UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Texture-boundary detection in real-time

A thesis submitted in partial fulfilment of the requirements for the Degree of

Doctor of Philosophy

in the University of Canterbury

by
Ray Hidayat

2010

Supervisor: Dr. Richard Green
Co-supervisor: A/Prof. R. Mukundan

PUBLICATIONS

The research presented in this thesis has been published in the following peer-reviewed papers:

Hidayat, R. and Green, R. (2009). Real-time texture boundary detection from ridges in the standard deviation space. *British Machine Vision Conference*, London, United Kingdom, September 2009.

Hidayat, R. and Green, R. (2008). Texture-suppressing edge detection in real-time. *Image and Vision Computing New Zealand*, Lincoln, New Zealand, November 2008.

ABSTRACT

Boundary detection is an essential first-step for many computer vision applications. In practice, boundary detection is difficult because most images contain texture. Normally, texture-boundary detectors are complex, and so cannot run in real-time. On the other hand, the few texture boundary detectors that do run in real-time leave much to be desired in terms of quality. This thesis proposes two real-time texture-boundary detectors – the Variance Ridge Detector and the Texton Ridge Detector – both of which can detect high-quality texture-boundaries in real-time. The Variance Ridge Detector is able to run at 47 frames per second on 320 by 240 images, while scoring an F-measure of 0.62 (out of a theoretical maximum of 0.79) on the Berkeley segmentation dataset. The Texton Ridge Detector runs at 10 frames per second but produces slightly better results, with an F-measure score of 0.63. These objective measurements show that the two proposed texture-boundary detectors outperform all other texture-boundary detectors on either quality or speed. As boundary detection is so widely-used, this development could induce improvements to many real-time computer vision applications.

ACKNOWLEDGEMENTS

First and foremost, thank you to my supervisor Dr. Richard Green for making this work possible. Your direction and forward-thinking always made sure this research was heading in the right direction.

Thank you to my co-supervisor A/Prof. Mukundan for the positive encouragement and useful feedback.

Thank you to my fellow computer scientists for making this a pleasant learning experience.

Thank you also to the Tertiary Education Commission, University of Canterbury and Freemasons New Zealand for the awards of the Top Achiever Doctoral Scholarship, Canterbury Scholarship and the Freemasons Postgraduate Scholarship respectively, which all made this research possible.

Most of all, thank you to my parents, Rudy and Rini Hidajat, and brother Ryan, for your constant love and support. I could not have done this without you.

TABLE OF CONTENTS

1 INTRODUCTION

A boundary detector is an algorithm that finds boundaries – the borders that divide different parts of the same image (Martin, Fowlkes, & Malik, 2004). An example of this is illustrated below in Figure 1-1.

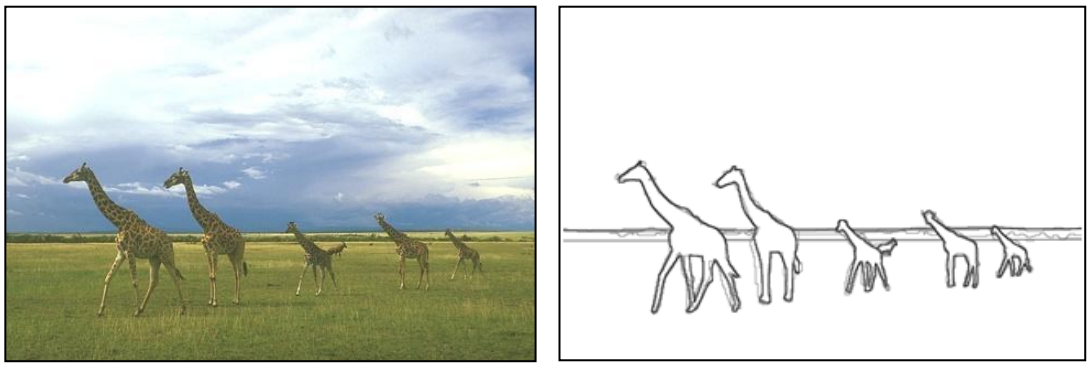


FIGURE 1-1: An image (left) and its boundary map (right) according to human subjects. These images were taken from the Berkeley segmentation dataset and benchmark (Martin, Fowlkes, Tal, & Malik, 2001).

1.1 APPLICATIONS OF BOUNDARY DETECTION

Boundary detection is an essential step to many computer vision applications.

In automatic **car/robot navigation**, the car or robot must know where the boundaries of its obstacles are in order to drive around them. Figure 1-2 below illustrates a system (Vaudrey, Wedel, Rabe, Klappstein, & Klette, 2008; Klappstein, Vaudrey, Rabe, Wedel, & Klette, 2009) that endeavours to automatically drive a car. It is essential that the system knows where the boundaries of the moving objects are so that it can avoid collisions.

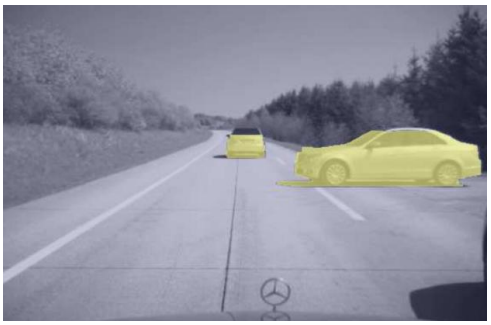


FIGURE 1-2: This algorithm automatically detects boundaries of moving objects so that they can be avoided. Image taken from Vaudren *et al.* (2008).

Face detection/recognition sometimes uses boundary detection to identify the parts of the face or the position of the face as a whole. Figure 1-3 below illustrates a face detector (Hsu, Abdel-Mottaleb, & Jain, 2002) which locates the position of the face from the boundaries in the image.

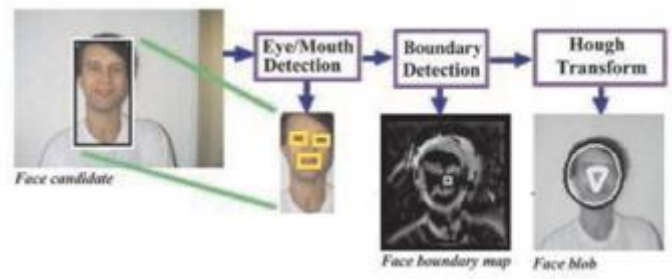


FIGURE 1-3: This face detector uses boundary detection to identify the position of the face. Reproduced from Hsu *et al.* (2002)

One of the most widely-used methods for **object model reconstruction** is visual hull carving. This technique takes multiple images of the object from many different views, and then uses those views to sculpt the object out of a cube. The boundary map for each view is used as a stencil for the carving process – it determines where the algorithm should carve the cube. Figure 1-4 illustrates a system that does this (Furukawa & Ponce, 2009).

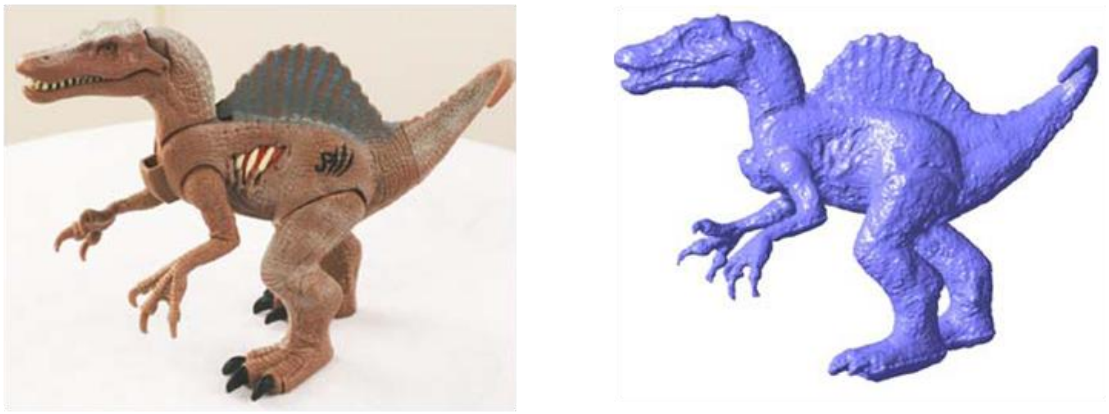


FIGURE 1-4: The object shown in the left image was sculpted into a visual hull (right) using boundary detection. Reproduced from Furukawa and Ponce (2009).

Drummond and Cipolla (2002) developed an efficient and robust method of **3D object model tracking** which involves tracking only the boundaries of the object. What makes it so efficient is that boundaries are one-dimensional, which means that tracking only boundaries reduces the number of degrees of freedom substantially. This is illustrated below in Figure 1-5.

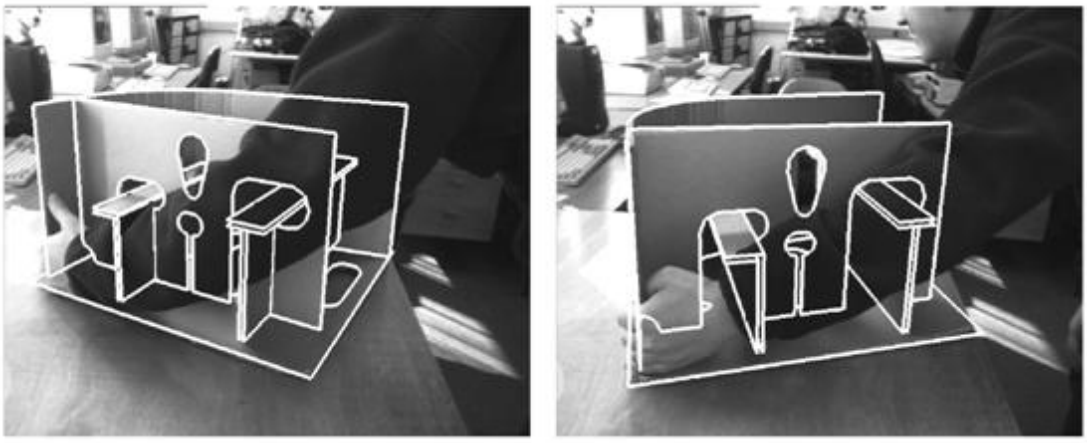


FIGURE 1-5: This object tracking technique works by comparing the boundaries of a known object model with the actual boundaries of the image, found using a boundary detector (Drummond & Cipolla, 2002).

Figure 1-6 below illustrates a technique for **object recognition** which uses boundary detection (Shotton, Blake, & Cipolla, 2008). Obviously, objects are often made up of characteristic shapes, and so boundary detection is needed because it reveals the shapes in an image.

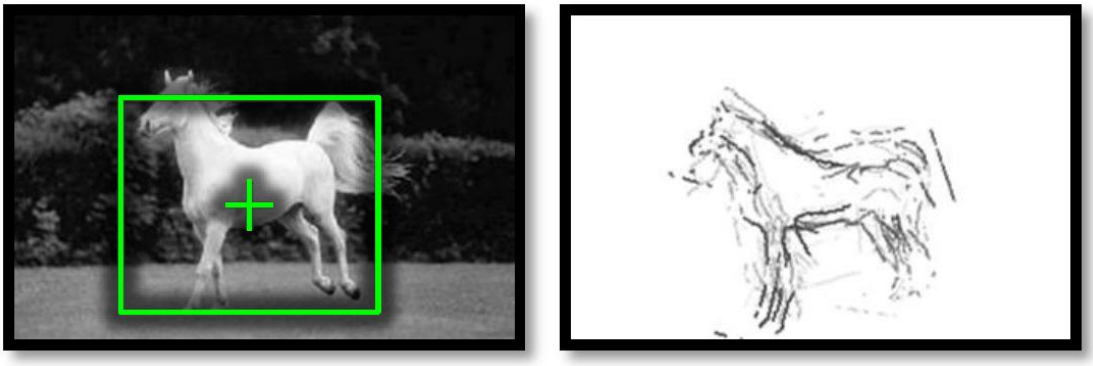


FIGURE 1-6: This system has recognised a horse object (left) from a collection of learnt boundary fragments (right). Reproduced from Shotton *et al.* (2008).

Inpainting, or **object removal**, involves painting over an object in order to make it look like the object was never in the image. Boundary detection can determine automatically where the inpainting should occur. Figure 1-7 shows a technique (Whyte, Sivic, & Zisserman, 2009) that has removed an object, given its boundaries.

FIGURE 1-7: Boundary detection can be used to identify where inpainting should occur. Reproduced from Whyte *et al.* (2009)

The above examples have shown that boundary detection is a key step to many computer vision applications such as robot navigation, face detection and recognition, object model reconstruction, object tracking, object recognition and object removal. That is why any improvements to boundary detection are very useful.

1.2 BOUNDARY DETECTION WITHOUT TEXTURE

Most boundary detectors rely on one basic assumption: a boundary exists wherever there is **significant** change in the image (Sobel & Feldman, 1973). The problem here is, when is the change **significant**?

Early attempts at boundary detection assumed that any large changes were significant. The well-known Canny edge detector (Canny, 1986) is a good example of an algorithm that follows this decision rule. Sometimes, this does not work very well:

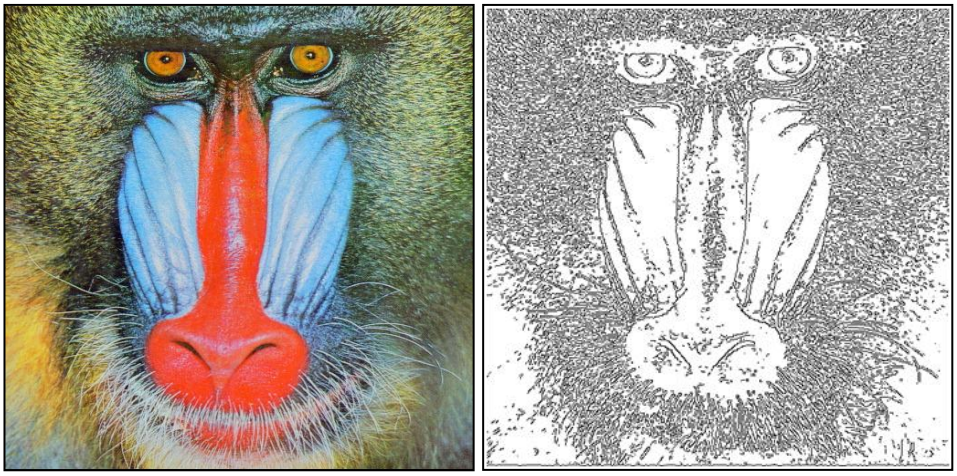


FIGURE 1-8: The Canny edge detector is applied to an image of a mandrill (left), producing a boundary map (right).

In Figure 1-8, there are great variations in colour within the fur of the mandrill. The Canny edge detector has identified each of these variations as a boundary. This has meant that the important boundaries, particularly the ones which separate the nose, cheeks and eyes, have become lost in the sea of unimportant boundaries.

Ideally, what a boundary detector should produce is something like the boundaries illustrated in Figure 1-9:

FIGURE 1-9: The Variance Ridge Detector, proposed by this thesis, produces this boundary map from the image in Figure 1-8.

Figure 1-9 above was produced by the **Variance Ridge Detector**, which is one of the primary contributions of this thesis. Notice, the Variance Ridge Detector has strongly detected the important inter-texture boundaries in the image, and suppressed the unimportant intra-texture variations.

More generally, **texture** is a large obstacle to high-quality boundary detection. Texture can be defined as variations in an image that repeat with a pattern (Malik, Belongie, Shi, & Leung, 1999). The mandrill's fur is one example of texture. As was stated previously, boundary detectors work by detecting areas of significant change in the image. By definition, texture introduces change into an image. That means, a boundary detector that does not account for texture can easily confuse changes due to texture as significant changes.

Texture is very common in the real world – almost everything is covered with some form of texture. Obvious examples include grass, trees with leaves, clouds, clothing or the windows on the sides of buildings to name a few. So, for a boundary detector to be useful to computer vision algorithms in practice, it is important that it accounts for texture.

1.3 THE PROBLEM WITH EXISTING TEXTURE-BOUNDARY DETECTORS

A boundary detector that attempts to suppress the variations in texture while detecting boundaries is called a texture-boundary detector. Existing texture-boundary detectors can be divided into two categories – real-time and non-real-time.

Normally, texture analysis is a computationally-intensive operation, and so almost all of the state-of-the-art texture-boundary detectors cannot run in real-time (see chapter 4 for examples of this). This is highly unfortunate, because it means real-time computer vision applications cannot benefit from the state-of-the-art in texture-boundary detection.

On the other hand, the few texture-boundary detectors that are capable of running in real-time produce low-quality results (see chapter 5 for examples of this).

There is a need for a boundary detector that both (a) produces high-quality boundaries and (b) runs in real-time.

1.4 RESEARCH OBJECTIVES

The primary goal of this research is to develop a real-time texture-boundary detector which produces high-quality results. The scope of each part of this goal must be defined.

Real-time

The definition of "real-time" depends very much on the application. The aim of this research is to investigate boundary detectors which could be used for interactive real-

time applications, which means speeds of approximately ten frames per second would be needed. This is similar to how other researchers have defined "real-time." (Kisačanin, Pavlović, & Huang, 2005; Brown & Terzopoulos, 1994; Ranganathan, 2009; Taylor & Cowley, 2009). However, depending on the application, speeds of at least one frame per second could still be considered fast enough for real-time interaction.

This research will focus on achieving real-time speeds with a single CPU – that means it will not investigate how to speed up boundary detection by adding more hardware, but will instead focus on achieving fast boundary detection through faster algorithms.

Texture

Texture will be defined as it is in section 1.2 – variations in an image that repeat with a pattern.

Boundary detector

A boundary detector will be defined as it is at the start of chapter 1: an algorithm which detects the borders that divide different parts of the same image.

Some boundary detectors are required to divide an image into segments (Shi & Malik, 2000; Comaniciu & Meer, 2002) – implying that they have a requirement that all boundaries must form closed loops. This research focuses on boundary detection without the closed loop constraint. Suggestions for achieving the closed-loop constraint will be discussed in the future work section (see section 10.2).

High-quality

Publicly-available image datasets and benchmarks will be used to compare the results of this research against existing work. Both these benchmarks and visual inspection will determine whether the proposed boundary detectors are high-quality.

1.5 THE CONTRIBUTION OF THIS THESIS

This thesis has two primary contributions:

1. The Variance Ridge Detector.
2. The Texton Ridge Detector.

Both of these are high-quality texture-boundary detectors that, unlike most texture-boundary detectors, are able to run in real-time. Each one of these algorithms takes a slightly different approach. The Variance Ridge Detector is faster, while the Texton Ridge Detector produces higher-quality boundaries.

This thesis will present experimental results which have shown that the Variance Ridge Detector and the Texton Ridge Detector both outperform all existing texture-boundary detectors on either speed or quality.

This thesis has also made two secondary contributions:

- A new, fast ridge detection algorithm is proposed. This ridge detection algorithm has been used as part of both the Variance Ridge Detector and the Texton Ridge Detector.
- A new adaptive background modelling algorithm was developed. This algorithm was used to validate the quality of the Variance Ridge Detector and Texton Ridge Detector.

1.6 THESIS OUTLINE

Chapter 2 lays out the fundamental concepts used throughout this thesis by introducing them in the context of edge detection. Originally, the field of boundary detection began as edge detection, and so its basic concepts provide a useful conceptual foundation for the rest of this thesis.

Chapter 3 introduces the mechanics of distinguishing texture from boundaries in the context of edge-preserving smoothing filters. Many texture-boundary detectors, including the proposed Variance Ridge Detector, were built from edge-preserving smoothing filters. Consequently, the techniques used in this chapter will be revisited throughout this thesis.

Chapter 4 examines the field of non-real-time texture-boundary detectors. Most real-time texture-boundary detectors are approximations of non-real-time counterparts. This chapter also demonstrates the complexity of the texture-boundary detection problem, justifying why most texture-boundary detectors cannot run in real-time.

Chapter 5 discusses real-time texture-boundary detectors, highlighting their shortcomings.

Chapter 6 proposes the Variance Ridge Detector to overcome those shortcomings.

Chapter 7 proposes the Texton Ridge Detector, which improves the quality of the Variance Ridge Detector at the cost of speed. It uses textons, which are widely used in non-real-time texture boundary detectors.

Chapter 8 discusses the methods used to evaluate the performance of the proposed boundary detectors.

Chapter 9 presents the experimental results which, as a whole, show that the proposed detectors outperform all previous texture-boundary detectors on either quality or speed.

Finally, chapter 10 concludes the thesis and discusses future directions.

2 EDGE DETECTION

As section 1.2 described, a boundary detector identifies a boundary by identifying a "significant change" in the image. Most of this thesis will discuss texture-boundary detectors, which define "significant change" as a change in texture. This particular chapter however, will discuss edge detectors, which consider any large change in brightness significant. Figure 2-1 illustrates the difference:

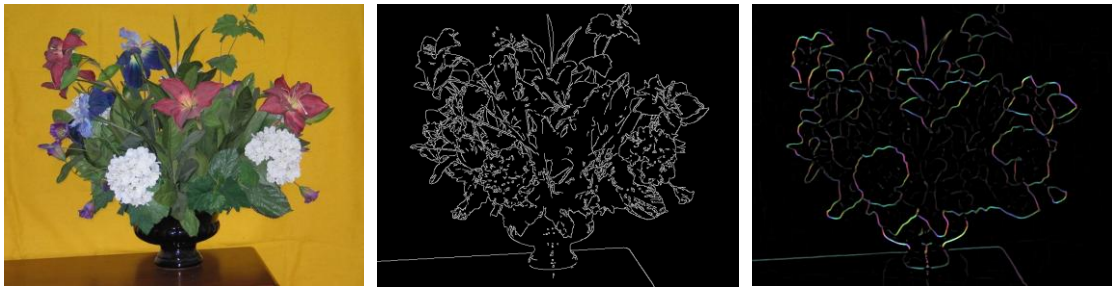


FIGURE 2-1: An image (left), its edge map (middle) and its boundary map (right). The edge map was generated using the Canny edge detector (section 2.4), and the boundary map was generated using the Variance Ridge Detector (proposed in chapter 6).

Essentially, texture-boundary detection is a more constrained and more complex version of edge detection. For that reason, edge detection and texture-boundary detection share much of the same conceptual foundation. The purpose of this chapter is not to go into detail about the classical field of edge detection, but to use edge detection's simple algorithms to describe the fundamental building blocks that will be seen in the more complex texture-boundary detectors. This will include techniques such as: gradients, convolution, sliding windows, thresholding and thinning.

This chapter ends by describing an early approach to texture-suppressing edge detection. This will give an indication of how texture-boundary detectors work – a topic which is developed through the rest of this thesis.

2.1 MATHEMATICAL CONVENTIONS

This thesis will use mathematical symbols to express ideas. This section covers the general mathematical conventions used throughout this thesis.

2.1.1 IMAGES

This thesis will represent images as functions over the spatial domain. For example, the pixel value at position $\boldsymbol{p}$ in image I, with x-coordinate p_x and y-coordinate p_y, would be referred to as follows:

$$I(\boldsymbol{p}) \text{ where } \boldsymbol{p} = \begin{pmatrix} p_x \\ p_y \end{pmatrix} \tag{2.1}$$

Many computer vision algorithms work by calculating each pixel individually. As a convention, the term $\boldsymbol{p}$ will be used to denote the spatial location of the pixel that is currently being calculated. Often, the calculation of $I(\boldsymbol{p})$ is influenced by the values of its neighbours. The symbol $\boldsymbol{p}'$ will denote the spatial offset of the neighbour that is currently being considered. So, the following expression will return the value of a pixel that is offset from $\boldsymbol{p}$ by $\boldsymbol{p}'$:

$$I(\boldsymbol{p} + \boldsymbol{p}') \tag{2.2}$$

2.1.2 VECTORS

Vectors, such as $\boldsymbol{p}$, will be rendered in **bold**.

The L1 and L2 metrics calculate the magnitude of a vector $\boldsymbol{v}$ in different ways:

$$\|\boldsymbol{v}\|_{L1} = \sum_i |v_i| \tag{2.3}$$

$$\|\boldsymbol{v}\| = \|\boldsymbol{v}\|_{L2} = \sqrt{\sum_i v_i^2} \tag{2.4}$$

The L2 metric is used often because it calculates the length of the vector if it existed in a multidimensional space. The L1 metric is the sum of the absolute value of all the elements in a vector, and so has other specific uses.

The dot product of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is defined as follows:

$$\boldsymbol{a} \cdot \boldsymbol{b} = \sum_i a_i b_i \tag{2.5}$$

If $\boldsymbol{a}$ and $\boldsymbol{b}$ are *unit* vectors (their length is 1), then $\boldsymbol{a} \cdot \boldsymbol{b}$ will return 1 if the two vectors are facing the same direction, 0 if they are perpendicular, or -1 if they are facing opposite directions. Essentially, the dot product can be used to calculate how much the two vectors agree in terms of direction.

2.1.3 SETS

Sets have a few special symbols associated with them.

An expression like $\{\boldsymbol{\phi} \in \Phi : c_{\phi}(\boldsymbol{\phi}) = \boldsymbol{c}\}$ is a conditional subset expression. This example generates the subset of all elements $\boldsymbol{\phi}$ in set Φ which match the condition $c_{\phi}(\boldsymbol{\phi}) = \boldsymbol{c}$ stated after the colon.

The cardinality (size) of a set A will be denoted $|A|$.

2.1.4 OPERATORS

Sometimes conditional expressions such as these will be used:

$$[\|\boldsymbol{p}'\|_{L1} \leq k] \tag{2.6}$$

In this expression, the […] notation is a function that evaluates to 1 if the contained expression is true, or evaluates to 0 if the contained expression is false.

The positive bounding operator $[\ldots]^{+}$ constrains values to zero or above:

$$[x]^{+} = \begin{cases} x & \text{when } x > 0 \\ 0 & \text{when } x \leq 0 \end{cases} \tag{2.7}$$

The positive bounding operation is sometimes called “half-wave rectification.”

All other mathematical terminology will be introduced as needed.

2.2 SOBEL OPERATOR

The Sobel operator (Sobel & Feldman, 1973) measures the change in brightness at around a particular pixel. The assumption is, the greater the change, the greater the likelihood of an edge.

It does this by performing **convolution** on the image with two specially-designed **kernels**:

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

FIGURE 2-2: The Sobel kernels.

The left kernel is the horizontal Sobel kernel, which measures the change in the horizontal direction. The right kernel is the vertical Sobel kernel, which measures the change in the vertical direction. The image is convolved with both of these kernels, using the process explained next.

2.2.1 CONVOLUTION

The convolution of image $I(\boldsymbol{p})$ with kernel $K(\boldsymbol{p}')$, denoted $(I * K)(\boldsymbol{p})$, can be defined as follows:

$$(I * K)(\boldsymbol{p}) = \sum_{\boldsymbol{p}'} I(\boldsymbol{p} - \boldsymbol{p}')K(\boldsymbol{p}') \tag{2.8}$$

Equation (2.8) above states that, for each pixel $\boldsymbol{p}$, convolution returns the weighted sum of the pixels in the local neighbourhood. The weights of the weighted sum are determined by the kernel. $K(\boldsymbol{p}') = 0$ for any values of $\boldsymbol{p}'$ which are outside the range of the kernel.

The next subsection illustrates convolution with the Sobel kernel.

2.2.2 APPLYING CONVOLUTION TO SOBEL

Let the horizontal Sobel kernel (shown earlier in section 2.2) be represented by the function $Sobel_x(\boldsymbol{p}')$, and let the vertical Sobel kernel be represented by the function $Sobel_y(\boldsymbol{p}')$. The origin $(0, 0)$ point of both Sobel kernels is the central element.

Given these definitions, the Sobel operator can be used to calculate the image gradient $\nabla I(\boldsymbol{p})$:

$$\begin{aligned} \nabla I(\boldsymbol{p}) &= \begin{bmatrix} \partial I_x(\boldsymbol{p}) \\ \partial I_y(\boldsymbol{p}) \end{bmatrix} \\ \partial I_x(\boldsymbol{p}) &= (I * Sobel_x)(\boldsymbol{p}) \\ \partial I_y(\boldsymbol{p}) &= \big(I * Sobel_y\big)(\boldsymbol{p}) \end{aligned} \tag{2.9}$$

This calculates the two-dimensional **gradient** for each pixel $\boldsymbol{p}$ in the image $I(\boldsymbol{p})$. The word gradient is used because it simply means the rate of change. Other definitions of the gradient $\nabla I(\boldsymbol{p})$ exist, these will be discussed later.

An image and its gradient magnitude according to the Sobel operator are shown below in Figure 2-3:

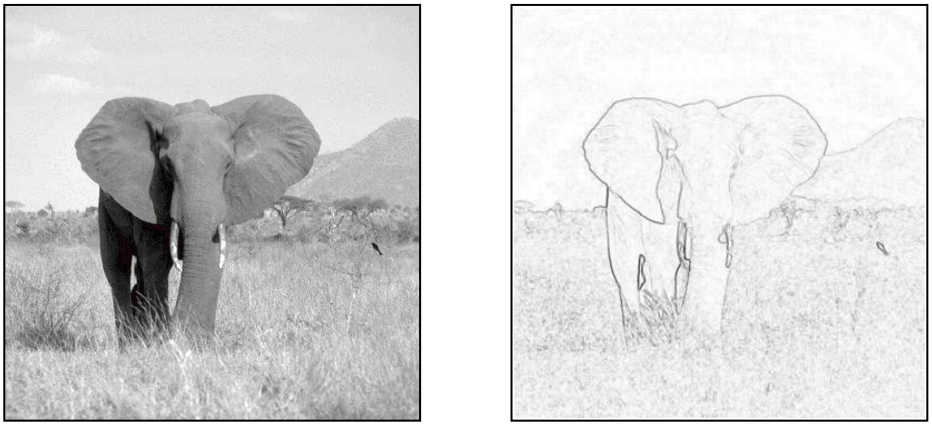


FIGURE 2-3: The result of applying Sobel operator (right) to an image (left).

So why does the Sobel operator calculate the gradient? Consider the horizontal Sobel kernel in Figure 2-2. The left side of the Sobel kernel is entirely negative, and the right side is entirely positive. So when an image is convolved by this kernel, the pixels on the left side are subtracted from the pixels on the right side – effectively measuring the difference between them. Clearly, this estimates the rate of change at each pixel, which is really just another name for the gradient.

2.2.3 SLIDING WINDOWS

It is necessary to define the term sliding window for future reference. When calculating the result for a particular pixel $\boldsymbol{p}$, the term **window** is used to describe the rectangle of pixels that are considered in order to calculate $\boldsymbol{p}$. For example, the Sobel operator would evaluate a 3×3 **window** centered on $\boldsymbol{p}$. Normally, the windows are evaluated starting from the top left pixel $(0, 0)$, then moving one pixel right to $(1, 0)$, and then moving one pixel right to $(2, 0)$ and so on. Due to the way the window is sliding through the image, sometimes this is called a **sliding window**.

2.3 BINARISING SOBEL

Sometimes it is useful for the edge detector to categorise each pixel into one of two states: edge or non-edge. The Sobel operator generates a continuous range of gradient values. Thresholding is a common way to map the range of gradients produced by Sobel onto these two states.

2.3.1 THRESHOLDING

The function $Threshold_I(\boldsymbol{p}; t)$ calculates the binary version of image I, at position $\boldsymbol{p}$, by thresholding it at level t as follows:

$$Threshold_I(\boldsymbol{p}; t) = [I(\boldsymbol{p}) \geq t] \quad \textbf{(2.10)}$$

After thresholding, any pixel which is greater than or equal to t is set to 1, and any pixel less than t is set to 0. The resulting image is called a **binary** image because each one of its pixels can only be in one of two states.

2.3.2 APPLYING THRESHOLDING TO GRADIENTS

The function $IsEdge(\boldsymbol{p}; t)$ below determines whether pixel $\boldsymbol{p}$ is an edge or not by identifying whether the magnitude of its gradient exceeds a threshold t. In this case, the gradient ∇I has been calculated by the Sobel operator, although this would still apply if the gradient was calculated by some other method.

$$IsEdge(\boldsymbol{p}; t) = Threshold_{\|\nabla I\|}(\boldsymbol{p}; t) \quad \textbf{(2.11)}$$

As equation (2.11) shows, the gradient magnitude is calculated by the L2 norm of the gradient $\|\nabla I\|$. Figure 2-4 shows the effect of a threshold operation.

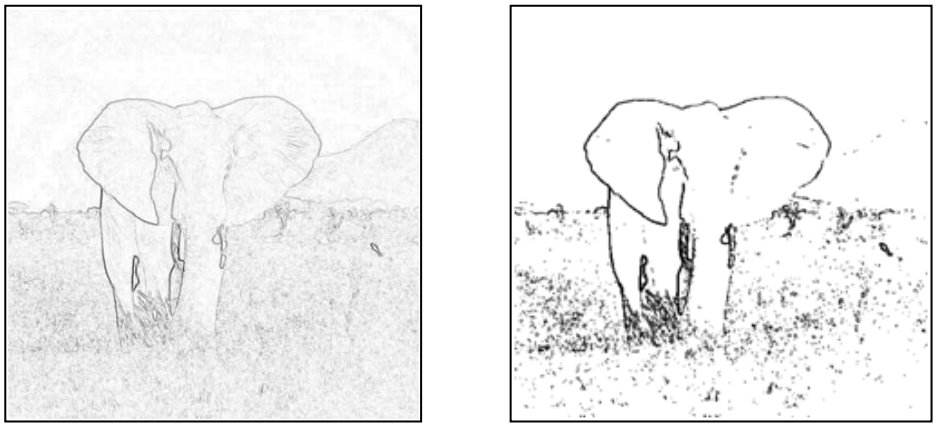


FIGURE 2-4: The unthresholded (left) and thresholded (right) results of the Sobel operator on the elephant image. The threshold was chosen to be the mean gradient magnitude in the image.

The choice of the threshold level t is critical to successful thresholding. If the threshold level is too high, no edges will exceed the threshold. If the threshold level is too low, some non-edges would exceed the threshold. This sensitivity to parameters is the primary drawback of using thresholding.

After thresholding, it is likely that the edges will be many pixels wide, which can make it difficult for an algorithm to pinpoint the exact location of the edge. **Morphological thinning**, described next, can be used to "thin" the edges so that they are always one-pixel wide.

2.3.3 MORPHOLOGICAL THINNING

Let $MorphThin_I(\boldsymbol{p})$ be a function that performs morphological thinning on the binary image I by using successive of morphological operations.

Morphological operations are simple operations that calculate a new value for each pixel based on each pixel's local neighbourhood. The two operations used by morphological thinning are called erosion and hit-and-miss. Both of these will now be explained.

Erosion takes an input image, and considers the 3 by 3 sliding window centered on each pixel. Erosion replaces each pixel with the minimum value in its 3 by 3 neighbourhood. When an image is eroded repeatedly, thick lines will become thinner and thinner, until they disappear completely. So to stop the lines from disappearing once they reach one pixel wide, the hit-and-miss operator is used.

The hit-and-miss operator takes an input image, and attempts to match each pixel's local window to a set of exact-match templates. If there is a match, then that pixel is replaced with a one, otherwise that pixel is replaced with a zero. So for morphological thinning, the hit-and-miss operator is used with templates that contain all possible structures of a one-pixel-wide line, allowing any thinned lines to be detected and preserved.

Using these two operations, the image is repeatedly eroded until the entire image becomes filled with zeroes. Before each erosion, the hit-and-miss operator is run, and the results are accumulated into a separate image. Once this process is complete, the accumulated image will contain only the one-pixel wide lines of all the edges in the image. This accumulated image is returned by the function $MorphThin_I(\boldsymbol{p})$. Figure 2-5 illustrates the effect of morphological thinning:

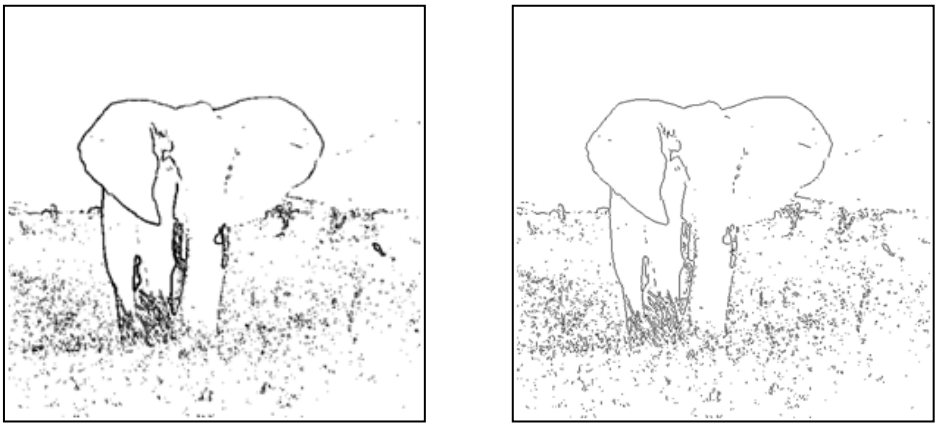


FIGURE 2-5: The unthinned (left) and thinned (right) results of the Sobel operator on the elephant image.

Using morphological thinning, the edges generated by the Sobel operator can be thinned, allowing each edge to be localised.

The Sobel operator runs very fast, but its main problem is it is not robust. It is very sensitive to noise – even a single noisy pixel will show up clearly in a Sobel edge map. Its thresholding stage is also very sensitive and prone to error – it is will normally remove some true edges or include some of the false edges. The next section describes the Canny edge detector, which is much more robust than Sobel.

2.4 CANNY EDGE DETECTOR

Even though it was published in 1986, the Canny edge detector (Canny, 1986) is still widely used today.

2.4.1 GRADIENT ESTIMATION

The Canny edge detector calculates its gradients by convolving the image with the first-derivative of the Gaussian. The first-derivative of Gaussian kernel is a lot like Sobel's kernel, except it considers more pixels and so is more robust to noise.

The Gaussian kernel $G_\sigma(\boldsymbol{p}')$ with scale σ can be defined as follows:

$$\begin{aligned} G_\sigma(\boldsymbol{p}') &= \frac{1}{Z} G_\sigma^Z(\boldsymbol{p}') \\ \text{where } G_\sigma^Z(\boldsymbol{p}') &= e^{\left(-\frac{\|\boldsymbol{p}'\|^2}{2\sigma^2}\right)} \\ \text{and } Z &= \sum_{\boldsymbol{p}'} G_\sigma^Z(\boldsymbol{p}') \end{aligned} \tag{2.12}$$

Conventionally, the symbol Z will be used as a normalisation divisor throughout this thesis, as it has been used here.

Taking the derivative of $G_\sigma(\boldsymbol{p}')$ in the direction $\boldsymbol{d}$ yields the Gaussian derivative kernel:

$$G'_{\sigma,\boldsymbol{d}}(\boldsymbol{p}') = \frac{-\boldsymbol{p}' \cdot \boldsymbol{d}}{\sigma^2} G_\sigma(\boldsymbol{p}'; \sigma) \tag{2.13}$$

The following figure illustrates what the Gaussian derivative kernel looks like:

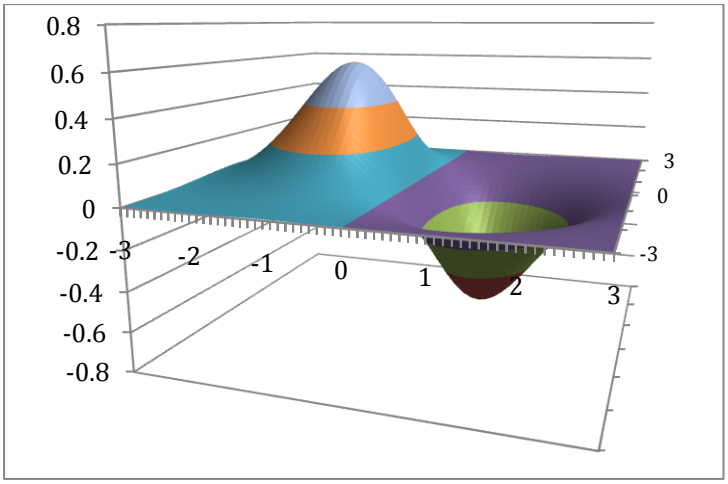


FIGURE 2-6: The Gaussian derivative function, for $\boldsymbol{\sigma} = \mathbf{1}$ and $\boldsymbol{d} = \begin{bmatrix}\mathbf{1}\\ \mathbf{0}\end{bmatrix}$.

From Figure 2-6, notice that on one side, the Gaussian-derivative kernel is positive, and on the other side it is negative. This is the same structure as the Sobel kernel, which is why the Gaussian-derivative kernel can be used to estimate gradient. Also notice that the Gaussian-derivative kernel is **oriented** and is not **isotropic** (the same in all directions). The parameter $\boldsymbol{d}$ in the Gaussian-derivative kernel function $G'_{\sigma,\boldsymbol{d}}(x)$ determines the orientation of the kernel. Finally, notice that at the limits of the graph, where x and y approach ± 3, the Gaussian-derivative kernel almost reaches zero. This is intuitive because the standard deviation σ was set to 1, and it is a well-known fact that the Gaussian function almost reaches zero at 3 standard deviations away from the mean.

Gradients can be calculated using the Gaussian derivative in the same way as with the Sobel kernel in equation (2.9):

$$\begin{aligned}\nabla I(\boldsymbol{p}) &= \begin{bmatrix}\partial I_x(\boldsymbol{p})\\ \partial I_y(\boldsymbol{p})\end{bmatrix}\\ \partial I_x(\boldsymbol{p}) &= \left(I * G'_{\sigma,\boldsymbol{d}_x}\right)(\boldsymbol{p}) \text{ where } \boldsymbol{d}_x = \begin{bmatrix}1\\0\end{bmatrix}\\ \partial I_y(\boldsymbol{p}) &= \left(I * G'_{\sigma,\boldsymbol{d}_y}\right)(\boldsymbol{p}) \text{ where } \boldsymbol{d}_y = \begin{bmatrix}0\\1\end{bmatrix}\end{aligned} \qquad \textbf{(2.14)}$$

With Sobel, the gradient was postprocessed with a threshold and then thinning. The next steps of the Canny edge detector are ridge detection and hysteresis. Ridge detection is like a generalisation of thinning, and hysteresis is an improvement to normal thresholding. So in effect, the Canny edge detector improves on the same process.

2.4.2 RIDGE DETECTION

The Canny edge detector uses a specially-designed ridge detection method to ensure it only detects one-pixel-wide edges.

First, the gradient orientation $\nabla I_\theta(\boldsymbol{p})$ is calculated:

$$\nabla I_\theta(\boldsymbol{p}) = \text{atan}\frac{\partial I_y(\boldsymbol{p})}{\partial I_x(\boldsymbol{p})} \tag{2.15}$$

The Canny edge detector performs ridge detection by doing the following two steps for each pixel $\boldsymbol{p}$. First, it rounds the gradient orientation $\nabla I_\theta(\boldsymbol{p})$ to the nearest multiple of 45°. Second, it checks the two gradients on either side of pixel $\boldsymbol{p}$, where "either side" depends on the orientation (see Figure 2-7 below). The pixel is only a ridge if it has a stronger gradient than the other two pixels:

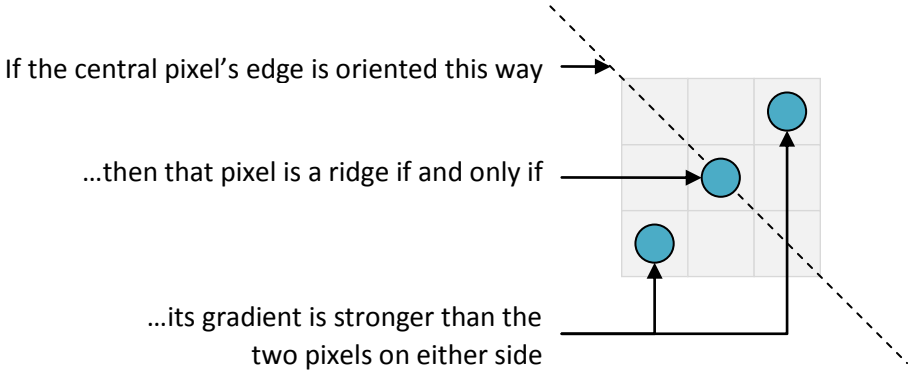


FIGURE 2-7: The Canny edge detector's ridge detection stage.

If a pixel is not a ridge, then its gradient is set to zero. This allows the Canny edge detector to produce one-pixel-wide edges very fast.

2.4.3 HYSTERESIS

The purpose of the hysteresis stage is to classify each pixel as either edge or non-edge. To do this, the Canny edge detector first thresholds the gradient image with a high threshold. The high threshold ensures that only the most likely edges are kept. The trouble is, a high threshold will eliminate some of the weaker edges as well. To solve this, the Canny edge detector uses thresholding with hysteresis.

Hysteresis means the Canny edge detector traces each of the edges that was detected with the high threshold, searching for connected sections of the edge curves which were too weak to pass the high threshold. Any gradients which are both, (a) connected to a known edge and (b) stronger than another, lower threshold, will be recovered by this process.

The result of this is a binary edge map which can be used as the starting point for many other applications. Figure 2-8 below shows the results of the Canny edge detector:

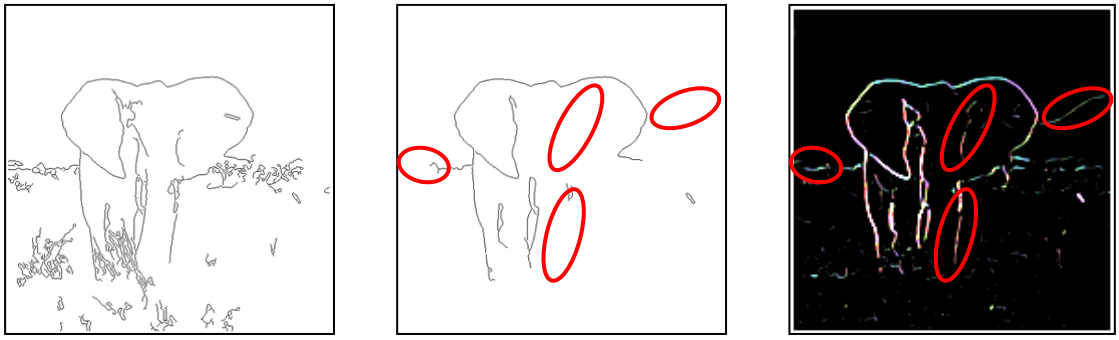


FIGURE 2-8: The result of the Canny edge detector on the elephant image. The left image has its high threshold set to 0.25, while the middle image has its high threshold set to 0.5. The low threshold has been set to one quarter of the high threshold in both cases. The image on the right is the Variance Ridge Detector, which will be proposed by this thesis later on, shown for comparison. The red circles indicate the boundaries which have been lost by the Canny, yet have been preserved by the Variance Ridge Detector.

Canny's approach to edge detection is much more robust than Sobel's approach for a couple of reasons. Its gradient stage integrates information from a much larger window, making it less sensitive to noise, and its hysteresis stage attempts to recover edges that are normally lost by thresholding.

Even with these improvements though, texture is still a problem for the Canny edge detector. Notice in Figure 2-8 above that Canny's approach cannot suppress the intra-texture edges without suppressing some of the inter-texture boundaries. This is because it does not explicitly account for texture as part of its algorithm. The result of the Variance Ridge Detector, which will be proposed later by this thesis, is also shown in Figure 2-8 to highlight this. The next section describes an algorithm which makes some attempt to overcome this problem.

2.5 EDGE DETECTION WITH VARIANCE THRESHOLDING

One of the earliest forays into intra-texture-edge suppression was developed by Ahmad and Choi (1999). Their edge detector was divided into two stages. The first stage was just traditional edge detection. The way this was done is not important, in essence it was not much different from Sobel or Canny. The second stage is the most interesting stage. The second stage would only preserve the edges that had high variance in their local sliding window. All other edges would be suppressed. Figure 2-9 presents some results of this algorithm.

FIGURE 2-9: An image (left), its edge map (middle) and its edge map with variance thresholding applied (right). Notice some intra-texture edges have been suppressed. Reproduced from Ahmed and Choi (1999).

This algorithm works because the variance between two different textures is normally much greater than the variance within a texture. Therefore, variance can be used to suppress intra-texture edges. Although Ahmad and Choi's work uses this technique to some level of success, Figure 2-9 clearly shows that their resulting edge maps are quite noisy, and so there is still much room for improvement. Later, this thesis will propose the Variance Ridge Detector (in chapter 6), which expands on some of Ahmad and Choi's ideas.

2.6 CHAPTER SUMMARY

This chapter presented three edge detectors – the Sobel operator, the Canny edge detector, and Ahmad and Choi's detector.

The Sobel operator suffers from the problem of being sensitive to noise because of its small sliding window and sensitive thresholding stage. The Canny edge detector overcomes this with its larger Gaussian-weighted sliding window and its hysteresis stage, but it cannot distinguish between intra-texture edges and inter-texture edges. Unlike Canny's approach, Ahmad and Choi's detector makes some distinction between intra-texture edges and inter-texture edges. Unfortunately, its results suffer from being quite noisy. So there is still a need for a detector that both suppresses noise and can distinguish between intra-texture and inter-texture boundaries. Chapter 4 will present some techniques for achieving this, but first, the next chapter will discuss some of the mechanisms required by those techniques in the context of edge-preserving smoothing filters.

3 EDGE-PRESERVING SMOOTHING FILTERS

The problem with normal methods of smoothing (such as Gaussian smoothing) is that they will normally smooth out both salient image features as well as noise. An edge-preserving smoothing filter attempts to solve this problem by preserving the edges while smoothing other areas. Some of the more recently developed edge-preserving smoothing filters can do even better than this – they can smooth out texture as well as noise while maintaining texture boundaries.

Intuitively, an edge-preserving smoothing filter must know, at least implicitly, where the edges or texture boundaries are, and so many of the mechanisms behind boundary detection originally came from edge-preserving smoothing. This chapter will introduce these mechanisms in the context of edge-preserving smoothing, and the chapters following will expand these mechanisms into texture-boundary detectors.

This chapter will first briefly explain the motivation for edge-preserving smoothing by illustrating the problems with non-edge-preserving smoothing. Then, five edge-preserving smoothing filters will be discussed: the bilateral filter, the Nitzberg operator, the Kuwahara filter, the Papari filter and the mean-shift filter.

3.1 NON-EDGE-PRESERVING SMOOTHING

All smoothing methods involve some type of weighted average.

The simplest and fastest method of smoothing is to weight all elements of the sliding window equally. This is often known as a box blur. Let $BoxBlur_I(\boldsymbol{p}; r)$ be the box blur of image I at pixel position $\boldsymbol{p}$, where r is the sliding window radius:

$$BoxBlur_I(\boldsymbol{p}; r) = \frac{1}{Z}\sum_{\boldsymbol{p'}} I(\boldsymbol{p} + \boldsymbol{p'})[\|\boldsymbol{p'}\|_{L1} \leq r] \quad \text{where } Z = (2r+1)^2 \tag{3.1}$$

The Gaussian blur generates a more natural looking smooth as the sliding window weights are determined by the Gaussian function:

$$GaussianBlur_I(\boldsymbol{p}; \sigma) = (I * G_\sigma)(\boldsymbol{p}) \tag{3.2}$$

The Gaussian kernel $G_\sigma(\dots)$ in the expression above was already defined previously in equation (2.12). The parameter σ determines the scale of the Gaussian blur. Figure 3-1 below illustrates these two different methods of smoothing.

FIGURE 3-1: An image (left), its box blur (middle) and its Gaussian blur (right). Both of these forms of blurring will smooth important image features as well as unimportant ones.

Notice in Figure 3-1 above that both these smoothing methods have smoothed out the useful edges. That is why edge-preserving smoothing is needed.

Edge-preserving smoothing generally also uses a weighted average like the smoothing methods above. The difference is, the weighting given to each pixel is modified for each pixel according to some function. The next section introduces this concept with the simplest possible edge-preserving weighting function.

