Biological and Medical Physics, Biomedical Engineering

Thomas Lindblad
Jason M. Kinser

Image Processing Using Pulse-Coupled Neural Networks

Applications in Python

Third Edition

Springer

Biological and Medical Physics,
Biomedical Engineering

For further volumes:
http://www.springer.com/series/3740

The fields of biological and medical physics and biomedical engineering are broad, multidisciplinary and dynamic. They lie at the crossroads of frontier research in physics, biology, chemistry, and medicine. The Biological and Medical Physics, Biomedical Engineering Series is intended to be comprehensive, covering a broad range of topics important to the study of the physical, chemical and biological sciences. Its goal is to provide scientists and engineers with textbooks, monographs, and reference works to address the growing need for information.

Books in the series emphasize established and emergent areas of science including molecular, membrane, and mathematical biophysics; photosynthetic energy harvesting and conversion; information processing; physical principles of genetics; sensory communications; automata networks, neural networks, and cellular automata. Equally important will be coverage of applied aspects of biological and medical physics and biomedical engineering such as molecular electronic components and devices, biosensors, medicine, imaging, physical principles of renewable energy production, advanced prostheses, and environmental control and engineering.

Thomas Lindblad · Jason M. Kinser

Image Processing Using Pulse-Coupled Neural Networks

Applications in Python

Third Edition

Springer

Thomas Lindblad
Department of Physics
Royal Institute of Technology (KTH)
Stockholm
Sweden

Jason M. Kinser
School of Physics and Computational Sciences
George Mason University
Fairfax, VI
USA

ISSN 1618-7210
ISBN 978-3-642-36876-9 ISBN 978-3-642-36877-6 (eBook)
DOI 10.1007/978-3-642-36877-6
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013935478

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Dedicated to
John L. Johnson
and
H. John Caulfield (1936–2012)

Preface to the Third Edition

This third edition has included two major components over the second edition. The first is that a selection of new applications has been addressed. There has been a recent surge in publications using the PCNN or ICM and a few of these have been included.

The second major change has been the inclusion of Python scripts. Over the past decade Python has emerged as a very powerful tool and its use is seen in many applications in the sciences. With the inclusion of a numeric library, Python has the ability to easily handle linear algebra operations with relatively few lines of code. With such efficiency it becomes possible to embed Python scripts in the text along with the theory and applications.

Every attempt has been made to ensure that the Python scripts are complete for applications that are demonstrated here. The scripts were written with Python 2.7 since it is still the standard in LINUX distributions. Users of Python 3.x will find minor differences which are customary when translating from 2.7.

Some readers who are experienced Python programmers will notice that the codes included here could be compressed to even fewer lines. However, the intent of including code is more educational in nature and so the scripts are designed to be readable before being highly compressed.

A website hosted at http://www.binf.gmu.edu/kinser/ will maintains a ZIP file with all of the Python scripts written by the authors. The Python system, the numeric Python (NumPy), scientific Python packages (SciPy), and the Python Image Library (PIL) can be obtained from their home sites as explained in Chap. 3. All of the scripts provided by the author are copyrighted and can be used only for academic purposes. Commercial applications without expressed written permission of the script's author is prohibited.

Stockholm and Manassas, 2012

Thomas Lindblad
Jason M. Kinser

Preface to the Second Edition

It was stated in the preface of the first edition of this book that image processing by electronic means has been a very active field for decades. This is certainly still true and the goal has been, and still is, to have a machine perform the same functions which humans do quite easily. In reaching this goal we have learned much about human mechanisms and how to apply this knowledge to image processing problems. Although there is still a long way to go, we have learned a lot during the last five or six years. This information and some ideas based upon it has been added to the second edition of this book.

The present edition includes the theory and application of two cortical models: the PCNN (the pulse coupled neural network) and the ICM (intersecting cortical model). These models are based upon biological models of the visual cortex and it is prudent to review the algorithms that strongly influenced the development of the PCNN and the ICM. The outline of the book is otherwise very much the same as in the first edition, although several new applications have been added.

In Chap. 7 a few of these applications will be reviewed including original ideas by co-workers and colleagues. Special thanks are due to Soonil D. D. V. Rughooputh, the dean of the Faculty of Science at the University of Mauritius and Harry C. S. Rughooputh, the dean of the Faculty of Engineering at the University of Mauritius.

We should also like to acknowledge that Guisong Wang, a doctoral candidate in the School of Computational Sciences at GMU, made a significant contribution to Chap. 5.

We would also like to acknowledge the work of several diploma and Ph.D. students at KTH, in particular Jenny Atmer, Nils Zetterlund, and Ulf Ekblad.

Stockholm and Manassas, April 2005

Thomas Lindblad
Jason M. Kinser

Preface to the First Edition

Image processing by electronic means has been a very active field for decades. The goal has been, and still is, to have a machine perform the same image functions which humans do quite easily. This goal is still far from being reached. So we must learn more about the human mechanisms and how to apply this knowledge to image processing problems. Traditionally, the activities in the brain are assumed to take place through the aggregate action of billions of simple processing elements referred to as neurons and connected by complex systems of synapses. Within the concepts of artificial neural networks, the neurons are generally simple devices performing summing, thresholding, etc. However, we show now that the biological neurons are fairly complex and perform much more sophisticated calculations than their artificial counterparts. The neurons are also very specialized and it is thought that there are several hundred types in the brain and messages travel from one neuron to another as pulses.

Recently, scientists have begun to understand the visual cortex of small mammals. This understanding has led to the creation of new algorithms that are achieving new levels of sophistication in electronic image processing. With the advent of such biologically inspired approaches, in particular with respect to neural networks, we have taken another step towards the aforementioned goal.

In our presentation of the visual cortical models we will use the term Pulse-Coupled Neural Network (PCNN). The PCNN is a neural network algorithm that produces a series of binary pulse images when stimulated with a gray scale or color image. This network is different from what we generally mean by artificial neural networks in the sense that it does not train. The goal for image processing is to eventually reach a decision on the content of that image. These decisions are generally far easier to accomplish by examining the pulse outputs of the PCNN rather than the original image. Thus, the PCNN becomes a very useful pre-processing tool. There exists, however, an argument that the PCNN is more than a pre-processor. It is possible that the PCNN also has self-organizing abilities which make it possible to use the PCNN as an associative memory. This is unusual for an algorithm that does not train.

Finally, it should be noted that the PCNN is quite feasible to implement in specialized hardware. Traditional neural networks have had a large fan-in and fan-out. In other words, each neuron was connected to several other neurons. In electronics a

different "wire" is needed to make each connection and large networks are quite difficult to build. The PCNN, on the other hand, has only local connections and in most cases these are always positive. This is quite plausible for electronic implementation.

The PCNN is quite powerful and we are just beginning to explore the possibilities. This text will review the theory and then explore its known image processing applications: segmentation, edge extraction, texture extraction, object identification, object isolation, motion processing, foveation, noise suppression, and image fusion. This text will also introduce arguments as to its ability to process logical arguments and its use as a synergetic computer. Hardware realization of the PCNN will also be presented.

This text is intended for the individual who is familiar with image processing terms and has a basic understanding of previous image processing techniques. It does not require the reader to have an extensive background in these areas. Furthermore, the PCNN is not extremely complicated mathematically so it does not require extensive mathematical skills. However, this text will use Fourier image processing techniques and a working understanding of this field will be helpful in some areas.

The PCNN is fundamentally unique from many of the standard techniques being used today. Many of these fields have the same basic mathematical foundation and the PCNN deviates from this path. It is an exciting field that shows tremendous promise.

Stockholm and Manassas, 1997

Thomas Lindblad
Jason M. Kinser

Acknowledgments

The work reported in this book includes research carried out by the authors together with co-workers at various universities and research establishments. Several research councils, foundations, and agencies have supported the work and made the collaboration possible. Their support is gratefully acknowledged. In particular, we would like to acknowledge the fruitful collaboration and discussions with the following scientists: Kenneth Agehed, Randy Broussard, Åge J. Eide, John Caulfield, Bruce Denby, W. Friday, John L. Johnson, Clark S. Lindsey, Steven Rogers, Thaddeus Roppel, Manuel Samuelides, Åke Steen, Géza Székely, Mary Lou Padgett, and Ilya Rybak. The authors would also like to extend their gratitude to Stefan Rydström for his invaluable editing.

Contents

Python Codes

Chapter 1
Biological Models

Humans have an outstanding ability to recognise, classify and discriminate objects with extreme ease. For example, if a person was in a large classroom and was asked to find the light switch it would not take more than a second or two. Even if the light switch was located in a different place than the person expected or it was shaped differently than expected it would not be difficult to find the switch. Humans also do not need to see hundreds of exemplars in order to identify similar objects. A person needs to see only a few dogs and then he is able to recognise dogs even from species that he has not seen before. This recognition ability also holds true for animals, to a greater or lesser extent. A spider has no problem recognising a fly as even a baby spider can do that. At this level we are talking about a few hundred to a thousand processing elements or neurons. Nevertheless the biological systems seem to do their job very well.

Computers, on the other hand, have a very difficult time with these tasks. Von Neumann machines need a large amount of memory and significant speed to even come close to the processing time of a human. Furthermore, the software for such simple general tasks does not exist. There are special problems where the machine can perform specific functions well, but the machines do not perform general image processing and recognition tasks to the extent that animals and humans do.

Implementations of neural systems in silicon hardware have been tried by companies Intel and IBM. The Electrically Trainable Neural Network (ETANN) chip [43] from Intel had 128 neurons[1] and the first Zero Instruction Set Computer (ZISC36) chip[1] from IBM had 36 neurons. However, these are all "mathematical neurons" based on the back-propagation algorithm [14] and the radial basis function algorithms [73] and really not any implementation of biological systems. The ZISC36 chip could easily be put in parallel [66] to make use of several hundreds of neurons. It has also been further developed and is today available as C1MK with a thousand neurons [21]. This chip is between a fly and a worm with respect to the number of interconnections, although a bit "faster" than both. However, there is still a long way to go before reaching small mammals and humans as illustrated in

[1] There are several examples of how 128 neurons can be used with one example show in [65].

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_1, © Springer-Verlag Berlin Heidelberg 2013

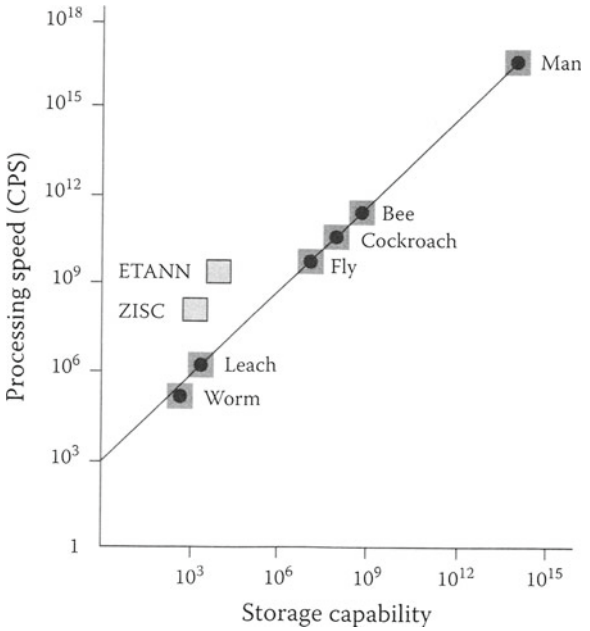


Fig. 1.1 Comparing "brains" of some animals in some neural networks systems as discussed in the text

Fig. 1.1. Recent work at Manchester University is to develop SpiNNaker (Spiking Neural Network Architecture) which is a parallel computer specifically designed to model large scale spiking neural networks. The design will allow each computer to contain more than one million cores. Completion of the first machine is expected by the end of 2013. The ETANN, ZISC and a few other neural network chips are shown in Fig. 1.2.

It is often claimed that neuromorphic machines outperform von Neumann machines at certain environmental complexity (e.g., input combinatorics). There is a "break even point" and after this point the machine complexity (e.g., size, power, memory, gates, synapses) increases steeply for von Neumann computers but not so much for the neural architectures. However, there are still many orders of magnitude of complexity before one reaches the "human" level of performance. Besides the above problem of the number of neurons, there are at least two other fundamental items to consider: the neurons designed for specific tasks and the intelligent mammal sensors. This is particularly true for the visual system. The neurons get auxiliary information from adjacent neurons, the information is sent in separate paths in the mid-brain, backward signals are used to prioritise important information. Using computer language one would say that the feature extraction system, its redundancy and the parallel triggering system in the mid-brain ensure that important information reaches the visual cortex. At the same time, there is a tremendous reduction in data volume from perhaps initially 10^8 neurons and a bit-rate of 50 Mbits/s to approximate video speed when we become aware of what we are seeing.

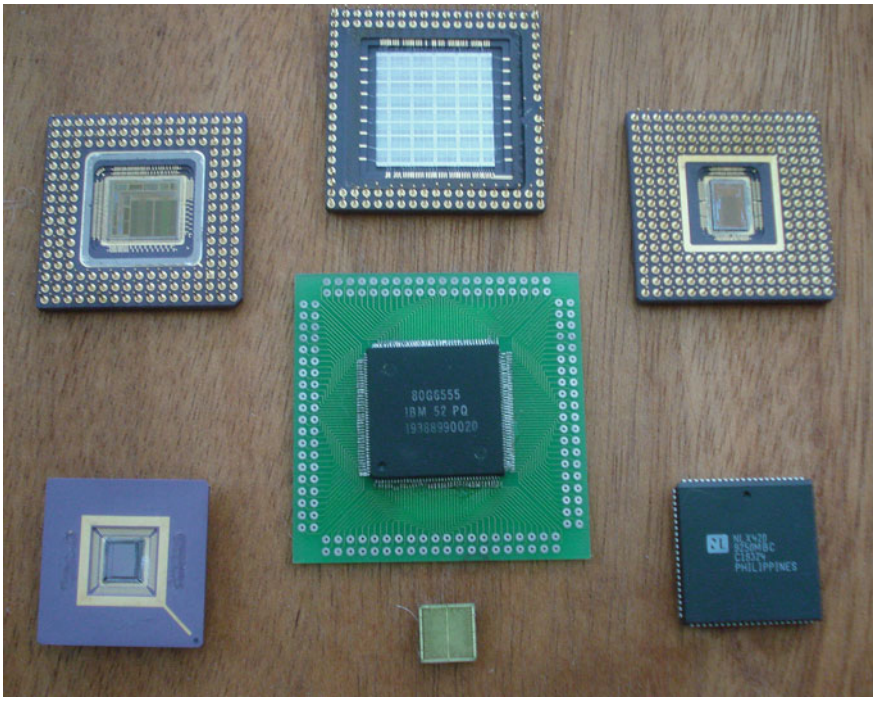


Fig. 1.2 A few neural network chips. The ETANN (*top right*) and ZISC036 (*middle* and *lower middle*) are mentioned in the text and plotted in Fig. 1.1

1.1 Introduction

One of the processes occurs in the visual cortex, which is the part of the brain that receives information from the eye. At this point in the system the eye has already processed and significantly changed the image. The visual cortex converts the resultant eye image into a stream of pulses. Synthetic models of this portion of the brain for small mammals have been developed and successfully applied to many image processing applications.

The mammalian visual system is considerably more elaborate than simply processing an input image with a set of inner products. Many operations are performed before decisions are reached as to the content of the image. Neuro-science does not yet understand all of processes. However, sometimes the visual system is fooled, in particular where we expect colour and shades to follow some rules and patterns. There are very many examples of this, (e.g. the shadow of a cylinder on a board of chess shown in Fig. 1.3). Most people would say that the grey scale of square “A” is darker than that of square “B”, but even the simplest Paint program of a von Neumann computer would say that they are exactly the same.

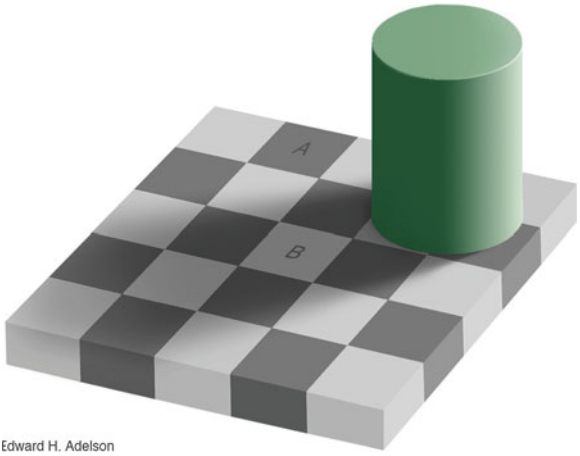


Fig. 1.3 The shadow of a cylinder on a checkerboard. Is square **a** really darker than square **b**?

Another example is the case of hiding a person from a searching adversary. Hiding in an open field may offer advantages over hiding in a ditch in that such a place is unexpected and away from visual edges which are natural attractors in human vision (see Sect. 5.4). "What you see is not always the truth." An excellent example is the awareness test [4] in which the viewer is asked to count the number of passes of a ball between a set of players wearing a particular jersey. In this video a dancer dressed as a bear moves across the frame of view and most viewers completely miss his presence. 80 % or more of our (university) students did not see the bear. Human processing of information is clearly not based solely on the visual input but also highly affected by other processes in the brain.

This chapter will mention a few of the important operations to provide a glimpse of the complexity of the processes. It soon becomes clear that the mammalian system is far more complicated than the usual computer algorithms used in image recognition. It is almost silly to assume that such simple operations can match the performance of the biological system. Of course, image input is performed through the eyes. Receptors within the retina at the back of the eye are not evenly distributed nor are they all sensitive to the same optical information. Some receptors are more sensitive to motion, colour, or intensity. Furthermore, the receptors are interconnected. When one receptor receives optical information it alters the behaviour of other surrounding receptors. A mathematical operation is thus performed on the image before it even leaves the eye. The eye also receives feedback information. We humans do not stare at images, we foveate. Our centre of attention moves about portions of the image as we gather clues as to the content. Furthermore, feedback information also alters the output of the receptors.

After the image information leaves the eye it is received by the visual cortex. Here the information is further analysed by the brain. The investigation of the visual cortex of the cat [26] and the guinea pig [93] have been the foundation of the digital models

used in this text. Although these models are a big step in emulating the mammalian visual system, they are still very simplified models of a very complicated system. Intensive research continues to understand fully the processing. However, much can still be implemented or applied already today.

1.2 Biological Foundation

While there are discussions as to the actual cortex mechanisms, the products of these discussions are quite useful and applicable to many fields. In other words, the algorithms being presented as cortical models are quite useful regardless of their accuracy in modelling the cortex. Following this brief introduction to the primate cortical system, the rest of this book will be concerned with applying cortical models and not with the actual mechanisms of the visual cortex.

In spite of its enormous complexity, two basic hierarchical pathways can model the visual cortex system: the pavocellular one and the mangnocellular one, processing (mainly) colour information and form/motion, respectively. Figure 1.4 shows a model of these two pathways. The retina has luminance and colour detectors which interpret images and pre-process them before conveying the information to the visual cortex. The lateral geniculate nucleus, LGN, separates the image into components that include luminance, contrast, frequency, etc. before information is sent to the visual cortex (labelled V in Fig. 1.4).

The cortical visual areas are labelled V1–V5 in Fig. 1.4. V1 represents the striate visual cortex and is believed to contain the most detailed and least processed image.

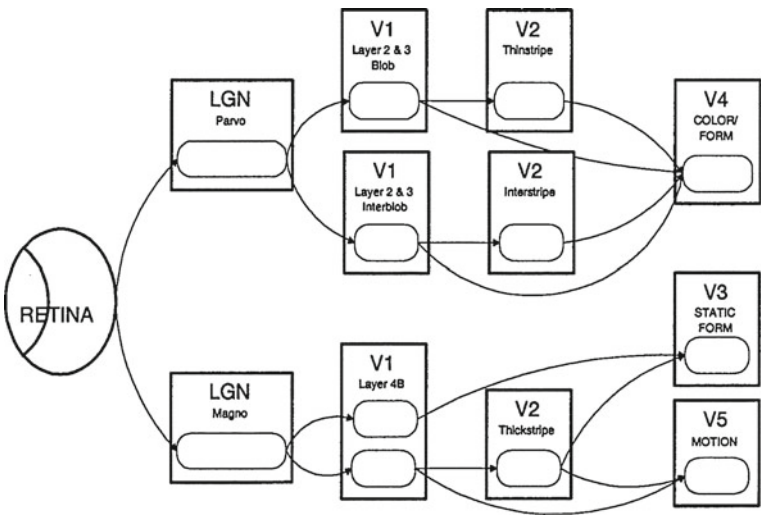


Fig. 1.4 A model of the visual system. The abbreviations are explained in the text. Only feedforward signals are shown

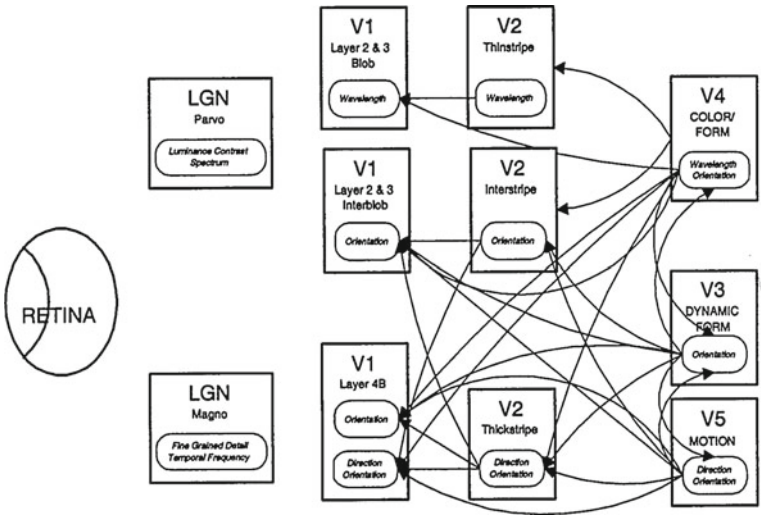


Fig. 1.5 Continuation of the model with the reverse connections displayed

Area V2 contains a visual map that is less detailed and pre-processed than area V1. Areas V3–V5 can be viewed as speciality areas and process only selective information such as, colour/form, static form and motion, respectively. Information between the areas flows in both directions, although only the feedforward signals are shown in Fig. 1.4. The processing area spanned by each neuron increases as you move to the right in Fig. 1.4, (i.e., a single neuron in V3 processes a larger part of the input image than a single neuron in V1). The re-entrant connections from the visual areas are not restricted to the areas that supply its input. It is suggested that this may resolve conflict between areas that have the same input but different capabilities.

The backward connections shown in Fig. 1.5 are not restricted to the areas from which the feedforward signals came. Indeed, the most probable reason for this is that the signals are used to resolve conflicts between areas and to set priorities on the most important paths and processings [120, 121].

Much is to be learned from how the visual cortex processes information, adapts to both the actual and feedback information for intelligent processing. However, a smart sensor will probably never look like the visual cortex system, but only use a few of its basic features.

1.3 Hodgkin-Huxley

Research into mammalian cortical models received its first major thrust about half a century ago with the work of Hodgkin and Huxley [39]. Their model is a mathematical (set of nonlinear ordinary differential equations) that describes how action

potentials in neurons are initiated and propagated. The basic components of the model are current sources, conductances and batteries. These are used in the model to represent the biophysical characteristic of cell membranes. Their system describes the membrane potentials (E) using the equation:

$$I = m^3 h G_{Na} (E - E_{Na}) + n^4 + G_K (E - E_K) + G_L (E - E_L), \tag{1.1}$$

where I is the ionic current across the membrane, m is the probability that an open channel has been produced, G is conductance (for sodium, potassium, and leakage), E is the total potential and a subscripted E is the potential for the different constituents. The probability term was described by,

$$\frac{dm}{dt} = a_m(1 - m) - b_m m, \tag{1.2}$$

where a_m is the rate for a particle not opening a gate and b_m is the rate for activating a gate. Both a_m and b_m are dependent upon E and have different forms for sodium and potassium.

The importance to cortical modelling is that the neurons are now described as a differential equation. The current is dependent upon the rate changes of the different chemical elements. Neuron activity is represented as oscillatory and dynamic processes. Although the original Hodgkin-Huxley model is regarded as one of the great achievements in biophysics, modern Hodgkin-Huxley-type models have been extended to include additional ion channel populations as well as highly complex dendrite and axon structures.

1.4 Fitzhugh-Nagumo

A mathematical advance published a few years later has become known as the Fitzhugh-Nagumo model [29, 75] in which the neuron behaviour is described as a van der Pol oscillator. This model is described in many forms but each form is essentially the same as it describes a coupled oscillator for each neuron. One example [61] describes the interaction of an excitation x and y using the expressions,

$$\epsilon \frac{dx}{dt} = -y - g(x) - I, \tag{1.3}$$

and

$$\frac{dy}{dt} = x - by, \tag{1.4}$$

where $g(x) = x(x - a)(x - 1), 0 < a < 1, I$ is the input current, and $\epsilon \ll 1$. This coupled oscillator model will be the foundation of the many models that would follow.

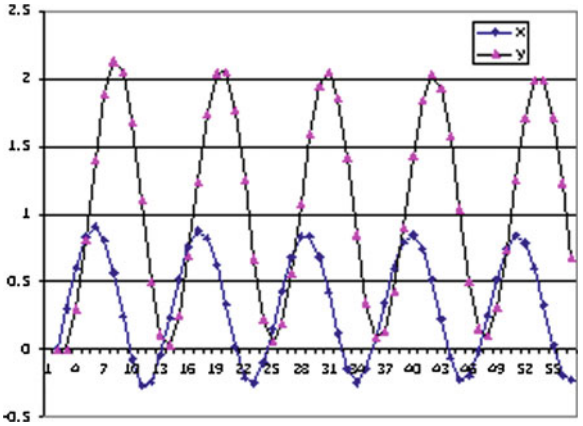


Fig. 1.6 An oscillatory system described through the Fitzhugh-Nagumo equations with the x-axis representing time and the y-axis representing excitations of x and y

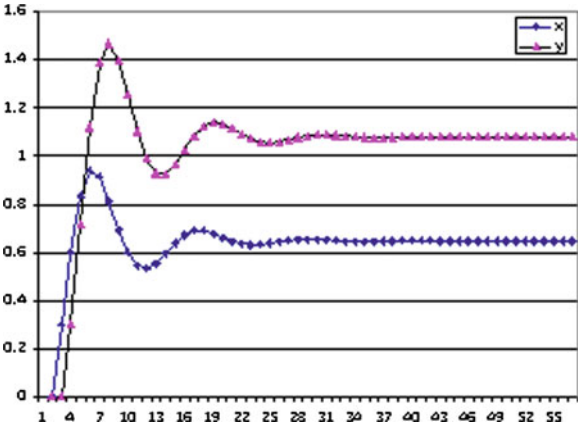


Fig. 1.7 A steady state system described through the Fitzhugh-Nagumo equations with the x-axis representing time and the y-axis representing excitations of x and y

These equations describe a simple coupled system and very simple simulations can present different characteristics of the system. By using ($\epsilon = 0.3$, $a = 0.3$, $b = 0.3$, and $I = 1$) it is possible to get an oscillatory behaviour as shown in Fig. 1.6. By changing a parameter such as b it is possible to generate different types of behaviour. For example, setting $b = 0.6$ will create a steady state which is shown in Fig. 1.7 where both x and y reach a constant value.

The importance of the Fitzhugh-Nagumo system is that it describes the neurons in a manner that will be repeated in many different biological models. Each neuron is two coupled oscillators that are connected to other neurons.

1.5 Eckhorn Model

In 1989, Eckhorn et al. [26] introduced a neural model to emulate the cat visual cortex. Shortly, thereafter, Johnson [46] extrapolated the model to a digital form creating the Pulse-Coupled Neural Network (PCNN). This was the seminal work in transferring the cortical model into the field of digital image processing and recognition. Since then the PCNN has been expanded into a variety of applications in image processing, image segmentation, feature generation, face extraction, motion detection, region growing, noise reduction and image signatures just to name a few. Several of these applications will be discussed in the subsequent chapters.

