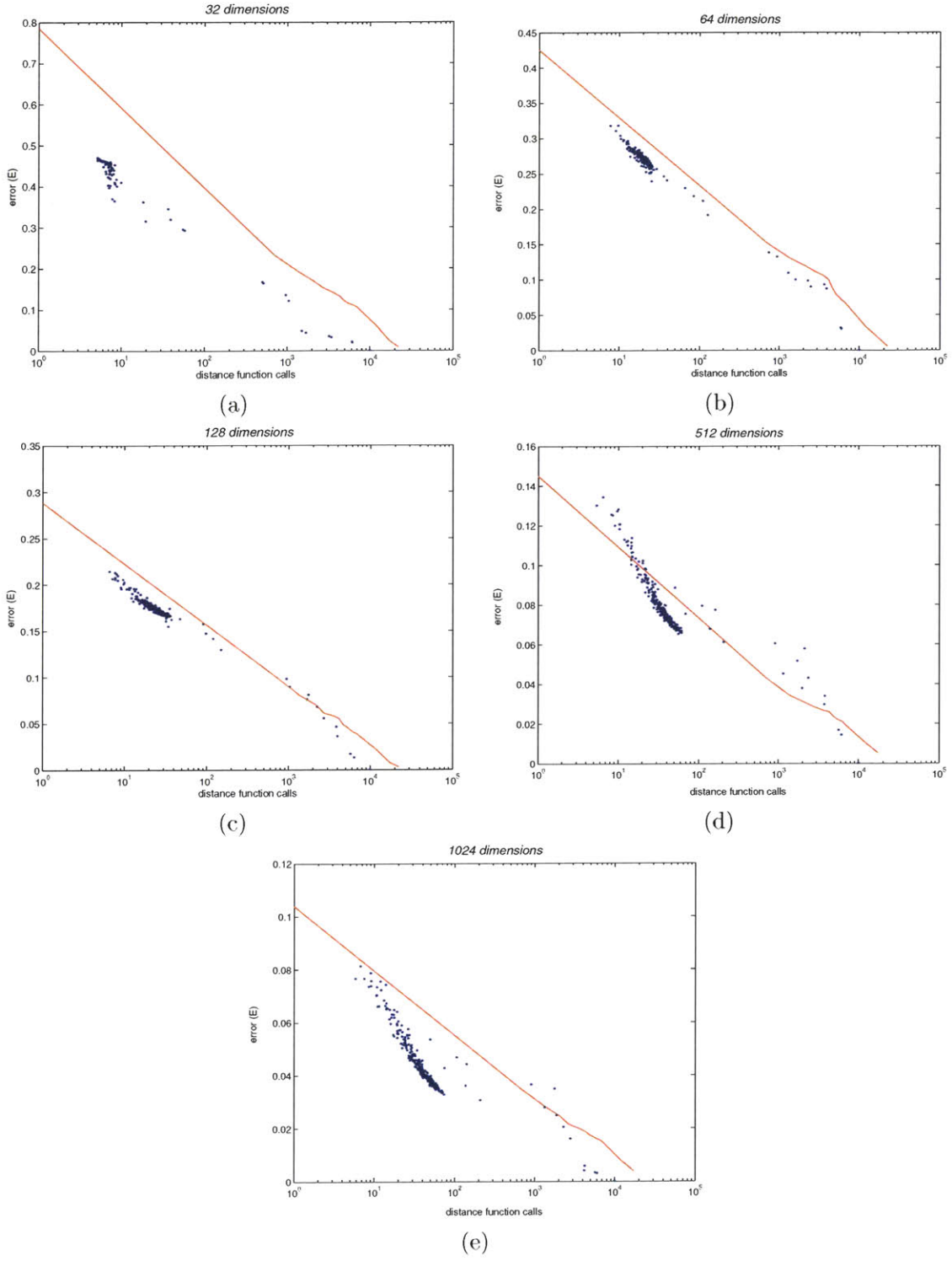


Figure 5-7: Error graphed as a function of the number of distance calculations for data in (a) 32 dimensions, (b) 64 dimensions, (c) 128 dimensions, (d) 512 dimensions, (e) and 1024 dimensions. The blue dots are for LSH whereas the red line is for BBF.

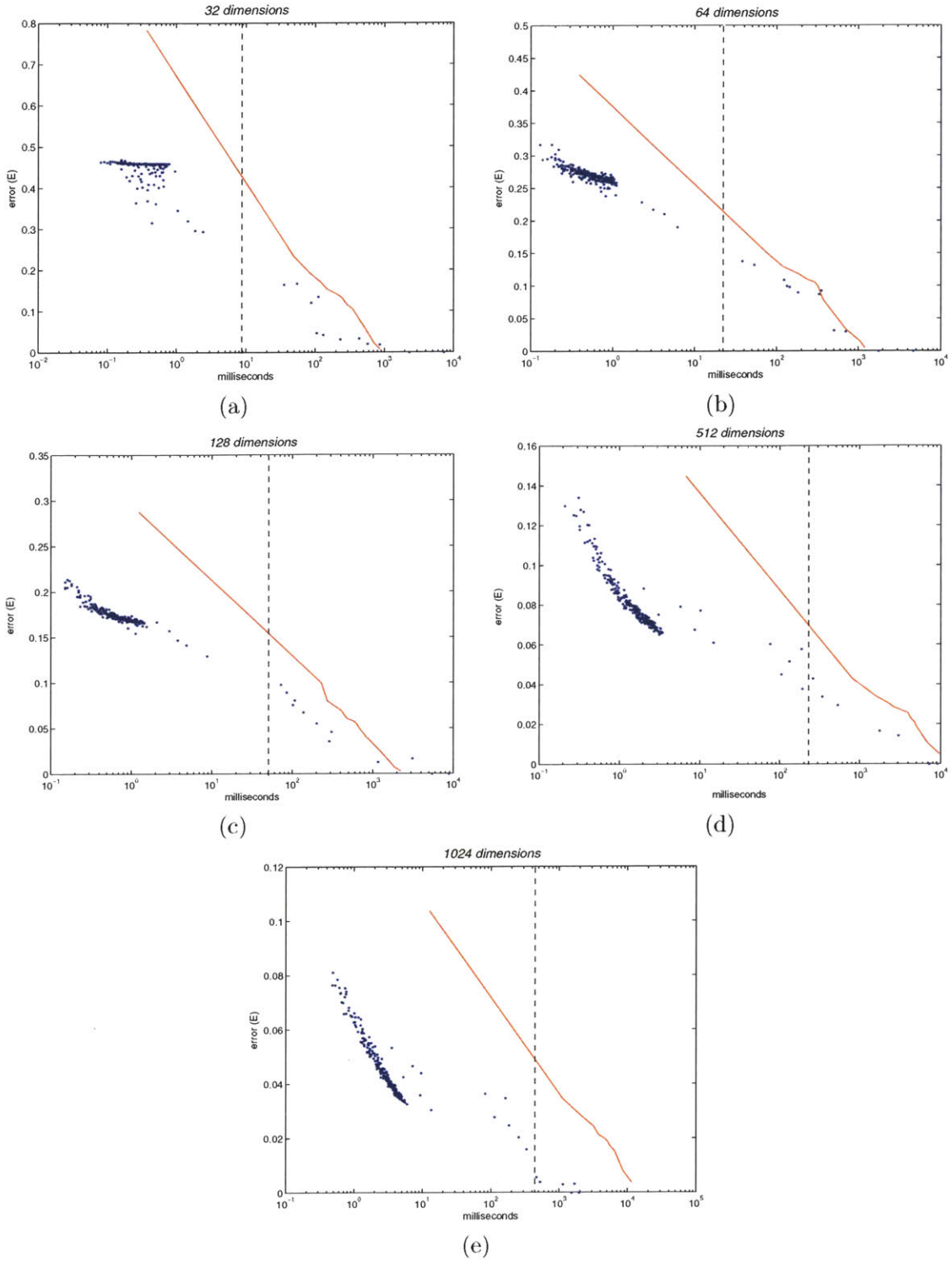
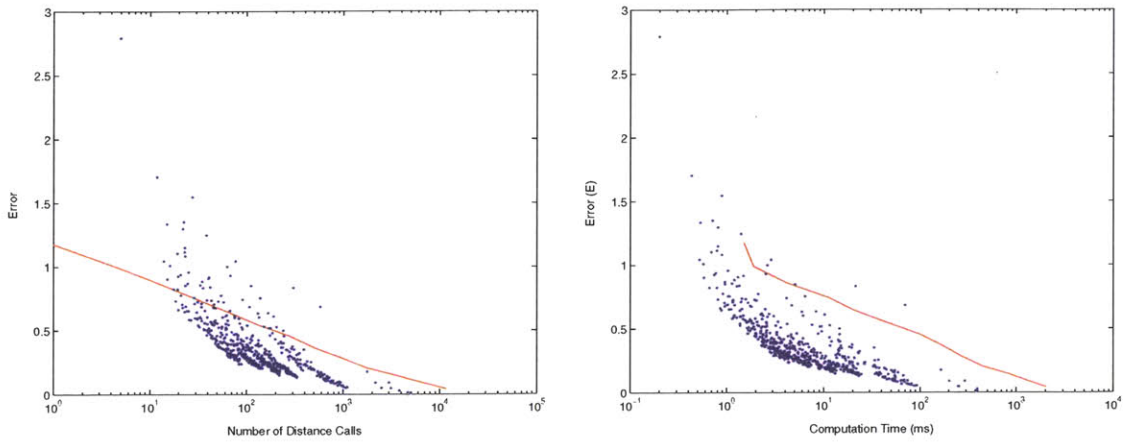


