Investigation of the FRAME Algorithm for Statistical Modelling of Visual Textures

MSc Thesis

Baruch Lubinsky
University of Cape Town
Department of Electrical Engineering

Supervised by Dr Fred Nicolls

September 28, 2012

DECLARATION

“Too much light often blinds gentlemen of this sort. They cannot see the forest for the trees.”

Christoph Martin Wieland

ABSTRACT

This thesis is a study of the Filters, Random fields and Maximum Entropy (FRAME) algorithm. The algorithm develops statistical models of visual textures. Visual textures are difficult to model in general, as they may be images of any patterned surface. The models produced by FRAME are for homogeneous, flat textures. They comprise a number of filters and constraints on those filters' responses. The filters are selected for maximum entropy and can be analysed to give insight into the structure of the model.

Within the field of computer vision, two main tasks relating to visual texture are considered: classification and synthesis. A number of existing solutions to each of these are presented. The FRAME algorithm is shown to be useful for both applications. Examples of the synthesis produced by the algorithm are shown and a comparative classification experiment is done which proves that a FRAME classifier can out-perform competing algorithms.

A number of modifications to the original algorithm are implemented and presented. The algorithm relies on a Gibbs sampler. This is modified to run in parallel on a GPU, greatly improving the performance of the algorithm as the complexity is increased. Other modifications attempt to improve the quality of the models or decrease the number of iterations required for convergence. A significant improvement is made by smoothing the histograms in the algorithm with a Gaussian kernel.

ACKNOWLEDGEMENTS

This research is funded by the NRF, through the Scarce Skills program. Additional funding is provided by the Anglo Platinum Machine Vision fund, via the Department of Chemical Engineering.

Thanks to my supervisor Dr Fred Nicolls for his guidance and mentorship.

Further thanks to all my colleagues in the Computer Vision group without whom this research would not have been possible.

CONTENTS

1. INTRODUCTION

Computer vision is a broad term applied to many areas of research, spanning fields from computer science and statistics to psychology. Vision is the most important human sensory input, providing us with the majority of the information used to navigate our world. The human vision system is extremely advanced, providing such faculties as instant, sub-conscience facial recognition. Computer vision research is in general focused on trying to emulate the tasks performed by the vision systems of the brain using a computer.

There are various approaches to this challenge taken by different researchers. Some seek to understand the inner workings of the human vision system with a view to emulating those systems. Such work has led to a number of interesting discoveries about the pre-attentive circuits in the visual cortex. These neural pathways perform many transformations on the incoming visual data before it is processed by higher level parts of the brain. For example, there appear to be neural circuits that perform transformations similar to those of Gaussian filters [1]. Consequently many researchers attempt to create computer systems analogous to the visual cortex to study how much information they can extract using similar signal processing. This type of work leads to great advancements in both the power of computer vision systems and our understanding of the human brain.

From an engineering point of view, we are more interested in the practical applications of the algorithms developed for computer vision. There is a vast amount of work on the application of machine learning and pattern recognition techniques to visual data. Algorithms that are developed for general machine learning tasks are applied to image data. Unlike other fields, the benchmark or ground truth in computer vision is based on human perception. Certainly most of the computer vision tasks we imagine are based on the tasks we perform routinely with our own vision system.

The work in this thesis is a practical analysis of the Filters, Random fields and Maximum Entropy (FRAME) algorithm [2]. The algorithm is designed to build statistical models of visual textures. "Visual texture" is a somewhat vague term with numerous interpretations. For the purpose of this work, a visual texture is an image of a homogeneous pattern; a realisation of a spatially stationary stochastic process. The human mind is predisposed to detecting patterns to the point where it will imagine a pattern even where there is none[1].

Consider Figure 1.1: if one looks at the image for a few seconds some shapes will begin to appear yet these are illusory in an entirely random image.

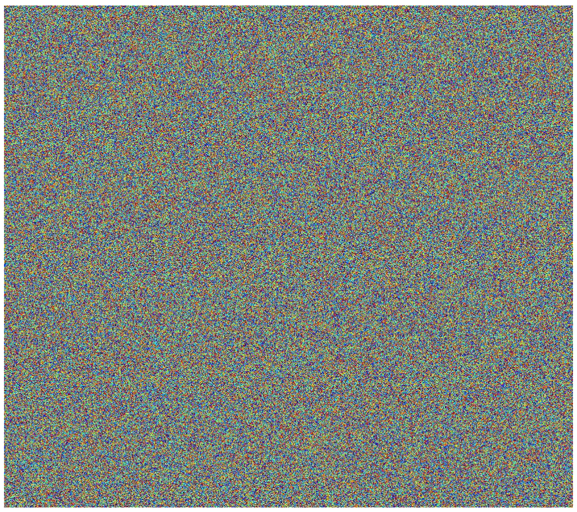


Fig. 1.1: Random image with the colour of each pixel being drawn from a uniform distribution.

As a result, we may consider an image to be a visual texture, and to have some definite pattern even when none exists. This presents a challenge to any mathematical model of texture. The FRAME model assumes some stochastic process underlying the texture image. In natural images this may be the actual

[1] This tendency is used by some scientists to account for the prevalence of belief in the supernatural.

process that produced the subject. The algorithm models the texture in the image by applying a sequence of filters and calculating a probability density function based on the response. The goal is to come up with a probability density function whose realisation is visually similar to the observed texture.

This thesis in an investigation into the FRAME algorithm; how it can be optimised and applied. Section 2.1 places the algorithm in context by describing a number of popular techniques for modelling visual textures for the purpose of either synthesis or classification. The algorithm itself is explained in Section 2.2, including some background theory. Chapter 3 gives details specific to this implementation. These include descriptions of the parameters in the algorithm whose values are studied in Chapter 4. The Experiments chapter, Chapter 4, gives insight into how the algorithm can be tuned, and how the various parameters affect its performance. The findings and conclusions of this thesis are presented in Chapter 5.

2. BACKGROUND

In the field of computer vision, visual texture problems can be broadly divided into two general classes: classification and synthesis [3]. Classification refers to the problem of identifying a texture; synthesis algorithms attempt to artificially generate texture images. The work in this dissertation builds a statistical model which is capable of solving both problems. The algorithm used is called FRAME (**F**ilters, **R**andom fields, and **M**aximum **E**ntropy) [2]. A number of other existing techniques are presented in this chapter.

2.1 Visual texture

Any image or image region containing a visual pattern may be referred to as a visual texture. Visual textures are common in real world imagery and can be seen in scenes such as grass, brick walls and smoke. The patterns may be highly regular or exhibit large stochastic variations [4]. This makes it difficult to provide a mathematical definition for visual textures.

Much of the computer vision research into visual textures stems from the work of Julesz [5]. This work studies the basis for texture discrimination in the human visual system. A number of stochastically generated images with two distinct texture regions are presented to human subjects. If the two regions can be identified immediately, then the textures are considered to be discriminated. Julesz showed that for many classes of visual textures, a difference in second order statistics is sufficient to allow for discrimination [6]. This is the first example of a statistical model for visual texture [2].

2.1.1 Classification

Most machine learning problems can be phrased in terms of a pattern classification problem. In general, a system is trained on a number of sample patterns each belonging to a specific class such that other instances of those classes can then be identified. For a visual texture, the classification problem is to identify what texture an image is of, given a set of training textures. This is particularly useful for scene recognition, where portions of the image are made up of visual textures, which may give context to the rest of the image [7, 8, 9].

Bag of features

A bag of features classifier works by dividing the texture into regions and calculating values for a number of descriptors (the bag of features or codebook) in each region. The distribution of the results from each region are aggregated to give a representation of the texture in the form of a vector in feature space. Some aggregation method, for example K-means clustering, is used to return the feature vector considered to be the best representative of the training images. The choice of how to divide texture images into regions, which descriptors to use, and how to aggregate the results into a single vector may all be varied in different implementations [10].

Textures are classified by calculating the feature vector for the unknown image and comparing it to those of the training images. A class is assigned to the image based on which training class it matches most closely. In the work of Nowak et al. [10] a support vector machine (SVM) is used to make this decision but any similar technique may be applied.

The features used may be any function that outputs a value for a given image region. Two type of features that are commonly used for texture classification are linear convolution filters [11] and SIFT descriptors [12, 13]. The robustness of the classifier in terms of scale, rotation and intensity variation depends on the features in the codebook. The feature vector may contain many thousands of elements, all of which form part of the class model. In general, the feature vector is much too large to offer any meaningful interpretation regarding the texture's structure. While the bag of features approach is very effective at classifying visual textures, this work pursues a more intuitive texture model.

Local binary patterns

Images captured in the real world may vary greatly in their appearance despite being of the same visual texture. We desire that a classifier is insensitive to variations that are caused by having a different viewpoint or illumination of the scene. The variations that can appear within images of the same texture can be classified as: a linear transformation, shearing, rotation, a change in scale and/or a change in illumination. Local binary patterns is a type of feature that are used to create an illumination invariant texture classifier and in special cases one that is also rotation invariant [14].

Local binary patterns (LBP) assigns a value at each pixel based on the intensity of its neighbours. Each pixel in the neighbourhood is assigned a zero or one based on whether its intensity is greater or less than the centre pixel as shown in Figure 2.1. Those values are read as a binary string that is interpreted as a decimal value. That value is then used as a feature for classification. Histograms are calculated of the values either for the whole image or for regions leading to a feature vector describing the texture [7].

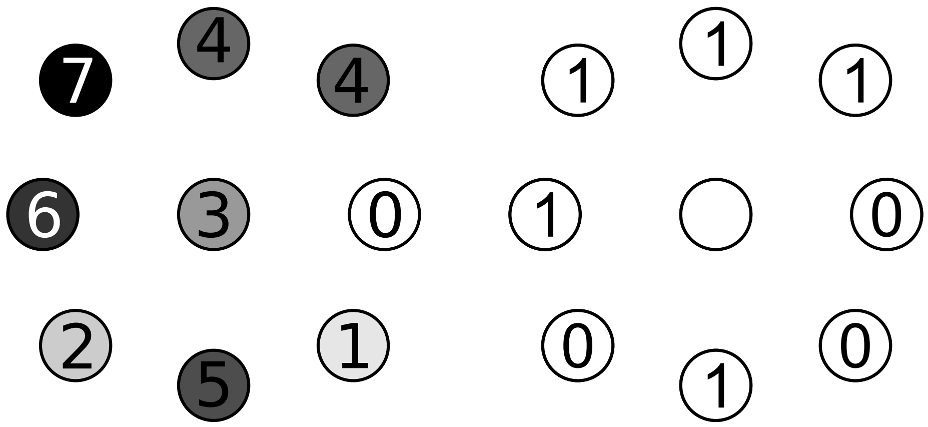


Fig. 2.1: Local binary pattern.

The computed feature vector is the same regardless of the illumination of the image as it is calculated from the relative intensities of neighbouring pixels.

Rotation invariance is achieved by assigning all rotations of a specific pattern the same value [14] for example 00111100 is the same as 00001111. Local binary patterns have proven to be very effective for texture classification and have been used in other applications such as facial recognition [15]. The classifier is rotation invariant when the patterns used are unique under rotation. Any pixels that exhibit a rotated feature are given the same label [14].

2.1.2 Synthesis

Texture synthesis is widely used in computer graphics. Any three dimensional object with a surface that is not simply one plain colour is painted with a texture. In some cases the texture is simply tiled over the object, however for more complex textures this is not satisfactory and a more complex method capturing the stochastic nature of the texture is required [16]. Most texture synthesis methods work either by patch based sampling [17, 18] or by pyramid based analysis [6, 16, 19]. The work of Han et al. [20] following Lefebvre and Hoppe [21] achieves superior results by combining those approaches.

Patch based

The patch based approach to texture synthesis allows for the creation of new and possibly larger textures based on an input example [17]. The texture is synthesised to look different from and yet similar to the input. The synthesised texture is built out of patches from the input. Small regions are selected at random from the input and pasted in the synthesis. The border regions in the synthesis are altered to be different to those in the input. This is done by searching the input for other patches with similar statistics at the edges. The boundary of the patch is then replaced with the boundary from another similar patch in the input. In this way a texture of any size may be synthesised. This method has been shown to produce very convincing and realistic syntheses for natural and artificial textures. It runs faster but unlike FRAME, no model of the texture is extracted.

Pyramid based

Pyramid based methods of texture synthesis attempt to capture features of the texture at different scales. The texture is first analysed by building an analysis pyramid. The input texture is downsampled into a sequence of lower resolutions. In this way, a pyramid is built of different sized versions of the input. In addition to these low-pass downsampled images, the bandpass information is also stored. Bandpass images are calculated by upsampling the low-pass image back to the resolution of the level above and subtracting that image from the original image at that resolution.