One of the innovations that the Eckhorn model brought to image processing was a system that relied solely on local connections. Figure 1.8 depicts a neuron which contains two types of input (linking and feeding) which are combined to create the neuron's potential or membrane voltage U_m. This potential is then compared to a dynamic threshold Θ to produce the neuron's output.

The Eckhorn model is expressed by the following equations,

$$U_{m,k}(t) = F_k(t)\left[1 + L_k(t)\right], \tag{1.5}$$

$$F_k(t) = \sum_{i=1}^{N}\left[w_{ki}^{f} Y_i(t) + S_k(t) + N_k(t)\right] \otimes I(V^a, \tau^a, t), \tag{1.6}$$

$$L_k(t) = \sum_{i=1}^{N}\left[w_{ki}^{l} Y_i(t) + N_k(t)\right] \otimes I(V^l, \tau^l, t), \tag{1.7}$$

$$Y_k(t) = \begin{cases} 1 & U_{m,k}(t) > \Theta_k(t) \\ 0 & Otherwise \end{cases}, \tag{1.8}$$

where, in general

$$X(t) = Z(t) \otimes I(v, \tau, t), \tag{1.9}$$

is

$$X[n] = X[n-1]e^{-t/\tau} + VZ[n]. \tag{1.10}$$

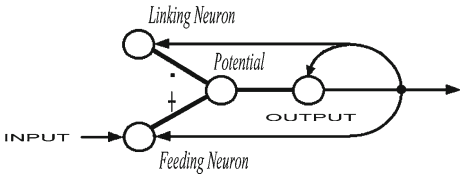


Fig. 1.8 The Eckhorn-type neuron

Here N is the number of neurons, w is the synaptic weights, Y is the binary outputs, and S is the external stimulus. Typical value ranges are $\tau_a = [10, 15]$, $\tau_l = [0.1, 1.0]$, $\tau_s = [5, 7]$, $V_a = 0.5$, $V_l = [5, 30]$, $V_s = [50, 70]$, and $\Theta_o = [0.5, 1.8]$. This model represents neural activity as coupled oscillators with two diffusion terms. Furthermore, there is now a second order (1.5) and a non-linear term (1.8) involved.

1.6 Rybak Model

Independently, Rybak [93] studied the visual cortex of the guinea pig and found similar neural interactions. While Rybak's equations differ from Eckhorn's the behaviour of the neurons is quite similar. Rybak's neuron has two compartments X and Z. These interact with the stimulus, S, as,

$$X_{ij}^S = F^S \otimes ||S_{i,j}||, \tag{1.11}$$

$$X_{ij}^I = F^I \otimes ||Z_{ij}||, \tag{1.12}$$

and,

$$Z_{ij} = f\left\{\sum X_{ij}^S - \left(\frac{1}{\tau p + 1}\right) X_{ij}^I - h\right\}, \tag{1.13}$$

where F^S are local On-Centre/Off-Surround connections, F^I are local directional connections, τ is the time constant and h is a global inhibitor. In the cortex there are several such networks which work on the input at differing resolutions and differing F^I. The non-linear threshold function is denoted by $f\{\}$.

The neural connections are quite different from the Eckhorn model in which the strength of the neural connections was inversely proportional to the physical distance between the neurons. In image processing terms this is a smoothing operation which would tend to blur the image. Rybak's model uses on-centre/off-centre connections in which there are positive connections neighbouring neurons and negative connections between neurons that are physically farther apart. In image processing terms this type of connections would enhance edges. This is very different from the Eckhorn model in which the smoothing operation would destroy edge information.

1.7 Parodi Model

There is still great disagreement as to the exact model of the visual cortex. Parodi [80] presented alternatives to the Eckhorn model. The arguments against the Eckhorn model included the lack of synchronisation of neural firings, the undesired similar

outputs for both moving and stationary targets and that neural modulations in the linking fields were measured considerably higher than the Eckhorn model allowed.

Parodi presented an alternative model, which included delays along the synaptic connections and would require that the neurons be occasionally reset *en masse*. Parodi's system followed these equations,

$$\frac{\partial V(x,y,t)}{\partial t} = -\frac{V(x,y,t)}{\tau} = D\nabla^2 V(x,y,t) + h(x,y,t), \tag{1.14}$$

where V_i is the potential for the ith neuron, D is the diffusion ($D = a^2/CR_c$), R_c is the neural coupling resistance, $t = CR_l$, R_l is the leakage resistance, and $R_c^{-1} < R_l^{-1}$, and

$$h_i(t) = \sum_j w_{ij}\delta(t - t^s - \tau_{ij}). \tag{1.15}$$

1.8 Summary

This book will focus on two digital models the PCNN and the ICM and their applications in the rest of the chapters. The PCNN is based on the Eckhorn model with the only modification being to unify the communication times between neurons to a discrete unit of time. The Eckhorn model is a set of coupled differential equations describing a multi-faceted neuron. This model has its foundation in earlier models of Hodgkin-Huxley and Fitzhugh-Nagumo. The Eckhorn model is not the only visual cortex model that exists and two others (Rybak and Parodi) have also been reviewed but this is not an exhaustive list of the proposed models.

The models of Eckhorn, Rybak and Parodi do have common mathematical foundations which are in congruence with the works from the previous decades. The ICM was developed in an attempt to capture common elements of these biological models and to reduce the components that are unique to each. Unlike the PCNN the ICM is a purely mathematical conjecture and not an attempt to replicate the visual cortex. Instead its purpose is to create a useful image processing engine. Both the PCNN and ICM will be reviewed in detail in Chap. 4. The ensuing chapters apply these two models to a variety of applications.

In several applications Python scripts are provided so that the reader can replicate the results and then pursue other avenues of related research. Before the PCNN and ICM are reviewed two chapters are presented to familiarize the reader with the Python language in support of the scripts that are provided throughout the rest of the chapters.

Chapter 2
Programming in Python

Implementation of digital neural models is straightforward and can be easily realised in modern scripting languages. For this text, examples will be presented with accompanying Python scripts so that readers can replicate the presented results. Furthermore, readers will also gain a small set of Python tools for the PCNN and ICM. There are several scripting languages from which to choose, but Python was selected since it exhibits several concurrent advantages:

- Free,
- Language-like scripting,
- Matrix/vectors operators (using NumPy),
- Easy access to images (using PIL),
- Platform independent, and
- Popularity gaining ground in many science fields.

This chapter introduces readers to simple Python scripting that will be used in other chapters. This is not a comprehensive Python instruction manual and readers that are familiar with Python scripting may wish to bypass this chapter. The provided scripts may not be the most efficient scripting in that some processes which could be performed in a single line are presented in a few lines. The reason for this is that the scripts are intended to be viewed by readers with a variety of skill levels and the scripts need to be easily understood rather than achieving the utmost compactness.

2.1 Environment

In its native form Python is run in a command line window or shell. However, there are several other user interfaces that provide more convenience and are easier to use. Python installations for PC users come with the IDLE environment which provides a useful editor and many shortcut keys. Even though this is one of the simpler environments it is an excellent environment to use for the projects in this

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_2, © Springer-Verlag Berlin Heidelberg 2013

book. There are many other development environments that are available to use [2] that are not used here.

This text will use examples in Python 2.7 (completely compatible with 2.6) and will not present code for Python 3.x. At the time of writing the necessary tools used here are not completely available for 3.x. Furthermore, current installations of LINUX still use 2.7.

2.1.1 Command Interface

When Python is started, a command line interface is created with a prompt preceded by `>>>`. Code 2.1 shows simple commands to perform basic mathematical operations.

Code 2.1 Python performing simple calculations.

```
>>> 5 + 6
11
>>> 4 - 1
3
>>> 8 * 4
24
>>> 8/4
2
```

2.1.2 IDLE

The IDLE interface provides a colour coded environment that also comes with an editor and a debugger. The IDLE interface also has hotkeys that make editing quick. For example the Ctl-p and Ctl-n keys retype previous and subsequent commands at the command line. When the user types a letter (or letters) and then uses Ctl-p the previous commands starting with those letters are placed at the command line. The IDLE environment also works like an editor so it is possible to copy, cut, paste, and save the text that is in the command window. Thus, during code development it is easy to copy commands to and from the editor.

2.1.3 Establishing a Working Environment

The IDLE environment has a major inconvenience in that it starts up in the `C:\python27` directory (for MS-Windows). This means that if the user saves a file then it will be stored in this directory. The problem is that vital Python files are

also stored there. At best the user mixes their files with the Python files and at worst the user can erase vital files. IDLE users should immediately change to a working directory so that their files will not be confused with Python files. Users should also create a directory that will contain their Python code files that is separate from directories that contain data files.

Code 2.2 shows an example of the first steps that should be used in order to establish a proper working environment. In this case, prior to starting Python, the user created a directory named `C:\\MyWork` and a subdirectory that will contain the Python codes named `C:\\MyWork\\pysrc`. The first command loads two modules that are used to establish the environment (more on modules in Sect. 2.9). The second command moves the current working directory to the new directory, and the third command sets up the path so that Python will look in the user's directory when searching for Python files.

Code 2.2 Setting up the Python environment for IDLE users. Directory names will be unique for each user.

```
>>> import os, sys
>>> os.chdir("C:/MyDir" )
>>> sys.path.append( "pysrc" )
```

The second command uses a forward slash to separate the C drive from the directory. Either a single forward slash (`/`) or two backslashes (`\\`) can be used to separate directory names. The use of a single backslash will cause problems as there are ASCII codes such as `\n` which are used to represent a new line character. So, if the user uses a single backslash for a directory that starts with the letter `n` then it will be interpreted as a new line character and not the name of the directory.

2.2 Data Types and Simple Math

As seen in Code 2.1 Python can be used as a simple calculator. The commands are similar to those of other languages. Python has data types similar to other languages such as integer, float, and double. It also has a complex data type with the imaginary part being represented by `j` as in `5 + 7j`. There are other data types available but for computations the integer, float, and complex numbers will be the only ones used here.

The modulus operator is denoted by the percent sign and the `divmod` function performs by the integer division and modulus function as seen in Code 2.3. When an integer is divided by another integer the result will be an integer even if there is a remainder.

Code 2.3 Division in Python.

```
>>> 9/2 # integer division
4
>>> 9.0/2 # floating point division
4.5
>>> 9 \% 2 # modulus
1
>>> divmod( 9,2 )
(4, 1)
```

The conversion of an integer to a floating point number can be accomplished by a type conversion or mathematics involving a floating point number. A couple of conversions are shown in Lines 1 and 2 of Code 2.4. The results of computations return the result as whichever data type was superior in the computation. In other words, if a float and a complex number are added then the result will be a complex number. To convert data to an inferior type a typecasting command is used as shown in Line 3. Of course, using Python solely as a calculator misses almost all of its potential.

Code 2.4 Conversion of integers to floats.

```
>>> a = float( 5 )
>>> b = 5 + 0.0
>>> c = int( 5.6 )
```

2.3 Tuples, Lists, and Dictionaries

There are data collections which are unique to Python that are extremely useful. These are tuples, lists, sets, and dictionaries. These are tools that can collect data of a variety of data types to keep it organized. The closest equivalent in the C language is a `struct`.

2.3.1 Tuples

A tuple is a collection of data of any type and is denoted by parenthesis. In Line 8 of Code 2.3 the `divmod` function returned two integers inside of a tuple. The data inside of a tuple does not have to be of the same type. Code 2.5 shows the creation of a tuple

with four elements each of a different data type. The next two commands demonstrate the retrieval of elements from the tuple. There are more advanced methods of data retrieval in Sect. 2.4.

Code 2.5 A simple tuple demonstration.

```
>>> mytuple = ( 4, 5.6, 'a string', 7+9j )
>>> mytuple[0]
4
>>> mytuple[1]
5.6
```

A tuple can contain any type of data including other tuples, sets, lists, dictionaries, objects, images, etc. For image processing, tuples are quite useful for containing coordinates or keeping the data organized. As an example, if a program used several images and also needed to maintain their file names then it would be prudent to create a tuple for each image that contained the file name and the image data.

2.3.2 Lists

A list is similar to a tuple in that it can contain multiple types of data, but a list can be altered. Code 2.6 shows the creation of a list which is delineated with square brackets instead of parenthesis. The `append` function adds an element onto the end of the list. The `insert` function places an element in the specified location. In Line 7, `mytuple` was inserted into the list. Thus the `mylist[1]` returned the entire tuple. The command `mylist[1][2]` returned the string that was inside of the tuple.

Code 2.6 A simple list demonstration.

```
>>> mylist = [4, 'another string']
>>> mylist[1]
'another string'
>>> mylist.append( -12 )
>>> mylist
[4, 'another string', -12]
>>> mylist.insert(1, mytuple )
>>> mylist
[4, (4, 5.6, 'a string', (7+9j)), 'another string', -12]
>>> mylist[1]
(4, 5.6, 'a string', (7+9j))
>>> mylist[1][2]
'a string'
```

Information can also be removed from a list. The function `remove` (Line 3 in Code 2.7) deletes the first occurrence of a data entry from the list. The function `pop` (Line 6 in Code 2.7) deletes data according to the index number. The `remove(4)` deletes the first occurrence of the integer 4. The `pop(2)` will remove the item that is at `mylist[2]` and put it into the variable `rcv`.

Code 2.7 The use of remove and pop.

```
>>> mylist
[4, (4, 5.6, 'a string', (7+9j)), 'another string', -12]
>>> mylist.remove( 4 )
>>> mylist
[(4, 5.6, 'a string', (7+9j)), 'another string', -12]
>>> rcv = mylist.pop( 2 )
>>> mylist
[(4, 5.6, 'a string', (7+9j)), 'another string']
```

Other commands that can be used with a list is to sort the data (`mylist.sort()`), count the number of times that a particular data *d* occurs (`mylist.count( d )`), and reverse the order of the elements in the list (`mylist.reverse()`).

2.3.3 Dictionaries

A list can contain any type of items and in any arrangement. However, for large projects searching a list can be slow. A much faster tool to employ is a *dictionary*. A dictionary is comparable to a hash used in other languages.

Each entry in a dictionary is comprised of two components: the *key* and the *data*. The key can be of any data type and is the item that is searched while the data, which can also be of any data type, is the item that is retrieved. A dictionary is delineated by curly braces as in Line 1 of Code 2.8. Lines 2–4 create entries into the dictionary and demonstrate that both the key and data can be composed of differing data types.

There are many different functions that accompany a dictionary but a few are commonly used here. In Code 2.9 the function `keys` returns a list of all of the keys that are in the dictionary. The function `has_key` returns a boolean value indicating if the dictionary contains a particular key.

The dictionary may rearrange the order in which data is stored and thus it is not feasible to retrieve the first item from the dictionary as in the `pop` function for a list.

Code 2.8 A simple dictionary example.

```
>>> dct = { } # empty dictionary
>>> dct[0] = 'more string'
>>> dct['a'] = [5, 'string again']
>>> dct[5.6] = 7.5
>>>
>>> dct[0]
'more string'
>>> dct[5.6]
7.5
```

Code 2.9 The key commands.

```
>>> dct.keys()
[0, 'a', 5.6]
>>> dct.has_key( 1 )
False
```

2.4 Slicing

The examples used in tuples and lists access data according to an index. Python is similar to C or Java in that the indexing starts with 0. Code 2.10 constructs a simple list and accesses the first and second items. A negative index accesses items from the end of the list and thus a−1 retrieves that last item.

Code 2.10 Simple indices.

```
>>> abet = ['a','b','c','d','e','f','g','h','i','j']
>>> abet[0]
'a'
>>> abet[1]
'b'
>>> abet[-1]
'j'
>>> abet[-2]
'i'
```

Multiple items can be retrieved by indicating the first and last index to be accessed. In Python the first index is included and the second index is excluded. This is shown in Code 2.11 where Line 1 retrieves items 1, 2, 3, 4, and 5 but not 6—it is excluded. Lines 3 and 5 show that the if the first or last index are omitted then it automatically assumes the beginning or the end of the list is used. Line 7 uses a negative index to retrieve the last four items of the list.

Code 2.11 Simple slicing.

```
>>> abet[1:6]
['b', 'c', 'd', 'e', 'f']
>>> abet[:6]
['a', 'b', 'c', 'd', 'e', 'f']
>>> abet[6:]
['g', 'h', 'i', 'j']
>>> abet[-4:]
['g', 'h', 'i', 'j']
```

The examples in Code 2.12 use a third integer to indicate that indexes can be skipped. In Line 1 every other element beginning with 0 and ending with 10 is accessed. Line 3 is the same except the first and last index are not needed. Line 5 uses the entire list but retrieves the items with a step of −1. In other words, the items are retrieved in reverse order.

Code 2.12 Slicing in steps.

```
>>> abet[0:10:2]
['a', 'c', 'e', 'g', 'i']
>>> abet[::2]
['a', 'c', 'e', 'g', 'i']
>>> abet[::-1]
['j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```

These slicing techniques are used for strings, lists, tuples, sets, and arrays. However, in the case of arrays, slicing can even be more advanced as shown in Sect. 3.1.9.

2.5 Strings

Strings are quite easy to manipulate in Python. Unlike other languages Python does not differentiate between a character and a string. Strings are created by surrounding text with either a set of single quotes or a set of double quotes. There is no difference between the two but they can not be mixed. Code 2.13 shows the creation of a few strings and the concatenation of two strings using the plus sign (Line 3).

Characters can be retrieved from a string through slicing, but the characters in a string can not be altered. Code 2.14 shows that it is possible to extract characters using slices but not possible to alter a character in that manner.

Code 2.13 Creating simple strings.

```
>>> mystr = 'this is a string'
>>> bstr = " and another"
>>> mystr+bstr
'this is a string and another'
```

Code 2.14 Accessing characters in a string.

```
>>> mystr[:4]
'this'
>>> mystr[4] = 'A'

Traceback (most recent call last):
 File "<pyshell#67>", line 1, in <module>
   mystr[4] = 'A'
TypeError: 'str' object does not support item assignment
```

2.5.1 String Functions

A string has several functions of which only a few are reviewed here. Code 2.15 shows examples of locating specific substrings within a string. In Line 1 the first occurrence of `s` in `mystr` is located. The function `find` returns the number 3 which indicates that the first occurrence is at `mystr[3]`. In Line 3 a second argument is added which indicates that the search begins at position 4. In Line 5 a third argument is added to indicate that the search begins at position 4 and ends before position 6. In `mystr` there is `s` between these two positions and therefore the function returns a−1.

The `rfind` function is a reverse find that begins the search from the end of the string. With only a single argument this function finds the last occurrence of the substring. The `count` function returns the number of occurrences of the substring. The `index` function works similar to the `find` function except that if the substring is not found, Python returns an error message instead of a−1.

Two more functions are shown in Code 2.16. The `upper` function converts all characters to upper case. Not shown is the function `lower` which converts the string to lower case. The `replace` function replaces all occurrences of the first substring with the second substring.

In Code 2.17 a string is split into components and then combined to make a long string. In line 1 the `split` function breaks the string up into several substrings wherever there is a white space character (space, tab, newline, etc.). The function returns a list of strings and the white space characters are absent. In line 3 the function receives an argument to indicate that the string is now to be separated wherever there is an `s`. Again the result is a list of substrings which now have the white spaces but not the splitting substring. A list of strings can be combined back into a single string

Code 2.15 Finding characters in a string.

```
>>> mystr.find( 's' )
3
>>> mystr.find( 's',4 )
6
>>> mystr.find( 's',4, 6 )
-1
>>> mystr.rfind( 's' )
10
>>> mystr.count( 's' )
3
>>> mystr.index( 's' )
3
>>> mystr.index( 's',4,6)

Traceback (most recent call last):
 File "<pyshell#76>", line 1, in <module>
   mystr.index( 's',4,6)
ValueError: substring not found
```

Code 2.16 Converting characters to upper case and replacing characters.

```
>>> mystr.upper()
'THIS IS A STRING'
>>> mystr.replace( 's', '$' )
'thi$ i$ a $tring'
```

using the `join` function. The leading string is placed in between each string in the list as shown in lines 6 and 7. It is possible to join all strings in the list without any new characters as shown in lines 8 and 9.

Code 2.17 Splitting and joining strings.

```
>>> mystr.split()
['this', 'is', 'a', 'string']
>>> mystr.split('s')
['thi', ' i', ' a ', 'tring']
>>> z = mystr.split()
>>> 'R'.join( z )
'thisRisRaRstring'
>>> ''.join(z)
'thisisastring'
```

2.5.2 Type Casting

Strings can be converted to other data types. In Code 2.18 the string is converted to a list, a tuple, and a set. In the latter case the set does not keep duplicates of elements. The conversion of a string to an integer is performed in Line 9. Similarly, a string can be converted to a float, double, long, etc. To convert a number into a string the `str` function is employed as shown in Line 11.

2.6 Control

Like other languages Python controls the flow of a program using loops. Before these are explained it is first necessary to discuss tabbing in Python. In languages like C and Java an `if` statement contains commands placed between curly braces, and languages like Fortran, Pascal, and Matlab use `BEGIN` and `END` statements. Python uses neither but instead uses indents. Code 2.19 shows a simple `if-else` statement. Line 2 ends with a colon and the next line is indented which indicates that this line will be executed if the `if` statement is `True`. In Code 2.20 Lines 2, 3, and 4 will be executed if the condition is `True`.

Editors that are dedicated to Python will generally manage the indentations. In the IDLE editor if a line ends with a colon then the next line is automatically indented. Care should be taken, though, if a program is written using different editors. Some editors will indent with space characters and others will indent with tab characters. While these appear the same in the editor the Python interpretor will know the difference and the program will not work.

Code 2.18 Converting strings to other data types.

```
>>> list( mystr )
['t', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 's', 't',
 'r', 'i', 'n', 'g']
>>> tuple( mystr )
('t', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 's', 't',
 'r', 'i', 'n', 'g')
>>> set( mystr )
set(['a', ' ', 'g', 'i', 'h', 'n', 's', 'r', 't'])
>>> int( '55' )
55
>>> str( 67 )
'67'
```

Code 2.21 shows a compound `if` statement. Key words such as `and, or` and `not` are used to create the logic. Parentheses are not required, but they can be used to further manage the logic. Also in Code 2.21 Line 3 shows the `elif` command which is *else-if*.

Code 2.19 A simple `if` statement.

```
>>> a = 5
>>> if a > 3:
   print 'Yes'
else:
    print 'No'

Yes
```

Code 2.20 A simple `if` statement with multiple commands.

```
>>> if a > 3:
    print 'Yes'
    b = a + 5
    print b
else:
    print 'No'

Yes
10
```

Code 2.21 A compound `if` statement.

```
>>> if a < 3 and a!=5:
        print 'Yes'
    elif a==5:
        print 'It is 5'

It is 5
```

The `while` loop is managed similarly to the `if` loop as shown in 2.22. The `if` and `while` loops behave similarly to other languages such as C and Java. The `for` loop behaves differently from the other languages. In the `for` command the index becomes elements that are contained in a list. Recall `mylist` from Code 2.7. In Code 2.23 the index `i` sequentially becomes each element in the list and the single command inside of the loop prints the value of `i` upon each iteration of the loop. In this case, the index can be a different data type in each iteration.

In Code 2.24 the `range` function is used to create a list of integers `[0,1,2,3,4,5]` and the index becomes each of these in each iteration. The comma in the `print` statement prevents a newline character from being printed. The range function can receive two arguments which indicate where the list should start and stop, or it can receive three arguments which indicate the start, stop, and step integers.

Code 2.22 A `while` statement.

```
>>> a = 9
>>> while a>3:
        print a
        a -=1 # decrement a

9
8
7
6
5
4
```

Code 2.23 A `for` loop.

```
>>> for i in mylist:
    print i

(4, 5.6, 'a string', (7+9j))
another string
```

Code 2.24 A traditional `for` loop.

```
>>> for i in range( 6 ):
    print i,

0 1 2 3 4 5
```

Like other languages the `break` command can be used inside of a loop to terminate the loop prematurely, and the `continue` statement can be used to forgo subsequent statements in the loop and continue on to the next iteration.

2.7 Input and Output

Writing and reading text and data files in Python is quite easy and straightforward. Like all languages there are three steps to reading (or writing) a file. The file is opened, the data is read (or written), and the file is closed.

2.7.1 Basic Files

Lines 2–4 in Code 2.25 show the three steps in writing to a text file, and likewise Lines 6–8 read in the file. In Line 6 the file is opened without the `'w'` option and

thus it is opened for reading. Line 10 performs the same steps as lines 6–8 but in a single command.

Code 2.25 Writing and reading a text file.

```
# writing a file
>>> fp = file( "mytext.txt", 'w' )
>>> fp.write( 'This is the stuff that is put into the file.' )
>>> fp.close()
# reading a file
>>> fp = file( "mytext.txt" )
>>> myst = fp.read()
>>> fp.close()
# reading in a single line
>>> myst = file( 'mytext.txt' ).read()
```

Binary data files can be written and read in Python by using `'wb'` or `'rb'` as the second argument in the `file` command. Without the `b` the files will be considered as text files and the binary codes for newline characters will be interpreted as such instead of as a binary number.

2.7.2 Pickle

In cases where complicated data types (tuples, lists, etc.) are to be written or read from the file it is convenient to use the *pickle* module. (Modules are discussed in Sect. 2.9.) The pickler can write/read any type of data in a single line. In Code 2.26 Lines 2–4 open a file and store all contents of `mylist`. This data is stored as a text file and the contents can easily be seen in any text editor. Pickling can also be performed on binary files, however, binary files are not compatible for different operating systems. In other words, a binary file pickled on a Mac is not readable on a PC.

Code 2.26 Pickling.

```
>>> import pickle
>>> fp = open( 'myfile.pickle', 'w' )
>>> pickle.dump( mylist, fp )
>>> fp.close( )
# reading
>>> fp = open( 'myfile.pickle' )
>>> alist = pickle.load( fp )
>>> fp.close( )
```

2.8 Functions

A function in Python is similar to functions, subroutines, or procedures in other languages. It is a set of instructions that can be executed by a single command. Code 2.27 defines a function named `MyFun`. The keyword `def` begins the definition and the arguments received by the function are enclosed in parenthesis. In this case a single argument is received and printed out in Line 3. Lines 5 and 8 show two different calls to this function and their results.

Code 2.27 A simple function.

```
>>> def MyFun( a ):
    print 'Inside the function'
    print a

>>> MyFun( 12 )
Inside the function
12
>>> MyFun( 'textual information' )
Inside the function
textual information
```

The function `MyFun2` in Code 2.28 receives two arguments and returns two arguments. In actuality the function returns a single item which is a tuple and within the tuple there are, in this case, two integers. In Line 9 the tuple is received by a tuple with the variables `m` and `n`. These become the individual elements of the tuple and so this step is equivalent to the function returning two integers.

Code 2.28 A function returning data.

```
>>> def MyFun2( a, b ):
    print 'Inside the function'
    return a+1,b-1

>>> a = MyFun2( 5, 7 )
Inside the function
>>> a
(6, 6)
>>> m, n = MyFun2( 10, -1 )
Inside the function
>>> m
11
>>> n
-2
```

Functions can also have default arguments. These are arguments the have predefined values that can be altered if the user desires. Default arguments have already been used in functions such as `string.find` and `range`. In Code 2.29 a function is created that receives four arguments but two of them are predefined. In Line 4 the function is called with only 2 arguments and thus `c` and `d` assume their default values. In Line 6 a third argument is used which redefines `c`. In Line 8 all arguments are defined. In Line 10 the `c` keeps the default value where `d` is redefined. Default arguments are extermely convienient but do have one caveat that they must be the last items in the function's argument list.

Code 2.29 Default arguments.