3.2 BILATERAL FILTER

Typically, pixels that are separated by a boundary tend to be different from each other. So, if the weighted average applies more weight to more similar pixels, then it is unlikely that the image will be smoothed across boundaries. The bilateral filter (Tomasi & Manduchi, 1998) is built on this concept. Let $BilateralFilter_I(\boldsymbol{p})$ be a function that evaluates the bilateral filter of image I at pixel position $\boldsymbol{p}$.

$$\begin{aligned} BilateralFilter_I(\boldsymbol{p}) &= \frac{1}{Z(\boldsymbol{p})} \sum_{\boldsymbol{p}'} I(\boldsymbol{p}+\boldsymbol{p}')\, w(\boldsymbol{p},\boldsymbol{p}') \\ w(\boldsymbol{p},\boldsymbol{p}') &= d_s(\boldsymbol{p}') + d_c\big(I(\boldsymbol{p}) - I(\boldsymbol{p}+\boldsymbol{p}')\big) \\ Z(\boldsymbol{p}) &= \sum_{\boldsymbol{p}'} w(\boldsymbol{p},\boldsymbol{p}') \end{aligned} \tag{3.3}$$

The weighting function $w(\dots)$ above calculates the weight by combining some function of the spatial distance $d_s(\dots)$ and some function of the chromatic distance $d_c(\dots)$. Intuitively, these distance functions are designed to give higher weight to those pixels that are more similar to the pixel being smoothed. See Tomasi and Manduchi's paper (1998) for example definitions of $d_s(\dots)$ and $d_c(\dots)$.

FIGURE 3-2: The bilateral filter (right) applied to an image (left).

In Figure 3-2 above, the bilateral filter has smoothed out some of the texture, such as the grass and the elephant's skin. At the same time, the boundaries between the sky, the elephant, and the grass, remain very sharp. This demonstrates the usefulness of the bilateral filter.

Due to this texture-suppressing ability, sometimes the bilateral filter is used to reduce the texture in an image before applying the Canny edge detector or Sobel operator (Kiranyaz, Ferreira, & Gabbouj, 2008). This helps to reduce the number of detected edges in textured regions. Unfortunately, the bilateral filter must evaluate a large number of pixel pairs and so this is not a real-time solution.

The bilateral filter cannot smooth out all types of texture. Particularly, it does not work well when textures have both large variations and a large wavelength. This is illustrated in Figure 3-2 by the fact that the wrinkles on the elephant's trunk and ears have not been smoothed away. This problem occurs because the bilateral filter only considers each pair of pixels in isolation – it does not analyse the influence of all pixels in the sliding window as a whole. The Nitzberg operator, which will be described next, attempts to overcome this problem.

3.3 NITZBERG OPERATOR

The gradient is an obvious choice for edge-preserving smoothing, because as shown in chapter 2, the gradient can be used to detect edges. The Nitzberg operator (Nitzberg & Shiota, 1992) reshapes and displaces a Gaussian smoothing kernel so that it avoids smoothing the local gradients. The Nitzberg operator is important because it was later redeveloped into a real-time texture-boundary detector called Konishi's detector (section 5.1).

The next few sections will discuss kernel reshaping and kernel displacement.

3.3.1 KERNEL DISPLACEMENT

Since gradients indicate likely positions of boundaries, when near gradients, the Nitzberg operator shifts the smoothing window so that it can avoid smoothing those gradients.

The kernel displacement vector $Displacement(\boldsymbol{p})$ for pixel $\boldsymbol{p}$ is calculated as follows:

$$Displacement(\boldsymbol{p}) = \sum_{\boldsymbol{p}'} \nabla I(\boldsymbol{p}+\boldsymbol{p}') \times FlipInwards(\boldsymbol{p}';\boldsymbol{p}) \times G_\sigma(\boldsymbol{p}')$$
$$FlipInwards(\boldsymbol{p}';\boldsymbol{p}) = \begin{cases} -1 & \nabla I(\boldsymbol{p}+\boldsymbol{p}') \cdot \boldsymbol{p}' > 0 \\ 1 & \text{otherwise} \end{cases} \quad \textbf{(3.4)}$$

As equation (3.4) states, to calculate the displacement for a particular pixel $\boldsymbol{p}$, the Nitzberg operator considers all the gradients in the vicinity of $\boldsymbol{p}$. Each gradient votes to push the kernel to one side of the boundary orientation it represents. If a boundary were vertically oriented, this would mean the kernel could be pushed either left or right to avoid smoothing that boundary. The Nitzberg operator always chooses to push the kernel away from the boundary and towards the pixel under consideration $\boldsymbol{p}$ using the $FlipInwards(\boldsymbol{p}';\boldsymbol{p})$ coefficient. If this was not done, the pixels on the right side could take on values from the left side of the boundary, and vice versa. This would mix the pixels on either side of the boundary and so would smooth out the boundary instead of preserving it.

To allow the Nitzberg operator to avoid smoothing boundaries even better, the kernel is reshaped as well as displaced.

3.3.2 KERNEL RESHAPING

When it is near gradients, the Nitzberg operator reshapes the Gaussian kernel from its normal, circular shape into a more elliptical one so that it can avoid smoothing those gradients. The elliptical shape is constructed so that its major axis runs parallel to the average gradient orientation. This allows it to avoid the gradient as much as possible, as illustrated in Figure 3-3.

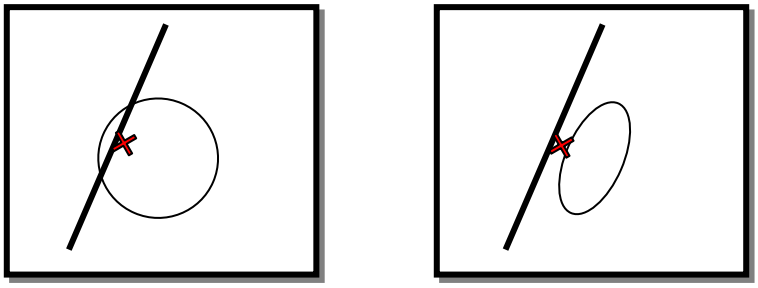


FIGURE 3-3: In the left image, there is a gradient (strong line) and a kernel (circle) which has been displaced from the current pixel (cross). Notice that the kernel is smoothing over part of the boundary. The right image is the same, except the kernel has been reshaped into an ellipse. Notice that the kernel now avoids smoothing the boundary. This illustrates the purpose of kernel reshaping.

The kernel reshaping is done using a technique called structure tensors.

3.3.2.1 STRUCTURE TENSORS

A structure tensor is a way of representing orientations (Bigun & Granlund, 1987; Knutsson, 1989). The structure tensor of a vector $\binom{x}{y}$ can be calculated by the function $\varphi\binom{x}{y}$:

$$\varphi\binom{x}{y} = \begin{bmatrix} x^2 & xy \\ xy & y^2 \end{bmatrix} \tag{3.5}$$

The key characteristic of a structure tensor is that it wraps around at 180°. That means, two vectors with opposite directions will generate the same structure tensor:

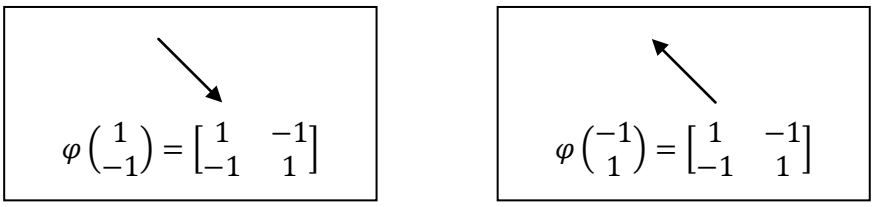


FIGURE 3-4: Structure tensors wrap around at 180°, which is why they are so useful.

An edge oriented at 0° is the same as an edge oriented at 180°, and so structure tensors provide a way to represent orientations like this in a mathematically sound manner.

Structure tensors as transformations

A structure tensor is actually a linear transformation matrix. That means that, when a point is multiplied by a structure tensor, that point will be transformed in a particularly useful way. Under the right conditions, the transformation will reveal what orientation the structure tensor represents.

Given a structure tensor that represents an unknown orientation, the best way to visually identify its orientation is to use the structure tensor to transform the points on a circle centered on the origin:

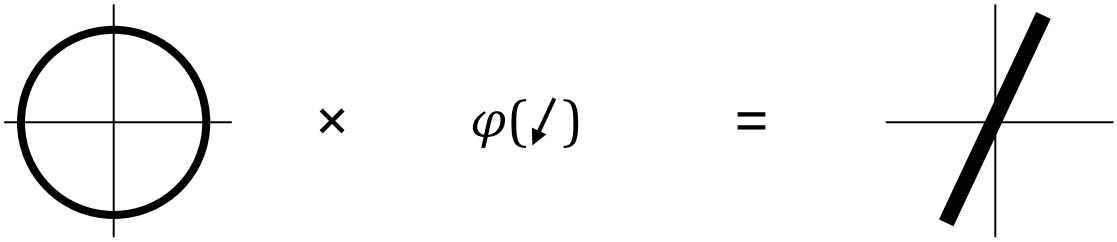


FIGURE 3-5: The circle on the left is transformed with the structure tensor of the vector specified in the brackets, producing the line on the right.

If a structure tensor represents one orientation, then its transformation will project all points onto a line of that orientation. Figure 3-5 shows how all the points on the circle have been projected (squashed) onto a single line. Most importantly though, the orientation of the projected line reveals the orientation represented by the structure tensor.

Any number of structure tensors can be added or averaged together, and the resulting structure tensor will represent an average of all the orientations contained within the input structure tensors.

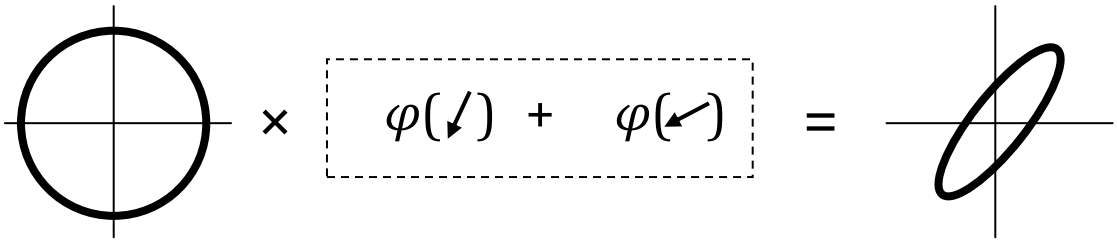


FIGURE 3-6: The circle on the left is transformed with the sum of two structure tensors, producing the ellipse on the right.

Figure 3-6 shows another example transformation with the sum of two structure tensors made from two different orientations. Notice that the circle is transformed into an ellipse, and that the major axis of the ellipse matches the average orientation of two orientations. As a rule, a structure tensor will always transform a circle into an ellipse, and the major axis of the ellipse will always be oriented according to the average orientation.

The minor axis of the ellipse will have different widths depending on the input. Figure 3-5 is simply a special case where the minor axis had zero-width, unlike Figure 3-6. The width of the minor axis holds some useful information.

The width of the minor axis

The width of the minor axis reveals how **coherent** the average orientation is. If a structure tensor is averaged from many input orientations that are quite similar, the ellipse will have a narrow minor axis. In fact, if all input orientations agree perfectly, meaning if they are all exactly the same, then the ellipse will have a zero-width minor axis, as illustrated in Figure 3-5. Conversely, if the input orientations are highly dissimilar, the ellipse will have a wide minor axis. In fact, if the input orientations all maximally dissimilar – for example, if two perpendicular orientations were input – then the major and minor axis will be equivalent in length. In other words, there will be no major or minor axis, because ellipse will actually be a circle. This makes sense, because if the input orientations are maximally dissimilar, then there is no average orientation.

Coherence

The ratio of the major axis to the minor axis indicates the level of **coherence** – how much the input orientations agree. This ratio is very important and has many applications. Konishi's detector uses this for boundary detection (section 5.1), and it is also used to detect corners (Harris & Stephens, 1988). Coherence will be discussed in more detail later.

3.3.2.2 KERNEL RESHAPING WITH STRUCTURE TENSORS

To know how the kernel needs to be reshaped, the Nitzberg operator calculates the local gradient orientation $GradientOrientation(\boldsymbol{p})$ at pixel $\boldsymbol{p}$ as a structure tensor:

$$GradientOrientation(\boldsymbol{p}) = GaussianBlur_{\varphi\nabla I}(\boldsymbol{p};\sigma)$$
$$\varphi\nabla I(\boldsymbol{p}) = \varphi\big(\nabla I(\boldsymbol{p})\big) \qquad \textbf{(3.6)}$$

The function $GaussianBlur_I(\boldsymbol{p};\sigma)$ was defined for an image I in equation (3.2). It is used here to average the gradient orientations surrounding each image position $\boldsymbol{p}$. Using this, the Gaussian kernel is reshaped:

$$ReshapedGaussian(\boldsymbol{p}';\boldsymbol{p}) = G_\sigma(GradientOrientation(\boldsymbol{p}) \times \boldsymbol{p}') \qquad \textbf{(3.7)}$$

Basically, equation (3.7) states that the normally circular Gaussian kernel is reshaped into an ellipse depending on the local gradient orientations. This is exactly the same as the circles being transformed into ellipses in the previous section.

In a sliding window where no boundary is present, the gradients will generally be randomly oriented, and so the Gaussian will approximately retain its normal circular shape. However, if a boundary is present, generally all the gradients in the local area will conform to a particular orientation, causing the circular Gaussian kernel to be

transformed into an ellipsoidal shape like in Figure 3-6. The major axis of the ellipsoidal shape will match the orientation of the local gradients, which, with displacement, allows the smoothing kernel to better avoid boundaries.

It turns out that the $GradientOrientation(\boldsymbol{p})$ structure tensor has some level of texture-suppressing ability. Konishi's detector, which will be described later in this thesis (section 5.1), takes advantage of this.

3.3.3 COMBINING RESHAPING AND DISPLACEMENT

Together, the Nitzberg operator works as follows:

$$\begin{aligned} Nitzberg(\boldsymbol{p}') &= \sum_{\boldsymbol{p}'} I\big(\boldsymbol{p} - \boldsymbol{p}' + D(\boldsymbol{p})\big) \times RG(\boldsymbol{p}'; \boldsymbol{p}) \\ \text{where } D(\boldsymbol{p}) &= Displacement(\boldsymbol{p}) \\ \text{and } RG(\boldsymbol{p}'; \boldsymbol{p}) &= ReshapedGaussian(\boldsymbol{p}'; \boldsymbol{p}) \end{aligned} \tag{3.8}$$

Notice that the Nitzberg operator has a similar form to the standard convolution equation (2.8), except the kernel that is used is the reshaped gaussian, and the displacement term has been added.

$Nitzberg(\boldsymbol{p})$ produces the final result of the Nitzberg operator.

FIGURE 3-7: The Nitzberg operator (right), applied to an image (left). These images are greyscale, but it is possible to use the Nitzberg operator on colour images as well.

Unlike the bilateral filter, the Nitzberg operator considers the entire sliding window as a whole when deciding how an area should be smoothed, which is why it is better at smoothing out texture, as shown in Figure 3-7. However, it has also smoothed out some of the texture-boundaries slightly. This occurs because the intra-texture gradients introduce distractions to the displacement process, sometimes causing it to smooth over boundaries. The Kuwahara filter, introduced next, is able to keep all boundaries sharp, unlike the Nitzberg operator.

3.4 KUWAHARA FILTER

The Kuwahara filter (Kuwahara, Hachimura, Ehiu, & Kinoshita, 1976) uses variance to detect and avoid boundaries. Many filters have been developed based upon its original concept. In particular, the Papari filter (described next in section 3.5) was developed based on Kuwahara *et al.*'s work. In turn, the Variance Ridge Detector, which will be proposed by this thesis, was developed based on the Papari filter. This makes the Kuwahara filter quite important.

The Kuwahara kernel has this structure:

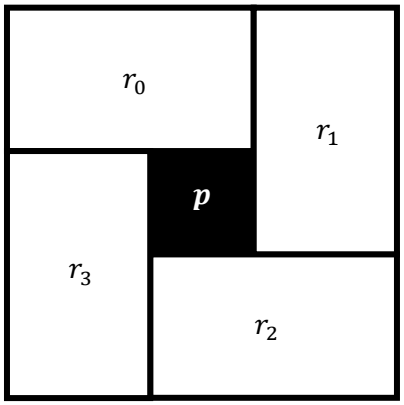


FIGURE 3-8: The Kuwahara kernel.

The Kuwahara filter considers each pixel $\boldsymbol{p}$ individually. The kernel center is first placed on pixel $\boldsymbol{p}$. Then the filter calculates the total variance in each of the neighbouring regions r_0, r_1, r_2 and r_4 (see Figure 3-8), and out of these four regions, it selects the region r^* that has the lowest variance. The output for pixel $\boldsymbol{p}$ is the average colour in region r^*.

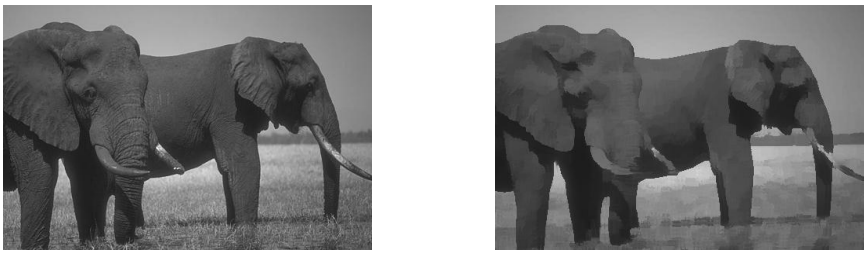


FIGURE 3-9: The Kuwahara filter (right) applied to an image (left). These images are greyscale, but it is possible to use the Kuwahara filter on colour images too.

The reason the Kuwahara filter works is that, an edge or boundary by definition will introduce variance into the image. By intentionally selecting the region of lowest variance, the Kuwahara filter avoids smoothing over boundaries.

Figure 3-9 shows that the Kuwahara has left the boundaries sharp, unlike the Nitzberg operator. However, it has also introduced smoothing artefacts throughout the image,

due to its hard-selection process. This can be seen by the way the image now appears to be made of many coloured patches. This occurs because the Kuwahara filter hard-assigns each pixel into one of the four regions of its kernel, and so when neighbouring pixels are assigned to different regions, a sharp change is introduced. The Papari filter, which will be introduced next, tries to avoid this by using soft-assignment.

3.5 PAPARI FILTER

The Papari filter (Papari, Petkov, & Campisi, 2007) is similar to the Kuwahara filter, except for two major differences. First, its kernel is circular:

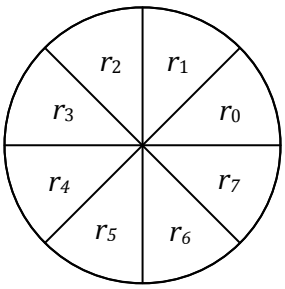


FIGURE 3-10: Papari's circular kernel.

Kuwahara's square-shaped kernel has the tendency to distort shapes, which is not useful for some computer vision applications such as object recognition for example. Papari *et al.*'s circular kernel is isotropic, meaning it will not distort shapes.

Secondly, instead of hard-assigning each pixel to one region only, the Papari filter takes a weighted average of all regions. The regions with higher variance are given smaller weights, which stops the filter from smoothing over boundaries. If no boundaries are present and variance is approximately the same in all directions, all regions will be given approximately equal weight. This means no artefacts will be introduced from an arbitrary hard-assignment of a pixel to one particular region.

3.5.1 FORMULATION

Let the Papari kernel have N_s sectors, where N_s is a user-defined parameter. Let the function $S(\boldsymbol{p}', s)$ return 1 if the vector $\boldsymbol{p}'$ belongs to the angle owned by sector s.

$$S(\boldsymbol{p}', s) = \begin{cases} 1 & s\theta - \frac{\theta}{2} \leq \operatorname{atan}\frac{p'_y}{p'_x} < s\theta + \frac{\theta}{2} \\ 0 & \text{otherwise} \end{cases}$$
$$\text{where } \theta = \frac{360°}{N_s} \tag{3.9}$$

Papari *et al*. chose to use Gaussian weights for their filter, and so each sector s has its own slice $G_s(\boldsymbol{p}')$ of the Gaussian kernel $G_\sigma(\boldsymbol{p}')$:

$$G_s(\boldsymbol{p}') = G_\sigma(\boldsymbol{p}') \times S(\boldsymbol{p}', s) \tag{3.10}$$

Using $G_s(\boldsymbol{p}')$, the mean $\mu_s(\boldsymbol{p})$ and variance $\sigma_s(\boldsymbol{p})^2$ for each sector s can be calculated at each pixel $\boldsymbol{p}$ as follows:

$$\begin{aligned} \mu_s(\boldsymbol{p}) &= (I * G_s)(\boldsymbol{p}) \\ \sigma_s(\boldsymbol{p})^2 &= (I^2 * G_s)(\boldsymbol{p}) - \mu_s(\boldsymbol{p})^2 \end{aligned} \tag{3.11}$$

For each pixel $\boldsymbol{p}$, the weighting $w_s(\boldsymbol{p})$ of each sector s is inversely related to the variance of that sector:

$$w_s(\boldsymbol{p}) = \frac{1}{\sigma_s(\boldsymbol{p})^q} \tag{3.12}$$

The value q is an external parameter, set by the user. The larger the value of q, the more the sectors of large variance are avoided. Using the weights, the Papari filter can be defined as the weighted average of the sectors:

$$\begin{aligned} PapariFilter(\boldsymbol{p}) &= \frac{1}{Z(\boldsymbol{p})}\sum_{s}^{N_s} \mu_s(\boldsymbol{p}) \times w_s(\boldsymbol{p}) \\ Z(\boldsymbol{p}) &= \sum_{s}^{N} w_s(\boldsymbol{p}) \end{aligned} \tag{3.13}$$

3.5.2 IMAGE RESULTS

Figure 3-11 below illustrates some results of the Papari filter:

FIGURE 3-11: The Papari filter (right) run on a textured image (left). Reproduced from Papari *et al.* (2009)

Unlike the Kuwahara filter, the Papari filter does not introduce artefacts into the image because it does not hard-assign pixels to single regions. Figure 3-11 illustrates this. Also notice that the Papari filter is able to smooth out all the grass and water textures while clearly retaining strong inter-texture boundaries.

The Papari filter can distinguish between texture boundaries and texture because, within the same texture, variance tends to be approximately equal. This will cause all the sectors of the Papari filter to have the same weight, and so the texture will be smoothed. When an inter-texture boundary is present, the boundary will introduce high variance into some of the sectors, causing them to be excluded from the smoothing. The Variance Ridge Detector, which is proposed later in this thesis, functions based on a similar mechanism to this.

Despite the excellent results of the Papari filter, its results can still be improved on some textures. For example, in Figure 3-11, the tree leaf textures have not been smoothed away completely. This occurs because the smoothing kernel is not large enough in this case. A larger smoothing kernel can handle textures of larger wavelengths, but the problem is, it will make the boundaries coarser and less accurate. The mean-shift filter, introduced next, can smooth out larger scale texture without the boundaries becoming coarser and less accurate.

3.6 MEAN-SHIFT FILTER

Like the bilateral filter, the mean-shift filter (Comaniciu & Meer, 2002) uses similarity as part of its edge-preserving smoothing process.

The mean-shift filter is important because it has been developed into a texture-boundary detector called mean-shift segmentation (section 3). In turn, mean-shift segmentation lays down the concept of how the real-time texture-boundary detector

called Randomised Hashing (later in section 5.5) works. On top of all this, the mean-shift filter will be used to introduce the concept of clustering – an important concept which powers most texture boundaries today.

Mean-shift clustering

Clustering is simply grouping similar observations together. The mean-shift filter works by grouping pixels that are similar (both chromatically and spatially), and then replacing each pixel's value with the average of its cluster. The mean-shift filter uses mean-shift clustering to achieve this.

In mean-shift clustering, each pixel becomes a point in five-dimensional ($\mathbb{R}^5$) space. Those five dimensions are made up of two dimensions for the spatial coordinates (x and y), and three dimensions for the colour coordinates (for example, red, green and blue).

An imaginative way to understand mean-shift clustering is to imagine that each of the $\mathbb{R}^5$ points have gravity. Over time, the points will fall towards each other, and eventually clouds of points will collapse to a singularity – that is when an entire cloud has collapsed to occupy the same position in space. Fortunately, images will normally consist of multiple point clouds which will each collapse to separate singularities. Each one of these point clouds is a natural cluster of the data. Once the mean-shift clustering process has run, the clusters can be found by identifying the singularities that have formed.

Using clusters for smoothing

Now that the clusters of pixels have been found, each pixel is simply replaced with the average colour of its cluster. Figure 3-12 shows an example of what this looks like:

FIGURE 3-12: The mean-shift filter (right) run on the mandrill image (left). Images from Comaniciu and Meer (2002).

One of the biggest difficulties about the mandrill image in Figure 3-12 is that the eye regions are small in comparison to the nose, cheek and fur regions. If the Papari filter

were to be used, this means its scale would have to be set small enough so that the eyes would not be smoothed out, which in turn limits the amount the other, larger-scale textures can be smoothed. Unlike the Papari filter, the mean-shift filter can function well on images where the regions are not similarly-sized, as illustrated in Figure 3-12.

The mean-shift filter's excellent ability to both smooth unequal-sized texture regions while preserving texture boundaries makes it one of the best edge-preserving smoothing filters. Its mechanism has been developed into a texture-boundary detector called mean-shift segmentation.

3.7 CHAPTER SUMMARY

This chapter has investigated five edge-preserving filters, each of which have some relationship to texture-boundary detection.

Section 3.2 introduced the bilateral filter, which is sometimes used with the Canny edge detector for non-real-time texture-boundary detection.

Section 3.3 introduced the Nitzberg operator, which is the core of Konishi's detector, discussed later in section 5.1.

Section 3.4 introduced the Kuwahara filter, which was built on by the Papari filter in section 3.5, which in turn was the inspiration for the Variance Ridge Detector in chapter 6.

Section 3.6 introduced the mean-shift filter, which is the core of mean-shift segmentation (section 4.2) which in turn has inspired boundary detection via Randomised Hashing (section 5.5).

Other edge-preserving smoothing filters exist which have not been discussed as they are not related to texture-boundary detection. In particular, Perona and Malik's (1990) anisotropic diffusion has not been discussed here.

The next two chapters will investigate how these techniques, and others, have been used in the field of texture-boundary detection.

4 NON-REAL-TIME TEXTURE-BOUNDARY DETECTION

Almost all texture-boundary detectors cannot run in real-time because boundary detection is far too complex a problem. The few real-time texture-boundary detectors that exist are mostly approximations of non-real-time algorithms. The purpose of this chapter is to outline the non-real-time algorithms which will be approximated in the next few chapters. Five non-real-time texture-boundary detectors will be introduced: normalised cut segmentation, mean-shift segmentation, TextonBoost, the probability of boundary (Pb) detector and the global probability of boundary (gPb) detector. In addition to these algorithms, section 4.3 will discuss textons, which is a technique used by TextonBoost, Pb and gPb, as well as a few of the real-time detectors.

4.1 NORMALISED CUT SEGMENTATION

Shi and Malik (2000) developed **normalised cut segmentation** by first defining an objective function which identifies what good boundary detection would look like. They then developed an algorithm that would optimally solve this objective function. This section will first discuss the objective function, and then discuss how it is solved. It will conclude with some image results.

Normalised cut segmentation will only be described briefly here, so see Shi and Malik's original paper for a more detailed explanation.

4.1.1 OBJECTIVE FUNCTION

Let P be the set of all pixel positions in image $I(\boldsymbol{p})$, where $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \dots, \boldsymbol{p}_{|P|}\}$. The goal of the normalised cut is to divide P into two disjoint subsets, A and B, so that the cut simultaneously meets the following two criteria:

- Minimum external similarity – the pixels in each subset must be as different as possible from the pixels in the other subset.
- Maximum internal similarity – the pixels in each subset must all be as similar as possible to each other.

The two constraints $A \cup B = P$ and $A \cap B = \emptyset$ mean that the two subsets A and B together form a two-class segmentation of the image. This section will describe how the two criteria can be formulated into two objective functions so that the optimal segmentation can be found.

Let $s(\boldsymbol{p}_i, \boldsymbol{p}_j)$ be the similarity score between pixels $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$. Many different scoring functions can be used – these will be discussed later. Without loss of generality, this thesis will assume the similarity score is always in the range [0, 1].

Let the subset similarity $Sim(A, B) = \sum_{\boldsymbol{a}\in A, \boldsymbol{b}\in B} s(\boldsymbol{a}, \boldsymbol{b})$. This calculates the similarity not just between pixels, but between two entire subsets A and B. Let the normalised similarity $NSim(A, B)$ from subset A to subset B be defined as:

$$NSim(A, B) = \frac{Sim(A, B)}{Sim(A, P)} \tag{4.1}$$

This calculates the similarity from A to B as a proportion of the total similarity score that belongs A. Note that this is not commutative: $NSim(A, B) \neq NSim(B, A)$. This normalisation is important because each pixel has a positive similarity score, and so without this normalisation, simply adding more pixels to a subset increases its similarity. Normalisation eliminates the dependence on the subset size.

Now the objective functions can be defined as follows:

$$\begin{aligned} ExternalSimilarity(A, B) &= NSim(A, B) + NSim(B, A) \\ InternalSimilarity(A, B) &= NSim(A, A) + NSim(B, B) \end{aligned} \tag{4.2}$$

The goal is to find the segmentation of P into (A, B) so that $ExternalSimilarity(A, B)$ is minimised and $InternalSimilarity(A, B)$ is maximised. Shi and Malik showed that if the equations are rearranged, $ExternalSimilarity(A, B) = 2 - InternalSimilarity(A, B)$, which means that these two goals are the same. That means that either objective function can be optimised and the result will be identical.

4.1.2 SIMILARITY SCORES

The similarity score function $s(\boldsymbol{p}_i, \boldsymbol{p}_j)$ can be calculated in many ways. This section will describe the simplest method – **maximum intervening gradient** (Leung & Malik, 1998).

Let $n(\boldsymbol{p}_i, \boldsymbol{p}_j)$ be the maximum intervening gradient between pixels $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$. The process for this is straightforward. Initially, the gradient magnitude at every pixel is estimated using any method, for example, the Sobel operator (section 2.2). Now $n(\boldsymbol{p}_i, \boldsymbol{p}_j)$ is defined as the maximum of all of the gradient magnitudes that lie on the line between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$. From here, the similarity score can be defined as:

$$s(\boldsymbol{p}_i, \boldsymbol{p}_j) = e^{-n(\boldsymbol{p}_i, \boldsymbol{p}_j)/\sigma} \tag{4.3}$$

In equation (4.3), σ is a user-defined scaling parameter, which Leung and Malik set to the standard deviation of the image's gradient magnitudes. The Figure 4-1 shows what this might look like for a particular pixel $\boldsymbol{p}_i$:

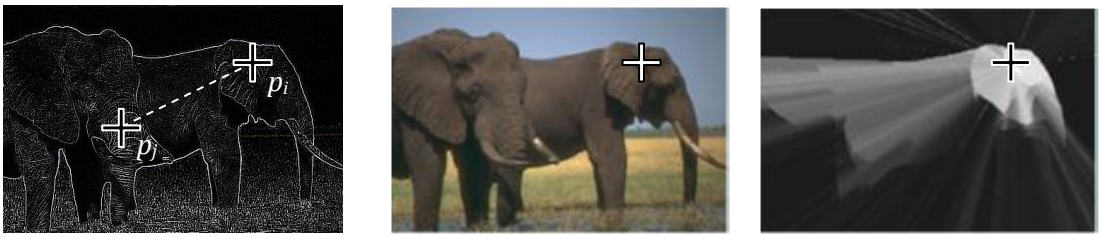


FIGURE 4-1: The Sobel gradient magnitude (left) of an image (middle). The function $\boldsymbol{n}(\boldsymbol{p}_i, \boldsymbol{p}_j)$ is equal to the maximum gradient magnitude between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ as illustrated in the left image. The right image shows the similarity scores of all pixels in the image from pixel $\boldsymbol{p}_i$. Brighter means more similar. Reproduced from Leung and Malik (1998).

Now that the similarity function has been defined, the objective function is ready to be solved.

4.1.3 SOLVING THE OBJECTIVE

Let the solution vector be denoted $\boldsymbol{x} = [x_1, x_2, x_3, \dots, x_N]^T$, where x_i equals $+1$ when pixel $\boldsymbol{p}_i$ belongs to subset A, or -1 when it belongs to subset B. There is a process for finding an approximate optimal solution vector $\boldsymbol{x}^*$ which maximises the objective function $ExternalSimilarity(A, B)$. This subsection will briefly summarise this process, see Shi and Malik (2000) for full details.

Shi and Malik rearranged the objective function $ExternalSimilarity(A, B)$ into this:

$$\begin{aligned} ExternalSimilarity(A, B) &= \frac{\boldsymbol{y}^T(\mathbf{D} - \mathbf{S})\boldsymbol{y}}{\boldsymbol{y}^T\mathbf{D}\boldsymbol{y}} \\ \text{where } \mathbf{S}_{i,j} &= s(\boldsymbol{p}_i, \boldsymbol{p}_j) \\ \mathbf{D}_{i,j} &= \begin{cases} d_i & i = j \\ 0 & i \neq j \end{cases} \\ d_i &= \sum_j s(\boldsymbol{p}_i, \boldsymbol{p}_j) \\ y &= (\mathbf{1} + \boldsymbol{x}) - \frac{k}{1-k}(\mathbf{1} - \boldsymbol{x}) \\ k &= \frac{\sum_{x_i > 0} d_i}{\sum_i d_i} \end{aligned} \tag{4.4}$$

Now the objective is to minimise $ExternalSimilarity(A, B)$ to find the optimal solution $\boldsymbol{y}^*$, and then $\boldsymbol{x}^*$ can be extracted from $\boldsymbol{y}^*$. This rearrangement of the objective function is

a Rayleigh quotient, which already has a proven solution process if the problem is relaxed and $\boldsymbol{x}^*$ is allowed to take real values. It can be shown that the eigenvector of the matrix $(\mathbf{D} - \mathbf{S})/\mathbf{D}$ that has the second-smallest eigenvalue is equal to the optimal real-valued solution of $\boldsymbol{y}^*$. The reason it is the second smallest and not the first is because the first eigenvector will be entirely ones, which is not useful.

So once $\boldsymbol{y}^*$ is found using an eigensolver (the Lanczos eigensolver algorithm is well-suited to this case), the optimal real-valued solution has been found.

4.1.4 BINARISING THE SOLUTION

Normally, the optimal solution vector $\boldsymbol{x}^*$ is not extracted from the optimal vector $\boldsymbol{y}^*$ as $\boldsymbol{y}^*$ already indicates the optimal solution sufficiently. This can be seen on some example images by reshaping $\boldsymbol{y}^*$ back into an image shape:

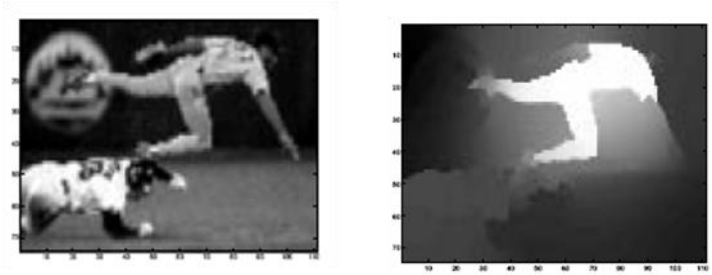


FIGURE 4-2: An image (left) and its optimal solution eigenvector $\boldsymbol{y}^*$ (right) reshaped into the image's shape. Reproduced from Shi and Malik (2000).

Figure 4-2 clearly shows that the solution eigenvector $\boldsymbol{y}^*$ is a real-valued vector. To convert $\boldsymbol{y}^*$ into a two-class segmentation, $\boldsymbol{y}^*$ is simply thresholded. Normally, the threshold level is found by trying many different levels, and the threshold level that best satisfies the dual objective functions is taken.

4.1.5 SUBDIVIDING FURTHER

Running the process just described will divide the image into two segments. This process can be repeated to further subdivide those segments. One particular problem with this is that it is slow to run the entire normalised cut algorithm multiple times in order to subdivide the image further. There is a way to speed this up.

Section 4.1.3 stated that if the eigenvectors are sorted by ascending eigenvalue, the first eigenvector is not useful, while the second eigenvector represents the optimal real-valued solution. It turns out that the third eigenvector and above also contain some information about further subdivisions in the image, although the eigenvectors get less and less accurate as the eigenvalues increase. In their paper, Shi and Malik describe a

way to use these higher eigenvalues to achieve a slightly approximated solution in order to achieve greater speeds.

4.1.6 IMAGE RESULTS

These are some results from running normalised cut segmentation (Barnard, Duygulu, Guru, Gabbur, & Forsyth, 2003):

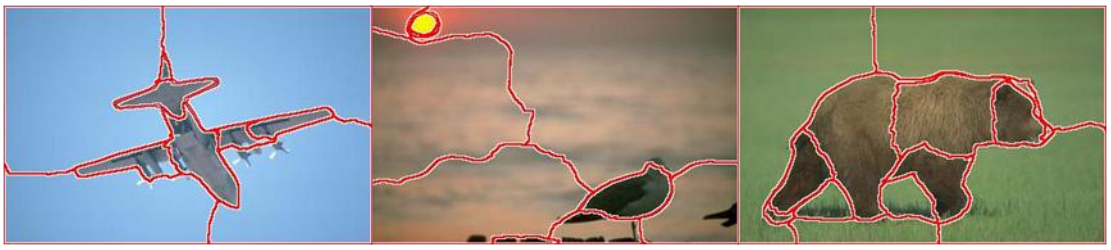


FIGURE 4-3: The normalised cut on some images. Reproduced from Barnard *et al.* (2003)

As the normalised cut is always looking for a global solution, it generally is not distracted by intra-texture edges, as Figure 4-3 shows. It has two major problems though.

First, it is not able to run in real-time, due to its intensive high-dimensional eigensolver stage.

Second, normalised cut segmentation suffers from the "broken sky" problem. Take the example of the sky, shown in three of the images above. The sky smoothly changes from dark to light, and vice versa, throughout the image. That means, some parts of the sky are quite dissimilar to other parts of the same sky, and so it is optimal for normalised cut segmentation to cut the sky into pieces to in order to achieve maximum internal similarity.

Mean-shift segmentation, which will be introduced next, does not suffer from the "broken sky" problem.

4.2 MEAN-SHIFT SEGMENTATION

Previously, section 3.6 already described the mean-shift filter, which is the basis of **mean-shift segmentation** (Comaniciu & Meer, 2002). Mean-shift segmentation first takes the mean-shift filtered image, and splits it into a number of initial regions. Initially, each spatially-contiguous island of equally-coloured pixels becomes one **region**. A boundary is detected at all positions where two neighbouring pixels belong to a different region.

Unfortunately, using the initial regions directly from the mean-shift filter is not useful. Recall that mean-shift filtering works by clustering pixels together. As there is no requirement that a cluster must be spatially-contiguous, the initial regions will consist of many, small, disconnected, interwoven regions. This is illustrated in Figure 4-4:

FIGURE 4-4: This is the top-right section of the mean-shift filtered mandrill image that was originally shown in Figure 3-12. Adapted from Comaniciu and Meer (2002).

There are five different clusters represented in Figure 4-4, yet because they all overlap and are not required to be spatially-contiguous, there are probably around one hundred separate regions of pixels, most of them only a few pixels large.

To deal with this, mean-shift segmentation imposes a minimum region size. Any regions smaller than the minimum region size are simply combined to the neighbouring region of most similar colour. The regions are repeatedly combined until all regions meet the minimum region size requirement. At that point, mean-shift segmentation has finished, and a boundary is detected wherever two neighbouring pixels belong to different regions.

4.2.1 IMAGE RESULTS

The minimum region size requirement of mean-shift segmentation gives it a certain amount of ability to ignore intra-texture boundaries, as illustrated in Figure 4-5 below:

FIGURE 4-5: Mean-shift segmentation on example images. Reproduced from Comaniciu and Meer (2002).

As the above figure shows, mean-shift segmentation is an excellent texture-boundary detector. However it has two problems. First, it is not able to run in real-time, because its clustering process must iterate over the image many times in order to converge. Second, it subdivides some textured areas, introducing boundaries in places where they should not exist. In Figure 4-5 this is illustrated in some of the grass and tree textures. This occurs because mean-shift segmentation has no explicit understanding of texture, and so it cannot see that differently-shaded grass areas are all actually the same texture and so should not be subdivided. The next section presents textons, which allow an algorithm to explicitly model texture and overcome this problem.

4.3 TEXTONS

Many state-of-the-art texture-boundary detectors use **textons** to recognise or distinguish between textures. The purpose of this section is to detail the texton technique as it will be used by most of the remaining boundary detectors in this thesis.

4.3.1 THEORY: AUTOCORRELATION

Central to the idea of textons is the idea of that texture is **autocorrelated**. That means, within a texture, there is normally some mathematical relationship between each pixel and other nearby pixels. By definition, texture repeats with a particular pattern, and so

of course the value of each pixel cannot be entirely random and independent of its neighbours.

Every texture has its own unique autocorrelation pattern, which can be used to distinguish it from all other textures.

Example of an autocorrelation pattern

The following example has been adapted from Varma and Zisserman's (2003) paper.

Three samples of texture are shown in the Figure 4-6. Two of the samples are of the same texture.

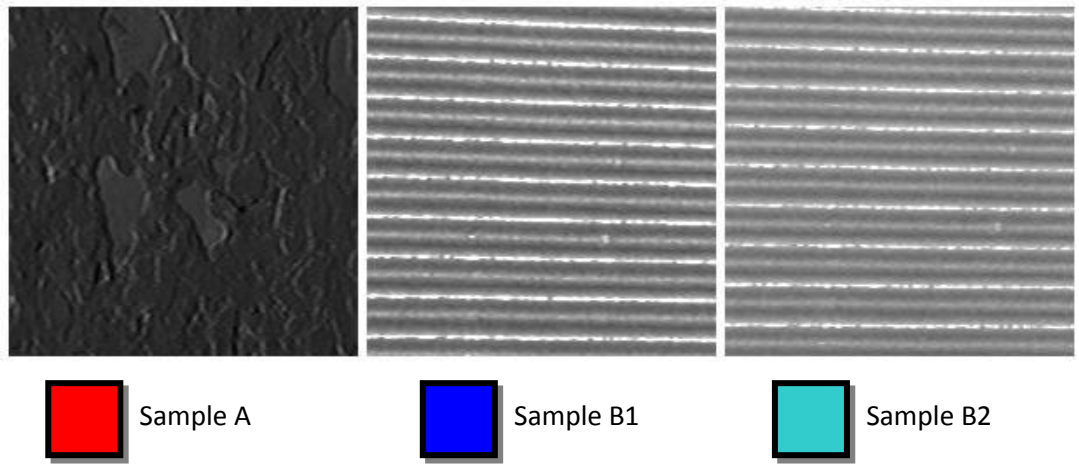


FIGURE 4-6: Three samples of textures. Samples B1 and B2 are different samples of the same texture. Adapted from Varma and Zisserman (2003).

The graph below plots two **features** against each other: (1) the brightness value of each pixel $I(\boldsymbol{p})$, and (2) the brightness of an offset pixel $I(\boldsymbol{p}+\boldsymbol{p}')$ where $\boldsymbol{p}' = [2,2]^T$. In words, this means that each pixel is plotted against another pixel which is two pixels right and two pixels down. Each sample of texture has been plotted separately in a different colour in order to illustrate how they can be differentiated.

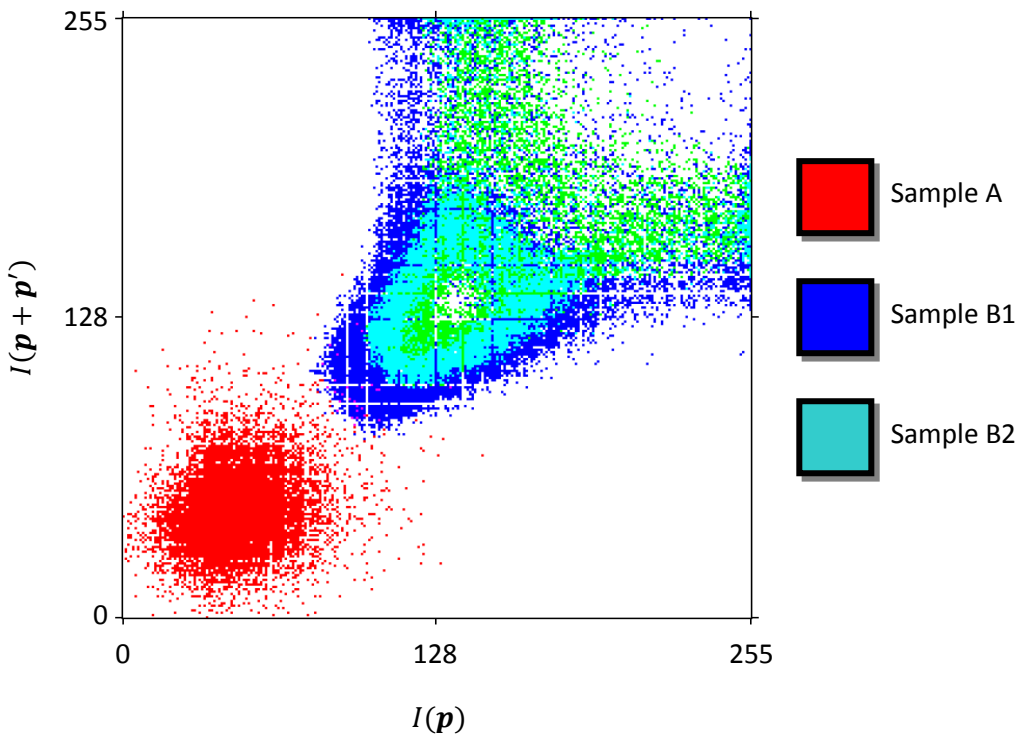


FIGURE 4-7: A plot of offset brightness values in all three samples of texture. Adapted from Varma and Zisserman (2003).

What is most obvious in Figure 4-7 is that there are two separate **clusters** of data. Sample A has clustered to the lower-left part of the graph, while samples B1 and B2 have both clustered to the upper-right part of the graph. The fact that the data has formed clusters indicates that the two features are not independent, but are actually correlated. If they were independent, there would be no obvious pattern between the two features. Furthermore, since two features are both extracted from the same texture, this says that these textures are correlated with themselves. That is, they are **autocorrelated**.

So each texture class will form clusters in different parts of the feature space. The word "clusters" is plural here because often the same texture will generate multiple clusters at different places in the feature space. In the context of texture, one of these clusters is called a **texton**.

Different samples of the same texture class will form the same textons, while samples from the different texture classes will form different textons. In other words, each texture class has its own characteristic set of textons, like a fingerprint. Each texture's "fingerprint" is different, which means textures can be recognised and distinguished. This is the basis of the texton technique.

4.3.2 FEATURES

Let the function $\Phi_I(\boldsymbol{p})$ calculate the feature vector for pixel $\boldsymbol{p}$. In the previous section, $\Phi_I(\boldsymbol{p})$ would have been defined like this:

$$\Phi_I(\boldsymbol{p}) = \begin{bmatrix} I(\boldsymbol{p}) \\ I(\boldsymbol{p} + \boldsymbol{p}') \end{bmatrix}$$
$$\text{where } \boldsymbol{p}' = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \tag{4.5}$$

There are many ways to extract features; this method above is only simple illustrative example. In practice, two simple features like this do not have enough information to distinguish between textures. That is why, more commonly, convolution is used for feature extraction. In this approach, the algorithm will have a filter bank $F = \{f_1, f_2, f_3, \dots, f_{|F|}\}$, that consists of a number of convolution kernels f_i. Now, the feature vector will be calculated by convolving the image with all the kernels in the filter bank:

$$\Phi_I(\boldsymbol{p}; F) = \bigcup_{f \in F} (I * f)(\boldsymbol{p}) \tag{4.6}$$

TextonBoost (Shotton J. , Winn, Rother, & Criminisi, 2009), one of the best texture-boundary detectors today, uses the Winn-Criminisi-Minka filter bank (2005):

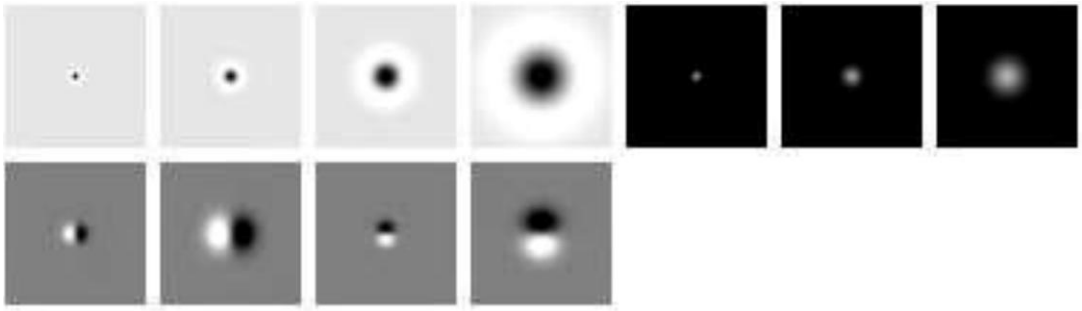


FIGURE 4-8: In reading order, the Winn-Criminisi-Minka (2005) filter bank consists of four scales of Laplacian of Gaussian kernels, three scales of Gaussian kernels, and four scales of Gaussian first derivative kernels. These kernels have had their values scaled to the [0,1] range so they could be viewed.

One thing to notice about the Winn-Criminisi-Minka filter bank is that, the same filters are repeated at different scales. This ensures that textures will be able to be distinguished, regardless of their scale.

In general, any filter bank will be comprised of two categories of filters – comparison filters and averaging filters.

Comparison filters

Comparison filters compare one part of the texture to another, similar to the example in section 4.3.1.

In the Winn-Criminisi-Minka filter bank, the Gaussian first-derivative and Laplacian kernels belong to this category. For each sliding window, the Gaussian first-derivative compares one side of to the other, while the Laplacian compares the inside to the outside. Naturally, some textures will cluster more uniquely with the Gaussian first-derivative while others will cluster more uniquely with the Laplacian, so it is good to have both.

Averaging filters

Averaging filters extract the average colours that comprise a texture. Obviously, textures often have a normal colour – grass is usually green, while the sky is usually blue for example. So extracting average colour information is useful for distinguishing between textures.

In the Winn-Criminisi-Minka filter bank, the averaging filters are the three Gaussian kernels. They extract a Gaussian-weighted average of the local colour around each pixel at various different scales. Naturally, if a texture has a usual colour or colours, then its average colour values will cluster in a characteristic way, enabling it to be recognised.

4.3.3 LEARNING TEXTONS WITH K-MEANS CLUSTERING

The texton technique must first learn textons in a training stage before they can be used. This is done from a set of training images.

Initially, a large set of features are extracted from a training set of images using the previously-defined feature-extraction function $\Phi_I(\boldsymbol{p}; F)$. Let this training feature set be denoted $\Phi = \{\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \boldsymbol{\phi}_3, \dots, \boldsymbol{\phi}_{|\Phi|}\}$, where each $\boldsymbol{\phi}_i$ is a feature vector sampled from a training image. Now, the textons are simply the clusters in the training set Φ. Normally, k-means clustering (Lloyd, 1982) is used to find these clusters.

K-means clustering

K-means clustering is a method to automatically find exactly k clusters (or more specifically, textons) in the set of feature points Φ. The number of clusters k is set as an external parameter to the clustering algorithm. This is different to other clustering algorithms, such as mean-shift clustering (section 3.6), where the number of clusters

arises organically. In the case of textons, previous researchers have set k to values as low as $k = 32$ and as high as $k = 400$.

In k-means clustering, a cluster is defined by its central point $\boldsymbol{c}_i$. Let $C = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3, \dots \boldsymbol{c}_k\}$ be the set of cluster centers. As the feature points Φ exist in $\mathbb{R}^d$ space, where d is the number of features in a feature vector, each cluster center is also an $\mathbb{R}^d$ point in the feature space.