Figure 5-8: Error graphed as a function computation time in milliseconds for data in (a) 32 dimensions, (b) 64 dimensions, (c) 128 dimensions, (d) 512 dimensions, (e) and 1024 dimensions. The dotted line indicates the time required to perform an exact brute-force search with the additional optimization discussed in section 5.2. The blue dots are for LSH whereas the red line is for BBF.



(a) (b)

Figure 5-9: LSH vs. BBF using the 128-dimensional SIFT features extracted from a query test set of 20 images. 1000 features were selected at random from the approximate 40,000 features that were extracted from those 20 images. The training data used was the keypoints extracted from 10 background-subtracted template images. In this way, each template image corresponded to 2 test images. (a) shows the comparison in number of distance calls, while (b) shows the comparison in the number of milliseconds.

Chapter 6

Hypothesis Testing and Integration

This chapter covers the last step of a feature-based object recognition system, *hypothesis testing*. Recall from chapter 1 that hypothesis testing is a method of verifying all feature matches found, making sure that it is consistent with the orientation of the features on the original model. It also covers *integration*, which can be used to incorporate new views into a particular model to make it more robust.

We begin by discussing hypothesis testing, and then cover the method of model view integration.

6.1 Hypothesis Testing

Once we have a set of matches, we must determine whether those matches are consistent with the object model in question. For example, if the matches show that the model performed a flip, it is impossible that the object appears in the query image.

Furthermore, the orientation and scale data in each keypoint must be consistent throughout the matches for a particular object. If one match appears to have been scaled by 5 times its original value, whereas another match only was scaled by 2, then those matches are inconsistent with each other. Similarly, if the dominant orientation of one keypoint match shows a rotation of 30 degrees, whereas another match shows a rotation of 90 degrees, then those matches are inconsistent with each other.

6.1.1 Hough Transform

To this end, SIFT performs a Hough transformation to identify clusters of features with a consistent interpretation. Each match identifies four parameters in Hough space – x differential, y differential, scale differential, and orientation differential, (i.e., dx, dy, ds, $d\theta$). For all objects detected, each match votes for a different pose change for that object. The bin sizes for each parameter are 30 degrees for orientation, a factor of 2 for scale, and 0.25 times the maximum projected training image dimension for both x and y differentials.

An unfortunate aspect of the Hough Transform is that it is prone to boundary effects. For example, if a point lies very near to the edge of a bin, it could potentially be placed in the wrong bin due to errors in its parameter values. Therefore, a point is also added to all eight surrounding bins. That is, we increment only those bins which straddle the current bin along a given dimension. Note that this is different from placing the points in every 80 possible surrounding bin, as it would likely smooth the peak in the Hough transform too much to produce desirable results. (One can verify the number 80 by using the formula for the number of surrounding hyperrectangles in d-dimensional space: $n = 3^d - 1$).

6.1.2 Least-Squares Affine Transform Verification

Each cluster with more than 3 points in its bin is then subject to a geometric verification using an iterative least-squares method. The least-squares verification uses an affine transformation model. Although this does not account for 3D rotations of non-planar objects, it makes a relatively good approximation that would require more than 3 points otherwise [13].

An affine transformation from a model point $[x\ y]^T$ to a query point $[u\ v]^T$ can be written as follows:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} m_1 & m_2 & t_x \\ m_3 & m_4 & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

We wish to solve for the parameters $[m_1\ m_2\ m_3\ m_4\ t_x\ t_y]^T$, so we rewrite the above equation as follows:

$$\begin{bmatrix} x_1 & y_1 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_1 & y_1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 0 & 0 & 1 & 0 \\ 0 & 0 & x_n & y_n & 0 & 1 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ t_x \\ t_y \end{bmatrix} = \begin{bmatrix} u_1 \\ v_1 \\ \vdots \\ u_n \\ v_n \end{bmatrix}$$

This is simply a linear system $\mathbf{Ax} = \mathbf{b}$. The least-squares solution for $\mathbf{x}$ can be found using the familiar formula, $\mathbf{x} = [\mathbf{A}^T\mathbf{A}]^{-1}\mathbf{A}^T\mathbf{b}$, which minimizes the sum of squares of the distances from the projected model feature location to the query feature location.

This least-squares solution is applied in an iterative fashion, at each iteration removing outliers that differ from the projected model location by more than half the error range that was used in the Hough transform [13]. Experiments were performed to determine whether RANSAC could perform better, but this method produced better results.

In addition, we might have made mistakes in the initial Hough transformation, due to possible boundary effects, or errors in the orientation or scale values of the keys. Thus, at each iteration, a top-down search is added that incorporates any other matches that also agree with the projected model position. Note that this does not have the effect of ignoring the outcome of the Hough transform. It simply takes the results of the Hough transform and accounts for potential errors by adding other possible matches [13].

6.1.3 Accepting an Hypothesis

We now must determine whether we should accept a given hypothetical object match. The naive approach to accepting a hypothesis is to choose a threshold for the number of matches for a particular model, and accept the hypothesis if the number of matches is above that threshold.

The issue with this approach is that it does not take into account the reliability of various feature matches, nor the accuracy of the best-fit model solution. It also ignores the orientation and scale values of the keypoint matches, which could be slightly off due to the top-down search described above.

Put simply, a more robust model for hypothesis testing is needed. We instead determine

$P(m|f)$, where m is the presence of the model m at some pose, and f is the existence of k feature matches for that particular model [12].

To determine $P(m|f)$, Bayes rule is used:

$$\begin{aligned} P(m|f) &= \frac{P(f|m)P(m)}{P(f)} \\ &= \frac{P(f|m)P(m)}{P(f|m)P(m) + P(f|\neg m)P(\neg m)} \end{aligned}$$

We can approximate $P(f|m)$ as 1, since we expect to see at least k features present when the model m is present. We can also approximate $P(\neg m)$ as 1, since there is a very low probability that we will see a model at a particular pose [12]. Thus, we have

$$P(m|f) \approx \frac{P(m)}{P(m) + P(f|\neg m)} .$$

Note that if $P(f|\neg m) \ll P(m)$, then $P(m|f)$ approaches 1. Intuitively, this means that if the probability of false matches appearing for this model is much lower than our prior probability that this model will appear in this pose, then we have high confidence that this model exists in the query image.