```
>>> def MyFun3( a, b, c=12, d=0 ):
    print a,b,c,d

>>> MyFun3( 1,2 )
1 2 12 0
>>> MyFun3( 1,2,3 )
1 2 3 0
>>> MyFun3( 1,2,3,4 )
1 2 3 4
>>> MyFun3( 1,2,d=5 )
1 2 12 5
```

2.9 Modules

A module is merely a text file that contains Python instructions. These can be commands, functions, or objects (see Sect. 2.10). Lines 2 and 3 in Code 2.30 define a function and this text is stored in a file named "mymodule.py". Line 6 is run in the command window and this loads the module into Python using the `import` command. This assumes that `sys.path` is set correctly (see Sect. 2.1.3). The function is called in Line 7 by *module.functionName*. It is possible to import a module, modify it and then reload it. In this case the `>>> reload mymodule` command is called to refresh contents.

It can be tedious to write the module name each time a function or variable from a module is used. A second method of importing a module is shown in Code 2.31. Using `from—import` it is no longer necessary to use the module name to call the function. There are, however, two caveats to this method. The first is that a module can not be reloaded. The second is that it is possible to erase a previously defined function or variable. If the function `MyFun4` had been previously defined then Line 1 would have eliminated the previous definition.

Code 2.30 Creating a module.

```
# contents of a file named: mymodule.py
def MyFun4( a ):
    print a + 6

# commands
>>> import mymodule
>>> mymodule.MyFun4( 5 )
11
```

Code 2.31 From: import.

```
>>> from mymodule import MyFun4
>>> MyFun4( 5 )
11
```

A third method of accessing functions and definitions in a file is to use `execfile`, which is a useful command that meets an unfortunate demise in Python 3.x. Technically, this is not loading a module and does not use `sys.path`. Rather, this command is similar to just typing in all of the lines in a file. Code 2.32 shows the loading of a python file requiring the path and full file name. Modifications to mymodule.py can be reloaded by running line 1 again. This command is useful when developing codes because it is easy to reload.

Code 2.32 Using execfile.

```
>>> execfile( 'pysrc/mymodule.py')
>>> MyFun4( 5 )
11
```

There will be two types of code presentations in the subsequent chapters. Like Code 2.32 commands that are to be entered directly into the command shell are preceded by >>>. The second type is to show what is contained within a module. An example of this is shown in Code 4.1 in which the name of the module is located in Line 1. These commands are not preceded by >>> and are to be imported or accessed through `execfile`. There are some like Code 6.10 which show both the content within a module and a few commands for the shell. These are paired such that the latter commands demonstrate how to use the function and achieve the shown results.

2.10 Object Oriented Programming

Python is technically an object-oriented language although such programming can appear transparent to the user. An object (or a class) can contain variables and functions. A good example of this is the string class (see Sect. 2.5). In this case the variable was the character data and the functions extracted information or modified the data. Objects are easy to write but are absolutely not required when creating Python scripts.

2.10.1 Content of a Class

Code 2.33 creates the class named `MyObject` which contains a variable `g` and a function `MyFun`. Line 8 creates an instance of this object and as seen in Line 9 and 10 it contains the variable `g`. The function is called in Line 11 and Lines 13 and 14 shows the new value of the variable.

Readers that are familiar with C++ and Java will recognise the keyword `self` as equivalent to `*this` or `this`. Every function inside of the object will have `self` as the first argument and it is transparent to the user when the function is called. The `self` connects the object to the instance which is `myinst` in this case. Thus, inside of the function `self.g` is equivalent to `myinst.g` outside of the object.

An object can have global and local variables. A global variable is Lines 2 and 4 whereas a local variable is Line 5. The local variable exists only inside of the function and once the function terminates this variable is destroyed. In this example it was a poor programming skill to have a global and local variable of the same name but it does illustrate that they are separate variables. The result in Line 14 shows that the global variable is available but the local one is not.

2.10.2 Operator Definitions

An object can have any number of functions and variables. Python does not offer overloading as does C++ but it does allow for the definition of operators. Code 2.34 defines a new object that redefines two functions. In each case the function names are preceded and followed by two underscores. The `__init__` (two underscores each for and aft) function is the constructor. This function is called when the object is instantiated (Lines 11 and 14). As seen in Lines 12 and 13 the function is called and the value of `g` is set to -1. The second function is `__add__` which defines the action taken by the plus sign (Line 16). Inside of `MyObject.__add__` the `self.g` becomes the `a1.g` in Line 16 and `b` becomes `a2` in line 16. Almost every function can be defined in an object. This includes math operators, slicing operators, and many more. Interested readers should refer to Python manuals to learn more.

Code 2.33 A simple object.

```
>>> class MyObject:
    g = 5
    def MyFun( self, a ):
        self.g = 7
        g = 0
        print 'argument = ',a

>>> myinst = MyObject()
>>> myinst.g
5
>>> myinst.MyFun( 12 )
argument = 12
>>> myinst.g
7
```

Code 2.34 Operator definition.

```
>>> class MyObject:
        g = 5
        def __init__( self ):
                self.g = -1
        def __add__( self, b ):
                answer = self.g + b.g
                return answer
        def MyFun( self, a ):
                self.g = 7

>>> a1 = MyObject( )
>>> a1.g
-1
>>> a2 = MyObject()
>>> a1.MyFun( )
>>> c = a1 + a2
>>> c
6
```

2.10.3 Inheritance

One of the advantages of using objects is that it can provide building blocks for complicated codes. Objects can be built using other objects through inheritance. Code 2.35 shows a simple example which defines the class *Sentence*. This class has a single class variable and function. The command `stn = Sentence( )` creates an instance of this sentence. The second class is *BigSentence* which inherits

Sentence. An instance of *BigSentence* inherits all of the properties of *Sentence* and adds the new function **Double**. The command `db = BigSentence( )` creates an instance of *BigSentence* and the next line shows that *db.e* is already defined. The *db* has two functions **Erase** and **Double** that it can use.

Code 2.35 Inheritance.

```
>>> class Sentence:
        e = 'default sentence'
        def Erase( self ):
           self.e = ''

>>> class BigSentence( Sentence ):
        def Double( self ):
            self.e = self.e + self.e

>>> stn = Sentence( )
>>> stn.e
'default sentence'
>>> db = BigSentence()
>>> db.e
'default sentence'
>>> db.Double()
>>> db.e
'default sentencedefault sentence'
```

2.11 Error Checking

Some of the previous programs have intentional errors and Python reports these errors with a few lines indicating the problem. If the error occurs in a module then Python reports the filename and the line number where the error is detected. This does not necessarily mean that this line is the culprit though. The function `MyObject.__add__` requires data of the same type and if `a1.g` were an integer and `a2.g` were a string then an error would occur inside of this function. However, the real problem was that the wrong types of data were fed into the program.

It is possible to catch errors inside of a function. Line 1 in Code 2.36 performs an illegal function and Python returns TypeError error. Lines 7–10 use the `try` and `except` commands to manage the error from the same command. This is useful for cases when it is not desirable for a program to quit when an error is encountered. There are several types of errors and Python allows the user to define their own errors. Interested readers should consult the Python manuals for more information.

Code 2.36 Trapping an error.

```
>>> 5 + 't'

Traceback (most recent call last):
  File "<pyshell#168>", line 1, in <module>
    5 + 't'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> try:
    5 + 't'
except TypeError:
    print 'Ooooops'

Ooooops
```

2.12 Summary

Python is a powerful scripting language and this chapter just presents the essentials that are necessary to perform the tasks in the subsequent chapters. Python can be used to perform numerical calculations, manage strings, manage data, and as seen in later chapters operate on images and data arrays. Python has advantages over other languages since it is free, easy to install, and easy to use. Properly coded Python scripts can actually run at fairly good speeds as well.

Chapter 3
NumPy, SciPy and Python Image Library

Two related third-party packages named NumPy and SciPy [3] provide tools for creating arrays (vectors, matrices, and tensors) and a plethora of tools for manipulating these arrays. These tools provide very efficient codes that make programming easier and computations quick. Since neural models consider arrays of neurons these tools are essential. A third package named Python Image Library (PIL) provides tools or reading and writing image data.

3.1 NumPy

The NumPy package provides functions to create and manipulate arrays. An array is a collection of uniform data types such as vector, matrix, or tensor.

In order to use the NumPy package it must first be imported. Since there will be many calls to the routines in this package it is prudent to use `from–import`.

3.1.1 Creating Arrays

There are several ways to create a vector or matrix of which a few are shown in Code 3.1. Line 2 creates a vector with ten elements all are set to 0. Line 5 creates a ten element vector in which all of the elements are integers. Lines 8–10 demonstrate that vectors can also contain complex numbers.

One of the advantages of using arrays is that global operations can be performed in a single command. Code 3.2 demonstrates that a scalar can be added to a vector in a single command (Line 2). The division of an integer array with another integer will result in an integer array as seen in Line 4. Again, this may not be an exact answer as shown in Line 6.

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_3, © Springer-Verlag Berlin Heidelberg 2013

Code 3.1 Creation of vectors.

```
>>> from numpy import zeros, ones, random
>>> vec = zeros( 10 )
>>> vec
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
>>> vec = ones( 10, int )
>>> vec
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> vec = zeros( 4, complex )
>>> vec
array([ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j])
```

Code 3.2 Math operations for vectors.

```
>>> vec = ones( 4, int )
>>> vec + 2
array([3, 3, 3, 3])
>>> (vec + 12)/4
array([3, 3, 3, 3])
>>> (vec + 12.)/4
array([ 3.25,  3.25,  3.25,  3.25])
```

Code 3.3 Math operations for two vectors.

```
>>> from numpy import set_printoptions
>>> set_printoptions( precision=3)
>>> vec1 = random.rand( 4 )
>>> vec2 = random.rand( 4 )
>>> vec1
array([ 0.938,  0.579,  0.867,  0.066])
>>> vec2
array([ 0.574,  0.327,  0.293,  0.649])
>>> vec1 + vec2
array([ 1.512,  0.906,  1.159,  0.716])
```

Code 3.3 shows three conveniences. The first is to control the precision of the print to the console. The `set_printoptions` command is used to limit the number of digits that are printed to the screen. This function comes from NumPy and does not apply to lists or tuples. The second item in this code is the creation of two vectors of random numbers. The `random.rand` function creates a vector with random numbers ranging from 0 to 1 with equal distribution. The `random` module provides other types of distributions as well, and these are described in the manual that accompanies NumPy. The third part of this code is to demonstrate addition of two vectors (Line 9). As long as the vectors have the same number of elements it is possible to perform many different mathematical operations.

A couple of methods of creating matrices are shown in Code 3.4. In Line 1 the `zeros` function receives a single argument-this is a tuple that contains two elements. For a matrix the vertical dimension is given first and the horizontal dimension is second. In Line 5 the `random.ranf` function is used to create a matrix with random elements.

Code 3.4 Creating matrices.

```
>>> mat = zeros( (2,3) )
>>> mat
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> mat = random.ranf( (2,3) )
>>> mat
array([[ 0.181,  0.283,  0.185],
       [ 0.978,  0.468,  0.612]])
```

Tensors are created in the same manner as matrices except that there are at least three elements in the tuple. However, in allocation of tuples the user should be aware that memory can be consumed rather quickly. A 256×256 matrix with floating point elements consumes 0.5 MB. However, a tensor that is $256 \times 256 \times 256$ with floating point elements consumes 128 MB. Furthermore, if two tensors $c = a + b$ are added then there needs to be space for four tensors (a, b, c, and $a + b$). Before tensors are allocated the user needs to make sure that there is sufficient memory in the computer. Code 3.5 demonstrates the allocation of a much smaller tensor.

Code 3.5 Creating tensors.

```
>>> tensor = zeros( (12,15,16), float )
```

Accessing data in arrays is performed through slicing. Line 1 in Code 3.6 demonstrates how to retrieve a single cell, which in this example, is from the first row and second column. Indexing in matrices uses the vertical dimension first and the horizontal dimension second following standard math convention. Advanced slicing is discussed in Sect. 3.1.9.

Code 3.6 Accessing data in a matrix.

```
>>> mat[0,1]
0.283461998819
>>> mat[1,1] = 4
```

3.1.2 Converting Arrays

Converting between a matrix and a vector is easily performed with the `resize` and `ravel` commands. Line 1 in Code 3.7 converts the 2×3 matrix into a 6 element vector. Line 4 converts `data` into a 3×2 matrix. In Line 1 the answer is returned to a new variable, but in Line 4 the variable is changed. It is possible to convert a matrix of one size ($M \times N$) to a matrix of another size ($P \times Q$) using `resize` as long as $M \times N = P \times Q$. The `shape` command returns the dimensions of the array as seen in Line 9.

Code 3.7 Converting between vectors and matrices.

```
>>> data = mat.ravel()
>>> data
array([ 0.181,  0.283,  0.185,  0.978,  4.   ,  0.612])
>>> data.resize( (3,2) )
>>> data
array([[ 0.181,  0.283],
       [ 0.185,  0.978],
       [ 4.   ,  0.612]])
>>> data.shape
(3, 2)
```

3.1.3 Matrix: Vector Multiplications

The matrix-vector or vector-matrix multiplications are performed using the `dot` function. In Code 3.8 the vM multiplication is performed in Line 12 and the Mv multiplication is performed in Line 14.

NumPy also allows an elemental multiplication in which the columns of the matrix are multiplied by the respective elements in the vector, as in

$$P_{i,j} = M_{i,j} v_j. \tag{3.1}$$

In this case, the length of the vector must be the same as the second dimension of the matrix. Line 1 in Code 3.9 performs this computation of Eq. 3.1. Unfortunately, NumPy will allow the same computation in the form shown in Line 5 which, in appearance, should be a different computation. Line 9 confirms the action by replicating the multiplication of the first column of the matrix with the first element of the vector. The result can be compared to the first column of the previous computations.

Code 3.8 Vector-matrix and matrix-vector multiplications.

```
>>> M = random.ranf( (3,2) )
>>> v1 = random.rand( 3 )
>>> v2 = random.rand( 2 )
>>> M
array([[ 0.566,  0.378],
       [ 0.782,  0.428],
       [ 0.727,  0.966]])
>>> v1
array([ 0.148,  0.2  ,  0.578])
>>> v2
array([ 0.103,  0.913])
>>> dot( v1, M )
array([ 0.661,  0.7  ])
>>> dot( M, v2 )
array([ 0.403,  0.472,  0.958])
```

Code 3.9 Multiplying columns of a matrix with elements in a vector.

```
>>> M*v2
array([[ 0.059,  0.345],
       [ 0.081,  0.391],
       [ 0.075,  0.882]])
>>> v2*M
array([[ 0.059,  0.345],
       [ 0.081,  0.391],
       [ 0.075,  0.882]])
>>> M[:,0] * v2[0]
array([ 0.059,  0.081,  0.075])
```

3.1.4 Justification for Arrays

Readers familiar with sequential programming languages such as C, Fortran, and Java are used to thinking in terms of `for` loops to perform the tasks such as multiplying matrices and vectors. Certainly, Python can also use `for` loops to perform the computations of Code 3.9.

The reason to use the NumPy commands is that there is a tremendous speed advantage. A script file named *timestudy.py* is used to demonstrate the advantage of using NumPy commands in Code 3.10. In Line 2 the `time` module is imported. It also contains a function also named `time()` which returns a float that is computer's clock in terms of seconds. In Line 9 the variable `t1` is set to the current time and the next few lines perform the process of Code 3.9. At the end of these loops the computer time is sampled a second time and the difference is printed to the console. This is the amount of time necessary to perform the double-loop computation. In Line 18 the same process is performed again using the NumPy command, and the time for this process is printed to the console as well.

In the second part of Code 3.10 the commands are executed and the computation times are printed. The `for` loops needed 2.5 s to complete whereas the NumPy command needed less than 0.02 s. The NumPy script is 159 times faster.

Code 3.10 Comparing the computational costs of interpreted commands.

```
# timestudy.py
import time
from numpy import zeros, random

M = random.ranf( (1000,1000) )
v = random.rand( 1000 )
N = zeros( (1000,1000) )

t1 = time.time()
for i in range( 1000 ):
    for j in range( 1000 ):
        N[i,j] = M[i,j] * v[j]
t2 = time.time()

print 'First time:', t2-t1

t3 = time.time()
N = M * v
t4 = time.time()

print 'Second time:', t4-t3
```

```
>>> execfile('timestudy.py')
First time: 2.54699993134
Second time: 0.0159997940063
>>> 2.54/.016
158.75
```

The reason for the massive difference in computational time is that Python is an interpreted language. The computer receives one command at a time, converts it to an executable command, and then performs the computation. Thus, a line inside of a `for` loop must be interpreted many times. Whereas, in the case of the single NumPy step the Python calls a step that is already compiled.

Basically, the rule of thumb is that NumPy commands will be significantly faster than a set of Python commands that perform the same computations using loops. The savings is large enough that it is worth the reader's time to read NumPy documentation and learn the available commands. A properly written Python/NumPy script can rival computational speeds of compiled languages. Certainly, once the reader becomes familiar with NumPy it becomes much faster and easier to write programs.

Table 3.1 Data types offered by NumPy

Name	Character	Number of bytes
bool		1
int	l	4
float	d	8
complex	D	16
int8	1	1
int16	s	2
int32	i	4
int64		8
float32	f	4
float64	f	8
float96		8
uint8	u1	1
uint16	u2	2
uint		4
int8	1	1
int16	s	2
int32	i	4
int64		8
int128		16
complex64		8
complex128		16
complex192		24

3.1.5 Data Types

In an array all of the elements must be the same data type. Thus, an array can have all integers or all floats. An array by default will contain floats but NumPy does offer several alternate data types as shown in Table 3.1.

There are several parameters or functions that return information from an array. The `dtype` parameter indicates the type of data that is in the array. This is not a function but rather a parameter defined in the class. Therefore, it does not have parenthesis. An example is shown in Code 3.11.

Code 3.11 Retrieving the type of data within an array.

```
>>> mat = random.ranf( (3,4) )
>>> mat
array([[ 0.592,  0.514,  0.885,  0.995],
       [ 0.587,  0.764,  0.528,  0.988],
       [ 0.799,  0.011,  0.616,  0.687]])
>>> mat.dtype
dtype('float64')
```

The max function returns the maximum value of the array and it has some very useful options. In Line 1 of Code 3.12 the function is used to retrieve the maximum value over the entire array. The function has the option of specifying which axis (or dimension) is used. Recall that in a matrix the first axis is the vertical dimension and the second axis is the horizontal dimension. In Line 3 the axis is specified and the maximum is computed over the vertical dimension. In other words, this computes the maximum for each column of the matrix. By specifying the second axis (Line 5) the maximum is found for each row.

Code 3.12 Using the max function.

```
>>> mat.max()
0.995156726117
>>> mat.max(0)
array([ 0.799,  0.764,  0.885,  0.995])
>>> mat.max(1)
array([ 0.995,  0.988,  0.799])
```

There are many other functions that operate in this manner. These include the retrieving the minimum (min), computing the sum (sum), computing the average (mean), and the standard deviation (std). A few examples of these are shown in Code 3.13.

Code 3.13 Using the similar functions.

```
>>> mat.sum(0)
array([ 1.978,  1.289,  2.029,  2.671])
>>> mat.mean(1)
array([ 0.746,  0.717,  0.528])
>>> mat.std()
0.253734951758
```

The max function returns the maximum value but doesn't indicate where the maximum value is in the array. This is accomplished through the argmax function. For a vector this returns the location of maximum as shown in Line 4 of Code 3.14.

The argmax function for a matrix requires a little bit more work. It returns a single number which indicates the location of the max value. In the example in Code 3.15 the function returns a value of 5 but the matrix is only 3×4. The function treats the matrix as though it were a raster and thus the location of the max in terms of row and column is computed by dividing the argmax value by the number of rows. The result in line 10 shows that the max value is located at row 1 and column 1 (recalling that the first row is row 0).

The nonzero function returns a list of indices were the array has values that are not 0. Code 3.16 shows how this is used to find the locations within a vector

Code 3.14 Using the similar functions.

```
>>> vec = random.rand( 5 )
>>> vec
array([ 0.21 ,  0.268,  0.023,  0.998,  0.386])
>>> vec.argmax()
3
>>> vec[3]
0.9979
```

Code 3.15 Using the similar functions.

```
>>> mat = random.ranf( (3,4) )
>>> mat.argmax()
5
>>> mat.shape
(3, 4)
>>> mat
array([[ 0.17 ,  0.824,  0.414,  0.265],
       [ 0.018,  0.865,  0.112,  0.398],
       [ 0.721,  0.77 ,  0.083,  0.065]])
>>> divmod( mat.argmax(), 4 )
(1, 1)
>>> mat[1,1]
0.8649
```

and a matrix that are greater than a threshold. The `nonzero` function returned an array inside of a tuple which at first may seem odd. Consider the case of the matrix that starts in Line 12. In Line 17 the `nonzero` function is applied to the matrix and Line 18 indicates that there are two items in the tuple `nz`. There is an entry for each dimension in the array and these are shown in Lines 20–23. These are the locations in which `gmat` is not zero with `nz[0]` displaying the vertical coordinates and `nz[1]` displaying the horizontal components. Thus, the four locations in which `gmat` is not zero are (0,1), (1,1), (2,0), and (2,1).

NumPy offers several mathematical functions that are applied to all elements of the array. Code 3.17 shows the square root function applied to all elements of an array. Several other popular functions are shown in Table 3.2. These functions can be applied to the entire array or any axis similar to **sum** function uses in Sect. 3.1.5.

3.1.6 Sorting

The elements in an array can be sorted in two ways. The first is the `sort` function which rearranges the elements in the array. The second is `argsort` which returns

Code 3.16 Using the `nonzero` function.

```
>>> vec
array([ 0.21 ,  0.268,  0.023,  0.998,  0.386])
>>> gvec = vec > 0.3
>>> gvec
array([False, False, False, True, True], dtype=bool)
>>> nz = gvec.nonzero()
>>> nz
(array([3, 4]),)
>>> nz[0]
array([3, 4])

>>> mat
array([[ 0.17 ,  0.824,  0.414,  0.265],
       [ 0.018,  0.865,  0.112,  0.398],
       [ 0.721,  0.77 ,  0.083,  0.065]])
>>> gmat = mat > 0.5
>>> nz = gmat.nonzero()
>>> len( nz )
2
>>> nz[0]
array([0, 1, 2, 2])
>>> nz[1]
array([1, 1, 0, 1])
```

Code 3.17 Mathematical functions for an array.

```
>>> mat
array([[ 0.17 ,  0.824,  0.414,  0.265],
       [ 0.018,  0.865,  0.112,  0.398],
       [ 0.721,  0.77 ,  0.083,  0.065]])
>>> sqrt( mat )
array([[ 0.412,  0.908,  0.643,  0.514],
       [ 0.134,  0.93 ,  0.335,  0.631],
       [ 0.849,  0.878,  0.287,  0.256]])
```

Table 3.2 Math functions

sin	arcsin	sinh	arcsinh
cos	arccos	cosh	arccosh
tan	arctan	tanh	arctanh
exp	log	log2	log10
sqrt	conjugate	floor	ceil

an array indicating the sort order. Line 3 in Code 3.18 uses the `argsort` function to determine the order in which the data should be rearranged from the lowest to the highest values. In this case the lowest value is determined to be `vec[2]` and the largest value is at `vec[3]`. In Lines 6 and 7 the `argsort` and slicing techniques

are used to extract the data in the sorted order. In this case the `vec` is not altered. In Line 10 the `sort` function is used to irreversibly change the data in `vec`. The data is sorted but `vec` has been changed. Starting with Line 17 the `argsort` command is applied to a matrix. The function has the optional argument of specifying which axis is used during the sorting process.

Code 3.18 Sorting arrays.

```
>>> vec
array([ 0.21 ,  0.268,  0.023,  0.998,  0.386])
>>> vec.argsort()
array([2, 0, 1, 4, 3])

>>> ag = vec.argsort()
>>> vec[ag]
array([ 0.023,  0.21 ,  0.268,  0.386,  0.998])

>>> vec.sort()
>>> vec
array([ 0.023,  0.21 ,  0.268,  0.386,  0.998])

>>> mat
array([[ 0.17 ,  0.824,  0.414,  0.265],
       [ 0.018,  0.865,  0.112,  0.398],
       [ 0.721,  0.77 ,  0.083,  0.065]])
>>> mat.argsort(0)
array([[1, 2, 2, 2],
       [0, 0, 1, 0],
       [2, 1, 0, 1]])
>>> mat.argsort(1) # same as mat.argsort( )
array([[0, 3, 2, 1],
       [0, 2, 3, 1],
       [3, 2, 0, 1]])
```

3.1.7 Conversions to Strings and Lists

Data in an array can be converted to and from a string using the commands `tostring` and `fromstring`. The conversion is dependent upon the type of data. In Code 3.19 a vec is created with random float data. Each float requires 8 bytes of storage. In line 4 the array is converted to a string which is 40 characters long. Recall that not all ASCII values have an associated character. Thus, the first entry in `ts` is the hexadecimal number `C0` as shown in Lines 8 and 9.

A string can be converted back to an array as shown in Line 10. The `fromstring` function as a default converts the string to an array of `ints`. In this case the data type was changed to `float`. Line 13 converts the same string to `ints` and Line 17

converts then to unsigned integers which is equivalent to converting the individual characters to their 8-bit values.

Code 3.19 Conversions to and from a string.

```
>>> vec = random.rand( 5 )
>>> vec
array([ 0.12 ,  0.643,  0.507,  0.144,  0.407])
>>> ts = vec.tostring()
>>> ts
'\xc0\xb9\xc2\xbb\xa3\xcb\xbe?\x82\xa4\xbc\xcc\xa8\x94\xe4?\x00B
\xc5>G;\xe0?\x80‘_\x8a\xf3_\xc2?\xae\xe9\xd7\x93n\x0f\xda?'
>>> ts[0]
'\xc0'
>>> vec2 = fromstring( ts, float )
>>> vec2
array([ 0.12 ,  0.643,  0.507,  0.144,  0.407])
>>> fromstring( ts, int )
array([-1144866368,  1069468579,  -860052350,  1071944872,
        1053114880,  1071659847, -1973460864,  1069703155,
       -1814566482,  1071255406])
>>> fromstring( ts, uint8 )
array([192, 185, 194, 187, 163, 203, 190,  63, 130, 164, 188,
       204, 168, 148, 228,  63,   0,  66, 197,  62,  71,  59,
       224,  63, 128,  96,  95, 138, 243,  95, 194,  63, 174,
       233, 215, 147, 110,  15, 218,  63], dtype=uint8)
```

These two functions are most useful when reading or writing a file with binary data. However, reading and writing binary data is no longer a platform independent operation. A binary file written on a Macintosh will be different than a binary file of the same data written on a PC. The reason is that the operating systems store data in different arrangements. Traditionally, Macintosh and UNIX machines store data as Big Endian and Windows based machines store data as Little Endian. The latter reverses the byte order of the data. For example, a little endian machine will store a two-byte number with the least significant byte first. The result is that data stored on a big endian machine and read on a little endian machine will need to have the bytes swapped. In the bioinformatics field this is required since many early experimental devices were hosted by Macintosh computers.

Code 3.20 shows the function `byteswap` which exchanges the byte order of data in an array. Line 1 creates a random vector and is converted to a `int16` data type so that the single element in the array is stored in two bytes. These two bytes are printed out as characters in Line 5 and as byte values in Line 7. The `byteswap` function is used in Line 8 and the reversed data is shown in Line 9. In Line 11 a new array is created in which there are two elements and these are 32 bit elements (four bytes each). The `byteswap` function is again used demonstrating that the swap occurs for individual elements. In this case, a group of 4 bytes are reversed.

Code 3.20 Swapping bytes in an array.

```
>>> a = (16384 * random.rand( 1 )).astype(int16)
>>> a
array([15165], dtype=int16)
>>> a.tostring()
'=;'
>>> map( ord, a.tostring() )
[61, 59]
>>> map( ord, a.byteswap().tostring() )
[59, 61]

>>> a = (16000 * random.rand( 2 )).astype(int)
>>> a
array([10320, 11416])
>>> map( ord, a.tostring() )
[80, 40, 0, 0, 152, 44, 0, 0]
>>> map( ord, a.byteswap().tostring() )
[0, 0, 40, 80, 0, 0, 44, 152]
```

3.1.8 Changing the Matrix

The transpose of a matrix **M** is,

$$T_{i,j} = M_{j,i} \ \ \forall i, j. \tag{3.2}$$

The `transpose` function converts the matrix to its transpose as shown in Line 5 of Code 3.21. For arrays that have more than two axes the `transpose` function allows the user to select the axes that are transformed. In Line 9 a three-dimensional array is created. The transpose computed in Line 19 indicates the order in which the axes should be extracted. The first axis is 2 and thus the length of the first dimension of the result is 4 which was the length of axis-2 in **M**.

The `resize` function rearranges the elements of a matrix to fit a new shape. The function requires that the number of elements in the new size is the same as the number of elements in the original matrix, although the number of axes does not have to be the same. Examples of the `resize` function are shown in Code 3.22.

3.1.9 Advanced Slicing

Arrays can be sliced in manners similar to strings, tuples, etc. However, arrays offer advanced slicing techniques that are also quite useful in making efficient code. In Code 3.23 a vector of ten elements is created. In Line 5 the first three elements are retrieved and in Line 7 every other element is retrieved. These methods behave the same as in the case of strings, tuples, lists, etc. In Line 9 a list of integers is created and in Line 10 this list is used as an index to the vector. The result is that the elements

Code 3.21 Examples of the `transpose` function.