From here, the cluster of a feature point $\boldsymbol{\phi}$ can be determined by $c(\boldsymbol{\phi})$:

$$c_{index}(\boldsymbol{\phi}) = \arg\min_{i \in \{1 \dots |C|\}} \|\boldsymbol{c}_i - \boldsymbol{\phi}\|$$
$$c(\boldsymbol{\phi}) = \boldsymbol{c}_i \text{ where } i = c_{index}(\boldsymbol{\phi}) \quad \textbf{(4.7)}$$

Basically, a feature always belongs to its closest cluster. The function $c_{index}(\boldsymbol{\phi})$ only returns the index i of the cluster, while $c(\boldsymbol{\phi})$ returns the actual cluster center. Given these definitions, k-means clustering proceeds as follows:

ALGORITHM 4-1

1. Initialise C to be a set of k random of cluster centers.
2. Let $\Phi_{\boldsymbol{c}} \leftarrow \{\boldsymbol{\phi} \in \Phi : c(\boldsymbol{\phi}) = \boldsymbol{c}\}$. In words, $\Phi_{\boldsymbol{c}}$ is the set of training features that belong to cluster $\boldsymbol{c}$.
3. Let $\boldsymbol{c} \leftarrow \sum \Phi_{\boldsymbol{c}} / |\Phi_{\boldsymbol{c}}|$ for all $\boldsymbol{c} \in C$. In words, move each cluster to the mean of the features in the cluster.
4. If the clusters moved a total distance less than ϵ, a final set of clusters has been found, so return C. The parameter ϵ should be set to a small value.
5. Go back to step 2

To ensure a good result, Algorithm 4-1 is run multiple times from multiple different random starting clusters, and the most compact set of clusters is chosen. Compactness can be measured as follows:

$$Compactness(C) = \sum_{\boldsymbol{\phi} \in \Phi} \|\boldsymbol{\phi} - c(\boldsymbol{\phi})\| \quad \textbf{(4.8)}$$

Now that the clusters C are known, it is possible to textonise images.

4.3.4 TEXTONISING IMAGES

Textonising is the process of transforming an image $I(\boldsymbol{p})$ into a **texton map** $T(\boldsymbol{p})$. A texton map is an image which describes which texton each pixel has been assigned to.

The textonisation $T(\boldsymbol{p})$ of pixel $\boldsymbol{p}$ in image I is defined as:

$$T(\boldsymbol{p}) = t_{\phi}\big(\Phi_I(\boldsymbol{p})\big)$$
$$\text{where } t_{\phi}(\boldsymbol{\phi}) = c_{index}(\boldsymbol{\phi}) \qquad \textbf{(4.9)}$$

Basically, the above equation states that each pixel in the image is assigned to its nearest texton. The textons will have already been decided through clustering, as described in the previous section.

Notice that the texton map $T(\boldsymbol{p})$ consists of the texton indices. So if there are $k = 100$ textons, then each pixel in the textonised image $T(\boldsymbol{p})$ will be an integer in the range [1, 100].

4.3.5 IMAGE RESULTS

Many texton-based algorithms exist. Figure 4-9 shows three texton map examples from three different algorithms.

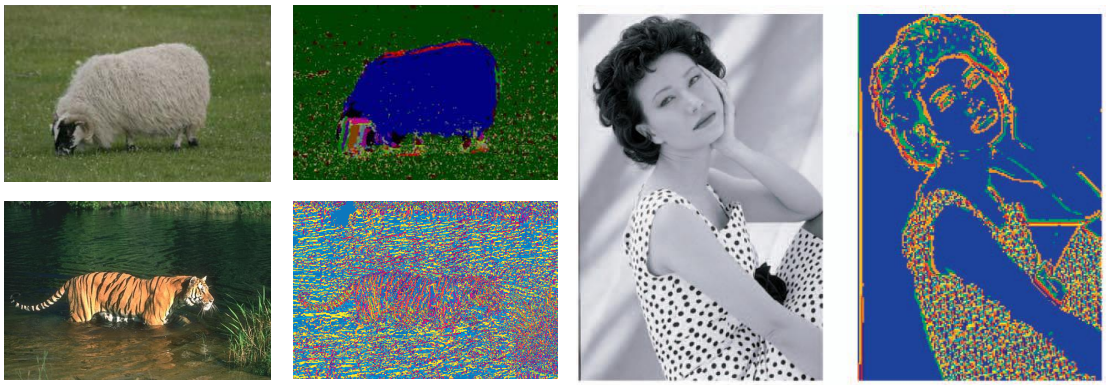


FIGURE 4-9: Texton map examples taken from some existing algorithms. Top left pair is from TextonBoost (section 4.4), which uses $\boldsymbol{k} = \mathbf{400}$ textons. Bottom left pair is from the Texton Ridge Detector (proposed in chapter 7), which uses $\boldsymbol{k} = \mathbf{32}$ textons. The right pair is from the probability of boundary detector (section 4.5), which uses $\boldsymbol{k} = \mathbf{64}$ textons. The texton maps have been false-coloured – each texton is rendered in a different colour.

The key point to notice is that each texture is made up of a different distribution of textons. This is particular noticeable in the tiger image (bottom left of Figure 4-9), where the tiger itself and the water surrounding the tiger have clearly different texton distributions, even if they share some textons.

Different algorithms use the texton map in different ways. The next sections will discuss the two main approaches by introducing three different algorithms:

- TextonBoost (section 4.4) tries to learn how to recognise textures from the texton map. It does extremely well at this problem, but suffers from being inflexible because it always must be trained on a limited number of textures.
- The probability of boundary detector (section 4.5) takes a flexible approach which can work with an unlimited number of textures, but unlike TextonBoost, it does not attempt to find a globally-optimal solution.
- The global probability of boundary detector (section 4.6) improves the probability of boundary detector so that it attempts to find a globally-optimal solution.

4.4 TEXTONBOOST

TextonBoost (Shotton J. , Winn, Rother, & Criminisi, 2009) is one of the most influential texture-boundary detectors. Both real-time and non-real-time texture-boundary detectors have been based on its concepts.

In the 2009 paper, TextonBoost was trained to recognise 23 different textures. It does this by learning from human-labelled training images such as the ones below in Figure 4-10:

FIGURE 4-10: TextonBoost learns from human-labelled images like these. Reproduced from Shotton *et al.* (2009)

TextonBoost simultaneously tries to achieve not only pixel-perfect boundary detection, but also texture recognition, as shown in Figure 4-11:

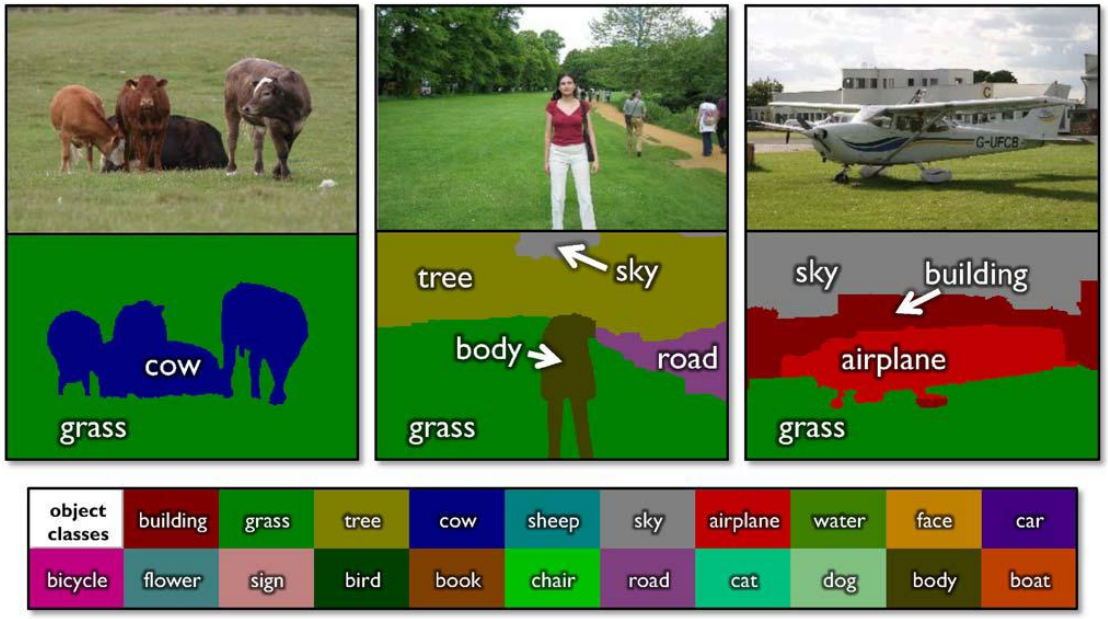


FIGURE 4-11: The result of applying TextonBoost to some images. Notice its high-quality boundaries. Reproduced from Shotton *et al.* (2009)

TextonBoost has a number of parts to it. First, it has a training phase, where it learns to recognise textures from textons. Second, it has a recognition phase, where it classifies new images using the model that it has learnt. The recognition phase can be split into two parts:

1. **Soft-assignment.** TextonBoost maps textons to textures using a technique called texture-layout filters. Soft-assignment means that each pixel is not assigned to a single texture class yet. Instead, the soft-assignment for a pixel consists of the likelihoods of it belonging to each of the textures.

2. **Hard-assignment**. From there, TextonBoost uses an alpha-expansion graph cut to hard-assign the pixels to texture based on what appears to be globally optimal.

Each of these parts will be discussed individually.

4.4.1 TEXTON FEATURES

Before any texture processing can occur, the images must first be textonised. This is done in the same way as described in section 4.3, using the Winn-Criminisi-Minka filter bank (2005) shown in Figure 4-8. TextonBoost was trained with $k = 400$ textons in its original paper.

4.4.2 TEXTURE-LAYOUT FILTERS

The soft-assignment of pixels to textures is done using texture-layout filters.

A texture-layout filter generates a response equal to the frequency of a texton within a particular offset rectangle, as illustrated in Figure 4-12:

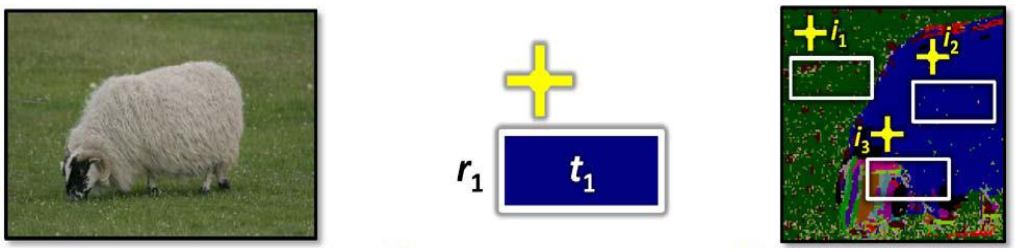


FIGURE 4-12: (a) An image. (b) One example of a texture layout filter. (c) The texture layout filter, applied to different areas of the image. Reproduced from Shotton *et al.* (2009)

The texture-layout filter shown in Figure 4-12 generates a response for the pixel denoted by the yellow cross by counting the frequency of texton t_1 within the offset rectangle r_1. Figure 4-12(c) shows three representative placements of the texture-layout filter:

- At position i_1, the texture-layout filter will generate a 0% response, as texton t_1 does not occur in the offset rectangle r_1.
- At position i_2, the response will be approximately 100%, as texton t_1 occurs at almost every pixel in the offset rectangle r_1.
- At position i_3, the response will be approximately 50%, as texton t_1 occurs in approximately 50% of the offset rectangle r_1.

TextonBoost will transform this filter response into a vote for one or more textures. The strength of the vote will be proportional to the response. The votes from an ensemble of texture-layout filters are tallied together to find each pixel's soft-assignment. TextonBoost also uses other visual cues – namely colour and pixel location – to cast votes for textures as well. However, these cues are secondary to the texture-layout filters, and so will not be discussed here.

Learning the texture-layout filters

Before any textures can be recognised, TextonBoost must first learn the ensemble of texture-layout filters in its training stage. It does this using a machine learning technique called **boosting**.

Shotton *et al*. (2009) trained TextonBoost with two thousand texture-layout filters so that it could recognise 23 different textures. Boosting is made up of many rounds, where each round of boosting trains one additional texture-layout filter. Consequently, this would have required two thousand rounds of boosting.

In the case of TextonBoost, each boosting round is a random search algorithm. That is, in each round, a large number of random candidate texture-layout filters are tried, and the "best" one is chosen. The "best" texture-layout filter is basically the most accurate one – that is, the one that votes for the correct texture the highest proportion of the time. The exact details of how boosting works in TextonBoost can be seen in the TextonBoost paper.

Once the soft-assignments have been generated by the texture-layout filters, the pixels must then be hard-assigned to textures. This is done using an alpha-expansion graph cut. The next section (section 4.4.3) will explain why an alpha-expansion graph cut is needed at all. This section is important because two of the real-time texture boundary detectors attempt to remove the alpha-expansion graph cut stage, to varying levels of success. The section after that (section 4.4.4) will discuss the minimum cut, which forms the basis of the alpha-expansion graph cut (section 4.4.5).

4.4.3 WHY NOT JUST HARD-ASSIGN A PIXEL TO ITS MODAL TEXTURE?

Whenever texture is involved, the local soft-assignments must be combined with the local context in order to make a high-quality hard-assignment of pixels to textures. If this is not done, the resulting boundary map will be very noisy. Consequently, simply assigning a pixel to its most likely texture (its **modal texture**) will produce a low-quality boundary map because it does not consider any local context. One of the best examples of this is in the results of a texture-boundary detector developed by He, Zemel and Carreira-Perpinan (2004), shown in Figure 4-13.

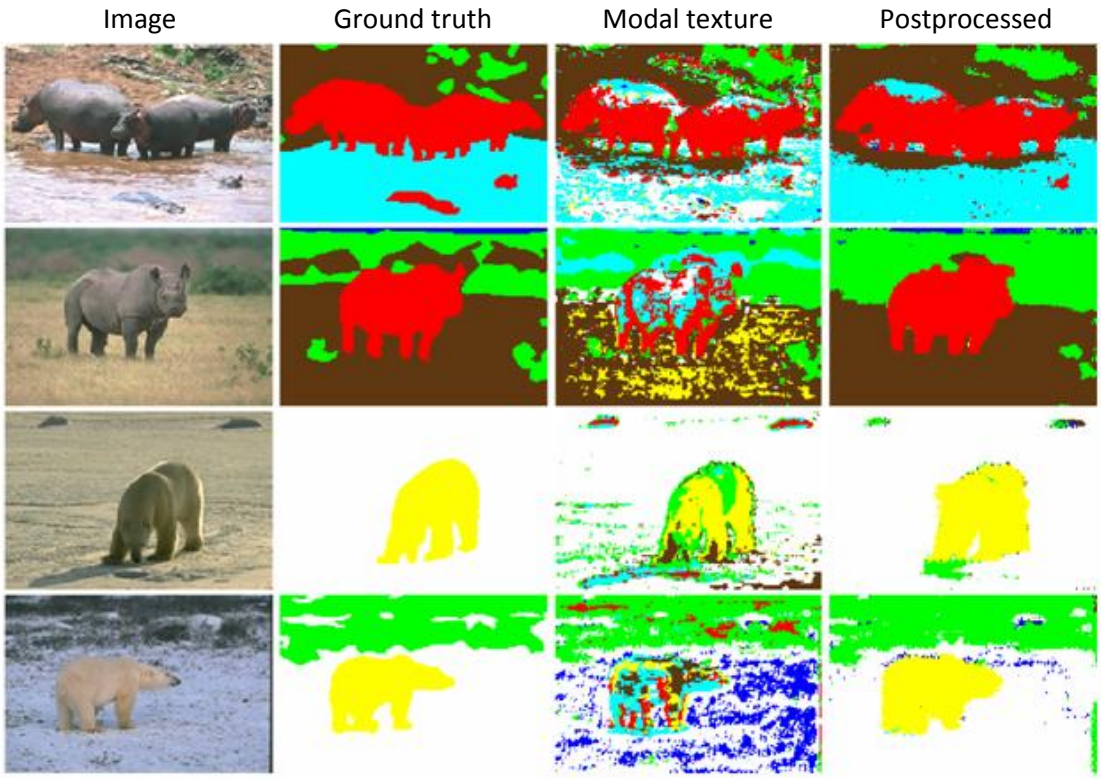


FIGURE 4-13: The effect of post-processing on He *et al.*'s (2004) texture-boundary detector. Columns from left to right: (1) the original image, (2) the human-assigned ground truth, (3) the output of the algorithm when each pixel is hard-assigned to its modal texture, (4) the output of the algorithm when hard-assignment is done with an alpha-expansion graph cut. Adapted from He *et al.* (2004)

He *et al*. were the original proponents of using an alpha-expansion graph cut to smooth the textures calculated by a texture-boundary detector, and their results in Figure 4-13 clearly show how much difference this makes. The unsmoothed results are highly noisy, while the smoothed results are substantially more accurate.

The reason why smoothing is so necessary is as follows. Texture potentials are calculated from local features, and so they are heavily subject to local pixel variations. This makes it absolutely necessary to integrate these local features with their local context to produce a good image-level interpretation of the textures in the image. The alpha-expansion graph cut, used by TextonBoost, does exactly this.

4.4.4 THE MINIMUM CUT

Understanding the alpha-expansion graph cut first requires understanding its most essential component – the minimum cut.

The purpose of minimum cut segmentation (Ford & Fulkerson, 1956; Greig, Porteous, & Seheult, 1989) is to divide an image into two separate classes. This section will explain the minimum cut with a graphical example, using the image below:

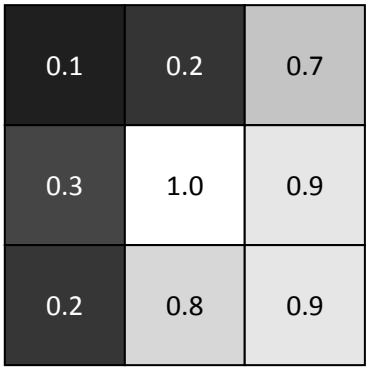


FIGURE 4-14: This 3×3 image is made up of both dark and light pixels. The numbers overlaid on each pixel are the pixel brightness values, where all brightnesses are in the range [0, 1]. The minimum cut will be demonstrated on this image.

The above example image has two classes of pixels – dark and light. In this example, the goal of minimum-cut segmentation is to find the cut which best separates the dark class from the light class. This is not straightforward because there is some intra-class variation.

Converting the image to a graph

The minimum cut algorithm first converts an image into a graph, as illustrated on the example image below:

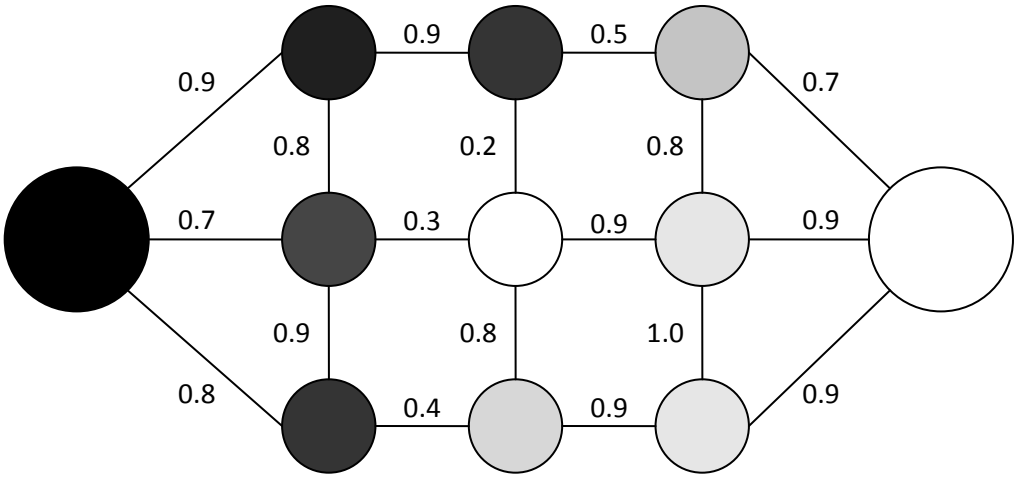


FIGURE 4-15: The example image, converted to a graph. The large nodes on the left and right side are class nodes, while the nine nodes in the middle are the pixel nodes – one for each pixel.

In the image graph, each pixel is represented as a pixel node. Additionally two class nodes are created for each of the two classes. In Figure 4-15, the dark class is

represented by the large dark node on the left, while the light class is represented by the large light node on the right.

An arc connects each of the pixel nodes to its neighbours. Also, the pixels on the sides have been connected to the class nodes. Normally, each class node would be connected to every pixel node in the entire image. This has not been done to make the example look clearer. Without loss of generality, this example has been engineered so that the result will still be the same with this simplification.

The minimum cut algorithm requires all the arcs of the graph to be given a **weight**, where the weight is a similarity score. This similarity score can be calculated in a multitude of ways. The example will use the simplest ways possible.

In this example, the similarity of two pixel nodes can be defined as follows. First the difference in their brightness values is taken, and then the difference is subtracted from the maximum brightness value to make it a similarity score and not a difference score. If the two pixels are denoted by $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$, then the similarity score between them is defined as: $1 - \left|I(\boldsymbol{p}_i) - I(\boldsymbol{p}_j)\right|$. The 1 in this equation is used because it is the maximum brightness value.

The similarity of a pixel node to a class node will be calculated by simple colour similarity in this example. Therefore, the similarity of a pixel $\boldsymbol{p}$ to the dark class node will be: $1 - |0 - I(\boldsymbol{p})|$. Likewise, the similarity of a pixel $\boldsymbol{p}$ to the light class node will be: $1 - |1 - I(\boldsymbol{p})|$.

The similarity scores of the example image according to these calculations can be seen above in Figure 4-15 where each arc has been given a weight equal to its similarity score.

Finding the minimum cut

In graph theory, a **cut** is defined as a set of arcs which, when removed (or "cut"), divide the graph into two disjoint subsets. In this context, each cut will have a cost, where the cost is the sum of the weights of the arcs that are cut. The **minimum cut** is the cut which has the minimum cost. In this case, the cut of minimum cost is equivalent to the cut of maximum dissimilarity, which makes it useful for boundary detection.

The minimum cut can be found exactly by applying a simple algorithm (Ford & Fulkerson, 1956):

ALGORITHM 4-2: The minimum cut algorithm.

1. Find the shortest path *p* between the class nodes.
   1.1. If no path exists, the minimum cut has been found, so stop.
2. Let *m* = the minimum value of all arcs on path *p*.
3. Subtract *m* from all arcs on that path *p*.
4. Cut any arcs that are now equal to zero.
5. Go back to step 1.

The following series of figures show how the minimum cut algorithm listed in Algorithm 4-2 will eventually progress to the optimal solution.

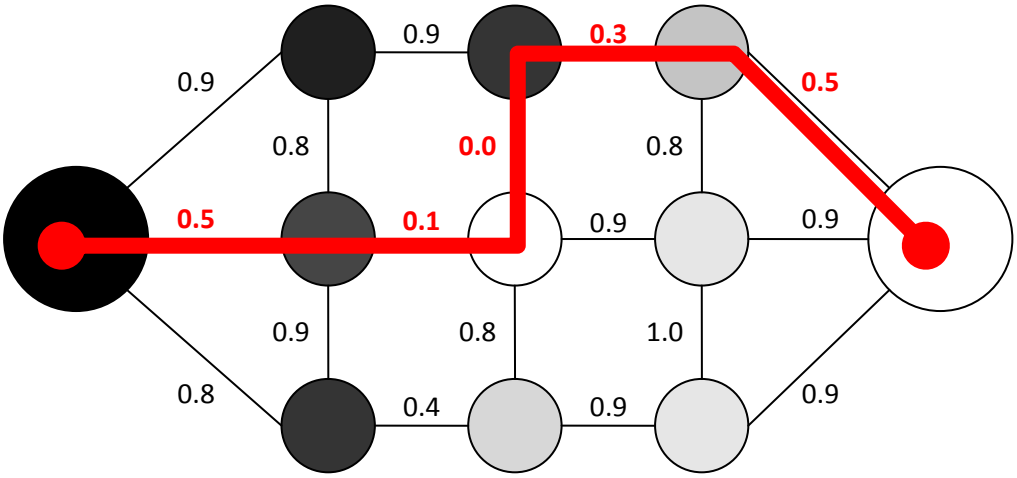


FIGURE 4-16: The shortest path is found (shown in red), and its minimum value is subtracted from all arcs on the path. One of the arcs becomes zero, and so it is cut (shown in the next figure).

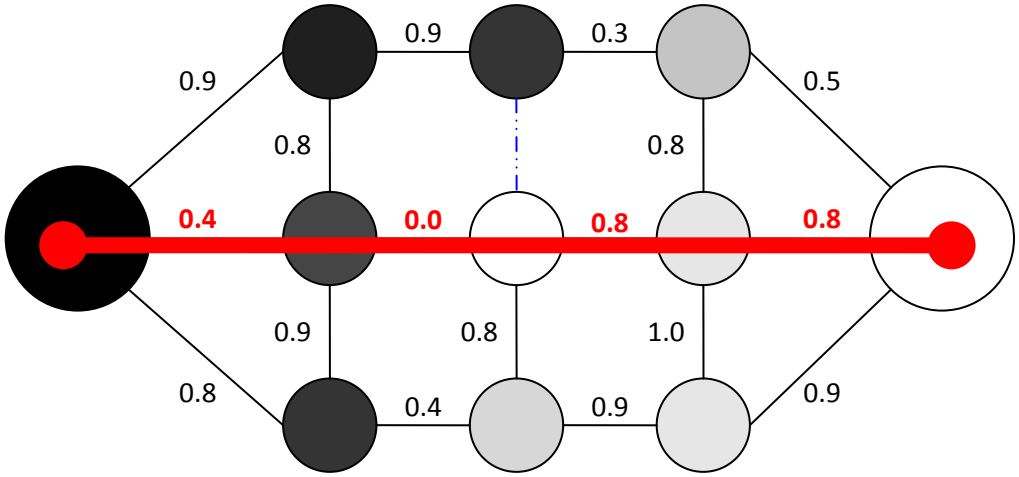


FIGURE 4-17: One of the arcs has been cut (dotted blue line). The algorithm repeats. The shortest path (in red) cannot cross the arc that was just cut, so that is why the shortest path has changed. Again, the minimum is subtracted from the shortest path. One of the arcs becomes zero, and so it is cut (shown in the next figure).

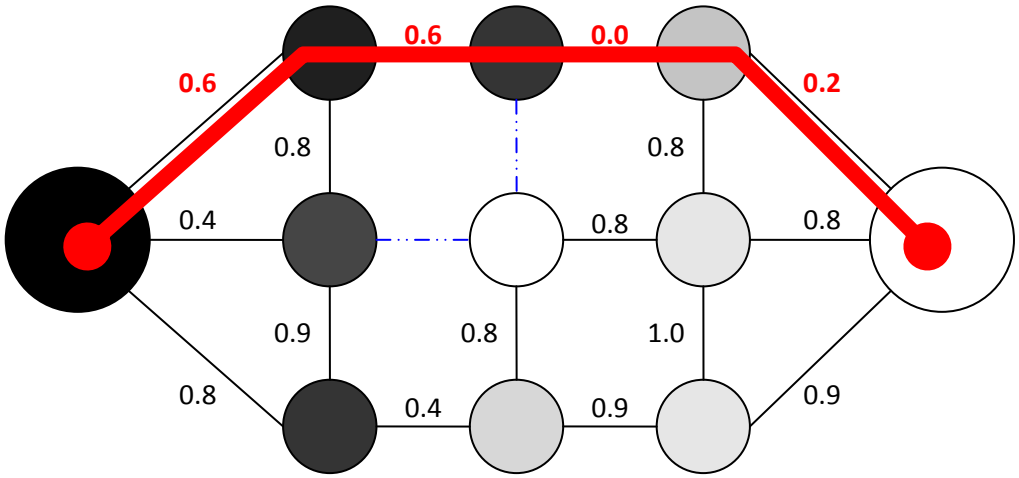


FIGURE 4-18: Two arcs have now been cut. The algorithm repeats, and another arc is cut (shown in next figure).

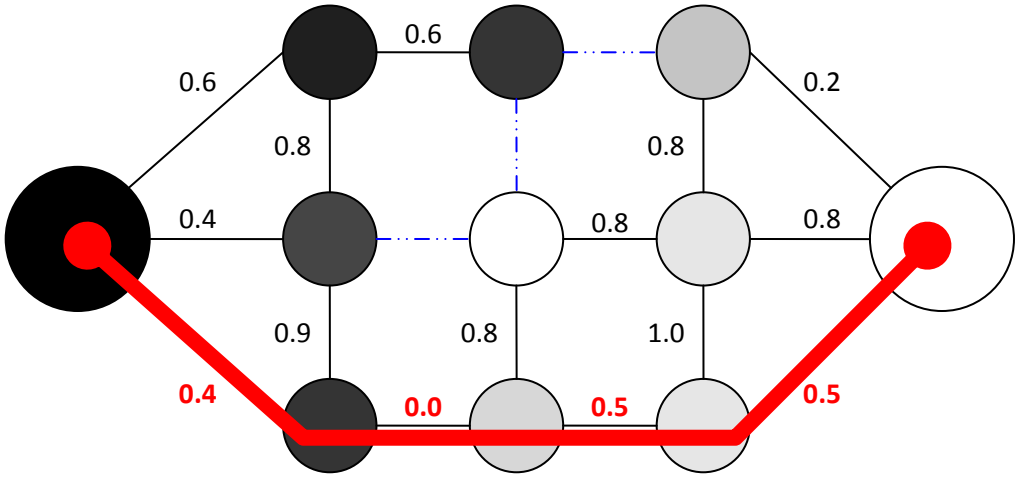


FIGURE 4-19: Now three arcs have been cut. The algorithm repeats and cuts another arc (shown in next figure).

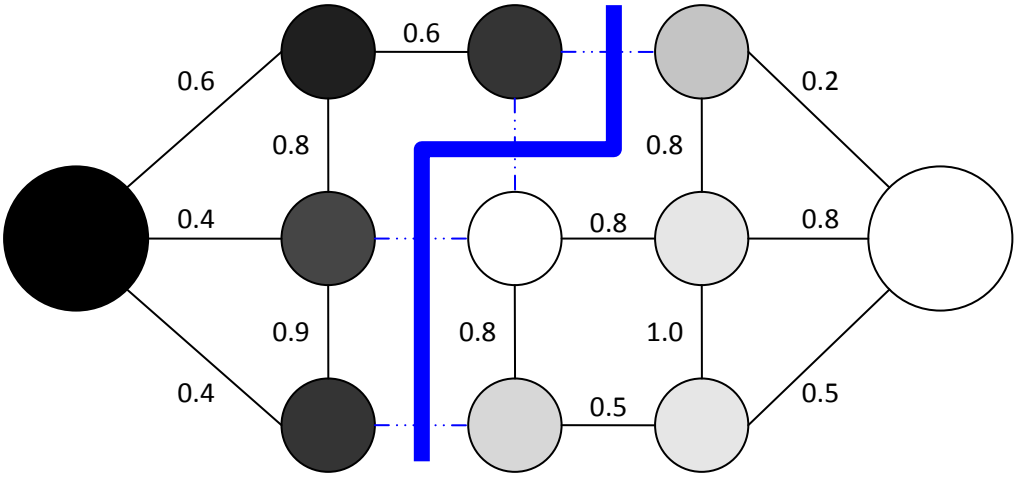


FIGURE 4-20: Now four arcs have been cut. Now no path exists between the two class nodes, so the algorithm has finished. The minimum cut is shown as a thick blue line.

Figure 4-20 above shows the final result of the algorithm on the example initially shown in Figure 4-14. Notice that the minimum cut algorithm has perfectly separated the dark pixels from the light pixels. The class of a particular pixel can be found by tracing the graph to find out which of the class nodes it is connected to. This illustrates one of the most important characteristics of minimum-cut segmentation – it always finds the optimal solution.

As its similarity scoring functions are fully customisable, it is possible to use the minimum cut to solve a much more complicated texture-based boundary detection problem. The problem is though, the minimum cut algorithm can only ever separate two classes. TextonBoost needs to be able to separate between many more than two classes in order to be useful. For this reason, the alpha-expansion graph cut was developed.

4.4.5 ALPHA-EXPANSION GRAPH CUTS

Alpha-expansion graph cuts (Boykov & Jolly, 2001) are a workaround to the problem of a graph cut only being able to separate two classes.

Initially, TextonBoost begins with a simple solution, where each pixel is assigned to its most likely texture. TextonBoost then repeatedly uses an alpha-expansion graph cut to improve the solution. When the solution stops improving, TextonBoost stops and returns that solution as the final hard-assignment.

An alpha-expansion graph cut works like this. Let the set of all textures be denoted $\tau = \{\tau_1, \tau_2, \tau_3, \dots, \tau_{|\tau|}\}$. TextonBoost chooses one of the textures and calls it α, where $\alpha \in \tau$. Now each pixel must now make a decision to either:

1. Remain as its current texture τ_i, or

2. Switch to the new texture α

Clearly, the alpha-expansion graph cut problem has two classes, and so it can be solved optimally using the minimum cut. The weights of the graph can be calculated easily because each pixel's affinity to each of the textures is already known from the soft-assignment stage.

The alpha-expansion graph cut is applied repeatedly in this manner. To ensure there is no bias in the results, each texture in τ should get a chance to be α an equal number of times over the course of the hard-assignment stage. Eventually, this process will converge on a strong local maximum, and this is the final result of TextonBoost.

4.4.6 IMAGE RESULTS

Figure 4-21 shows some examples of how TextonBoost performs.

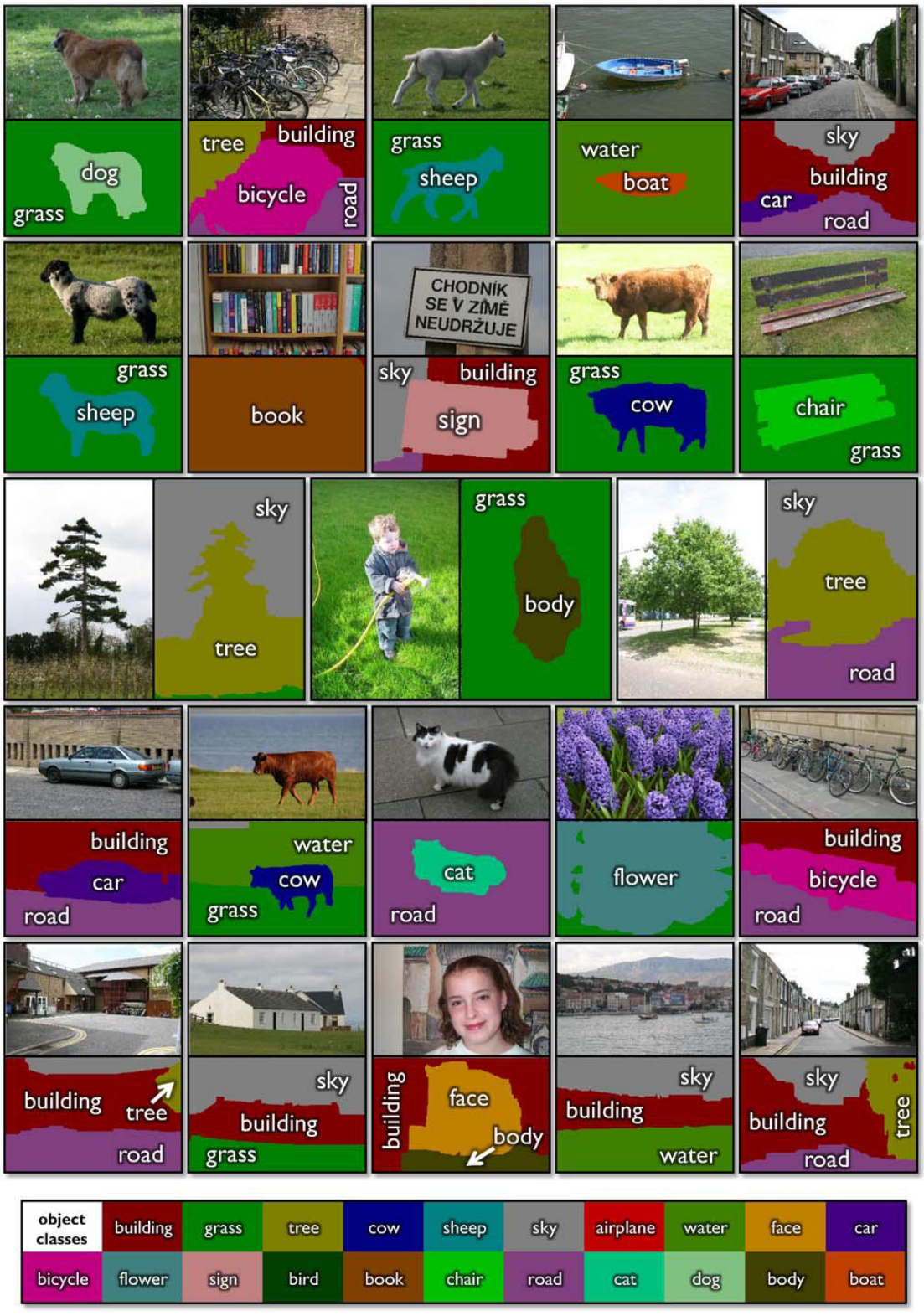


FIGURE 4-21: Some example results of TextonBoost. Reproduced from Shotton *et al.* (2009)

As Figure 4-21 shows, TextonBoost produces excellent boundaries and can recognise texture well. It has two problems though. First, it is unable to run in real-time, primarily because of its iterative alpha-expansion graph cut stage. Second, it is always limited to the textures that it is trained to recognise, which limits its practical applications to

controlled conditions. The probability of boundary detector, introduced next, does not have this problem.

4.5 PB: THE PROBABILITY OF BOUNDARY DETECTOR

Like TextonBoost, the probability of boundary (Pb) detector (Martin, Fowlkes, & Malik, 2004) uses textons to find boundaries. However, it takes a very different approach from TextonBoost. The Pb detector calculates the local change in texton distributions as the texton gradient. It then performs ridge detection on the texton gradient (called localisation in the original paper). Finally, a logistic regression model combines the texton ridges with some other visual cues to produce a boundary map. Each of these stages will be discussed individually.

4.5.1 TEXTON FEATURES

The Pb detector uses 13 filters to extract features for textonisation. This filter bank is illustrated below in Figure 4-22:

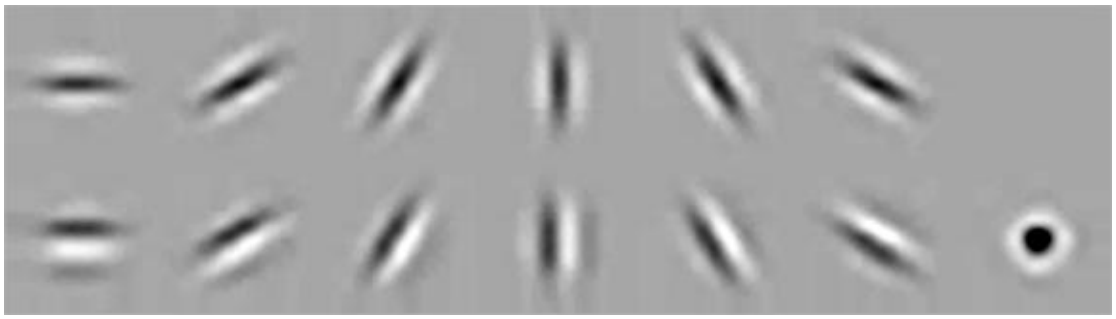


FIGURE 4-22: The filter bank used by the Pb detector. Reproduced from Martin *et al.* (2004)

This is different from the Winn-Criminisi-Minka (2005) filter bank shown in Figure 4-8 in a three main of ways. Firstly, there is only one scale of filters. Martin *et al*. (2004) found that multiscale integration was not necessary when the textons are combined with other visual cues. Secondly, all filters are oriented, except for the center-surround filter. This allows it to distinguish between different orientations of the same texture. Thirdly, there are no averaging filters (described in section 4.3.2), which means textons do not depend on colour. Instead, Pb integrates colour separately.

In the original paper, Pb was trained with $k = 64$ textons, using the k-means clustering algorithm as normal.

4.5.2 TEXTON GRADIENTS

Pb calculates texton gradients using a circular kernel split into two half-discs:

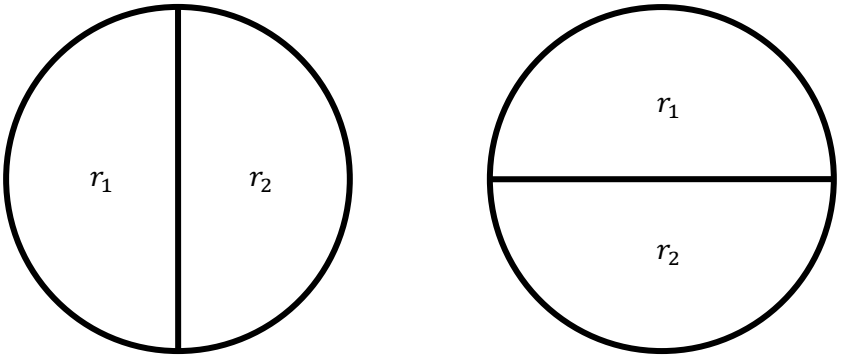


FIGURE 4-23: The Pb kernel, used to calculate gradients, shown at two orientations. The left one calculates horizontal gradients, the right one calculates vertical gradients.

As Figure 4-23 shows, the circular kernel can be oriented in different ways. At any particular orientation θ, the kernel is split into two regions, which will be denoted r_1^θ and r_2^θ.

The texton gradient for pixel $\boldsymbol{p}$ and orientation θ is calculated by the following process. The kernel is centered on pixel $\boldsymbol{p}$, which establishes which of the surrounding pixels belong to r_1^θ and r_2^θ. Now the frequency of each texton is counted in both regions. This generates two texton histograms, H_1^θ and H_2^θ, one each for regions r_1^θ and r_2^θ respectively. The texton gradient $\partial T(\boldsymbol{p};\theta)$ is equal to the chi-squared distance $\chi^2(\dots)$ between the two histograms:

$$\partial T(\boldsymbol{p};\theta) = \chi^2\left(H_1^\theta, H_2^\theta\right)$$
$$\chi^2(\boldsymbol{g},\boldsymbol{h}) = \frac{1}{2}\sum \frac{(g_i - h_i)^2}{g_i + h_i} \qquad \textbf{(4.10)}$$

The texton gradients are calculated at ω orientations. Martin *et al*. chose to use $\omega = 8$ orientations in their original paper.

4.5.3 RIDGE DETECTION

Normally, the texton gradient process will produce large, spatially-extended responses for each boundary. This is because texton gradients are built from a large area of support, and so a single texture boundary will influence a wide area. Ridge detection is needed to "thin" the responses so that they are better localised.

Ridges in the texton gradient $\partial T_R(\boldsymbol{p};\theta)$ are detected within each orientation θ separately, using this formulation:

$$\begin{aligned} \partial T_R(\boldsymbol{p};\theta) &= \frac{g(\boldsymbol{p})}{d(\boldsymbol{p})} \\ \text{where } g(p) &= \partial T(\boldsymbol{p};\theta) \\ \text{and } d(\boldsymbol{p}) &= -\frac{|\nabla g(\boldsymbol{p})|}{\nabla^2 g(\boldsymbol{p})} \end{aligned} \quad \textbf{(4.11)}$$

In equation (4.11) above, the function $g(\boldsymbol{p})$ is the texton gradient $\partial T(\boldsymbol{p};\theta)$ for one particular orientation θ. This has been defined to make the equations simpler.

The function $d(\boldsymbol{p})$ estimates the distance to the nearest ridge. The general concept is, dividing the texton gradient $g(\boldsymbol{p})$ by the distance $d(\boldsymbol{p})$ should cause the texton gradient to be emphasised infinitely on ridges, where $d(\boldsymbol{p}) = 0$.

In practice, a few modifications have to be made to the formulation presented in (4.11):

$$\partial T_R(\boldsymbol{p};\theta) = \frac{\mu_g(\boldsymbol{p})}{d(\boldsymbol{p}) + \epsilon} \quad \textbf{(4.12)}$$

In equation (4.12) above, a small value ϵ is added to the $d(\boldsymbol{p})$ to ensure computational divide-by-zero errors do not occur. Also, as $d(\boldsymbol{p})$ is merely a ridge estimate, some ridges may exist where $d(\boldsymbol{p}) > 0$, and so ϵ ensures these are still detected as ridges.

The smoothed texton gradient $\mu_g(\boldsymbol{p})$ is used in equation (4.12) to avoid the double response phenomenon. That is, it is normally a boundary causes two gradients, one on either side, instead of a single response on the actual boundary. Smoothing allows the response to occur directly on the boundary itself.

The function $\mu_g(\boldsymbol{p})$, can be calculated by using a Gaussian blur (section 3.1) of $g(\boldsymbol{p})$. The function $\nabla g(\boldsymbol{p})$ can be calculated by convolving $g(\boldsymbol{p})$ with the Gaussian derivative (section 2.4.1). Convolving $\nabla g(\boldsymbol{p})$ again with another Gaussian derivative will yield $\nabla^2 g(\boldsymbol{p})$. In their original paper, Martin *et al*. calculated $\mu_g(\boldsymbol{p})$, $\nabla g(\boldsymbol{p})$ and $\nabla^2 g(\boldsymbol{p})$ by fitting a parabola, but they also stated that the Gaussian method described here generated similar results.

After this, the Pb detector combines the texton ridges $\partial T_R(\boldsymbol{p};\theta)$ with other visual cues to improve the detection further.

4.5.4 COMBINING WITH OTHER VISUAL CUES

The texton gradient does not detect changes in colour, just texture. This can cause it to miss some important boundaries. For this reason, Pb also calculates a colour gradient, and combines it with the texton gradient.

Colour gradient

The colour gradient $\partial \mathrm{C}(\boldsymbol{p};\theta)$ is calculated in the same way as the texton gradient, except colour histograms, not texton histograms, are constructed. Like the texton gradient, the colour gradient is also equal to the chi-squared distance between the histograms.

Martin *et al*. also experimented with ridge detection on the colour gradient, but found it did not make any difference. Consequently, ridge detection is only performed on the texton gradient.

Logistic regression model

The texton gradient and colour gradient are combined using a logistic regression model. A logistic regression model takes a weighted sum of its inputs, then transforms them using a logistic function:

$$Pb(\boldsymbol{p};\theta) = Logistic\big(w_0 + w_c\,\partial \mathrm{C}(\boldsymbol{p};\theta) + w_t\,\partial T_R(\boldsymbol{p};\theta)\big)$$
$$\text{where } Logistic(x) = \frac{1}{1+e^{-x}} \qquad \textbf{(4.13)}$$

This model is illustrated in Figure 4-24:

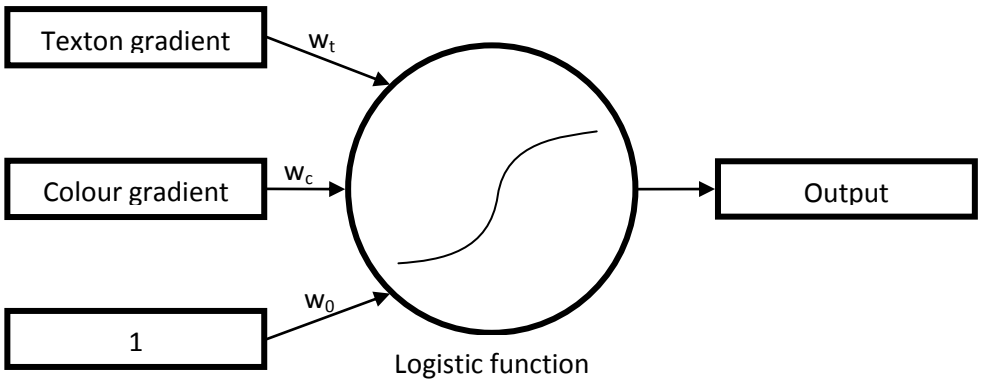


FIGURE 4-24: An illustration of Pb's logistic regression model.

The logistic function in particular has been chosen because it works as a soft-threshold function – it does not hard-assign input values to either zero or one, but instead will soft-assign them to a real number between zero and one. Figure 4-25 illustrates this with a plot of $Logistic(x)$ for various values of x.

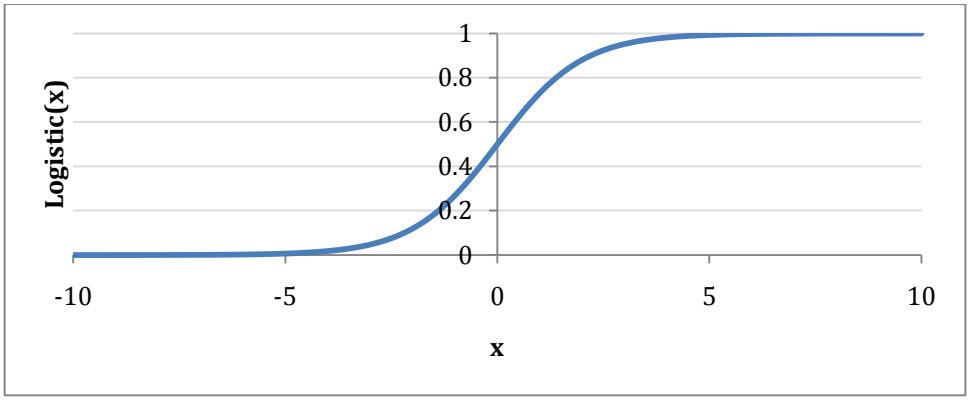


FIGURE 4-25: The logistic function.

Notice in Figure 4-25 above, the logistic function is almost zero for $x \ll 0$, and it is almost one for $x \gg 0$. In the area close to $x = 0$, the logistic function gradually changes from zero to one. This behaviour makes the logistic function work like a soft-threshold function.

As Figure 4-14 shows, the logistic regression model used by the Pb detector takes three inputs: the texton gradient, the colour gradient, and a constant value of 1. The weighted constant input is needed because it allows the threshold level of the logistic function to be set.

The Pb detector first learns the optimal logistic function model from the 200 training images of Berkeley dataset (which will be detailed later in section 8.1). Learning is possible because each image in the Berkeley dataset has a human-defined ground truth.

In the training stage, the texton and colour gradients are first calculated on each of the training images. Then, using these as inputs, the weights of the logistic regression model are optimised so that the output of the model best matches the ground truth. Newton-Raphson's method is used for the optimisation process.

The output of the logistic regression model is the final boundary map for the image.