Thus, we must determine the following values: $P(f|\neg m)$ and $P(m)$.

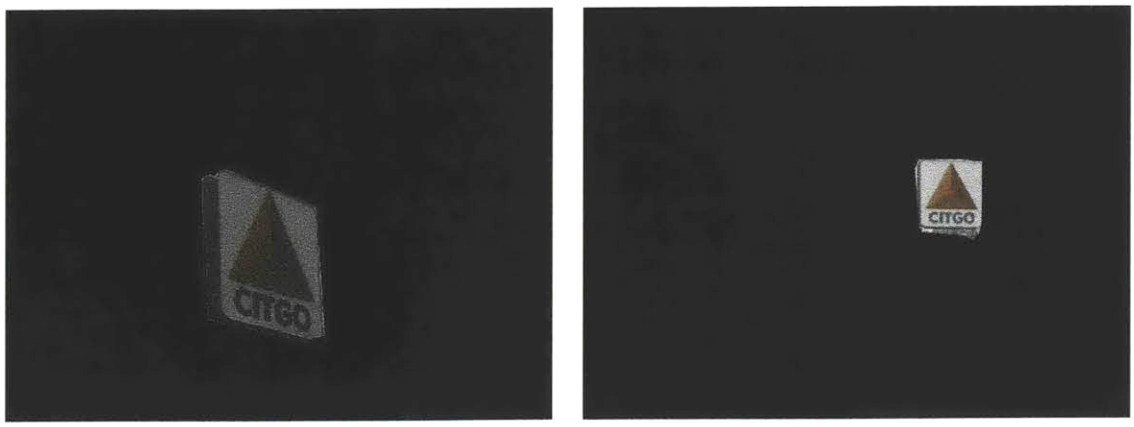
We start by determining $P(f|\neg m)$, the probability that our k feature matches arose by accident. First, we determine the number of matches, n, that are candidates for giving rise to such false matches. This can be done by simply counting the number of features within the projected bounding box of the original object [12].

Next, we find p, the probability that a single false match can arise. This value is determined following the approach outlined in [12].

Then, $P(f|\neg m)$ can be computed by using the binomial distribution function in the following manner:

$$P(f|\neg m) = \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j}$$

This can be computed efficiently by using an approach given in [15].



(a) (b)

Figure 6-1: Two example model views of the a particular landmark. The region surrounding the landmark in question is occluded so only those features useful in detecting that landmark will be extracted and added to the database.

Following the approach outlined in [12], $P(m)$ can be approximated as 0.01. Then, we accept a given hypothesis if $P(m|f) > 0.95$.

6.2 Integration

Following [12], I implemented a means of incorporating query images into the database. The general method is to first match a particular query image with a model, then use that image to add keypoints to the model, thus making it more robust. More formally, for each *model* in the database, we wish to have a collection of *model views*. For example, a model in the database could be the Citgo sign in Boston, and a particular model view could be a view of the Citgo sign from within Kenmore square, while another could be a view of the Citgo sign from across the Charles river at MIT. (See figure 6-1).

The rule for adding an image into a model is as follows. Using the affine solution to the equation $\mathbf{x} = [\mathbf{A}^T\mathbf{A}]^{-1}\mathbf{A}^T\mathbf{b}$ computed earlier, we determine the error, ϵ, between every model view in the system:

$$\epsilon = \sqrt{\frac{2\|\mathbf{Ax} - \mathbf{b}\|^2}{r - 4}}$$

where r are the number of rows in the matrix A. One determines whether or not to integrate a given image into a model by comparing ϵ to a threshold T. The value of T used

is the same as in [12], namely, 0.05 times the maximum dimension of the training image. As each new training image enters the system, it can either not match any model, match a model view, or match a model view so well that the error ϵ is less than T. Refer to [12] for more details. Although [12] focused on producing a system that could potentially bootstrap its way up from no existing models, I focused on building a simpler application that had a set number of models, and each model could simply be made more robust. This decision was made due to poor results using the bootstrapping method.

Chapter 7

The Application

The application I wished to create was a method of recognizing landmarks within the Boston area. The general idea is as follows: I would create a "virtual tour guide" of a particular region by allowing people to use a device capable of taking photographs and sending these photographs to a server, which would then respond with two possible outcomes:

- The item is not recognized in the database of models. At this point, the user could be instructed to try a more frontal view of the landmark in question.
- The item matched a particular landmark. Various information regarding the landmark in question may then be retrieved from the device, perhaps including information entered by users of the system. In addition, a user may choose to select the region more carefully using a simple bounding box in order to incorporate the model view into the match, using the method described in chapter 6.

The device used could be a cell phone with a camera attached. Such a device is relatively easy to program, and has been used to create a landmark detection system very similar to the one I describe [22].

7.1 Choice of System

From the results of chapters 4 and 5, I decided to use the original SIFT code with LSH approximate nearest neighbor searching. SIFT-PCA simply did not perform well on my data set, and, in addition, did not seem extensible enough to create a system that could recognize

30 different models with multiple model views. That is, performing SVD decomposition on such a large matrix would be too memory-intensive and time-consuming.

Furthermore, LSH performed much better than BBF when estimating the distances for SIFT keyoints. This was especially true when a reasonably low error was desired, as the case needed to be to have any reasonable performance.

7.2 Data Collection

I took a large number photographs of various landmarks from around the Boston area using a digital camera. Although the images were taken at 640×480 resolution, they were sub-sampled to 0.7 times their original size, or 448×336, when inputted into the system. This choice of resolution was made since the call to David Lowe's keypoint extraction code took too long on the original 640×480 images, and the loss in accuracy that resulted was minimal.

I separated all images into 30 models, each model consisting of two separate test sets:

1. **Test Set A**: Contains 5 images of the landmark, sometimes in various lighting conditions, but all within approximately $20°$ of a frontal pose.

2. **Test Set B**: Contains 5 more difficult images of the landmark, sometimes taken at a greater distance (for example with the citgo sign), and often within approximately $40 - 80°$ of a frontal pose.

For each model, I had between 2 to 5 background-subtracted template images of the landmark in question. The first of these images is the template images used when no integration was allowed. When I allowed model integration, all available template images were used.

Model integration simply involved initializing the system by starting off with the first template image, then running a query on all other template images available, only using those keypoints found on the model in question. For example, for model 0, I would load its first template image. Then, I would query the system with its second template image, allowing it to only search through the keypoints in model 0. Based on the rules outlined in section 6.2, I either added a new view to the model, or integrated the keypoints into an existing model view. If the system could not find a match with any currently existing model

views, I simply created a new model view without linking any matching features. Then, I would query this updated system with the third template image, now using all keypoints found in both the first and second template images. This process would repeat until all template images for all models were loaded into the system.

The queries that took place during the initialization stage all ran using a naive brute-force matching scheme, to ensure that the matches were correct. Furthermore, I used a nearest-neighbor ratio cutoff level of 0.8, whereas in subsequent testing, the tolerance was reduced to 0.69 to minimize the number of incorrect matches resulting from the approximate nearest-neighbor error.

7.3 Featureless Models

As mentioned in the introduction, I found it extremely difficult to match objects that were relatively featureless. Although some features were extracted from such models, they paled in comparison to the number of features that were extracted from the other models. Furthermore, when the approximate nearest-neighbor matching algorithm was applied, it became nearly impossible to recognize these models.

I was hoping that by integrating a number of model views into the model, detection could be made more robust. This, unfortunately, was only the case when a brute-force nearest-neighbor matching method was used, and even then, almost identical lighting and pose conditions were required in the model to recognize an object. The only solution was to remove these models from the data set. Examples of such models are shown in figure 7-1.