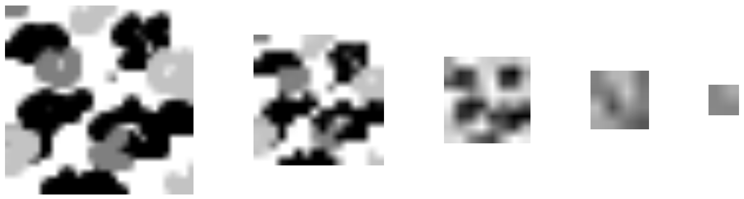


Fig. 2.2: Pyramid of downsampled textures (not to scale).

For each resolution, a number of filters are applied and histograms are calculated from the responses. These histograms provide local texture features in each subband. Different implementations use different filters but the general approach is that the filters must each capture some information about the structure in a local neighbourhood. Heeger and Bergen [16] use filters that can detect structures in different directions; De Bonet [19] uses Laplacian of Gaussian filters; and Portilla and Simoncelli [6] use pairs of wavelets.

The texture synthesis relies on the assumption that at some resolution high frequency changes to the pixel values do not affect the texture statistics at lower resolutions in the pyramid. The tolerance to this variation is how the randomness of the texture is expressed. Synthesis starts from the top of the pyramid, the lowest resolution, which is taken directly from the analysis pyramid [19]. Each lower level of the pyramid is then sampled from the analysis pyramid. Locations in the synthesis are given values based on the statistics of the features in all the parent locations — those locations in higher levels of the pyramid whose values are impacted by the pixels in the current region.

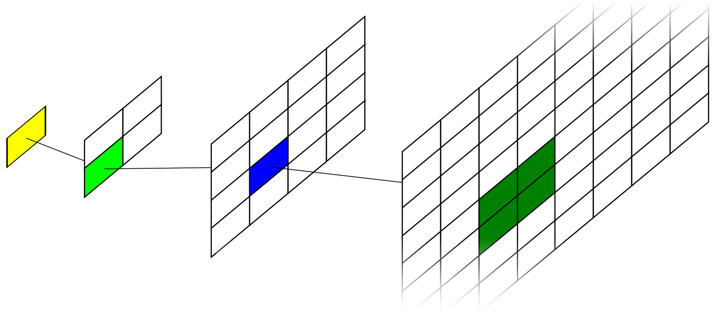


Fig. 2.3: Parent locations in synthesis pyramid [19].

The parent locations of a region are shown in Figure 2.3. A feature vector is formed from each feature at each parent location. A region is sampled from the corresponding scale of the analysis pyramid where the parent vector in the analysis pyramid has the same values (to some threshold) as that region in the synthesis pyramid. Pixel values are copied from the analysis pyramid into the synthesis pyramid. In cases where more than one region in the analysis pyramid is acceptable, a choice is made at random, thereby introducing variation in the synthesis. If the assumption stated above holds, this variation does not change the appearance of the texture. The process continues until a complete synthesis pyramid has been sampled which is then collapsed to give the synthesised texture.

Combined

Han et al. achieve some particularly interesting synthesis results by combining the two approaches mentioned above [20]. Their approach builds on the work of Lefebvre and Hoppe [21]. Both methods create a synthesis using a sequence of images built from different scales of the observation. The synthesis at each scale is performed using patch matching rather than histogram matching.

Lefebvre and Hoppe use a “Gaussian stack” rather than a pyramid for synthesis. The elements of the stack are all images of the same size, and are the filter responses of successively larger Gaussian kernels. This stack can perform the same function as the pyramid as each level contains sub-band information from the level above. The synthesis algorithm is then very similar to the traditional pyramid-based approach. The motivation for using a stack as opposed to a pyramid is that it avoids upscaling; each level has the same dimensions, allowing the synthesis to be calculated in parallel. As a result, their method produces accurate texture synthesis in a short amount of time [21].

The work of Han et al. extends the concept of texture synthesis. Their approach can be used to synthesise a texture at multiple resolutions, so that the synthesised image can be zoomed without any loss of resolution. The algorithm requires observations at each scale. Those observations are arranged in a graph indicating the relationships between the exemplars at different scales. These images at each scale are used to augment the Gaussian stack. The resulting synthesis is accurate at each level in the stack. This leads to a very compact representation of a texture at many scales. The algorithm is able to synthesise images with features at many different scales which is impossible with the other synthesis methods described here [20]. The example given in the paper shows a synthesised image of an island with realistic looking shape, terrain and coasts.

2.2 FRAME

The work in this thesis is based on the algorithm of Zhu, Wu and Mumford – Filters, Random fields and Maximum Entropy [2]. A statistical model of the visual texture is produced. This model can be used to synthesise new instances of the texture and to classify other images by considering how well they fit the model. This approach produces concise and expressive texture models and can be used on many different types of visual textures. This section contains some background theory followed by an explanation of the algorithm itself.

2.2.1 Filters

Convolution is a mathematical function that computes the amount of overlap between two functions [22]. In image processing it is used to apply a filter to an image. The filter response of an image I convolved with a kernel K is $F = I * K$:

$$F(x, y) = \sum_{k_1} \sum_{k_2} K(k_1, k_2) I(x - k_1, y - k_2), \tag{2.1}$$

for each x and y, components of the pixels in the image and k_1, k_2 elements of the sites in the kernel. The calculation can be envisioned as moving the kernel over the image and calculating the product of the kernel and the image region beneath it, at each point. Regions in the image which are similar to the filter will have large values in the response. This is what is meant by using filters to detect features in an image.

Convolution is a linear function, meaning that the convolution of a sum is equal to the sum of convolutions:

$$K * (I_1 + I_2) = K * I_1 + K * I_2. \tag{2.2}$$

The work in this dissertation takes advantage of this important property of convolution filters.

Convolution is defined as an infinite integral; in the discrete case the integral becomes a summation. Equation 2.1 applies specifically to the convolution over a finite 2-dimensional vector. For any kernel which is larger than one pixel, the kernel will extend beyond the edge of the image for x and y values near the border. This can be accounted for in one of three ways: zero padding the border of the image with half as many pixels as the width of the filters, calculating the convolution of the central region of the image or treating the image as a toroid so that coordinates beyond the border are wrapped to the opposite side. In this work all the convolution is done on a toroid. This is based on the assumption that the images are of a homogeneous texture. As a result, all the synthesised textures can be tiled seamlessly, without any artefacts at the borders.

2.2.2 Markov random fields

Visual textures may be modelled as a Markov random field (MRF) [23]. This is an appropriate model for textures that are stochastic and stationary. The model defines a set of pixels $N(x, y)$ as the neighbours of (x, y). The stationarity of the model means that set of neighbours is the same throughout the image so $(a, b) \in N(r, c) \Leftrightarrow (a+i, b+j) \in N(r+i, c+j)$. The model assumes a conditional independence, that the probability of a pixel given all the remaining pixels is equal to the probability of a pixel given its neighbours:

$$P(\mathbf{I}(x,y)|\mathbf{I}(i,j) : (i,j) \neq (x,y)) = P(\mathbf{I}(x,y)|N(x,y)). \tag{2.3}$$

MRF models are used extensively in statistical mechanics and a number of useful theoretical results are developed.

2.2.3 Bayesian inference and Markov chain Monte Carlo

Bayesian statistics differs from the more traditional frequentist approach in the way that the model is treated. In a frequentist paradigm the model is considered to be fixed yet unknown; to Bayesians, the model is a random variable. Results from frequentist statistics are obtained by sampling the population and making confidence statements about how likely the sample statistics are to match the population statistics. These results are based on all possible outcomes of the procedure [24].

A Bayesian model is made up of a parametric model $P(x|\theta)$, with an observation x and parameters θ of the process and a prior distribution $\pi(\theta)$ which encapsulates our assumptions about the process. Both x and θ are considered to be outcomes of a random process. Given a data observation x the probability of the parameters can be updated with the posterior distribution [25]:

$$\pi(\theta|x) = \frac{P(x|\theta)\pi(\theta)}{\int P(x|\theta)\pi(\theta)d\theta}. \tag{2.4}$$

In this statistical model, both the parameters and the outcomes are random variables. Consequently probability statements can be made about the parameters of the model and the values of those parameters can be estimated from a finite set of observations as opposed to the frequentist case which requires all possible outcomes to be considered [24]. This is crucial in image processing where outcomes are images so the random variable has as many dimensions as there are pixels. In such a high-dimensional space it is infeasible to attempt to enumerate every possible outcome.

Historically, Bayesian inference was less popular than a frequentist approach because it was infeasible to calculate the integral $\int P(x|\theta)\pi(\theta)d\theta$. Developments in Monte Carlo sampling have made Bayesian inference a far more practical tool by allowing the integral to be estimated [26]. Markov chain Monte Carlo (MCMC) in particular is a computationally efficient method for estimating complex integrals [27].

MCMC encompasses a large class of algorithms, the most popular of which is the Metropolis algorithm [27]. The concept behind Monte Carlo algorithms is to approximate an integral over a density $p(x)$ by drawing a number of independent and identically distributed samples from the distribution. The integral may be any function of the distribution, such as the expectation or the variance. Although the integral itself is difficult to calculate, it can be approximated by averaging a large number of samples from the distribution. Those samples $x^{(i)}$ are summed to give an approximation of the integral:

$$\frac{1}{N}\sum_{i=1}^{N} f(x^{(i)}) \xrightarrow[N\to\infty]{a.s.} \int_{\mathcal{X}} f(x)p(x)dx \tag{2.5}$$

where $\mathcal{X}$ is the multidimensional space in which $p(x)$ is defined. Each sample value depends only on the previously drawn values so the samples form a Markov chain. At each iteration of the algorithm, a new sample is drawn and added to the chain with a certain probability – the acceptance rate. After a large number of iterations, $N \to \infty$, the calculated value is an acceptable approximation of the true distribution. The number of samples required for this convergence depends on the distribution and the details of the implementation of the algorithm [26].

2.2.4 Gibbs sampler

The Gibbs sampler is a special case of the popular Metropolis-Hastings [28] algorithm. In a Gibbs sampler elements of the distribution are sampled one at a time. All samples are included in the chain – the acceptance rate is equal to 1.

Consider a two dimensional Gaussian distribution:

$$(X_1, X_2) \sim N\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & \sigma \\ \sigma & 1 \end{pmatrix}\right). \tag{2.6}$$

The Gibbs sampler in two dimensions draws samples from alternating conditional distributions. Given a starting point $\left(x_1^{(0)}, x_2^{(0)}\right)$ the first iteration is,

$$x_1^{(1)} = x_1^{(0)} \tag{2.7}$$

$$x_2^{(1)} \sim \left(X_2 | x_1^{(0)}\right) \tag{2.8}$$

followed by,

$$x_1^{(2)} \sim \left(X_1 | x_2^{(1)}\right) \tag{2.9}$$

$$x_2^{(2)} = x_2^{(1)}. \tag{2.10}$$

The algorithm proceeds in this manner producing the Markov chain,

$$\left(x_1^{(0)}, x_2^{(0)}\right), \left(x_1^{(1)}, x_2^{(1)}\right), \ldots, \left(x_1^{(N)}, x_2^{(N)}\right), \tag{2.11}$$

until the samples in the chain are a good representation of the distribution. At that point, for example, the mean of the values in the chain is a good approximation of the expectation of the true distribution.

For images each pixel is a dimension of the distribution. In each iteration the colour of a single pixel is changed. The colour is changed based on the full conditional distribution, but the effect of changing only one pixel value at a time is localised — the Markov property of the chain of samples is preserved. This can be done far more efficiently than attempting to draw a sample from the full image space. After a large number of iterations, the samples become representative of the target distribution [26].

The algorithm creates a sequence of possible samples that form a Markov chain. It is initialised with a white noise image. At each step, the colour of a single pixel is updated according to the probability under the current proposal distribution $p(\mathbf{I})$. The probability of that pixel having each possible intensity is calculated from the distribution and a new intenisty value is selected with likelihood in proportion to those probabilities. After enough steps in the chain the samples are being drawn from the target distribution. However, knowing the exact number of steps required before the Markov chain has converged is not straightforward [29].