4.5.5 IMAGE EXAMPLES

The Pb detector produces results such as the ones in Figure 4-26:

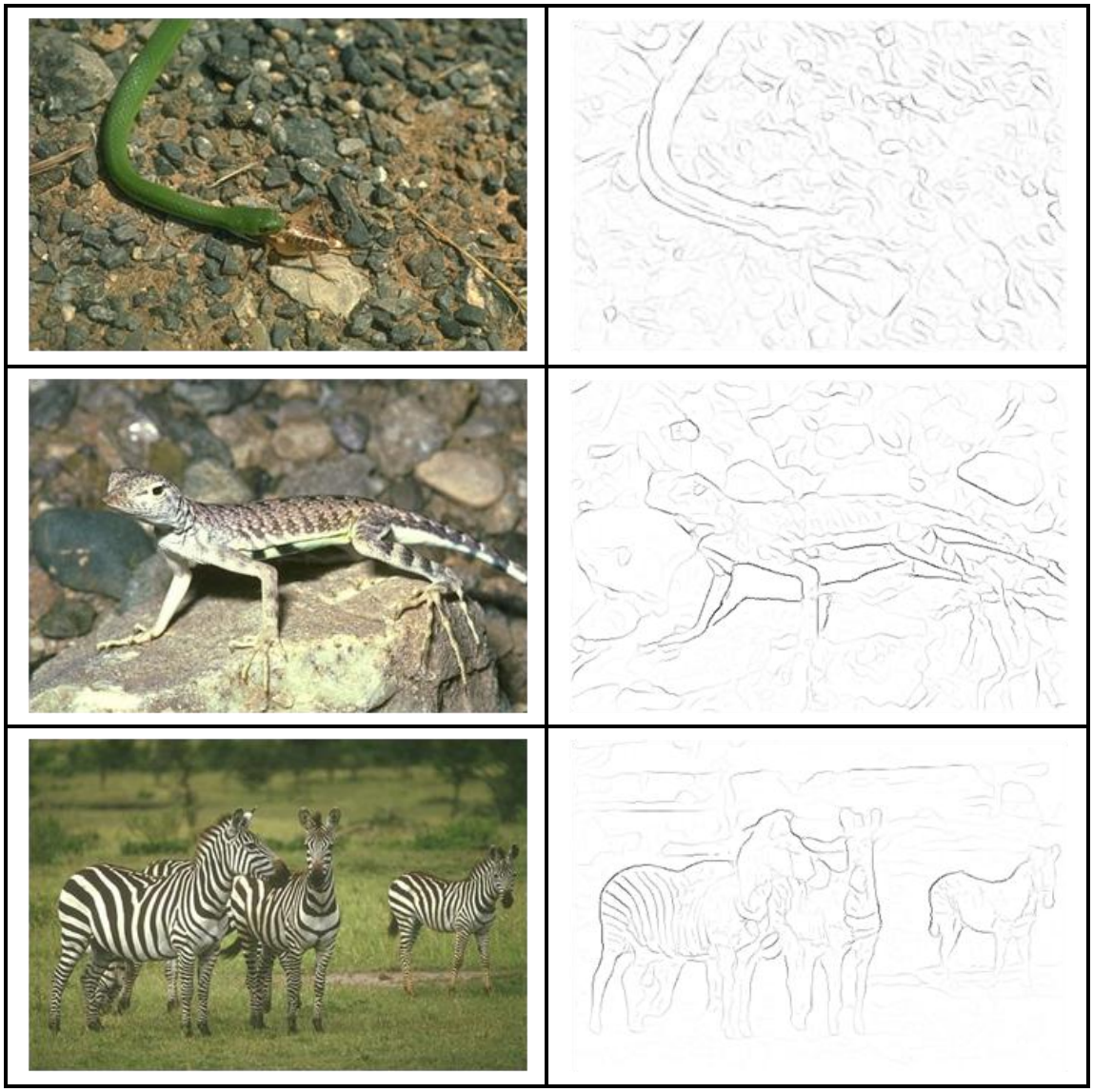


FIGURE 4-26: A series of results from the probability of boundary (Pb) detector. Original images are on the left, and their boundary maps are on the right. Original images are from the Berkeley dataset (Martin *et al.*, 2001).

The images above in Figure 4-26 show that the Pb detector is able to suppress most of the smaller scale textures (such as grass), but it still struggles with some of the larger scale textures (such as the zebra stripes). This is because, unlike normalised cut segmentation, mean-shift segmentation and TextonBoost, the Pb detector only interprets the image at the local level, as opposed to the image level. The next section will discuss the global probability of boundary detector, which integrates the normalised cut with the Pb detector so that it can handle these larger-scale textures as well.

Another problem with the Pb detector is that it cannot run in real-time. It requires thirteen convolutions for each of the thirteen filters in its filter bank, and then its ridge

detection stage requires three convolutions per orientation. Any algorithm with this many convolutions simply cannot run in real-time.

4.6 GPB: THE GLOBAL PROBABILITY OF BOUNDARY DETECTOR

The global probability of boundary (gPb) detector (Maire, Arbelaez, Fowlkes, & Malik, 2008) is essentially a modified normalised cut (section 4.1), with two differences.

The normalised cut traditionally uses gradient magnitude for its similarity score (as described in section 4.1.2). The gPb detector instead uses the result of the probability of boundary detector. This gives the normalised cut algorithm texture-awareness.

Also, the normalised cut traditionally splits the image into only two segments because it only uses the information from one of the eigenvectors. The gPb detector modifies this stage so that more than two segments can be detected. It does this by taking the gradient magnitudes for the first k eigenvectors, and summing them together. This works because as section 4.1.5 stated, the higher eigenvectors contain information on further subdivisions of the image. Maire *et al*. set the parameter $k = 8$ in their original paper. The gradient magnitude is used because it avoids hard-assigning each pixel into two classes like the traditional binarising process. The gradients can be calculated by convolving with Gaussian derivatives (already explained in 2.4.1).

All other stages of the gPb detector are identical to the normalised cut.

4.6.1 IMAGE EXAMPLES

The gPb detector produces results such as shown in Figure 4-27 on some example images:

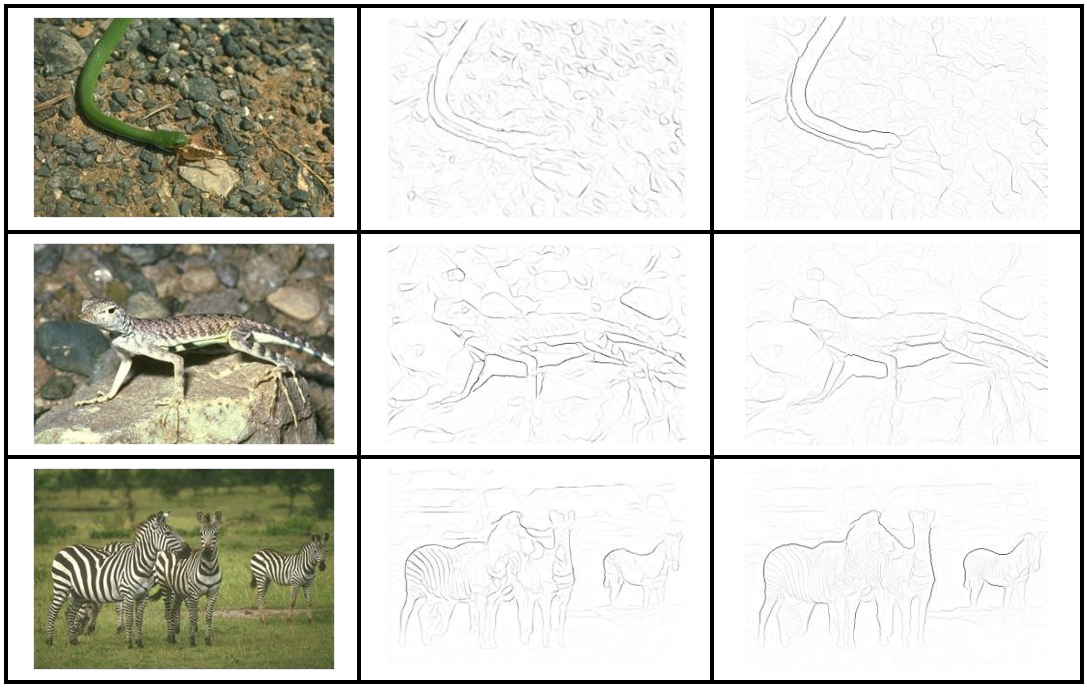


FIGURE 4-27: The gPb detector (right) versus the Pb detector (middle) on some example images (left). Original images are from the Berkeley dataset (Martin *et al.*, 2001).

Notice in the images above that the global probability of boundary detector has produced a much higher-quality image interpretation because it integrates the information at a global scale.

The global probability of boundary detector is one of the best boundary detectors currently. It has the highest score of all algorithms on the Berkeley benchmark (described later in section 8.1). Unfortunately, as gPb is a combination of two already non-real-time algorithms – normalised cut segmentation and the Pb detector, it is not possible for it to run in real-time.

4.7 CHAPTER SUMMARY

This chapter has discussed the inner workings of five excellent texture-boundary detectors: normalised cut segmentation, mean-shift segmentation, TextonBoost, the probability of boundary detector, and the global probability of boundary detector.

Normalised cut segmentation produces optimal results according to an objective function, but it suffers from the "broken sky" problem.

Mean-shift segmentation avoids the "broken sky" problem, but it sometimes finds intra-texture boundaries as opposed to inter-texture boundaries because it has no explicit understanding of texture.

TextonBoost explicitly learns and models texture, but it is always limited to the textures it is trained on, and so can only really be used in controlled conditions.

The probability of boundary detector does not have this limitation, but it does not integrate information at the image level and so cannot handle large-wavelength textures.

The global probability of boundary detector combines the normalised cut with the probability of boundary detector to make the highest-quality boundary detector according to the Berkeley benchmark (section 8.1).

Unfortunately, none of these high-quality approaches can run in real-time. This shows that texture-boundary detection as a whole is a difficult problem, and it is best solved without time constraints. Consequently, very few texture-boundary detectors are able to run in real-time. This is a problem because it means real-time applications cannot benefit from the state-of-the-art in texture-boundary detection. The next chapter will investigate five real-time texture-boundary detectors which attempt to bring texture-boundary detection to real-time.

5 REAL-TIME TEXTURE-BOUNDARY DETECTION

This chapter will investigate five existing real-time texture boundary detectors:

- Konishi's detector (section 5.1)
- Surround Suppression (section 5.2)
- TextonRML (section 5.3)
- Semantic Texton Forests (section 5.4)
- Randomised Hashing (section 5.5)

These detectors are relevant for two reasons. Firstly, the detectors that will be proposed later by this thesis use similar processes to some of these other real-time detectors. Secondly, each of the existing real-time detectors has its problems, which provides a motivation for the development of new real-time texture-boundary detectors.

5.1 KONISHI'S DETECTOR

Konishi's detector (Konishi, Yuille, & Coughlan, 2002) is the boundary detector counterpart of the Nitzberg operator, previously introduced in section 3.3. It works by detecting a boundary wherever the local gradients are all similarly oriented, or in other words, wherever they are highly **coherent**. This detects boundaries quite accurately, and is able to suppress intra-texture boundaries to some level of success.

Recall from section 3.3.2.2 that Nitzberg's operator calculates the average local gradient orientation $GradientOrientation(\boldsymbol{p})$ at each pixel $\boldsymbol{p}$ according to equation (3.6). Konishi's detector calculates the local coherence from the result of that function. This is possible because $GradientOrientation(\boldsymbol{p})$ returns a structure tensor.

The rest of this section will explain how coherence can be calculated, using a pedagogical example.

Example definition

Normally, there would be many gradients of varying strengths in the local sliding window, but for this example let there be only two gradients of equal strength in the local neighbourhood of pixel $\boldsymbol{p}$:

Gradient A: 180° Gradient B: 45°

FIGURE 5-1: Two example gradient orientations.

Now the average gradient orientation $A = GradientOrientation(\boldsymbol{p})$ will be equal to the average structure tensor of these two gradients.

$$A = \frac{\varphi\begin{pmatrix}\cos 180° \\ \sin 180°\end{pmatrix} + \varphi\begin{pmatrix}\cos 45° \\ \sin 45°\end{pmatrix}}{2} = \begin{bmatrix}0.75 & 0.25 \\ 0.25 & 0.25\end{bmatrix} \quad \textbf{(5.1)}$$

Previously, section 3.3.2.1 showed that a structure tensor would transform a circle into an ellipse, where the ellipse's major axis will be parallel to the average orientation, and the ratio of the major and minor axis of the transformed ellipse will measure coherence. Structure tensor A will transform a circle of radius 1 into the ellipse in Figure 5-2:

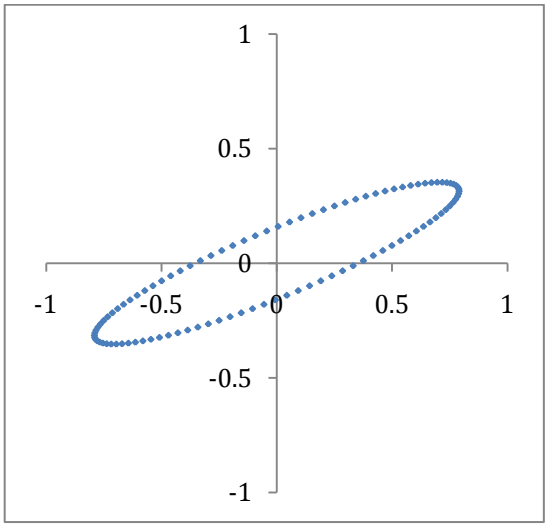


FIGURE 5-2: The transformed ellipse from structure tensor A.

As expected, the orientation of the ellipse in Figure 5-2 above is equal to the average orientation of the two vectors illustrated in Figure 5-1. Coherence requires that the lengths of the major and minor axis of this ellipse be calculated. This can be done by calculating the eigenvectors of the structure tensor A.

Eigenvectors

A structure tensor is a transformation matrix. A transformation matrix can have zero or more eigenvectors. An **eigenvector** is a vector that will not change direction under that

particular transformation. It is a fact that, for a particular structure tensor, the only places eigenvectors can occur are on the major and minor axes of the transformed ellipse. This makes sense, as all other vectors will change direction. See Figure 5-3 below for an illustration of this.

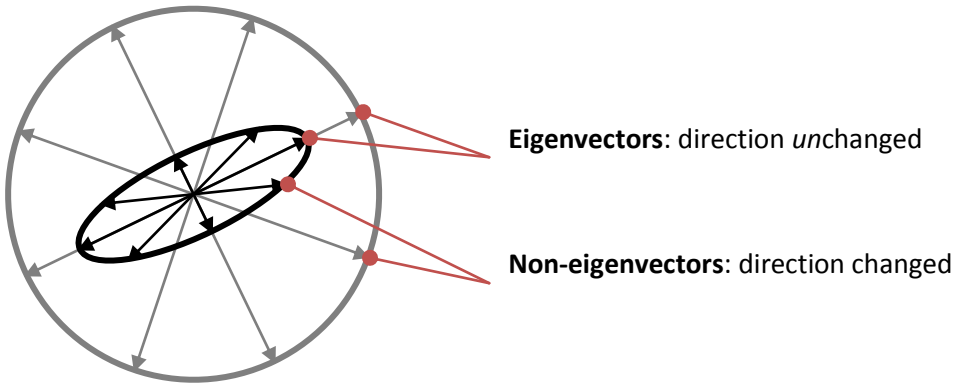


FIGURE 5-3: The outer circle (of radius 1) is transformed to the inner ellipse by a structure tensor. Notice that the vectors on the major and minor axes do not change direction – that means they are eigenvectors.

So finding the eigenvectors of a matrix and finding the axes of an ellipse are equivalent problems.

Eigenvalues

Coherence is only interested in the *lengths* of the major and minor axes, which means the actual eigenvectors are not required. That means, only the **eigenvalues** need to be calculated.

Eigenvectors are not allowed to change direction, but they are allowed to change in length. The factor at which an eigenvector will be scaled by a particular transformation is called its **eigenvalue**. As a structure tensor is a linear transformation, all eigenvectors on the same axis will be scaled by the same amount – that is, they will all have the same eigenvalue. That means, the major axis will have one eigenvalue, and the minor axis will have another eigenvalue. So the axial lengths of the ellipse are proportional to the eigenvalues.

If a circle of radius 1 is transformed by a structure tensor, then it will generate an ellipse with axial lengths equal to the eigenvalues because 1 is the multiplicative identity. So, for simplicity, a circle of radius 1 is always used. Now the coherence is simply the ratio of the eigenvalues of the structure tensor.

Eigenvalues of the example structure tensor

An eigenvalue λ of transformation matrix A is a value that satisfies the condition:

$$|A - \lambda I| = 0 \tag{5.2}$$

In the above equation, I is the identity matrix, and $|\dots|$ is the determinant of a matrix.

Now the eigenvalues for structure tensor A can be found by solving equation (5.2). First the equation is rearranged into a quadratic:

$$\begin{aligned} |A - \lambda I| &= 0 \\ \begin{vmatrix} 0.75 - \lambda & 0.25 \\ 0.25 & 0.25 - \lambda \end{vmatrix} &= 0 \\ (0.75 - \lambda)(0.25 - \lambda) - 0.25 \times 0.25 &= 0 \\ \lambda^2 - \lambda + 0.125 &= 0 \end{aligned} \tag{5.3}$$

Now it can be solved with the quadratic formula:

$$\begin{aligned} \lambda &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ \lambda &= \frac{-(-1) \pm \sqrt{(-1)^2 - 4(1)(0.125)}}{2(1)} \\ \lambda &= \frac{1 \pm \sqrt{0.5}}{2} \\ \lambda &= 0.8536 \text{ or } 0.1464 \text{ (both 4 s.f.)} \end{aligned} \tag{5.4}$$

Now the two eigenvalues of the structure tensor are known. Another way of seeing this is, if structure tensor A were to transform a circle of radius 1, the resulting ellipse would have a major axis length of 0.8536 and a minor axis length of 0.1464. These values can now be used to calculate coherence.

Coherence

Given a structure tensor A, which has two eigenvalues λ_1 and λ_2, the normalised coherence measure (Weickert J. , 1999) is defined as:

$$Coherence(A) = \frac{(\lambda_1 - \lambda_2)^2}{(\lambda_1 + \lambda_2)^2} \tag{5.5}$$

The function $Coherence(A)$ will always return values in the range [0, 1].

Eigenvalues to boundaries

Instead of actually calculating coherence itself, Konishi's detector works directly on the eigenvalues of $GradientOrientation(\boldsymbol{p})$. The detector is essentially a classifier which, for each pixel, takes the two eigenvalues as input and then outputs either "boundary" or "non-boundary". The most recent implementation of Konishi's detector (Martin, Fowlkes, & Malik, 2004) used a logistic regression model as the classifier. This classifier

was trained from a training set of human-labelled images in the same way as it was for the Pb detector (see section 4.5.4).

5.1.1 IMAGE RESULTS

Some results of Konishi's detector are shown below in Figure 5-4:

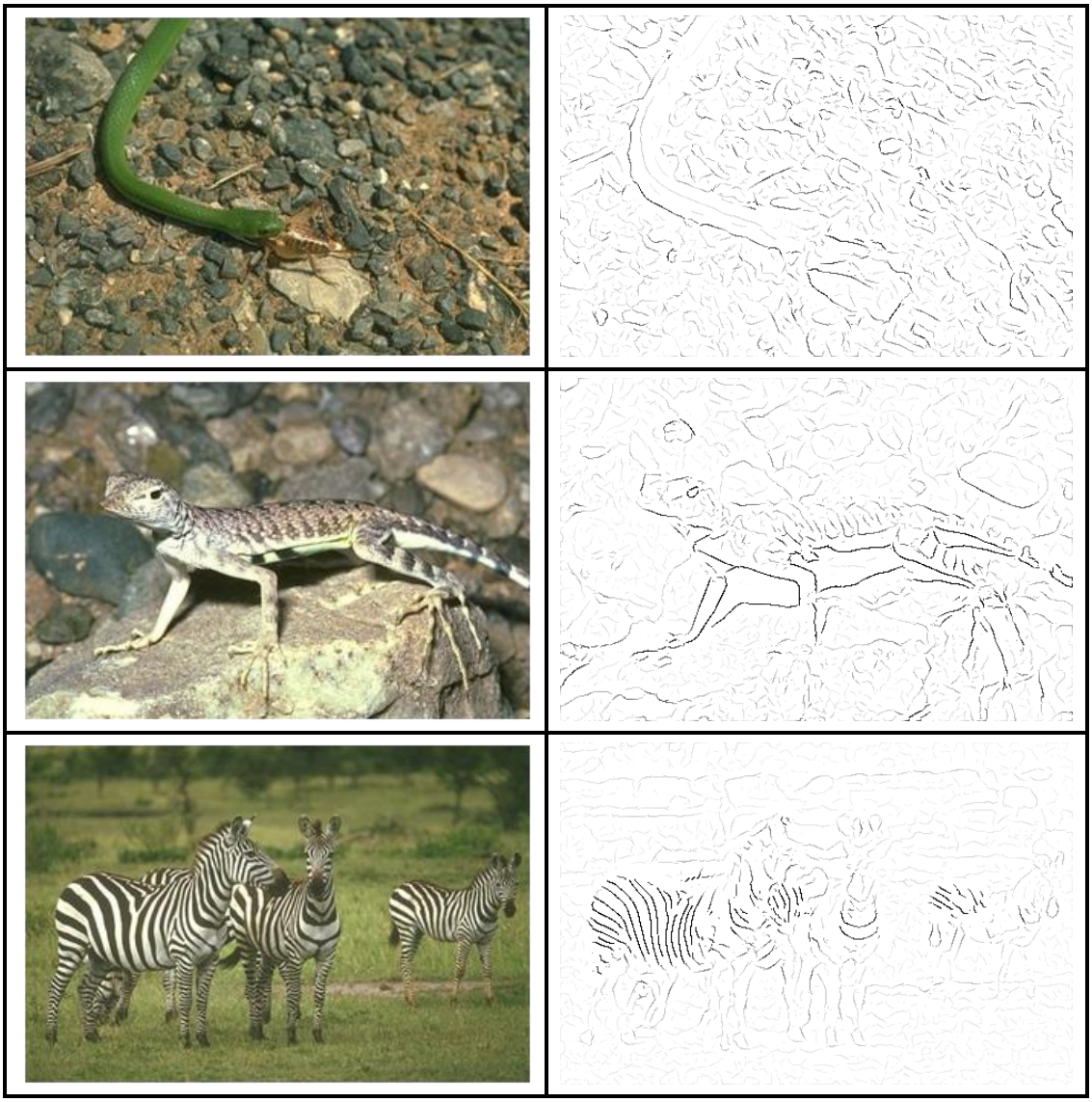


FIGURE 5-4: Some results of Konishi's detector. Original images are from the Berkeley dataset (Martin *et al.*, 2001).

5.1.2 CRITIQUE

Although Konishi's detector generally detects boundaries well, it has limited texture-suppressing ability, as it can only suppress the textures that have low coherence. This works well for textures such as grass, but not for strongly-oriented textures like zebra stripes, as illustrated in Figure 5-4. The reason for this is, by definition, strongly-oriented textures will consist of highly coherent gradients. This leaves much room for

improvement. Surround Suppression, which will be introduced next, does not have this problem.

5.2 SURROUND SUPPRESSION

Surround Suppression (Grigorescu, Petkov, & Westenberg, 2003; 2004) is a modification to the Canny edge detector that attempts to remove intra-texture edges. This section will describe the most basic version of Grigorescu *et al.*'s Surround Suppression algorithm. Other variations of the Surround Suppression algorithm exist, but all of them are based on the same concept.

The concept is quite simple. An intra-texture edge is likely to be surrounded by many other intra-texture edges of the same strength, simply because of the fact that texture is a pattern that repeats itself. So, if a gradient is of similar strength to its surrounding gradients then it should be suppressed.

5.2.1 FORMULATION

For the most part, Grigorescu *et al.*'s algorithm is exactly the same as the Canny edge detector (already described in section 2.4), except a new step called "Surround Suppression" has been added.

Let the Surround Suppression kernel $SS(\boldsymbol{p}')$ be defined as follows:

$$\begin{aligned} SS(\boldsymbol{p}') &= \frac{1}{Z} SS^z(\boldsymbol{p}') \\ \text{where } SS^z(\boldsymbol{p}') &= [G_{4\sigma}(\boldsymbol{p}') - G_{\sigma}(\boldsymbol{p}')]^+ \\ \text{and } Z &= \sum_{\boldsymbol{p}'} SS^Z(\boldsymbol{p}') \end{aligned} \tag{5.6}$$

The function $G_{\sigma}(\boldsymbol{p}')$ is the Gaussian function with scale σ, as defined previously in equation (3.2). The scale σ is an external parameter which is set by the user.

The Surround Suppression kernel can be used to find a weighted average of the surrounding region. It is used to calculate the surround potential $SP(\boldsymbol{p})$, which is then subtracted from the gradient magnitude to generate the edge potential $EP(\boldsymbol{p})$:

$$\begin{aligned} EP(\boldsymbol{p}) &= \|\nabla I(\boldsymbol{p})\| - SP(\boldsymbol{p}) \\ \text{where } SP(\boldsymbol{p}) &= (\|\nabla I\| * SS)(\boldsymbol{p}) \end{aligned} \tag{5.7}$$

The algorithm then proceeds in exactly the same way as the Canny edge detector, except now it finds ridges in the edge potential $EP(\boldsymbol{p})$ instead of the gradient magnitude $\|\nabla I(\boldsymbol{p})\|$.

At the conceptual level, Surround Suppression works exactly as its name implies. Each gradient is suppressed by the average surrounding gradient. The result is that some intra-texture edges are suppressed.

5.2.2 IMAGE RESULTS

Grigorescu *et al.*'s 2004 paper shows some good results, reproduced in Figure 5-5:

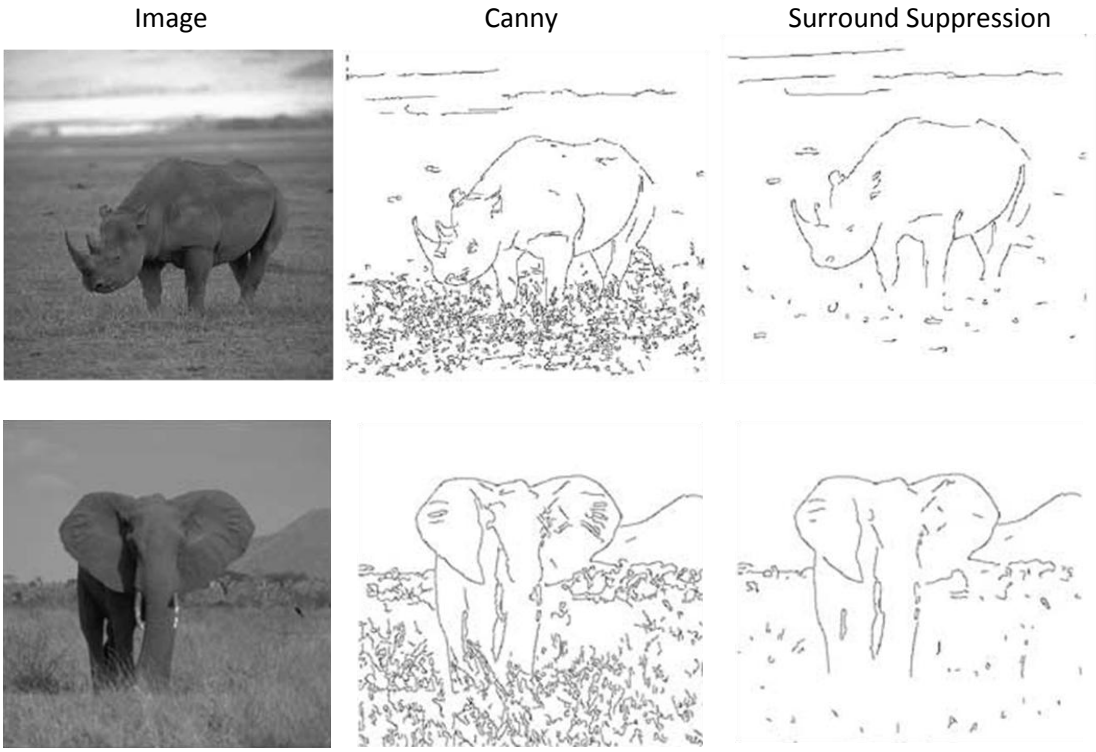


FIGURE 5-5: Surround Suppression (right) versus the Canny edge detector (middle) on some example images (left). Adapted from Grigorescu *et al.* (2004).

5.2.3 CRITIQUE

Surround Suppression is extremely fast and suppresses most texture edges. It is not confused by strongly-oriented textures like Konishi's detector is. In fact, Grigorescu et al.'s original paper (2003) presents some modifications to Surround Suppression that are specifically designed to handle strongly-oriented textures well.

Unfortunately, Surround Suppression still has some problems. As Figure 5-5 shows, Surround Suppression sometimes produces fragmented boundaries, and the grass texture has not been completely suppressed. These problems occur because Surround Suppression is based on edge detection, and so it focuses on the low-level interpretation of the image. The next section describes TextonRML, a method which uses high-level analysis in an attempt to avoid these problems.

5.3 TEXTONRML

Section 4.5.4 described logistic regression models in the context of the probability of boundary detector. A logistic regression model is useful because it is fast, but its main problem is it can only distinguish between two classes. **Multinomial logit** is an extension of logistic regression to more than two classes. **Random multinomial logit** (RML) is a way to combine an ensemble of multinomial logit models together to improve classification accuracy. Ranganathan (2009) replaced the boosting stage in TextonBoost with random multinomial logit, making it much faster. This new algorithm will be referred to as TextonRML throughout this thesis.

In this context, the term **classifier** refers to the algorithm used to soft-assign pixels to textures – in this section the classifier will either be boosting (for TextonBoost) or random multinomial logit (for TextonRML).

In the same way as TextonBoost, TextonRML uses texture-layout filters as input to the classifier. Also in the same way as TextonBoost, after the classifier generates a soft-assignment of pixels to textures, the hard-assignment is found using alpha-expansion graph cuts. The only difference between TextonBoost and TextonRML is the classifier. Intuitively, because the classifiers are different, the training stages are also different. Each of these points will be discussed in turn.

5.3.1 RANDOM MULTINOMIAL LOGIT

Like all classifiers, random multinomial logit takes a number of features as inputs, and then outputs the predicted class for those inputs. Similar to logistic regression, the inputs are combined using weighted sums, and so a weight must be learnt for each input feature during the training process. The weights can be learnt from a training set using well-known gradient descent methods. This is all that is necessary to understand random multinomial logit at a high level, see Ranganathan's (2009) paper for details.

5.3.2 FEATURE SELECTION

Like TextonBoost, TextonRML must learn a good set of texture-layout filters to use as input features to the classifier. Let N be the number of features that need to be learnt by the training process. Both boosting and random multinomial logit divide their training process into rounds. However, what they do in each round is different.

As section 4.4.2 described, one round of boosting will generate one additional texture-layout filter to classifier, and so *N* rounds of boosting will generate *N* texture-layout filters.

In contrast, random multinomial logit initially begins with a random set of *N* features. Each round of RML training will incrementally improve this random set.

At the beginning of each round, one of the texture-layout filters is chosen to be replaced by a new randomly-generated texture-layout filter. A new random multinomial logit model is now trained on the new set of features. The accuracy of this new classifier is compared to the previous classifier, and only the classifier of highest accuracy is kept.

Choosing which filter to replace in each round is an important process, and it works as follows. If some of the texture-layout filters have small weights, then that means they do not contribute much, and so one of them will be replaced. However, if none of the texture-layout filters have small weights, then a random one will be chosen to be replaced.

Eventually after many rounds, this yields an accurate random multinomial logit classifier which can be used to soft-assign pixels to the textures. All other parts of TextonBoost's algorithm remain the same, so see section 4.4 for more details.

5.3.3 IMAGE RESULTS

Ranganathan's 2009 paper publishes some results of TextonRML, reproduced in Figure 5-6 below.

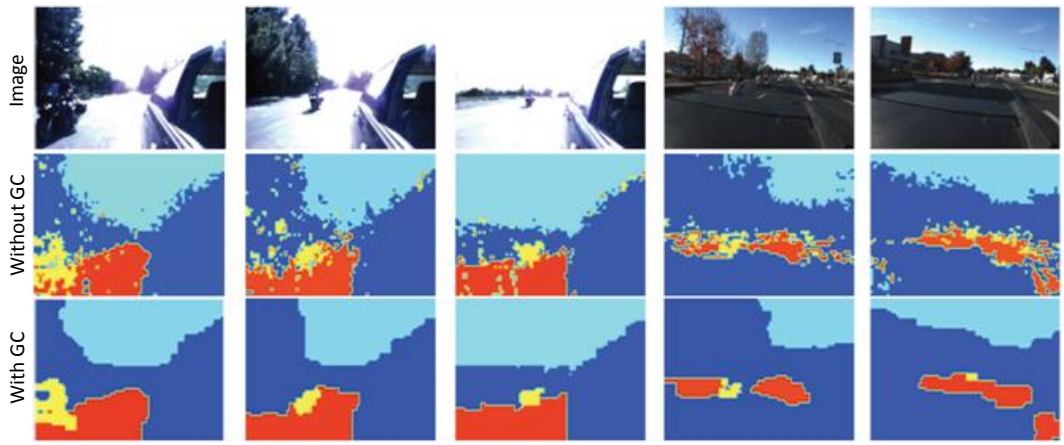


FIGURE 5-6: TextonRML applied to some images, both with and without alpha-expansion graph cuts (denoted GC). Reproduced from Ranganathan (2009).

Notice from Figure 5-6 the clear improvement in results when an alpha-expansion graph cut is used.

5.3.4 CRITIQUE

When the alpha-expansion graph cut is used, TextonRML produces excellent results. This is because it is generating an image-level interpretation of the image, unlike Surround Suppression and Konishi's detector.

Unfortunately, TextonRML can only run in real-time when the alpha-expansion graph cut stage is omitted. As Figure 5-6 shows, when the graph cut is omitted, the pixel classifications are generally correct, but the boundaries between them are quite noisy. The reasoning for this is the same as it was for TextonBoost – texture needs to be smoothed for the boundaries to be useful. This was already described fully in the section titled "Why not just hard-assign a pixel to its modal texture?" (section 4.4.3). Without an alpha-expansion graph cut stage, TextonRML's real-time boundary maps are inadequate.

Semantic Texton Forest segmentation, described next, is another algorithm that interprets the image at the image level, but instead of omitting the alpha-expansion graph cut stage entirely like TextonRML, it substitutes it with a real-time approximation.

5.4 SEMANTIC TEXTON FORESTS

The Semantic Texton Forests (Shotton J. , Winn, Rother, & Criminisi, 2009) algorithm was developed by the same research group as TextonBoost (section 4.4) and is considered to be its successor. Semantic Texton Forests (STF) segmentation achieves the same purpose as TextonBoost – that is, it performs simultaneous segmentation and texture recognition, but its approach is quite different. STF segmentation can be divided into three major stages:

1. Textonisation using Semantic Texton Forests. This replaces TextonBoost's textonisation stage, which used convolution and k-means clustering (section 4.3.4).
2. Texture classification. At this stage, the pixels are soft-assigned to textures. This is almost identical to TextonBoost's boosting stage (section 4.4.2) except the classifier is different.
3. Improving boundary detection with image categorisation. This replaces TextonBoost's alpha-expansion graph cut stage (section 4.4.5).

Each of these stages will be discussed separately.

5.4.1 TEXTONISATION WITH DECISION FORESTS

Semantic textons are different from the normal concept of textons introduced in section 4.3, but ultimately they fill the same purpose. Semantic textons are calculated using decision forests. This section will describe how this is done.

Decision forests

A decision tree is a classifier that soft-assigns observations to classes based on simple decisions. In Figure 5-7, a decision tree is used to determine the class a pixel belongs to.

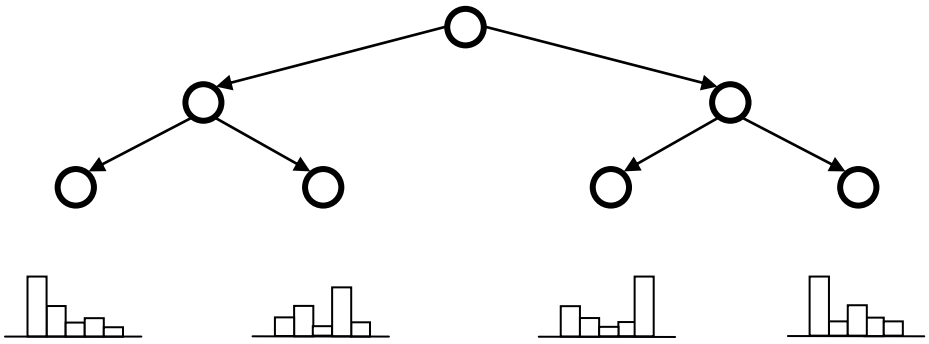


FIGURE 5-7: A decision tree. Each node represents one simple decision, where an observation will choose to proceed to either the left or right node based on some simple criteria. The histograms at the bottom indicate the soft-assignments given to observations which reach each of the respective leaf nodes.

To train a decision tree, first there must be a training set of classified observations. Next, a wide range of random decision rules are tried, and the decision which splits the training set in the "most informative" way is chosen. Normally, the "most informative" decision rule is the one that creates the purest split between the classes. The pureness of a split can be measured using Shannon entropy.

The chosen decision rule will split the data into two subsets. Now, each subset is subdivided with the same process. The algorithm stops subdividing when a desired level of classification accuracy is reached. Once the decision tree is constructed, the soft-assignment for each leaf-node can be determined by running every training observation through the decision tree, and then counting the mixture of classes that ends up at each leaf node.

A decision forest is simply an ensemble of decision trees. Each tree in a decision forest is trained on a different subset of the training data, which helps avoid overfitting.

Semantic textons forests

A semantic texton forest is just a decision tree that has specifically been trained to classify a sliding window to one of many classes of texture. Semantic Texton Forests are trained from the same human-labelled training input as TextonBoost (see section 4.4).

Each decision node in a Semantic Texton Forest will compare one of four different features against a decision threshold. The four features are: (1) the colour of one of the pixels in the sliding window, or (2) the sum, (3) difference, or (4) absolute difference between two pixels in the sliding window. There are many random variations on these four decision formats, which allows highly-discriminative decision trees to be constructed.

Semantic textons

A **semantic texton** is one of the nodes in the decision tree. That means, each sliding window can be described by a set of many semantic textons, instead of just one texton like in the normal texton approach (discussed previously in section 4.3). So for example, if there are ten decision trees, each ten levels deep, then every sliding window can be described by one hundred semantic textons. This makes semantic textons quite different from normal textons, but at the same time, both semantic textons and normal textons achieve the same purpose. That is, they provide the information required to distinguish between textures.

5.4.2 SEGMENTATION

To perform boundary detection, STF segmentation begins by calculating the semantic textons for every sliding window. From here, the next step is to recognise the textures from those textons.

Even though the Semantic Texton Forests are already capable of soft-assigning each pixel to a texture, STF segmentation works by using the same texture-layout filters used by TextonBoost (described in section 4.4.2). The reason for this is, the Semantic Texton Forests used to classify each sliding window only work at the low-level, using local information. Using texture-layout filters adds some mid-level context to the soft-assignment process, making STF segmentation more accurate.

The texture-layout filters work exactly the same way as with TextonBoost, except the boosting classifier is replaced with a decision forest classifier – similar to how boosting was replaced with random multinomial logit in section 5.3.

This generates a soft-assignment of pixels to textures. Next, the hard-assignment must be calculated.

5.4.3 IMAGE CATEGORISATION

TextonBoost uses alpha-expansion graph cuts to hard-assign each pixel to a texture. As section 4.4.5 described, the reason this is needed is that, if the modal texture were simply taken, then the result would be noisy and the resulting boundary detection would not be useful.

The problem is that alpha-expansion graph cuts are slow, and definitely cannot run in real-time. So, STF segmentation applies something called an image category prior. This will be described next.

Theory

Different categories of images contain different textures. For example, outdoor images are likely to contain trees and grass, while indoor images are likely to contain desks and chairs. Therefore, if the image category is known, then it can be used to suppress the unlikely textures. It turns out that this makes the modal texture much more useable, albeit not as good as an alpha-expansion graph cut. However, some loss in quality must be expected for an algorithm that is constrained to real-time.

Automatic image categorisation algorithm

A classifier is used to recognise an image's category from its semantic texton histogram. Generating the semantic texton histogram is straightforward – it is simply the frequencies of each semantic texton tallied over the entire image. The most difficult part of this stage is the classifier. Shotton *et al*. chose to use a multi-class support vector machine, which will not be described here as it is beyond the scope of this thesis. The end result though, is that the system can automatically determine what category an image belongs to.

So, in the training stage, STF segmentation will count the frequencies of each texture for each image category from the training set. Then in the online stage, it will modify the soft-assignment for each pixel so that the unlikely textures for an image's category are suppressed. This allows each pixel to be hard-assigned to its modal texture with adequate results.

5.4.4 IMAGE RESULTS

STF segmentation can only run in real-time when it does not have to evaluate every possible sliding window in the image. Instead, the algorithm is only run on the cells of a grid, where each grid cell is 21×21 pixels large. This generates a downsampled boundary map, as shown in Figure 5-8:

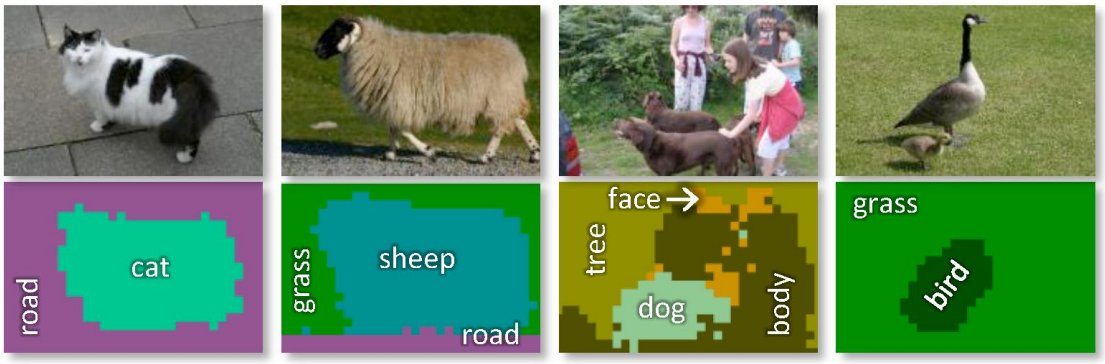


FIGURE 5-8: The results of Semantic Texton Forest segmentation. Reproduced from Shotton *et al.* (2009).

5.4.5 CRITIQUE

As Figure 5-8 shows, STF segmentation produces excellent real-time image labelings. However, the focus of this thesis is on boundary detection. STF segmentation can only generate low-resolution boundary maps in real-time. This is inadequate for most real-time applications. For example, in the context of face recognition, different faces look the same at low resolution. Or in real-time tracking, the trajectory of the object cannot be predicted at such a low resolution.

Boundary detection via Randomised Hashing, which will be introduced next, generates full-resolution boundary maps unlike STF segmentation. It also generates a high-level interpretation of the image, unlike Konishi's detector and Surround Suppression.

5.5 RANDOMISED HASHING

Boundary detection via Randomised Hashing (Taylor & Cowley, 2009) is similar in spirit to mean-shift segmentation (section 4.2) in that it finds boundaries by clustering the pixels in the image. Mean-shift segmentation however, is highly iterative and so is unable to run in real-time. Randomised Hashing has been designed so that its clustering is non-iterative which makes it able to run in real-time.

5.5.1 ALGORITHM

Randomised Hashing begins by extracting a feature vector $\Phi(\boldsymbol{p})$ from each pixel $\boldsymbol{p}$. Taylor and Cowley (2009) chose to use the RGB values as the feature vector for a pixel.

The feature space is subdivided by n randomly-chosen hyperplanes, where n is a user-specified parameter. These hyperplanes will subdivide the feature space into at most 2^n partitions.

Each partition is given a unique binary **partition code** of length n, where each bit of the code is determined by a different hyperplane. A hyperplane's bit will be set to either zero or one, depending on which side of the hyperplane the partition lays. For example, consider the partition labelled "0110" in Figure 5-9. Its first bit is "0" because the partition is on the right side of the first hyperplane s_0. Its second bit is "1" because the partition is on the left side of the second hyperplane s_1. The other bits are calculated in a similar way.

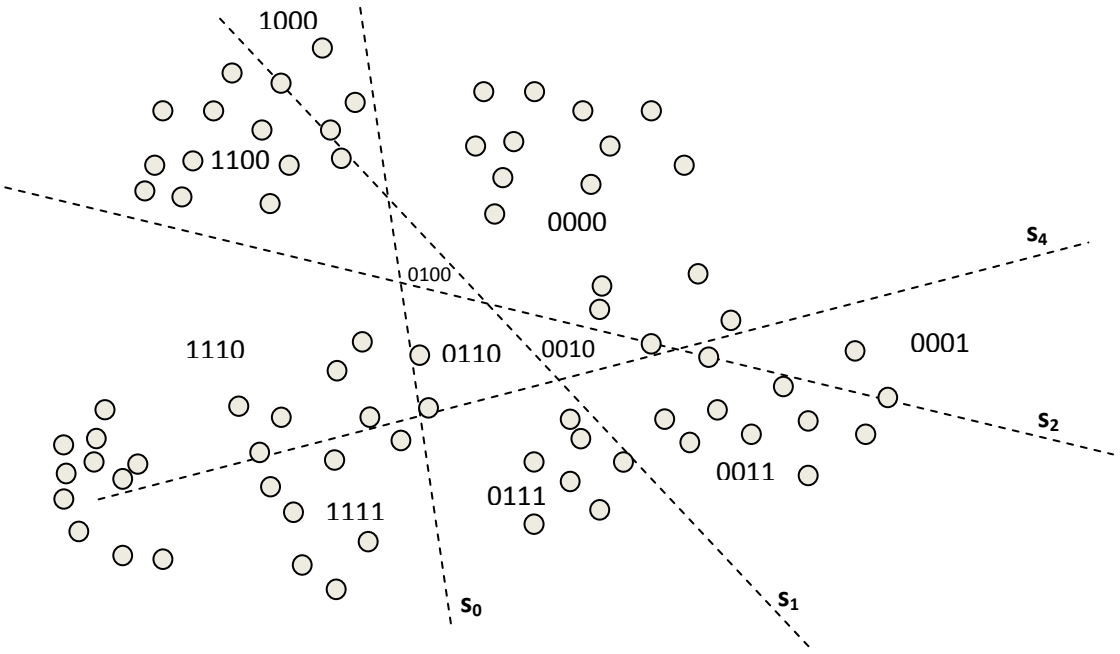


FIGURE 5-9: This diagram represents a hypothetical two-dimensional feature space. Each point represents the feature vector for a particular pixel. The feature space has been subdivided by hyperplanes, and a partition code has been assigned to each partition. Diagram adapted from Taylor and Cowley (2009).

The **neighbours** of each partition can be found using the partition codes. A partition is considered to be a **neighbour** of another partition if their partition codes differ by at most k bits. Now, clustering can begin.

Randomised Hashing will count how many of an image's feature points have been assigned to each of the partitions. Naturally, some partitions will have more feature

points than its neighbours. In Randomised Hashing, each partition is assigned to its most popular neighbour – that is the neighbour that has the most feature points in it. A cluster in Randomised Hashing is made up of a partition and all of the other partitions that are assigned to it by this process.

Each pixel $\boldsymbol{p}$ is assigned to the cluster that its feature vector $\Phi(\boldsymbol{p})$ belongs to. A possible example of this is shown in Figure 5-10.

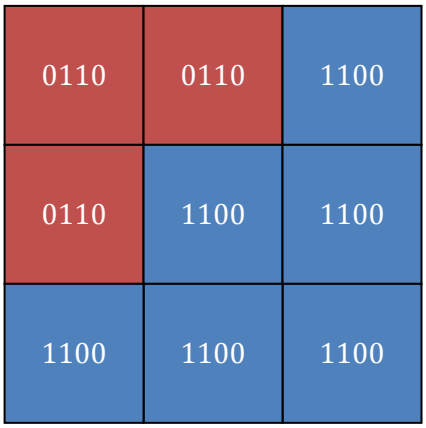


FIGURE 5-10: In this 3 by 3 image, each pixel has been assigned a partition code, depending on where its feature vector falls in the feature space.

A boundary is detected at all points where neighbouring pixels belong to different clusters.

5.5.2 IMAGE RESULTS

Running Randomised Hashing on some example images from the Berkeley dataset (see section 8.1) yields the results shown in Figure 5-11 below.

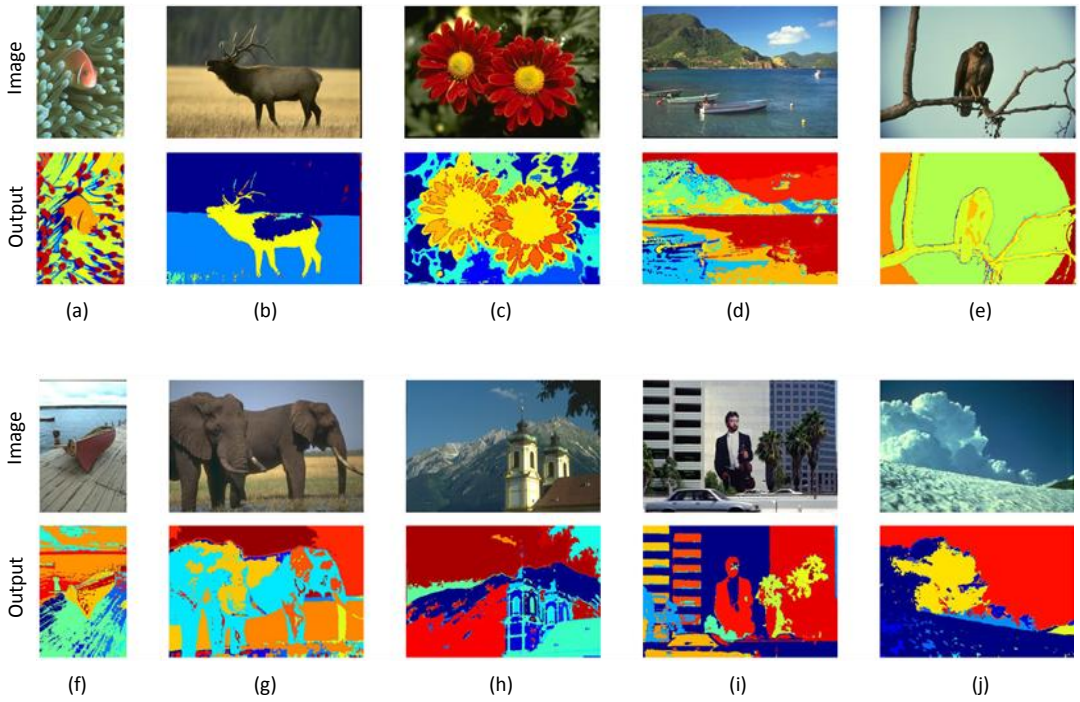


FIGURE 5-11: The results of Randomised Hashing on some images. Each colour is a different cluster. Adapted from Taylor and Cowley (2009).

5.5.3 CRITIQUE

In some of the example images in Figure 5-11 above, Randomised Hashing has subdivided areas of very similar colour. This can particularly be seen on the deer's back (b), and on the elephants (g). This happens because the clustering process is forced to introduce hard splits to the feature space in some way, and so sometimes similar feature points can be hard-assigned to entirely different clusters. Unfortunately in this case, this causes Randomised Hashing to introduce phantom boundaries – boundaries that exist where they should not. This is the main problem with Randomised Hashing.

5.6 CHAPTER SUMMARY

This chapter has explored the inner workings of five real-time texture boundary detectors. Each of these has its problems:

- Konishi's detector (section 5.1) is good at detecting boundaries, but oriented textures (like zebra stripes) cannot be suppressed by Konishi's detector.
- Surround Suppression (section 5.1) is fast and produces good results, but it is based on an edge detector, making it heavily focused on low-level information. This means it is prone to generating fragmented boundaries and also means it

cannot suppress the textures that can only be detected via higher-level interpretation.

- TextonRML (section 5.3) is able to generate pixelwise texture classifications which are generally correct, and so is useful for some real-time applications. However, the only reason why it can run in real-time is because it avoids doing the necessary alpha-expansion graph cut stage. Without it, TextonRML produces low-quality boundary maps.
- Semantic Texton Forests segmentation (section 5.4) attempts to overcome the need for the slow alpha-expansion graph cut stage by lowering the boundary map resolution and involving an image categorisation algorithm. This allows it to generate high-quality texture classifications in real-time, but unfortunately the low-resolution boundary maps leave much room for improvement.
- Boundary detection via Randomised Hashing (section 5.5) finds boundaries using real-time clustering. Unfortunately, the clustering process introduces non-existent "phantom" boundaries.

The next two chapters will propose two new texture-boundary detectors which overcome all the above problems, and most importantly, are able to run in real-time.

6 PROPOSAL: THE VARIANCE RIDGE DETECTOR

This chapter proposes the Variance Ridge Detector, a texture-boundary detector that overcomes the shortfalls of all previous methods.

6.1 RATIONALE

The Variance Ridge Detector is built on a single principal axiom, which is that *ridges in the variance space are likely positions of texture boundaries*. This is true for two reasons.

Firstly, consider a sliding window of pixels in an image. If only one texture is contained within this window, then the variance of this window only has to encapsulate the intra-class variation of the texture. Now, if the window is moved so that it now contains two textures, then the variance must now represent the inter-class variation between the textures in addition to the intra-class variation. For this reason, variance is likely to peak whenever two textures meet. Since a boundary is defined as the frontier at which two textures meet, variance is likely to form a ridge on a boundary.

Secondly, for different areas of the same texture, variance tends to be approximately uniform (Papari, Petkov, & Campisi, 2007). This occurs because different windows of the same texture are simply different samples from the same distribution. This uniformity of variance within texture means that it is unlikely that variance ridges will occur inside a texture.