This should come as no surprise, however. The models used for David Lowe's SIFT paper were replete with intricate details and features. (See figure 7-2).

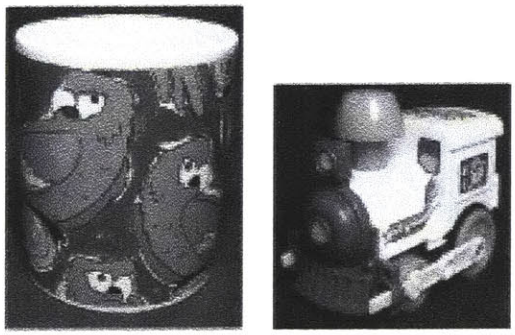
The final data set can be found in appendix A.

7.4 Experiments

I performed experiments on both data sets, with and without LSH, and with and without integration. Perhaps the most difficult aspect of these experiments was determining the correct parameters to use.

(a) (b) (c)

Figure 7-1: Examples of models that could not be reliably detected by the system. If I increased the nearest-neighbor matching threshold, these objects ended up being detected, but the number of false positives then increased dramatically. The only solution was to remove them from the data set.



(a) (b)

Figure 7-2: Examples of models used in David Lowe's SIFT experiments. Note the intricacy of these models, especially in (a).

I found the most important parameter to be determined was the nearest-neighbor ratio threshold. This parameter, as discussed in [13], is defined as the ratio of the distance to the closest match to that of the second-closest match.

The point of this parameter is to give a quantitative meaning to the "stability" of a keypoint match. Any match with a high ratio (close to 1.0) will be relatively unstable, as it implies that there are many keypoints that closely match within the data space. This situation could correspond to, for example, a repetitive texture on an image, which produces a large number of keypoints with very similar keypoint descriptors. When an approximate nearest-neighbor matching algorithm such as LSH is used, it is less likely that the exact nearest neighbor match will be found if the correct match is unstable, as there are so many similar keypoints in the data set.

Thus, when an approximate nearest neighbor matching algorithm is used, it is best to tighten this threshold, cutting off more "unstable" keypoint matches, as they are even more likely to be incorrect. Of course, if the ratio is set too low, however, we begin to cut off too many "stable" matches, producing excellent false positive resilience, but worse algorithm correctness.

I found that regardless of how I tuned the LSH parameters, I could not achieve the accuracy of brute-force matching, without setting them so high as to make the matching stage intolerably slow. Instead, I relaxed the LSH parameters (my final choices were $L = 160$, $\alpha = 3.0$, and $k_1 = k_2 = 14$) and opted for a fast solution whose matching accuracy was no less than 85%. In addition, I reduced the closest-to-next-closest ratio threshold, hoping that it would cut out these incorrect nearest neighbor matches. In the end, I found that a value of 0.69 as a threshold produced the best results for data sets A and B using LSH (both with and without integration). These results are shown in figure 7-3 and 7-4.

7.5 Discussion

As is evident in figure 7-5, LSH drastically improved the speed of the system while introducing only a minor amount of error. Furthermore, the error did not manifest itself as false positives or incorrect matches, since we reduced the threshold. Rather, the images were simply not recognized. If the system offered an incorrect match, it would get to the point where the user would simply never use the system anymore. Instead, it is guaranteed that

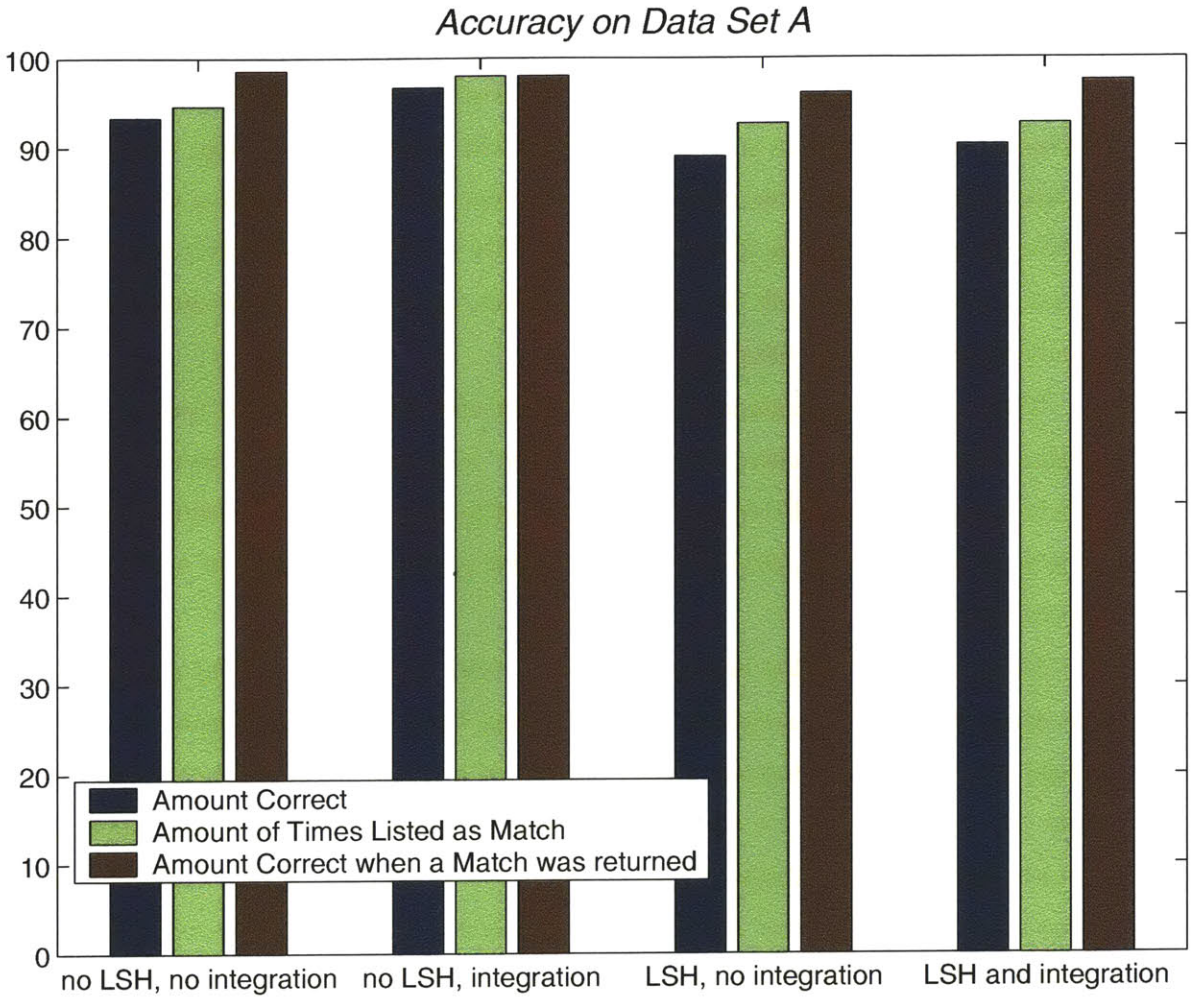


Figure 7-3: Data set A with and without LSH, and with and without integration at optimal settings. False positives were restricted to be less than 1%. In (a), the first bar in each group shows the amount correct, then the amount listed as a match, and finally the amount correct given that a match was returned. For LSH, optimal parameters were found by experimenting with data set A and then keeping the same settings when running data set B. The final choices for LSH parameters were $l = 160$, $\alpha = 3.0$, $k_1 = 14$, and $k_2 = 14$, with a resulting $B = 14$.