2.2.5 Maximum entropy

The principle of maximum entropy (ME) is a method for constructing a probability distribution p for a set of random variables X, given some available information about the random variables [30]. For example, the expectation of some known function $\phi_n(x)$ is $E_p[\phi_n(x)] = \int \phi_n(x)p(x)dx = \mu_n$, let Ω be the set of all probability distributions which produce that statistic. Then Ω is:

$$\Omega = \{p(x) : E_p[\phi_n(x)] = \mu_n\}. \tag{2.12}$$

The maximum entropy principle is to choose the probability distribution, $p(x)$, that maximises the entropy. That is:

$$p^*(x) = \arg\max \left\{ - \int p(x) \log p(x) dx \right\}, \tag{2.13}$$

subject to

$$E_p[\phi_n(x)] = \int \phi_n(x)p(x)dx = \mu_n \tag{2.14}$$

and

$$\int p(x)dx = 1. \tag{2.15}$$

By Lagrange multipliers, the solution is:

$$p(x, \Lambda) = \frac{1}{Z(\Lambda)} e^{-\sum_{n=1}^{N} \lambda_n \phi_n(x)}, \tag{2.16}$$

where Λ is the Lagrange parameter comprising the Lagrange multipliers $(\lambda_1, ..., \lambda_N)$ and $Z(\Lambda)$ is the partition function. In general it is impossible to find an exact, closed form, solution for Λ so a numerical approach is taken. The parameters are calculated by:

$$\frac{d\lambda_n}{dt} = E_{p(I;\Lambda)}[\phi_n(x)] - \mu_n. \tag{2.17}$$

It can be shown that a unique solution exists for $(\lambda_1, ..., \lambda_N)$. This iterative calculation is the focus of the FRAME algorithm [2].

2.2.6 FRAME algorithm

The FRAME algorithm develops a statistical model of a texture [2]. Images of a certain visual texture are considered to be instances drawn from a distribution $f(\mathbf{I})$, where $f(\mathbf{I})$ describes the process that produced the texture whether organic or man made. All samples from the distribution have a similar appearance. The goal of the algorithm is to make inferences about $f(\mathbf{I})$ based on the observed samples leading to a model of the texture $p(\mathbf{I})$. The model is considered an accurate representation if samples of $p(\mathbf{I})$ are indistinguishable from samples of $f(\mathbf{I})$. The metric for comparison may be either visual inspection of the samples or some mathematical measure. The texture is modelled by a set of filters and the statistics related to the observation's response to those filters. A flowchart of the main processes in the algorithm is shown in Figure 2.4.

The algorithm is initialised with a data observation and a filter bank of all the filters that may be used. The selection of filters to comprise the bank is very important as a texture with features that cannot be detected by any of the filters in the bank will be impossible to model accurately. A variety of common linear filters are used at different scales and rotations. The algorithm does not require that the filters be linear but this limitation is assumed as it allows for much faster computation [1].

To initialise the algorithm the observation is filtered by each filter in the bank and histograms of the responses are calculated. The filter resulting in a histogram with the maximum entropy is selected. The entropy of a probability distribution is calculated by

$$S = -\int p(\mathbf{I}) \log p(\mathbf{I}) d\mathbf{I}. \tag{2.18}$$

Selecting the histogram with maximum entropy is equivalent to selecting the histogram containing the most information. The filter is removed from the filter bank and added to the model.

With one filter, we have the statistic $H^{(1)}(\mathbf{I})$ being the marginal distribution (the histogram) of the observation's response to that filter. This marginal distribution is the ϕ function of the ME calculation described above. The model is defined by a set of filters and an array of Lagrangian multipliers associated

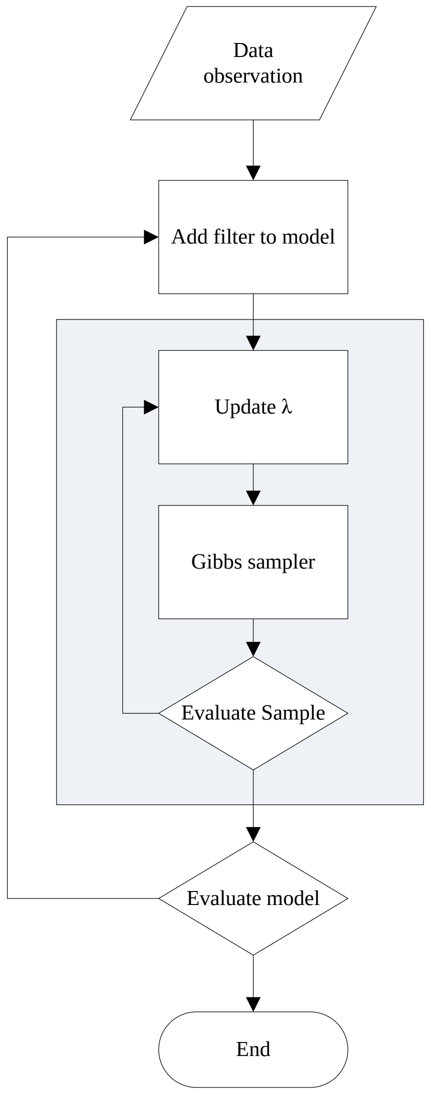


Fig. 2.4: Basic flowchart of the FRAME algorithm.

with each filter:

$$p(\mathbf{I}) = \frac{1}{Z(\Lambda_K)} e^{-\sum_{\alpha=1}^{K} \langle \lambda^{(\alpha)}, H^{(\alpha)} \rangle} \tag{2.19}$$

for K filters. This is a discretised version of Equation 2.16 and the same solution applies. The Lagrangian multipliers λ, which act like weights to the histogram bins, are calculated numerically by iterative gradient descent.

At each iteration a sample is drawn from $p(\mathbf{I})$ and histograms, H, are calculated for each filter in the model. The term $Z(\Lambda_K)$ is a normalising constant which is not generally necessary to calculate because the probabilities that are calculated from the model are compared to each other. This model treats the texture as a realisation of a Markov random field (MRF) — the probability of each pixel being a certain colour is influenced only by the neighbouring region of pixels. In this algorithm, the neighbourhood is the set of pixels covered by a filter centered on the pixel in question.

Samples are drawn from $p(\mathbf{I})$ using the Gibbs sampler. The sampler is run for a fixed number of iterations, some multiple of the number of pixels in the image — a sweep. The number of steps used in the sampler is kept quite low to speed up the execution of the whole algorithm. This means that the Gibbs sampler does not have time to converge on a true sample of the distribution at each iteration.

The model is updated after each sweep by making small changes to the λ values so it is not so important that a true sample is drawn each time. Nevertheless these changes are small enough such that each time the Gibbs sampler is run, the target distribution is approximately the same. As long as the Gibbs sampler is run for long enough to allow the samples to approach samples from the true distribution, the FRAME algorithm is able to converge on the correct λ values through Equation 2.20.

The algorithm alternates between steps of updating λ and drawing samples from $p(\mathbf{I})$ until the λ values converge or some other stopping criterion is met. The values are updated by

$$\lambda_{n+1}^{(\alpha)} = \lambda_n^{(\alpha)} + \left(H_{syn}^{(\alpha)} - H_{obs}^{(\alpha)} \right), \tag{2.20}$$

where H_{syn} and H_{obs} are the histograms of the current sample and the observation images respectively for each filter α in the model. The differences between the elements of those histograms are used to push the elements of λ towards values which make $p(\mathbf{I})$ best represent the true distribution underlying the texture. If a certain bin in the sample's histogram has a higher value than that of the observation, the corresponding element of λ is increased. Referring to Equation 2.19 it can be seen that positive λ values decrease the probability of the corresponding histogram bin. Samples which minimise that bin are preferred by the model, and the histograms of the samples tend to become similar to those of the observation.

Once the λ values have converged another filter is added to the model from the filter bank. Subsequent filters are selected based on the difference between the histograms obtained from filtering the observation and from filtering the latest sample of $p(\mathbf{I})$. The filter with the largest sum of absolute differences is chosen so that at each step the filter which contains the most new information about the texture is added to the model. Then the algorithm starts again with a new synthesis to obtain λ values for the current set of filters. The algorithm continues adding filters until the model contains a specified number of filters or the remaining filters in the bank are deemed not to capture any valuable information about the observation.

3. IMPLEMENTATION

Monte Carlo algorithms in general require significant computation. They are iterative procedures that can typically require thousands of iterations to converge [26]. In the case of FRAME, each iteration itself is time consuming leading to an algorithm that can take too long to run for practical purposes. One of the goals of this work is to improve the efficiency of the algorithm. The computer program that implements the alogirthms used for this research is described. This chapter explains the parameters which are available for experimentation and a GPU implementation for part of the algorithm. The GPU is used in order to decrease the time that the algorithm takes to produce a viable model, with a view to making FRAME more useful for practical applications.

3.1 Algorithm implementation

The FRAME algorithm is implemented, for the experiments in this dissertation, in a C# program. The algorithm is computationally very expensive, requiring many iterations to converge. The implementation attempts to reduce computation by storing as many results as possible. There is a trade off between memory usage and the number of calculations required. However the amount of memory required does not approach the limits on a modern computer with the program typically requiring about 60 Mb of RAM for the inputs described here. C# is chosen for the development in order to take advantage of the Visual Studio environment and for its data structures which simplify the handling of the vast numbers of variables in the program.

3.1.1 Constants

A single experiment comprises running the FRAME algorithm on an image until a specified number of filters have been included in the model. There are a number of parameters which can be specified to alter the behaviour of the algorithm. One of the goals of this work is discover which values for those parameters optimise the performance of the algorithm. These parameters define upper limits on the number of iterations to run (if the convergence conditions are not met), the number of colours to include in the palette and the number of bins in each histogram.

Colour palette

The colour palette is formed of the specified number of levels evenly distributed between 0 and 1 inclusive. The first step in the program is to resample the input image to only contain colours from the palette. In some cases a lot of the structure in the original texture is lost when resampling so it is important to compare the resampled image to the original when assessing the results.

The number of colours, more specifically grey levels, to use in the algorithm is a compromise between accuracy and execution time. The more colours available, the greater the detail which can be seen in images at the same resolution. Each time the colour of a pixel is changed, the probability of each available colour is calculated, so the number of calculations done by the Gibbs sampler is proportional to the number of colours. More colours also lead to slower convergence as the problem space becomes much larger. For the sake of the majority of the experiments — which are designed to optimise the other parameters — four colours are used. This is the minimum number of colours which can be used that still allows for visually interesting textures.

Histograms

The behaviour of the FRAME algorithm depends largely on histograms and how they are implemented. The histograms used in the algorithm are counts of the relative frequencies of ranges of values within filter responses. The λ

vectors, which define the model, have the same structure as the histograms as they are calculated from the differences between histograms. In order to allow meaningful subtraction of histograms, all the histograms within the algorithm are constrained to have the same bins and then the bin values are treated as vectors.

The number of bins to use is a parameter of the algorithm. The images themselves have a finite set of intensity values, but the filters and consequently the responses are continuous. If there are too many bins the histograms will be too sparse to allow λ to converge. When the number of bins increases, the width of each bin necessarily decreases, so that if there are too many bins, the information is lost as all of the bins will have low counts. Conversely is there are too few bins the histograms will not capture enough information. The histograms used have 64 bins on the range $[-8; 8]$. The number of bins used is a compromise corroborated by experimental results. The maximum magnitude of a value that the responses to the available filters might contain is 8.

Histograms are a type of probability distribution function. If there is a small offset in one region of the data being histogrammed, it may lead to a large change in the histogram if the values are near to the border of the bins. Consequently the difference between two histograms of similar responses may be much larger than we desire for the difference between the information they represent. To account for this, the histograms and λ vectors used in this implementation are smoothed by a Gaussian function. Each bin in the histogram is multiplied by a Gaussian kernel and the results are summed to produce a smooth curve [31]. The windowed functions are calculated by:

$$H(x) = \frac{1}{Nh}\sum_{i=1}^{N} K\left(\frac{x - X_i}{h}\right), \tag{3.1}$$

where h is the width of the bins and there are N bins. The kernel function K is the Gaussian,

$$K(x, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2}, \tag{3.2}$$

with the variance, σ, as a parameter of the algorithm. The effect of varying σ is explored in Section 4.4. An example of a smoothed histogram is shown in Figure 3.1.