```
>>> M = random.ranf( (2,3) )
>>> M
array([[ 0.984,  0.816,  0.158],
       [ 0.081,  0.86 ,  0.836]])
>>> M.transpose()
array([[ 0.984,  0.081],
       [ 0.816,  0.86 ],
       [ 0.158,  0.836]])
>>> M = random.ranf( (3,2,4) )
>>> M
array([[[ 0.067,  0.894,  0.789,  0.905],
        [ 0.314,  0.757,  0.288,  0.649]],

       [[ 0.846,  0.951,  0.338,  0.746],
        [ 0.717,  0.004,  0.113,  0.22 ]],

       [[ 0.36 ,  0.168,  0.569,  0.302],
        [ 0.542,  0.969,  0.943,  0.335]]])
>>> B = M.transpose( (2,0,1) )
>>> B.shape
(4, 3, 2)
>>> B
array([[[ 0.067,  0.314],
        [ 0.846,  0.717],
        [ 0.36 ,  0.542]],

       [[ 0.894,  0.757],
        [ 0.951,  0.004],
        [ 0.168,  0.969]],

       [[ 0.789,  0.288],
        [ 0.338,  0.113],
        [ 0.569,  0.943]],

       [[ 0.905,  0.649],
        [ 0.746,  0.22 ],
        [ 0.302,  0.335]]])
```

are extracted from the vector in the order prescribed by the list `n`. This allows the user to extract data in a specified order. Likewise, it is possible to set values in the array in a specified order as shown in Line 12.

This same advanced technique applies to arrays with multiple dimensions. In Code 3.24 elements in a matrix are accessed using two index lists `v` and `h`. The first index represents element locations along the first axis (hence `v` represents the vertical dimension). The first element extracted is `M[1,1]`, the second element is `M[2,1]`, and the third element is `M[0,2]`. Using this technique the user can access elements in an array in any order without employing a Python `for` loop. This will dramatically increase execution time especially for large arrays.

Code 3.22 Examples of the `resize` function.

```
>>> M = random.ranf( (3,4) )
>>> M
array([[ 0.957,  0.561,  0.262,  0.556],
       [ 0.21 ,  0.704,  0.537,  0.048],
       [ 0.829,  0.404,  0.278,  0.335]])
>>> M.resize( (2,6) )
>>> M
array([[ 0.957,  0.561,  0.262,  0.556,  0.21 ,  0.704],
       [ 0.537,  0.048,  0.829,  0.404,  0.278,  0.335]])
>>> M.resize( (3,2,2) )
>>> M
array([[[ 0.957,  0.561],
        [ 0.262,  0.556]],

       [[ 0.21 ,  0.704],
        [ 0.537,  0.048]],

       [[ 0.829,  0.404],
        [ 0.278,  0.335]]])
```

Code 3.23 Advanced slicing for arrays.

```
>>> V = random.rand( 10 )
>>> V
array([ 0.06 ,  0.796,  0.775,  0.592,  0.174,  0.764,  0.952,
        0.871,  0.569,  0.545])
>>> V[:3]
array([ 0.06 ,  0.796,  0.775])
>>> V[::2]
array([ 0.06 ,  0.775,  0.174,  0.952,  0.569])
>>> n = [4,1,3,0]
>>> V[n]
array([ 0.174,  0.796,  0.592,  0.06 ])
>>> V[n] = -1, -2, -3, -4
>>> V
array([-4.   , -2.   ,  0.775, -3.   , -1.   ,  0.764,  0.952,
        0.871,  0.569,  0.545])
```

3.2 SciPy

The NumPy package provides a large foundation of array processing routines. The SciPy [3] package provides a plethora of advanced arrays processing routines. Interested readers are highly encouraged to view SciPy websites (http://www.scipy.org/SciPy_packages) to learn about the wide variety of available routines. For example, SciPy offers linear algebra routines in the `linalg` module. Functions in this module include matrix inversion and eigenvalue computations. A comprehensive list can be

Code 3.24 Advanced slicing for arrays with multiple dimensions.

```
>>> M = random.ranf( (3,4) )
>>> M
array([[ 0.181,  0.663,  0.933,  0.791],
       [ 0.561,  0.145,  0.687,  0.877],
       [ 0.876,  0.881,  0.774,  0.347]])
>>> v = [1,2,0]
>>> h = [1,1,2]
>>> M[v,h]
array([ 0.145,  0.881,  0.933])
```

viewed from SciPy web pages such as (http://www.scipy.org/doc/api_docs/SciPy.linalg.html) and its sub-pages.

SciPy offers packages for Fourier transforms, clustering, integration, interpolation, linear algebra, signal processing, statistics, and optimization. A major advantage for using these routines is that they are optimized to run quickly. As there are a large number of functions they are not reviewed here. As functions are needed in later chapters they will be explained. Code 3.25 demonstrates a call to the `inv` function which performs a matrix inversion. The inverse is computed in Line 8 and the results are confirmed in Line 13 which shows that the multiplication of the inverse with the original matrix produces the identity matrix.

Code 3.25 Matrix inverse using the SciPy function.

```
>>> from numpy import dot
>>> from scipy.linalg import inv
>>> M = random.ranf( (3,3) )
>>> M
array([[ 0.772,  0.588,  0.282],
       [ 0.85 ,  0.817,  0.388],
       [ 0.052,  0.292,  0.618]])
>>> Mi = inv( M )
>>> Mi
array([[ 6.145, -4.406, -0.04 ],
       [-7.918,  7.254, -0.939],
       [ 3.221, -3.054,  2.065]])
>>> dot( M, Mi ) # confirmation
array([[  1.000e+00,  -8.882e-16,   0.000e+00],
       [ -2.220e-16,   1.000e+00,   0.000e+00],
       [  0.000e+00,   0.000e+00,   1.000e+00]])
array([ 0.145,  0.881,  0.933])
```

SciPy is not required to perform operations necessary for the PCNN, ICM or other neural models. However, many of the applications involve images and SciPy

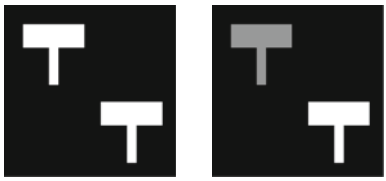
has libraries such as *ndimage* and *signal* which have several useful functions for image processing and analysis.

One simple example pertinent to neural models is to isolate regions outlined by the expanding wave (which is discussed in Sect. 4.2.3.3. Consider a case in which the PCNN creates an output such as Fig. 4.4 at $n = 1$. There are two regions outlined in the image as it is desired that the pixels in these interior regions be isolated. The process can easily be accomplished as shown in Code 3.26. The first three lines load in the appropriate models. Line 4 loads the image of the two T's and Line 5 converts this image to a matrix. At this point the matrix `data` looks like Fig. 4.4 with all pixels shown in white in the figure set to 1 and all pixels shown as black set to 0.

Line 6 creates a new matrix `seeds` which is the same size as `data` and all of the values are initially 0. Line 7 sets two points to 1. These two points are located in the interior of the two T shapes, but any two points located in the interior would work as well. Line 8 calls the **binary_propagation** function from the *scipy.ndimage* module. This function receives a seed (or multiple seeds) and a binary value image. It will expand from the seeds until the borders of the shape are reached and the result is shown in Fig. 3.1a. This is one of the few cases in this text which is shown without inversion. In this case the white pixels are shown as white. As seen the interior regions are now filled.

Code 3.26 Isolating two contiguous regions.

```
>>> from scipy.ndimage import  binary_propagation, label
>>> from numpy import zeros
>>> import Image, mgconvert
>>> mg = Image.open( 'tex/pil/tt2.png')
>>> data = mgconvert.i2a( mg.convert('L'))/255.0
>>> seeds = zeros( data.shape )
>>> seeds[14,20] = seeds[47,51] = 1
>>> b = binary_propagation( seeds, mask=data )
>>> lbls, cnt = label( b )
>>> cnt
2
>>> mgconvert.a2i( lbls == 1 ).show()
```



(a) **(b)**

Fig. 3.1 **a** Is the output from the **binary_propagation** function and **b** is the output from the **labels** function

The two regions do appear to be isolated but technically they are not detected. There is not a single variable that shows one T but not the other. This can be accomplished using the **label** function also from the *scipy.ndimage* model. This function receives an image with objects that are isolated by a background pixels with a 0 value. It then assigns an identifying integer value to all of the pixels in each region. In this case there are two regions and thus all of the pixels in one region will be assigned a value of 1 and all of the pixels in the other region will be assigned the value of 2. The result is shown in Fig. 3.1b.

Line 11 shows that the value of `cnt` is 2. This is the number of separate regions that are found. The matrix `lbls` contains the answer shown in Fig. 3.1b. Line 12 shows a method of isolating the first region as it sets to True all pixels that are 1 and to False all other pixels. The other shape can be found by using `lbls == 2` .

3.3 Designing in Numpy

Readers with experience with languages such as C++, Fortran, Java etc. will find it beneficial to adjust their approach to implementing an algorithm for a large array. Consider the simple equation,

$$c_{i,j} = a_{i,j} + b_{i,j}. \tag{3.3}$$

A traditional method of realizing this equation is shown in Code 3.27 which creates three large arrays in Lines 3 and 4. The matrices a and b are filled with random numbers. The function **Fun** contains the loops necessary to implement the addition. The function **time.time** prints out the current computer time with seconds. This little test program indicates that the process takes 1.641 s.

Code 3.27 Execution time for a double loop.

```
>>> from numpy import random, zeros
>>> import time
>>> a = random.ranf((1024,1024))
>>> b = random.ranf((1024,1024))
>>> c = zeros((1024,1024))
>>> def Fun( a,b ):
        for i in range( 1024):
                for j in range(1024):
                    c[i,j] = a[i,j] + b[i,j]
        return c
>>> time.time(); c = Fun(a,b); time.time()
1319970615.734
1319970617.375
```

The *NumPy* package provides compiled and optimized functions. This means that matrix operations can be performed with a greater efficiency. The same code can be written in less lines and executed with a far greater speed. The same process is performed in Code 3.28 and excepting the time stamps the equation is realized in a single command.

Code 3.28 Execution time for a single command.

```
>>> time.time(); c = a + b; time.time()
1319970623.062
1319970623.093
```

More important is that the execution time is only 0.031 s. The execution time is reduced by more than 50×. While 1.641 s is not a tremendous amount of time for a single job, it can be considerable for larger jobs. Consider a case in which the operation needs to be applied to a database of 1000 images. It would take nearly 30 min to perform 1000 iterations of Code 3.27 and just over 5 min for Code 3.28.

A rule of thumb is that nested loops are to be avoided. Python is a scripting language which means that every line of code is interpreted before it is executed. Whereas Code 3.27 is perfectly acceptable in compiled languages like C++ and Fortran, it is much less acceptable in scripting languages like Python. It does provide the correct answer but takes a lot of time to do so. An approach that a programmer may wish to use is to think in a more parallel fashion. If a 1024×1024 node computer were available Eq. 3.3 could be executed in a parallel fashion with each computer node being assigned a single pixel. In these situations, NumPy commands are available to perform the task efficiently.

Code similar in nature to 3.27 still can be useful in prototyping. In more complicated scenarios programmers may want to write nested loop coding to get all of the steps of their program working. Once working they can then search the NumPy/SciPy libraries for more efficient functions that will achieve the same answer.

Some implementations of Python using interfaces such as IDLE can become bogged down in loops. If instead of simple addition the function had several complicated steps inside of the loops then the prototype code may take a prohibitively large amount of time. In some implementations of Python it is not possible to break out of loops while they are running with Control-C. This is a feature of the interface and not of the programming language. So, a simple trick is to place a print statement in the loops. When the print statement is executed there is also a poll of the inputs and a Control-C can be executed at the time of the print statement. Such a print statement is inserted in Line 3 of Code 3.29. The comma at the end of the statement will make consecutive prints on the same line rather than new lines. The print statement in Line 3 at the onset of each iteration is executed and if there is a pending Control-C from the user then the program will terminate.

Code 3.29 Inserting a safety print statement.

```
>>> def Fun( a,b ):
        for i in range( 1024):
                print i,
                for j in range(1024):
                        # lots of nasty code goes here
        return c
```

For programs that are causing serious debugging issues the programmer may consider running them in a command line shell. For example, MS-Windows users can use Python (Command Line) instead of IDLE from the start up menus. This interface is not user friendly but it does have the ability to immediately stop a loop when the user employs Control-C.

3.4 Python Image Library

The PIL is a third-party module that provides the ability to read, write, and manipulate images. It can be obtained from http://www.pythonware.com/products/pil/. This section just shows a few of the many functions that this module offers.

3.4.1 Reading an Image

Code 3.30 shows the simplicity of reading an image. This is one of the few cases in which the module name requires a capital letter as in Line 1. The **Image.open** command from Line 2 reads the specified image into the variable `mg`. Lines 3–8 demonstrate methods of obtaining information about the image. The size of an image provides the number of pixels in the horizontal and vertical directions, unlike the dimensions of a matrix which are given vertical first and horizontal second.

Code 3.30 Loading an image.

```
>>> import Image
>>> mg = Image.open( 'figs/bird.jpg')
>>> mg.size
(653, 519)
>>> mg.mode
'RGB'
>>> mg.getpixel( (10,20) )
(123, 54, 147)
```

The `mode` of an image indicates if it is colour or grey scale. This image is an RGB which means that it is colour and the colour is represented in red, green, and blue components. Two other modes are "L" which indicates that the image is grey scale and "P" which indicates that the image uses a palette. A palette is a look-up table that contains the colours within the image.

There are many different types of images and PIL can read and write almost all popular formats. However, not all formats store the same information. For example, the JPEG format does store colour images but not perfectly as it has problems restoring very sharp edges. The GIF format uses a palette which can store up to 256 colours. Because of this limit it does not store colour photographs very well, but it does store grey scale images without flaw. The PNG and TIFF formats store the images well but do not provide the same compression as the other formats. A good rule of thumb is to use JPEG for photographs, GIF for cartoons and grey scale photographs, PNG for most images if speed and file size are not at issue, and bitmaps (BMP, TGA, PPM) if it is critical that the image not be compressed.

3.4.2 Writing an Image

Writing an image to a file also takes a single command. Code 3.31 shows this command but has a little fun first. The **mg.split** function will convert an RGB image into three grey scale images one for each of the colour components. So, in this case the `r` is a grey scale image (mode=L) but it contains the red information in `mg`. Thus, `r` will be bright where *mg* had a lot of red. The **show** command will pop up a new window and display the image. This will use the computer's default image viewer. The **merge** command creates a new image from others. In this case the new image will be an RGB image and the second argument should be a tuple (r,g,b). However, in this case the order is changed placing the green information in the red position, etc. The effect is that the image is reproduced with the wrong colours. The last two commands show the image and save the image as a PNG file. Some versions of MS-Windows may not show the image from Line 6. A simple fix is to replace the **show** command with the **save** to save the image to the hard drive.

Code 3.31 Writing an image.

```
>>> r,g,b = mg.split()
>>> r.mode
'L'
>>> r.show()
>>> mg2 = Image.merge( 'RGB', (g,b,r) )
>>> mg2.show()
>>> mg2.save( "mypix.png" )
```

3.4.3 Transforming an Image

Transforming an image from one mode to another is accomplished with the **convert** command as seen in Code 3.32. In this case the `mg` which was originally an RGB image is converted to a grey scale. The PIL contains image altering routines as well. Three examples are shown in Code 3.33 which show an image after a 20

Code 3.32 Converting an image.

```
>>> mg3 = mg.convert('L')
>>> mg3.mode
'L'
```

Code 3.33 Other transformations.

```
>>> mg.rotate(20).show()
>>> mg.resize( (100,100)).show()
>>> mg.crop( (10,20,100,200) ).show()
```

3.5 Summary

The NumPy package provides a very powerful set of tools to allow Python to manipulate vectors, matrices and tensors. These tool set is compiled in a very efficient manner and so the speed of using NumPy is on par with C or Matlab. The speed of using NumPy packages as opposed to using straight Python scripts can be more than a hundred fold improvement and require less script writing. Learning NumPy tools is well worth the time and effort. The SciPy package offers a large array of scientific functions of which only a few are used in the following applications. The Python Image Library provides the ability to read and write image files as well as several image manipulation tools.

Chapter 4
The PCNN and ICM

In this section two digital models evolved from biological cortical models will be presented. The first is the Pulse-Coupled Neural Network (PCNN) which for many years was the standard model for many image processing applications. The PCNN is based solely on the Eckhorn model but there are many other cortical models that exist. These models all have a common mathematical foundation, but beyond the common foundation each also had unique terms. Since the goal here is to build image processing routines and not to exactly simulate the biological system a new model was constructed. This model contained the common foundation without the extra terms and is therefore viewed as the intersection of several cortical models and it is named the Intersecting Cortical Model (ICM).

4.1 The PCNN

The Pulse-Coupled Neural Network is to a very large extent based on the Eckhorn model except for a few minor modifications required by digitisation. The early experiments demonstrated that the PCNN could process images such that the output was invariant to images that were shifted, rotated, scaled, and skewed. Subsequent investigations determined the basis of the working mechanisms of the PCNN and led to its eventual usefulness as an image-processing engine.

4.1.1 Original Model

A PCNN neuron shown in Fig. 4.1 contains two main compartments: Feeding and Linking. Each of these communicates with neighbouring neurons through the synaptic weights **M** and **W** respectively. Each retains its previous state altered by a decay factor. Only the Feeding compartment receives the input stimulus, **S**. The values of

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_4, © Springer-Verlag Berlin Heidelberg 2013

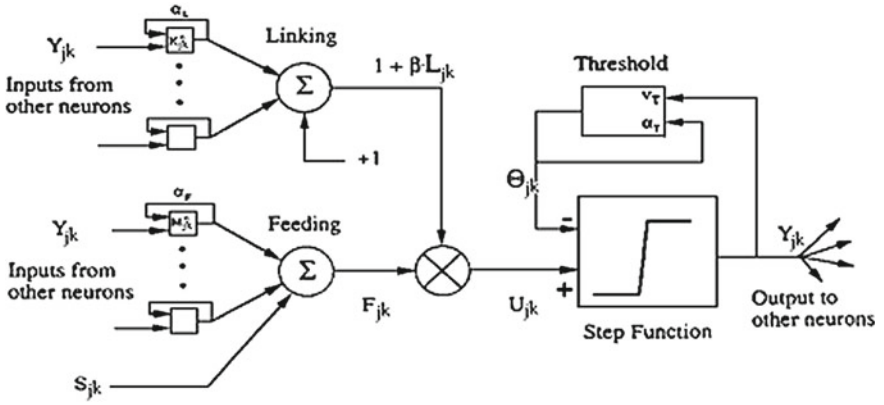


Fig. 4.1 Schematic representation of a PCNN processing element

these two compartments are determined by,

$$F_{ij}[n] = e^{\alpha_F \delta_n} F_{ij}[n-1] + S_{ij} + V_F \sum_{kl} M_{ijkl} Y_{kl}[n-1], \tag{4.1}$$

$$L_{ij}[n] = e^{\alpha_L \delta_n} L_{ij}[n-1] + V_L \sum_{kl} W_{ijkl} Y_{kl}[n-1], \tag{4.2}$$

where F_{ij} is the Feeding compartment of the (i, j) neuron embedded in a two-dimensional array of neurons, and L_{ij} is the corresponding Linking compartment. The Y_{kl}'s are the outputs of neurons from a previous iteration $[n-1]$. Both compartments have a memory of the previous state which decays in time by the exponent term. The constants V_F and V_L are normalising constants. If the receptive fields of **M** and **W** change then these constants are used to scale the resultant correlation to prevent saturation.

The state of these two compartments are combined in a second order fashion to create the internal state of the neuron, **U**. The combination is controlled by the linking strength, β. The internal activity is calculated by,

$$U_{ij}[n] = F_{ij}[n]\left(1 + \beta L_{ij}[n]\right). \tag{4.3}$$

The internal state of the neuron is compared to a dynamic threshold, $\boldsymbol{\Theta}$, to produce the output, **Y**, by

$$Y_{ij}[n] = \begin{cases} 1 & \texttt{if } U_{ij}[n] > \Theta_{ij}[n-1] \\ 0 & \texttt{Otherwise} \end{cases}. \tag{4.4}$$

The threshold is dynamic in that when the neuron fires ($Y_{ij} > \Theta_{ij}$) the threshold then significantly increases its value. This value decays until the neuron fires again.

The progression of the neuron thresholds is controlled by

$$\Theta_{ij}[n] = e^{\alpha_\Theta \delta_n} \Theta_{ij}[n-1] + V_\Theta Y_{ij}[n], \tag{4.5}$$

where V_Θ is a large constant that is generally more than an order of magnitude greater than the average value of U_{ij}.

The PCNN consists of an array (usually rectangular) of these neurons. Communications, **M** and **W** are traditionally local and Gaussian, but this is not a strict requirement. Initially, the elements of the arrays, **F**, **L**, **U**, and **Y** are all set to zero. The values of the Θ_{ij} elements are initially 0 or some larger value depending upon the user's needs. Each neuron that has any stimulus will fire in the initial iteration, which, in turn, will create a large threshold value. It will then take several iterations before the threshold values decay enough to allow the neuron to fire again. The algorithm consists of iteratively computing Eqs. (4.1)–(4.5) until the user decides to stop. In a case where the input image has no pixels that are 0 then $Y_{ij} = 1, \forall i, j$ and $Y_{ij}[k] = 0, \forall k = 1, \ldots, \approx 6$. All neurons fire in the first pulse image and it takes a few iterations before the thresholds are low enough for the neurons to fire again. Circumvention is accomplished by setting initial threshold value high (e.g., $\Theta_{ij}[0] = 1, \forall i, j$). These initial pulse images contain very little information.

4.1.2 Implementing in Python

Implementing the PCNN in a scripting language like Python is quite easy and straightforward. Only one line of script is required to implement each equation (4.1)–(4.5). Since it is possible that a problem may employ more than one PCNN an object-oriented approach is adopted. Code 4.1 shows the first part of the file *pcnn.py* which defines the PCNN class and the constructor `__init__`.

Code 4.1 Part 1 of *pcnn.py*.

```
# pcnn.py
from numpy import zeros
from scipy.signal import cspline2d
class PCNN:
    f,l,t1,t2, beta = 0.9, 0.8, 0.8, 50.0, 0.2
    # constructor
    def __init__ (self,dim):
        self.F = zeros( dim,float)
        self.L = zeros( dim, float)
        self.Y = zeros( dim,float)
        self.T = zeros( dim,float) + 0.0001
```

Line 5 establishes the constants in the PCNN where f in the code represents $e^{\alpha_F \delta_n}$ in Eq. (4.1). Likewise, $l \leftarrow e^{\alpha_L \delta_n}$, $t1 \leftarrow e^{\alpha_\Theta \delta_n}$, and $t2 \leftarrow V_\Theta$. The constants V_F and V_L are built into the correlation kernels, and $beta \leftarrow \beta$. The constructor receives the variable `dim` which is the dimensions of the input array and is not restricted to $\mathcal{R}^2$ space. The `dim` is a *tuple* and in the case of image data the dimensions are given as (V, H) where V is the vertical dimension and H is the horizontal dimension. Lines 8–11 allocate space for the matrices required in the calculations.

A PCNN iteration is scripted in the function **Iterate** shown in Code 4.2. It receives an input `stim` which is **S** in Eq. (4.1). If there have been previous pulses then the program uses Line 4 which emulates the connection matrices **M** and **W** with the function **cspline2d** from *scipy.signal*. The **cspline2d** function acts as a smoothing operator which is essentially similar to the behaviour of **M** and **W** being Gaussian connections.

Code 4.2 Part 2 of *pcnn.py*.

```
# pcnn.py
    def Iterate (self,stim):
        if self.Y.sum() > 0:
            work = cspline2d(self.Y.astype(float),90)
        else:
            work = zeros(self.Y.shape,float)
        self.F = self.f * self.F + stim + 8*work
        self.L = self.l * self.L + 8*work
        U = self.F * (1 + self.beta * self.L )
        self.Y = U > self.T
        self.T = self.t1 * self.T + self.t2 * self.Y + 0.1
```

Lines 7–11 are the five PCNN equations. The function has no output but `self.Y` has become the pulses for all neurons and it is accessed by scripts that drive this program. However, before this program can be driven it is necessary to create a simple input.

In this simple case, the input constructed in Code 4.3 creates a 360×360 frame and centred in this frame is a solid 'T'.

Code 4.3 Creating an image of a 'T'.