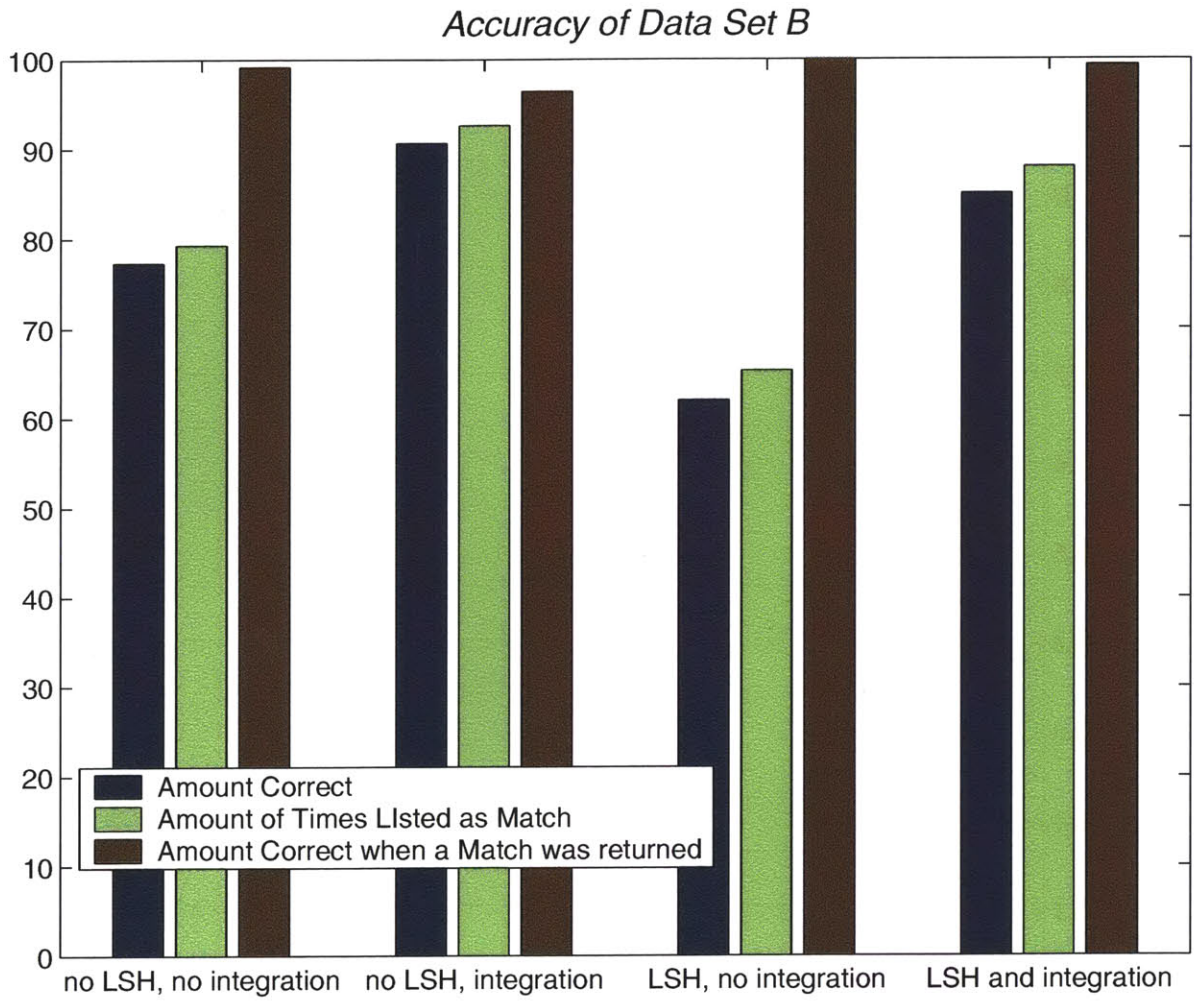


Figure 7-4: Data set B with and without LSH, and with and without integration at optimal settings. False positives were restricted to be less than 1%. In (a), the first bar in each group shows the amount correct, then the amount listed as a match, and finally the amount correct given that a match was returned. For LSH, optimal parameters were found by experimenting with data set A and then keeping the same settings when running data set B. The final choices for LSH parameters were $l = 160$, $\alpha = 3.0$, $k_1 = 14$, and $k_2 = 14$, with a resulting $B = 14$.

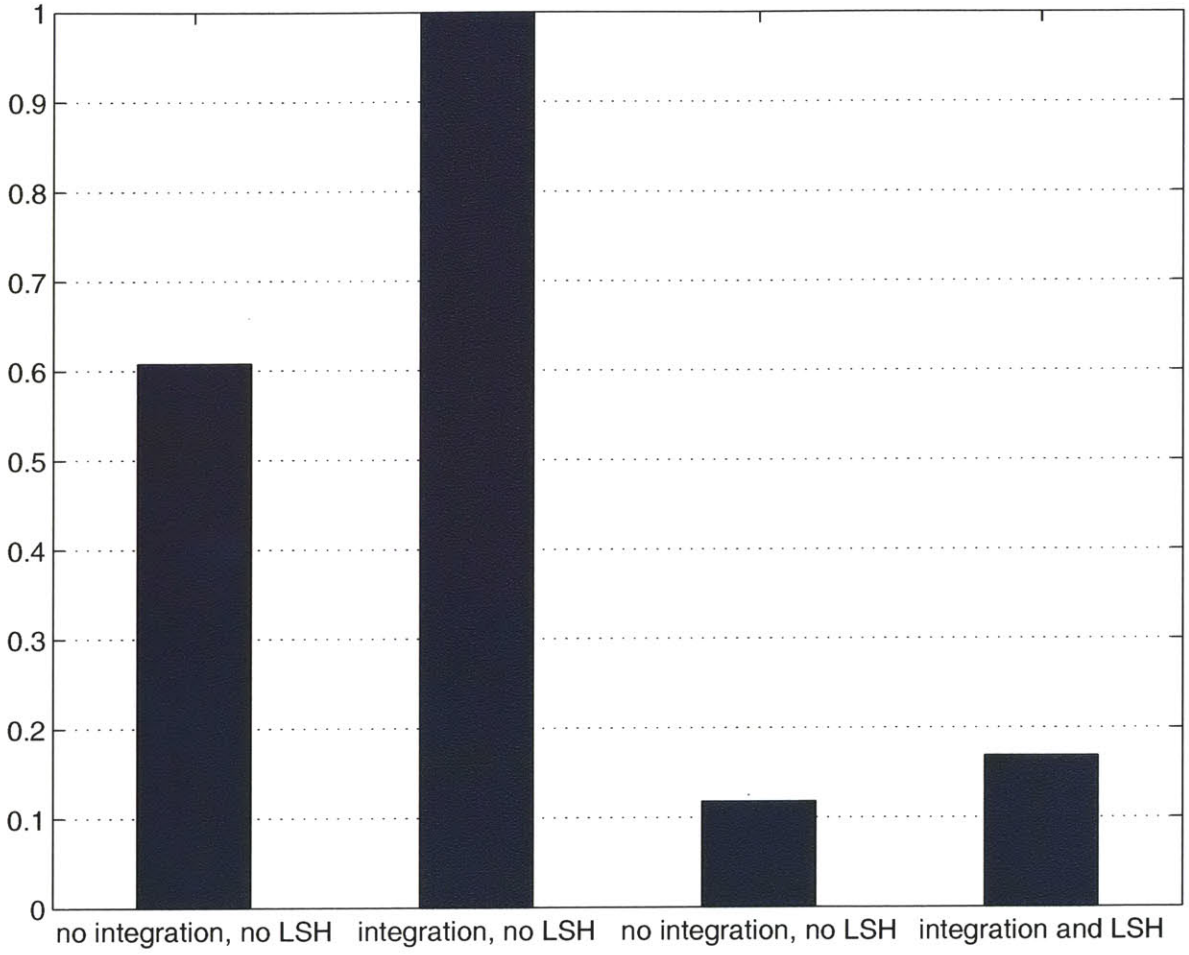


Figure 7-5: Average matching times in terms of the time taken to perform a query with integration and without LSH. Note that there is a substantial improvement in the speed of the system, costing only a moderate match loss 7-4

if a match is returned, chances are high that it is correct (evident from the final bar in each group of figures 7-3 and 7-4).