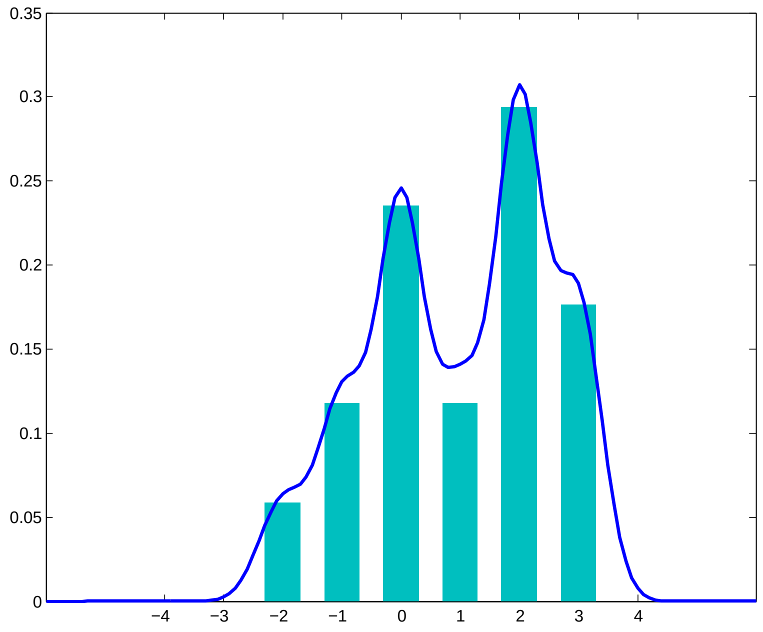


Fig. 3.1: Example of a histogram and its smoothed function.

Filter bank

The information contained in the texture model depends on the filters that are available in the filter bank. Filters are added to the model based on the amount of information in their responses. The responses containing the most information correspond to the filters that detect features in the texture. If all the filters in the bank are inadequate to detect the features of a texture, the model will not be very accurate.

In this implementation only linear filters are used. The filter bank is based on the filter bank of the original authors [2]. Four types of filter are used: Gabor filters, Laplacian of Gaussian filters, Sobel edge detectors and Difference of Gaussian filters. The Gabor filters come in pairs with kernels given by a

Gaussian,

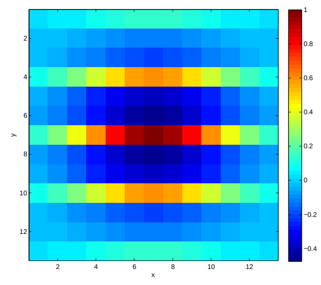
$$\mathrm{g}(x, y, \theta, t) = \exp\left(\frac{1}{2t^2}\left((x\cos\theta + y\sin\theta)^2 + (-x\sin\theta + y\cos\theta)^2\right)\right), \tag{3.3}$$

modulated by a cos and sin component respectively:

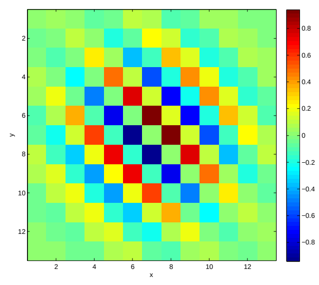
$$\mathrm{Gcos}(x, y, \theta, t) = \mathrm{g}(x, y, \theta, t)\cos\left(\frac{2\pi}{t}(x\cos\theta + y\sin\theta)\right), \tag{3.4}$$

$$\mathrm{Gsin}(x, y, \theta, t) = \mathrm{g}(x, y, \theta, t)\sin\left(\frac{2\pi}{t}(x\cos\theta + y\sin\theta)\right), \tag{3.5}$$

for a range of rotations θ and scales t. The range of t values is $\{1, 1.5, 2, 3, 4\}$ and of rotations is $\{0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}, \pi, \frac{5\pi}{4}, \frac{3\pi}{2}, \frac{7\pi}{4}\}$. An example of each is shown in Figure 3.2.



(a) Gcos$(0°, 3)$



(b) Gsin$(135°, 3)$

Fig. 3.2: Examples of Gabor filters.

The second type of filter comprising the bank is the Laplacian of Gaussians:

$$\mathrm{LG}(x, y, t) = \frac{(x^2 + y^2 - t^2)}{t^2} e^{-\dfrac{x^2+y^2}{t^2}}, \tag{3.6}$$

for scales t. These are circular filters of various sizes. In the filter bank, t has the values $\{0.7, 1, 1.5, 2, 3\}$. Figure 3.3 shows an example $t = 2$. The kernel is scaled by t^2 to limit the range of values in the response. This is important when working with standardised histograms that are treated like vectors.

Sobel edge detection filters are also included in the filter bank, however these are rarely chosen for the model, presumably because the Gcos filters are better for detecting those edges.

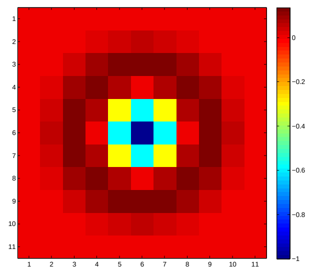


Fig. 3.3: $LG(2)$.

A special case of linear convolution filter is the impulse filter. It has a 1×1 kernel with value of 1. The convolution of an image with the impulse filter is the image itself. This is useful in this application because the histogram of the response is then the distribution of the different colours in the texture. A white noise image has a flat histogram as its response to the impulse filter. Images which are dominated by one or two colours will have peaks in their intensity histograms.

Another class of filters included in the bank are difference of Gaussians (DoG). These are not used by the original authors, but have application in texture modelling [32]. The filters are constructed by subtracting two Gaussian kernels with different variances. The effect is analogous to a bandpass filter, so they are useful for detecting specific spatial frequencies in the textures. The filters are parameterised by s, the smaller variance, and K, the ratio to the larger one:

$$\mathrm{DoG}(x,y,s,K) = \frac{1}{2\pi s^2} e^{-\dfrac{x^2+y^2}{2s^2}} - \frac{1}{2\pi K^2 s^2} e^{-\dfrac{x^2+y^2}{2K^2s^2}} \tag{3.7}$$

These filters are similar to Mexican hat filters and in the case of $K = 1.6$ are approximately equal to LG filters. An example of a DoG filter is shown in Figure 3.4.

The filters in the bank have $K = \{1.2, 1.5, 1.8, 2, 2.5, 3, 4, 5\}$ and $s = \{0.4, 0.6, 0.8, 1, 1.2, 1.4\}$, provided that the total width of the filter is less than or equal to 16 pixels. That constraint is enforced because larger filters cover too much of the images and take very long to compute.

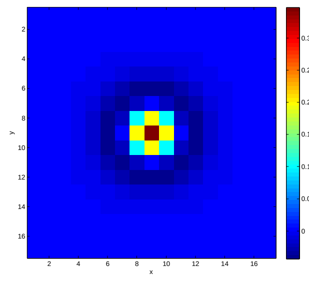


Fig. 3.4: DoG$(1.2, 2)$.

Convergence

FRAME is run for a maximum of 5 000 iterations for each set of filters, with 4 sweeps per iteration. That is 5 000 steps of updating λ and 20 000 sweeps of the Gibbs sampler. That number of iterations takes a very long time to run – on the order of 4 hours. In most cases, the algorithm will terminate before then if the model is considered to have converged. There are three heuristic convergence conditions which are tested for after each iteration: the error is sufficiently small; the error is increasing rapidly (considered to be diverging) and; there is no change in the synthesis.

The error is measured for each filter independently by

$$e = \sum_n \left(H_{syn}(n) - H_{obs}(n)\right)^2 , \tag{3.8}$$

for each element n of the histogram. The execution stops and another filter is added to the model when one of two conditions is met. The filter is selected by using Equation 2.18 to determine which one captures the most new information. The two conditions are: if the error value for each filter in the current iteration is less than 5 % of the initial error; or if it is twice that of the error in the previous iteration. Those limits are chosen heuristically based on observations that the synthesis does not improve much beyond those points.

The second condition is useful for cases when the synthesis breaks down and becomes unstable. In some examples the differences in probability is so small that the colours are simply selected at random and the algorithm diverges.

There are some cases when the model becomes so specific that the Gibbs sampler does not change any of the values in the synthesis. In these cases the execution is also stopped.

3.2 Parallelisation

When the FRAME algorithm is executed, the Gibbs sampler is required to produce a sample of the proposal distribution each time the λ values are adjusted. It takes hundreds of iterations for those values to converge. The number of pixels to be updated at each step is chosen to be four times the total number of pixels in the image, following the original work of Zhu et al. [2]. For each of these pixels, the Gibbs sampler requires a probability of the pixel being each colour. That probability depends on the histogram produced by each of the filters in the current model. For a simple case of a 32×32 pixel image with eight colours and three filters that equates to producing ten thousand histograms per iteration. This is extremely inefficient so a number of measures are taken to optimise the process.

All the filters in the filter bank are linear and smaller than the entire image. Consequently changing the colour of a single pixel affects a small region of the filter response and this change can be calculated by a single addition [1]. The change to the filter response when adjusting the intensity of a pixel is simply the response plus the impulse response of the filter scaled by the change in intensity. If the intensity pixel p is changed from C_0 to C_1 the filter response $\mathbf{F}$ becomes:

$$\mathbf{F}_{n+1} = \mathbf{F}_n + (C_1 - C_0)\mathbf{K}_p \tag{3.9}$$

where $\mathbf{K}_p$ is the filter kernel centred on p. This calculation is far simpler than computing the filter response each time and only requires the convolution be computed once per filter.

Extracting a histogram from the filter response is also an expensive operation, requiring the intensity of each pixel to be assigned to one of the bins. This can be obviated in a similar way. The histograms of all the responses for the current sample are stored. As the pixels are flipped to create new samples a count is kept of the changes to each bin in the histogram. For each element of the response that changes its value, 1 is added to the bin containing the new value and subtracted from the bin that previously contained it.

This is the operation that is run on the GPU. The pixels to be considered are passed to the GPU code and it returns the changes in histogram bin counts for each colour for each filter. The data required by the GPU are the current sample; the filters in the model; the current sample's responses to each filter; the histograms of those responses; and the set of colours (or intensities) that may be used. One thread is created for each set of pixel, colour and filter. Each thread is required to perform the addition of the filter kernel and to count the changes to the histogram. It is a very simple computational task, requiring very few instructions to execute — making it ideal for processing on the GPU. There is an inefficiency caused by the fact that all the threads must wait for the thread handling the largest filter to complete. However, none of the filters is very large so this is not a serious bottleneck.

3.2.1 Hardware considerations

The GPU hardware used is a NVIDIA GeForce GTX 470 [33]. Programming for a GPU is sensitive to the specific details of the hardware. Some specifications of the card are given in Table 3.2.1.

Tab. 3.1: GTX 470 specifications.

CUDA cores per SM	32
Maximum threads per SM	1536
Maximum threads per block	1024
32-bit registers per SM	32 K
Global memory	1280 MB
Shared memory	48 KB
Constant memory	64 KB

These are significant because in order to get the best performance the program must run within the constraints of the hardware while using as much of the power available in each streaming multiprocessor (SM) as possible.

3.2.2 GPU program

The GPU code is written in CUDA C [33]. In this platform threads are grouped into blocks with each block containing an array of threads. The fastest type of memory to use is the registers but the space available is limited. Threads from the same block have access to the same block of “shared memory”. Shared memory is similar to cache memory on a CPU; it is much faster than the global memory and it has much higher bandwidth to the SM. In addition there is a region of memory called “constant memory” which is available globally, to all threads. Access to data stored in constant memory is cached to increase efficiency. If all the threads in a block need to access something in the constant memory, one slow fetch is required, and subsequent reads are done much more quickly from the cache.

To initialise the GPU code the filters, current sample and its responses are copied from CPU memory to the GPU. The filters are placed in constant memory as they are the data to be accessed the most often and do not change throughout the execution. If the maximum size of a filter is 32×32 and the values are single precision floating point numbers, the amount of constant memory available limits the maximum number of filters to 16. This is an acceptable restriction as the textures considered in this work can be well modelled using fewer filters (or the filters required are not available in the bank). The current sample and responses cannot fit in constant memory, although they are constant in the scope of the GPU. There is still a performance gain by storing them globally as it reduces the amount of data that needs to be transferred.

A thread block is created for each pixel under consideration. The thread block contains a two dimensional array of threads — one for each combination of filter and colour. The parameters to the thread are the pixel’s coordinates, the set of colours and the borders of the histogram bins. Each thread in the block computes the histogram changes. These can be stored in registers as only one integer is required per histogram bin. Once a thread completes, it waits for all the threads in its block and then the histogram changes are combined into one array to be returned to the CPU.

The probabilities of the colours are calculated on the CPU from the histograms. The CPU updates the sample accordingly and changes the responses for the next iteration. The GPU returns the array of bin changes to the CPU. These are then interpreted by the main program, giving probabilities for each colour. A new colour for the pixel is selected at random, with likelihood in proportion to its probability. After all the colour changes have been made, the filter responses are updated by Equation 3.9 and another iteration is sent to the GPU. These computations are not trivial; they contribute significantly to the overall running time of the algorithm.