```
>>> data = zeros( (360,360) )
>>> data[120:160, 120:240] = 1
>>> data[160:240, 160:200] = 1
```

Code 4.4 shows the script to drive the PCNN. The pulse images are collected in the list `Y`. Line 6 may seem a bit odd with the `net.Y + 0`, but this is simply a

manner that creates a duplicate array in the computer memory instead of just making a copy of the pointer to the data. Since the threshold at $n = 0$ is very low the ON pixels from the input are also the neurons that pulse and so Fig. 4.2a also represents the output of the network at $n = 0$. The results are shown in Fig. 4.2 with the original being shown in Fig. 4.2a. In this case the images are inverse of the actual input where in the printed page the black pixels are ON and the white pixels are OFF.

Code 4.4 Driver for the PCNN.

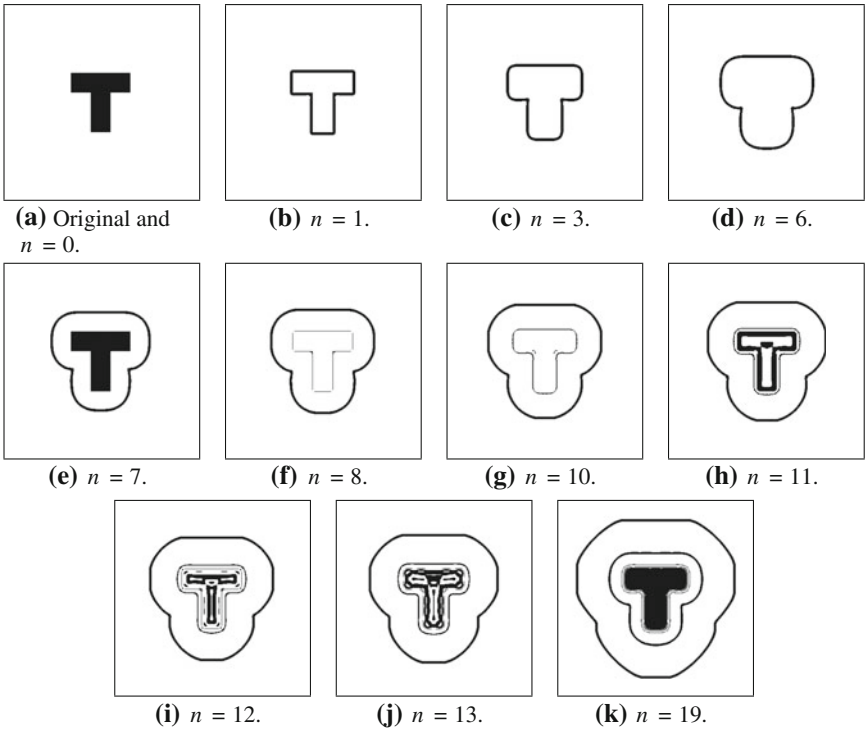
```
>>> import pcnn, mgconvert
>>> net = pcnn.PCNN( data.shape )
>>> Y = []
>>> for i in range( 20 ):
      net.Iterate( data )
      Y.append( net.Y + 0 )
>>> mgconvert.a2i( Y[0] ).show()
>>> mgconvert.a2i( Y[0] ).save('y0.png')
```

The neurons that pulse at $n = 0$ obtain a very high threshold from Eq. (4.5) and therefore they are not able to pulse again for many iterations. However, the pixels neighbouring those pixels are encouraged to pulse through the connections **M** and **W**. Therefore, the neurons along the perimeter pulse. This perimeter continues to expand as demonstrated at $n = 3$ and $n = 6$. At $n = 7$ the threshold of the on-target pixels have decayed according to the first term in Eq. (4.5) and now the internal energy again surpasses the threshold and the neurons pulse again (second cycle). As shown in the rest of the sampled images the waves are generated and continue to expand from the original target perimeter.

Line 8 in Code 4.4 saves the first pulse image as a PNG file. It should be noted that some image compression formats are inappropriate for storing a pulse image. Namely, the JPEG format does not store very sharp edges well and therefore will induce noise into the stored image. Users should use PNG, GIF or TIF to store pulse images.

4.1.3 Spiking Behaviour

Consider the activity of a single neuron. It is receiving some input stimulus, **S**, and stimulus from neighbours in both the Feeding and Linking compartments. The internal activity rises until it becomes larger than the threshold value. Then the neuron fires and the threshold sharply increases then begins its decay until once again the internal activity becomes larger than the threshold. This process gives rise to the pulsing nature of the PCNN. Figure 4.3 displays the states within a single neuron embedded in a 2D array as it progresses in time.



(a) Original and $n = 0$. **(b)** $n = 1$. **(c)** $n = 3$. **(d)** $n = 6$.

(e) $n = 7$. **(f)** $n = 8$. **(g)** $n = 10$. **(h)** $n = 11$.

(i) $n = 12$. **(j)** $n = 13$. **(k)** $n = 19$.

Fig. 4.2 Figure **a** shows the input image which is also the same as the output at $n = 0$. Figures **b**–**k** shows the pulse images at selected informative values of n

Figure 4.2 shows an expanding wave for each pulse cycle. These waves do not reflect or refract. Furthermore, colliding wavefronts are annihilated. Such waves have been observed in nature [10, 74] and are called *autowaves*.

Consider the image in Fig. 4.4. The original input consists of two 'T's. The intensity of each 'T' is constant, but the intensities of each 'T' differ slightly. At $n = 4$ the neurons that receive stimulus from either of the 'T's will pulse in step $n = 16$ (denoted as black). As the iterations progress, the autowaves emanate from the original pulse regions. At $n = 10$ it is seen that the two waves did not pass through each other. At $n = 12$ the more intense 'T' again pulses.

The network also exhibits some synchronising behaviour. In the early iterations segments tend to pulse together. However, as the iterations progress, the segments tend to de-synchronise. Synchronicity occurs by a pulse capture, which occurs when one neuron is close to pulsing ($U_{ij} < \Theta_{ij}$) and its neighbour fires. The additional input from the neighbour will provide an additional input to U_{ij} thus allowing the neuron to fire prematurely. The two neurons, in a sense, synchronise due to their linking communications.

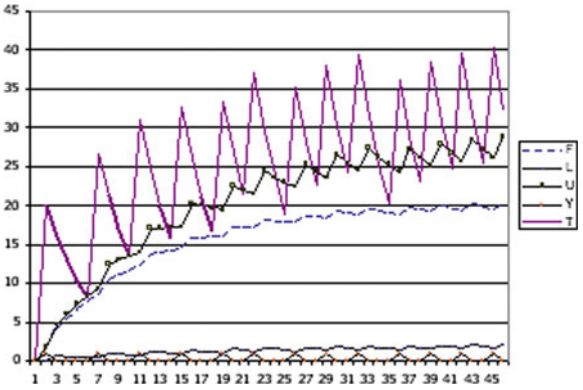


Fig. 4.3 An example of the progression of the states of a single neuron. See the text for explanation of **L**, **U**, **T** and **F**

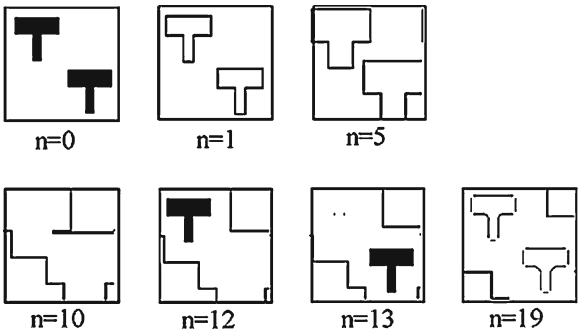


Fig. 4.4 A typical PCNN example

The de-synchronisation occurs in more complex images due to residual signals. As the network progresses the neurons begin to receive information indirectly from other non-neighbouring neurons. This alters their behaviour and the synchronicity begins to fail. The beginning of this failure can be seen by comparing $n = 1$ to $n = 19$ in Fig. 4.4. Note that the corners of the 'T' autowave are missing in $n = 19$. This phenomenon is more noticeable in more complicated images. Gernster [33] argues that the lack of noise in such a system is responsible for the de-synchronisation. However, applications demonstrated in subsequent chapters specifically show the PCNN architecture does not exhibit this link. Synchronisation has been explored more thoroughly for similar integrate and fire models [72].

The PCNN has many parameters that can be altered to adjust its behaviour. The (global) linking strength, β, in particular, has many interesting properties (in particular effects on segmentation). While this parameter, together with the two weight matrices, scales the feeding and linking inputs, the three potentials, V, scale the internal signals. Finally, the time constants and the offset parameter of the firing threshold

are used to adjust the conversions between pulses and magnitudes. The extent of the range of **M** and **W** directly affects the speed that the autowave travels by allowing the neurons to communicate with neurons farther away and thus allowing the autowaves to advance farther in each iteration.

The pulse behaviour of a single neuron is greatly affected by α_Θ and V_Θ. The α_Θ affects the decay of the threshold value and the V_Θ affects the height of the threshold increase after the neuron pulses. It is quite possible to force the neuron to enter into a multiple pulse regime in which the neuron pulses in consecutive iterations by lowering V_Θ. The autowave created by the PCNN is greatly affected by V_F. Setting V_F to 0 prevents the autowave from entering any region in which the stimulus is also 0. There is also a range of V_F values that allows the autowave to travel but only for a limited distance.

There are architectural changes that can alter the PCNN behaviour. One such alteration is quantised linking where the linking values are either 1 or 0 depending on a local condition. In this system the Linking field is computed by,

$$L_{ij}[n] = \begin{cases} 1 & \texttt{if } \sum_{kl} w_{ijkl} Y_{kl} > \gamma \\ 0 & \texttt{Otherwise} \end{cases}. \tag{4.6}$$

Quantized linking tends to keep the autowaves clean. In the previous system autowaves travelling along a wide channel have been observed to decay about the edges. In other words a wave front tends to lose its shape near its outer boundaries. Quantized linking has been observed to maintain the wave fronts shape. As stated earlier, setting $\Theta_{ij}[0] = 1, \forall i, j$ bypasses early uninformative pulse images. De-synchronisation may be controlled by resetting Θ after a few iterations [80].

The influence of a dynamic V_Θ was considered by Yamaguchi et al. [117] in which a static V_Θ was replaced by,

$$V_\Theta = 1 + a_0 \sin(\omega_0 t), \tag{4.7}$$

where a_0 and ω_0 were the variables that they altered. In a simulation of a one-dimensional network they demonstrated that certain combinations of a_0 and ω_0 produced a stable case in which two neurons receiving the same input pulse in synchrony, but other combinations produced a chaotic behaviour.

4.1.4 Collective Behaviour

Code 4.5 runs the PCNN again and collects neural activity for each compartment. This allows for visual inspection of the collective behaviour of the neurons. Figure 4.5 shows the activity of the **F** compartments of each neuron. At $n = 0$ only the neurons that are receiving a stimulus have a significant value since the only active part of Eq. (4.1) is the stimulus **S**.

Code 4.5 Collecting the internal neural activities.

```
>>> net = pcnn.PCNN( data.shape )
>>> F, L, U, Y, T = [],[],[],[],[]
>>> for i in range(50):
        net.Iterate( data )
        F.append( net.F+0 )
        L.append( net.L + 0 )
        U.append( net.F * (1 + net.beta * net.L ) )
        Y.append( net.Y + 0 )
        T.append( net.T + 0 )
```

(a) $n = 0$. **(b)** $n = 1$. **(c)** $n = 11$. **(d)** $n = 12$. **(e)** $n = 49$.

Fig. 4.5 Collective behaviour of **F**

(a) $n = 1$. **(b)** $n = 11$. **(c)** $n = 12$. **(d)** $n = 49$.

Fig. 4.6 Collective behaviour of **L**

At $n = 1$ the first and third terms in Eq. (4.1) begin to contribute to the activity in **F**. Basically, the autowave appears and continues to expand up to $n = 11$. In this example, at $n = 12$ the second cycle begins and the values in **F** are increased for the neurons that fire. The image shown is scaled and therefore intensities shown on the page between $n = 11$ and $n = 12$ do not have the same magnitude. The last example shown is at $n = 49$ which shows three cycles of waves expanding. Figure 4.6 shows the intensities of the **L** compartments with $n = 0$ not shown since there is no activity in the first iteration. While the expanding autowave is present it is also noticed that the **L** compartment is more sensitive to edges.

The threshold $\boldsymbol{\Theta}$ responses are shown in Fig. 4.7. These have larger values when a neuron fires. Clearly, at $n = 12$ the expanding wave is shown to have decreasing values closer to the target since those neurons fired further back in time. The output arrays are shown in Fig. 4.2.

(a) $n = 0$. **(b)** $n = 1$. **(c)** $n = 11$. **(d)** $n = 12$. **(e)** $n = 49$.

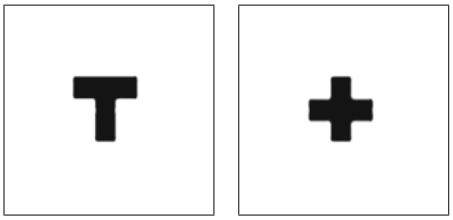
Fig. 4.7 Collective behaviour of Θ

4.1.5 Time Signatures

The early work of Johnson [46] was concerned with converting the pulse images to a single vector of information. This vector, **G**, has been called the 'time signal' and is computed by,

$$G[n] = \sum_{ij} Y_{ij}[n]. \tag{4.8}$$

This time signal was shown to have an invariant nature with regard to alterations of the input image. For example, consider the two images in Fig. 4.8. These images are of a T and a +. Each image was presented to the PCNN and each produced a time signal, $\mathbf{G}_T$ and $\mathbf{G}_+$, respectively. These are shown in Fig. 4.9.



(a) Original and $n = 0$. **(b)** $n = 1$.

Fig. 4.8 Johnson's inputs of a 'T' and '+'

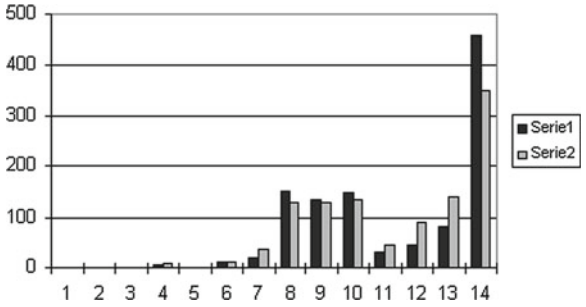


Fig. 4.9 A plot of $\mathbf{G}_T$ (series 1) and $\mathbf{G}_+$ (series 2) in arbitrary units (vertical axis). The horizontal axis shows the frame number and the vertical axis the values of **G**

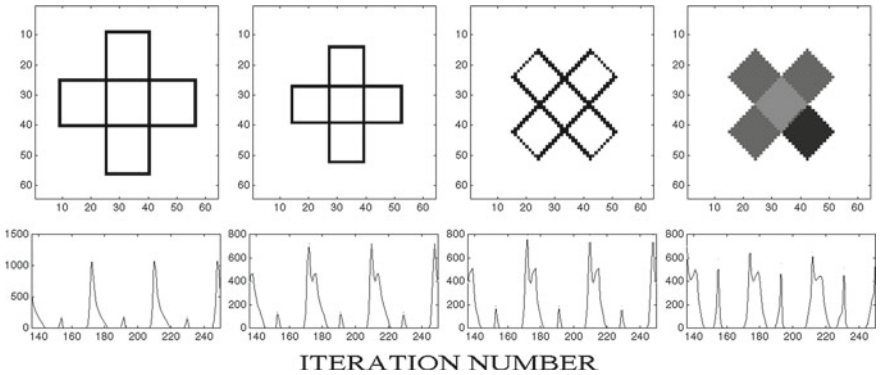


Fig. 4.10 Plot of G for a slightly more complicated cross than in Fig. 4.9. The cross is then scaled and rotated and filled with shades of *grey* to show what happens to the time series

Johnson showed that the time signal produces a cycle of activity in which each neuron pulses once during the cycle. The two plots in Fig. 4.9 depict single cycles of the 'T' and the '+'. As time progressed the pattern within the cycle stabilised for these simple images. The content of the image could be identified simply by examining a very short segment of the time signal—a single stationary cycle. Furthermore, this signal was invariant to large changes in rotation, scale, shift, or skew of the input object. Figure 4.10 shows several cycles of a slightly more complicated input and how the peaks vary with scaling and rotation as well as intensities in the input image. However, note that the distances between the peaks remain constant, providing a fingerprint of the actual figure. Furthermore, the peak intensities could possibly be used to obtain information on scale and angle.

4.1.6 Neural Connections

The PCNN contains two convolution kernels **M** and **W**. The original Eckhorn model used a Gaussian type of interconnections, but when the PCNN is applied to image processing problems these interconnections are available to the user for altering the behaviour of the network. The few examples shown here all use local interconnections. It is possible to use long range interconnections but two impositions arise. The first is that the computational load is directly dependent upon the number of interconnections. The second is that PCNN tests to date have not provided any meaningful results using long range interconnections, although long range inhibitory connections of similar models have been proposed in similar cortical models [76].

Subsequent experiments replaced the interconnect pattern with a target pattern in the hope that on-target neurons would pulse more frequently. The matrices **M** and **W** were similar to the intensity pattern of a target object. In actuality there was very little difference in the output from this system than from the original PCNN.

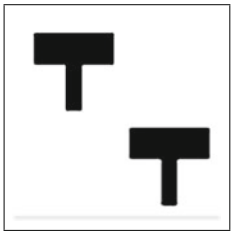


Fig. 4.11 An example input

Further investigations revealed the reason for this. Positive interconnections tend to smooth the image and longer-range connections provide even more smoothing. The internal activity of the neuron may be quite altered by a change in interconnections. However, much of this change is nullified since the internal activity is compared to a dynamic threshold. The amount by which the internal activity surpasses the dynamic threshold is not important and thus the effects of longer-range interconnections are reduced.

Manipulations of a small number of interconnections do, however, provide drastic changes in the PCNN. A few examples of these are shown. For these examples we use the input shown in Fig. 4.11. This input is a set of two 'T's.

The first example computes the convolution kernel by,

$$K_{ij} = \begin{cases} 0 & \texttt{if } i = m \texttt{ and } j = m \\ 1/r & \texttt{Otherwise} \end{cases}, \tag{4.9}$$

where r is the distance from the centre element to element ij, and m is half of the linear dimension of K. In this test K was 5×5. Computationally, the feeding and linking equations are,

$$F_{ij}[n] = e^{\alpha_F \delta_n} F_{ij}[n-1] + S_{ij} + (\mathbf{K} \otimes \mathbf{Y})_{ij}\,, \tag{4.10}$$

and

$$L_{ij}[n] = e^{\alpha_L \delta_n} L_{ij}[n-1] + (\mathbf{K} \otimes \mathbf{Y})_{ij}\,, \tag{4.11}$$

where $\otimes$ represents the convolution operator.

The resultant outputs of the PCNN are shown in Fig. 4.12. The output first pulses all neurons receiving an input stimulus. Then autowaves are established that expand from the original pulsing neurons. These autowaves are two pixels wide since the kernel extends two elements in any direction from the centre. These autowaves expand at the same speed in both vertical and horizontal dimensions again due to the symmetry of the kernel.

Setting the elements of the previous kernel to zero for $i = 0$ and $i = 4$ defines a kernel that is asymmetric. This kernel will cause the autowaves to behave in a slightly

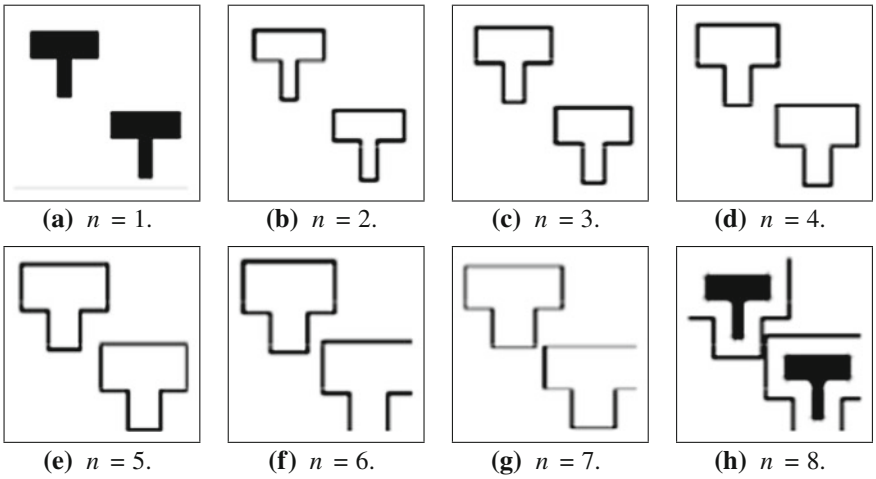


Fig. 4.12 Output pulse images

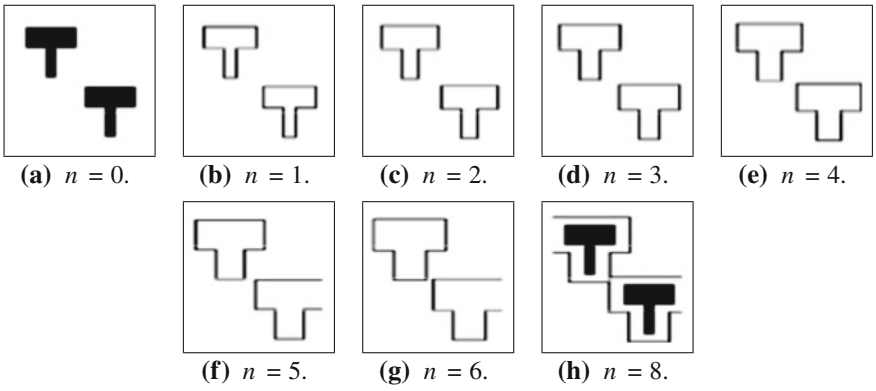


Fig. 4.13 Outputs of a PCNN with an asymmetric kernel, as discussed in the text. These outputs should be compared to those shown in Fig. 4.14

different fashion. The results from these tests are shown in Fig. 4.13. The autowave in the vertical direction now travels at half the speed of the one in the horizontal direction. Also the second pulse of the neurons receiving stimulus is delayed a frame. This delay is due to the fact that these neurons were receiving less stimulus from their neighbours. Increasing the values in K could eliminate the delay.

The final test involves altering the original kernel by simply requiring that,

$$K_{ij} = \begin{cases} K_{ij} & \texttt{if } i = m \texttt{ and } j = m \\ -K_{ij} & \texttt{Otherwise} \end{cases}. \tag{4.12}$$

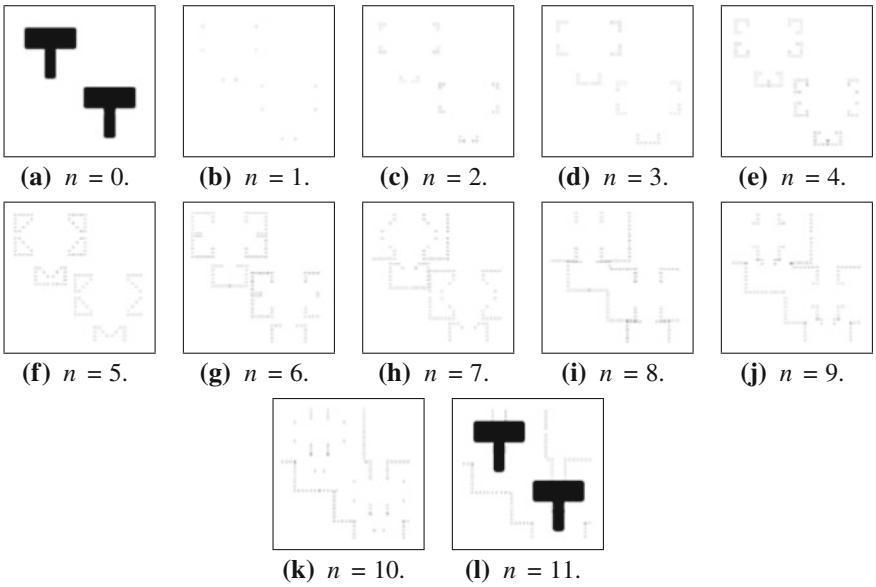


Fig. 4.14 Outputs of a PCNN with an on-centre/off-surround kernel

The kernel now has a positive value at the centre and negative values surrounding it. This configuration is termed On-Centre/Off-Surround. Such configurations of interconnections have been observed in the eye. Furthermore, convolutions with a zero-mean version of this function are quite often used as an "edge enhancer". Employing this type of function in the PCNN has a very dramatic effect on the outputs as is shown in Fig. 4.14. The autowaves created by this system are now dotted lines. This is due to *competition* amongst the neurons since each neuron is now receiving both positive and negative inputs.

4.1.7 Fast Linking

The PCNN is a digital version of an analogue process and this quantisation of time does have a detrimental effect. Fast linking was originally installed to overcome some of the effects of time quantisation and has been discussed by McEniry and Johnson [71] and by Johnson and Padgett [47]. This process allows the linking wave to progress a lot faster than the feeding wave. Basically, the linking is allowed to propagate through the entire image for each iteration. This system allows the autowaves to fully propagate during each iteration. In the previous system the progression of the autowaves was restricted by the radius of the convolution kernel.

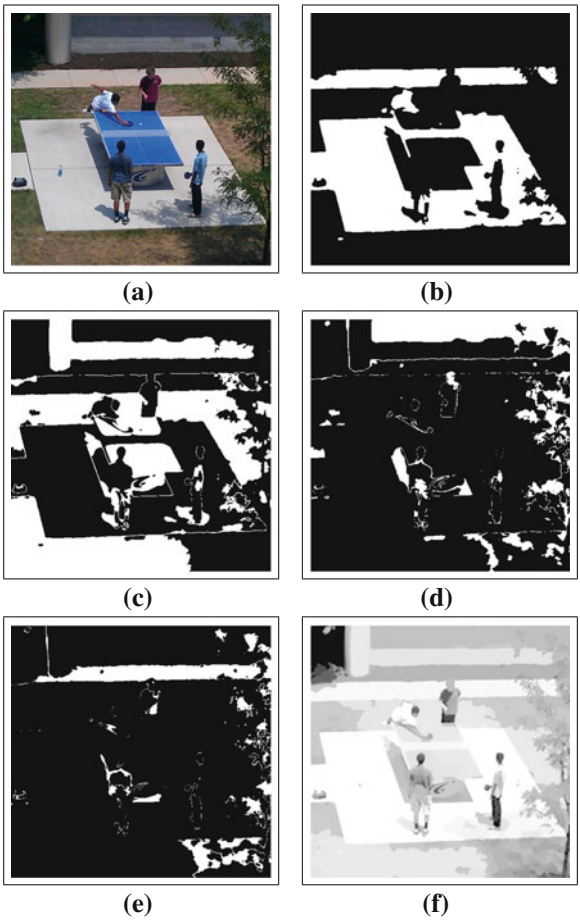


Fig. 4.15 An input and fast linking outputs. The pulses are shown as *white* pixels

Fast linking iterates the **L**, **U**, and **Y** equations until **Y** becomes static. The equations for this system are similar to Eqs. (4.1)–(4.5). A single iteration of the process is:

1. Compute a single iteration of Eqs. (4.1)–(4.5).
2. Repeat
   a. Compute

$$L_{ij}[n] = e^{\alpha_L \delta_n} L_{ij}[n-1] + V_L \sum_{kl} W_{ijkl} Y_{kl}[n-1]$$

   b. Compute Eqs. (4.3) and (4.4).
3. Until **Y** does not change.
4. Compute (4.5).

Codes 4.1 and 4.2 show the functions for generating the PCNN class. The fast linking is accomplished by appending a new function, **FastLIterate**, to the PCNN class. This function is shown in Code 4.6. There are a few changes from the **Iterate** function shown in Code 4.2. The first is that there is an internal loop starting at Line 10 the recursively computes Eqs. (4.2)–(4.4). The loop terminates when the variable `ok` becomes False which occurs when the changes between consecutive `Y` arrays are very similar (less than 100 pixels different). Once this is complete Eq. (4.5) is computed and the single PCNN iteration is complete.

The image in Fig. 4.15a is an original image with four young lads playing a game. The image is interesting in that the men are shown in darker pixels than the background. Traditional PCNN networks have no problems in generating pulses for the bright objects, but due to interference (Sect. 4.2.3.1) the darker objects are generally not collectively pulsed. The first few pulse images are shown in Fig. 4.15 and as seen there are no frames in which all of the pixels of a person pulse. Figure 4.15f shows a weighted collective behaviour of the pulses. The image was generated by,

Code 4.6 Fast linking iteration for *pcnn.py*.

```
# pcnn.py
    def FastLIterate( self, stim ):
        ok = 1
        self.Y = old + 0
        if self.Y.sum() > 0:
            work = cspline2d(self.Y.astype(float),90)
        else:
            work = zeros(self.Y.shape,float)
        self.F = self.f * self.F + stim + 8*work
        while ok:
            print '.',
            if self.Y.sum() > 0:
                work = cspline2d(self.Y.astype(float),90)
            else:
                work = zeros(self.Y.shape,float)
            self.L = self.l * self.L + work
            U = self.F * (1 + self.beta * self.L )
            old = self.Y + 0
            self.Y = np.logical_or( U > self.T, self.Y )
            if abs(self.Y - old).sum() <100: ok = 0
        self.T = self.t1 * self.T + self.t2 * self.Y + 0.1
```

$$A = \sum_{n=1}^{N} 0.95^n Y[n], \tag{4.13}$$

where N is the number of pulse images. Bright regions are associated with regions that pulsed early.

Code 4.7 shows the steps for generating the PCNN iterations and the output shown in Fig. 4.15f. The image was loaded in Line 1 and converted to grey scale in Line 2. Line 3 performs a small smoothing on the image to reduce noise. Line 4 creates an empty list which is populated in the iterations from Lines 5 to 7. The rest of the code is dedicated to creating the composite pulse image. The function **FastLIterate** does have a print statement (Line 11 in Code 4.6) which is used for stopping the iterations. In some operating systems the IDLE environment will not break out of a loop with Ctl-C unless a print statement is called. Thus, if the user decides to stop the iteration it will stop when it reaches Line 11.

4.1.8 Models in Analogue Time

As stated earlier the PCNN is a simulation in discrete time of a biological system that operates in analogue time. This is due solely to the ease of computation in discrete time. It is possible to more closely emulate an analogue time system. Computationally, this is performed by keeping a table of events. These events include the time in which each neuron is scheduled to pulse and when each inter-neural communication reaches its destination. This table is sorted according to the scheduled time of each event.

The system operates by considering the next event in the table. This event is computed and it either fires a neuron or modifies the state of a neuron because a communication from another neuron has reached this destination. All other events that are affected by this event are updated. For example, if a communication reaches its destination then it will alter the time that the neuron is predicted to pulse next. Also new events are added to the table.

Code 4.7 Executing a fast linking.