Furthermore, as expected, integrating a number of different views into each model added a substantial improvement in accuracy in data set B, since many of the images in data set B were taken from side views or other views of the model.

Chapter 8

Contributions

In this thesis, I have made the following contributions:

- I have performed an extensive empirical comparison between LSH and BBF, finding that LSH is in general a faster, more accurate approximate nearest-neighbor matching method.
- I have compared SIFT to PCA-SIFT to find that, unlike in [9], SIFT performs better, although this might be simply due to the data set used.
- Based on my findings, I have built a system that accurately detects landmarks in the vicinity of Boston and Cambridge. I have also demonstrated the ability for the system to be made more robust by maintaining multiple model views.
- I have shown that SIFT does not perform reasonably well when recognizing objects that have relatively uniform texture and color. These "featureless" objects had to be removed from my final data set due to the fact that they could not be accurately recognized.

8.1 Future Work

There is a tremendous amount that can be done to extend the work of this thesis. In terms of the LSH and BBF comparison, it is not yet clear that LSH outperforms BBF on all types of synthetic data, and it would be beneficial to compare the two on many different data sets, as well as different data sizes.

There is also work that can be done to optimize the system already built, for example by writing an implementation of LSH in assembly. Furthermore, such a system has yet to be shown to perform well on a larger data set.

Perhaps one of the most interesting extensions to this system can be to improve it so as to be able to bootstrap from a few models into a full-blown object recognition system, much like the same way children learn objects by observing them and matching them. It is clear that, currently, such a system would not be able to run self-sufficiently. Any error introduced by an incorrect match would only become compounded, and thus an outside mediator would need to ensure that the models remain consistent. In addition, the restrictions on reasonable behavior for such a system become tighter – that is, not only does each match have to be correct, but there cannot be any misses. Namely, the system cannot return that it does not recognize an object when that same object is listed in its database.

Perhaps the broadest implication of this thesis revolves around the fact that SIFT, and other feature-based object recognition methods, must be extended in some way so that they can recognize more uniform objects, such as those that had to be removed from the data set due to extremely poor recognition. It is very likely that any feature-based object recognition system can be improved tremendously if global descriptions are incorporated into each model. Perhaps, a system could be built that could choose which object recognition model to use as a pre-processing step so as to maximize the number of correct responses.