4. EXPERIMENTS

The algorithm is investigated through a number of experiments. The experiments are done by running the algorithm repeatedly while changing a single element, to determine the effect of that variation and to find the optimal parameters. The first section in this chapter describes the values which are observed in the experiments and their significance. Baseline tests are done to produce a comparison point for those values. Three novel variations to the algorithm are implemented and tested: simulated annealing on the rate of change of λ; including the impulse filter in all models; and applying a Gaussian smoothing function to the histograms. Timing data is collected specifically in regard to the GPU implemented portion of the algorithm to determine its effectiveness in decreasing the computation time. Finally an experiment is performed to test FRAME models for classification purposes. A specific classification experiment with froth images is presented.

4.1 Experimental set up and methodology

A number of experiments are performed to test the FRAME algorithm and explore possible improvements to it. The performance of the algorithm is assessed in two ways: quantitatively, based on an error value; and qualitatively, by comparing the synthesised texture to the observation. The error measure is the sum of squared differences between points in the observation histogram and the synthesised histogram. It is recorded each time λ is updated. This is an appropriate error measure as it will be zero for a perfect synthesis; however it can also go to zero for a poor synthesis under an inadequate model. The error also gives an indication of how much the λ values are being altered by at each iteration. Snapshots of the synthesis are recorded every 50 iterations.

A baseline experiment is done on a number of textures as a point of comparison for the experiments. The settings and parameters for those experiments are as described in Section 3.1. The observation images used are shown in Figure 4.1 with the name they are given for convenience.

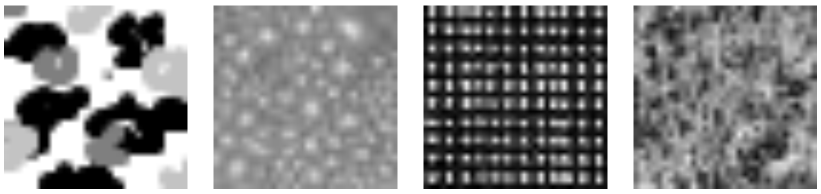

(a) Patches (b) Froth class 3 (c) Grid [34] (d) Fur [34]

Fig. 4.1: Texture images – 32×32 pixels.

The algorithm is run until the model contains 4 filters. As expected, the synthesis improves for each added filter. For example consider the Grid texture. The best synthesised image — the one having the lowest error — for a model with 1 through 4 filters is shown in Figure 4.2. The filters in that model are Gcos$(90°, 3)$, Gcos$(0°, 1)$, Gcos$(0°, 3)$, Gsin$(90°, 4)$. It is intuitive that those filters would be chosen as the observation is made up of vertical and horizontal structures.

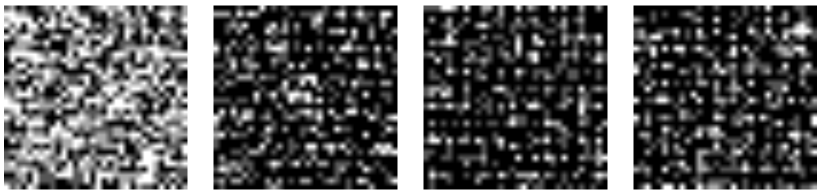

(a) 1 filter (b) 2 filters (c) 3 filters (d) 4 filters

Fig. 4.2: Synthesised images for the Grid texture.

The appearances of the synthesised images in these tests are not very impressive. There are clearly vertical and horizontal lines on a dark background but these are poorly defined as compared with the observation. One purpose of the experiments in this chapter is to investigate whether this performance can be improved.

To demonstrate how the synthesised textures are able to be tiled, the output from the Patches example is copied four times. This image is shown in Figure 4.3.

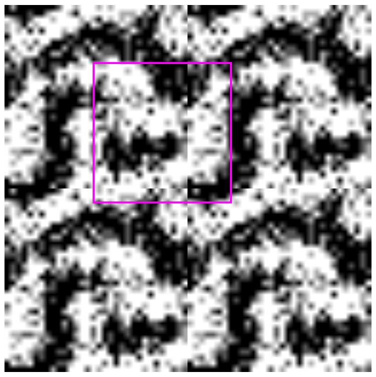


Fig. 4.3: Tiled synthesis of the Patches texture.

A number of interesting observations can be made from this experiment. The assumption that the observation is on a toroid gives a different perception to the structures in the texture. Although the image in Figure 4.3 does not look very similar to that in Figure 4.1(a), the boxed region in the synthesis shows how that sample may have been taken from the same texture.

This synthesis is produced from a model containing 3 filters: $\text{Gcos}(0°, 1)$, LG(3) and $\text{Gcos}(0°, 4)$. The results from this run are better than those from the run with 4 filters because the fourth filter is $\text{Gcos}(90°, 4)$ which is perpendicular to $\text{Gcos}(0°, 4)$. Consequently the syntheses in that experiment tend to contain diagonal bands, however such images do not satisfy the other two filters well. The synthesis tends to switch between two states and the convergence is unstable. A plot of the error values is shown in Figure 4.4.

This figure shows how the errors for the first two filters spike upwards relative to the other two, indicating that the convergence of those parameters are at odds with each other. Nevertheless the algorithm is eventually able to converge.

Figures 4.5 and 4.6 show the observation histograms and the λ functions for the $\text{Gcos}(0°, 1)$ and $\text{Gcos}(0°, 4)$ respectively. These graphs show that the λ values for the larger $\text{Gcos}(0°, 4)$ filter are much more finely tuned than those for $\text{Gcos}(0°, 1)$. This difference can be quantified by calculating the average displacement from 0 of each λ function:

$$\bar{d} = \frac{1}{N}\sum_i |\lambda(i)|, \tag{4.1}$$

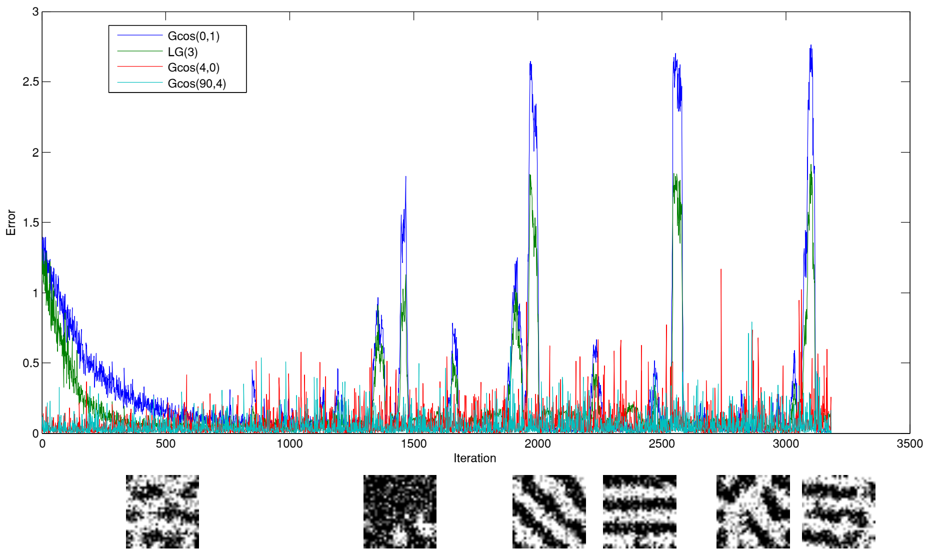


Fig. 4.4: Graph of errors against iterations from the Patches experiment with 4 filters. Snapshots are shown below the graph at 500, 1450, 2100, 2400, 2928 and 3200 iterations. The snapshot at 2928 (second from the right) is the one with minimum sum of errors.

where N is the number of iterations. In this case the values are 2.18 and 0.33 respectively. These values are a good indication of how much refinement is done on the corresponding λ parameter, with a larger value indicating that the values changed more quickly. The smaller value relating to Figure 4.6 is indicative of the small changes that produced a more varied curve.

The values in Figure 4.5 are all greater than 0, this is because the Gcos filter is all positive, so the response in turn is also positive. In both figures it can be seen that large positive values correspond with negative λ values. Referring to Equation 2.19, a negative product between λ and H increases the probability.

The results in this section are intended to give an overview of the data that is collected during the experiments and its interpretation. The following sections describe the experiments which are done and explain their results.

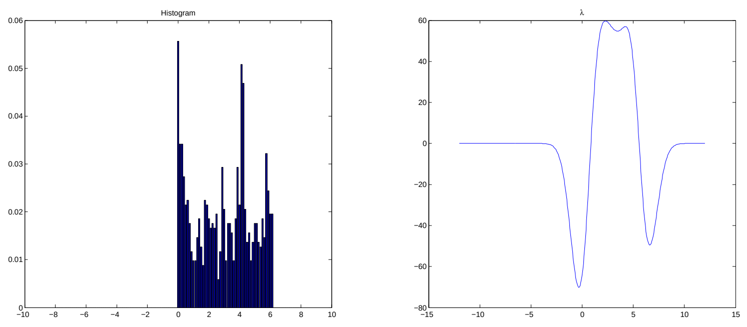


Fig. 4.5: Graphs of the observation histogram and final λ function for the $\mathrm{Gcos}(0^{\circ}, 1)$ filter.

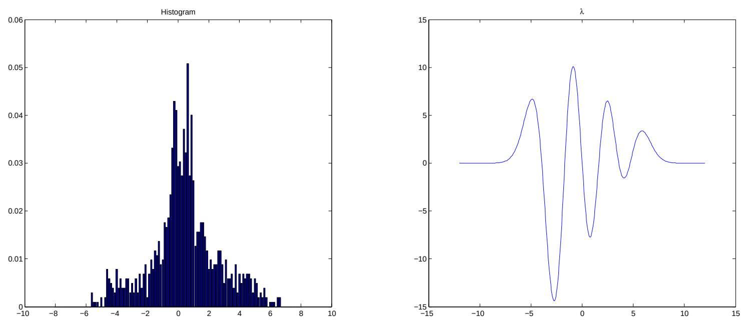


Fig. 4.6: Graphs of the observation histogram and final λ function for the $\mathrm{Gcos}(0^{\circ}, 4)$ filter.

4.2 Simulated annealing

Simulated annealing is an optimisation technique inspired by statistical mechanics [35]. Annealing is a process for tempering metals whereby the metal is heated to high temperature and then cooled in order to improve some property, such as the strength, at room temperature [36]. This is related to statistical mechanics because the energy of the atoms increases during the heating, allowing them to be rearranged in a more stable configuration as the metal cools. This concept is applied to multivariate optimisation by initially adjusting the parameters in large increments (at high temperature) and then by decreasing that rate (as the optimisation cools). This is an attractive method to implement in FRAME because the lattice structure of atoms in a solid is an MRF.

This experiment is done "heating" the λ parameter — causing it to change more quickly in the first few iterations. Equation 2.20 is modified to be

$$\lambda_{n+1}^{(\alpha)} = \lambda_n^{(\alpha)} + T_n \left(H_{syn}^{(\alpha)} - H_{obs}^{(\alpha)} \right) \tag{4.2}$$

with temperature T. There are 3 parameters in this experiment: the initial temperature, the annealing rate and the minimum temperature. The temperature decays in each iteration after the λ values are updated according to:

$$T_{n+1} = T_n(1 - \Delta t) \tag{4.3}$$

where Δt is the annealing rate. The minimum temperature is constrained because if it is allowed to be much less than 1, the λ values will be changing too slowly to converge.

Introducing the parameter T causes the algorithm to converge much more quickly, as the rate of change of λ becomes much higher. While this is desirable, the value of T_0 cannot be too high because the model will get stuck in a local minimum. The highest acceptable value depends on the observation texture. The Patches and Grid textures are more easily modelled by the available filter bank than the Froth or Fur textures. Consequently a higher value may be used for those observations.

4.2.1 Results

The simulated annealing experiment has three parameters: the initial temperature T_0, the annealing rate Δt and the minimum temperature T_{min}. The effects of each of these is tested by running the FRAME algorithm on each texture for different values. The annealing schedule in Equation 4.3 causes the temperature to decay exponentially.

The rate of decay is controlled by Δt. To illustrate the effect of this parameter on the convergence of the model, consider two of the results from the patches texture with annealing settings $\{T_0, \Delta t, T_{min}\}$ of $\{16, 0.05, 1\}$ and $\{16, 0.001, 1\}$ respectively.