```
>>> mg = Image.open( 'pingpong.png')
>>> data = mgconvert.i2a( mg.convert('L'))/255.0
>>> data = cspline2d( data, 2 )
>>> Y = []
>>> for i in range( 30 ):
        net.FastLIterate( data )
        Y.append( net.Y + 0 )
>>> A = np.zeros( data.shape )
>>> k = 1.0
>>> for i in range( 30 ):
        A += k*Y[i]
        k *= 0.95
```

For example, if a neuron pulses then it will generate new communications that will eventually reach their destinations.

More formally, the system is defined by a new set of equations. The stimulus is **U** and it is updated via,

$$\mathbf{U}(t+dt) = e^{-dt/\tau_U}\mathbf{U}(t) + \beta\mathbf{U}(t) \otimes \mathbf{K}, \tag{4.14}$$

where **K** defines the inter-neural communications, $\otimes$ is the convolution operator and β is an input scaling factor. The neurons fire when a nonlinear condition is met,

$$Y_{ij}(t+dt) = \begin{cases} 1 & \texttt{if } (\beta\mathbf{U} \otimes \mathbf{K})_{ij} > \Theta_{ij} \\ 0 & \texttt{Otherwise} \end{cases}, \tag{4.15}$$

and the threshold is updated by,

$$\mathbf{\Theta}(t+dt) = e^{-dt/\tau_\Theta}\mathbf{\Theta}(t) + \gamma\mathbf{Y}(t). \tag{4.16}$$

The effect is actually an improvement over the digital system, but the computational costs are significant. Figure 4.16 displays an input and the neural pulses. In order to display the pulses it is necessary to collect the pulses over a finite period of time, so even though they are displayed together the pulses in each frame could occur at slightly different times.

4.2 The ICM

As stated earlier there are several biological models that have been proposed. These models are mathematically similar to the Fitzhugh-Nagumo system in that each neuron consists of coupled oscillators. When the goal is to create image processing

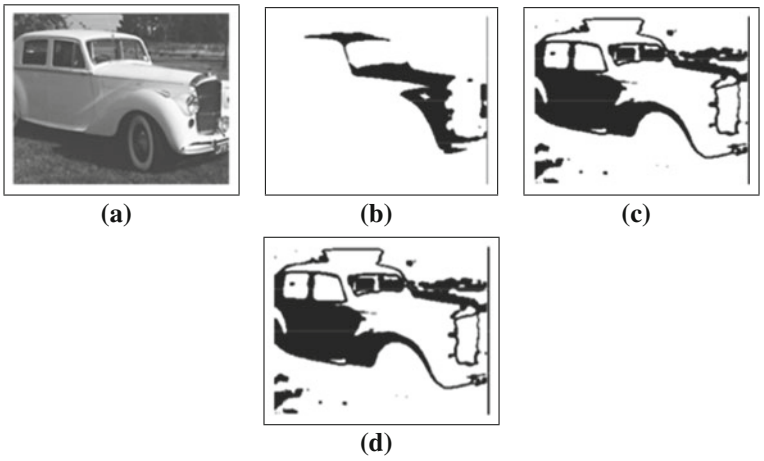


Fig. 4.16 An original image and collections of neural pulses over finite time windows

applications it is no longer necessary to exactly replicate a biological system. The important contribution of the cortical model is to extract information from the image and there is little concern as to the deviation from any single biological model.

The ICM is a model that attempts to minimize the cost of calculation but maintain the effectiveness of the cortical model when applied to images. Its foundation is based on the common elements of several biological models.

4.2.1 Minimum Requirements

Each neuron must contain at least two coupled oscillators, connections to other neurons, and a non-linear operation that determines decisively when a neuron pulses. In order to build a system that minimizes the computation it must first be determined which operation creates the highest cost. In the case of the PCNN almost all of the cost of computation stems from the interconnection of the neurons. In many implementations users set $\mathbf{M} = \mathbf{W}$ which would cut the computational needs in half. One method of reducing the costs of computation is to replace the traditional Gaussian type connections.

Another method is to reduce the number of connections. What is the minimum number of neurons required to make an operable system? This question is answered by building a minimal system and then determining if it created autowave communications between the neurons [54]. Consider the input image in Fig. 4.17 which contains two basic shapes.

The system that is developed must create autowaves that emanate from these two shapes. So, a model was created that connected each neuron to P other neurons. Each neuron was permanently connected to P random nearest neighbours and the simulation was allowed to run several iterations. The results in Fig. 4.18 display the results of three simulations. In the first $P = 1$ and the figure displays which neurons pulsed during the first 10 iterations. After 10 iterations this system stabilised. In other words the autowave stalled and did not expand. In the second test $P = 2$ and again the autowave did not expand. In both of these cases it is believed that the system had insufficient energy to propagate the communications between the neurons. The third test used $P = 3$ and the autowave propagated through the system, although due to the minimal number of connections this propagation was not uniform. In the image it is seen that the autowaves from the two objects did collide only when $P = 3$.

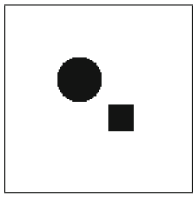


Fig. 4.17 An input image

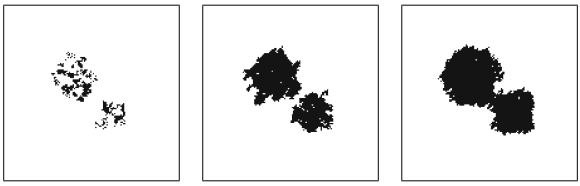


Fig. 4.18 Neuron that fired in the first 10 iterations for systems with $P = 1$, $P = 2$, and $P = 3$

The conclusion is that at least three connections between neurons are needed in order to generate an autowave. However, for image processing applications the imperfect propagation should be avoided as it will artificially discriminate the importance of parts of the image over others.

Another desire is that the autowaves emanate as a circular wave front rather than a square front. If the system only contained 4 connections per neuron then the wave would propagate in the vertical and horizontal directions but not along the diagonals. The propagation from any solid shape would eventually become a square and this is not desired. Since the input image will be defined as a rectangular array of pixels the creation of a circular autowave will require more neural connections. This circular emanation can be created when each neuron is connected to two layers of nearest neighbours. Thus, $P = 24$ seems to be the minimal system.

4.2.2 ICM Theory

The minimal system now consists of two coupled oscillators, a small number of connections, and a non-linear function. This system is described by the following three equations [55],

$$F_{ij}[n+1] = f F_{ij}[n] + S_{ij} + W\{\mathbf{Y}[n]\}_{ij}, \tag{4.17}$$

$$Y_{ij}[n+1] = \begin{cases} 1 & \texttt{if } F_{ij}[n+1] > \Theta_{ij}[n] \\ 0 & \texttt{Otherwise} \end{cases}, \tag{4.18}$$

and

$$\Theta_{ij}[n+1] = g\Theta_{ij}[n] + hY_{ij}[n+1]. \tag{4.19}$$

Here the input array is $\mathbf{S}$, the state of the neurons are contained in $\mathbf{F}$, the outputs are $\mathbf{Y}$, and the dynamic threshold states are $\mathbf{\Theta}$. The scalars f and g are both less than 1.0 and $g < f$ is required to ensure that the threshold eventually falls below the state and the neuron pulses. The scalar h is a large value that dramatically increases the threshold when the neuron fires. The connections between the neurons are described

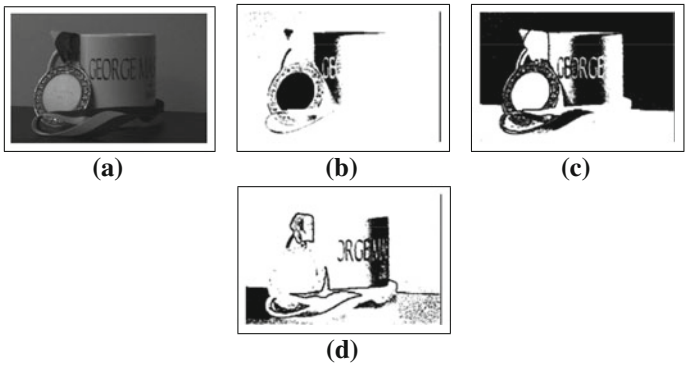


Fig. 4.19 **a** is an input image and **b–d** are a few of the pulse outputs from the ICM

by the function W and for now these are still the $1/r$ type of connections. A typical example is show in Fig. 4.19.

Distinctly the segments inherent in the input image are displayed as pulses. This system behaves quite similar to the PCNN and is done so with simpler equations. Comparisons of the PCNN and the ICM operating on the same input are shown in Figs. 4.20 and 4.21.

Certainly, the results do have some differences, but it must be remembered that the goal is to develop an image processing system. Thus, the results that are desired from these systems is the extraction of important image information. It is desired to have the pulse images display the segments, edges and textures that are inherent in the input image.

4.2.3 Connections in the ICM

The function $W\{\cdot\}$ manages the connections between neurons as did the **M** and **W** matrices in the PCNN. However, the PCNN connections were static whereas the ICM connections are dependent upon the pulse activity of the previous iteration. Both the PCNN and ICM do rely on only local connections. Before the function $W\{\cdot\}$ is explained the foundation for this dynamic nature through the phenomenon of interference are considered.

4.2.3.1 Interference

In the PCNN model the connections are proportional to $1/r$ and static. This lead to the expanding waves seen in Fig. 4.2. The expanding nature of the waves caused an *interference* problem when the PCNN was applied to images with multiple objects. The waves expanding in Fig. 4.4 are autowaves and so when they collide the wave

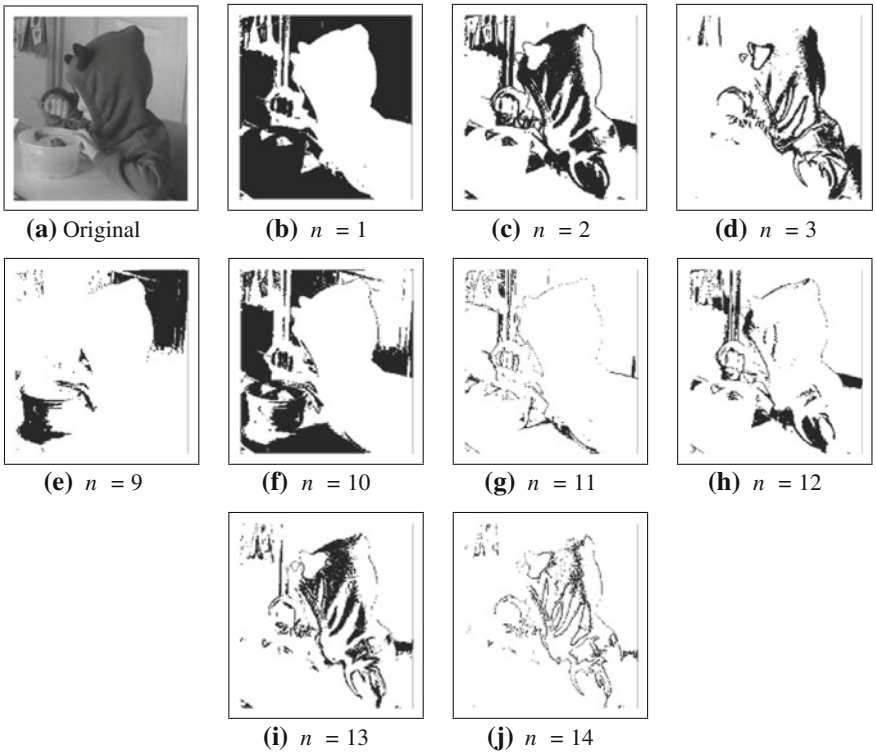


Fig. 4.20 An original image and several selected pulse images from the PCNN

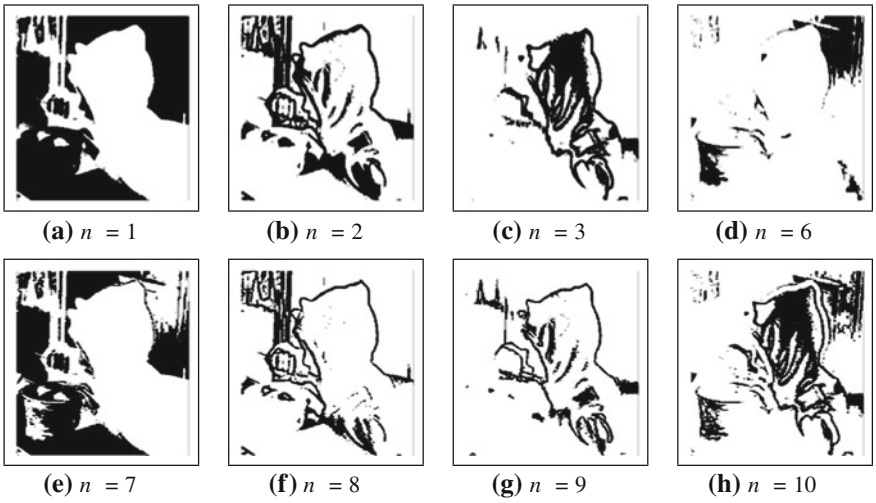


Fig. 4.21 Results from the ICM

Fig. 4.22 A target (flower) pasted on a background

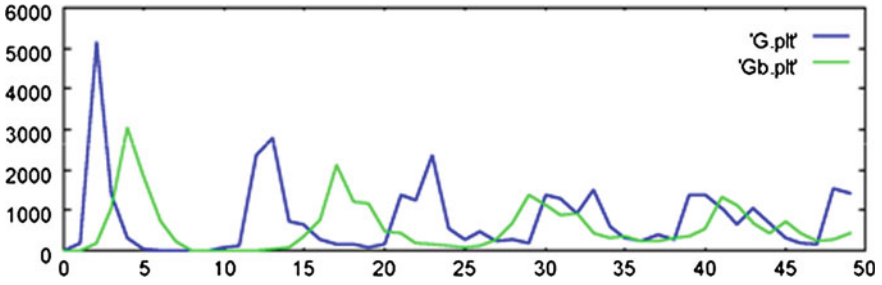


Fig. 4.23 The signature of the flower without a background (G.plt) and the signature of the flower with a background (Gb.plt). The x-axis represents the iteration index and the y-axis is the number of on-target neurons that are pulsing

fronts are annihilated. This means that the presence of the second object interferes with the expanding wave of the first object and this is the root cause of interference. The autowaves expanding from non-target objects will alter the autowaves emanating from target objects. If the non-target object is brighter it will pulse earlier than the target object autowaves, and its autowave can pass through the target region before the target has a change to pulse. The values of the target neurons are drastically altered by the activity generated from non-target neurons. Thus, the pulsing behaviour of on-target pixels can be seriously altered by the presence of other objects. This was shown in Fig. 4.15 where the bright objects pulse in unison but the darker objects did not.

Consider the input image in Fig. 4.22 in which the target (a flower) is pasted onto a background. The target was intentionally made to be darker than the background to amplify the interference effect. Two inputs $\mathbf{F}_1$ and $\mathbf{F}_2$ were created where $\mathbf{F}_1$ was the image shown in Fig. 4.22 and $\mathbf{F}_2$ was only the flower without a background. The pulse images from a modified ICM were computed for each input. The modification was to use the connections $\mathbf{M}$ that were also used in the PCNN. In this case the signatures were computed by using only the pixels on-target. Figure 4.23 shows the signature

Fig. 4.24 The propagation of curvature flow boundaries

for the flower without the background and for the target pixels for the image with the background. As seen the two signatures are very different and thus the pulse activity for the neurons on-target is different solely due to the presence of the background. It would be quite difficult to recognise an object from the neural pulses if those pulses are so susceptible to the content of the background.

4.2.3.2 Curvature Flow

The solution to the interference effect is based on curvature flow theory [69]. In this scenario the waves do not propagate outwards but instead propagate towards the centripetal vectors that are perpendicular to the wave front. Basically, they propagate towards local centre of curvatures. For solid 2D objects the curvature flows will become a circle and then collapse to a point [34]. (There is an ongoing debate as to the validity of this statement in dimensions higher than two.)

Such propagation from Malladi and Sethian [69] is shown in Fig. 4.24. The initial frame presents a intricate 2D shape. This figure will eventually evolve into a circle and then collapse to a point. There is a strong similarity between this type of propagation and the propagation of autowaves. In both cases the wave front will evolve to a circle. The difference is that the autowaves will also expand the circumference with each iteration whereas the curvature flow will be about the same size as the original shape.

The interference in the ICM that lead to the deleterious behaviour in Fig. 4.23 was caused when the neural communications of one object interfered with the behaviour of another. In other words, the autowaves from the background infringed upon the territory owned by the flower. This stems from the ever expanding nature of the autowaves.

Curvature flow models evolve to the same shape as autowaves but do not have the ever-expanding quality. Thus, the next logical step is to modify the connection function $W\{\cdot\}$ to behave more like curvature flow wave fronts.

4.2.3.3 Centripetal Autowaves

A *centripetal autowave* follows the mechanics of curvature flow. When a segment pulses its autowave will propagate towards a circle and then collapse. It does not

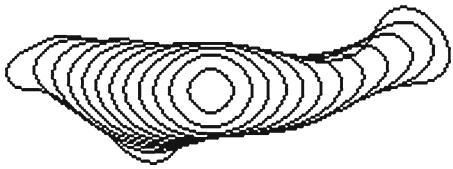


Fig. 4.25 The progression of an autowave from the larger initial shape tending towards a circle and then collapsing to a point

propagate outwards as does the traditional autowave. The advantageous result is that autowaves developed from two neighbouring objects will have far less interference.

The propagation of a curvature flow boundary is towards the local centre of curvature. The boundary, C, is a curve with a curvature vector $\boldsymbol{\kappa}$. The evolution of the curve follows,

$$\frac{\partial C}{\partial t} = \boldsymbol{\kappa} \cdot \hat{n}, \tag{4.20}$$

where $\hat{n}$ is normal. In two-dimensional space all shapes become a circle and then collapse to a point. Such a progression is shown in Fig. 4.25 where a curve evolves to a circle and then collapses.

The ever-expanding nature of the autowaves leads to the interference and this quality is absent in a curvature flow model. Thus, the logical step is to modify the neural connections to behave as in the curvature flow model. This requires that the connections between the neurons be dependent upon the activation state of the surrounding neurons. However, in creating such connections the problem of interference is virtually eliminated. In this new scenario neural activity for on-target neurons is the same independent of the presence of other objects. This is a major requirement for the employment of these models as image recognition engines.

The new model will propagate the autowaves towards the local centre of curvature and thus obtain the name *centripetal autowaves*. The computation of these connections requires the re-definition of the function $W\{\cdot\}$.

Computations for curvature can be cumbersome for large images, so, an image-friendly approach is adopted. The curves in figure start with the larger, intricate curve and progress towards the circle and then collapse to a point. The neural communications will follow this type of curvature flow progression. Of course, in the ICM there are other influences such as the internal mechanics of the neurons which influence the evolution of the neural communications.

The function $W\{A\}$ is computed by,

$$W\{A\} = \left[\left[F_{2,A'}\left\{M(A')\right\} + F_{1,A'}(A')\right] < 0.5\right], \tag{4.21}$$

where,

$$A' = A + \left[F_{1,A}\left\{M(A)\right\} > 0.5\right]. \tag{4.22}$$

The function $M(A)$ is a smoothing function. The function $F_{1,A}\{X\}$ is a masking function that allows only the pixels originally OFF in A to pass as in,

$$\left[F_{1,A}\right]_{ij} = \begin{cases} X_{ij} & \texttt{if } A_{ij} = 0 \\ 0 & \texttt{Otherwise} \end{cases}, \tag{4.23}$$

and likewise $F_{2,A}\{X\}$ is the opposing function,

$$\left[F_{2,A}\right]_{ij} = \begin{cases} X_{ij} & \texttt{if } A_{ij} = 1 \\ 0 & \texttt{Otherwise} \end{cases}. \tag{4.24}$$

The inequalities are passing thresholds as in,

$$[X > d]_{ij} = \begin{cases} 1 & \texttt{if } x_{ij} \geq d \\ 0 & \texttt{Otherwise} \end{cases}, \tag{4.25}$$

and

$$[X < d]_{ij} = \begin{cases} 1 & \texttt{if } x_{ij} \leq d \\ 0 & \texttt{Otherwise} \end{cases}. \tag{4.26}$$

This system works by basically noting that a smoothed version of the original segment produces larger values in the off-pulse region and lower values in the on-pulse region in the same areas that the front is to propagate. The non-linear function isolates these desirable pixels and adjusts the communication wave front accordingly.

The centripetal autowave signatures of the same two images used to generate the results in Fig. 4.23 are shown in Fig. 4.26. It is easy to see that the background no longer interferes with the object signature. The behaviour of the on-target neurons are now almost independent of the other objects in the scene. This quality is necessary for image applications.

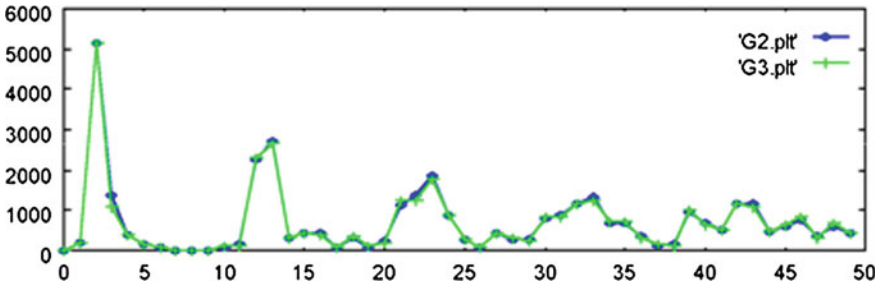


Fig. 4.26 The signatures of the flower and the flower with a background using the centripetal autowave model. The x-axis represents the iteration index and the y-axis is the number of on-target neurons that are pulsing

4.2.4 Python Implementation

Python implementation of the ICM is very similar to the PCNN. Since a problem may require multiple instantiations of the ICM the script is contained within an object. The first part of the object is shown in Code 4.8 which begins with the initial definition on Line 5. Line 7 defines the three variables and the constructor begins on Line 9. Similar to the PCNN the constructor allocates space for the necessary arrays. One difference is that the initial values of the threshold are set to 1. If they are set to 0 then the first pulse image is all pixels that have a value greater than 0 which is usually the entire image. Then about six iterations are needed before the threshold is reduced so that neurons can again pulse. Thus, the first seven iterations are meaningless. By starting the threshold values at 1 the first pulse images are meaningful.

Code 4.8 Constructor for ICM.

```
# icm.py
from numpy import ones, zeros
from levelset import LevelSet

class ICM:
    """Intersecting Cortical Model"""
    f,t1,t2 = 0.9,0.8,20.0

    def __init__ (self,dim):
        self.F = zeros( dim,float)
        self.Y = zeros( dim,float)
        self.T = ones( dim,float)
```

The function **Iterate** is shown in Code 4.9. This uses the Gaussian connections which create the expanding autowaves, and Code 4.10 shows the **IterateLS** function which creates the centripetal autowaves. which are more desirous for image applications. It calls the **LevelSet** function (Code 4.12) which computes the connections for each iteration twice. The second call to the function increases the speed at which the autowaves travel and through experience two iterations provide the better results. However, the user may wish to alter the speed of the autowaves for their application. Lines 8–10 perform the three ICM equations.

A typical instantiation of the ICM is shown in Code 4.11 in which the first three lines import the necessary tools. The image is loaded in Line 4 and converted to a matrix in Line 5 with all pixel values between 0 and 1. Line 6 calls the **cspline2d** function which acts as a smoothing operator. This is common for both PCNN and ICM applications as it removes single pixel noise and provides a significantly better segmentation. The results of Code 4.11 are shown in Fig. 4.21.

Code 4.9 Iteration for ICM.

```
# icm.py
    def Iterate (self,stim):
        if self.Y.sum() > 0:
            work = Smooth(self.Y.astype(float),3)
        else:
            work = zeros(self.Y.shape,float)
        self.F = self.f * self.F + stim + 8*work
        self.Y = self.F > self.T
        self.T = self.t1 * self.T + self.t2 * self.Y + 0.1
```

Code 4.10 Iteration for ICM creating centripetal autowaves.

```
# icm.py
    def IterateLS( self, stim ):
        if sum(sum(self.Y))>10:
            work = LevelSet( self.Y)
            work = LevelSet( work )
        else:
            work = zeros(self.Y.shape,float)
        self.F = self.f * self.F + stim + work
        self.Y = self.F > self.T
        self.T = self.t1 * self.T + self.t2 * self.Y + 0.1
```

Code 4.11 Driving the ICM.

```
>>> import icm
>>> import Image
>>> from scipy.signal import cspline2d
>>> mg = Image.open( 'icecream0.png' )
>>> data = mgconvert.i2a( mg.convert('L') )/255.0
>>> data = cspline2d( data, 2 )
>>> net = icm.ICM( data.shape)
>>> Y = []
>>> for i in range( 20 ):
    net.IterateLS( data )
    Y.append( net.Y + 0 )
```

4.3 Summary

Cortical models have been expressed in mathematical form for five decades now. The same basic premise of coupled oscillators or reaction-diffusion systems still apply to current models. Furthermore, in an image processing application the differences between the different models may not be that important. Therefore, speed and

simplicity of implementation are more important here than replication of a biological system.

Code 4.12 The **LevelSet** function.