The combination of the two above reasons enables variance to be an excellent choice for boundary detection. In addition to this, variance can be calculated significantly faster than most other features of texture (such as textons), which makes it ideal for constructing a real-time texture-boundary detector.

Based on this premise, the Variance Ridge Detector was developed. The steps of this algorithm can be divided into two phases. The first phase calculates the local variance at each pixel. The second phase detects ridges in the variance space. These two phases will be detailed into further steps in this chapter.

6.2 VARIANCE IN PREVIOUS WORK

Variance has been used for texture some notable previous work.

The most relevant work is the edge-preserving smoothing filter developed by Papari, Petkov and Campisi (2007), introduced in section 3.5. The heart of the Variance Ridge

Detector was inspired by their work. The Papari filter does not explicitly detect boundaries, but it does have an indirect mechanism for determining where the boundaries are so that it can avoid smoothing them. The Variance Ridge Detector was inspired by this mechanism. Having said that, Papari *et al*. did not intend for their filter to run in real-time, and so they used slower techniques such as convolution and non-rectangular smoothing, which meant many changes had to be made to adapt their work to the Variance Ridge Detector.

Another relevant work is the edge detector developed by Ahmad and Choi (1999), introduced in section 2.5. Their edge detector first detects edges in the image, as all traditional edge detectors do, but then it improves the result by only including the edges that occur on areas of high variance. Although they did not connect this to the concept of texture, it is likely that the reason this worked so well was because of the fact that variance peaks at texture boundaries, as was stated previously. Suppressing edges in areas of low variance would have removed many unimportant texture edges. In some ways, Ahmad and Choi's edge detector is like an early predecessor of the Variance Ridge Detector.

Tuzel, Porikli and Meer (2006) interestingly used covariance to recognise textures from the Brodatz texture dataset. Their covariance-based features achieve a recognition rate of 97.7%, which actually outperforms all texton-based methods that it was compared against.

The covariance-based features used by Tuzel et al.'s are much more complicated than simple variance, which is used by the Variance Ridge Detector. No texture-boundary detector based on this feature exists yet, and so that is a direction for future research.

Sharon and Brandt (2000) proposed a non-real-time segmentation algorithm which handles texture implicitly. The algorithm iteratively combines pixels into segments, and then combines those segments into bigger segments. This process is continued until the entire image is one big segment.

The segments are combined based on a similarity measure. If desired, variance can be used as part of this similarity measure. Their algorithm appears to produce good results, which can be seen in their paper. However, the algorithm cannot run in real-time, and it also does not take advantage of the critically useful fact that variance peaks at boundaries.

Overall, variance has been used for texture in some instances, but it appears that, unlike the Variance Ridge Detector, no method so far has taken advantage of the fact that it forms ridges on boundaries.

6.3 ALGORITHM OVERVIEW

The Variance Ridge Detector takes an input image $I_{rgb}(\boldsymbol{p})$, and it transforms the image through five major stages:

1. Convert to CIELab: $I_{rgb}(\boldsymbol{p}) \rightarrow I(\boldsymbol{p})$. Described in section 6.4.
2. Variance transform: $I(\boldsymbol{p}) \rightarrow V(\boldsymbol{p})$. Described in section 6.5.
3. Gradient transform: $V(\boldsymbol{p}) \rightarrow \nabla V(\boldsymbol{p})$. Described in section 6.6.
4. Ridge transform: $\nabla V(\boldsymbol{p}) \rightarrow R(\boldsymbol{p})$. Described in section 6.7.
5. Gradient magnitude subtraction: $R(\boldsymbol{p}) \rightarrow B(\boldsymbol{p})$. Described in section 6.8.

The final result of the algorithm is the boundary map $B(\boldsymbol{p})$.

Parameters

The proposed algorithm takes only one parameter – the window radius *r*. The choice of *r* should depend on the wavelength of textures in the image.

If *r* is much smaller than the texture wavelength, then the texture will not repeat within the algorithm's sliding window, meaning it will not look like texture to the algorithm, so it cannot be suppressed. On the other hand, if *r* is too large, then the boundary map will be coarser, and will not include the finer details. In essence, *r* is a scaling parameter. Ideally, *r* should match the general texture wavelength seen in the image. However, in practice *r* is not a sensitive parameter, and so it does not need to be chosen precisely.

Example images

The progress of the various stages of the algorithm will be illustrated with three example images, shown in Figure 6-1.

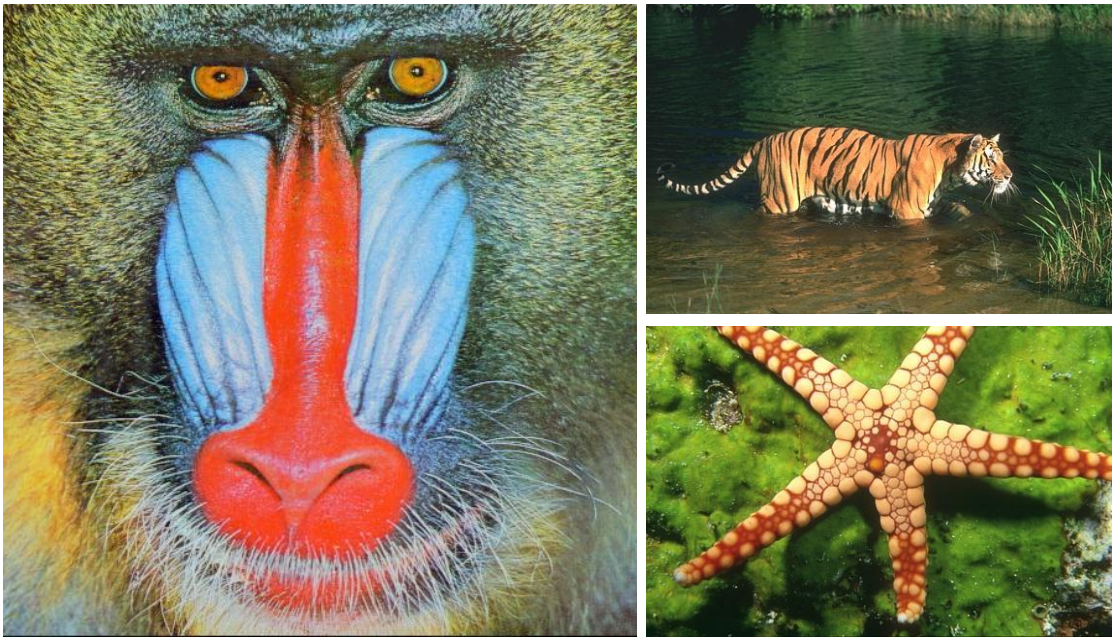


FIGURE 6-1: The various stages of the Variance Ridge Detector will be demonstrated on these images. Mandrill image (left) taken from Comaniciu and Meer (2002). Tiger and starfish images (right) taken from Berkeley segmentation dataset (Martin *et al.*, 2001).

The goal for the Variance Ridge Detector is to detect the boundaries, and suppress the edges within the most obvious textures in these example images. The most obvious textures are: the mandrill's fur (left picture), the tiger's stripes and the water (top right picture), and finally the scales on the starfish (bottom right picture). To achieve this, the parameter r has been set to $r = 6$ pixels, in order to match the general texture wavelength.

The Canny edge detector has been run on each of these images to show how difficult texture-boundary detection is on these images, as shown in Figure 6-2:

FIGURE 6-2: The results of running the Canny edge detector on the example images.

The Canny edge detector reveals that there is plenty of texture in the example images. The goal of the Variance Ridge Detector is to ignore this texture entirely and detect only the important boundaries.

6.4 CONVERT TO CIELAB COLOUR SPACE

The image is first converted from the RGB colour model to the CIELab colour model:

$$I(\boldsymbol{p}) = CIELab\left(I_{rgb}(\boldsymbol{p})\right) \quad \textbf{(6.1)}$$

It is very common for boundary detection to use the CIELab colour space – most of the detectors introduced in chapters 4 and 5 use CIELab. This is because the CIELab colour space was designed to match experimental measurements of human colour perception. Therefore, using CIELab allows an algorithm to better approach human performance.

To convert a colour from RGB to CIELab, the RGB model must first be converted to CIEXYZ, and then to CIELab, using this algorithm (OpenCV, 2008; Poynton, 2006):

$$
\begin{aligned}
\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} &\leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \\
X &\leftarrow \frac{X}{X_n} \text{ where } X_n = 0.950456 \\
Z &\leftarrow \frac{Z}{Z_n} \text{ where } Z_n = 1.088754 \\
L &\leftarrow \begin{cases} 116 \times Y^{1/3} - 16 & \text{when } Y > 0.008856 \\ 903.3 \times Y & \text{when } Y \leq 0.008856 \end{cases} \\
a &\leftarrow 500\big(f(X) - f(Y)\big) \\
b &\leftarrow 200\big(f(Y) - f(Z)\big) \\
\text{where } f(t) &= \begin{cases} t^{1/3} & \text{when } t > 0.00856 \\ 7.787t + 16/116 & \text{when } t \leq 0.00856 \end{cases}
\end{aligned}
\tag{6.2}
$$

This algorithm is used to convert the image $I(\boldsymbol{p})$ to CIELab space.

6.5 VARIANCE TRANSFORM

As its name suggests, the key to the Variance Ridge Detector is variance. The variance transform $V(\boldsymbol{p})$ calculates the local variance for every pixel $\boldsymbol{p}$:

$$
\begin{aligned}
V(\boldsymbol{p}) &= \| \mu_{I^2}(\boldsymbol{p}) - \mu_I(\boldsymbol{p})^2 \| \\
\mu_I(\boldsymbol{p}) &= BoxBlur_I(\boldsymbol{p}; r)
\end{aligned}
\tag{6.3}
$$

The smoothing function $\mu_I(\boldsymbol{p})$ is defined as $BoxBlur_I(\boldsymbol{p}; r)$, which was defined previously in equation (3.1). Unless otherwise noted, the box blur will always be used as the smoothing function throughout this chapter.

$V(\boldsymbol{p})$ calculates the variance separately in each of the three colour channels, and then combines the channels together using the L2 norm.

6.5.1 IMAGE EXAMPLES

Running the variance transform on the example images yields this:

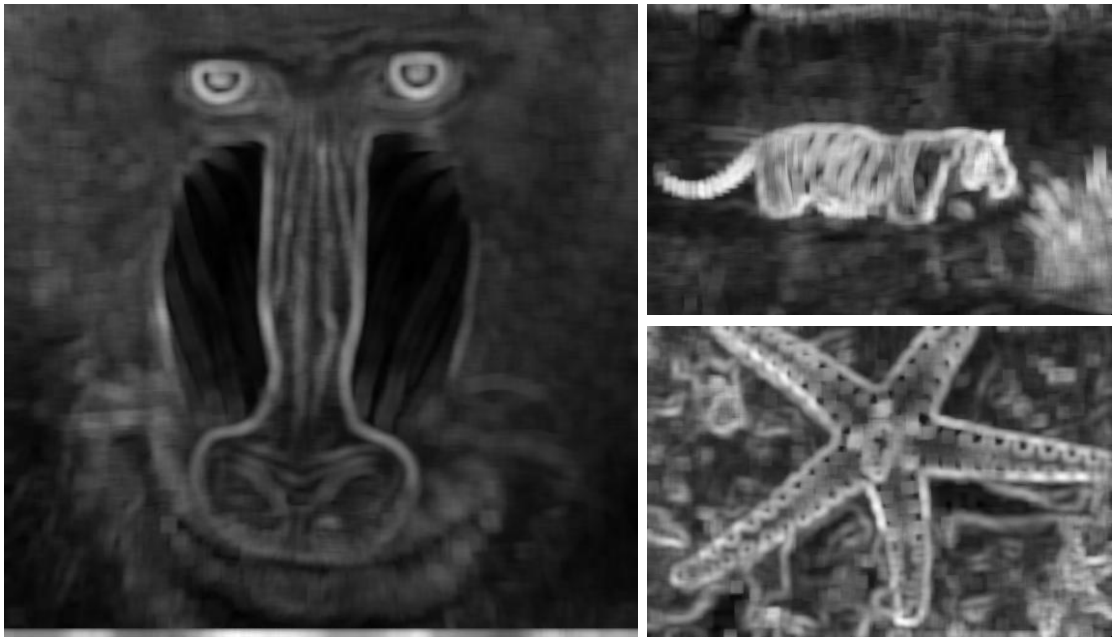


FIGURE 6-3: The variance transform on the example images.

Notice that the two important points stated in section 6.1 can both be seen in these images:

1. Variance peaks at the texture boundaries.
2. Within a texture, variance tends to be approximately the same.

These two facts provide a platform to run the rest of the Variance Ridge Detector.

6.5.2 JUSTIFICATION FOR THE REARRANGED VARIANCE EQUATION

$V(\boldsymbol{p})$ calculates the variance for a window surrounding pixel $\boldsymbol{p}$ by using a well-known rearrangement of the standard variance formula, as derived below:

$$\begin{aligned} Var(X) &= E[(X-\mu)^2] \\ &= E[X^2 - 2X\mu + \mu^2] \\ &= E(X^2) - 2\mu^2 + \mu^2 \end{aligned} \tag{6.4}$$

$$Var(X) = E(X^2) - \mu^2 \tag{6.5}$$

The rearrangement stated in equation (6.5) states that variance is equal to the squared mean minus the mean squared. The variance transform uses this equation, employing the smoothing function $\mu_I(\boldsymbol{p})$ to calculate the mean and squared mean. The rearranged variance equation (6.5) is used instead of the standard variance equation (6.4) simply because it is faster. This can be explained as follows.

The rearranged variance equation requires two means to be calculated. The mean of a sliding window is equal to its sum divided by its size. Calculating the sum of a sliding

window can be done quickly because information can be shared, as illustrated in Figure 6-4:

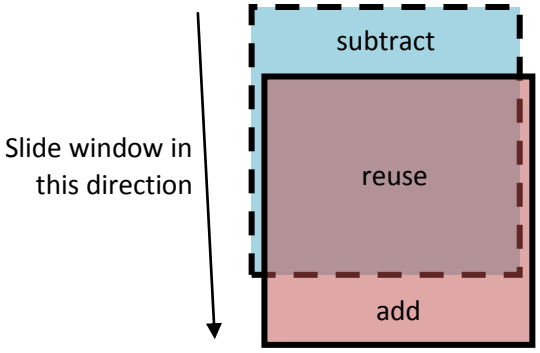


FIGURE 6-4: When calculating the sum of a sliding window, some calculations can be reused as the window slides. Normally the sliding direction will be either horizontal or vertical, this diagram just slides it in a slightly off-vertical direction to make the diagram clearer.

If the standard variance equation (6.4) were to be used, then no calculations could be reused. For a sliding window centered on pixel $\boldsymbol{p}$, the standard variance equation must calculate the difference between each pixel $I(\boldsymbol{p}+\boldsymbol{p}')$ and the mean of that particular window $\mu_I(\boldsymbol{p})$. The mean of each window is different, which means two different windows cannot share any calculations. Hence it is faster for the algorithm to use the rearranged variance equation (6.5).

6.5.3 JUSTIFICATION FOR SQUARE-SHAPED SLIDING WINDOWS

The Variance Ridge Detector calculates variance in a square-shaped sliding window. Doing this means the pixels on the perimeter of the window are not equidistant from the center pixel, which introduces a bias into the variance transform.

The obvious solution to this is to use a circular sliding window. This can be done by changing the box blur into a circular blur:

$$\begin{aligned} \mu_I(\boldsymbol{p}) &= CircularBlur(\boldsymbol{p};r) \\ CircularBlur(\boldsymbol{p};r) &= \frac{1}{Z}\sum_{\boldsymbol{p}'} I(\boldsymbol{p}+\boldsymbol{p}')[\|\boldsymbol{p}'\|_{L2} \le r] \\ Z &= \sum_{\boldsymbol{p}'} [\|\boldsymbol{p}'\|_{L2} \le r] \end{aligned} \tag{6.6}$$

The only difference between $CircularBlur(\boldsymbol{p};r)$ above and $BoxBlur(\boldsymbol{p};r)$ in equation (3.1), is that the L2 norm is used instead of the L1 norm.

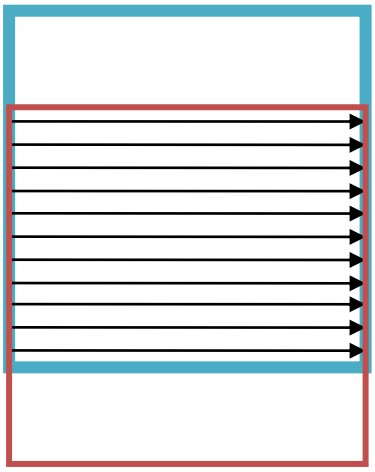
Doing this yields results such as these:

FIGURE 6-5: The variance transform on the example images, when using a circular sliding window instead of the proposed square window.

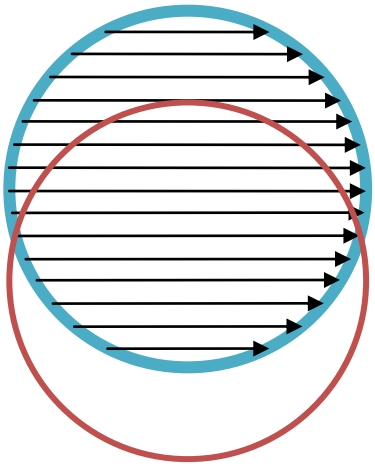
Comparing the results in Figure 6-5 to those shown in Figure 6-3 illustrates the effect of using a circular window versus a square window respectively. Notice that in Figure 6-3, the peaks look jagged – indicative of the bias introduced by using square-shaped sliding windows. This bias is not present when using circular windows, as shown in Figure 6-5.

The reason why square-shaped windows have been chosen is because they are faster. Even though it degrades the quality of the variance transform slightly, the result is a similar enough approximation that the high-quality results (presented in chapter 9) can still be achieved.

The reason they are faster is because they allow more calculations to be reused, as illustrated in Figure 6-6:



Two different square windows can share some calculations because the rows are the same length.



Two different circular windows cannot share calculations (as easily) because the rows are different lengths.

FIGURE 6-6: The reason why square-shaped sliding windows are faster.

That is why square-shaped sliding windows have been used.

6.5.4 JUSTIFICATION FOR AN EQUALLY-WEIGHTED WINDOW

Interestingly, Papari, Petkov and Campisi (2007) chose to calculate variance using a Gaussian-weighted window. That means that each pixel contributes to the variance with a different weight. This was tested with the Variance Ridge Detector, but ultimately it was found that not only did this produce lower-quality results, but it also runs slower. There are some clear reasons for this.

Calculating a Gaussian-weighted variance is clearly more complex than a uniformly-weighted variance, which is why it is more computationally-intensive and slower.

The lower-quality results can also be explained. Using a Gaussian-weight distorts the texture and introduces false boundaries, as illustrated in the Figure 6-7:

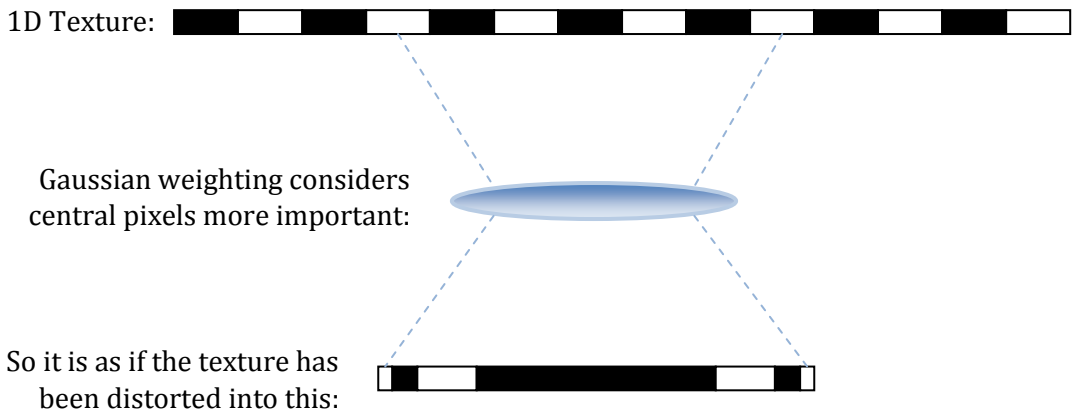


FIGURE 6-7: Using a Gaussian weighting distorts texture like a lens would, making it non-uniform.

The distortion of the texture makes it look non-uniform, which makes its variance non-uniform. Ultimately, this means intra-texture boundaries are detected when they should instead be suppressed. Texture needs to be uniform in order to be suppressed effectively.

This can be explained as follows. A seemingly non-uniform texture has non-uniform variance. To be able to be non-uniform, there must be peaks and troughs in the variance. The Variance Ridge Detector identifies peaks as boundaries, and so false peaks means false boundaries. Therefore using a Gaussian-weighted variance produces lower-quality results.

6.6 GRADIENT TRANSFORM

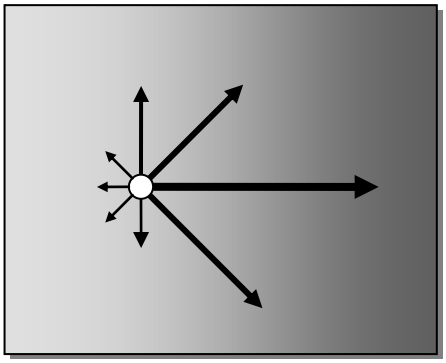
The gradient transform calculates the gradient at each pixel. The reasons for having this stage are twofold.

Firstly, variance is approximately equal for different areas of the same texture. This means that textured areas will have very little gradient in the variance space, which enables the variations in texture to be suppressed.

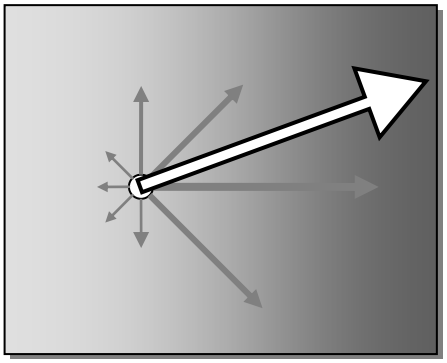
Secondly, the gradient transform will allow variance ridges to be detected in the next stage.

6.6.1 VISUALISATION

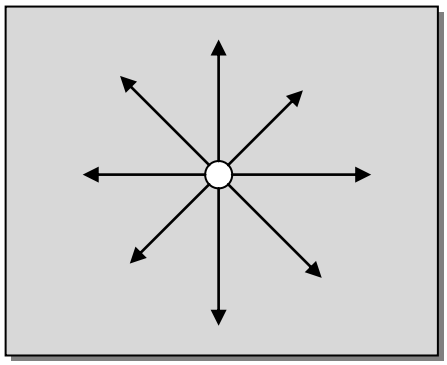
One way to visualise the gradient transform is shown in Figure 6-8:



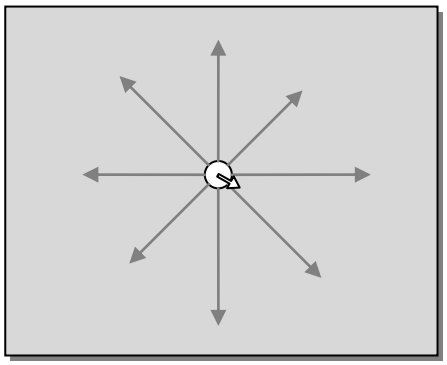
(a) Each pixel is "attracted" towards different directions, where each the attraction force is proportional to the variance in that direction.



(b) The attraction forces are all summed together to find the overall gradient for that pixel.



(c) Variance is about the same throughout texture, which means the attraction forces are approximately equal



(d) That means texture (almost) cancels itself out.

FIGURE 6-8: An illustration of how the gradient transform works.

Figure 6-8(a) and Figure 6-8(b) illustrate what normally happens near a boundary. Boundaries introduce strong variance into the image, and so they introduce strong gradients in the variance space. Figure 6-8(c) and Figure 6-8(d) illustrate that within a texture and away from boundaries, the gradient cancels itself out. The fact that the gradient transform responds differently to each situation is the core reason why the Variance Ridge Detector can handle texture.

6.6.2 FORMULATION

The gradient transform function $\nabla V(\boldsymbol{p})$ calculates the gradient vector for the pixel at position $\boldsymbol{p}$. The term "gradient vector" is used because both the gradient strength and direction are calculated. This can be formulated as follows:

$$\nabla V(\boldsymbol{p}) = \sum_{\boldsymbol{d}\epsilon\Theta(\omega_g)} \boldsymbol{d} \times \partial V(\boldsymbol{p}; \boldsymbol{d})$$

$$\Theta(\omega) = \bigcup_{i}^{\omega} \begin{bmatrix} \cos\frac{i\pi}{\omega} \\ \sin\frac{i\pi}{\omega} \end{bmatrix} \tag{6.7}$$

$$\partial V(\boldsymbol{p}; \boldsymbol{d}) = \mu_V(\boldsymbol{p} + r\boldsymbol{d}) - \mu_V(\boldsymbol{p} - r\boldsymbol{d})$$

The high-level view of the equations above is this. $\partial V(\boldsymbol{p}; \boldsymbol{d})$ calculates the gradient strength at pixel $\boldsymbol{p}$ for a particular direction $\boldsymbol{d}$. $\nabla V(\boldsymbol{p})$ uses $\partial V(\boldsymbol{p}, \boldsymbol{d})$ to calculate the gradient over ω_g total directions, taking the vector sum over all of the directions.

The parameter ω_g defines the number of directions to use. A small value of ω_g means lower-quality results, while a high value of ω_g slows down the algorithm. Throughout this thesis, ω_g has implicitly been set to 8 as this has allows for both good results and speed.

The function $\Theta(\omega_g)$ calculates unit vectors for ω_g directions and will be reused throughout this chapter and the next one.

6.6.3 JUSTIFICATION FOR SMOOTHED VARIANCE

The gradient calculation $\nabla V(\boldsymbol{p})$ only samples the variance at a small number of points and so is prone to sampling errors. Using the smoothed variance $\mu_V(\boldsymbol{p})$ in the gradient calculation is a fast method to reduce noise and make the gradient calculation robust.

It is possible to achieve a similar result by simply doubling the window size that is used for variance transform. However, it was found that this approach runs faster, due to the greater locality of reference when using smaller windows.

6.6.4 IMAGE EXAMPLES

The gradient transform produces the images shown in Figure 6-9:

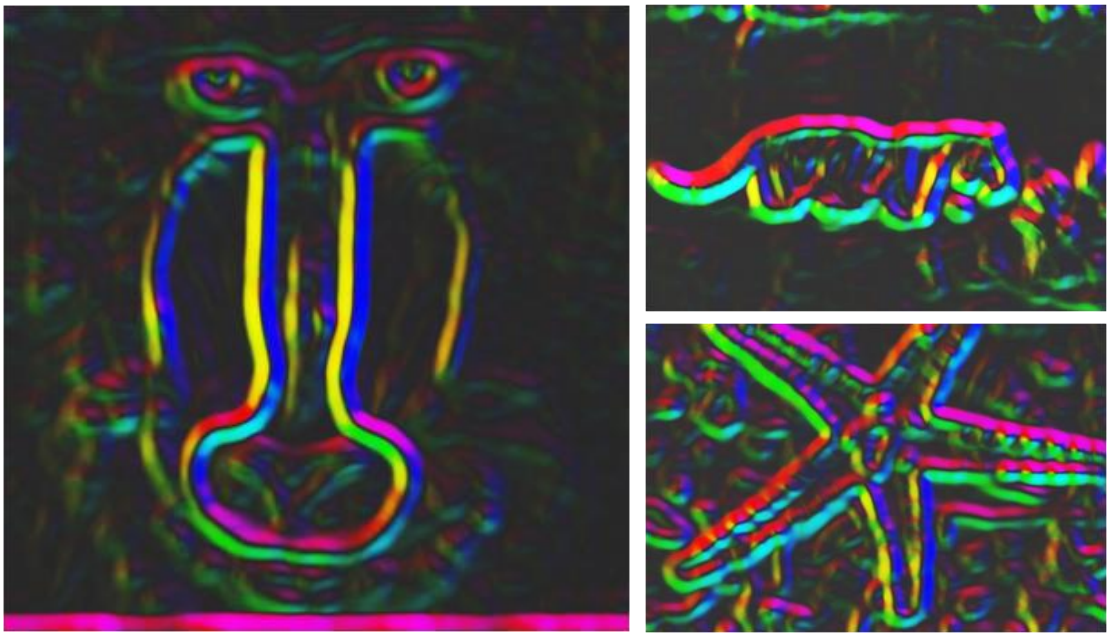


FIGURE 6-9: The gradient transform on the example images. Hue represents gradient orientation.

There are two points to take note of in Figure 6-9:

- As stated before, textured areas do not have much gradient in their variance, so much of the texture has been eliminated from the algorithm. Particularly, notice the mandrill's fur and the tiger's stripes are almost all gone.
- The gradients indicate where the variance ridges, and therefore boundaries are. More specifically, the boundaries are surrounded by a particular pattern of gradients. This fact is utilised by the next stage to detect boundaries.

6.7 RIDGE TRANSFORM

In one dimension, a peak will produce a double response in the gradient space – a positive gradient on one side, and a negative gradient on the other. This is illustrated in Figure 6-10.

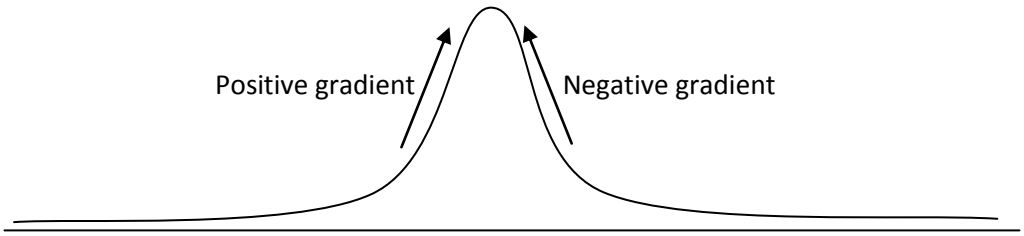


FIGURE 6-10: A peak has a positive gradient on one side, and a negative gradient on the other.

In two dimensions, gradients point inwards towards the ridge, for the same reasons as the one-dimensional case:

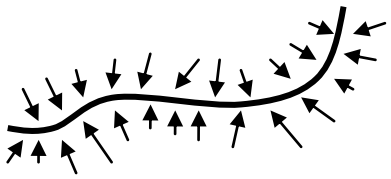


FIGURE 6-11: A ridge can be detected because the surrounding gradients will point inwards towards it.

The pattern described in Figure 6-10 can be seen in the gradient transform of the example images (Figure 6-9). In the figure, it can be seen that each boundary has opposite hues on either side of it. This occurs because, the gradient direction is represented by hue, and when the gradients face inwards, one side of the boundary must face an opposite direction to the other.

The purpose of the ridge transform is to detect these inward-facing gradient responses as ridges.

6.7.1 FORMULATION

The ridge transform $R(\boldsymbol{p})$ can be expressed by the following equations:

$$
\begin{aligned}
R(\boldsymbol{p}) &= \max_{\boldsymbol{d}\in\Theta(\omega_r)} R_S(\boldsymbol{p};\boldsymbol{d}) \\
R_S(\boldsymbol{p};\boldsymbol{d}) &= R_S^{geo}(\boldsymbol{p};\boldsymbol{d}) + R_S^{arith}(\boldsymbol{p};\boldsymbol{d}) \\
R_S^{geo}(\boldsymbol{p};\boldsymbol{d}) &= \sqrt{[-\nabla V_{in}(\boldsymbol{p},-\boldsymbol{d})\cdot\nabla V_{in}(\boldsymbol{p},\boldsymbol{d})]^+} \\
R_S^{arith}(\boldsymbol{p};\boldsymbol{d}) &= \frac{\|\nabla V_{in}(\boldsymbol{p},-\boldsymbol{d}) - \nabla V_{in}(\boldsymbol{p},\boldsymbol{d})\|}{2} \\
\nabla V_{in}(\boldsymbol{p},\boldsymbol{d}) &= \begin{cases} \nabla V(\boldsymbol{p}+r\boldsymbol{d}) & \text{when } \nabla V(\boldsymbol{p}+r\boldsymbol{d})\cdot\boldsymbol{d} < 0 \\ [0,0]^T & \text{otherwise} \end{cases}
\end{aligned}
\tag{6.8}
$$

Optionally, the ridge normal $R_d(\boldsymbol{p})$ can be calculated as well:

$$
R_d(\boldsymbol{p}) = \arg\max_{\boldsymbol{d}\in\Theta(\omega_r)} R_S(\boldsymbol{p};\boldsymbol{d}) \tag{6.9}
$$

The high-level view of the above equations is this. $R_S(\boldsymbol{p};\boldsymbol{d})$ calculates the strength of a single ridge orientation $\boldsymbol{d}$. $R(\boldsymbol{p})$ uses $R_S(\boldsymbol{p};\boldsymbol{d})$ to find the maximum ridge strength over ω_r possible ridge orientations.

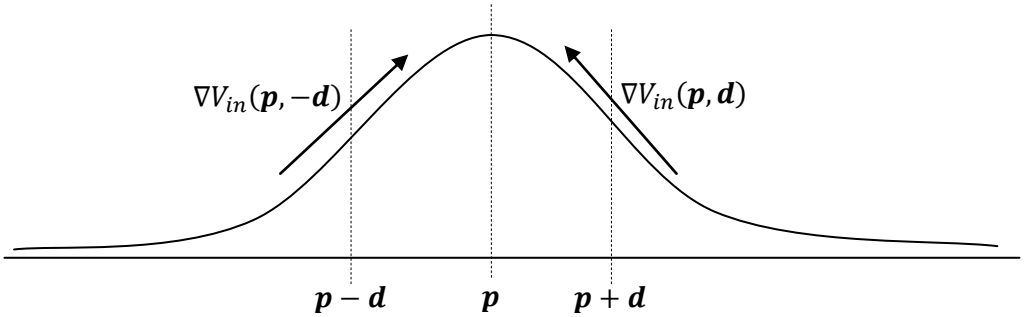


FIGURE 6-12: Variance (vertical axis) at different spatial offsets (horizontal axis) from image position $\boldsymbol{p}$ on a cross section along direction $\boldsymbol{d}$. The ridge strength at $\boldsymbol{p}$ is calculated by combining the variance gradients on either side of $\boldsymbol{p}$.

The ridge strength function $R_S(\boldsymbol{p}; \boldsymbol{d})$ measures how strongly the pixel $\boldsymbol{p}$ matches the pattern of having a negative gradient on one side, and a positive gradient on the other. In practice, this looks like two gradients pointing towards each other, as shown in Figure 6-11. The ridge strength function $R_S(\boldsymbol{p}, \boldsymbol{d})$ linearly combines two different methods for this: $R_S^{geo}(\boldsymbol{p}, \boldsymbol{d})$, which is similar to the geometric mean, and $R_S^{arith}(\boldsymbol{p}, \boldsymbol{d})$, which is similar to the arithmetic mean. In these functions, $\nabla V_{in}(\boldsymbol{p}, \boldsymbol{d})$ is used to ensure that only gradients pointing *inwards* are considered, but otherwise it is identical to $\nabla V(\boldsymbol{p} + r\boldsymbol{d})$.

The heart of the $R_S^{geo}(\boldsymbol{p}, \boldsymbol{d})$ function calculates the negated dot product of the two gradients on either side. This will produce a strong positive response only when the gradients are pointing towards each other. Combining the dot product with a square root essentially makes the function work like a geometric mean – that means, the function will only produce the maximum response when both gradients on either side are of similar strength. This minimises the occurrence of false positives.

In contrast, the $R_S^{arith}(\boldsymbol{p}, \boldsymbol{d})$ simply subtracts the gradient on one side from the gradient on the other. If the gradients are facing towards each other, this will generate a large response. This response is divided by two, which makes the function work like the arithmetic mean. Unlike the geometric mean, this means a gradient can be one-sided and still produce a response. This ensures some ridges are not eliminated prematurely.

$R_s(\boldsymbol{p}, \boldsymbol{d})$ then linearly combines $R_s^{arith}(\boldsymbol{p}, \boldsymbol{d})$ and $R_s^{geo}(\boldsymbol{p}, \boldsymbol{d})$ with equal weight.

6.7.2 RIDGE STRENGTH APPROXIMATION

In practice, the ridge strength is calculated with an approximation:

$$R_S^{geo}(\boldsymbol{p}, \boldsymbol{d}) = \sqrt{[-(\nabla V(\boldsymbol{p} - r\boldsymbol{d}) \cdot \boldsymbol{d}) \times (\nabla V(\boldsymbol{p} + r\boldsymbol{d}) \cdot \boldsymbol{d})]^+}$$
$$R_S^{arith}(\boldsymbol{p}, \boldsymbol{d}) = [\nabla V(\boldsymbol{p} - r\boldsymbol{d}) \cdot \boldsymbol{d} - \nabla V(\boldsymbol{p} + r\boldsymbol{d}) \cdot \boldsymbol{d}]^+ \quad \textbf{(6.10)}$$

Equation (6.10) above and the original equations proposed in (6.9) have a couple of differences worth noting.

Instead of comparing the gradients to each other directly, these functions first compare the gradients to the direction $\boldsymbol{d}$, by using the dot product $\nabla V(\boldsymbol{p}) \cdot \boldsymbol{d}$, and then they compare those dot products to each other using the geometric or arithmetic mean, in the same way as the original equations (6.9).

This produces a similar result, but is much faster due to a number of reasons:

1. The values of $\nabla V(\boldsymbol{p}) \cdot \boldsymbol{d}$ can be precalculated for all pixels $\boldsymbol{p}$, and their cost amortised over multiple usages. In fact, for a particular pixel $\boldsymbol{p}$, $\nabla V(\boldsymbol{p}) \cdot \boldsymbol{d}$ will be reused four times over the course of the ridge transform stage, which significantly reduces computation.
2. The $R_S^{geo}(\boldsymbol{p}, \boldsymbol{d})$ and $R_S^{arith}(\boldsymbol{p}, \boldsymbol{d})$ functions now only have to work with scalars instead of $\mathbb{R}^2$ vectors, due to the fact that $\nabla V(\dots) \cdot \boldsymbol{d}$ produces scalar values. This halves the number of operations required.
3. Calculating the vector magnitude in the original $R_S^{arith}(\boldsymbol{p}, \boldsymbol{d})$ equation (6.9) is a costly process, involving multiple operations. Now that only scalars are used instead of vectors, this process reduces to just a single subtraction operation.
4. The function $\nabla V_{in}(\dots)$ is not necessary anymore, instead it has been replaced with positive bounding operators $[\dots]^+$. This is faster because the positive bounding operator takes a single CPU instruction, while $\nabla V_{in}(\dots)$ involved a dot product and so required many more CPU instructions.
5. Another difference between the approximation (6.10) and the original equations (6.9) is, $R_S^{arith}(\boldsymbol{p}, \boldsymbol{d})$ no longer involves a division. It was found that this division did not make much difference to the results, and so removing it means one less operation.

These approximations allow the Variance Ridge Detector to better achieve real-time.

6.7.3 IMAGE EXAMPLES

The ridge transform produces the following on the example images shown in Figure 6-13:

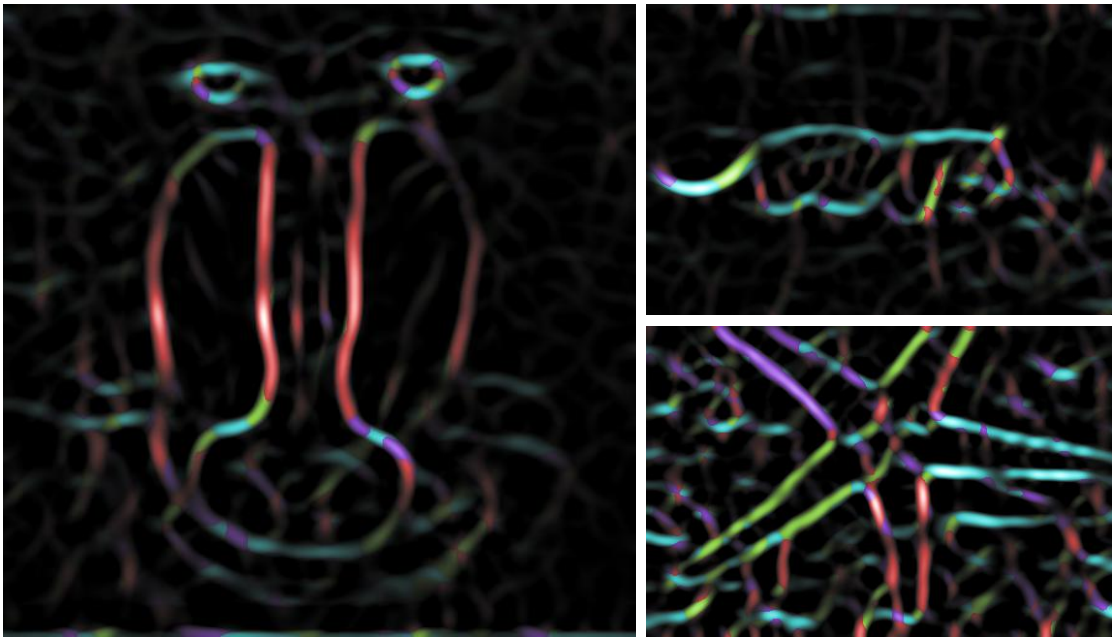


FIGURE 6-13: The ridge transform on the example images. Hue represents ridge orientation.

In Figure 6-13, notice that all the boundaries have been detected, and the textures have been suppressed. Although, this could be the final output of the algorithm, the results are further improved when Variance Ridge Detector performs one last stage, described later in section 6.8.

6.7.4 ALTERNATIVE APPROACH: OPPOSITES FILTER

Several alternative methods to the standard ridge transform have been developed. One of them was called the "opposites filter," which convolves the image with kernels that look like similar to the one shown in Figure 6-14:

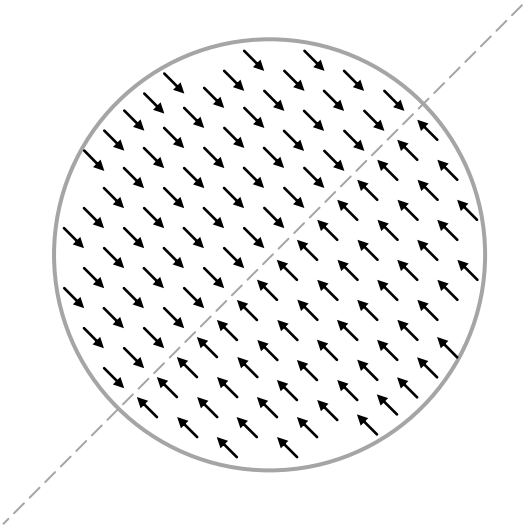


FIGURE 6-14: The opposites filter uses convolution kernels like these.

Figure 6-14 shows a kernel made up of vectors. All the vectors are pointing inwards towards a dividing line. This arrangement of vectors mimics the inward pattern of gradients on either side of a boundary, as explained earlier. This type of kernel will produce the maximum response on a boundary, as that is where the gradients will match this pattern.

The kernel is produced using the following equations:

$$OppositesKernel(\boldsymbol{p}';\boldsymbol{d}) = G_\sigma(\boldsymbol{p}') \times sign(\boldsymbol{p}\cdot\boldsymbol{d}) \times \boldsymbol{d}$$

$$\text{where } sign(v) = \begin{cases} 1 & \text{when } v > 0 \\ 0 & \text{when } v = 0 \\ -1 & \text{when } v < 0 \end{cases} \qquad \textbf{(6.11)}$$

The Gaussian kernel $G_\sigma(\boldsymbol{p}')$ was already defined previously in equation (2.12). The Gaussian scaling parameter σ is normally set to $r/3$ to ensure that the Gaussian approaches zero near the edges of a sliding window of radius r. In the above equations, the $OppositesKernel(\boldsymbol{p}';\boldsymbol{d})$ function generates the value of the kernel for direction $\boldsymbol{d}$ at position $\boldsymbol{p}'$. The vectors in the kernel are weighted according to the Gaussian function.

Now the ridge transform can be defined as follows:

$$\begin{aligned} R(\boldsymbol{p}) &= \max_{\boldsymbol{d}\in\Theta(\omega_r)} R_S(\boldsymbol{p};\boldsymbol{d}) \\ R_S(\boldsymbol{p};\boldsymbol{d}) &= (\nabla V * K_d)(\boldsymbol{p}) \\ &\text{where } K_d(\boldsymbol{p}') = OppositesKernel(\boldsymbol{p}';\boldsymbol{d}) \end{aligned} \qquad \textbf{(6.12)}$$

Although this produces a more robust response than the standard ridge transform, it is much slower because it must use convolution. It was found that this slowdown was not necessary to achieve high-quality results, and so the opposites filter is merely presented here as an interesting alternative approach and is not proposed as part of the Variance Ridge Detector.

6.7.5 ALTERNATIVE APPROACH: STRUCTURE TENSORS

Another alternative approach to the standard ridge transform is to use structure tensors, in the same way that Konishi's detector does (see section 5.1). In this approach, each gradient in the local area votes for a boundary orientation, and the coherence of the votes is taken.

$$R(\boldsymbol{p}) = Coherence\left(\sum_{\boldsymbol{p}'} \varphi\big(\nabla V(\boldsymbol{p}+\boldsymbol{p}')\big)[\|\boldsymbol{p}'\|_{L1} \le r]\right) \qquad \textbf{(6.13)}$$

The $\varphi(\dots)$ and $Coherence(\dots)$ functions were already defined in equations (3.5) and (5.5) respectively.

There are two problems with this approach.

Firstly, this approach is slower than the standard ridge detection approach. One of the primary reasons for this is that it accesses more pixels. Notice that the standard ridge transform in equation (6.8) will only access two pixels per direction, while the structure tensor approach in equation (6.13) above must access every pixel in the sliding window. Memory access is the slowest operation of all, and so this slows down the algorithm substantially.

Secondly, this approach does not try to compare whether there are opposing gradients on either side of the boundary. The gradients can be positioned anywhere and still contribute equally to the average boundary orientation. This tends to introduce false-positives.

The combination of these two problems is why the structure tensor approach to the ridge transform is only presented here as an interesting approach, and is not proposed as part of the variance ridge transform.

6.8 GRADIENT MAGNITUDE SUBTRACTION

The ridge transform is effective at identifying ridges, but it turns out that ridges are often over-detected in the ridge transform. That means, often they appear thicker than they actually are, particularly around corners. This occurs because the ridge transform only takes an extremely small sample of two points for each ridge orientation.

One solution to this would be to increase the sample size. However, this would slow down the algorithm. This section proposes a much faster solution which achieves the same purpose.

As its name suggests, the gradient magnitude subtraction simply subtracts the gradient magnitude from the ridge transform. This produces the boundary map, which is the final result of the algorithm. This can be formulated as follows:

$$B(\boldsymbol{p}) = R(\boldsymbol{p}) - \|\nabla V(\boldsymbol{p})\| \qquad \textbf{(6.14)}$$

This is useful because a ridge can only exist between two gradients – a positive gradient on one side, and a negative gradient on the other. Therefore, if a ridge occurs at the same location as a gradient, instead of between gradients, then it is unlikely to be a true ridge. This stage subtracts the gradient magnitude from the ridge transform, which means that

any ridges that exist at the same location as a gradient will be removed. This results in thinner, more accurate ridges. These ridges are output as the final result of the detector.

6.8.1 IMAGE EXAMPLES

The final result of the Variance Ridge Detector on the example images is shown in Figure 6-15:

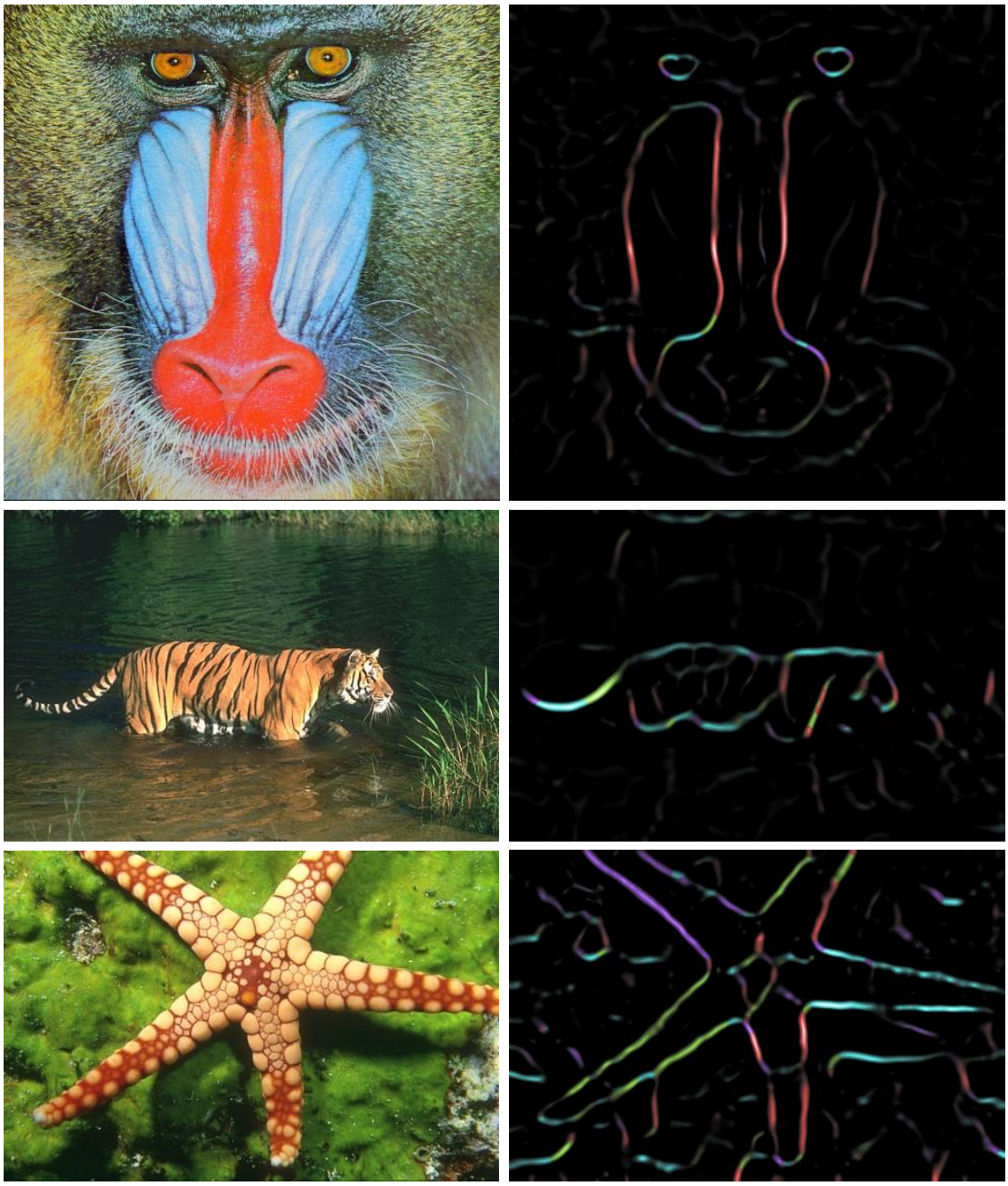


FIGURE 6-15: The final result of the Variance Ridge Detector on the example images. Hue represents boundary orientation.

Comparing this output (Figure 6-15) to those of the previous stage (Figure 6-13) shows that the gradient magnitude subtraction process makes the boundary map much clearer and better localised. More results will be presented in chapter 9.