Comparing Figures 4.7 and 4.8 to Figure 4.4, where $T = 1$, the benefit of increasing T_0, can be seen. With $T = 1$ it takes over 500 iterations to reach an

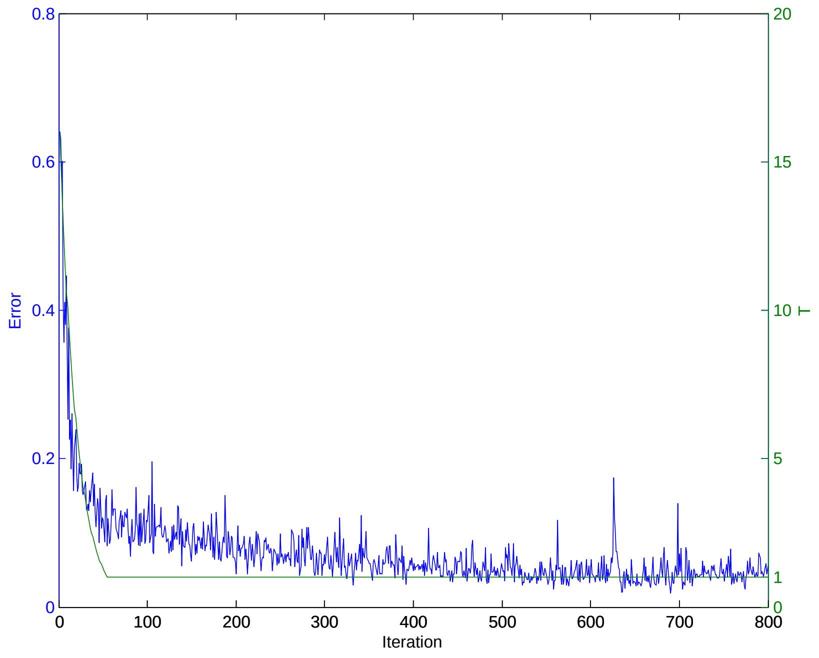


Fig. 4.7: Graph of the mean error for 4 filters and the T value with annealing settings $\{16, 0.05, 1\}$.

error less than 0.1, compared to 100 and 20 iterations in this experiment. In the case of $\Delta t = 0.001$, the convergence is unstable with the algorithm exiting on a high error condition. Although it takes longer to converge, the performance with $\Delta t = 0.05$ is preferable. In order to obtain an accurate model, the algorithm must be able to make small changes to λ so it is necessary to allow T to decrease to 1 fairly quickly.

The synthesis images with the lowest errors are shown in Figure 4.9. Both images show an acceptable likeness in subjective appearance to the patches texture which may expected since both runs achieved a similar minimum error. However the synthesis for Δt is superior as it better matches the colour profile of the observation – it has significantly more grey pixels. This is indicative of the longer sampling time that is able to produce a more refined synthesis because it is drawn from a more precise distribution – one which more accurately represents the observation.

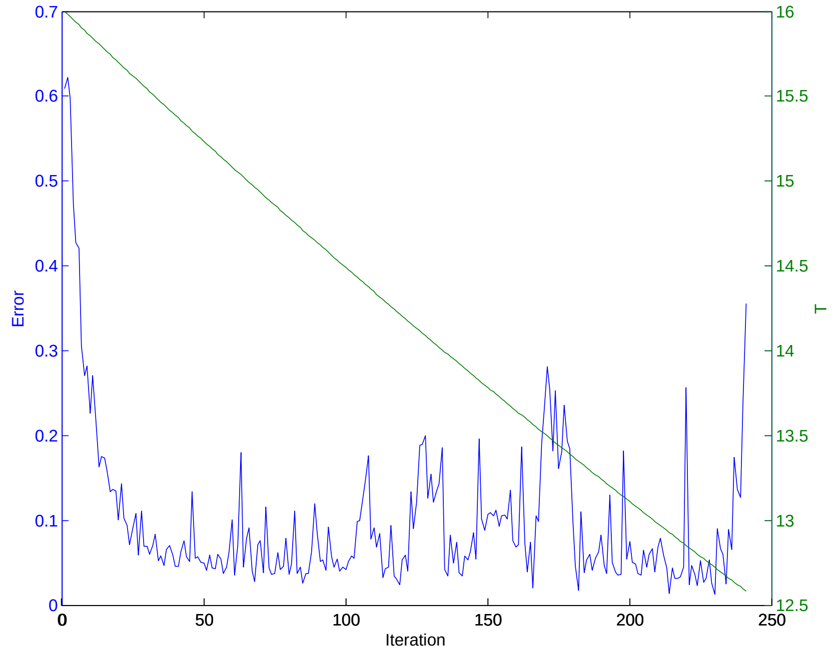
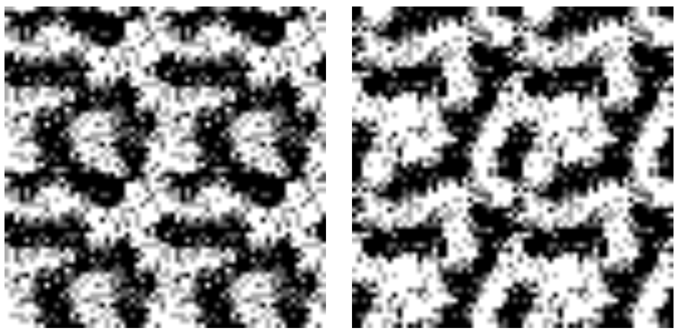


Fig. 4.8: Graph of the mean error for 4 filters and the T value with annealing settings $\{16, 0.001, 1\}$.

The results for each texture show a similar pattern. The grid pattern, which is the most easily modelled by the filter bank, is stable even with $T_0 = 64$. In general the best set of values across all textures is $\{32, 0.05, 1\}$ as it gives the fastest convergence while still providing an accurate model. The introduction of the T variable into the algorithm provides a significant improvement in the number of iterations required to converge, thereby increasing the practicality of the algorithm.



(a) $\Delta t = 0.05$ (b) $\Delta t = 0.001$

Fig. 4.9: Synthesised images of the patches texture from the annealing rate experiment. Synthesis images have been tiled in the same way as Figure 4.3.

4.3 Impulse filter

The authors of a related work that uses a texton based model, suggest that it is always beneficial to include the impulse filter in the texture model [37]. A texton is the minimal unit of a texture, analogous to a pixel. The model in that work has a similar feel to FRAME and the textons are detected by filters. The impulse filter is not automatically included in any of the preliminary tests described in Section 4.1.

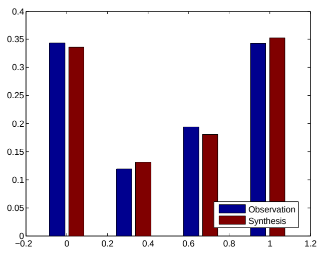
Clearly the intensity profile of an image is an important part of its appearance. An experiment is done to assess the impact of forcing the impulse filter to be included in the model. The model is initialised to contain the impulse filter. A sample is not drawn from that model, as it would not have any structure and would simply be a random image with the same distribution of colours as the input. Other than that one change, FRAME is run as before.

For some of the textures in the baseline tests, an impulse filter is not required as the colour histogram for the synthesis is similar to that of the observation. The colour histograms from the baseline tests are shown in Figure 4.10.

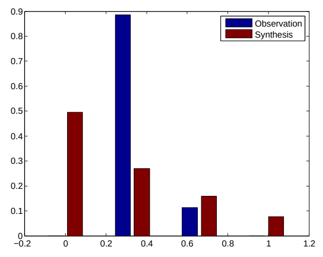
The test with the worst matching colour profile is the froth image. This is also the only model which does not include the $\text{Gcos}(0°, 1)$ filter. This implies that that filter captures the colour information in the image. The filter kernel is simply a 5×5 matrix with a central peak, a standard low-pass filter. The response has similar structure to the input but the edges are smoothed. The range of values in the response is also greater than that of the input, thus making it easier for those statistics to be captured by the smoothed histogram.

The results shown in Figure 4.10 suggest that the implementation has a bias towards selecting 0. This is probably caused by all colours being assigned very low probabilities (since the value is calculated in terms of the whole image) and then the program assigns 0 as a default. This behaviour is not ideal and suggests that some refinement is required.

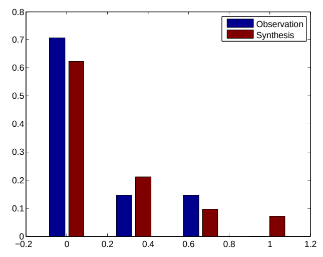
All the experiments which included the impulse filter in the model showed no noticeable improvement in the synthesis. The intensity information is included implicitly by other filters. Furthermore, the effect of the impulse filter is diminished because it carries less information than the other filters. This is an



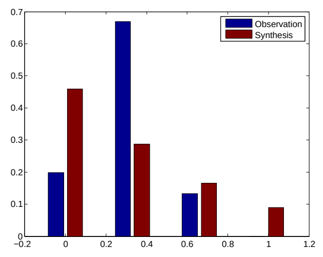
(a) Patches



(b) Froth class 3



(c) Grid



(d) Fur

Fig. 4.10: Colour histograms for the observation and synthesis images from the baseline tests.

indication of how the maximum entropy principle applies throughout the algorithm, it works best when the filters are chosen accordingly.

4.4 *Histogram variance*

The model which is produced by this algorithm depends heavily on the way in which histograms are treated. The histogram is a discretisation of a probability density function. As discussed in Section 3.1.1 the histograms in this implementation are smoothed with a Gaussian kernel such that they resemble continuous functions. The variance of that Gaussian controls how broad the kernel is, and how smooth the histogram becomes. In this experiment, FRAME is run with the variance set to 0.0001 (essentially no smoothing takes place), 0.25, 0.5, 0.75, 1 and 1.25.

In tests with the variance set to 0.0001 the algorithm takes much longer to converge, in the order of 100 times. For the more complex textures, it does not converge at all with the detected features being too specific to replicate from the Gibbs sampling. This affirms the modification to the algorithm.

Changing the variance changes the histograms throughout the algorithm, so different filters are selected in the different runs. We will focus on the results from the Grid texture since the same filters are included in the model for variance equal to 0.5, 0.75, 1, 1.25. In each case the model produced is approximately the same, containing the same set of 4 filters. For example, Figure 4.11 shows the λ functions relating to the Gcos$(90°, 3)$ filter for each case.

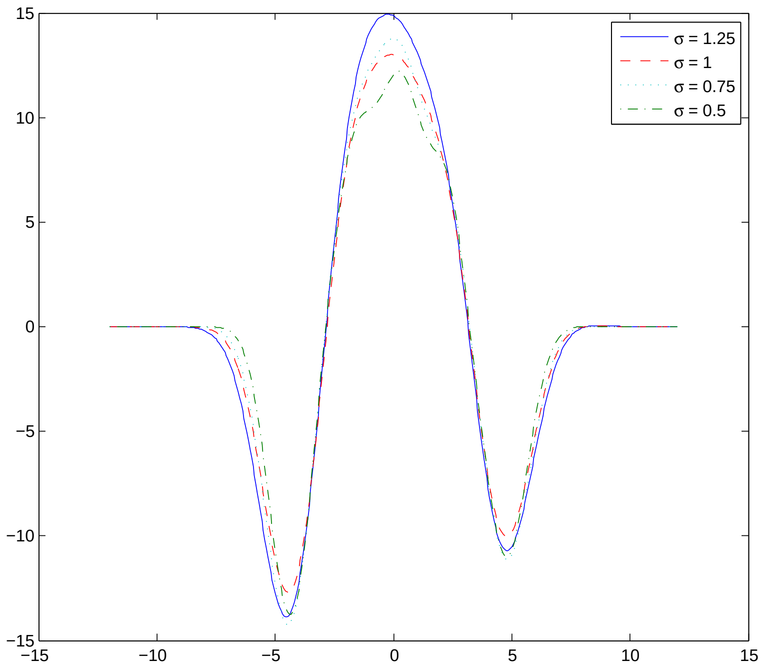


Fig. 4.11: λ functions for different variance values in the histogram Gaussian.

Figure 4.11 shows that the functions are almost the same in each case, indicating that the same model is developed. The main difference can be observed in the height of the central peak, however the shape of these functions is more significant than their magnitude and the maximum value is not a function of the variance. Only the case of $\sigma = 0.5$ shows some variation in shape. It is interesting to note that the same slope appears in each case. One might have expected that with a larger variance, the gradient would be lower. This graph suggests (at least for this texture) that each value of the variance is acceptable and leads to a model which is a good representation. This is corroborated by the synthesised images which all appear similar to each other and to the input.