```
# levelset.py
from scipy.signal import cspline2d

def LevelSet(A):
    # A is the input array
    # returns an array
    # The addition portion
    Aofftarg = A <= 0
    Aontarg = A > 0
    M = cspline2d(Aontarg.astype('d'),70)
    Mofftarg = M * Aofftarg.astype('d')
    Aadd = (Mofftarg.astype('d') > 0.5).astype('d')
    A = A + Aadd
    # The subtraction portion
    Aontarg = A > 0
    Aofftarg = 1 - Aontarg
    M = cspline2d( Aontarg.astype('d'),70)
    Montarg = M * Aontarg.astype('d') + Aofftarg.astype('d')
    Akill = (Montarg<0.5).astype('d')
    A = A - Akill
    return A
```

For image processing applications the model selected here is the ICM which consists of just three simple equations. Each neuron has two oscillators (the neuron potential and the neuron threshold) and each neuron has a non-linear operation. Thus, when stimulated, each neuron is capable of producing a spike sequence, and groups of locally connected neurons have the ability to synchronize pulsing activity. When stimulated by an image these collectives can represent inherent segments of the stimulating image. Thus, a cortical model can become a powerful first step in many image processing applications.

The traditional neural connection schemes, however, allow neural communications to continually progress away from the originating region. While this may have some biological foundation, this property has been found to be deleterious to object recognition. Activity from one region can so drastically alter the activity in another region that object recognition becomes very difficult. The solution to this problem is to alter the connections to the neurons so that they become sensitive to previous pulsing steps. In the model presented, these connections are described as centripetal autowaves such that the wave front progresses towards the local centre of curvature of the pulsing regions. This eliminates the ever-expanding nature of the waves without altering their shape-describing form.

The simplest applications of this ICM is to extract segments from images. A few examples were given though out the chapter to demonstrate the ability of the cortical models in image processing applications. This is only the beginning of the power that these algorithms can provide and the subsequent chapters will present more involved applications and results.

Chapter 5
Image Analysis

The development of the PCNN and ICM in the previous chapter was solely for the purpose of application to a variety of image processing and recognition tasks. In this chapter the PCNN and ICM will be used to directly extract pertinent information from a variety of images for the purpose of recognition.

Image recognition engines usually have multiple stages and often the first stage is to extract the information that is important to the recognition process. It could be argued that this is the most important stage, because the proper information is not extracted then the subsequent decision stage, no matter how powerful, will be unable to recognise the target. Furthermore, if the first stage can extract enough information then the decision stage could be a very simple algorithm. The extraction of enough information is basically the determining factor in the success of the recognition algorithm. In the following examples the PCNN and ICM are used to extract the important information for the individual application.

5.1 Pertinent Image Information

Images have components that are important for the image-processing task. For example, in image recognition it is generally the edges, texture or segments of an image that are the most important features. Of course, this is strongly application dependent. One traditional method of recognising objects within an image is through a Fourier filter. The logic of this type of filter is shown in Fig. 5.1. Basically, a filter containing the target centred in the frame is created and it is correlated with the input image. If the target exists in the input then a large correlation signal appears in the output correlation surface at the location of the target in the input image.

The Fourier filter system does have some serious drawbacks. First, if the target within the input scene does not exactly match the target image in the filter then the correlation signal is weaker. Thus, if the target were an aeroplane, which could be viewed at any angle with differing scale, and illumination, (perhaps obscured by a cloud), it is very difficult to design a filter that can recognise the target. The point of

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_5, © Springer-Verlag Berlin Heidelberg 2013

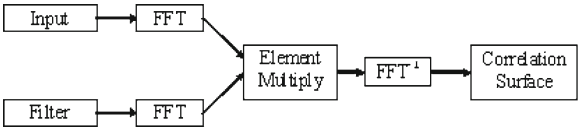


Fig. 5.1 Logical diagram of a Fourier filter

this text though is not to discuss the pros and cons of the Fourier filter. It is, however, to indicate that this most popular system still relies heavily on the three main image components: edges, textures and segments. In the Fourier space the lower frequencies of the image are located at the centre of the image and the higher frequencies are contained at the edges. Lower frequencies are present when the image has large areas of fairly uniform intensities. An aeroplane has lower frequencies from the hull and interior of the wings. Higher frequencies exist if the image has edges. For the aeroplane the edges of the craft and wings give rise to the higher frequencies.

There are two types of differentiation in image recognition. One is generalisation in which the filter is designed to recognise a class of objects even though it may not be trained on the particular target in the input image. For example, a filter may be designed to recognise aeroplanes as opposed to helicopters. This filter would need the general shape of an aeroplane, which is contained in the lower frequencies. These lower frequencies are from the larger, more uniform areas of the image—in other words the segments. Conversely, if the filter were designed to differentiate a particular type of aeroplane, from other flying vehicles then the general shape of an aeroplane is no longer useful. The second differentiation is discrimination which distinguishes between two objects of the same class. Usually, this type of information is contained in the higher frequencies (edges).

The Fourier filter works by matching the frequencies of the target with the frequencies in the input image. If the target exists in the input image then a strong match occurs and a large correlation signal is produced. Another manner in which to envision a Fourier filter is to take the target image and collocate it sequentially at every position in the input image. Eventually, the target in the filter and the target in the input are aligned and a match is easily seen. A Fourier filter is no more than performing a texture, segment, and edge match.

The whole point of this discussion is to indicate that the most common method of image recognition relies on the fundamentals of textures, edges and segments. Other types of image processing such as neural networks, morphology, and statistical processing also rely on these fundamentals.

While these methods are well understood theoretically, they have performed very poorly in the real world. Problems immediately arise when the training target(s) do not exactly match the input which is quite often the case. The signal of the correlation drops and the noise from the background rises until the two become indistinguishable. The problems are far too complicated for these types of processors.

The PCNN/ICM models provide tremendous advantages here. First, the PCNN/ICM have an inherent ability to extract the fundamentals of the image. Second,

the PCNN/ICM can simplify the image to allow recognition engines to perform a far easier task than is within their realm. The advantages of the PCNN/ICM in this form should not be a surprise since it is based on how mammals perform recognition. The PCNN/ICM can extract the image fundamentals inherently. The PCNN/ICM do not need training or adjustments to extract these fundamentals from a wide range of images. Edges and segments are extracted at different iterations and segments can easily be seen over the course of a few iterations. Segment extraction occurs since groups of neurons in a similar state tend to pulse in unison. Edges are extracted as the autowave expands from these segments. In the original form, the PCNN/ICM neurons will lose the unison pulsing according to the texture of the input. So, in time, the segments will tend to separate according to the texture.

The most important aspect of the PCNN/ICM performing these extractions is that it is an inherent quality of the PCNN/ICM. Traditional image processing has had engines that perform similar extractions but these are usually trained or designed to perform the task for particular applications. Furthermore, the PCNN/ICM will provide a higher quality of performance. For example, a popular method of extracting edges from an image is a Sobel filter, which consists of a small kernel in which the central elements are positive, and the surrounding elements are negative. Convoluting this kernel over the input image results in an image with only the edge pixels 'on'. The problem is that this filter will produce a double line output for the edges that it sees. In other words, the edges are extracted but not cleanly. The PCNN/ICM extracts sharp, clean edges.

Also recall that the PCNN/ICM produce binary images. Segments that are extracted are shown as a solid uniform segment in the PCNN/ICM output. Edges are also of the same intensity in the output even though the input edges may have a gradient of intensities. These binary segments and edges are well organised in the Fourier plane. Performing recognition on these binary images is far easier to accomplish than performing recognition on the original input. This argument will be discussed in subsequent chapters. Thus, the PCNN/ICM are powerful pre-processors. They extract the fundamentals of an image (edges, textures, and segments) and can present far simpler binary images to recognition engines for analysis.

One of the major tools in image processing is the ability to extract segments from the image. This can lead directly to object identification algorithms as well as many other types of analysis. For this example, consider the ICM which has the ability to isolate the input image into its segments. One method that demonstrates the segmentation ability of the ICM is to accumulate the pulses weighted by their iteration. The output image is computed by,

$$P_{i,j} = \sum_{N} \alpha_n Y_{i,j}[n], \tag{5.1}$$

where α_n is a scaling factor that is inversely, monotonically proportional to n. An accumulation example is shown in Fig. 5.2. In this image there are only a few grey levels in this image (one for each iteration over only the first cycle of pulses). Thus, segmentation of the image can be viewed in a collective context. If the ICM were a

(a) Original. **(b)** Accumulation.

Fig. 5.2 The original image (**a**) and the weighted accumulation of pulses (**b**)

poor segmentation engine then the details of the original image would be lost in the cumulative pulse image. Small details such as those in the upper left corner, wrinkles in the fabric, the spoon, etc. are still quite distinct in the cumulative image, thus the segmentation is faithful to the segments inherent in the original image.

The PCNN and ICM also extract important edge information, although they are certainly not the first algorithms to do so. The purpose of edge extraction algorithms is to enhance the edges contained within an image and this enhancement is generally proportional to the sharpness of the edges. There are two properties of the ICM pulses that make it ideal for edge extraction. The first is the obvious property that the pulses are binary and thus the edges are sharp. The second property is that the pulse segments are usually solid, and these assist in separating edges of objects from the edges due to texture.

The picture in Fig. 5.3a displays a skater with some notable properties. There are some very distinct edges, but there are also subtle edges due to texture such as the inside of his coat. There are also edges due between objects with similar grey scale values (i.e., the gloves and the coat sleeves).

One simple method of extracting edges from an image is to accumulate the differences between neighbouring pixels in both the vertical and horizontal direction,

$$a_{i,j} = \sqrt{\Delta^2_{x:i,j} + \Delta^2_{y:i,j}}\,, \tag{5.2}$$

where $\Delta_{x:i,j}$ describes the change in values in the horizontal direction, as in,

$$\begin{aligned} \Delta_{x,i,j} &= \frac{\left(M_{i,j} - M_{i,j-1}\right) + \left(M_{i,j} - M_{i,j+1}\right)}{2} \\ &= \frac{2M_{i,j} - M_{i,j-1} - M_{i,j+1}}{2}, \end{aligned} \tag{5.3}$$

(a) Original. **(b)** Edge enhanced.

Fig. 5.3 Figure **a** displays the original image and **b** an edge enhanced version

and

$$\Delta_{y,i,j} = \frac{2M_{i,j} - M_{i-1,j} - M_{i+1,j}}{2}. \tag{5.4}$$

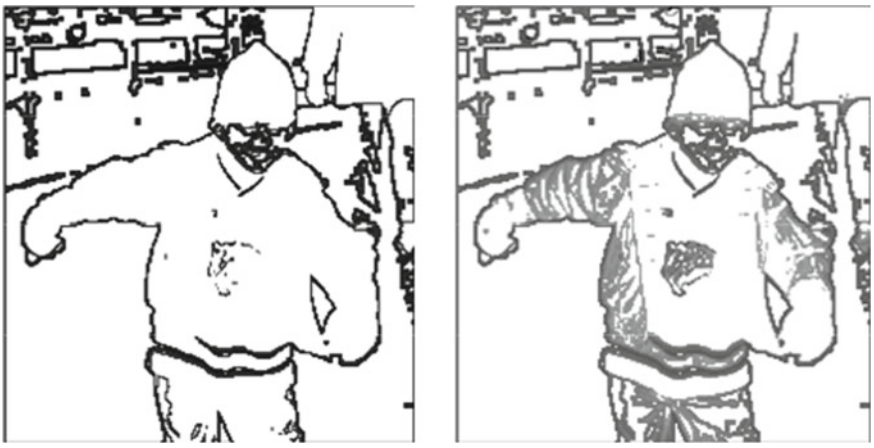
The enhanced version of the image in Fig. 5.3a is shown in Fig. 5.3b. To display this image in a print format it has been inverted such that the darkness of the lines indicates the sharpness of the edges. Subtle edges due to coat texture are detected but are too faint to see in the image.

Edge extraction with the ICM entails the demarcation of the edges from the pulse images. The level of edge detection (the intensity of the resultant edge) is inversely proportional to the ICM iteration number n,

$$b_{i,j} = \sum_{n=0}^{M} \alpha_n Y_{i,j}[n]. \tag{5.5}$$

The scalar α is the proportionality term and M is the number of iterations that are considered. The image in Fig. 5.4a displays the edged detection process with $M = 2$. These edges are similar to the strong edges in Fig. 5.3b. Allowing M to increase produces more interesting results. As the ICM iterates segments will pulse again with de-synchronisation. Segments that pulsed together in early iterations will tend to break apart in subsequent iterations. The image in Fig. 5.4b displays the process with $M = 6$.

The de-synchronisation is strong enough through the higher texture regions (coat) that there are now many edges to display. Obviously, adjusting the value of M determines the types of edges that the ICM can extract.



(a) $M = 2$. **(b)** $M = 6$.

Fig. 5.4 Edge detection for **a** $M = 2$ and **b** $M = 6$

5.2 Image Segmentation

The cohesive nature of the neural firings allows the ICM to act as an image segmentation engine. Neurons synchronise through autowave communications and create solid segmentations of pulses based upon the input.

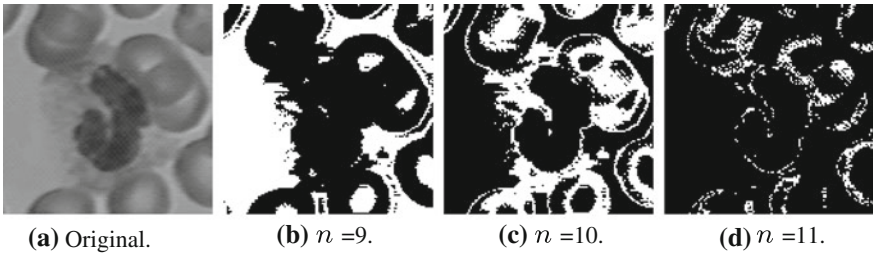
5.2.1 Blood Cells

An example begins with an image of a red blood cell surrounded by white blood cells shown in Fig. 5.5a. Selected output pulse images are also shown in Fig. 5.5. As seen the background, white blood cells, and red blood cells pulse at different iterations thus providing a segmentation.

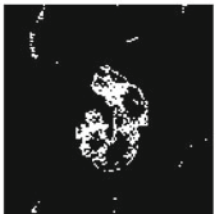
However, as noted before, the ICM performance is much improved if the input is smoothed first. This was introduced in Code 4.11 and the results are shown in Fig. 5.6. This step provides cleaner edges. The process is quite simple and the implementation is shown in Fig. 5.6. Line 4 employs the **cspline2d** function as the smoothing operator and the results of Fig. 5.5 are obtained by removing this line.

5.2.2 Mammography

Breast cancer is one of the leading causes of death for women the world over and its early detection is thus very important. In clinical examinations, physi-



(a) Original. **(b)** n =9. **(c)** n =10. **(d)** n =11.



(e) n =13.

Fig. 5.5 Examples of the ICM segmentation with the pulses being shown as *white pixels* in print

(a) n =9. **(b)** n =10. **(c)** n =13.

Fig. 5.6 Examples of the ICM segmentation with a smoothed input

cians check for breast cancer by looking for abnormal skin thickenings, malignant tissues and microcalcifications. The latter are hard to detect because of similarity to normal glandular tissues. Wavelet transforms [8] have been used for automated processes has PCNN [56]. Examples of wavelet and PCNN processing of mammograms are shown in Figs. 5.7 and 5.8, respectively. The segmentation ability of the PCNN is clearly demonstrated in its ability to isolate these regions.

Code 5.1 Iterations for the ICM.

```
>>> from scipy.signal import cspline2d
>>> data = mgconvert.i2a( mg.convert('L') )
>>> data /= data.max()
>>> data = cspline2d( data, 1 )
>>> Y = []
>>> for i in range( 20 ):
        net.IterateLS( data )
        Y.append( net.Y + 0 )
>>> mgconvert.a2i( Y[9] ).show()
```

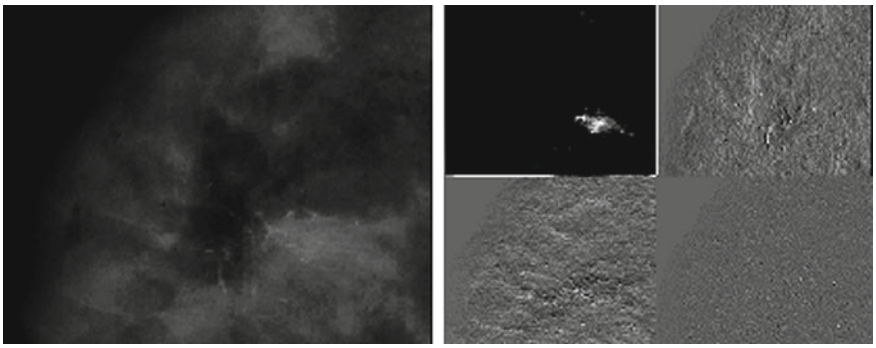
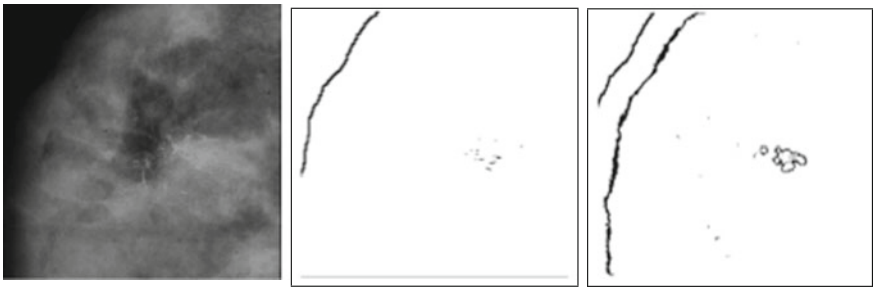


Fig. 5.7 A 2D Haar wavelet transform applied to an input image showing clusters of branching, pleomorphic calcification associated with poorly defined mass diagnosed as *duct carcinoma*



(a) Input **(b)** **(c)**

Fig. 5.8 The input to the PCNN is a mammogram showing clusters of branching, pleomorphic calcification associated with poorly defined mass diagnosed as duct carcinoma. The pulsing pixels are shown in *black*

5.3 Adaptive Segmentation

Raya et al. [86] explore the optimization of the β parameter in a fast-linking PCNN for applications in image segmentation. In the traditional PCNN the β is a positive constant, but in this modified model it is defined as,

$$\beta = 0.2\left(\frac{P}{\Theta - k_2\sigma_0} - 1\right), \tag{5.6}$$

where k_2 is a constant between 0.5 and 1.0, P is the primary firing threshold, and σ_0 is the standard deviation over the object pixels. The primary firing threshold is defined as,

$$\mu_o + k_1\sigma_0, \tag{5.7}$$

where μ_0 is the average over the object pixels and k_1 is between 1 and 2.

They also proposed an adaptive system in which the image was divided into $K \times K$ blocks and the threshold was adapted to each block. Results indicated superior and two examples are reprinted in Figs. 5.9 and 5.10. The input is shown with its histogram to indicate that the distribution of pixels from each region are overlapping. The final image in each figure shows clean segmentation.

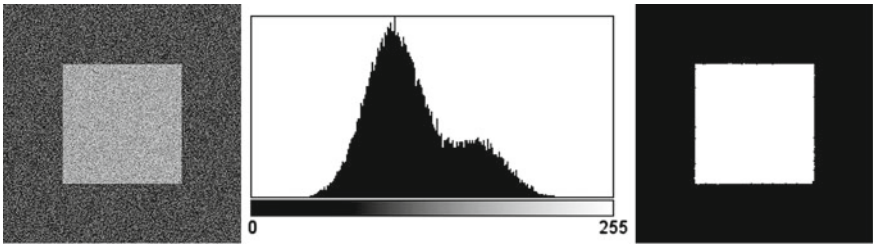


Fig. 5.9 A bimodal image with Gaussian random noise was created and shown in (**a**) with the histogram shown in (**b**). Figure **c** shows the segmentation using the PCNN [86]

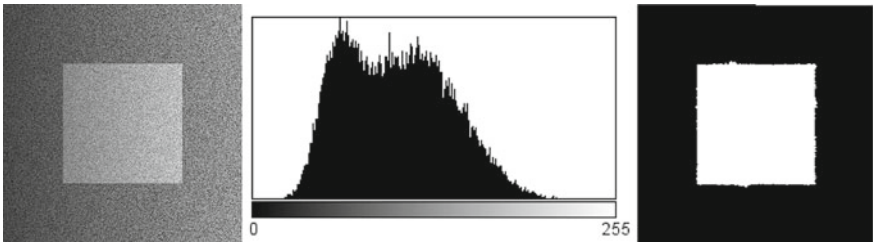


Fig. 5.10 A second PCNN example with the original shown in (**a**), its histogram shown in (**b**), and the results shown in (**c**) [86]

Another method of adaptive segmentation was proposed by Lu et al. [24, 68] which modifies the fast linking PCNN to have and adaptive threshold. The process also simplifies the input to,

$$F_{ij}[t] = S_{ij}. \tag{5.8}$$

The linking operation changes the Gaussian connections to unit connections for a region N_{ij} defined for each i, j location,

$$L_{ij}[t] = \sum_{z \in N_i} Y_z[t] - d. \tag{5.9}$$

The internal and external excitations is still computed in the same manner,

$$U_{ij}[t] = F_{ij}[t]\left(1 + \beta_t L_{ij}[t]\right), \tag{5.10}$$

and,

$$Y_{ij}[t] = \begin{cases} 1 & \texttt{if } U_{ij}[t] > \Theta_{ij}[t] \\ 0 & \texttt{Otherwise} \end{cases}. \tag{5.11}$$

The threshold is determined by,

$$\Theta_{ij}[t] = \begin{cases} w_t & \texttt{if } P_{ij}[t-1] = 0 \\ \Omega & \texttt{Otherwise} \end{cases}, \tag{5.12}$$

where

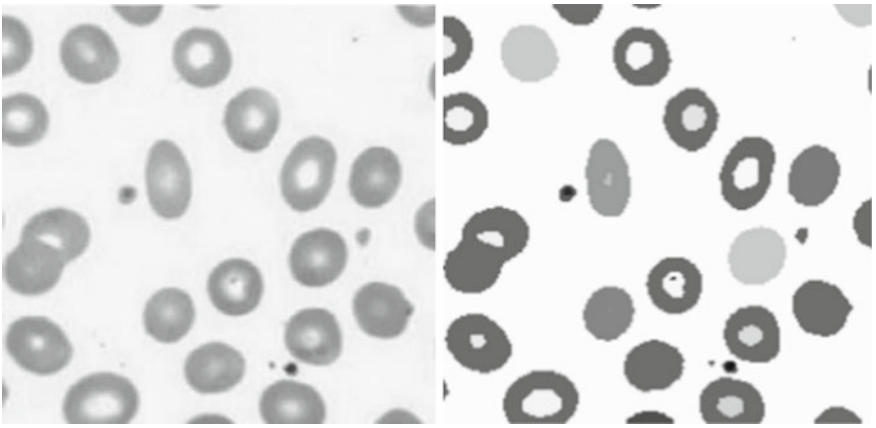
$$P_{ij}[t] = \begin{cases} t & \texttt{if } Y_{ij}[t] = 1 \\ P_{ij}[t-1] & \texttt{Otherwise} \end{cases}. \tag{5.13}$$

The w_t is set to be slightly less than the maximum intensity in the target region. Results of segmentation are shown in Fig. 5.11.

5.4 Focus and Foveation

The human eye does not stare at an image, but rather it moves to different locations within the image to gather clues as to the content of the image. This moving of the focus of attention is called *foveation*. A typical foveation pattern [119] is shown in Fig. 5.12. Many of the foveation points are on the corners and edges of the image. More foveation points indicate an area of greater interest.

A foveated image can be qualitatively described as an image with very rich and precise information at and around those areas of the image under intense observation, and with poorer information elsewhere. The concept of foveation, as applied here, exploits the limitations of the biological eye to discriminate and detect objects in the image that are not in direct focus. In mammalian psychophysics, most tasks



(a) Input **(b)** Segmentation.

Fig. 5.11 Segmentation offered by Lu et al. [68] using a modified PCNN

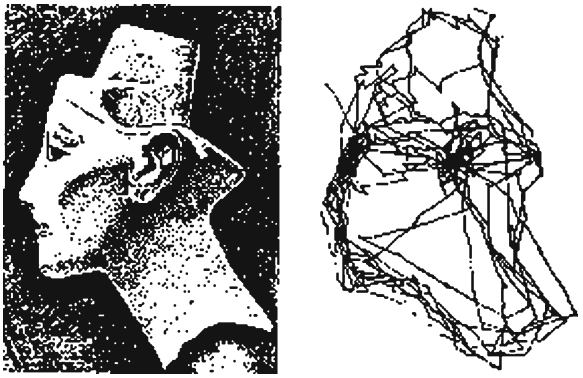


Fig. 5.12 A typical foveation pattern [92, 93, 119]

are performed better with foveal vision, with performance decreasing toward the peripheral visual field. The visual acuity can change by as much as a factor of 50 between the fovea and peripheral retina.

5.4.1 The Foveation Algorithm

Foveation points are generally located at the corners and strong edges within an image. The ICM has the ability to extract segments that contain strong edges. Therefore, a simple foveation algorithm can be constructed by detecting the corners from

(a) Input. **(b)** Pulse Image.

Fig. 5.13 The original image and third pulse image from the ICM

ICM segments. Employed here is a simple corner detector based upon a SUSAN detector [100]. Given an image **I**, the modified SUSAN detector is the peaks in,

$$\mathbf{P} = \exp\{\mathbf{S} - \mathbf{I}\}, \tag{5.14}$$

where **S** is a slightly smoothed version of **I**.

Consider the image in Fig. 5.13a as an example. This image contains sharp edges and corners from the buildings and softer edges and corners from nature. Figure 5.13b displays the pulse image from the third iteration of the ICM (where dark ink indicates a pulsed neuron). The goal of the foveation algorithm is to find the corners and edges in this frame.

The next step is to enhance the corners and edges of the image which follows from Eq. 5.14. Code 5.2 displays the **Corners** function. This function receives the pulse image and returns an edge and corner enhanced image as shown in Fig. 5.14a. To be more presentable in print form the image that is shown is `-abs(corners-1)`.

Since the pulse image has binary values the peaks in **P** are at the corners of the image and can be extracted through a simple peak detector. Such a detector is shown in Code 5.3 with the function **PeakDetect**. Line 3 computes the maximum value of the input and if that is over a strong threshold (Line 7) then the process continues. The largest peak is found in Line 9 and if it is too small then the loop is terminated (Line 11). Otherwise, the location of the peak is appended to the list `peaks` (Line 13). Lines 14 and 15 erase pixels with a radius of 10 so that foveation points are not located too close to each other. The function returns a list `peaks` that contains the pixel locations of the peaks.

The list of peaks is converted to an image through the **Mark** function shown in Code 5.4. This function creates a 5×5 marker at each peak location. This image is combined with the original image in the **Mix** also shown in Code 5.4. This function

Code 5.2 The **Corners** function.

```
def Corners( data ):
    if data.sum() > 10:
        a = cspline2d( data, 5 )
        corners = np.exp(-(a-data))
    else:
        corners = np.zeros( data.shape )
    return corners
```

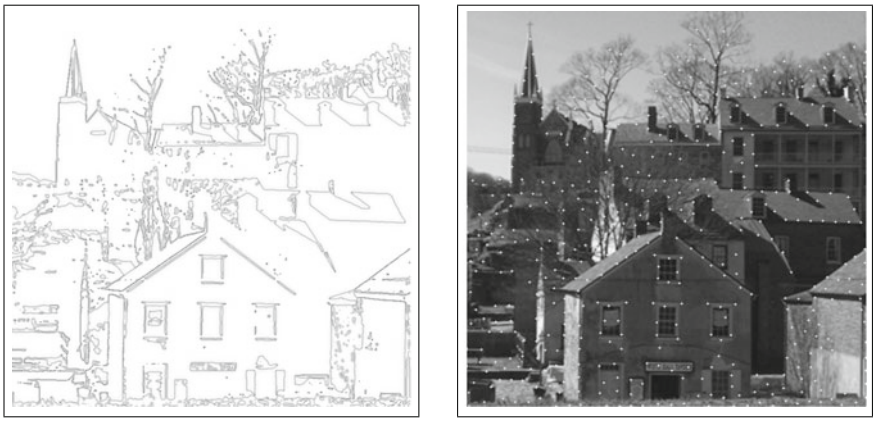
scales the input image so that the maximum value is 250 and that the maximum value of the peak marker pixels is 255. Thus, the markers will always be slightly brighter than any pixel in the input.

Code 5.5 shows a typical run in which the original colour image is loaded in Line 4 and converted to a grey scale matrix in Line 5. Lines 8–11 create the ICM and the first three pulse images. The third is then sent to the **Corners** function in Line 12. The peaks are detected and marked. The output `answ` is shown in Fig. 5.14b.

For a full run Lines 12–15 are applied to each pulse image thus gathering foveation points across several pulse images. The result is shown in Fig. 5.15.

5.4.2 Target Recognition by a PCNN-Based Foveation Model

A test consisting of handwritten characters demonstrates the ability of a PCNN-based foveation system. The PCNN generates foveating points which are now centres of attention or possible centres of features. The features of these centres can be identi-



(a) Corners and edges. **(b)** Foveation points.

Fig. 5.14 The edges and corners inherent in Fig. 5.13b and the foveation markers

Code 5.3 The peak detecting function.

```
# foveation.py
def PeakDetect( matx ):
    mx = matx.max() # max value
    V, H = matx.shape
    peaks = []
    ok = 0
    if mx > 0.9: ok = 1
    while ok:
        v,h = divmod( matx.argmax(), H )
        if matx[v,h] < 0.5*mx:
            ok = 0
        else:
            peaks.append( (v,h) )
            circ = geometry.Circle( matx.shape, (v,h), 10 )
            matx *= 1-circ   # zap
    return peaks
```

Code 5.4 The **Mark** and **Mix** functions.

```
# foveation.py
def Mark( marks, peaks ):
    for v,h in peaks:
        marks[v-2:v+3,h] = 1
        marks[v,h-2:h+3] = 1

def Mix( marks, indata ):
    answ = indata / indata.max()
    answ *= 250
    mask = marks>0
    answ = (1-marks)*answ + marks*255
    return answ
```

fied, and using a fuzzy scoring algorithm [103] it is possible to identify handwritten characters from an extremely small training set [102].

Typical handwritten characters are shown in Fig. 5.16.

Once the foveation points are produced, new images are created by a barrel transformation centred on each foveation point. Examples of the letter 'A' and barrel transformations centred on the foveation points are shown in Fig. 5.19. This distortion places more emphasis on the information closer to the foveation point. Recognition of these images constitutes the recognition of a set of features within the image and combining the recognition of these features with the fuzzy scoring method.

Recognition of the feature images is performed through a Fractional Power Filter (FPF) (see Appendix C) [13]. This filter is a composite filter that has the ability to manipulate the trade-off between generalization and discrimination that is inherent in first order filters. In order to demonstrate the recognition of a feature by this

Code 5.5 Loading the image and finding the peaks.