Appendix A

Data Set

Models

Set A

Set B

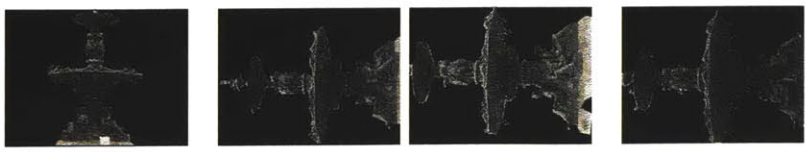
Figure A-1: Model 000

Models

Set A

Set B

Figure A-2: Model 001

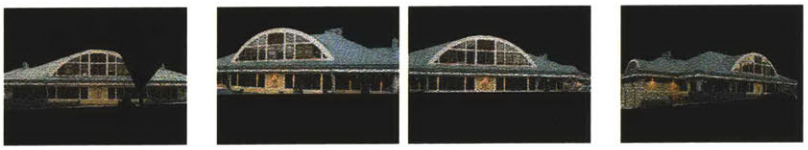


Models

Set A

Set B

Figure A-3: Model 002

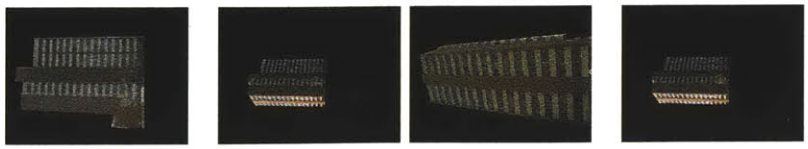


Models

Set A

Set B

Figure A-4: Model 003

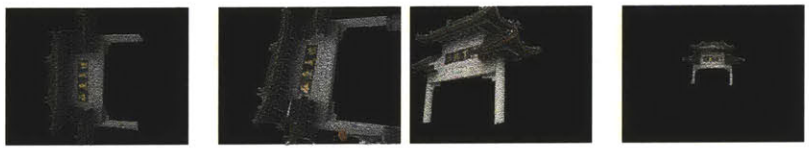


Models

Set A

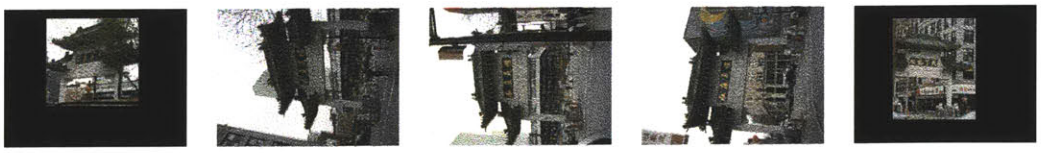
Set B

Figure A-5: Model 004



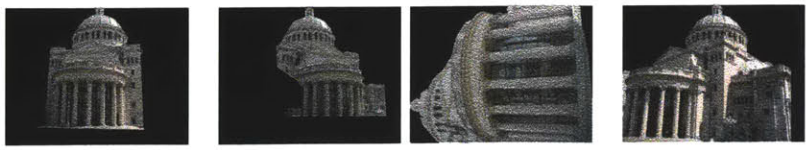
Models

Set A



Set B

Figure A-6: Model 005



Models

Set A

Set B

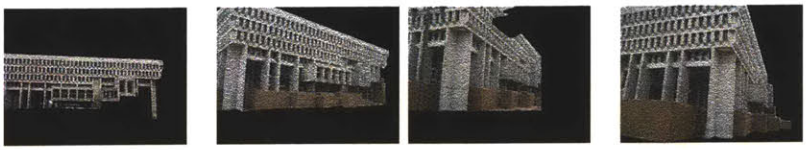
Figure A-7: Model 006

Models

Set A

Set B

Figure A-8: Model 007



Models

Set A

Set B

Figure A-9: Model 008

Models

Set A

Set B

Figure A-10: Model 009

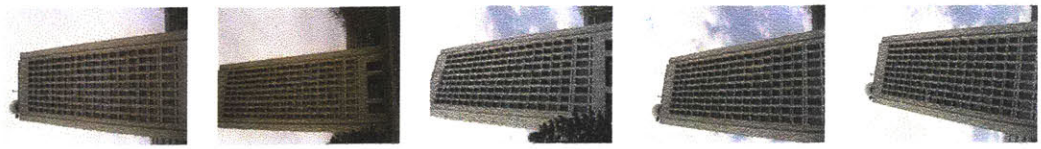
Models

Set A

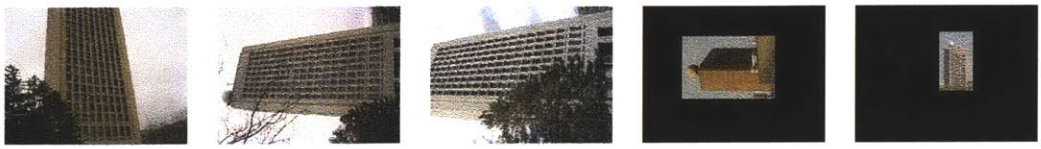
Set B

Figure A-11: Model 010

Models



Set A



Set B

Figure A-12: Model 011

Models

Set A

Set B

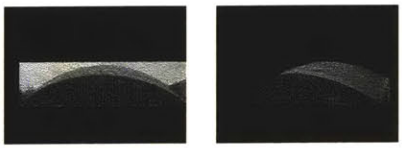
Figure A-13: Model 012

Models

Set A

Set B

Figure A-14: Model 013

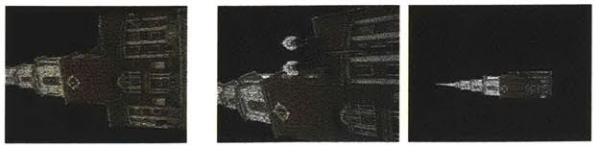


Models

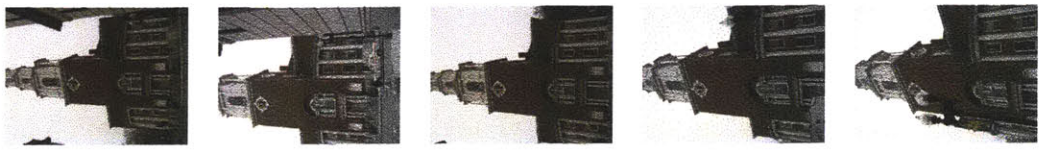
Set A

Set B

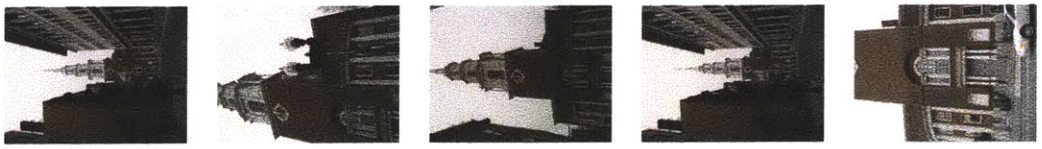
Figure A-15: Model 014



Models



Set A



Set B

Figure A-16: Model 015

Models

Set A

Set B

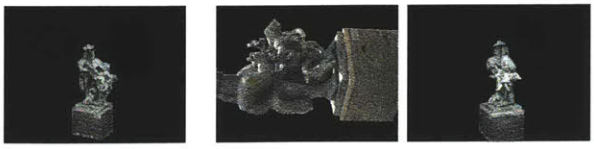
Figure A-17: Model 016

Models

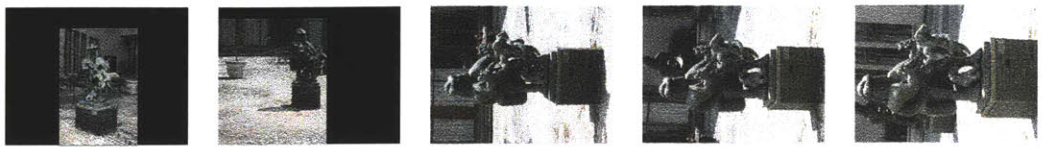
Set A

Set B

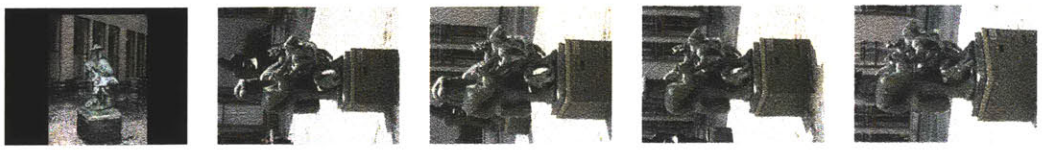
Figure A-18: Model 017



Models

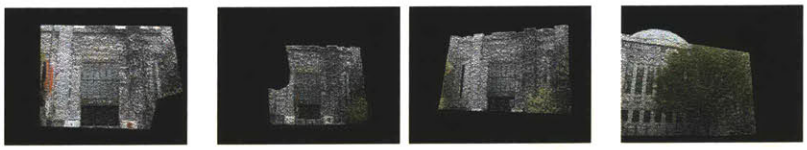


Set A



Set B

Figure A-19: Model 018



Models

Set A

Set B

Figure A-20: Model 019

Models

Set A

Set B

Figure A-21: Model 020

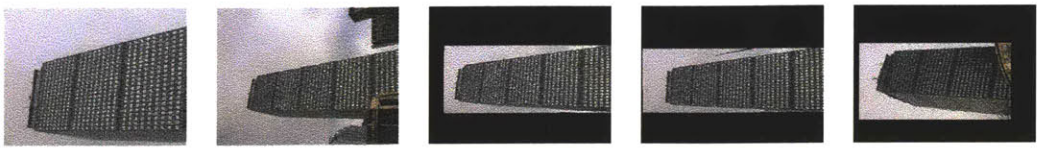
Models

Set A

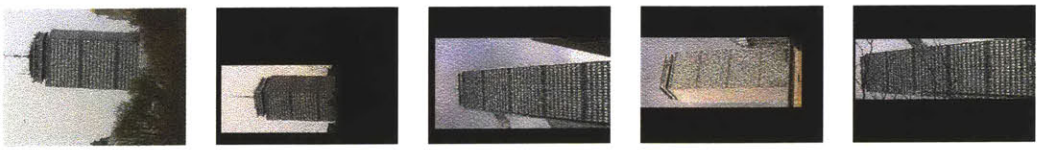
Set B

Figure A-22: Model 021

Models



Set A



Set B

Figure A-23: Model 022

Models

Set A

Set B

Figure A-24: Model 023

Models

Set A

Set B

Figure A-25: Model 024

Models

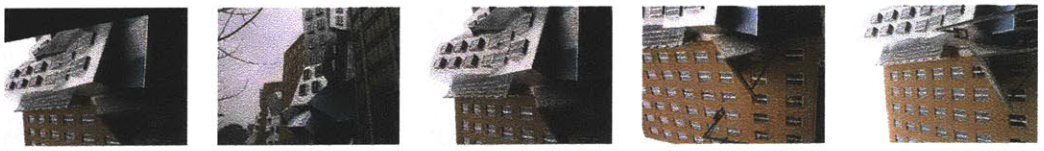
Set A

Set B

Figure A-26: Model 025

Models

Set A

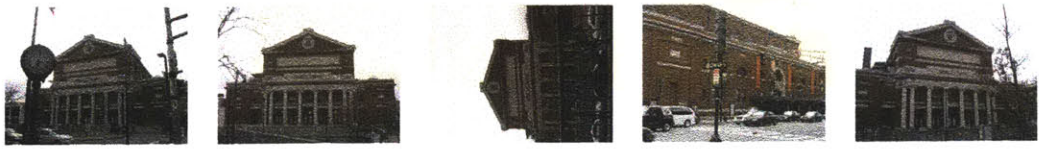


Set B

Figure A-27: Model 026

Models

Set A



Set B

Figure A-28: Model 027

Models

Set A

Set B

Figure A-29: Model 028

Models

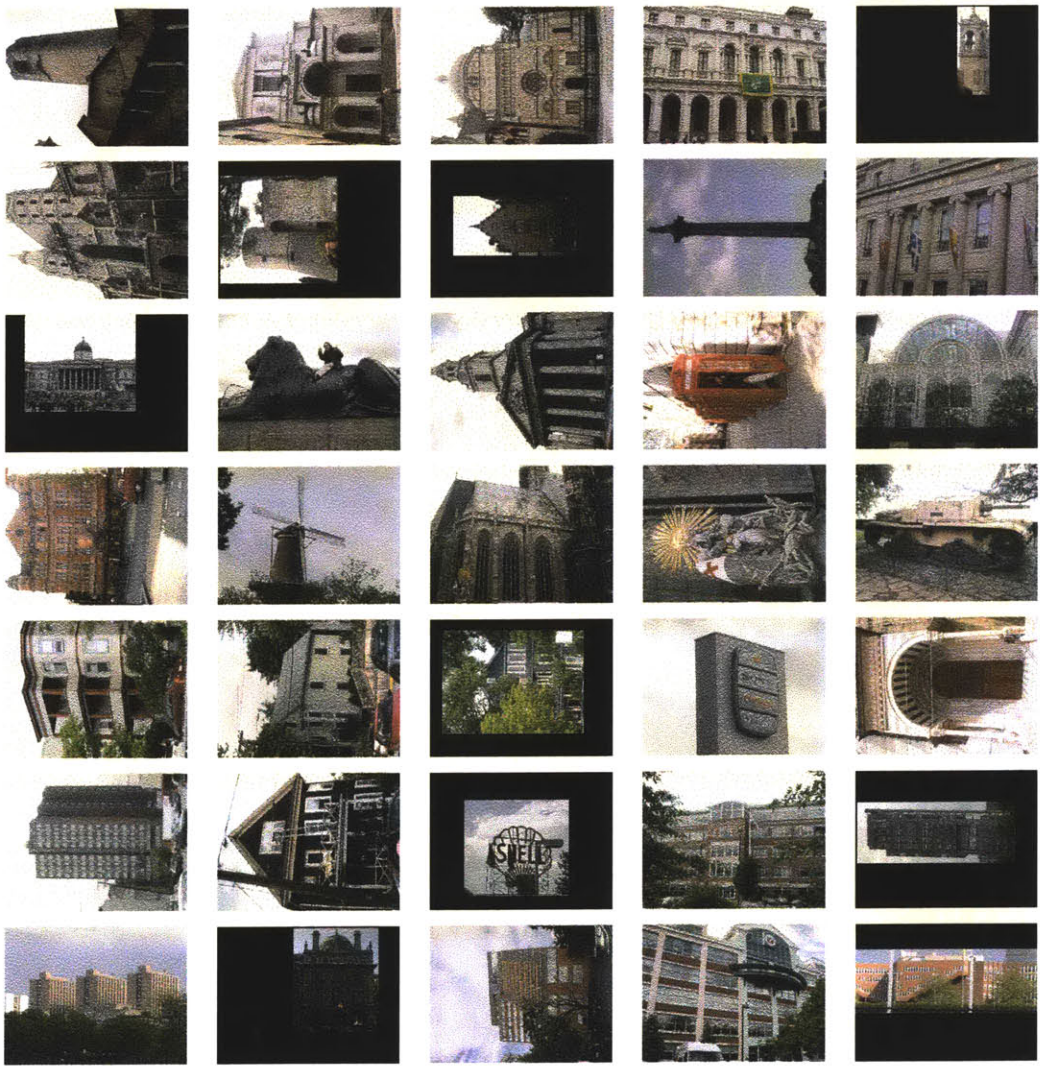
Set A

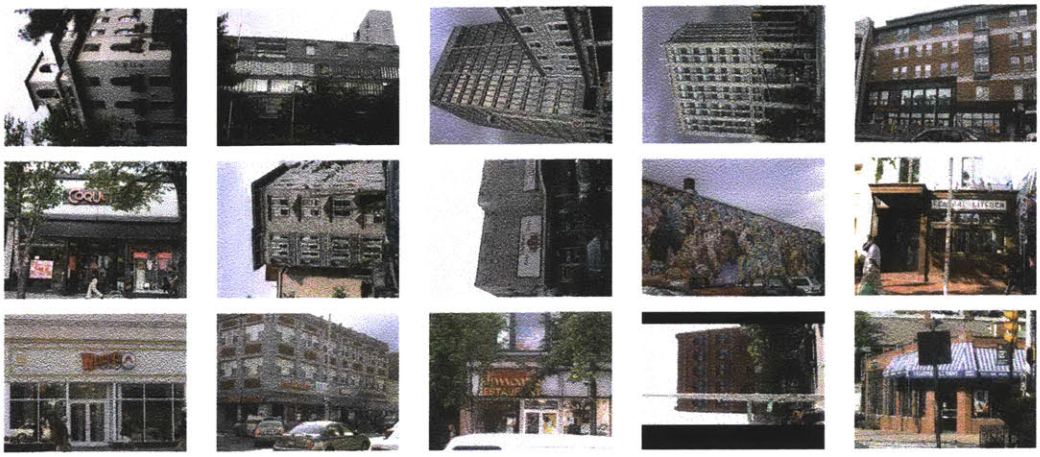
Set B

Figure A-30: Model 029

Appendix B

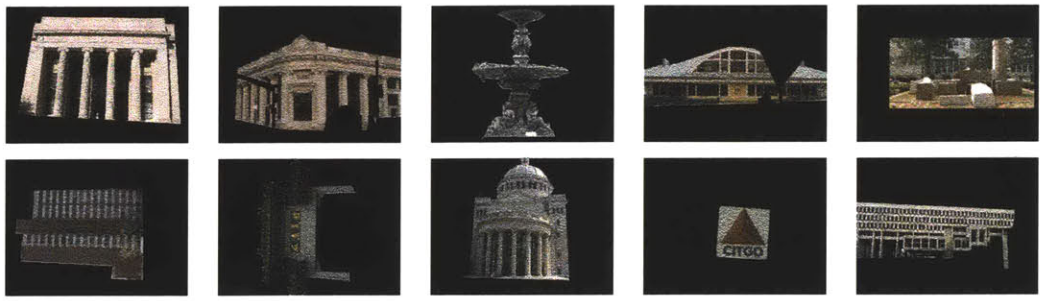
False Positive Set





Appendix C

Nearest-Neighbor Test Data Set



Training Data

Test Data

Bibliography

[1] J. Beis and D. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Conference on Computer Vision and Pattern Recognition*, pages 1000–1006, Puerto Rico, June 1997.