The variance has an impact on the rate of convergence. This can be seen by the displacement measure in Equation 4.1. The average displacement values, as per Equation 4.1, for each filter in each experiment with the Grid are shown in Table 4.1. A higher value of average displacement indicates that the algorithm is more efficient; it takes fewer iterations to reach the same point.

Tab. 4.1: $\bar{d}$ values in the Grid experiment, for different filters and variance values.

	$\sigma = 0.5$	$\sigma = 0.75$	$\sigma = 1$	$\sigma = 1.25$
Gcos$(0°, 1)$	11.2470	10.0835	10.7823	9.5935
Gcos$(0°, 3)$	4.1333	3.2425	3.1606	2.6477
Gcos$(90°, 3)$	6.5531	5.2127	4.9631	4.1586
Gsin$(90°, 4)$	2.8372	1.4865	1.5576	1.3471

The results in Table 4.1 indicate that the lower the variance the faster the algorithm converges. However if the value is too low, the model tends not to capture the appearance of the texture successfully. This is because with sharper distributions, the algorithm is more likely to converge on a local minimum. So the textures which are synthesised minimise the error for the selected filters, but do not display the larger structures of the observed texture.

These tests suggest that a σ value of 0.75 is optimal. However any value within that region is suitable. The improvement is a result of smoothing the histograms and the effects of small changes to σ are minimal. All the results with a smooth histogram are superior to those with a discrete histogram.

4.5 GPU

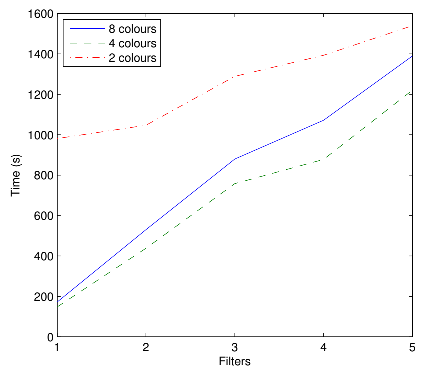
An experiment is done to assess the efficacy of the parallel implementation and to tune the parameters of the implementation. The experiment is run on the 32×32 pixel Patches texture. FRAME is run for 50 iterations per filter until the model contains 5 filters and repeated with a palette containing 2, 4 and 8 colours. This is far too few iterations to achieve a good model but is sufficient for the timing analysis that is done. The experiment is run 3 times to determine the ideal number of threads to run concurrently.

The Gibbs sampler is set to update 16, 64 and 128 pixels per iteration which are the number of threads being run simultaneously on the GPU. A modern PC can run 8 threads, so 16 is considered as the minimum reasonable number of threads to use. Also, given the slower clock speed of the GPU and the overhead, this test approximates running the algorithm on the CPU. At the other extreme, updating 128 pixels on a 32×32 image means that 12.5% of the pixels are updated simultaneously. Even at this value, the overlapping filter regions prevent the convergence of the Gibbs sampler so using more threads is not feasible. The timing results are shown in Figure 4.12.

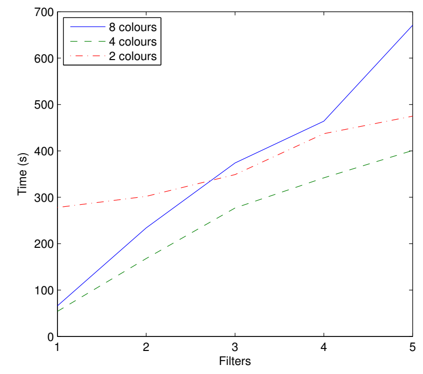
As expected the time taken increases as the model becomes more complex, when more filters or colours are added. However, there are some notable anomalies to this. In each experiment the 2 colour run takes the longest time with one filter. One would expect the 2 colour case to execute in the shortest time because with fewer colours there are fewer calculations. This inconsistency arises from the fact that all the filters are not the same size. When the observation is quantised to contain only two colours, the features become very large, so the first filter to be selected is larger than that selected for the 4 and 8 colour cases. The greater number of operations required to use this large filter outweighs the time saved by having fewer colours to consider.

The results clearly show that the more pixels which are updated simultaneously, the faster the algorithm runs. The upper bound on that number is set by the hardware and the size of the image itself. If a program attempts to launch more threads than the hardware can handle, some of the threads will be queued, decreasing the performance. The negative impact of this can be avoided by ensuring that the number of threads is a multiple of the capacity of the GPU.

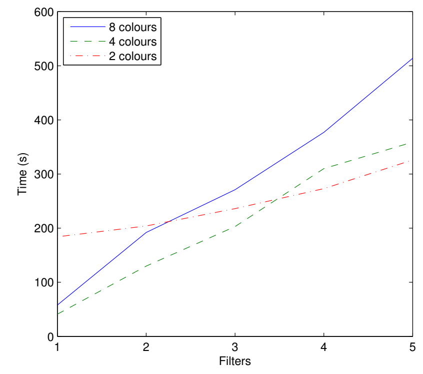
In this application it is incorrect to update too many components at once. Since the filters are larger than a single pixel, changing one pixel's colour has an effect on a region of the filter response. If two or more pixels with overlapping filter regions are flipped simultaneously, the probabilities assigned to their colours will be slightly inaccurate. The set of pixels to be updated at each iteration of the Gibbs sampler is chosen at random. It is possible that some of those will affect overlapping regions. However, since only a small proportion of the pixels



(a) 16 pixels



(b) 64 pixels



(c) 128 pixels

Fig. 4.12: Timing data for different numbers of simultaneous pixels.

is treated simultaneously — about 5 % — this is unlikely and the chain is still able to converge.

The issue of overlapping filter regions limits the maximum number of threads that may be used for a certain image size. When the size of the image increases, in width and height, the number of pixels and therefore calculations required to produce a sample increases quadratically. The number of pixels that can be safely updated simultaneously increases proportionally to the number of pixels. As a result the time taken to run the GPU code increases linearly with the image size within the limits of the hardware. This is a huge advantage as it allows the algorithm to scale up to more complex problems without incurring the penalty in execution time that it theoretically should. The time spent on the GPU is constant provided all thread can still run simultaneously.

The same applies when the complexity is increased by adding more colours to the palette. When going from 4 colours to 8 (the observation is almost the same in this case) the number of calculations required to produce a sample doubles. The graphs show that time increase between these experiments is minimal. This is because the execution time on the GPU does not change; more threads are added that can run simultaneously. The small increase is due to the linear operations done on the CPU in selecting the new colour to assign to each pixel.

Consequently, the complexity of the problem can be increased with minimal effect on the execution time. This feature of the implementation is extremely useful for studying the algorithm. It allows the code and parameters to be fine-tuned on simple test cases with the knowledge that the same program can be applied to more complex problems. These results demonstrate a significant improvement in performance by using a GPU. They suggest that the execution time could be even further decreased by implementing more of the algorithm on the GPU.

4.6 Classification

The statistical models generated by FRAME may be used for texture classification. Once the λ parameter has been calculated for a texture, the model can be used to assign a probability that another image is of the same texture. The image is filtered by each filter in the model and the histograms are calculated. These are then used in Equation 2.19 to give a probability that the model fits that image. (The term probability is used loosely here as the partition function is unknown. All values are considered relative to each other.) More accurately, it is the probability that this image is produced by the same process as the observation that led to the model. Given a set of models each for a different class of texture image, an unknown image can be classified as belonging to the class whose model returns the highest probability.

A dataset of images to be used in a classification experiment is constructed from the images shown in Figure 4.13. The dataset is created by taking random 32×32 pixel sections from the much larger images. In this way, we can ensure that the assumption that each image is produced by the same process is true. 100 images of each class are used as a training set, and another 100 are used as a test set. The images do not wrap a toroid as the algorithm assumes, however the textures are specifically chosen with that constraint in mind. The features are small and very repetitive.

For each texture in the experiment, a FRAME model is calculated based on one example image. The probability of each image in the training dataset is then evaluated under its model. A mean and standard deviation of the probability for each texture is calculated. The probability values are aggregated in this way to remove the effect of the partition function, which is not available and is different for each texture. When evaluating the testing dataset, each image is passed to the model of each texture. The class is assigned based on the statistics from the training phase. The probability is expected to be closest to the mean probability of the correct class. The results of this experiment are tabulated in a confusion matrix in Table 4.2.

Tab. 4.2: Confusion matrix for the FRAME classifier on simple textures.

		Assigned class			
		Fence	Fur	Lines	Wicker
Actual class	Fence	85	7	1	5
	Fur	0	88	0	10
	Lines	0	1	78	19
	Wicker	0	0	0	98

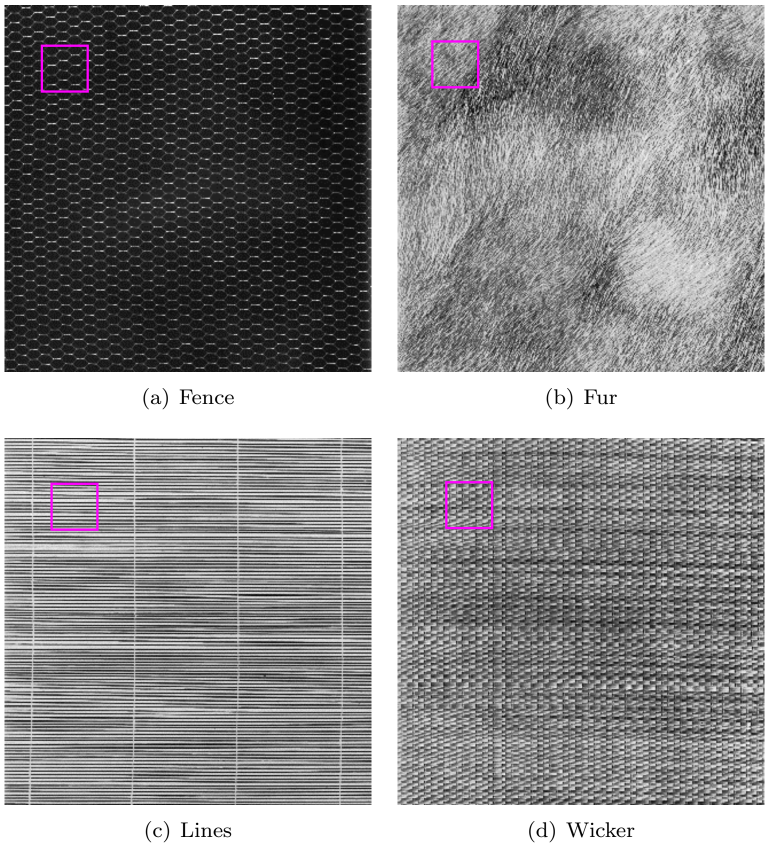

(a) Fence (b) Fur (c) Lines (d) Wicker

Fig. 4.13: Texture images used for the classification experiment, with a 32×32 pixel region indicated.

The overall success rate of this classifier is 89.03 %. The worst misclassification is between the Lines and Wicker textures. This is not surprising as both images contain prominent horizontal features. This experiment proves that FRAME models can be appropriate for texture classification. Once the model has been evaluated, the classification itself is extremely efficient.

4.6.1 Froth classification

An experiment is done on images of froth from platinum processing. One of the stages of extracting platinum from ore involves crushing the ore into a powder and mixing it in a tank of liquid. This mixture is then caused to froth in order that the platinum may be skimmed off the top. The nature of the froth varies depending on the chemicals in the mixture and how it is agitated. In order to optimise the process, the conditions in the tank have to be accurately controlled. At the moment, this is done by a human expert. It is desirable to have an automated system that can control the process. A crucial part of such a system would be the ability to classify the froth. This experiment tests whether the FRAME models of visual textures are effective for classifying the froth.

The dataset used contains 5 different classes of froth images. An example of each class is shown in Figure 4.14.

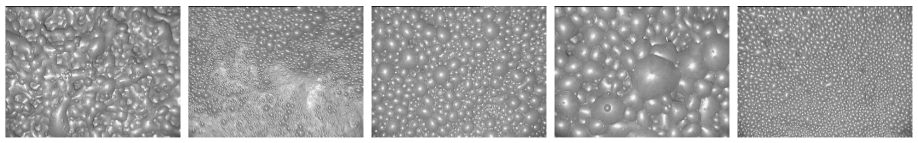

(a) Class 1 (b) Class 2 (c) Class 3 (d) Class 4 (e) Class 5

Fig. 4.14: Example images of the froth textures.

These images show that in this case, a scale invariant classifier is not desirable. The most noticeable difference between each image is the size of the bubbles. Also in this application, the camera angle is fixed so the scale, illumination and rotation (less significantly) of the images does not change.