6.8.2 ALTERNATIVE APPROACH: ANISOTROPIC SUBTRACTION

The method that was just presented is called isotropic gradient magnitude subtraction. The word isotropic here means that the subtraction is always the same, regardless of direction. This can sometimes cause problems, as illustrated in Figure 6-16:

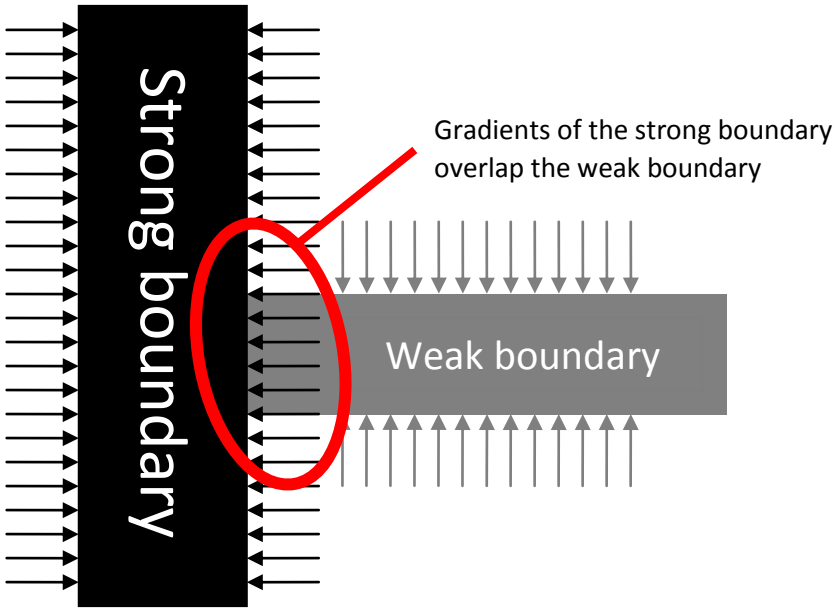


FIGURE 6-16: The problem with isotropic gradient magnitude subtraction.

The problem is, when at a junction where multiple boundaries meet, the gradients of one boundary might overlap another valid boundary. A simple isotropic subtraction will therefore remove some valid boundaries.

One possible solution to this relies on gradients always being perpendicular to their boundaries. So, if a boundary and gradient appear at the same position, the subtraction should only occur if they are perpendicular. This can be formulated into the anisotropic gradient subtraction equation:

$$B(\boldsymbol{p}) = R(\boldsymbol{p}) - |G(\boldsymbol{p}) \cdot R_d(\boldsymbol{p})| \qquad \textbf{(6.15)}$$

$R_d(\boldsymbol{p})$ was defined in equation (6.9) to return the ridge normal, or in other words, the boundary normal. The boundary normal is already perpendicular to the boundary, which makes the calculation quite simple. The dot product of the unit boundary normal

and the gradient is found. Effectively this weights the amount of gradient subtraction so that the full gradient magnitude is only subtracted when the boundary is perpendicular to the gradient.

Doing this produces interesting results, as illustrated in Figure 6-17.

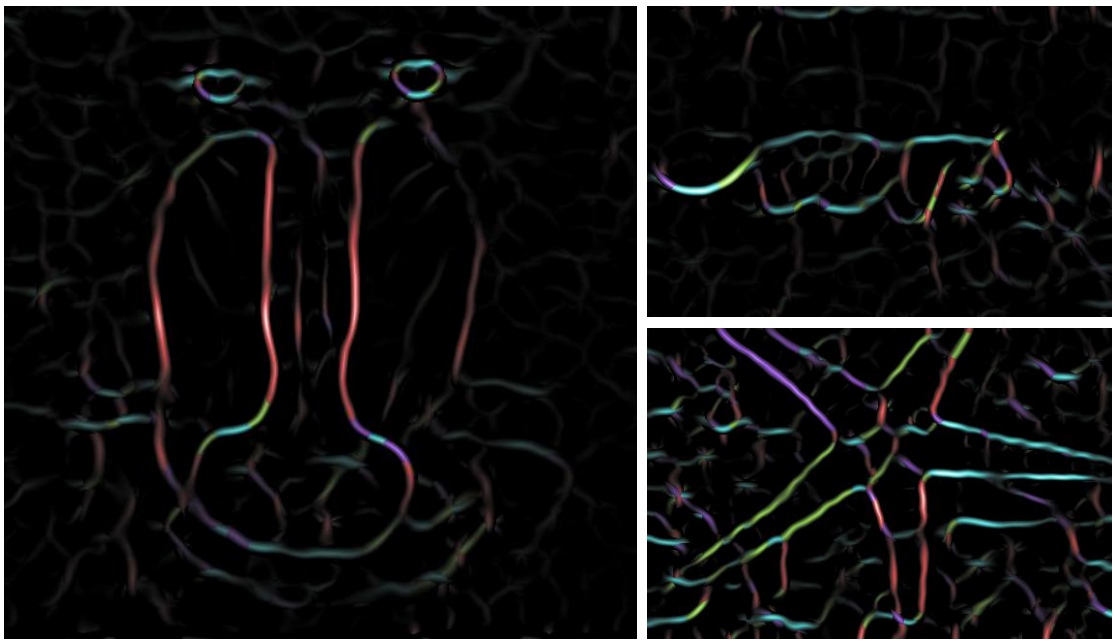


FIGURE 6-17: Anisotropic gradient magnitude subtraction applied to the example images.

Unfortunately, anisotropic gradient magnitude subtraction introduces some unwanted artefacts into the boundary map. In Figure 6-17, this can be seen around the mandrill's eyes (left image), as well as at various corners throughout the other images.

The non-homogeneous subtraction of gradient magnitude leaves artefacts when it does not completely subtract away all unwanted boundaries. This particularly happens around corners, where the boundary orientation is ambiguous.

At this point, isotropic gradient magnitude subtraction produces better results, and so anisotropic gradient magnitude subtraction is simply presented here as an interesting alternative approach, and is not proposed as part of the variance transform.

6.9 COMPARISON WITH OTHER RIDGE DETECTION APPROACHES

The gradient transform (section 6.6), ridge transform (section 6.7) and gradient magnitude subtraction (section 6.8) stages are all part of the proposed ridge detection method. Previous approaches to ridge detection include: morphological thinning, Canny's non-maximum suppression and convolution thinning. However, these approaches were inadequate for the Variance Ridge Detector, for the following reasons.

As described in section 2.3.3, morphological thinning requires a binary image as input. Normally, thresholding would be used to convert an image into a binary image. Doing the threshold before the ridge detection means there is a strong possibility that thresholding might eliminate useful information, or introduce artefacts into the boundary detection. Hence morphological thinning was not used.

In contrast, section 2.4.2 described the non-maximum suppression algorithm used by the Canny edge detector. This approach performs ridge detection first before thresholding. This is a better approach, as reduces the chance that thresholding will eliminate ridges or introduce artefacts.

The problem with Canny's ridge detection is that it has no substitute for the proposed gradient transform of the Variance Ridge Detector (section 6.6). As mentioned earlier, the gradient transform eliminates texture by taking advantage of the fact that within a texture, variance is approximately uniform. Canny's ridge detection does not use this fact, and so would not perform as well as the proposed ridge detection method. In other words, Canny's ridge detection is good at detecting ridges, but it has no method for suppressing false positives like the proposed ridge detection method does.

Section 4.5.3 described convolution thinning, which is used as part of the state-of-the-art probability of boundary detector. In this case, convolution is unnecessarily computationally expensive – the proposed ridge detection method already produces an satisfactory result with much less computation.

All other ridge detection methods were inadequate for this situation, that is why a new ridge detection method was proposed.

6.10 IMPLEMENTATION

The Variance Ridge Detector was implemented so that its speed and quality could be measured. The results of this are presented in chapter 9. There were a number of issues with implementing the algorithm, and the purpose of this section is to detail these issues.

6.10.1 EXPANDING THE IMAGE

Whenever a sliding window partially lies outside the bounds of the image, the pixel values are interpolated by mirroring the image at the image bounds.

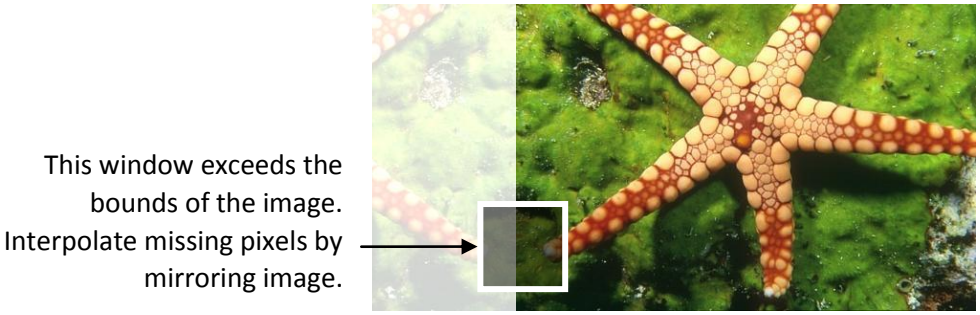


FIGURE 6-18: Illustration of how missing pixels are interpolated.

An alternative to mirroring at the image boundaries would be to just use a solid colour such as white or black beyond beyond the image boundaries. This was not done because it would be likely to introduce strong variance and thus reduce the reliability of the results near the image extremities.

6.10.2 DISCRETISATION

Often an expression such as $\nabla V(\boldsymbol{p} + r\boldsymbol{d})$ has been used. In this expression, $\boldsymbol{p} + r\boldsymbol{d}$ could potentially refer to a non-integral pixel position in the gradient image ∇V. In the implementation, $\boldsymbol{p} + r\boldsymbol{d}$ would be rounded to the nearest integral position.

6.10.3 SLIDING WINDOWS

Many expressions, such as the box blur in equation (3.1) have a window term such as $[\|\boldsymbol{p}'\|_{L1} \leq r]$. This ensures that only pixels within the window radius r have an effect on the calculation – other pixels are zeroed out by this term. Naturally, the implementation does not waste time on values outside of the window – it does not calculate them only to zero them out.

6.10.4 IMPLEMENTATION RESOURCES

The Variance Ridge Detector was implemented as a highly-optimised single-threaded C++ program. Images were represented as 32-bit floating-point numbers. The implementation utilised OpenCV and SSE instructions.

OpenCV (the Open Computer Vision library) is an open-source computer vision library which contains common image processing functions and algorithms. OpenCV 1.1 was used for this implementation.

SSE stands for Streaming SIMD Extensions, where SIMD stands for Single-Input Multiple Data. The SSE instruction set is a special collection of CPU instructions which allows the CPU to process multiple pieces of data at a time. For example, the SSE add instruction

can add four pairs of floating point numbers in one operation. A normal add instruction would only add one pair of numbers. Intuitively, this allows an algorithm to run up to four times faster. Instructions from SSE, SSE2 and SSE3 were used for this implementation.

6.10.5 USING SSE INSTRUCTIONS

OpenCV 1.1 is highly optimised, but does not utilise SSE instructions even though they are widely available on modern CPUs. For that reason, only a few of OpenCV's functions were used, and most of the operations were separately coded so that they could be accelerated with SSE instructions.

There were three cases in particular where OpenCV's implementation was faster than our SSE implementation, and so in those cases, OpenCV's implementation was used. These cases included the averaging function $\mu_I(\dots)$, the conversion from RGB to CIELab, and the mirroring of the image at image bounds. SSE instructions were used in all other cases.

One of the cases where using SSE was not straightforward was the three-channel sum algorithm, which is needed in the variance transform. This case provides an interesting insight into how the Variance Ridge Detector was optimised, and will be detailed in the next section.

6.11 THE THREE-CHANNEL SUM ALGORITHM

The variance transform calculates the variance separately in each colour channel, and then combines them using the L2 norm. As described in section 2.1, the L2 norm simply squares each channel, sums the channels together, and then takes the square root. The three-channel sum operation required some thought to be able to implement it with SSE. The purpose of having this section is not so much to explain the solution, but to provide an insight into what was involved in making the Variance Ridge Detector run at maximum speed.

What makes it difficult?

In a three-channel image, each pixel is a tuple of three values, one value for each colour channel. Floating-point SSE works with tuples of four values. This mismatch of tuple sizes makes using SSE difficult.

The memory layout of an image can be visualised like in Figure 6-19:

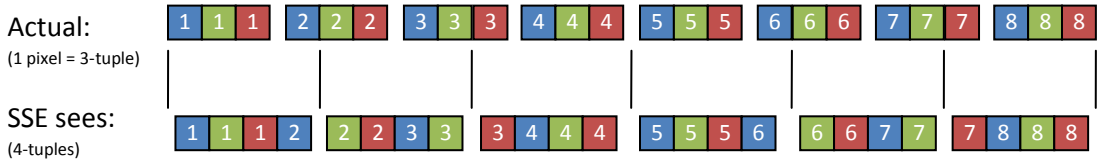


FIGURE 6-19: SSE's grouping of the image data is different from how the pixels should be grouped, making the three-channel sum difficult.

SSE groups the numbers differently, and in the process, it groups different pixels into the same tuple. This makes it difficult to perform a three-channel sum with SSE instructions.

One solution would be to rearrange the memory layout to make it easier for SSE to work. This is possible, but would be slow. There is a better solution, which will be described next.

Introducing the SSE instructions

There are a few instructions in the SSE instruction set that can be used to solve this three-channel sum problem.

The SSE **add** instruction is obviously useful when calculating sums. Figure 6-20 illustrates what it does:

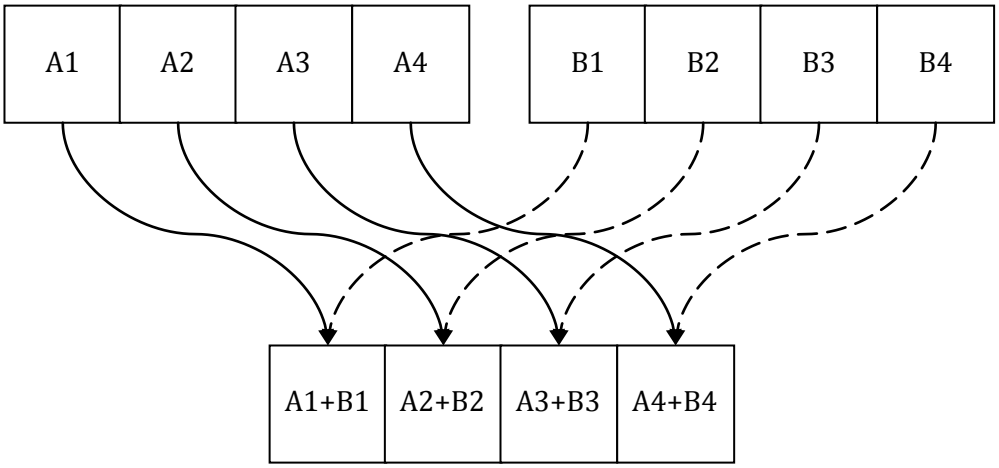


FIGURE 6-20: The SSE add operation adds each pair in two different tuples together.

Another instruction, **horizontal add** will add neighbouring pairs together:

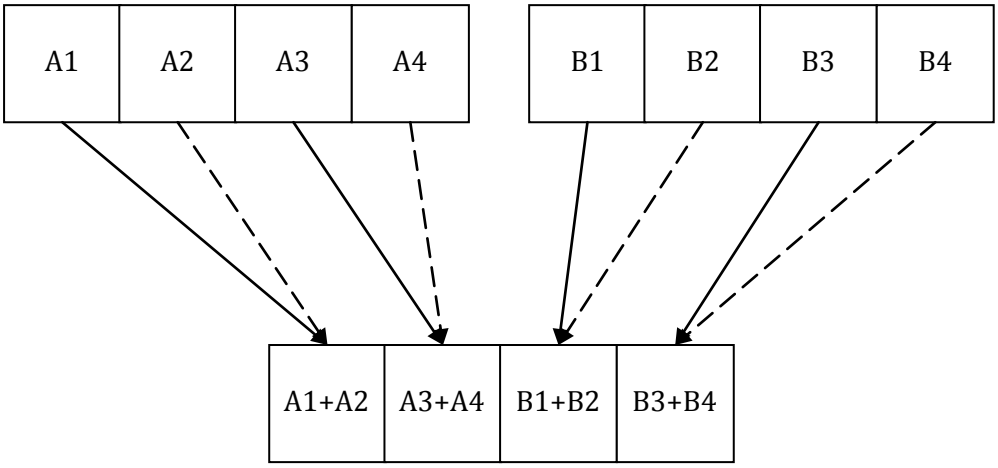


FIGURE 6-21: The SSE horizontal add operation.

Shuffle extracts 4 out of the 8 elements in two input 4-tuples, producing a resultant 4-tuple. Any 4 elements can be extracted, subject to some conditions.

The first 2 elements in the resultant tuple must always come from the first input 4-tuple. In the same way, the last 2 elements in the resultant tuple must always come from the second input 4-tuple. Consequently, it is not possible to, for example, take three elements from the first 4-tuple and only one element from the second 4-tuple. This restriction is the basis for certain design decisions presented later.

The SSE three-channel sum algorithm

The SSE three-channel sum algorithm has two halves. The first half sums two out of the three channels in each pixel. The second half adds the remaining channel.

Figure 6-22 below outlines the first half of the algorithm.

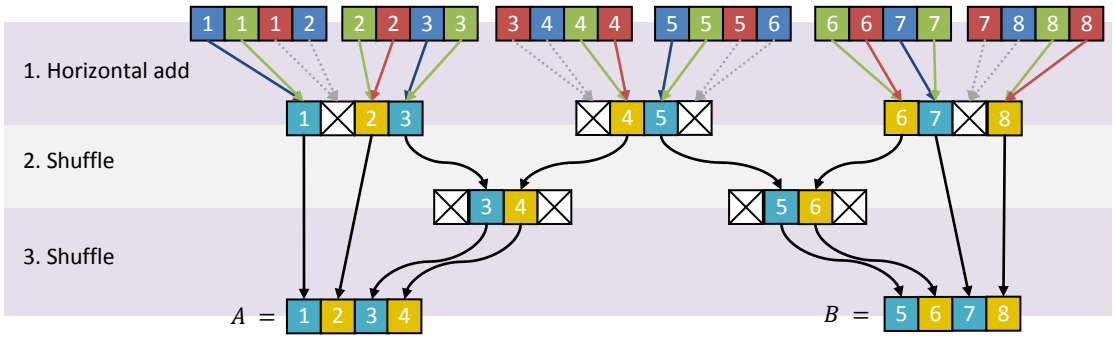


FIGURE 6-22: The first half of the three-channel sum algorithm. Each cell represents one value in the image. The cells are numbered according to which pixel they come from. The cell colours represent what sums they contain.

The SSE three-channel sum algorithm operates on groups of eight pixels at a time, one of which is represented at the top of Figure 6-22. The image has three channels of colour,

in this case, red, green and blue. Each cell has been coloured according to which of these channels it belongs to.

From here, the three-channel sum algorithm proceeds as follows. In step 1, horizontal adds are used to add two out of the three channels in each pixel, forming partial sums. Some of the add operations will have summed channels from different pixels, which is not useful. The diagram marks the useless sums as a white square with an X drawn through them. The useful sums are rendered in the combined colour. Intuitively, adding red and green cells results in a yellow cell, and adding green and blue cells results in a cyan cell.

Steps 2 and 3 then rearrange the partial sums into the correct order. This cannot be done in one step because of the restrictions of the shuffle operation discussed previously.

Figure 6-23 below outlines the second half of the algorithm:

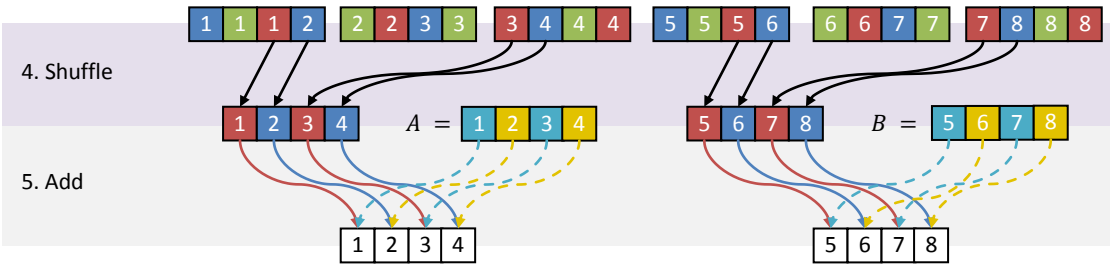


FIGURE 6-23: The second half of the three-channel sum algorithm.

The second half of the algorithm extracts the remaining unsummed values using a shuffle (step 4), and then adds them to the partial sums previous calculated (step 5) to generate the final three-channel sums. As the sums are already in the correct order, they can simply be stored into the image directly with no further manipulation or reordering.

Advantages of the SSE three-channel sum algorithm

Performing a three-channel sum for one pixel without SSE would require:

- Three load operations, one for each channel.
- Two addition operations. For example, $R + G + B$ involves two additions.
- One store operation, to store the result.

That is six total operations per pixel.

The SSE three-channel sum requires the following operations for eight pixels:

- Six SSE load operations, to load the image data.

- Three horizontal add operations (for step 1).
- Six shuffle operations (for steps 2-4).
- Two addition operations (for step 5).
- Two SSE store operations, to store the results.

That comes to a total of twelve operations for eight pixels. For comparison, if the non-SSE three-channel sum were run on eight pixels, it would take 48 operations (6 operations/pixel × 8 pixels). This is four times the number of operations as the SSE version. Hence, the SSE three-channel sum is much faster.

6.12 CHAPTER SUMMARY

This chapter proposed the Variance Ridge Detector, a novel texture-boundary detector that is both capable of detecting texture boundaries, and is also able to run in real-time. The next chapter explains further modifications that can be made to the Variance Ridge Detector algorithm to enable it to generate even higher quality results.

7 PROPOSAL: THE TEXTON RIDGE DETECTOR

Textons have already been introduced as state-of-the-art (section 4.3), and some authors have managed to calculate them in real-time (Ranganathan, 2009; Shotton, Johnson, & Cipolla, 2008). However, all real-time texton-based boundary detectors have had their problems. This chapter proposes a way to overcome these problems by improving the Variance Ridge Detector with textons.

7.1 RATIONALE

The Variance Ridge Detector relies on the fact that variance will peak at boundaries. The Texton Ridge Detector is based on a similar idea, except instead of variance, a texton gradient is used. The magnitude of the texton gradient will peak at boundaries. The texton gradient can be explained as follows.

Section 4.3 stated that each texture has its own characteristic distribution of textons. Consider then, what happens at a texture boundary. The two different textures on either side of the boundary will have vastly different distributions of textons. In contrast, consider a non-boundary pixel. As the textures on both sides of this pixel are the same, the texton distributions on either side will be similar.

The texton gradient is simply the distance between the texton distributions on either side of a pixel. This distance will peak at texture boundaries, forming the **texton ridges** underlying the Texton Ridge Detector.

7.2 ALGORITHM OVERVIEW

The Texton Ridge Detector is effectively a real-time version of the Pb (probability of boundary) detector (Martin, Fowlkes, & Malik, 2004), described in section 4.5. Both of these algorithms generally follow this procedure:

1. **Extract features for textons**. Pb convolves with a filter bank for this, whereas the Texton Ridge Detector extracts brightness gradients.
2. **Textonise the image**. Pb matches a feature vector to its nearest texton using linear search, whereas the Texton Ridge Detector proposes the use of an approximate nearest neighbour algorithm.
3. **Calculate the texton gradient** as the distance between sliding window histograms. Pb uses semicircle shaped-sliding windows for this, while the

Texton Ridge Detector uses square-shaped sliding windows. Both algorithms measure the histogram distance using the chi-squared distance measure.

4. **Combine the texton gradient with other information**. Pb uses a logistic regression model to combine the texton gradient with colour information. The Texton Ridge Detector combines the texton gradient with variance by multiplying them together.

5. **Perform ridge detection**. Pb uses convolution thinning for this. The Texton Ridge Detector uses the same ridge detection method used by the Variance Ridge Detector.

Both Pb and the Texton Ridge Detector have an offline training phase. In this stage, both algorithms must find the textons via k-means clustering (see chapter 4.3). The Texton Ridge Detector also must train its approximate nearest neighbour model during this stage.

The remaining sections in this chapter will describe the stages of the Texton Ridge Detector in more detail.

7.3 TEXTURE FEATURES

The Texton Ridge Detector uses brightness gradients as texture features. The brightness gradients are calculated at a scale determined by the user-defined window radius parameter r. This parameter is the same as the one already introduced in the previous chapter (in section 6.3).

Section 4.3 explained that textons work because each texture has its own characteristic autocorrelation pattern. As long as r is somewhat similar to the texture wavelength, simple brightness gradients are enough to make the autocorrelation pattern evident, making textures distinguishable. Section 6.3 already stated that the parameter r should already be approximately similar to the texture wavelength, and so the reuse of r for this purpose is ideal.

7.3.1 FORMULATION

The brightness gradients are calculated in greyscale. Using the same mathematical conventions as the previous chapter (see section 2.1), the conversion of a CIELab image $I(\boldsymbol{p})$ into a greyscale image $I_g(\boldsymbol{p})$ can be expressed as follows:

$$I_g(\boldsymbol{p}) = I(\boldsymbol{p}) \times \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{7.1}$$

This simply extracts the L (luminosity) channel from the CIELab image I.

The features are then extracted by the function $\Phi_I(\boldsymbol{p})$:

$$\Phi_I(\boldsymbol{p}) = \bigcup_{\boldsymbol{d} \in \Theta(\omega_f)} \partial I(\boldsymbol{p}; \boldsymbol{d})$$
$$\partial I(\boldsymbol{p}; \boldsymbol{d}) = \mu_{I_g}(\boldsymbol{p} + r\boldsymbol{d}) - \mu_{I_g}(\boldsymbol{p} - r\boldsymbol{d}) \tag{7.2}$$

The high-level view of the above equations is this. $\partial I(\boldsymbol{p}; \boldsymbol{d})$ calculates the brightness gradient in direction $\boldsymbol{d}$ from pixel $\boldsymbol{p}$ by comparing the means of two offset windows. The function $\Phi_I(\boldsymbol{p})$ concatenates the brightness gradients in ω_f directions from that pixel to form a **feature vector** (also known as a **feature point**). The number of directions w_f has implicitly been set to 2 in all cases throughout this thesis as this produces adequate results while maintaining speed.

The features extracted by $\Phi_I(\boldsymbol{p})$ are then used as input for the textonisation stage.

7.4 APPROXIMATE TEXTONISATION

As stated previously in section 4.3, a texton is a cluster of feature points. Most texton-based algorithms assign feature points to their nearest texton. This section proposes approximate textonisation, meaning that feature points will be assigned to a near texton, not necessarily the nearest. This allows for greater speeds at some cost to the textonisation quality.

The approximate textonisation algorithm has many similarities to Randomised Hashing, which was discussed in section 5.5, and so many of the terms and symbols from that section will be reused.

7.4.1 VISUALISATION

The novel approximate textonisation algorithm partitions the feature space by splitting it with a number of hyperplanes, as illustrated in Figure 7-1:

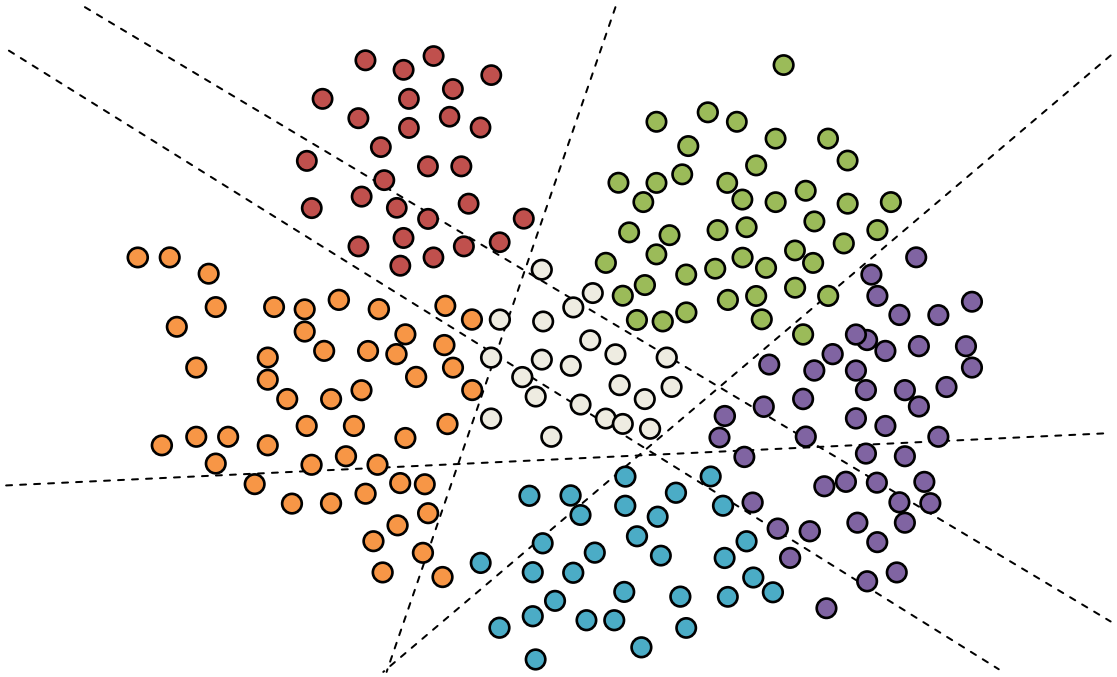


FIGURE 7-1: A diagram representing a feature space, with each feature point coloured according to the texton it belongs to. The dotted lines represent the separating hyperplanes in the feature space.

The separating hyperplanes split the feature space into a number of isolated cells. The word **partition** refers to one of these cells.

The task of finding the approximate texton for a given feature point is called **querying**. When querying, the approximate textonisation algorithm finds which partition the feature point belongs to, and then returns the modal (most common) texton for that partition.

The approximate textonisation algorithm needs to be trained before it can perform any querying. Once textons have already been found using the standard k-means clustering method (previously described in section 4.3.3), training involves generating a good set of separating hyperplanes which can be used to approximate the textons.

Querying and training are described in more detail in the next few sections.

7.4.2 QUERYING

Let the set of separating hyperplanes $S = \{s_1, s_2, \ldots, s_{|\mathbf{S}|}\}$. In this subsection, S will sometimes be expressed implicitly in equations because it is fixed during querying. For example, a function may be defined as $\rho(\boldsymbol{\phi}; S)$, but since S is fixed during querying, this will sometimes be written as $\rho(\boldsymbol{\phi})$ to make the equations easier to read.

Each hyperplane s_i is defined as a binary function which takes a feature point ϕ and returns either 0 or 1 depending on which side of the hyperplane the feature is on:

$$s_i(\boldsymbol{\phi}) = [\boldsymbol{\phi} \cdot \boldsymbol{a}_i - b_i \geq 0] \quad \textbf{(7.3)}$$

Let the feature points exist in an $\mathbb{R}^d$ feature space, where d is the number of features in a feature vector. In this function, $\boldsymbol{a}_i$ is an $\mathbb{R}^d$ unit vector that determines the orientation of the hyperplane, and b_i determines the offset of the hyperplane from the origin.

Given this information, the algorithm can calculate which side a feature point lays for all of the hyperplanes in S. It is useful to collapse these values into a single number, called the partition code:

$$\rho(\boldsymbol{\phi}; S) = \sum_{i}^{|S|} s_i(\boldsymbol{\phi}) \times 2^{i-1} \quad \textbf{(7.4)}$$

In the above equation, the partition code $\rho(\dots)$ can be thought of as a binary number, where each hyperplane in S contributes its respective bit of the partition code. The the partition code is important because all the points in the same partition will be assigned the same partition code, and points which do not belong to the same partition will be assigned to different codes. Effectively, the partition code is an ID number of each partition, and $\rho(\boldsymbol{\phi})$ is a fast method of calculating which partition ID a feature point $\boldsymbol{\phi}$ belongs to. This concept is identical to the partition code used by Randomised Hashing (section 5.5.1).

Let $t_a(\boldsymbol{\phi}; S)$ be a function that approximately assigns feature point $\boldsymbol{\phi}$ to a nearby texton. Let $t_m(\rho; S)$ be a function that returns the modal texton for partition ρ. This allows $t_a(\dots)$ to be defined as follows:

$$t_a(\boldsymbol{\phi}) = t_m\big(\rho(\boldsymbol{\phi})\big) \quad \textbf{(7.5)}$$

In practice, $t_m(\rho)$ is precalculated for all $\rho \in \mathrm{P}_S$, where P_S is the set of all possible partition codes:

$$\mathrm{P}_S = \{0, 1, 2, \dots, 2^{|S|} - 1\} \quad \textbf{(7.6)}$$

That means, the modal texton of every partition is precalculated and stored in a lookup table, so that once the partition code for a feature point $\boldsymbol{\phi}$ is calculated by $\rho(\boldsymbol{\phi})$, the modal texton can be found with a single memory access. This allows the approximate textonisation algorithm to run extremely fast.

Finally, let the function $T(\boldsymbol{p})$ denote the textonisation of the image I:

$$T(\boldsymbol{p}) = t_a\big(\Phi_I(\boldsymbol{p})\big) \quad \textbf{(7.7)}$$

In the above equation $\Phi_I(\dots)$ is the feature extraction function defined previously in equation (7.2). That is how an image is textonised by the Texton Ridge Detector.

7.4.3 TRAINING

The quality of the approximate textonisation algorithm depends on the quality of the hyperplane set. The task is to find the set S^*, the set of separating hyperplanes that maximises the objective function $Accuracy(S^*)$. This section will first construct the objective function. Then an algorithm that attempts to maximise the objective function will be introduced.

Training data

Let the training data be contained in the set $\Phi = \{\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \boldsymbol{\phi}_3, \dots, \boldsymbol{\phi}_{|\Phi|}\}$. Φ is a set of feature points. As per the normal texton-learning process (already described in section 4.3.3), the training feature points are first clustered using k-means clustering (Lloyd, 1982). This process can be sped up using the k-means++ optimisation (Arthur & Vassilvitskii, 2007). The output of k-means clustering is the function $t_\phi(\boldsymbol{\phi})$, which returns the texton to which feature point $\boldsymbol{\phi}$ belongs to. The function $t_\phi(\boldsymbol{\phi})$ was already defined previously in equation (4.9) in section 4.3.4.

The objective function

Let k be the number of textons. Let $\Phi_t = \{\boldsymbol{\phi} \in \Phi : t_\phi(\boldsymbol{\phi}) = t\}$, which means Φ_t is the set of all training features in Φ which belong to texton t.

The function $H_\rho(\rho; S)$ defined below is a histogram consisting of the training frequencies of each texton in one partition, identified by its partition code ρ:

$$H_\rho(\rho; S) = \bigcup_t^k h_\rho(\rho, t; S)$$
$$\text{where } h_\rho(\rho, t; S) = \sum_{\boldsymbol{\phi} \in \Phi_t} [\rho(\boldsymbol{\phi}; S) = \rho] \qquad \textbf{(7.8)}$$

The histogram is more useful if it is normalised. That is, if the total of all its bins equals one. The normalised texton histogram for a partition, $H_n(\rho; S)$, is defined as:

$$H_n(\rho;S) = \bigcup_{t}^{k} h_n(\rho,t;S)$$

$$\text{where } h_n(\rho,t;S) = \frac{h_p(\rho,t;S)}{N(\rho;S)} \tag{7.9}$$

$$N(\rho;S) = \sum H_p(\rho;S)$$

In the above equations, $N(\rho;S)$ equals the number of training feature points that have been assigned to partition ρ.

Using the histogram, the modal texton $t_m(\rho;S)$ of partition ρ can be found:

$$t_m(\rho;S) = \underset{t\in\{1\ldots k\}}{\arg\max}\, h_n(\rho,t;S) \tag{7.10}$$

If there is a tie for the modal texton, then one of the modal textons is chosen arbitrarily.

The querying function $t_a(\boldsymbol{\phi})$ defined in equation (7.5) states that, when querying for the texton for any feature point in partition ρ, the modal texton $t_m(\rho;S)$ is returned. Knowing this, the accuracy of hyperplanes S on the training set can be determined as follows:

$$Accuracy(S) = \frac{\sum_{\rho\in \mathrm{P}_S} N(\rho;S)\; h_n(\rho, t_m(\rho;S);S)}{|\Phi|} \tag{7.11}$$

The $Accuracy(S)$ function is equal to the probability that a random feature point from the training set will be assigned correctly by the approximate textonisation algorithm. Defining $Accuracy(S)$ this way allows it to be used as the objective function, which provides a framework for the training algorithm to run.

Training algorithm

Repeated random-restart hill climbing (Russell & Norvig, 2009; Jacobson & Yücesan, 2004) is used to find a good set of hyperplanes S^* according to the objective function.

Hill climbing works like this. First, a random solution is taken. The algorithm then searches neighbouring solutions, looking for one which improves the current solution. After many iterations of this, eventually the solution will reach a local maximum and will therefore be unable to be improved by just hill climbing. To be able to find the global maximum, **random-restart hill climbing** runs hill climbing from many random starting points, taking the best out of all runs.

The training algorithm takes a number of parameters:

- N_S specifies how many separating hyperplanes are to be found.

- N_I specifies the number of random-restart hill climbing iterations.
- $p_{restart}$ specifies the probability that the hill climbing will be restarted from a new random starting solution.
- α specifies the maximum amount to adjust each value by when hill climbing. Effectively, this determines how far a "neighbouring solution" can be from the current solution.
- β specifies the maximum offset from the origin for a separating hyperplane.

The training stage is described in Algorithm 7-1:

ALGORITHM 7-1: Training stage of the approximate textonisation algorithm

1. Initialise S^* as an empty set: $S^* \leftarrow \emptyset$
2. Use random-restart hill climbing to find the best new hyperplane s^*
3. Store the new hyperplane: $S^* \leftarrow S^* \cup \{s^*\}$
4. If $|S^*| < N_S$, go back to step 2
5. Return S^*

The random-restart hill climbing step, which is step 2 of Algorithm 7-1, can be subdivided further into the steps listed in Algorithm 7-2:

ALGORITHM 7-2: The random-restart hill climbing stage of the approximate textonisation algorithm

Let $rand(min, max)$ be a function that randomly chooses a real number between min and max inclusive, with all numbers in that range having equal probability. Given this, random-restart hill climbing does the following steps to find the best new hyperplane s^*:

1. Initialise the best new hyperplane $s^* \leftarrow null$
2. Initialise the number of iterations $n_I \leftarrow 0$
3. Randomly generate s' to be a random hyperplane, with the orientation $\boldsymbol{a}'$ and offset b':
   - 3.1. Let $a_i' \leftarrow rand(-1,1)$ for all $i \in \{1, \dots, d\}$, where d is the number of dimensions in the feature space.
   - 3.2. Let $b' \leftarrow rand(-\beta, \beta)$
4. Improve s' by hill climbing
   - 4.1. Let s'' be a random adjustment of s', where s'' is a hyperplane with orientation $\boldsymbol{a}''$ and offset b'':
     - 4.1.1. Let $a_i'' \leftarrow a_i' + rand(-\alpha, \alpha)$ for all $i \in \{1, \dots, d\}$, where d is the number of dimensions in the feature space.
     - 4.1.2. Normalise $\boldsymbol{a}'' \leftarrow \boldsymbol{a}''/\|\boldsymbol{a}''\|$
     - 4.1.3. Let $b'' \leftarrow b' + rand(-\alpha, \alpha) \times \beta$
   - 4.2. If $Accuracy(S^* \cup \{s''\}) > Accuracy(S^* \cup \{s'\})$ then:
     - 4.2.1. Update $s' \leftarrow s''$
   - 4.3. Increment $n_I \leftarrow n_I + 1$
   - 4.4. With a probability of $(1 - \alpha)$, go back to step 4.1
5. If $s^* = null$ or $Accuracy(S^* \cup \{s'\}) > Accuracy(S^* \cup \{s^*\})$ then:
   - 5.1. Update $s^* \leftarrow s'$
6. If $n_I < N_I$ then go back to step 3
7. Return s^*

7.4.4 TRAINING PARAMETERS

Unless otherwise stated, whenever results of the Texton Ridge Detector are discussed in this thesis, the following parameters were used for training:

- $k = 32$ textons.
- K-means++ clustering is run 1000 times.
- $|\Phi| = 1\,000\,000$ feature points were used for training, sampled from the Berkeley dataset (described later in section 8.1).
- $N_S = 20$ separating hyperplanes.
- $N_I = 100\,000$ iterations for random-restart hill climbing.
- $p_{restart} = 5\%$ chance of random restart.
- $\alpha = 5\%$ random adjustment for the separating hyperplanes.
- $\beta = 100\sqrt{2}$ maximum separating hyperplane offset from the origin.

This particular value of β was chosen for the following reasons. A hyperplane has the form $\boldsymbol{a} \cdot \boldsymbol{\phi} = b$. In that equation, $\boldsymbol{a}$ is a unit vector describing the orientation of the hyperplane, while b is the offset of the hyperplane from the origin. The value β determines the maximum valid range of b. This valid range can be determined as follows:

(1) The L channel, which is the brightness channel in CIELab, has the range [0,100].

(2) The function $\Phi_I(\dots)$ calculates a feature as the difference between two brightness values.

(3) Because of (1) and (2), each element in a feature vector $\boldsymbol{\phi}$ generated by $F(\cdot)$ can be in the range $[-100,100]$.

(4) Section 7.3.1 stated that a feature vector $\boldsymbol{\phi}$ has two elements – it is $\mathbb{R}^2$.

(5) The hyperplane orientation $\boldsymbol{a}$ must be a unit vector.

(6) Because of (4) and (5), it is known that the value of $\boldsymbol{a}$ which generates the maximum b is $\boldsymbol{a} = \left[\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$. This is because for this value, $\|\boldsymbol{a}\|$ is at maximum.

(7) A hyperplane has the form $\boldsymbol{a} \cdot \boldsymbol{\phi} = b$.

(8) Therefore, β can be found by substituting the maximal values of $\boldsymbol{a}$ and $\boldsymbol{\phi}$, known in (3) and (6), into the hyperplane equation (7).

If the feature vector generated by $\Phi_I(\boldsymbol{p})$ was different, then β would be different.

Training using these parameters took approximately ten hours. The trained set S^* of separating hyperplanes had an accuracy score of $Accuracy(S^*) = 91\%$.

More textons or more features?

Different versions of the Texton Ridge Detector with up to 128 textons and an $\mathbb{R}^9$ feature vector, but none of them improved the results, even though they did slow down the algorithm. The parameters stated here are the smallest parameters required to make the algorithm run fast while still achieving the high-quality results presented in chapter 9.

7.4.5 TEXTONISATION IMAGE EXAMPLES

The figure below demonstrates what the texton maps of the example images (used in the previous chapter) look like:

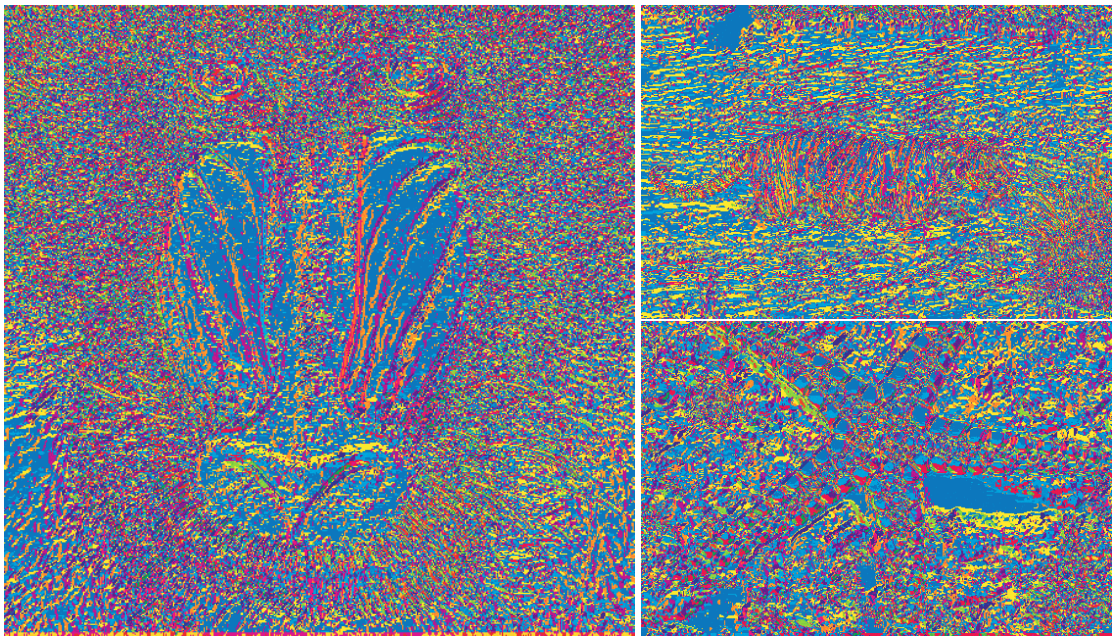


FIGURE 7-2: The texton maps of the example images. A different colour has been assigned to each texton. The scale $\boldsymbol{r} = \mathbf{3}$.

The texton maps are difficult for a human to fully interpret because it is difficult to visualise the distribution of textons at different places in the image. However, some interesting points can be seen. For example, it is clear that the water in the tiger's image (top right) has a much different texton distribution to the tiger itself.

Once the approximate textonisation algorithm is trained, the Texton Ridge Detector can textonise images using the function $T(\boldsymbol{p})$, which was defined previously in equation (7.7). Next, the texton gradient is calculated from this texton map.

7.5 TEXTON GRADIENT

The Pb algorithm calculates the magnitude of the texton gradient by comparing the texton histograms of two semicircle-shaped windows using the chi-squared distance. The Texton Ridge Detector does the same, except it uses square-shaped windows, allowing for faster speeds.

7.5.1 FORMULATION

Given a pixel position $\boldsymbol{p}$, a texton histogram of the square-shaped window centered on $\boldsymbol{p}$ can be calculated by the function $H(\boldsymbol{p})$:

$$H(\boldsymbol{p}) = \bigcup_{t \in \tau} h(\boldsymbol{p}, t)$$
$$\text{where } h(\boldsymbol{p}, t) = \sum_{\boldsymbol{p}'} [T(\boldsymbol{p} + \boldsymbol{p}') = t][\|\boldsymbol{p}'\|_{L1} \leq 2r] \quad \textbf{(7.12)}$$

The texton gradient magnitude can then be calculated by the function $\|\nabla T(\boldsymbol{p})\|$:

$$\|\nabla T(\boldsymbol{p})\| = \max_{\boldsymbol{d} \in \Theta(\omega_t)} \partial T(\boldsymbol{p}; \boldsymbol{d})$$
$$\partial T(\boldsymbol{p}; \boldsymbol{d}) = \chi^2\big(H(\boldsymbol{p} + 2r\boldsymbol{d}), H(\boldsymbol{p} - 2r\boldsymbol{d})\big) \quad \textbf{(7.13)}$$

The high-level view of the above equations is this. $\partial T(\boldsymbol{p}; \boldsymbol{d})$ calculates the texton gradient for direction $\boldsymbol{d}$ at pixel $\boldsymbol{p}$ using the chi-squared distance χ^2, which was already defined for the probability of boundary detector in section 4.5.2 equation (4.10). $\|\nabla T(\boldsymbol{p})\|$ takes the maximum $\partial T(\boldsymbol{p}; \boldsymbol{d})$ over ω_t directions to find the magnitude of the texton gradient. The direction of the texton gradient $\nabla T(\boldsymbol{p})$ is not important, so to maximise computational speed, only the magnitude $\|\nabla T(\boldsymbol{p})\|$ is calculated. Throughout this thesis, ω_t has been set to 2 as this produces good results while maintaining speed.

7.5.2 TEXTON GRADIENT IMAGE EXAMPLES

The texton gradient magnitudes of the example images are shown in Figure 7-3:

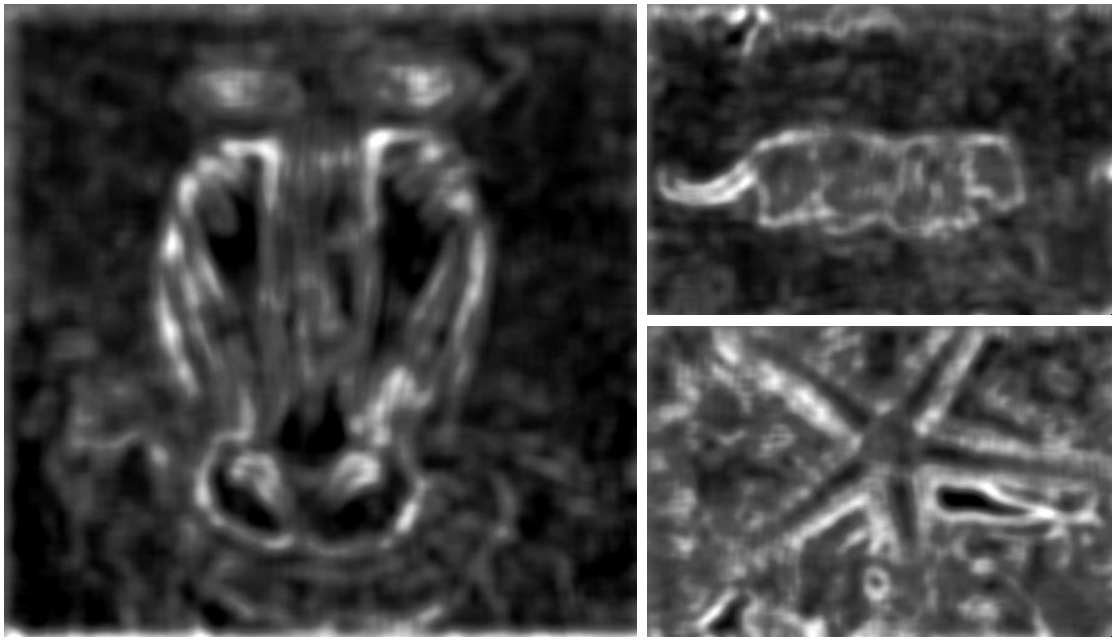


FIGURE 7-3: The texton gradient magnitude of the example images.

The texton gradient magnitude (Figure 7-3 above) is very different from the variance, shown previously in Figure 6-3. The boundaries of the tiger (top right) are detected much more clearly than with variance. However, the boundaries of the starfish (bottom right) are detected less clearly. Later, section 7.6 describes how both texton gradients and variance can be combined two allow to further improve this result.

7.5.3 JUSTIFICATION FOR THE DOUBLED SCALE

The equations (7.12) and (7.13) show that the texton gradient is calculated at the scale $2r$ instead of just r. This was found to produce higher-quality results. One possible reason for this is suggested here.

The features chosen in section 7.3 involve smoothing with scale r, which causes each pixel to influence all other pixels within radius r. That means pixels that are closer than r tend to be assigned to the same texton.

Figure 7-4 demonstrates how "nearby" textons tend to be same, where "nearby" means "closer than the scale *r*." The two images in Figure 7-4 are texton maps of the "starfish" example image (see Figure 6-1). In the left image, $r = 3$. In the right image, $r = 12$.

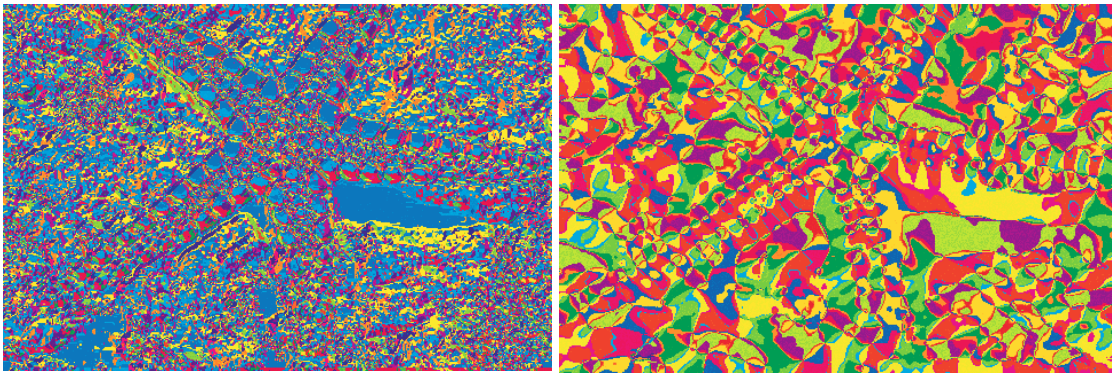


FIGURE 7-4: Pixels that are closer than *r* tend to be assigned to the same texton.

Notice that, in the right image where the scale is four times bigger, there are large blobs of equal colour where many nearby pixels have been assigned to the same texton. This is also happening in the left image, but at a much smaller scale. Indeed, it is clear that when pixels are closer than *r*, they have the tendency to be assigned to the same texton.