[2] S. Edelman, N. Intrator, and T. Poggio. Complex cells and object recogntion. In *NIPS*97*, 1997.

[3] D. Forsyth and J. Ponce. *Computer Vision: A Modern Approach.* Prentice Hall, Upper Saddle River, NJ, 2003.

[4] W. Freeman and E. Adelson. The design and use of steerable filters. *IEEE Transactions Pattern Analysis and Machine Intelligence*, 13(9):891–906, September 1991.

[5] J. Friedman, J. Bentley, and R. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(2):209–226, September 1977.

[6] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th VLDB Conference*, pages 518–529, Edinburgh, Scotland, 1999.

[7] B. Horn and B. Schunck. Determining optical flow. Massachusetts Institute of Technology AI Memo 572, 1980.

[8] M. Hu. Visual pattern recognition by moment invariants, February 1962.

[9] Y. Ke and R. Sukthankar. Pca-sift: A more distinctive representation for local image descriptors. *Conference on Computer Vision and Pattern Recognition*, 2004.

[10] T. Lindeberg. *Scale-Space Theory in Computer Vision.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1994.

[11] D. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision*, September 1999.

[12] D. Lowe. Local feature view clustering for 3d object recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Kauai, Hawaii, December 2001.

[13] D. Lowe. Distinctive image features from scale-invariant keypoints. Accepted for publication in the International Journal of Computer Vision, 2004, January 2004.

[14] K. Mikolajczyk and C. Schmid. A performance evaulation of local descriptors.

[15] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C++.* Cambridge University Press, 2002.

[16] C. Schmid and R. Mohr. Local grayvalue invariants for image retrieval. *IEEE Transactions Pattern Analysis and Machine Intelligence*, 19(5), May 1997.

[17] J. Shi and C. Tomasi. Good features to track. In *Conference on Computer Vision and Pattern Recognition*, Seatle, June 1994.

[18] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *Proceedings of the 9th IEEE International Conference on Computer Vision (ICCV 2003)*, 2003.

[19] G. Strang. *Linear Algebra and its Applications.* Thomson Learning, inc., third edition, 1988.

[20] O. Trier, A. Jain, and T. Taxt. Feature extraction methods for character recognition - a survey, 1996.

[21] E. Weisstein. *CRC Concice Encyclopedia of Mathematics.* Chapman and Hall/CRC, second edition, 2003.

[22] T. Yeh, K. Tollmar, and T. Darrell. Searching the web with mobile images for location recognition. In *Conference on Computer Vision and Pattern Recognition*, 2004.