For the sake of comparison a different algorithm is tested on the same dataset. A classifier similar to the one described by Varma and Zisserman [38] is implemented. This algorithm is chosen for its similarities to FRAME. The classifier is able to model a texture class based on a single observation image and the model depends on a bank of convolution filters. In order to make the comparison more meaningful, these two elements of each algorithm are held the same — each algorithm is trained on the same observation and uses the same filter bank.

The classifier falls into the broad category of bag of features classifiers described in Section 2.1.1. The features used are textons [39]. The first step is to build a dictionary of textons. This is done by filtering a number of images with the filter bank. In this implementation, 5 examples of each class are used. The filter responses are then combined to form a number of vectors in filter space — each pixel induces an N-vector, for N filters in the filter bank. The most significant vectors are chosen by K-means clustering, to form the texton dictionary. The parameter K is set to 20, following [38], so the dictionary contains 20 textons.

A model is learnt by filtering the observation with each filter in the bank. The filter space vectors of the resulting responses are each assigned to the nearest texton. Thus a histogram of texton frequencies is calculated for the training image. These histograms are the models for each class. Testing images are classified by comparing their texton histogram to the models and the image is classified to the most similar model. Texton histograms are compared by the sum of squared differences, which is an appropriate measure assuming that there is no correlation between adjacent bins.

The classifier is run on two versions of the data, containing the same images at different resolutions. The high resolution data set is the original, however those images are too large to be processed efficiently by the FRAME implementation. The low resolution images have been reduced by a factor of 4 in each dimension. The results for the texton classifier on the high resolution dataset are presented in Table 4.3.

Tab. 4.3: Confusion matrix for the texton classifier.

		Assigned class				
		1	2	3	4	5
Actual class	1	151	0	0	0	0
	2	1	51	0	0	47
	3	27	0	72	0	0
	4	0	0	0	151	0
	5	0	0	0	0	107

This is an overall accuracy of 87.64 %. This texton based algorithm is an effective classifier for this data.

For the sake of comparison, the texton classifier and the FRAME classifier, described in the previous section, are evaluated on the low resolution data set. The texton classifier performs significantly worse in this experiment, achieving an accuracy of 66.92 %. Reducing the resolution of the images removes information which leads to this lower accuracy. Compared to that result, the FRAME classifier on the same dataset achieves an accuracy of 78.17 %. That is a significantly better result which implies that the FRAME models are effective at capturing the maximum available information about a texture image.

5. CONCLUSIONS AND RECOMMENDATIONS

This thesis is an investigation of the FRAME algorithm for modelling visual textures. The algorithm is presented in the context of other approaches that may be used to solve similar problems. Specifically, the tasks of synthesising and of classifying textures are addressed. A number of modifications to the original algorithm are implemented. This chapter summarises the findings related to those modifications and suggests a number of further refinements.

5.1 Recommendations

This work attempts to find ways to improve the FRAME algorithm. The improvements involve either reducing the amount of time taken to run the algorithm or improving the accuracy of the models that are produced.

The implementation of the FRAME algorithm described uses massively parallel code running on a GPU to improve performance. This code is responsible for calculating the effect of changing pixels within an image on the various filter responses. There remains a significant amount of calculation done by the CPU. It would be valuable to implement more of the algorithm in the GPU to further decrease the running time. In particular, the comparisons between histograms could be parallelised. These calculations are performed on large vectors (of about 400 elements) and could well be sped up by using a GPU.

The element of the algorithm which most limits the accuracy of its models is the filter bank. Features of the texture which cannot be easily captured by the filters in the bank cannot be modelled. The filters used in this implementation are the same as those in the original work. It would be meaningful to investigate the effect of using different types of filter banks.

The parallelisation of the algorithm relies on the assumption that all the filters are linear. Within this constraint there is room for many more filters that have not been explored in this work. The filter bank itself could be dynamic, with randomly generated filters being added. Such a process might generate new filter banks that are tuned to a specific class of textures and yield more accurate models in their respective scopes.

Many competing algorithms use SIFT features and local binary patterns for texture modelling. Since these have been proven to be useful for texture analysis, it would be interesting to incorporate them into the FRAME algorithm. The extraction of these features is not a linear operation so it would not be possible to take advantage of the parallel Gibbs sampler. The use of these filters could provide more accurate models in an application where execution time is less crucial.

5.2 Conclusion

Visual textures provide humans with much of their information about the world around them. Textures pose a specific problem for computer vision systems because they are difficult to describe mathematically — any pattern image may be considered a texture. Many approaches have been used in attempts to accurately model visual textures.

The FRAME algorithm creates a statistical model of textures. The model is made up of a number of filters, chosen to maximise entropy. For each filter, a λ function is calculated that leads to a calculation of the likelihood of a filter response belonging to the texture in question. These probabilities are combined to form the model of the visual texture. FRAME models can be easily interpreted, since the filters which they contain correspond to features within the pattern.

The algorithm is applied to two types of problems relating to visual textures: synthesis of the texture and classification of an image as belonging to one of a number a known texture classes. Texture synthesis is the process of learning a texture model and using that to create new images of the texture. The derivation of the FRAME model produces synthetic texture images. Those models can be used to recognise images of the same texture. The efficacy of FRAME models for texture classification is demonstrated with an experiment on a froth dataset. It is shown to be particularly useful with low resolution images. This is because the algorithm requires a large number of computations and it is very time consuming to calculate the model of a large image.

In order to reduce the computation time, a portion of the algorithm is implemented for a GPU. This alteration to the original algorithm is successful at reducing computation time. Specifically, it allows the complexity of the models to be improved while incurring only a small increase in computation time. The algorithm is parallelised by a modification to the Gibbs sampler. This requires that all the filters used be linear, which is the case in the original implementation.

A number of other modifications to the algorithm are implemented in an attempt to improve either the accuracy of the models or the convergence of the algorithm. A process similar to simulated annealing is added to alter the rate of change of the λ functions. This proves to be a useful method for decreasing the number of iterations required for convergence. Another experiment is done which forces the impulse filter to be used in each model. These tests show little or no improvement, implying that the intensity information can be suitably captured by other filters in the bank.

The most significant modification to the algorithm is the smoothing of the histograms. Most of the important variables in the algorithm are in the form of histograms. In the original implementation, these are discrete. The inaccuracies that arise from this discretisation impede the convergence. In the current implementation the histograms are smoothed by Gaussian kernels. The smoothing makes the histograms far less sensitive to small variations. Experiments are done to optimise the shape of these kernels and show that this is an effective improvement to the algorithm.

The models produced by FRAME can be used for texture classification. The effectiveness of such a classifier is tested on a dataset of froth images. In these tests, the classifier based on FRAME models is compared to an algorithm that uses texton dictionaries and an SVM classifier. Both approaches are shown to be able to discriminate between drastically different texture images. The froth images present a more challenging classification problem. In a test with low resolution images, for which the most accurate FRAME models were produced, the FRAME based classifier performed considerably better than the texton based classifier.

The FRAME algorithm is an interesting approach to modelling of visual textures. The models that it produces can be easily interpreted in terms of the pattern features that are represented. This dissertation demonstrates the application of the FRAME algorithm to both synthesis and classification tasks. The modifications to the algorithm presented here serve to make it more practically useful.

BIBLIOGRAPHY

[1] S.C. Zhu, XW Liu, and Ying Nian Wu. Exploring Texture Ensembles by Efficient Markov Chain Monte Carlo - Toward a "Trichromacy" Theory of Texture. *Pattern Analysis and Machine Intelligence*, 22(6):554–569, 2000.

[2] S.C. Zhu, Yingnian Wu, and D. Mumford. Filters, random fields and maximum entropy (FRAME): Towards a unified theory for texture modeling. *International Journal of Computer Vision*, 27(2):107–126, 1998.

[3] A. Materka and M. Strzelecki. Texture analysis methods - a review. Technical Report COST B11, Technical University of Lodz, Institue of Electronics, Brussels, 1998.

[4] Zhu Song-Chun and Guo Cheng-en. Conceptualization and modeling of visual patterns. In *Proceedings of 3rd International Workshop on Perceptual Organization in Computer Vision*, 2001.

[5] B. Julesz. Visual pattern discrimination. *Biological Cybernetics*, 20(3):151–180, 1976.

[6] J. Portilla and E. P. Simoncelli. A Parametric Texture Model Based on Joint Statistics of Complex Wavelet Coefficients. *International Journal of Computer Vision*, 40(1):49–71, 2000.

[7] T. Ojala, M. Pietikäinen, and D. Harwood. A comparative study of texture measures with classification based on featured distributions. *Pattern recognition*, 29(1):51–59, 1996.

[8] Joo-Hwee Lim, J.P. Chevallet, and Sheng Gao. Scene identification using discriminative patterns. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 2, pages 642 –645, 0-0 2006.

[9] L. W. Renninger and J Malik. When is scene identification just texture recognition? *Vision Research*, 44(19):2301 – 2311, 2004.

[10] E. Nowak, F. Jurie, and B. Triggs. Sampling strategies for bag-of-features image classification. *Computer VisionECCV 2006*, pages 490–503, 2006.

[11] Xiuwen Liu and DeLiang Wang. Texture classification using spectral histograms. *Image Processing, IEEE Transactions on*, 12(6):661 – 670, june 2003.

[12] D.G. Lowe. Object recognition from local scale-invariant features. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1150 –1157 vol.2, 1999.

[13] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(10):1615 –1630, oct. 2005.

[14] T. Ojala, M. Pietikäinen, and T. Maenpaa. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):971 –987, jul 2002.

[15] T. Ahonen, A. Hadid, and M. Pietikäinen. Face description with local binary patterns: Application to face recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(12):2037 –2041, dec. 2006.

[16] D. J. Heeger and J. R. Bergen. Pyramid-based texture analysis/synthesis. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 229–238, New York, NY, USA, 1995. ACM.

[17] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph.*, 20:127–150, July 2001.

[18] A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 341–346, New York, NY, USA, 2001. ACM.

[19] J. S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of the 24th annual conference on*

Computer graphics and interactive techniques, SIGGRAPH '97, pages 361–368, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[20] C. Han, E. Risser, R. Ramamoorthi, and E. Grinspun. Multiscale texture synthesis. *ACM Trans. Graph.*, 27:51:1–51:8, August 2008.

[21] S. Lefebvre and H. Hoppe. Parallel controllable texture synthesis. *ACM Trans. Graph.*, 24:777–786, July 2005.

[22] C.L. Phillips, J.M. Parr, and E.A. Riskin. *Signals, systems, and transforms.* Prentice Hall, 2003.

[23] R. M. Haralick and L. G. Shapiro. *Computer and Robot Vision*, volume 1. Addison-Wesley, 1992.

[24] W. J. Bolstad. *Introduction to Bayesian Statistics.* Wiley, Hoboken, New Jersey, second edition, 2007.

[25] C. P. Robert. *The Bayesian Choice.* Springer-Verlag, New York, 1994.

[26] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter. *Markov Chain Monte Carlo in Practice*, chapter Introducing Markov chain Monte Carlo, pages 1–19. Chapman & Hall/CRC, 1996.

[27] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50:5–43, 2003.

[28] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[29] D. J. Spiegelhalter, N. G. Best, W. R. Gilks, and H. Inskip. *Markov Chain Monte Carlo in Practice*, chapter Hepatitis B: a case study in MCMC methods, pages 21–43. Chapman & Hall/CRC, 1996.

[30] E. T. Jaynes. Information theory and statistical mechanics. *The Physical Review*, 106:620–630, May 1957.

[31] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):pp. 1065–1076, 1962.

[32] Anil K. J. and Farshid F. Unsupervised texture segmentation using gabor filters. *Pattern Recognition*, 24(12):1167 – 1186, 1991.

[33] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. Technical Report Version 3.2, NVIDIA, 2010.

[34] P. Brodatz. *Textures: A Photographic Album for Artists and Designers.* Peter Smith Publisher, Incorporated, 1981.

[35] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[36] Wikipedia. Annealing (metallurgy) — wikipedia, the free encyclopedia, 2012. [Online; accessed 26-January-2012].

[37] Song-Chun Zhu, Cheng-en Guo, Yizhou Wang, and Zijian Xu. What are textons? *International Journal of Computer Vision*, 62:121–143, 2005.

[38] M. Varma and A. Zisserman. A statistical approach to texture classification from single images. *International Journal of Computer Vision*, 62:61–81, 2005.

[39] T. Leung and J. Malik. Representing and recognizing the visual appearance of materials using three-dimensional textons. *International Journal of Computer Vision*, 43:29–44, 2001.