Due to this phenomenon, if the texton distribution is also calculated at scale *r*, then the distribution will be locally biased and will not represent the texture well. Doubling the scale for the texton distribution alleviates this problem.

7.5.4 IMPLEMENTATION DETAILS

The slowest part of the Texton Ridge Detector is the part that calculates the texton histogram $H(\boldsymbol{p})$ for all sliding windows on the image. To ensure maximum speed, this stage was optimised as much as possible using a number of techniques.

Square-shaped windows

Square-shaped windows are used instead of the semicircle shape used by the Pb detector because they are much faster, for the reasons already stated in section 6.5.

Rolling sum

The histogram is accumulated using a rolling sum algorithm. A rolling sum can be demonstrated with an example problem:

5	10	17	6	3	12	8	10	4	2

This problem concerns an array of numbers, like the one illustrated above. The sum of the first sliding window is known (highlighted above), where in this example, the window is six elements wide. Now the window is slid one place to the right (highlighted below).

5	10	17	6	3	12	8	10	4	2

The fastest way to calculate the new sum given the sum of the previous window is to first take the known total of elements 1-6, then subtract element 1 and add element 7. This gives the desired sum for elements 2-7. Only two operations need to be performed even though the sliding window contains six numbers.

Rolling sums for two dimensions

Calculating the rolling sum in two-dimensions is done as follows. First the rolling sums are calculated vertically for each column using a vertical window of size $1 \times s$. Then, on the resulting vertical sums, the rolling sums are calculated horizontally for each row using a horizontal window of size $s \times 1$. This gives the sums for a window of size $s \times s$.

Figure 7-5 illustrates why this works:

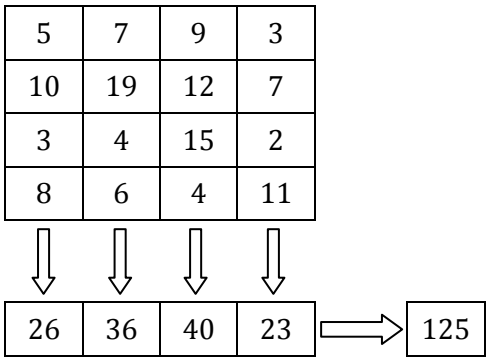


FIGURE 7-5: Vertical sums followed by a horizontal sum can be used to find the total of a two-dimensional sliding window.

Using rolling sums for texton histograms

Equation (7.12) requires the texton histograms to be calculated. The rolling sum technique can easily be applied to this situation. The only change is that, textons are added and subtracted from a histogram instead of just a sum. The rest of the rolling sum technique remains the same.

Summing order

Memory is linear but an image is two-dimensional, and so some mapping must occur between the two spaces. The conventional way to unravel a two-dimensional image for storage in memory is "rows-first". That is, in this order:

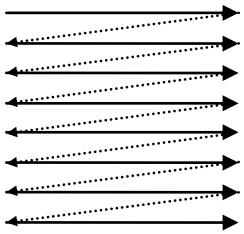


FIGURE 7-6: Images needs to be stored linearly in memory. They are conventionally stored in "row-first" order, which stores an image row by row.

Assuming the conventional image-memory layout, it is faster to calculate the rolling sum is vertically first, and then horizontally. The reason for this is as follows.

The first rolling sum can be calculated by reading data directly from the texton map. Each pixel in the texton map only contains one value – the ID of the texton that pixel belongs to.

The input to the second rolling sum is the output of the first rolling sum – this is an image where each pixel is a texton histogram. Section 7.4.4 suggests that 32 textons be used, implying that each histogram will have 32 values in it. This means that the second rolling sum must read 32 times more data than the first rolling sum, and so its memory access pattern is much more important to the speed of the algorithm.

Choosing to do the second rolling sum horizontally ensures greater locality of reference, because the histograms that need to be added/subtracted for the rolling sum will always be right next to each other in memory. This increases the chance that they will be stored on the same memory page, and also allows for better hardware caching. Hence, choosing to calculate the rolling sum vertically first and horizontally second allows the Texton Ridge Detector to run faster.

SSE acceleration

The SSE instruction set was also used wherever possible to improve the speed of calculating the histograms. In particular, it was found that having a multiple of 32 textons is most efficient. This is because there are eight SSE registers, each storing four values each, and 8 registers × 4 values each = 32 values in total. That is why 32 textons were recommended in section 7.4.4.

The texton gradient is useful for boundary detection, but it needs to be combined with other information to produce robust results. This is explained in the next section.

7.6 COMBINING VISUAL CUES

Using texton ridges in isolation for boundary detection is not an optimal solution for the following reasons. Firstly, some boundaries will change the brightness or colour of an image greatly, but will not change its texture, and so cannot be recognised by a texton ridge. Secondly, texton ridges can appear where no boundaries exist because of the way similar pixels might be binned to entirely different textons – something section 5.5.3 called "phantom boundaries." For these reasons, it is necessary to combine the texton gradient with another visual cue to ensure optimal results.

The previous chapter demonstrated the power of variance as a visual cue. It is fast, and it has the important characteristic that it peaks at boundaries. That is why the Texton Ridge Detector combines the texton gradient with variance. The combination of the two will be called the boundary potential $BP(\boldsymbol{p})$:

$$BP(\boldsymbol{p}) = \|\nabla T(\boldsymbol{p})\| \times V(\boldsymbol{p}) \tag{7.14}$$

There are two reasons why the two visual cues are multiplied together in the above equation, instead of adding them.

Firstly, if the two visual cues were simply added, then the variance would not suppress the phantom boundaries. Those phantom boundaries would still be added into the image from the texton gradient.

Secondly, the variance by itself already makes an excellent boundary detector. Multiplying variance with the texton gradient effectively suppresses variance ridges from occurring where there is no change in the texture. Doing this removes boundaries which would have been detected inaccurately by the Variance Ridge Detector, while preserving the other high-quality boundaries.

7.6.1 IMAGE EXAMPLES

The boundary potentials of the example images, calculated by combining the texton gradients and variances, are shown in Figure 7-7.

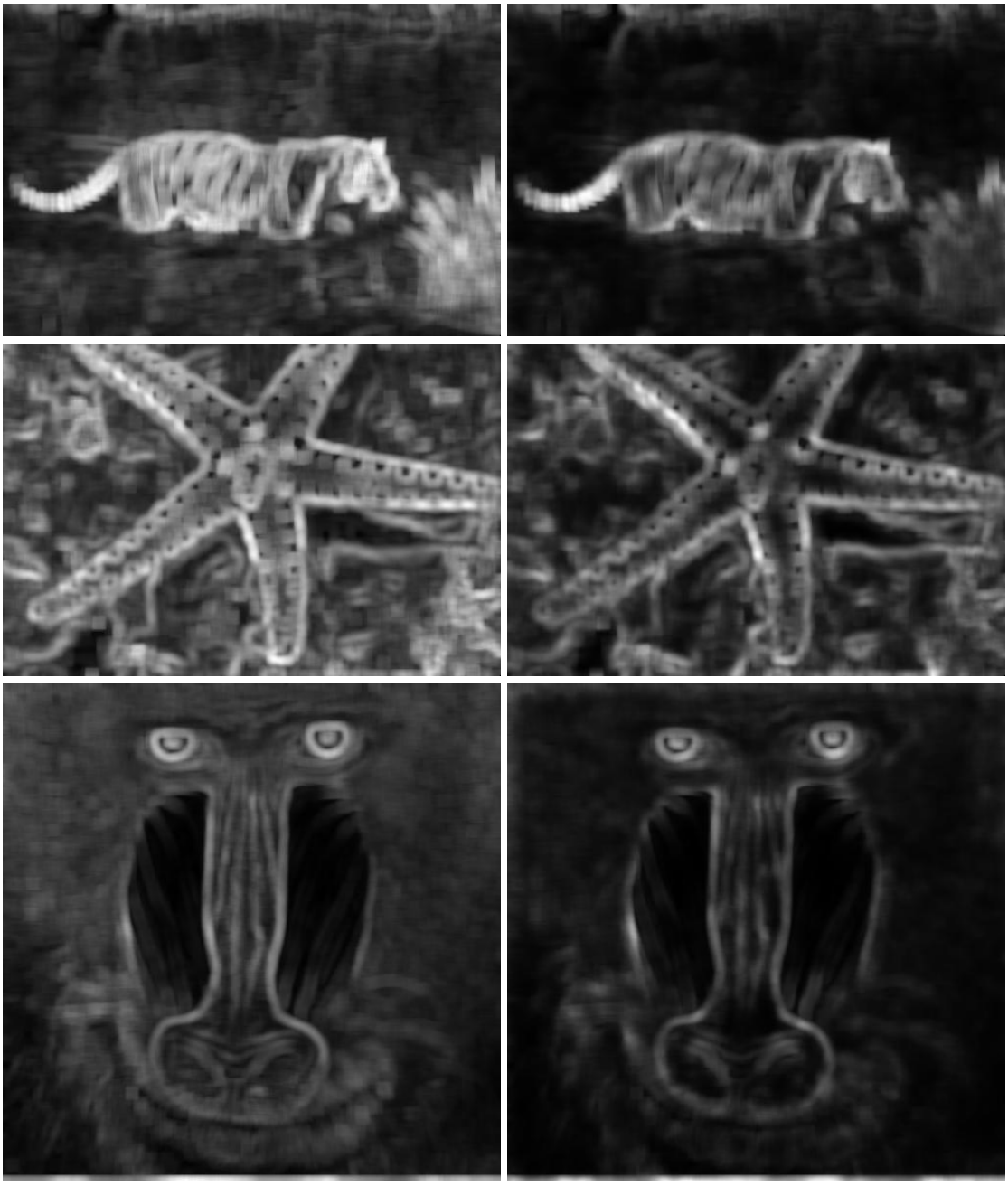


FIGURE 7-7: The combined texton gradient and variance (right) of the example images, versus just variance (left).

The example images above show that when the variance is combined with the texton gradient, more texture is suppressed. For example, the mandrill's fur (bottom row) is suppressed almost entirely when using the texton gradient, whereas it can still be seen in the variance. The water in the tiger's image (top row) is also suppressed when using textons.

7.7 RIDGE DETECTION

Once the boundary potentials are calculated, ridge detection is performed to find boundaries. This proceeds in exactly the same way as with the Variance Ridge Detector, see sections 6.6 to 6.8 for the details.

7.8 IMAGE EXAMPLES

Applying the Texton Ridge Detector to the example images yields the results shown in Figure 7-8:

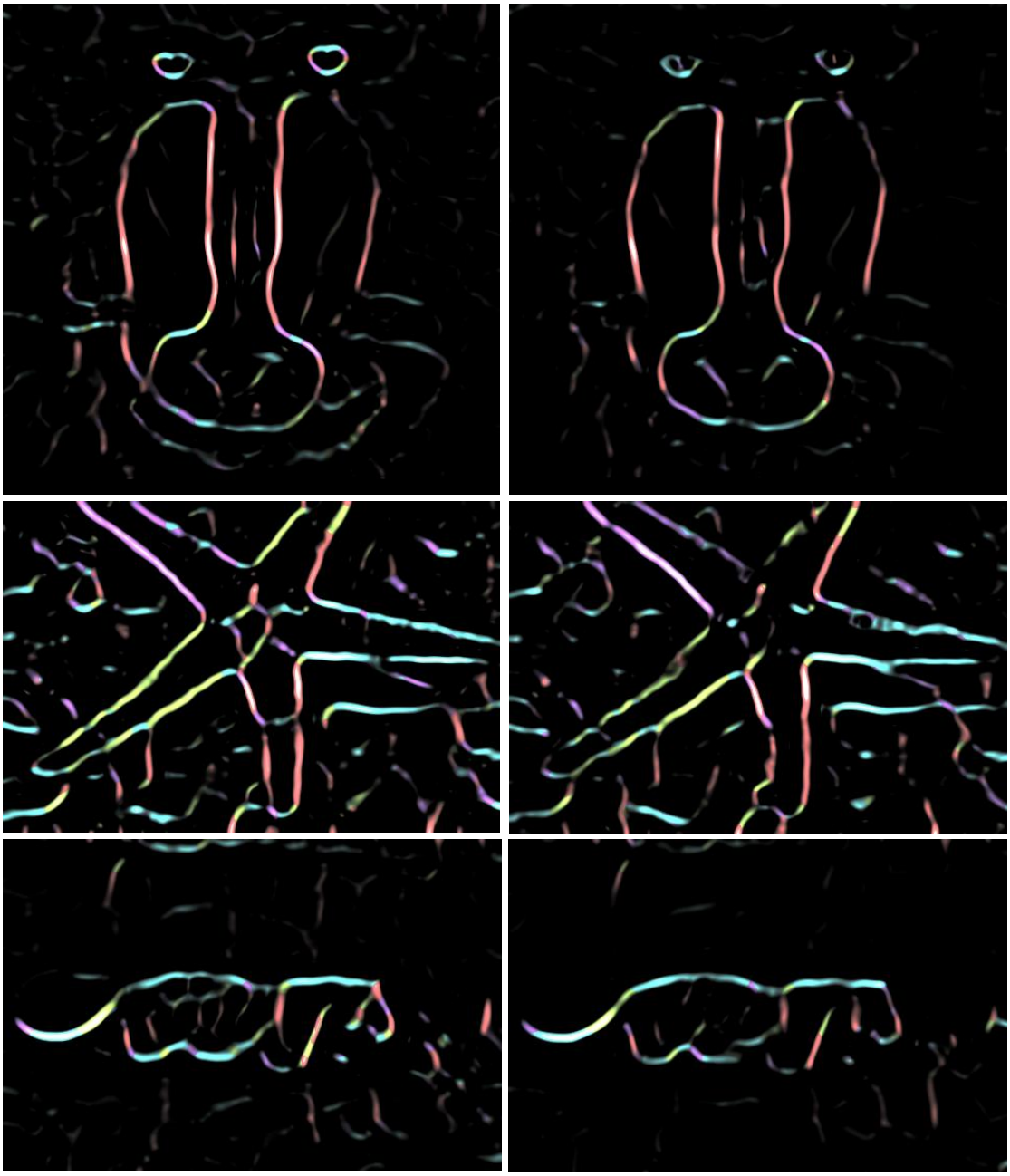


FIGURE 7-8: The Variance Ridge Detector (left column) versus the Texton Ridge Detector (right column). Hue represents boundary orientation. All images have been brightened to make the subtle differences easier to see.

The example results above are similar for both the Variance Ridge Detector and the Texton Ridge Detector. The Texton Ridge Detector has suppressed a few more intra-texture boundaries, but has also suppressed some inter-texture boundaries. Generally speaking, the Variance Ridge Detector focuses on detecting all texture boundaries, while the Texton Ridge Detector focuses on detecting only the boundaries with a high confidence. It is up to the user to choose which algorithm is best for their situation.

7.9 COMPARISON TO PREVIOUS WORK

The proposed Texton Ridge Detector differs from previous work in certain key ways. This subsection will explore what these differences are, and the reasons why these differences improve the algorithm for this situation.

Locality-sensitive hashing

The proposed approximate textonisation algorithm is a type of approximate nearest neighbour search algorithm which uses **locality-sensitive hashing** (LSH) (Indyk & Motwani, 1998; Gionis, Indyk, & Motwani, 1999). Generally, all LSH methods partition the feature space with a set of separating hyperplanes in the same way as the proposed approximate textonisation algorithm. However, the proposed approximate textonisation algorithm differs in a two substantial ways.

Most approximate nearest neighbour algorithms are designed to find the nearest neighbour out of hundreds of thousands of candidate points. That is why LSH algorithms normally have multiple candidates per partition, meaning once the partition is found, some further searching has to occur, and sometimes some backtracking, to produce a suitable result. In this case, there are much fewer candidate points; in fact section 7.4.4 recommended there be only 32 textons. Having so much fewer candidate points means that it is feasible for each partition to only have one candidate, which made it possible for the lookup table optimisation to be used (described in section 7.4.2).

Additionally, LSH algorithms use random hyperplanes, whereas this approximate textonisation algorithm attempts to find the best set of hyperplanes using machine learning techniques.

These optimisations make this approximate textonisation algorithm much more suited to this problem than traditional LSH algorithms.

Boundary detection via Randomised Hashing

Boundary detection via Randomised Hashing (Taylor & Cowley, 2009), introduced in section 5.5, is also based on locality-sensitive hashing, and so it shows some similarities to the Texton Ridge Detector.

Both algorithms use hyperplane splits of the feature space. In Randomised Hashing, the hyperplanes are random, whereas with the Texton Ridge Detector, they are learnt from a training set. Intuitively, randomness makes no guarantee of quality, which is one drawback of Randomised Hashing.

Both algorithms have the potential to introduce phantom boundaries because they quantise features. The Texton Ridge Detector uses variance to eliminate these phantom boundaries, whereas Randomised Hashing does not have any method for dealing with this problem.

Also, the Texton Ridge Detector calculates texton histograms, which allows it to integrate texture information at a higher level. Randomised Hashing does not explicitly do anything beyond low-level processing, which means its results are noisier because it is sensitive to the low-level variations in the image.

Boundary detection via Semantic Texton Forests

Semantic Texton Forest segmentation (Shotton, Johnson, & Cipolla, 2008), introduced in section 5.4, uses a decision forest to transform an image into textons. This textonisation approach is quite different from the proposed approximate textonisation approach.

One reason why decision forests can produce such high-quality results is that, each decision node in each of the decision trees uses the most discriminant feature possible. The problem is, when there are thousands of decisions in the decision forest, there will be thousands of different features. This is the primary reason why decision trees are slow.

Consider an image being textonised using Semantic Texton Forests. Each pixel must follow a different path down each decision tree, which means different features must be calculated for each pixel. This creates an unpredictable memory access pattern. This is a problem because a memory fetch operation is normally 10-100 times slower than a normal CPU operation, and so the inefficient memory access of Semantic Texton Forests slows it down dramatically.

Unlike Semantic Texton Forests, the proposed Texton Ridge Detector uses a limited, fixed set of features. Additionally, every pixel is analysed with the same set of decisions (in this case, each hyperplane is one decision). This creates a predictable memory access pattern, allowing the Texton Ridge Detector run much faster than Semantic Texton Forests. Consequently, unlike Semantic Texton Forests, the proposed Texton Ridge Detector can run at full resolution in real-time.

7.10 CHAPTER SUMMARY

This chapter proposed the Texton Ridge Detector, a texture-boundary detector. It uses the existing state-of-the-art texton approach and applies it to real-time. The next two

chapters will evaluate the proposed boundary detectors against the prior real-time boundary detectors.

8 VALIDATION METHODS

The purpose of this chapter is to introduce three methods which were used in the experiments to compare the proposed detectors with existing work:

- Section 8.1 discusses the Berkeley segmentation dataset and benchmark, which will be used to compare the quality of the proposed boundary detectors against other boundary detectors.
- Section 8.2 discusses the Microsoft Research Cambridge 9-class dataset, which will be used to compare the speed of the proposed boundary detectors against other boundary detectors.
- Section 8.3 discusses an adaptive background learning method, which was used to evaluate the speed of one of the stages of TextonBoost.

The experiments in which these methods are used will be described in the next chapter.

8.1 BERKELEY SEGMENTATION DATASET AND BENCHMARK

The Berkeley segmentation dataset (BSDS) and benchmark (Martin D. , Fowlkes, Tal, & Malik, 2001) is a publicly-available method for objectively measuring the performance of a boundary detector. It will be used in the next chapter to validate the proposed boundary detectors, and compare their performance to other boundary detectors.

The benchmark consists of three hundred 481 by 321 images, separated into a training set of two hundred images and a test set of one hundred images. In addition, every image has several sets of human-labelled boundaries, produced by twelve human subjects. These human-labelled boundary maps form the ground truth which boundary detectors should strive for.

Benchmarking a boundary detector using the Berkeley benchmark produces a precision-recall curve (van Rijsbergen, 1979; Baeza-Yates & Ribeiro-Neto, 1999), which shows how the algorithm performs at different levels of trade-off between precision and recall (this will be explained in section 8.1.5). An algorithm's entire precision-recall curve can be summarised in one value called the F-measure – a number between 0 and 1, where a higher number is better. Both the precision-recall curve and the F-measure will be used to compare the performance of different algorithms.

The authors of the benchmark have run two informative tests to show the range of useful values of the F-measure. First, a random number generator scores $F = 0.43$. So

this is the lower bound of what a boundary detector should score. Second, humans score $F = 0.79$. The reason humans do not score $F = 1.0$ is because boundaries are subjective, and so different humans do not agree exactly as to where the boundaries should be placed. This means, if a boundary detector scored equal to or above this value, it has achieved human performance.

8.1.1 BENCHMARKING ALGORITHM OVERVIEW

The BSDS benchmark rates machine-generated boundary maps by comparing them to the human-labelled boundary maps. Given one machine boundary map and a set of human boundary maps for the same image, the benchmark is calculated via the following algorithm:

ALGORITHM 8-1: The Berkeley benchmarking algorithm for one machine-generated image.

1. Threshold the boundary map at thirty different thresholds to generate thirty different binary boundary maps.
2. For each thresholded boundary map:
    a. Thin the boundary map, using morphological thinning.
    b. Match the machine boundary map with each human boundary map by solving an assignment problem.
    c. Calculate the precision, the recall and the F-measure from the number of matched and unmatched boundaries.
3. Return the precision-recall curve and the maximum F-measure as the final result for that image.

After this algorithm has been run on each image, the Berkeley benchmark averages all the results over all images to calculate the overall precision-recall and overall F-measure for that boundary detector.

The remaining subsections in this section will examine the Berkeley benchmark algorithm in more detail.

8.1.2 THRESHOLDING

The benchmark only functions on binary images, and so thresholding is applied to convert a boundary map into a binary image. The BSDS benchmark thresholds the boundary map at thirty evenly-spaced levels. Each of these threshold levels will become

one point on the precision-recall curve. Thresholding was already been described in section 2.3.1.

8.1.3 THINNING

Next, morphological thinning is applied to each binary boundary map. Thinning is needed because it allows for a simple one-to-one matching with the human-labelled boundary maps. Morphological thinning was already described in section 2.3.3.

8.1.4 MATCHING

The most important part of the Berkeley benchmark is matching stage. This stage takes the machine boundary map and one of the human boundary maps, and compares how close they are. This is repeated for all of the human boundary maps, and the results are combined in the next stage.

It is highly unlikely the two boundary maps will be exactly the same, and so the Berkeley benchmark finds the lowest-cost bipartite matching between the two boundary maps, as illustrated in Figure 8-1:

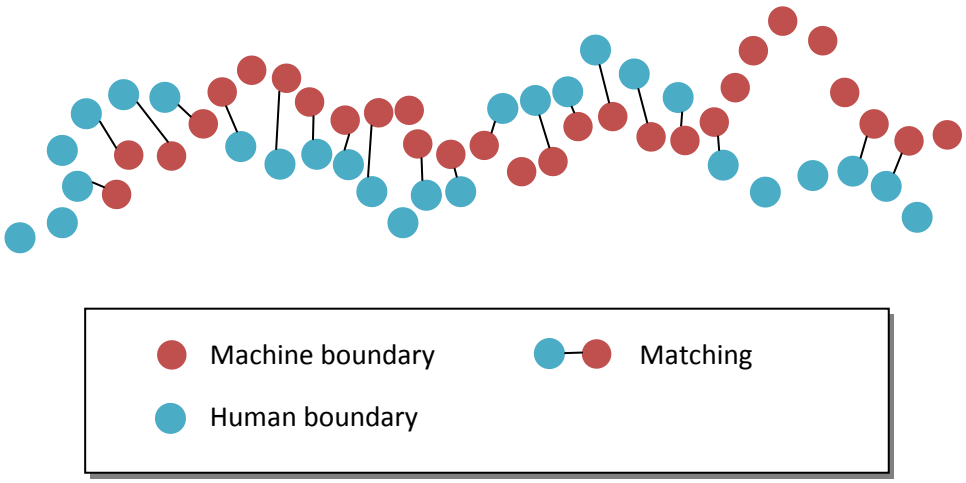


FIGURE 8-1: The machine boundaries are matched to the human boundaries by solving an assignment problem

A bipartite matching is a matching where each of the machine boundary pixels (found from the previous stage) is matched to exactly one of the human boundary pixels. Some of the boundary pixels will be left unmatched. The proportion of boundary pixels that were able to be matched is a measure of the machine's performance on the Berkeley benchmark.

The optimal bipartite matching is found by solving an assignment problem.

8.1.4.1 SOLVING THE ASSIGNMENT PROBLEM

In the assignment problem, there are a number of workers and a number of tasks. The problem is to assign each worker to a task so that the total cost of the assignments is minimised. The Hungarian method, sometimes known as the Kuhn-Munkres algorithm (Kuhn, 1955; Munkres, 1957), is a polynomial-time algorithm for solving the assignment problem optimally. This algorithm will be illustrated with an example.

In this example, there are four workers – A, B, C and D – and four tasks – W, X, Y and Z. Each possible assignment has a different cost, as shown in the cost matrix:

	W	X	Y	Z
A	164	140	80	180
B	100	40	30	140
C	150	80	64	200
D	126	60	52	150

There are six steps to the Kuhn-Munkres algorithm.

Step 1: Subtract the row minimum from each row

Subtract the minimum of each row from each cell in the row:

	W	X	Y	Z
A	84	60	0	100
B	70	10	0	110
C	86	16	0	136
D	74	8	0	98

Step 2: Subtract the column minimum from each column

Subtract the minimum of each column from each cell in the column:

	W	X	Y	Z
A	14	52	0	2
B	0	2	0	12
C	16	8	0	38
D	4	0	0	0

Step 3: Assign greedily

Assign each worker (row) to the first available task (column) that has a zero in it. No task can be assigned to two workers, so a task becomes unavailable once it has been assigned. The assignments are shown in blue:

	W	X	Y	Z
A	14	52	0	2
B	0	2	0	12
C	16	8	0	38
D	4	0	0	0

If all workers have been assigned to a task, then the optimal solution has been found – so the algorithm stops. In this case, worker C could not be assigned to a task, so further processing is required.

Step 4: Assign optimally

Greedy assignment does not find the optimal all-zeroes assignment. For this reason, another step is required to improve the greedy assignment. However, at this point in the example, the greedy assignment cannot be improved, and so this step will be revisited later.

Step 5: Minimum cover

(a) Mark all unassigned rows (shown in red):

	W	X	Y	Z
A	14	52	0	2
B	0	2	0	12
C	16	8	0	38
D	4	0	0	0

(b) Mark all the unmarked columns that have a zero in the rows just marked:

	W	X	Y	Z
A	14	52	0	2
B	0	2	0	12
C	16	8	0	38
D	4	0	0	0

(c) Find all the assignments in the columns just marked, and mark those rows:

	W	X	Y	Z
A	14	52	0	2
B	0	2	0	12
C	16	8	0	38
D	4	0	0	0

(d) Keep repeating from step (b) until no additional rows or columns get marked. In this example, no further rows or columns get marked when repeating these steps.

(e) Draw lines over all *unmarked* rows and all *marked* columns:

	W	X	Y	Z
A	14	52	0	2
B	0	2	0	12
C	16	8	0	38
D	4	0	0	0

The drawn lines will cover all of the assignments, as shown. This is called the minimum cover because it covers all of the assignments with the minimal amount of lines.

Step 6: Reweight matrix

Find the minimum uncovered value. Subtract this minimum from all uncovered elements, and add this minimum to all intersections (where the minimum cover lines cross).

	W	X	Y	Z
A	12	50	0	0
B	0	2	2	12
C	14	6	0	36
D	4	0	2	0

The result of this is a new matrix to be solved. Now the algorithm goes back to step 3 with this new matrix.

Back to step 4: Assign optimally

Step 3 was run, and the following greedy assignment was found:

	W	X	Y	Z
A	12	50	0	0
B	0	2	2	12
C	14	6	0	36
D	4	0	2	0

This assignment can be improved, and that is the purpose of the previously unexplained step 4. The assignment is improved by finding alternating paths. This is best illustrated by reimagining the assignments in the above matrix as a graph:

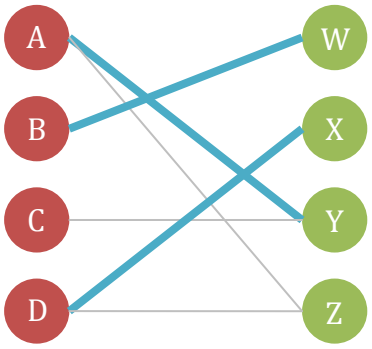


FIGURE 8-2: In this graph, the workers are the nodes on the left, and the tasks are the nodes on the right. The greedy assignments are illustrated on this graph. The strong thick lines are the chosen assignments (transferred from the matrix), the weak grey lines are the possible assignments (the zeroes in the matrix).

An alternating path is a path that traverses the arcs of the graph in an alternating fashion: unassigned, assigned, unassigned, assigned, unassigned… and so on. The path must always begin and end with an unassigned arc. There is only one alternating path in this example, highlighted in Figure 8-3:

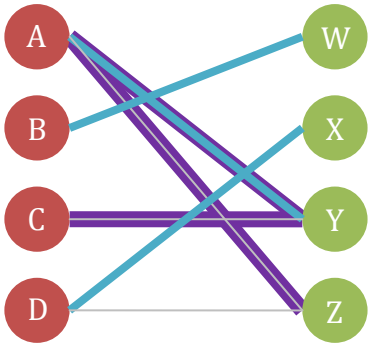


FIGURE 8-3: The only alternating path in this example is highlighted in purple.

Alternating paths can be found by running a breadth-first search algorithm, starting from one of the unassigned workers. When an alternating path is found, the algorithm flips all assignments along the path. As all alternating paths must both start and end on an unassigned arc, all alternating paths will have more unassigned arcs than assigned arcs, and so flipping all assignments along the path will always cause a net increase to the number of assignments – improving the solution. The algorithm repeats this process until no further alternating paths can be found.

Flipping the assignments along the alternating path has the following effect on the example:

	W	X	Y	Z
A	12	50	0	0
B	0	2	2	12
C	14	6	0	36
D	4	0	2	0

In the above table, the elements along the alternating path have been shown with a purple border.

At this point, no further alternating paths can be found, and so the optimal all-zeroes assignment has been found. If some workers were still left unassigned, the algorithm would proceed to step 5. However, in this case, all workers have been assigned, and so the optimal solution has been found, so the algorithm stops here.

8.1.4.2 APPLYING THE ASSIGNMENT PROBLEM TO BERKELEY BENCHMARK

A few steps must be taken to apply the assignment problem to find a bipartite matching of the boundary maps.

First, each boundary pixel in the machine boundary map becomes a "worker" and each boundary pixel in the human boundary map becomes a "task." With these changes, now the assignment problem will find the lowest-cost matching between the machine boundaries and the human boundaries.

Second, the cost of assigning a machine boundary pixel to a human boundary pixel is set so that it is equal to the distance between their positions. The Berkeley benchmark only allows matching between pixels that are at most two pixels apart, and so any possible assignment between boundaries greater than two pixels apart is set to infinite cost.

Third, a high-cost dummy assignment is created for every boundary pixel (human or machine), so if the algorithm cannot find a suitable matching for that boundary, then the boundary will be assigned to this dummy assignment. If a boundary pixel is assigned to this high-cost dummy assignment instead of an actual matching, then that means it was unable to be matched. This is important, because this allows for every boundary pixel in both the human and machine boundary maps to be put into one of two states: matched or unmatched. These two states provide all the information required to calculate the precision/recall, explained in the next section.

8.1.5 CALCULATING PRECISION/RECALL

Precision measures the fraction of machine boundaries that were correct (van Rijsbergen, 1979; Martin D. , Fowlkes, Tal, & Malik, 2001):

$$Precision = \frac{\text{Number of machine boundaries that are correct}}{\text{Number of machine boundaries in total}}$$

The Berkeley benchmark has multiple human boundary maps for each image. If a machine boundary matched a human boundary from at least one of the human boundary maps, then it is considered correct, and so it will contribute to the precision score.

Recall measures the fraction of true boundaries that were found:

$$Recall = \frac{\text{Number of human boundaries that were matched}}{\text{Number of human boundaries in total}}$$

If a human boundary did not match a machine boundary, then it was not successfully recalled, and so is not counted in the recall score. This statistic is calculated individually for each of the human boundary maps, and then the average is taken. That means a machine boundary map must be able to explain the boundaries of all human subjects in

order to achieve a perfect score. This is different from precision – precision only requires one human subject's boundaries to match each of the machine boundaries.

The precision/recall values are calculated individually for each of the thirty threshold levels using the aforementioned process, resulting in thirty precision/recall pairs for each image. These thirty pairs form the precision-recall curve.

8.1.6 THE F-MEASURE

The F-measure(van Rijsbergen, 1979) summarises the precision/recall in one number. It is calculated as follows:

$$F = \frac{PR}{\alpha P + (1-\alpha)R}$$

The F-measure can be modified to consider either precision or recall more important by changing the value of α. The Berkeley benchmark considers both to be equally important, and so it sets $\alpha = 0.5$.

The Berkeley benchmark calculates the F-measure for each of the threshold levels using the precision/recall values calculated previously. The F-measure of a boundary detector on a particular image is equal to its maximum F-measure over all threshold levels for that image. Furthermore, the average F-measure can be calculated over all images in the Berkeley benchmark to indicate the overall performance of a boundary detector.

8.1.7 RESULTS OF THE BERKELEY BENCHMARK

The previous sections described how a boundary detector would be scored on a single image using the Berkeley benchmark. A boundary detector's overall performance over all images in the benchmark can be measured in two ways. First, the average F-measure over all images is used as an overall score for each boundary detector. Second, a precision-recall curve over all images can be constructed by taking the average precision/recall values for each threshold over all images. Both of these forms of results will be used in the next chapter to illustrate the performance of various boundary detectors.

8.2 THE MSRC-9 DATASET

The publicly available Microsoft Research Cambridge 9-class dataset contains 240 images, where each image is comprised of up to nine classes of objects: cow, horse, sheep, tree, building, aeroplane, face, car or bike. There are also two non-object classes:

sky and grass. This dataset will be used in the next chapter to validate the speed of various boundary detectors. Figure 8-4 shows a selection of images from this dataset.

FIGURE 8-4: A random selection of images from the MSRC-9 dataset

This dataset was chosen because of a number of reasons:

1. It is publicly available, and so it provides a platform for other researchers to compare their results.
2. Every image has a human-labelled ground truth, which labels each pixel according to which of the nine classes it belongs to. Without this information, some algorithms such as TextonBoost, Semantic Texton Forests and TextonRML, would not be able to be tested against this dataset.
3. It has a comparatively small number of classes when compared to other common options such as the MSRC-21 dataset or PASCAL VOC2008 dataset. The argument is that real-time applications are likely to be trained on fewer classes, enabling less computational demands and higher speeds. With fewer classes, the MSRC-9 dataset meets this requirement.
4. The image size is 320 by 213, which slightly smaller than the commonly-used camera resolution of 320 by 240, and so the results are indicative of how these algorithms might perform when using real-time input from a camera.

All of the above reasons meant the MSRC-9 dataset was a good choice to measure boundary detector speed. It would have been useful to also use the Berkeley dataset for this purpose, but that is not possible because the Berkeley dataset does not provide class-labelled ground truths which TextonBoost requires.

The use of the MSRC-9 dataset will be seen in the next chapter.

8.3 ADAPTIVE BACKGROUND LEARNING

An adaptive background learning algorithm will be used in the next chapter to test the speed of the minimum cut – a critical part of TextonBoost. The adaptive background

algorithm that will be used was actually a novel innovation developed as a side-project during the course of this research.

The novel part about this adaptive background learning algorithm is that it learns each pixel at a different rate, depending on a novel concept called **stability**.

8.3.1 OVERVIEW

The adaptive background learning algorithm takes a single frame of input $I(\boldsymbol{p})$ and a learning rate parameter η, and returns an error image $E(\boldsymbol{p})$. $E(\boldsymbol{p})$ is the difference between the frame $I(\boldsymbol{p})$ and the background $B_1(\boldsymbol{p})$.

8.3.2 STABILITY

The most important concept in this adaptive background learning algorithm is **stability**. Stability is dependent on the temporal variance of the error image $E(\boldsymbol{p})$. This is illustrated in Figure 8-5:

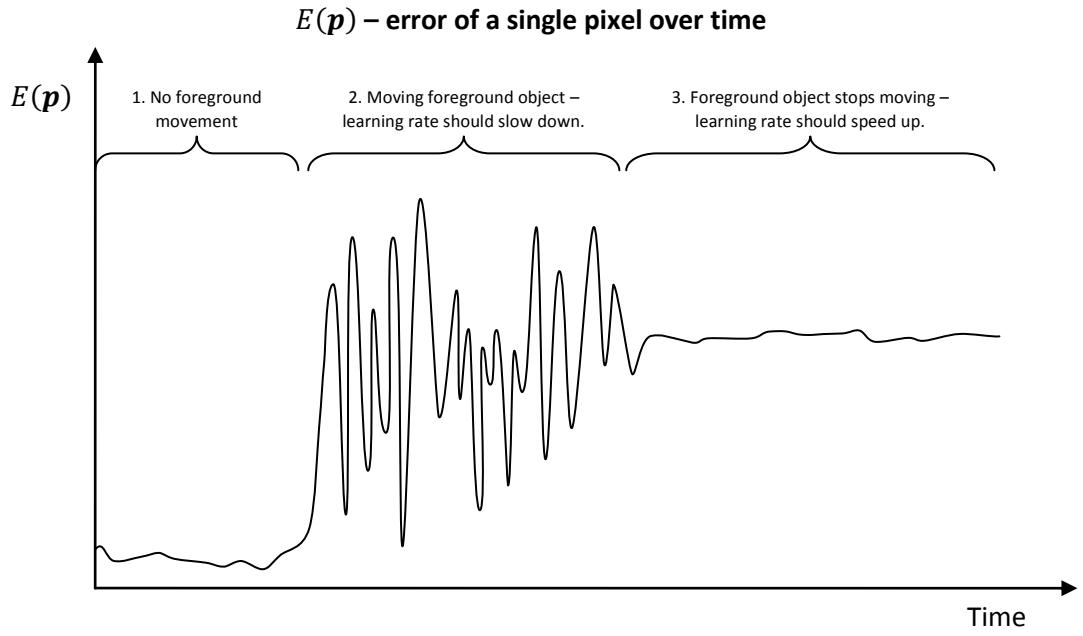


FIGURE 8-5: This diagram is an illustration of how the error of a pixel would change when a new object is added into the background.

Figure 8-5 illustrates how the error of a pixel would change when a new object is added into the background. There are three phases, each indicated in Figure 8-5 by its number:

1. Initially, when there is no movement in the frame, the error will be low and the variance of the error will also be low.

2. When there is foreground movement, there will be high temporal variance in the error, as the pixel will be constantly changing as the object moves through the image.
3. Finally, the foreground object stops moving and it becomes part of the background. This causes the temporal variance of the error to become low again.

So the likelihood of a pixel being part of the background depends on the temporal variance of the error. Stability is a calculation that transforms the variance of the error to a score in the [0, 1] range. The formula for stability will be introduced later.

8.3.3 BACKGROUND MODEL

On each invocation, the algorithm updates a number of variables, which store the current model of the background.

Two images are stored in order to keep track of the background mean and standard deviation:

- $B_1(\boldsymbol{p})$ is the mean background.
- $B_2(\boldsymbol{p})$ is the mean of the squared background.

Two more images are stored in order to keep track of each pixel's stability:

- $E_1(\boldsymbol{p})$ is the mean error.
- $E_2(\boldsymbol{p})$ is the mean of the squared error.

Storing the squares of both the background and the error means that the standard deviation of both the background and the error can be calculated at all times. Initially, all four variables above are initialised with zero-filled images at the start of the algorithm.

8.3.4 ALGORITHM

On each new captured frame $I(\boldsymbol{p})$, the adaptive background model is learnt using the Algorithm 8-2:

ALGORITHM 8-2: The adaptive background subtraction algorithm for a single new frame $\boldsymbol{I}(\boldsymbol{p})$.

1. Let the background standard deviation $B_\sigma(\boldsymbol{p}) = \sqrt{B_2(\boldsymbol{p}) - B_1(\boldsymbol{p})^2}$
2. Calculate the frame error $E(\boldsymbol{p}) = \frac{I(\boldsymbol{p}) - B_1(\boldsymbol{p})}{B_\sigma(\boldsymbol{p})}$
3. Calculate the stability image by performing the following steps:
   - 3.1. Update average error $E_1(\boldsymbol{p}) \leftarrow \eta E(\boldsymbol{p}) + (1-\eta)E_1(\boldsymbol{p})$
   - 3.2. Update average square error $E_2(\boldsymbol{p}) \leftarrow \eta E(\boldsymbol{p})^2 + (1-\eta)E_2(\boldsymbol{p})$
   - 3.3. Let the error standard deviation $E_\sigma(\boldsymbol{p}) = \sqrt{E_2(\boldsymbol{p}) - E_1(\boldsymbol{p})^2}$
   - 3.4. Calculate stability image $S(\boldsymbol{p}) = e^{-E_\sigma(\boldsymbol{p})^2}$
4. Update the background model using the stability matrix:
   - 4.1. Update average background:

     $B_1(\boldsymbol{p}) \leftarrow \big(\eta S(\boldsymbol{p})\big) I(\boldsymbol{p}) + \big(1 - \eta S(\boldsymbol{p})\big) B_1(\boldsymbol{p})$
   - 4.2. Update average square background:

     $B_2(\boldsymbol{p}) \leftarrow \big(\eta S(\boldsymbol{p})\big) I(\boldsymbol{p})^2 + \big(1 - \eta S(\boldsymbol{p})\big) B_2(\boldsymbol{p})$
5. Return the frame error image $E(\boldsymbol{p})$

As $E(\boldsymbol{p})$ indicates the likelihood of pixel $\boldsymbol{p}$ belonging to the foreground, a minimum cut can be used on $E(\boldsymbol{p})$ to separate the foreground from the background. This is how this algorithm is used in the next chapter.

9 EXPERIMENTAL RESULTS

This thesis proposes two texture boundary detectors – the Variance Ridge Detector and the Texton Ridge Detector. This chapter will compare the proposed detectors against other existing texture-boundary detectors.

9.1 OVERVIEW OF THE EXPERIMENTS

The first group of experiments validates whether the proposed detectors can be considered real-time, while the second group investigates the quality.

Real-time

The speed of the two proposed boundary detectors was measured with two experiments. First, the proposed detectors processed the real-time input from a camera. Second, the proposed detectors processed images from the publicly-available Microsoft Research Cambridge 9-class (MSRC-9) database. These two experiments will determine whether the proposed boundary detectors are capable of running in real-time.

For comparison, the speeds of the gPb detector and TextonBoost were both measured on the same MSRC-9 database as the proposed detectors. This will investigate whether it is true that these state-of-the-art detectors cannot run in real-time.

Speed measurements were not taken on the existing real-time texture-boundary detectors as their authors have already shown them to run in real-time.

Quality

The quality of the two proposed boundary detectors was measured on the publicly-available Berkeley segmentation dataset (BSDS), using the benchmarking algorithm provided with the dataset. This benchmark compares the output of a boundary detector against a database of human-segmented images. The purpose of this experiment is to objectively measure the quality of boundary maps produced by the proposed boundary detectors.

For comparison, the qualities of the Konishi detector and Surround Suppression have also both been measured on the Berkeley segmentation dataset and benchmark. These results will investigate whether the proposed detectors produce higher quality results than these other real-time algorithms.

The remaining real-time detectors – TextonRML (section 5.3), Semantic Texton Forests (section 5.4) and Randomised Hashing (section 5.5) – were not able to be benchmarked

on the Berkeley benchmark. Chapter 5 already showed that these algorithms cannot produce high-quality boundary detections.

The combination of all of the above results will show whether the two boundary detectors proposed by this thesis outperform other established texture-boundary detectors.

9.2 APPARATUS

The experiments were run on three separate machines.

The first machine had an Intel Core 2 Duo E6750 2.66 Ghz 32-bit CPU, 2 GB of RAM and used Windows XP Professional with service pack 3. This machine was used for all of the speed measurements. Even though this machine has two CPU cores, all speed measurements were executed as single-threaded applications and so only one CPU core was used in those cases.

The second machine was identical to the first, except it ran Fedora Core 8 instead of Windows XP. This machine was used to run the Berkeley benchmark, as the Berkeley benchmark required a Linux environment.

The third machine was a server which had sixteen Intel Xeon MP 2.70 Ghz CPUs, each with one CPU core only, and 32 GB of RAM. It ran Windows Server 2003 Enterprise edition with service pack 2. This machine was used for training classifiers, as its many CPUs and large memory size sped up the training time significantly.

Whenever real-time camera input was needed, a Logitech QuickCam 5000 was used, connected via USB.

The C++ programs used by the experiments were all compiled using Visual C++ 2008. When speed was being measured, the programs were compiled with maximum code optimisation and linked with whole program optimisation.

Some of the Boost C++ libraries[1] (version 1.38) were used in the C++ implementations, particularly the smart pointers, the filesystem libraries and timer library.

OpenCV[2] 1.1 was used for most experiments, but OpenCV 2.0 was used in some cases. This was because OpenCV 2.0 only became available more recently. Not all functions in OpenCV 1.1 were used as they are not SSE-accelerated (see section 6.10.5), and so

[1] http://www.boost.org/

[2] http://www.opencv.org/

separate SSE-accelerated functions were implemented. The functions in OpenCV 2.0 are generally already SSE-accelerated, and so they were used directly.

OpenMP[3] was used to create multithreaded implementations of some of the detectors. However, this was only used to speed up training – only a single thread was used when performing boundary detection. This was done to ensure that if a boundary detector was found to achieve real-time, it would be because of the algorithm's design and not because of the number of CPUs it was running on.

MATLAB R2007b was used to run the Berkeley benchmark and the normalised cut algorithm. Although MATLAB itself is slower than C++, the most computationally intensive parts of the MATLAB applications were implemented in other languages. The eigenvectors were solved using the ARPACK library[4], compiled natively from Fortran90, and C was used for all other computationally-intensive parts.

Finally, one of the programs was implemented using C++/CLI, compiled with Visual C++ 2008. Essentially, this language allows a programmer to write both native code (in a C++ manner) and managed code (in a .NET) manner together in the same program. As expected, the managed code is slower than the native code. However, this language was not used in any of the cases where speed was being measured.

9.3 SPEED OF PROPOSED DETECTORS ON REAL-TIME CAMERA INPUT

The proposed detectors are intended to be used on real-time camera input. This experiment measures the speed of the proposed detectors in that intended setup, and will investigate whether the detectors are capable of running in real-time.

9.3.1 APPARATUS

This experiment was run on the Windows Intel Core 2 Duo 2.66 Ghz machine using a Logitech Quickcam 5000, both of which were previously introduced in greater detail in section 9.2.

9.3.2 METHOD

The Variance Ridge Detector and the Texton Ridge Detector were implemented as described in chapters 6 and 7. They were compiled with Visual C++ 2008, using OpenCV

[3] http://www.openmp.org/

[4] http://people.sc.fsu.edu/~burkardt/m_src/arpack/arpack.html

1.1 and some of the Boost C++ libraries. SSE instructions were used wherever possible to speed up the implementations.

The Texton Ridge Detector was trained with 32 textons, using 20 hyperplane splits. The training data was 200-image training set from the Berkeley benchmark. Training took approximately eight hours, by which time the approximate textonisation algorithm had achieved an accuracy of 91%.

The average execution speed of the detectors was measured over 100 000 frames, captured in real-time from the camera. This was repeated using two different frame sizes: 320 by 240 pixels, and 640 by 480 pixels. For 320 by 240 pixels, the scaling parameter was set to $r = 3$, as this generates the highest-quality results on the Berkeley benchmark. For 640 by 480 pixels, the scaling parameter was doubled to $r = 6$.

Only the time spent processing images was included. That means the time capturing the images from the camera was not included in the execution time, as this is not part of the algorithm. The Boost timer library was used to measure time. It was found that the CPU only had a time granularity down to steps of approximately 0.015 seconds. So to measure the speed accurately, each frame was captured and the detector was run on it repeatedly until the elapsed time was at least one second, and the speed measured over that time period. This ensured the time granularity would not introduce errors into the results.

9.3.3 RESULTS

The speed of the proposed detectors on 320 by 240 images is shown in Table 9-1, while the speed of the proposed detectors on 640 by 480 images is shown in Table 9-2.

Detector	**Seconds per frame**	**Frames per second**
Variance Ridge Detector	0.021	47.9
Texton Ridge Detector	0.094	10.6

TABLE 9-1: The speed results of the proposed detectors on 320 by 240 images

Detector	**Seconds per frame**	**Frames per second**
Variance Ridge Detector	0.12	8.12
Texton Ridge Detector	0.46	2.19

TABLE 9-2: The speed results of the proposed detectors on 640 by 480 images

9.3.4 DISCUSSION

Clearly, the proposed detectors are able to run in real-time for the 320 by 240 images. For the 640 by 480 images, the detectors run between one and ten frames per second, and so could still be considered real-time depending on the application.

The Texton Ridge Detector is approximately 4.5 times slower than the Variance Ridge Detector. This is understandable as it not only has to run the Variance Ridge Detector itself, but it also must textonise the images and calculate their texton gradient.

When the resolution was doubled, the Variance Ridge Detector slowed down by a factor of 5.9, while the Texton Ridge Detector slowed down by a factor of 4.8. Part of this can be explained by the fact that doubling the resolution from increases the number of pixels by four times. The remainder of the slowdown factor is likely to be due to the fact that the scale parameter r was doubled, which quadrupled the size of the sliding window. This does not cause a further quadrupling of the slowdown factor though because rolling sums (described section 7.5.4) were used throughout the implementations.

9.4 SPEED MEASUREMENTS ON MSRC-9 DATABASE

The speed of TextonBoost (section 4.4) and the Pb detector (section 4.5) were both measured on the MSRC-9 dataset. The purpose of this was to investigate whether these state-of-the-art algorithms are capable of running in real-time. For comparison, the results of the proposed boundary detectors on this dataset have also been measured.

9.4.1 APPARATUS

All speed measurements were made on the Windows Intel Core 2 Duo 2.66 Ghz, which was already introduced in section 9.2. Some of the training was done on the sixteen-CPU server machine, also introduced in section 9.2. The MSRC-9 dataset (described in section 8.2) was used for testing.

9.4.2 METHOD

The same implementations of the Variance Ridge Detector and Texton Ridge Detector were reused from the previous experiment (section 9.3), they were just adapted so that they could take input from files. The Texton Ridge Detector was not retrained from the previous experiment, the same one was used. The scale parameter r was set to 3 for both of the proposed detectors.

The rest of this section will describe how the implementations used for the probability of boundary detector and TextonBoost, and then will finally describe how the speed benchmark was run.

The probability of boundary detector

Two versions of the probability of boundary (Pb) detector were used. First, a MATLAB implementation of the Pb detector, which was made publicly available by its authors, was used. Second, a C++ implementation of part of the Pb detector was developed.

The C++ implementation of Pb was compiled with Visual C++ 2008, using the libraries OpenCV 2.0.0a and some of the Boost C++ libraries. It only included the two core parts of Pb – the texton extraction stage and the texture gradient calculation stage. All possible optimisations were used:

- The feature extraction stage was implemented using OpenCV's filter2D method, which uses the discrete Fourier transform over different tiles throughout the image to achieve maximum speed.
- The textonisation stage uses a kd-tree, implemented with the Fast Library for Approximate Nearest Neighbours (FLANN) included with OpenCV 2.0.
- The texton frequencies for ω oriented half-discs is calculated by first counting the texton frequencies over 2ω slices of the disc, and then taking the rolling sum over the slices.
- The slices themselves are calculated by using precalculated slice masks for the first column, and then difference masks for the remaining columns.

All of the above points mean that this C++ implementation will demonstrate a lower bound for the speed of the Pb detector.

All parameters were set to the optimal values presented in Pb's 2004 paper. That is, $\omega = 8$ orientations were used, and all scales were set to 2% of the image diagonal except for the brightness gradient, which was set to 1% of the image diagonal. Both implementations were trained on the Berkeley 200-image dataset. See section 4.5 for further details on the Pb algorithm.

TextonBoost

Two implementations of TextonBoost were also used. First, a C# implementation of TextonBoost, provided by its authors, was used. Second, a C++ implementation of part of the TextonBoost algorithm was developed.

The C# implementation did not include the alpha-expansion graph cut stage, which means it runs slightly faster than it would normally.

The C++ implementation of TextonBoost was compiled with Visual C++ 2008, using the libraries OpenCV 2.0.0a and some of the Boost C++ libraries. It only included the two core parts of TextonBoost – the texton extraction stage and the texture-layout filter stage. All possible optimisations were used:

- TextonBoost's feature extraction stage happens to use linearly separable kernels. This allowed them to be applied separately in turn for each dimension – first across the rows and then across the columns – which is much faster than having to use a two-dimensional sliding window.
- Convolution is commutative, which means that applying the Laplacian of the Gaussian kernel to an image is the same as applying the Gaussian kernel to an image, and then applying the Laplacian operator to the Gaussian-filtered image. For this reason, the image was only filtered with Gaussian kernels, and then the Laplacian was applied afterwards. The same was done for the first-derivative of the Gaussian, where the derivative was calculated on the Gaussian-filtered image using the Sobel filter. This reduced the number of convolutions threefold, allowing for greater speeds.
- The textonisation stage uses a kd-tree, implemented with the Fast Library for Approximate Nearest Neighbours (FLANN) included with OpenCV 2.0.
- As the original TextonBoost paper prescribed, integral images were calculated for each texton to maximise the speed of the texture-layout filters.

All of the above points mean that this C++ implementation will demonstrate a lower bound for the speed of the TextonBoost detector.

Both implementations of TextonBoost were as trained with two sets of parameters.

The first set of parameters was 100 textons and 500 texture-layout filters. These parameters have been chosen to be the identical to Ranganathan's work (2009). Ranganathan's TextonRML was intended for real-time segmentation, and so their choice of parameters would have been optimised for real-time.

The second set of parameters was 400 textons and 5000 texture-layout filters. These parameters were the same as in TextonBoost's 2009 paper (Shotton J. , Winn, Rother, & Criminisi, 2009). When the C# implementation was trained with these parameters, it only produced 760 texture-layout filters because it could not improve its accuracy after

this point. Consequently, the C++ implementation was restricted to 1000 texture-layout filters in order to allow for effective speed comparisons to be made. It is likely that the C++ implementation could achieve more texture-layout filters because it was trained on the much more powerful sixteen-CPU server and so higher quality settings were used for its training.

TextonBoost should run faster (but less accurately) for the first set of parameters as they are smaller. Having both of these sets of parameters is useful as it demonstrates the performance of TextonBoost when either speed or quality is emphasised.

Both implementations were trained on a 48-image subset of the MSRC-9 dataset. Both implementations subsampled the training data by a factor of 5 in order to achieve acceptable training times.

Speed benchmark

Each of the algorithms was run ten times on each image in the MSRC-9 dataset, and the average speed was taken. In all cases, only the time spent processing images was included. That means the time loading the images from disk was not included in the execution time, as this is not a part of the algorithms. All measurements were taken on the Windows Intel Core 2 Duo 2.66 Ghz machine which was already described in section 9.2.

9.4.3 RESULTS

The results are presented in the table below:

Detector	Seconds per frame	Frames per second	Notes
Variance Ridge Detector	0.023	43.6	Proposed
Texton Ridge Detector	0.094	10.7	Proposed
Pb (C++)	2.78	0.36	
TextonBoost (C++) (100 textons, 500 texture-layout filters)	4.37	0.23	
TextonBoost (C++) (400 textons, 1000 texture-layout filters)	8.59	0.012	
Pb (MATLAB)	12.6	0.079	
TextonBoost (C#) (100 textons, 500 texture-layout filters)	24.9	0.040	
TextonBoost (C#) (400 textons, 760 texture-layout filters)	44.2	0.022	

TABLE 9-3: The speed results of various algorithms on the MSRC-9 dataset

Image results

Although this experiment was only measuring speed, for interest, some of the image results are presented below in Figure 9-1.

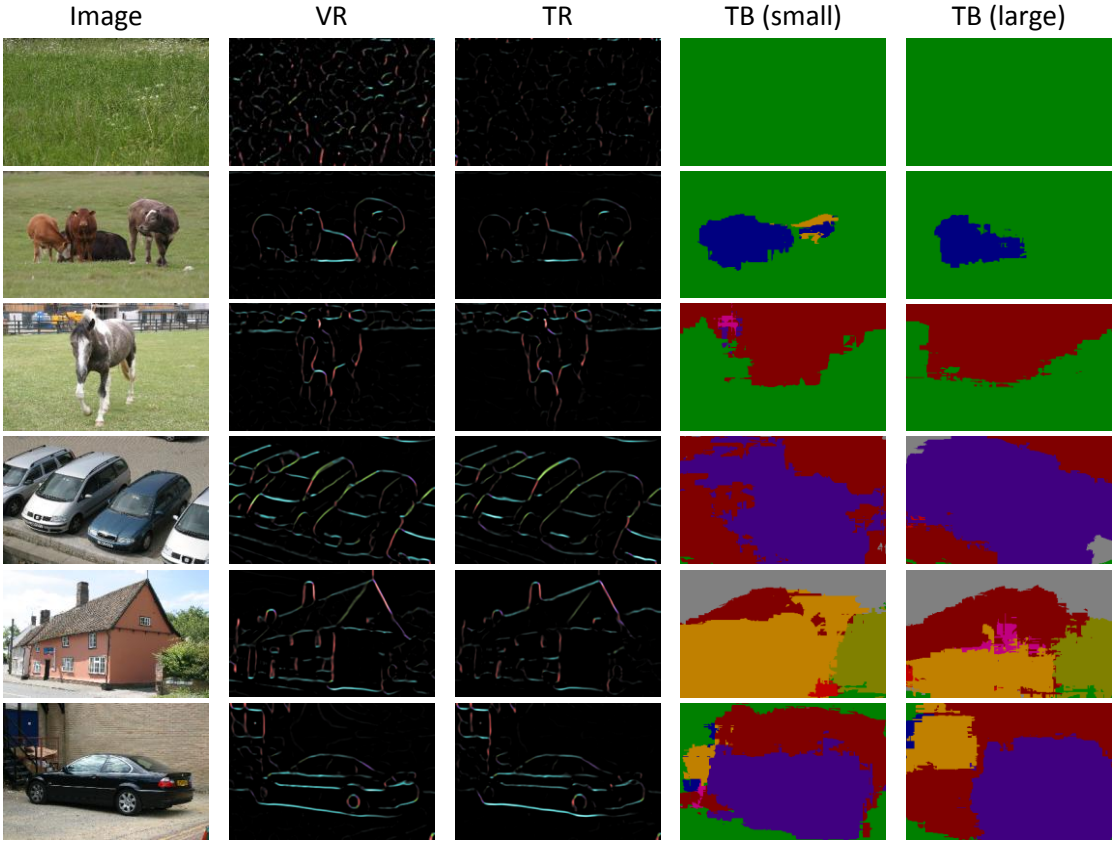


FIGURE 9-1: From left to right: the original image, the Variance Ridge Detector (proposed), the Texton Ridge Detector (proposed), TextonBoost (100 textons, 500 texture-layout filters), TextonBoost (400 textons, 760 texture-layout filters). These are the output of the C# implementation of TextonBoost, and so they do not include an alpha-expansion graph cut. Original images all from the MSRC-9 image database.

9.4.4 DISCUSSION

Clearly the results show that both TextonBoost and the Pb detector cannot run in real-time, even when only the core parts of the algorithms are implemented optimally. The results also clearly show that the Variance Ridge Detector and Texton Ridge Detector can clearly achieve real-time.

Interestingly, the Variance Ridge Detector is slightly slower here than when running on camera input as in the last experiment, even though the images in the MSRC-9 dataset are smaller. The experiment has been rerun and this pattern has been confirmed consistently. It is not clear why this is. Perhaps the image is cached differently when it comes from the camera as opposed to the hard drive. The Texton Ridge Detector does run slightly faster in this case though, as expected.

The C++ implementations of TextonBoost and Pb show large speedups over their C# or MATLAB counterparts, with Pb running 4.5 times faster, and TextonBoost running about 6 times faster. A significant part of this would be due to the fact that the C++ implementations do not include all stages of the algorithms. However, the fact that even these highly optimised stripped-down versions of the detectors cannot run in real-time shows that there is no way any version of TextonBoost or Pb can run in real-time.

TextonBoost also shows that it slows down linearly with the number of texture-layout filters, as expected. The results show that when the number of texture-layout filters were approximately doubled, the execution time approximately doubled as well.

Another important point is that, in Figure 9-1 it can be seen that TextonBoost produces extremely low-quality boundaries without its alpha-expansion graph cut stage. This is significant, because some texton-based algorithms claim they can achieve real-time execution by simply not running the graph cut stage, clearly with substantially lower results. This observation was discussed further in section 4.4.3.

These results show that the state-of-the-art texture-boundary detectors Pb and TextonBoost cannot run in real-time, as they take well over one second per frame. By contrast, the results have also shown that the proposed detectors can run in real-time.

9.5 ESTIMATING THE SPEED OF GPB

The global probability of boundary detector (gPb) was introduced as a state-of-the-art non-real-time texture-boundary detector in section 4.6. This section presents an experiment which will investigate whether it is possible for gPb to run in real-time.

The gPb algorithm consists of two halves. The first half of the algorithm is the same as the probability of boundary detector (Pb). The second half improves the boundary map by using global information. Section 9.4 already showed that Pb detector, which was the first half, cannot run in real-time. The purpose of this experiment is to show that the second half also cannot run in real-time.

The code for the gPb algorithm was not made publicly available. So instead of running the whole of gPb itself, this experiment runs only the most computationally-expensive part of the second half of gPb – the part where the eigenvectors are calculated. This will strongly indicate whether it is possible to run the second half of gPb in real-time.

9.5.1 APPARATUS

This experiment was run on the Fedora Core Intel Core 2 Duo 2.66 Ghz machine that was introduced in section 9.2.

9.5.2 METHOD

The normalised cut (Shi & Malik, 2000) is very similar to the second half of gPb – it involves calculating eigenvectors in the same way. Shi and Malik made their MATLAB implementation of the normalised cut publicly-available, and so this code was taken and modified to simulate eigenvector part of gPb.

Traditionally, the first three steps of the normalised cut are as follows:

1. Calculate the gradients
2. Construct the affinity matrix
3. Find eigenvectors in the affinity matrix

This is identical to gPb, except gPb replaces step 1 with the Pb detector. So, the code of the normalised cut was modified so that step 1 would in fact use the Pb detector. This was possible because the code for Pb had been made available by its authors. Making these modifications meant the code was identical to gPb up to the eigenvector stage. The gPb detector does further postprocessing after this stage, but these parts were not coded for this experiment. That means this experiments will only provide a lower bound of what the speed of gPb could be, which is acceptable for this purpose. The full details about the gPb algorithm can be found in section 4.6.

In this experiment, the partial gPb algorithm that was described above was run ten times on each image in the MSRC-9 dataset (explained in section 9.4), and the average speed of the eigenvector stage only (that is step 3 only, above) was taken. That means the time spent loading the images from disk or performing steps 1 or 2 above was not included in the execution time.

It was considered acceptable to use a MATLAB implementation for this speed test because the eigenvector stage, which is the only stage that is speed tested, calls natively compiled Fortran90 code.

9.5.3 RESULTS

Running the experiment showed that it would take an average of 30.0 seconds for gPb to calculate the eigenvectors for the images in the MSRC-9 dataset.

9.5.4 DISCUSSION

Thirty seconds is clearly slower than real-time, and so this experiment shows that gPb cannot run in real-time. This long execution time makes sense, as the solver must consider an extremely large number of pixel pairs. As the Pb algorithm, which constitutes the first half of gPb, was also shown to be non-real-time in section 9.4, this shows that running the entire gPb algorithm in real-time is impossible.

9.6 ESTIMATING THE SPEED OF ALPHA-EXPANSION GRAPH CUTS

Section 9.4 measured the speed of only part of TextonBoost. This section presents an experiment that attempts to indicate whether the remaining part of TextonBoost – the alpha-expansion graph cut stage – could be run in real-time. This is important because if alpha-expansion graph cuts can run in real-time, then they could be used with the real-time algorithm TextonRML (introduced in section 5.3), allowing for extremely high-quality boundary detection in real-time.

As shown in section 4.4.5, the alpha-expansion graph cut is simply a minimum cut with special inputs. So to investigate whether alpha-expansion graph cuts can be solved in real-time, this experiment will investigate whether it is possible to run multiple minimum cuts in real-time.

9.6.1 APPARATUS

This experiment was run on the Windows Intel Core 2 Duo 2.66 Ghz machine using the Logitech QuickCam camera, both of which were introduced previously in section 9.2.

9.6.2 METHOD

A system, involving a camera connected to a computer, was setup to watch a particular scene. The scene was simply an outdoor scene at the University where this research was undertaken, as shown in Figure 9-2. The scene contained a path, and people would frequently walk through the scene along the path. The system would continually learn the background of the scene using an adaptive background learning algorithm (described in section 8.3).

FIGURE 9-2: A still frame of the scene that was used for this experiment.

Whenever the total difference from the background exceeded a certain high threshold, the system would assume that a person had entered the scene, and so it would use a minimum cut to separate the person from the learnt background. From this, the average speed of the minimum cut algorithm was measured over a large number of frames. The system was left running over the course of one day.

The minimum cut algorithm that was the Boost C++ graph library implementation of the Kolgomorov algorithm (Boykov & Jolly, 2001; Boykov & Kolmogorov, 2004). Kolmogorov's algorithm was specifically designed with computer vision applications in mind, and so is currently the fastest known minimum cut algorithm for this purpose.

Only the speed of the minimum cut was measured. That means, the time spent capturing the image, learning the background or transforming the image into a graph was not included in the execution time. The minimum cut would always run in its own thread, while the rest of the program ran in another thread. This ensured the adaptive background model was correct at all times. The two threads would not have interfered with each other as the CPU had two separate cores.

9.6.3 RESULTS

Over eight hours of execution, the program ran the minimum cut over two thousand different frames of movement. It was found that it took 0.61 seconds on average to run the minimum cut algorithm.

9.6.4 DISCUSSION

An algorithm that uses alpha-expansion graph cuts, such as TextonBoost, must run the minimum cut algorithm at least as many times as the number of classes. The MSRC-9 dataset contains eleven classes (nine of them are object classes), and so at least eleven

minimum cuts would need to be run. Given that each cut takes 0.61 seconds, this would take approximately 6.71 seconds, which clearly cannot be considered real-time. Even if only two classes were used, this process would take 1.2 seconds, which would be too slow for most real-time applications.

Combining the results of this experiment with the fact that the first half of TextonBoost's algorithm also does not run in real-time (see section 9.4), this experiment shows that it is impossible for an algorithm such as TextonBoost to run in real-time.

9.7 QUALITY MEASUREMENTS ON BERKELEY BENCHMARK

The quality of the proposed detectors, Konishi's detector and Surround Suppression was measured using the Berkeley segmentation dataset and benchmark (explained previously in section 8.1). This is to show that the proposed detectors outperform both of these state-of-the-art real-time texture-boundary detectors.

Code for other real-time texture-boundary detectors was not publicly available. So instead, a visual comparison will be made in the next section. Together, this section and the next section will investigate whether the proposed detectors outperform all other real-time texture-boundary detectors in terms of quality.

9.7.1 APPARATUS

This experiment was run on the Fedora Core Intel Core 2 Duo 2.66 Ghz machine introduced in section 9.2.

9.7.2 METHOD

The same implementations of the Variance Ridge Detector and Texton Ridge Detector were reused from the experiment in section 9.3. The Texton Ridge Detector was not retrained from that experiment, the same one was used. The scale parameter r was set to 3 for both of the proposed detectors.

This experiment used the MATLAB implementation for Konishi's detector provided by the authors of the Pb detector (Martin, Fowlkes, & Malik, 2004). Konishi's detector was already described in full in section 5.1.

Surround Suppression was implemented in C++/CLI using OpenCV functions. The scale parameter, σ, was set to 2, which is what was used by Martin *et al*. (2004) for the gradient magnitude operator. See section 5.1 for details of the Surround Suppression algorithm.

9.7.3 RESULTS

The Berkeley benchmark was calculated in the way it was described in the previous chapter (see section 8.1). The results over a wide selection of algorithms are as follows:

Detector	F-measure	Comments
Humans	0.79	
gPb (Maire, Arbelaez, Fowlkes, & Malik, 2008)	0.70	Non-real-time
Boosted edge learning (Dollar, Tu, & Belongie, 2006)	0.66	Non-real-time
pB (colour) (Martin, Fowlkes, & Malik, 2004)	0.65	Non-real-time
pB (greyscale) (Martin, Fowlkes, & Malik, 2004)	0.63	Non-real-time
Texton Ridge Detector	**0.63**	**Real-time, proposed**
Variance Ridge Detector	**0.62**	**Real-time, proposed**
Surround Suppression (Grigorescu, Petkov, & Westenberg, 2003)	0.58	Real-time
Konishi's detector (Konishi, Yuille, & Coughlan, 2002)	0.57	Real-time
Gradient magnitude	0.56	Real-time
Random	0.43	

TABLE 9-4: The quality results of various algorithms on Berkeley segmentation dataset. Higher is better.

The graph below in Figure 9-3 shows how the precision-recall curves of the proposed detectors compare to other real-time texture-boundary detectors:

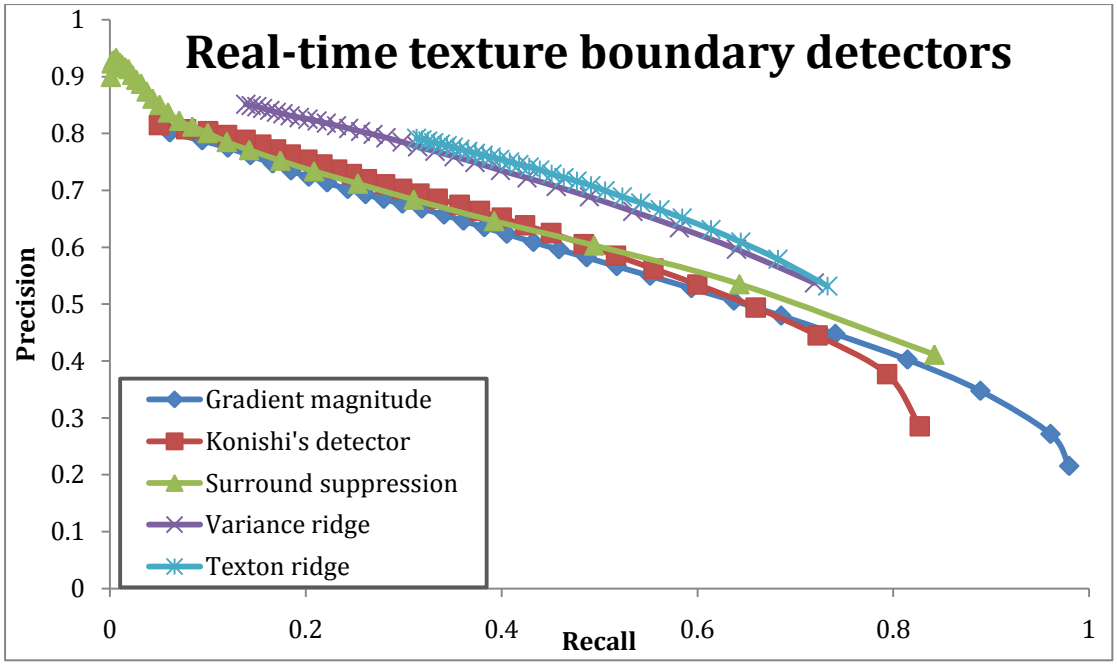


FIGURE 9-3: The precision-recall curves of the various real-time boundary detectors, including the proposed detectors.

A detector's precision-recall curve is better when it is closer to the top-right corner of the graph – indicating it has higher precision and recall.

The graph below shows how the precision-recall curves of the proposed detectors compare to non-real-time detectors:

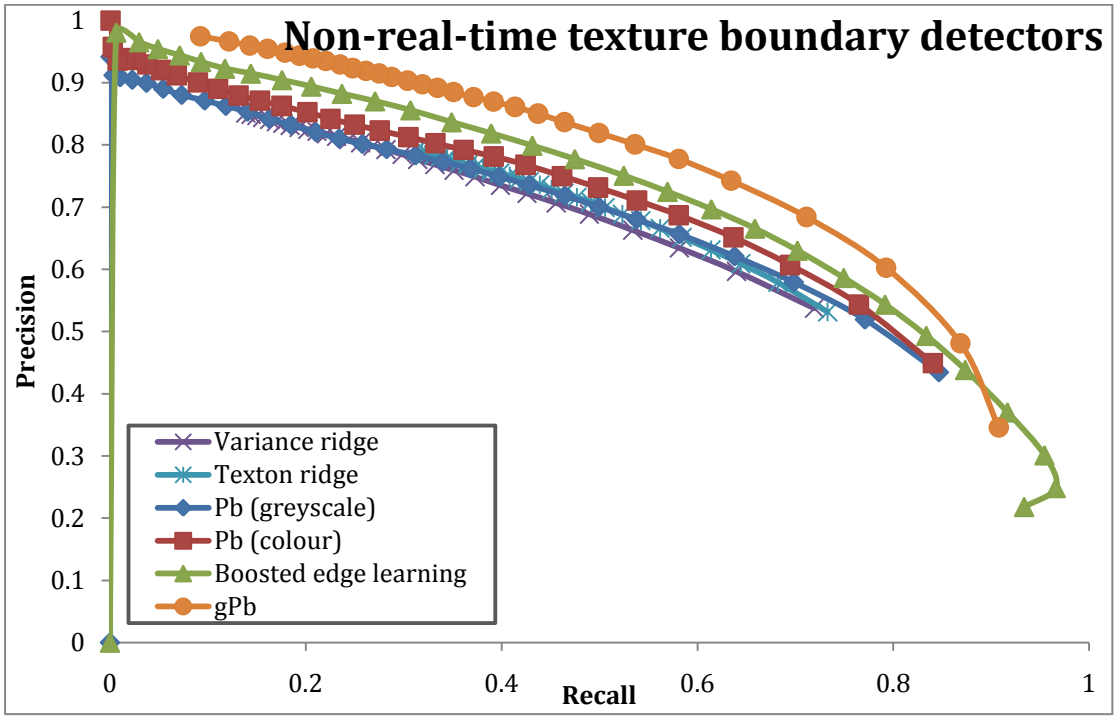


FIGURE 9-4: The precision-recall curves of the proposed detectors versus non-real-time boundary detectors.

Image results

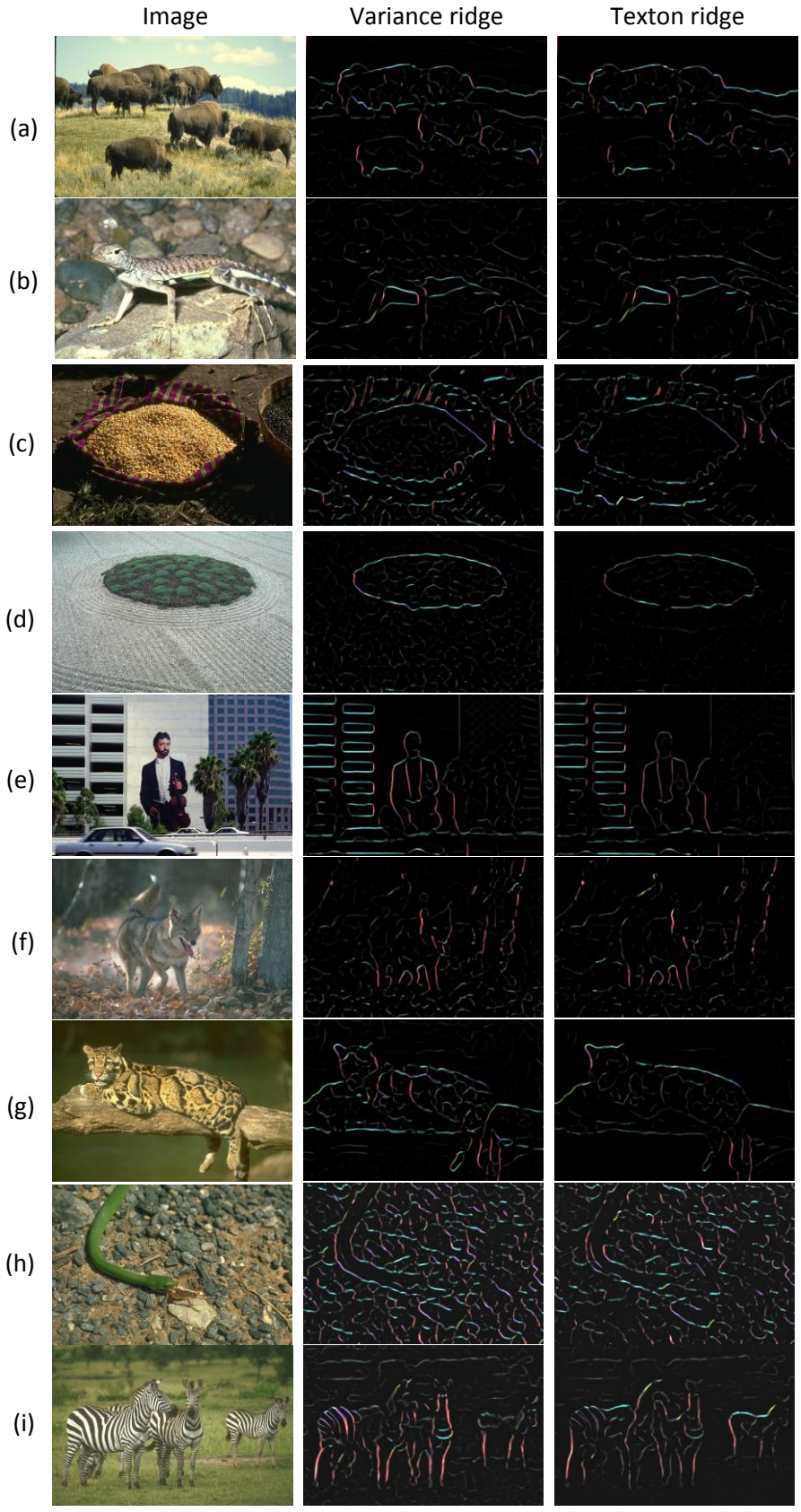


FIGURE 9-5: From left to right: images from the Berkeley dataset, the Variance Ridge Detector, and the Texton Ridge Detector. For the variance ridge and Texton Ridge Detectors, hue represents boundary orientation.

9.7.4 DISCUSSION

Table 9-4 shows that the proposed Variance Ridge Detector scores $F = 0.62$ and the Texton Ridge Detector scores $F = 0.63$. With these scores, the two proposed detectors are the highest-quality out of all the real-time detectors listed.

The precision-recall curves in Figure 9-3 and Figure 9-4 show the usual inverse relationship between precision and recall. This is expected, because as recall increases, more true boundaries are detected as well as more false boundaries, meaning precision will decrease.

Figure 9-3 confirms what the F-measures showed, with the precision-recall curves of the proposed detectors outperforming all the other established real-time texture boundary detectors in the graph. Figure 9-4 also shows that the results generated by the proposed detectors are not far from the best non-real-time detectors, even though the non-real-time detectors are many times slower.

Interestingly, the Texton Ridge Detector scores the same F-measure as the the greyscale version of Pb, and their precision-recall curves are very close, although the Texton Ridge Detector performs slightly worse at high recall. The Texton Ridge Detector is a real-time approximation of the Pb detector, and so this might explain why they can generate similar results.

A few interesting points can be seen from the example images in Figure 9-5.

Firstly, Figure 9-5 shows that the proposed detectors work well on small-scale texture. This can be seen particularly in rows (a), (b) and (c), where the proposed detectors produce little response to the grass, lizard and grain textures. Rows (g), (h) and (i) show larger-scale textures which have not been suppressed as much. This might have been able to be partially alleviated by increasing the scaling parameter r, however, it was found that any increases to r would decrease the overall score of the detectors, even though the score for some of the images increased. Perhaps future research could include some form of automatic scale selection to alleviate this problem.

Secondly, the Texton Ridge Detector is slightly better at suppressing texture than the Variance Ridge Detector. This can be seen in rows (b), (c) and (d), where the less response is generated by the Texton Ridge Detector in textured areas.

This experiment has shown that the proposed boundary detectors are able to produce higher-quality output than other established real-time boundary detectors.

9.8 COMPARISON TO THE REMAINING REAL-TIME DETECTORS

Chapter 5 introduced five real-time texture-boundary detectors. Two of these have already been outperformed in the previous section using the Berkeley benchmark. The remaining three algorithms – Semantic Texton Forests, TextonRML and Randomised Hashing – could not be measured on the Berkeley benchmark. However, it is still clear that the proposed detectors outperform these algorithms, simply because all of these three algorithms produce low-quality boundary maps. Chapter 7 already discussed why these low-quality results are produced by the algorithms. This section will discuss further why the proposed detectors outperform these three algorithms.

Comparison with Semantic Texton Forests

Semantic Texton Forest segmentation (section 5.4) is only able to run in real-time because it does not consider every pixel neighbourhood. Due to this, it produces a boundary map at a resolution 21 times smaller than the image. This severe reduction in resolution means a severe reduction in boundary quality. The proposed detectors produce full-resolution boundary maps, which is why the proposed detectors outperform Semantic Texton Forests. Section 5.4.5 goes into more detail about this.

Comparison with TextonRML

TextonRML (section 5.3) is able to run in real-time because it does not smooth its results with alpha-expansion graph cuts. For the purpose of boundary detection, this is not acceptable because unsmoothed texton approaches normally produce quite noisy class labelings. Section 4.4.3 illustrated how unsmoothed results are inadequate for use as a boundary map. Additionally, section 9.6 showed that adding a smoothing stage using alpha-expansion graph cuts would clearly make it impossible for any algorithm to run in real-time. For these reasons, the proposed detectors produce higher-quality results when compared to TextonRML. However, in real-time, TextonRML is useful for other purposes, such as solving the image labelling problem.

Comparison with Randomised Hashing

Like TextonRML, Randomised Hashing (section 5.5) also does not smooth its results, which is one reason why it produces low-quality boundary maps. However, the biggest problem with Randomised Hashing is its quantisation problem. Sometimes, similar pixels are assigned to entirely separate textons. The sudden change from one texton to another sometimes introduces phantom boundaries into the image. The proposed detectors do not have this problem. For this reason, the proposed detectors outperform Randomised Hashing. Section 5.5.3 goes into more detail about this.

This chapter has compared the two proposed detectors – the Variance Ridge Detector and the Texton Ridge Detector – to seven algorithms, and shown that it outperforms all of them on either quality or speed.

10 CONCLUSIONS

This thesis proposed two new methods for real-time texture-boundary detection, namely, the Variance Ridge Detector and the Texton Ridge Detector. It was found that the two proposed methods outperform the other established texture-boundary detectors on either speed or quality.

10.1 SUMMARY OF RESULTS

The reasons why the proposed detectors outperform established texture-boundary detectors will be summarised in this section.

Most texture-boundary detectors cannot run in real-time

Most texture-boundary detectors cannot run in real-time, simply because they are too computationally intensive. This means they are not useful for real-time computer vision applications.

Section 9.4 investigated the speeds of two established texture-boundary detectors – TextonBoost and GPb – and found that, even if all the non-essential parts of the algorithms are removed, both algorithms still could not run in real-time. This was made clear by the fact that TextonBoost took 4.37 seconds per frame, and GPb took 2.78 seconds per frame. In practice, TextonBoost would also have to run an alpha-expansion graph cut, and GPb would run its normalised cut algorithm, and so there is no chance for texture-boundary detectors such as these to run in real-time.

Section 9.3 measured the speeds of the two proposed detectors and found that both are capable of running in real-time. The Variance Ridge Detector ran at 47.9 frames per second, while the Texton Ridge Detector ran at 10.6 frames per second. This shows that, unlike most texture-boundary detectors, the proposed detectors are capable of real-time execution, which means they can be used for real-time applications.

So given that the proposed detectors run in real-time, the question is, how do they compare to other real-time boundary detectors? As the next two subsections will discuss, all other real-time boundary detectors are either too low level, or they make inadequate approximations of non-real-time counterparts.

Some real-time detectors are too low-level

Low-level computer vision algorithms normally run much faster than high-level algorithms because, generally, they consider less information when making their

decisions. Consequently, some texture-boundary detectors are designed to only interpret the image at a low level so that they can achieve the speed needed for real-time execution. However, low-level often means low-quality boundaries.

Section 9.7 compared the proposed detectors against two real-time low-level texture-boundary detectors using the Berkeley benchmark. The scores of the established texture-boundary detectors were $F = 0.57$ for Konishi's detector, and $F = 0.58$ for Surround Suppression. The proposed detectors clearly outperform these two methods with their scores of $F = 0.62$ for the Variance Ridge Detector, and $F = 0.63$ for the Texton Ridge Detector. This happens because, unlike the two established methods, the proposed detectors generate a higher-level interpretation of images.

Other real-time detectors make inadequate approximations

Many real-time texture boundary detectors were developed by approximating non-real-time counterparts. By definition, every approximation involves some degradation in quality in return for faster speeds. Section 9.8 investigated three established real-time texture-boundary detectors – TextonRML, Semantic Texton Forests and Randomised Hashing – each of which approximate a non-real-time method. It was shown that each approximation had its own set of shortcomings.

TextonRML (section 5.3) is a real-time approximation of TextonBoost (section 4.4). Its most important difference is that it removes the slow alpha-expansion graph cut stage, allowing it to run in real-time. Unfortunately, the graph cut stage is what allows TextonBoost to produce high-quality boundaries, and so without it, TextonRML can only produce low-quality boundaries. This was elaborated further in section 4.4.3. Unlike TextonRML, the proposed detectors produce high-quality boundaries.

Semantic Texton Forest segmentation (section 5.4) also approximates TextonBoost. It removes the need for the slow alpha-expansion graph cut stage by calculating the boundary map at a much lower resolution. Unfortunately, the lower resolution means the boundary map is of much lower quality. Unlike Semantic Texton Forest segmentation, the proposed detectors produce full-resolution boundary maps, and so the proposed detectors outperform Semantic Texton Forests.

Randomised Hashing (section 5.5) is a real-time approximation of mean-shift segmentation (section 4.1). It uses hashing instead of mean-shift clustering so that it can achieve real-time. Unfortunately, the clustering process will sometimes assign similar colours to separate clusters, which introduces boundaries into the image where they should not exist. These phantom boundaries degrade the quality of its boundary maps.

Unlike Randomised Hashing, the proposed detectors do not suffer the problem of phantom boundaries.

The primary goal of this research was to develop a real-time texture-boundary detector that produces high-quality results. All of the above points show this goal has been achieved.

10.2 FUTURE WORK

There are four areas in which future work will proceed from here. First, the texture-suppressing abilities of the detectors can still be improved, as there are still some textures which cannot be suppressed easily by the proposed detectors. Second, the proposed detectors cannot guarantee closed boundary contours at present. This may be useful in some applications and so is another area that could be improved. Third, the boundary maps that are produced currently only involve mid-level information, and so perhaps an image-level interpretation stage could be added to improve the boundary detection quality. Fourth, the detectors themselves could be used to improve real-time applications. Each of these areas will be discussed individually.

Improving texture-suppressing ability

It might be possible to improve the texture-suppressing ability of the proposed detectors by using some form of covariance instead of just variance. Tuzel, Porikli and Meer (2006) used covariance with great success for texture recognition, with their results outperforming even the widely-used texton approach on the Brodatz texture dataset. So perhaps covariance could be used for both fast and high-quality texture boundary detection.

Perhaps it is possible to improve the Texton Ridge Detector by using different machine learning methods to train its approximate textonisation stage. Techniques such as simulated annealing (Kirkpatrick, Gelatt, & Vecchi, 1983) or particle swarm optimisation (Kennedy & Eberhart, 1995) could be appropriate choices for further investigation.

Finding closed boundary contours

Normally, non-real-time boundary detectors find closed boundary contours using clustering or graph cuts. Many of the high-performance methods for this cannot run in real-time, but perhaps approximations can be made. Juan and Boykov (2006) have developed a minimum cut which can run fast when given a near solution to start with, making it capable of running in real-time at least some of the time. Additionally, Randomised Hashing (section 5.5) and the watershed algorithm (Beucher & Lantuéjoul,

1979) are able to perform some approximate forms of real-time clustering. Perhaps these could be starting points for future solutions.

Image-level interpretation

The proposed Texton Ridge Detector has a real-time approximation of every stage of the global probability of boundary detector (section 4.6), except for one – the normalised cut stage. The normalised cut is important because it detects boundaries at the image-level, but unfortunately it does not run in real-time. Future work could involve investigating ways of bringing the normalised cut into real-time.

Alternatively, perhaps higher-level or domain-specific knowledge could be incorporated into the algorithm. Achieving image-level interpretation of any kind though would be quite difficult due to the time constraints on the algorithm.

Applications

Most importantly, future work could also include applying the proposed Variance Ridge Detector and Texton Ridge Detector to other applications. As boundary detection is such a low-level feature, there are many possible fields of application – robot navigation, face recognition, object model reconstruction, object tracking, inpainting, and many more.

To assist with developing applications, it would be useful to do an investigation into the trade-off between speed and quality for the range of real-time texture-boundary detectors. This would allow users to choose the texture-boundary detector that suits their application best.

10.3 THESIS SUMMARY

In summary, objective measurements have shown that both the proposed Variance Ridge Detector and the proposed Texton Ridge Detector outperform all previous work, due to the following two reasons:

1. The proposed boundary detectors run in real-time, unlike most texture-boundary detectors.
2. The proposed boundary detectors produce higher quality-results than the few texture-boundary detectors that do run in real-time.

Boundary detection is an essential first step for many computer vision algorithms, and so potentially, the improvement to boundary detection that was presented by this thesis could induce improvements to a wide-range of applications throughout computer vision.

BIBLIOGRAPHY

Ahmad, M. B., & Choi, T. S. (1999). Local threshold and boolean function based edge detection. *International Conference on Consumer Electronics*, *45*, pp. 332-333.

Arbelaez, P. (2006). Boundary extraction in natural images using ultrametric contour maps. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops.*

Arthur, D., & Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. *The annual ACM-SIAM Symposium on Discrete Algorithms.*

Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern Information Retrieval.* New York: ACM Press, Addison-Wesley.

Barnard, K., Duygulu, P., Guru, R., Gabbur, P., & Forsyth, D. (2003). The effects of segmentation and feature choice in a translation model of object recognition. IEEE Computer Society Conference on Computer Vision and Pattern Recognition.

Beucher, S., & Lantuéjoul, C. (1979). Use of watersheds in contour detection. *International workshop on image processing, real-time edge and motion detection.*

Beucher, S., & Meyer, F. (1993). The morphological approach to segmentation: the watershed transformation. In E. R. Dougherty (Ed.), *Mathematical Morphology in Image Processing* (pp. 433-481).

Bigun, J., & Granlund, G. (1987). Optimal Orientation Detection of Linear Symmetry. *International Conference of Computer Vision.* London.

Boost C++ libraries. (2009). Retrieved from http://www.boost.org/

Boykov, Y., & Jolly, M. P. (2001). Interactive graph cuts for optimal boundary and region segmentation of objects in ND images. *IEEE International Conference on Computer Vision*, *1.*

Boykov, Y., & Kolmogorov, V. (2004). An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 26* (9).

Boykov, Y., Veksler, O., & Zabih, R. (2001). Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 23* (11).

Brown, C. M., & Terzopoulos, D. (1994). *Real-time computer vision.* Cambridge University Press.

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 8* (6), 679-698.

Comaniciu, D., & Meer, P. (2002). Mean Shift: A Robust Approach Toward Feature Space Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 24* (5).

Crezee, D. (2007). *Object Class Recognition and Localization using a Visual Vocabulary Framework.* Masters thesis, Universiteit van Amsterdam.

Csurka, G., & Perronnin, F. (2008). A simple high performance approach to semantic segmentation. *British Machine Vision Conference.*

Dollar, P., Tu, Z., & Belongie, S. (2006). Supervised learning of edges and object boundaries. *IEEE Conference on Computer Vision and Pattern Recognition.*

Drummond, T., & Cipolla, R. (2002). Real-time visual tracking of complex structures. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 24* (7).

Everingham, M., Van Gool, L., Williams, C. K., Winn, J., & Zisserman, A. (2007). The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results.

Everingham, M., Van Gool, L., Williams, C. K., Winn, J., & Zisserman, A. (2008). The PASCAL Visual Object Classes Challenge 2008 (VOC2008) Results.

Felzenszwalb, P., & McAllester, D. (2006). A min-cover approach for finding salient curves. *IEEE Conference on Computer Vision and Pattern Recognition Workshop*, (pp. 185-185).

Ford, L. R., & Fulkerson, D. R. (1956). Maximal flow through a network. *Canadian Journal of Mathematics , 8* (3).

Furukawa, Y., & Ponce, J. (2009). Carved Visual Hulls for Image-Based Modeling. *International Journal of Computer Vision , 81* (1), 53-67.

Gionis, A., Indyk, P., & Motwani, R. (1999). Similarity Search in High Dimensions via Hashing. *Proceedings of the 25th Very Large Database (VLDB) Conference* .

Greig, D. M., Porteous, B. T., & Seheult, A. H. (1989). Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society, Series B (Methodological) , 51* (2).

Grigorescu, C., Petkov, N., & Westenberg, M. A. (2004). Contour and boundary detection improved by surround suppression of texture edges. *Image and Vision Computing , 22* (8), 609-622.

Grigorescu, C., Petkov, N., & Westenberg, M. A. (2003). Contour detection based on nonclassical receptive field inhibition. *IEEE Transactions on Image Processing , 12* (7), 729-739.

Gupta, L., & Das, S. (2006). Texture Edge Detection using Multi-resolution Features and SOM. *18th International Conference on Pattern Recognition (ICPR'06)* (pp. 199-202). IEEE.

Harris, C., & Stephens, M. (1988). A combined corner and edge detector. *Fourth Alvey Vision Conference*, *pp*, pp. 147-151.

He, X., Zemel, R., & Carreira-Perpinan, M. (2004). Multiscale conditional random fields for image labeling. *IEEE Conference on Computer Vision and Pattern Recognition*, (pp. 695-702).

Hidayat, R., & Green, R. (2009). Real-time texture boundary detection from ridges in the standard deviation space. *British Machine Vision Conference.* London.

Hidayat, R., & Green, R. (2008). Texture-suppressing edge detection in real-time. *Image and Vision Computing New Zealand.* Lincoln.

Hsu, R. L., Abdel-Mottaleb, M., & Jain, A. K. (2002). Face detection in color images. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 24* (5).

Indyk, P., & Motwani, R. (1998). Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. *Proceedings of 30th Symposium on Theory of Computing* .

Jacobson, S. H., & Yücesan, E. (2004). Analyzing the Performance of Generalized Hill Climbing Algorithms. *Journal of Heuristics , 10* (4).

Jamzad, R. R. (2005). Real time classification and tracking of multiple vehicles in highways. *Pattern Recognition Letters , 26* (10).

Johnson, M. A. (2008). *Semantic Segmentation and Image Search.* University of Cambridge.

Juan, O., & Boykov, Y. (2006). Active graph cuts. *IEEE Conference on Computer Vision and Pattern Recognition.*

Kennedy, J., & Eberhart, R. (1995). Particle Swarm Optimization. *IEEE International Conference on Neural Networks.*

Kiranyaz, S., Ferreira, M., & Gabbouj, M. (2008). A generic shape/texture descriptor over multiscale edge field: 2-D walking ant histogram. *IEEE Transactions on Image Processing , 17* (3), 377-91.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science , New Series 220* (4598), 671-680.

Kisačanin, B., Pavlović, V., & Huang, T. S. (2005). *Real-time vision for human-computer interaction.* New York: Springer Science + Business Media, Inc.

Klappstein, J., Vaudrey, T., Rabe, C., Wedel, A., & Klette, R. (2009). Moving object segmentation using optical flow and depth information. *Lecture Notes in Computer Science , 5414*, 611-623.

Knutsson, H. (1989). Representing local structure using tensors. *6th Scandinavian Conf. on Image Analysis.* Oulu.

Konishi, S., Yuille, A. L., & Coughlan, J. (2002). Fundamental bounds on edge detection: an information theoretic evaluation of different edge cues. *IEEE Conference on Computer Vision and Pattern Recognition.*

Koschan, A., & Abidi, M. (2005). Detection and classification of edges in color images. *IEEE Signal Processing Magazine , 22* (1).

Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval research logistics quarterly , 2*, pp. 83-97.

Kuhn, H. W. (1956). Variants of the Hungarian method for assignment problems. *Naval Research Logistics Quarterly* .

Kuwahara, M., Hachimura, K., Ehiu, S., & Kinoshita, M. (1976). Processing of RI-angiocardiographic images. *Processing of Biomedical Images* , 187-203.

Leung, T., & Malik, J. (1998). Contour continuity in region-based image segmentation. *European Conference on Computer Vision*, (pp. 544-559).

Lloyd, S. P. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory , 28* (2), 129-137.

Maire, M., Arbelaez, P., Fowlkes, C., & Malik, J. (2008). Using contours to detect and localize junctions in natural images. *IEEE Conference on Computer Vision and Pattern Recognition.*

Majman, L., Couprie, M., & Bertrand, G. (2005). Watersheds, mosaics, and the emergence paradigm. *Discrete Applied Mathematics , 147* (2-3).

Malik, J., Belongie, S., Shi, J., & Leung, T. (1999). Textons, Contours and Regions: Cue Integration in Image Segmentation. *IEEE International Conference on Computer Vision.*

Martin, D. R., Fowlkes, C. C., & Malik, J. (2004). Learning to Detect Natural Image Boundaries Using Local Brightness, Color, and Texture Cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 26* (5), 530-549.

Martin, D., Fowlkes, C., Tal, D., & Malik, J. (2001). A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. *IEEE International Conference on Computer Vision.*

Maxwell, B. A., & Brubaker, S. J. (2003). Texture edge detection using the compass operator. *British Machine Vision Conference.*

Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics* .

Nitzberg, M., & Shiota, T. (1992). Nonlinear image filtering with edge and corner enhancement. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 14* (8), 826-833.

OpenCV. (2008). *OpenCV Documentation*. Retrieved from http://opencv.willowgarage.com/documentation/

Papari, G., Petkov, N., & Campisi, P. (2007). Artistic Edge and Corner Enhancing Smoothing. *IEEE Transactions on Image Processing , 16* (10), 2449-2462.

Paplinski, A. P., & Boyce, J. F. (1997). Segmentation of a class of ophthalmological images using a directional variance operator and co-occurrence arrays. *Optical Engineering , 36* (11), 3140-3147.

Perona, P., & Malik, J. (1990). Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 12* (7).

Poynton, C. (2006). *Frequently asked questions about color.* Retrieved 2009, from http://www.poynton.com/PDFs/ColorFAQ.pdf

Puzicha, J., Rubner, Y., Tomasi, C., & Buhmann, J. (1999). Empirical evaluation of dissimilarity measures for color and texture. *IEEE International Conference on Computer Vision*, (p. 1165).

Randen, T., & Husøy, J. H. (1999). Filtering for Texture Classification: A Comparative Study. *IEEE Transaction on Pattern Analysis and Machine Intelligence , 21* (4).

Ranganathan, A. (2009). Semantic Scene Segmentation using Random Multinomial Logit. *British Machine Vision Conference.*

Ren, X. (2008). Multi-scale improves boundary detection in natural images. *European Conference on Computer Vision.*

Russell, S. J., & Norvig, P. (2009). *Computer Vision and Information Technology: Advances and Applications.* Upper Saddle River, New Jersey: Prentice Hall.

Ruzon, M. A., & Tomasi, C. (2001). Edge, Junction, and Corner Detection Using Color Distributions. *IEEE Transaction on Pattern Analysis and Machine Intelligence , 23* (11).

Schroff, F., Criminisi, A., & Zisserman, A. (2008). Object class segmentation using random forests. *British Machine Vision Conference.*

Shao, H., Ji, J., Kang, Y., & Zhao, H. (2009). Application Research of Homogeneous Texture Descriptor in Content-Based Image Retrieval. *International Conference on Information Engineering and Computer Science.* IEEE.

Sharon, E., & Brandt, A. (2000). Segmentation and Boundary Detection Using Multiscale Intensity Measurements. *IEEE Conference on Computer Vision and Pattern Recognition.*

Sharon, E., Brandt, A., & Basri, R. (2000). Fast multiscale image segmentation. *IEEE Conference on Computer Vision and Pattern Recognition.*

Sharon, E., Galun, M., Sharon, D., Basri, R., & Brandt, A. (2006). Hierarchy and adaptivity in segmenting visual scenes. *Nature , 442* (7104), pp. 810-3.

Shi, J., & Malik, J. (2000). Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 22* (8).

Shotton, J., Blake, A., & Cipolla, R. (2008). Multiscale categorical object recognition using contour fragments. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 30* (7), 1270-81.

Shotton, J., Johnson, M., & Cipolla, A. (2008). Semantic texton forests for image categorization and segmentation. *IEEE Conference on Computer Vision and Pattern Recognition.*

Shotton, J., Winn, J., Rother, C., & Criminisi, A. (2009). TextonBoost for Image Understanding: Multi-Class Object Recognition and Segmentation by Jointly Modeling Texture, Layout, and Context. *International Journal of Computer Vision , 81* (1).

Shotton, J., Winn, J., Rother, C., & Criminisi, A. (2006). Textonboost: Joint appearance, shape and context modeling for multi-class object recognition. *European Conference on Computer Vision.*

Smith, S. M., & Brady, J. M. (1997). SUSAN - A New Approach to Low Level Image Processing. *International Journal of Computer Vision , 23* (1).

Sobel, I., & Feldman, G. (1973). A 3x3 isotropic gradient operator for image processing. In R. Duda, & P. Hart, *Pattern Classification and Scene Analysis* (pp. 271-272). John Wiley and Sons.

Taylor, C. J., & Cowley, A. (2009). Fast Segmentation via Randomized Hashing. *British Machine Vision Conference.*

Tomasi, C., & Manduchi, R. (1998). Bilateral filtering for gray and color images. *IEEE Conference on Computer Vision and Pattern Recognition.*

Tuceryan, M. (1994). Moment-based texture segmentation. *Pattern Recognition Letters , 15* (7).

Tuzel, O., Porikli, F., & Meer, P. (2006). Region covariance: A fast descriptor for detection and classification. *European Conference on Computer Vision. 3952.* Springer.

van Rijsbergen, C. (1979). *Information Retrieval.* London: Butterworth.

Varma, M., & Zisserman, A. (2003). Texture classification: are filter banks necessary? *IEEE Conference on Computer Vision and Pattern Recognition.*

Vaudrey, T., Wedel, A., Rabe, C., Klappstein, J., & Klette, R. (2008). Evaluation of moving object segmentation comparing 6D-vision and monocular motion constraints. *International Conference Image and Vision Computing New Zealand.* IEEE.

Verbeek, J., & Triggs, B. (2007). Region Classification with Markov Field Aspect Models. *IEEE Conference on Computer Vision and Pattern Recognition.*

Vincent, L., & Soille, P. (2002). Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 13* (6).

Weickert, J. (1999). Coherence-enhancing diffusion filtering. *International Journal of Computer Vision* .

Weickert, J. (1999). Coherence-enhancing diffusion of colour images. *Image and Vision Computing.*

Whyte, O., Sivic, J., & Zisserman, A. (2009). Get Out of my Picture! Internet-based Inpainting. *British Machine Vision Conference.*

Winn, J., & Jojic, N. (2005). Locus: Learning object classes with unsupervised segmentation. *IEEE International Conference on Computer Vision*, *1*, pp. 756-763.

Winn, J., Criminisi, A., & Minka, T. (2005). Object Categorization by Learned Universal Visual Dictionary. *IEEE Conference on Computer Vision and Pattern Recognition.*

Wu, Z., & Leahy, R. (1993). An optimal graph theoretic approach to data clustering: theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 15* (11), 1101-1113.

Ye, Q., Gao, W., & Wang, W. (2003). A new texture-insensitive edge detection method. *International Conference on Information, Communications and Signal Processing. 2*, pp. 768-772. IEEE.