```
>>> import Image
>>> import numpy as np
>>> from scipy.signal import cspline2d
>>> mg = Image.open( 'tavern2s.png')
>>> data = np.array( mg.convert('L'))/255.0
>>> V,H = data.shape
>>> marks = np.zeros( (V,H) )
>>> net = icm.ICM( (V,H) )
>>> net.IterateLS( data )
>>> net.IterateLS( data )
>>> net.IterateLS( data )
>>> corners = Corners( net.Y )
>>> peaks = PeakDetect( corners )
>>> marks = np.zeros( (V,H) )
>>> Mark( marks, peaks )
>>> answ = Mix( marks, data )
```

Fig. 5.15 Foveation points from multiple ICM pulse images

method an FPF was trained on 13 images of which 5 were target features and 8 were non-target features. For this example one target feature is the top of the 'A' (see Fig. 5.19b) and the non-targets are all other features.

The results of the test are presented as three categories. The first measures how well the filter recognised the targets, the second is how well the system rejected non-targets, and the third considers the case of a non-target that is similar to the target

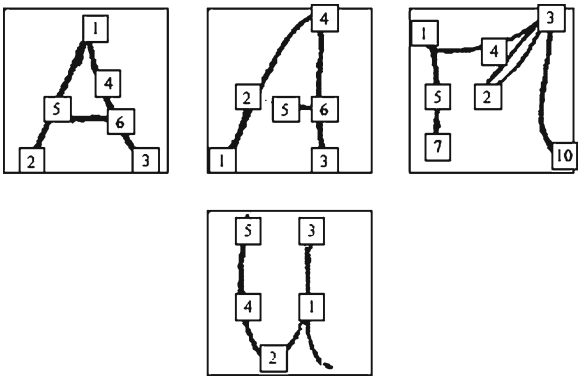


Fig. 5.16 Handwritten characters and their foveation points as determined by the PCNN-based model

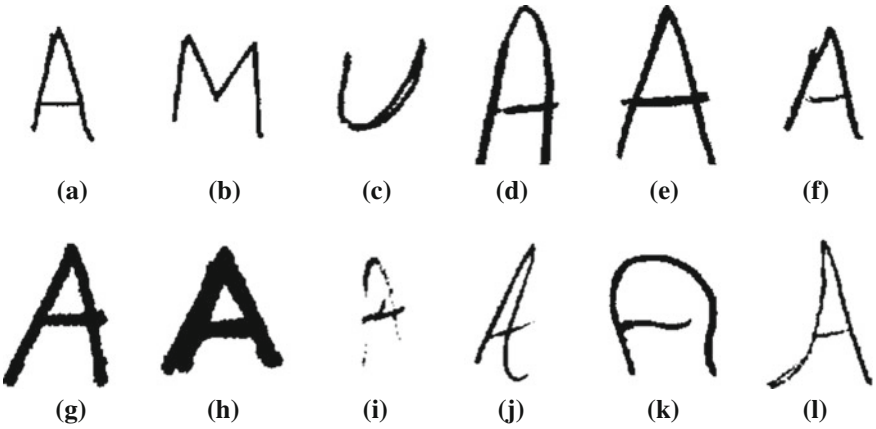
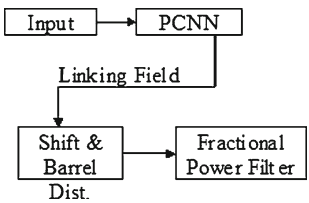


Fig. 5.17 Typical handwritten letters

Fig. 5.18 The logical flow of the recognition system



(such as the 'M' having two features similar to the top of the 'A'). The maximum correlation signature about the area of the foveation point was recorded. The FPF was trained to return a correlation peak of 1 for targets and a peak of 0 for non-targets. The results for (non-training) targets and dissimilar non-targets are shown in Table 5.1. Similar non-targets produced significant correlation signatures as expected. Certainly, a single feature cannot uniquely characterize an object. The similar features

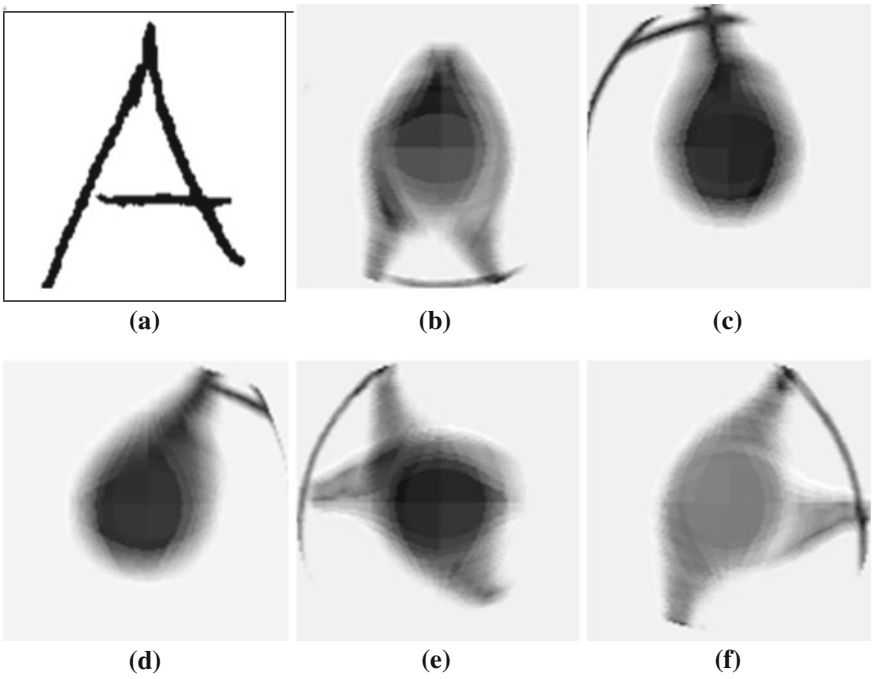


Fig. 5.19 An original 'A' and the 5 barrel distorted images based upon natural foveation points

Table 5.1 Foveation recognition

Category	Average	Low	High	Std. Dev.
Target	0.995	0.338	1.700	0.242
Non-target	0.137	0.016	0.360	0.129

of the 'M' also produced significant correlation signals. This indicates the obvious: that single features are insufficient to recognise the object.

The results demonstrate an exceedingly good separation between targets and non-targets. There were a few targets that were not recognised well. Such targets come from Fig. 5.16i–k. Figure 5.16i is understandable since the object is poorly represented. Figure 5.16j performed poorly since the top of the 'A' is extremely narrow, and Fig. 5.16k has an extremely rounded feature on top. These last two features were not represented in the training features. All of the types of 'A's produced a correlation signature above 0.8 which is clearly distinctive from the non-targets.

A few false negatives are not destructive to the recognition of the object. Following the example of [103], a collection of recognised features can be grouped to recognise the object. Noting the locations of the correlation peaks in relation to each other performs this. A fuzzy score is attached to these relationships. A large fuzzy score indicates that features have been identified and are located in positions that are indicative of the target.

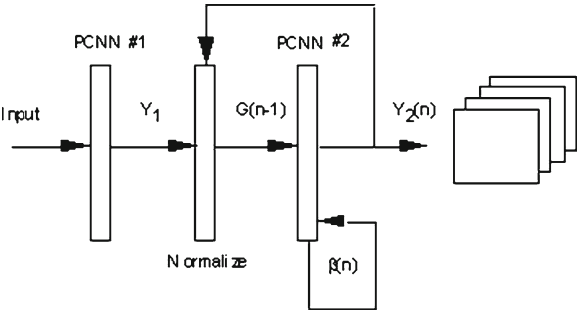


Fig. 5.20 Hierarchical image factor generation. An input image is decomposed into a set of factor images, ordered from coarse to fine in image detail. The product of the set is equal to the original image

It has been shown that the PCNN can extract foveation points, that attentive (barrel-distorted) images can be created and centred about foveation points. Furthermore, it has been shown that these images which now represent a feature of the image, can be recognised easily, and it has been shown elsewhere that a combination of these recognised features and their locations can be combined in a fuzzy scoring method to reach a decision about the content of the input.

5.5 Image Factorisation

The major problem in automatic target recognition is that images of the target change with scale, lighting, orientation, etc. One way to get around this problem has been suggested by Johnson [48, 79]. It involves a hierarchical image decomposition, which resolves an image into a set of image product factors. The set is ordered in scale from coarse to fine with respect to image detail. When the factored set is multiplied together it reproduces the original image. By selecting factors, coarse scene elements such as shadows, and fine scene factors such as noise, can be isolated. The scale of detail is controlled by the linking strength of a pulse coupled neural network, on which the system is based.

The factorisation system consists of three layers as shown in Fig. 5.20. The first layer is a PCNN and its purpose is to define the limit of detail to which the input will be resolved, both spatially and in intensity. The second layer serves to re-normalize the input to the third layer. The second layer is also a PCNN and together with the third layer it operates in a cyclic manner to provide the ordered output set of factors. The re-normalization is via a shunting action approximated by dividing the previous input to the third layer by the current output of the third layer. The output set consists of the outputs of the third layer in the order they were generated. Both PCNNs use a single-pass, linear decay model with nearest-neighbour sigmoidal linking.

The algorithm is discussed in some detail in [48], but is simply expressed as,

$$G[n] = \frac{G[n-1]}{Y_2[n]}, \tag{5.15}$$

and

$$\beta[n] = l\beta[n-1], \tag{5.16}$$

where G represents a signature (Eq. (4.8)), and $G[0] = Y_1$. Here Y_1 represents the output of the first PCNN, $G[n]$ is the input to the second PCNN at the beginning of the nth iteration, $Y_2[n]$ is the output of the second PCNN at the end of the $n-1$ iteration, and $k < 1$ is the linking strength reduction factor per iteration. In the above equation $\beta[0]$ is the initial value assigned by the operator. Together with the parameter k, it determines the initial coarseness resolution and the number of cycle's n. In the above, the spatial dependence of G and the PCNN output images Y_1 and Y_2 is suppressed, as the re-normalisation is applied on a pixel-by-pixel basis. The change of β is global, the same value being used by every pixel.

On the first iteration the second layer passes its input directly to the third layer. Its coarsely grey-scale quantifies it, giving an output that is coarse in both spatial and in intensity detail. When it is used by the second layer to normalise the original input, the new input, and all successive ones, will be between zero and one. As the second input is processed by the output PCNN, which now uses a reduced value of its linking strength, only the regions of intensity less than unity give values different than those of the first output.

5.6 Summary

The pulse images generated by the PCNN and ICM tend to isolate homogeneous segments inherent in the input image. This immediately leads to a segmentation tool which is demonstrated in a couple of examples. One of the many published alterations is reviewed here in which the PCNN is changed slightly to form an adaptive system. This adaptive system has shown the ability to segment images even in a noisy environment in which the pixel intensity distributions overlap.

Once the segments are available PCNN based systems evolved into foveation systems which direct the focus to regions of sharp corners and edges much like humans do. This is a necessary tool in some image recognition algorithms that attempt to divine the content of real but cluttered images.

Chapter 6
Feedback and Isolation

6.1 A Feedback PCNN

The PCNN can be a very powerful front-end processor for an image recognition system. This is not surprising since the PCNN is based on the biological version of a pre-processor. The PCNN has the ability to extract edge information, texture information, and to segment the image. This type of information is extremely useful for image recognition engines. The PCNN also has the advantage of being very generic. Very few changes (if any) to the PCNN are required to operate on different types of data. This is an advantage over previous image segmentation algorithms, which generally require information about the target before they are effective.

There are three major mechanisms inherent in the PCNN. The first mechanism is a dynamic neural threshold. The threshold, here denoted by Θ, of each neuron significantly increases when the neuron fires, then the threshold level decays. When the threshold falls below the respective neuron's potential, the neuron again fires, which raises the threshold, Θ. This behaviour continues which creates a pulse stream for each neuron.

The second mechanism is caused by the local interconnections between the neurons. Neurons encourage their neighbours to fire only when they fire. Thus, if a group of neurons is close to firing, one neuron can trigger the entire group. Thus, similar segments of the image fire in unison. This creates the segmenting ability of the PCNN. The edges have different neighbouring activity than the interior of the object. Thus, the edges will still fire in unison, but will do so at different times than the interior segments. Thus, this algorithm isolates the edges.

The third mechanism occurs after several iterations. The groupings tend to break in time. This "break-up" or de-synchronisation is dependent on the texture within a segment. This is caused by minor differences that eventually propagate (in time) to alter the neural potentials. Thus, texture information becomes available.

The Feedback PCNN (FPCNN) sends the output information in an inhibitory fashion back to the input in a similar manner to the rat's olfactory system. The outputs are collected as a weighted time average, A, in a fashion similar to the computation

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_6, © Springer-Verlag Berlin Heidelberg 2013

of Θ except for the constant V,

$$A_i[n] = e^{-\alpha_A \delta_n} A_{ij}[n-1] + V_A Y_{ij}[n], \tag{6.1}$$

where V_A is much lower than V and in this case $V_A = 1$. The input is then modified by,

$$S_{ij}[n] = \frac{S_{ij}[n-1]}{A_{ij}[n-1]}. \tag{6.2}$$

The FPCNN iterates the PCNN equations with Eqs. (6.1) and (6.2) inserted at the end of each iteration. Two simple problems are shown to demonstrate the performance of the FPCNN. The first problem used a simple square as the input image. Figure 6.1 displays both the input stimulus **S** and the output **Y** for a few iterations until the input stabilised.

At $n = 5$ an output pulse has been generated by the FPCNN which is a square that is one pixel smaller than the original square on all four sides. At this point in the process the output of the FPCNN matches that of the PCNN. The PCNN would begin to separate the edges from the interior. In the case of the FPCNN, however, the input will now experience feedback shunting that is not uniform for the entire input. This is where the PCNN and the FPCNN differ.

As the iterations continue the activations go from a continuous square to just the edges of a square, and finally to just the four corners. The four corners will remain on indefinitely. It is interesting to note that for the case of a solid triangle with unequal angles, the same type of behaviour occurs except that the corner with the smallest

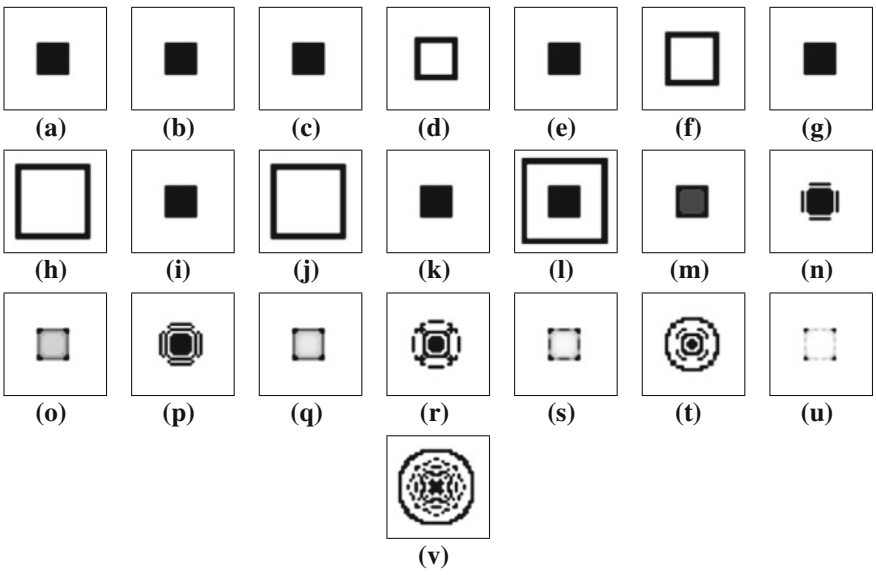


Fig. 6.1 Input and output pairs for FPCNN calculations for a solid square input

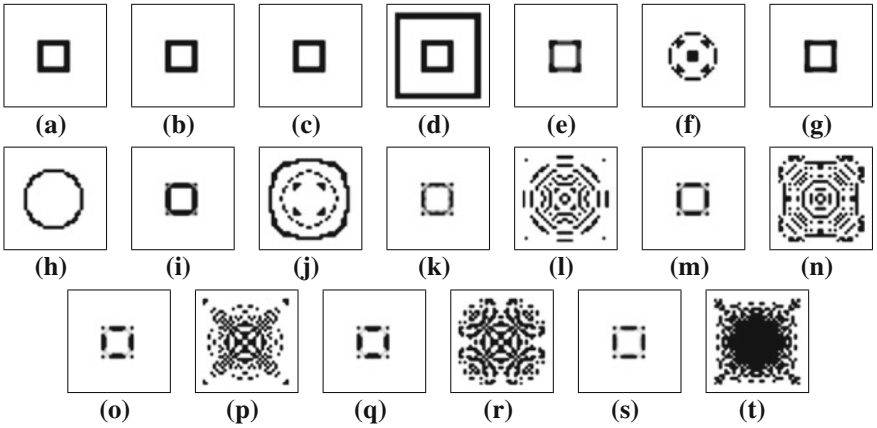


Fig. 6.2 Input and output pairs for FPCNN calculations for a square annulus input

angle will dominate. The input goes from a solid triangle to an edge mostly triangle to the three corners to the dominant corner.

The second test to be shown is that of a square annulus. The results of this test are shown in Fig. 6.2. This process took more iterations than the solid square so only a few are shown. For the initial iterations the behaviour of the network mimics the solid square case. The edges become dominant. However, the steady state image is a bit more complicated than the four corners.

In order for the input to stabilise, a few conditions must be met. First, the input values must be contained above a threshold. If the input values fall too low, then the decay (α terms) will dominate and the input will eventually disappear. The second condition is that the output must completely enter into the multiple pulse realms. In this scenario all of the output elements are always on. This condition can be seen in the examples. When this occurs, the feedback and the interconnections become constant in time. In actuality, the outputs are independent variables and it is the outputs that force all other fluctuations within the network. When the outputs stabilise, there exists no other force within the network that can alter the input or any other network value.

6.2 Object Isolation

Figure 6.3 shows a scheme that employs the PCNN and the FPF (fractional power filter [13]) in an iterative loop to isolate a target. This is useful for cases in which the target is dark and non-homogeneous.

There are two tools aside from the PCNN that are used in this system. The first is the FPF which is a composite filter that has the ability to manipulate the trade-off between generalisation and discrimination. It is detailed in Appendix C along with an explanation of the Python scripts. The pulse image and FPF filter are correlated in

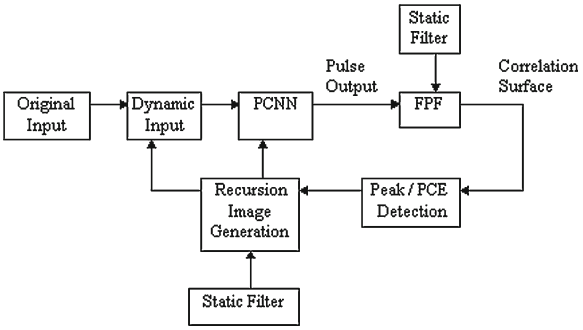


Fig. 6.3 The schematic of the feedback PCNN system

an attempt to find the target at any location in the image. The second tool that is used here is the correlation function which is also reviewed with scripts in Appendix D.

The PCNN creates a pulse image which may contain the outline of the target. A good example of this is shown in Fig. 4.15. Consider a case in which the man in the lower left is the target. He is shown with dark pixels that do not collectively pulse. However, in some iterations the background surrounding him does pulse and his outline is evident. The FPF is a filter constructed from the target that searches for an *edge encouraged* version of the target. If such a target is found through a peak detection then the system will enhance the input in that region with the shape of the target. There may be multiple regions in the image where a correlation peak may be found due to false positives. However, as the system iterates the true target will collect multiple enhancements and will eventually become the brightest object in the image.

The steps in the procedure are:

1. Normalise the input.
2. Create a normalised edge encouraged filter.
3. For each iteration.
    a. Compute one PCNN iteration.
    b. Edge enhance the pulse image.
    c. Compute the PCE of the correlation with the edge image and the filter.
    d. Detect peaks.
    e. RIG: Generate a feedback image.
    f. Modify the input and the PCNN.
4. The end result is the modified data image.

These steps are detailed with accompanying Python scripts in the following subsections. The final subsection creates a driver function for the entire process.

6.2.1 Input Normalisation

Given an input ($I[i, j]; i \in [1, V], i \in [1, H]$) where V and H are the vertical and horizontal frame size of the image, the normalisation process creates a new image that is a smoothed version with a specified range of pixel values. The modified input is,

$$J[i, j] = M\{0.6I + 0.4\}, \tag{6.3}$$

where $M\{\cdot\}$ is a smoothing operator which reduces random noise. The values of $J[i, j]$ are between 0.4 and 1.0 with the low value being raised up from 0 so that dark regions in the input still pulse synchronously. Given the file name of the input image the modified image is computed by the **LoadImage** shown in Code 6.1. Line 7 reads the image and Line converts the grey scale image into a matrix normalised with values between 0 and 1. In this case the matrix is rescaled so that the values are between 0.4 and 1 in Line 9. Line 10 uses the **cspline2d** function from the *scipy.signal* module to smooth the input image. The function receives the image file name and returns a matrix that is **J**.

6.2.2 Creating the Filter

The target construction starts with ($T[i, j] : i \in [1, V], i \in [1, H]$) which is a solid shape of the target centered in the frame. Rarely will the edges in the edge enhanced version of a pulse image coincide exactly with the target edges. Thus, the filter that is used to search for the target in the edge image is actually an *edge enhanced* version of the target. This is generated by manipulating the fractional power term in the FPF. (see Appendix C for details.) If the FPF were trained on a solid shape then the filter created with $\alpha = 0$ would be a duplicate of the shape. At the other end of the scale, when $\alpha = 2.0$ (the maximum value) the filter creates an edge only version of the input. Values between 0 and 2 encourage the edges on a non-linear scale.

Code 6.1 The **LoadImage** function.

```
# oi.py
from scipy.signal import cspline2d
import Image
import mgconvert

def LoadImage( fname ):
    mg = Image.open( fname )
    data = mgconvert.i2a( mg.convert('L'))/255.0
    data *= 0.6;  data += 0.4
    data = cspline2d( data, 2 )
    return data
```

There are three functions that produce the desired target. The first is **LoadTarget** (Code 6.2) which loads in the image of the target cut out from the original image. The cutout process here placed a white background for the pixels excluded in the cut out process. Line 7 performs a threshold operation so that the target has bright pixels and the background is black. The **Plop** function (see Appendix B.2) creates a frame that is the same size as the original image and places the centre of mass of the target at the centre of the frame. The inputs to the function are the name of the image file that contains the target cut out and the size of the original image. The output is a matrix that has binary values with the ON pixels being in the shape of the target.

Code 6.2 The **LoadTarget** function.

```
# oi.py
import geometry

def LoadTarget( fname, dataShape ):
    mg = Image.open( fname )
    targ = mgconvert.i2a( mg.convert('L'))/255.0
    targ = targ < 0.9
    targ = geometry.Plop( targ, dataShape )
    return targ
```

The second function is shown in Code 6.3. The **EdgeEncourage** function uses the fractional power filter (FPF) (Appendix C) to create an *edge encouraged* version of the target. Traditional edge enhancement processes extract the edges from input images and in the case of a binary valued input the edge enhancement would be the outline of the shape. Recognizing that the target may not exactly match the input it is necessary to relax the strictness of edge enhancement. Edge encouragement shows strong edges and an interior that decays towards the centre of the shape.

The result from the FPF is shown in Fig. 6.4a which shows the edge encouraged version of the target. The final process is to normalise the complex values of the target to create a zero sum filter. The output is shown in Fig. 6.4b and Code 6.4 shows the script.

The function **NormFilter** receives the output from **EdgeEncourage** and produces the output shown in Fig. 6.4b. Line 4 creates a circular region of interest (ROI) which has a radius large enough to encompass the entire target. Lines 5 and 6 normalise the region inside of this ROI such that it is zero sum. This helps reduce false positives generated during the filtering process. The circular region about the target has negative values.

Code 6.3 The **EdgeEncourage** function.

```
# oi.py
import numpy as np
import fpf

def EdgeEncourage( targ ):
    V,H = targ.shape
    X = np.zeros( (1,V*H), complex )
    X[0] = (np.fft.fft2(targ)).ravel()
    cst = np.ones( 1 )
    filt = fpf.FPF( X, cst, 0.3 )
    filt = filt.reshape((V,H))
    filt = np.fft.ifft2( filt ) * V*H
    return filt
```

(a) **(b)**

Fig. 6.4 The output of the FPF and the normalised version of the same. The normalised version is shown in reversed intensity with the *darkest* pixels displaying the highest energy

Code 6.4 The **NormFilter** function.

```
# oi.py
def NormFilter( filt, rad =128 ):
    V,H = filt.shape
    mask = geometry.Circle( (V,H), (V/2,H/2), rad )
    avg = (abs(filt)*mask).sum()/mask.sum()
    nfilt = mask*( abs(filt) - avg )
    return nfilt
```

6.2.3 Edge Enhancement of Pulse Images

Shapes are generally defined by the pixels along their perimeter and less so by the interior pixels. Thus, in matching the results from the pulse image to a target it will be the edge information that is useful. Another advantage of using edges is shown in the example in Fig. 4.15 in which the background is pulsing and creating the edge

around the target. Through edge enhancement of the pulse image it doesn't matter if the target is pulsing or if its background is pulsing.

A simple method of enhancing edges is to correlate the image with Sobel kernels. For a 2D image the kernels are,

$$G_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}; \; G_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The correlation of the G_x with an image will enhance the horizontal edges and the correlation with G_y will enhance the vertical edges. A single edge enhanced image is obtained by,

$$\mathbf{G} = \sqrt{(\mathbf{I} \otimes G_x)^2 + (\mathbf{I} \otimes G_y)^2}, \tag{6.4}$$

where $\otimes$ represents the correlation and $\mathbf{I}$ is the input image.

The *SciPy* package does contain a Sobel correlator and so this task is relatively easy to employ. Code 6.5 shows the computation of Eq. 6.4. Line 6 creates `g1` which is the correlation of G_x with the pulse image `Y[2]` shown in Fig. 4.15c. Likewise, Line 7 computes the correlation with G_y and Line 8 computes Eq. 6.4. The pulse image and edge enhanced versions are shown in Fig. 6.5 with the edges shown as dark pixels to facilitate viewing. As seen, even though the target did not pulse, there is a significant amount of target edge that is shown.

Code 6.5 The **EdgeEnhance** function.

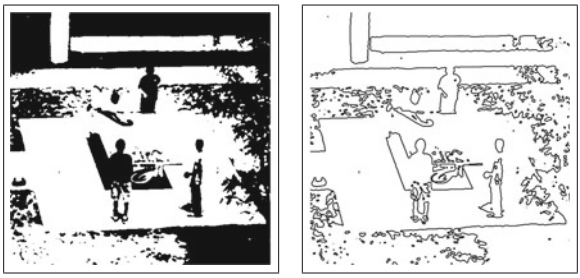
```
# oi.py
from scipy.ndimage import sobel

def EdgeEnhance( m ):
    temp = m.astype(float)
    g1 = sobel( temp, 0 )
    g2 = sobel( temp, 1 )
    gg = np.sqrt( g1*g1 + g2*g2 )
    return gg
```

Figure 6.5 shows a typical output from the PCNN and the edge enhanced version as generated by **EdgeEnhance**. The latter is shown in reversed intensity such that the high energy pixels are shown in black.

6.2.4 Correlation and Modifications

The correlation function is detailed in Appendix D which shows the function **Correlate** from the provided code. Correlations are merely dot products for every possible



(a) **(b)**

Fig. 6.5 Figure **a** is the output `Y[2]` from the network, and **b** is the edge enhancement of this pulse image

relative shift between the input and the kernel. In this case the input will be edge enhanced versions of the pulse images and the kernel will be the edge encouraged version of the target.

Usually, at the location of the target the correlation surface will contain a bright peak. However, there can be multiple other locations in which the correlation surface has bright values. Thus, an additional step of isolating peaks instead of bright correlation pixels is adopted.

The begins with the computation of the peak-to-correlation energy (PCE) [41]. This divides the peak by the localized energy of the correlation as in,

$$P_{ij} = \frac{C_{ij}}{\sum_{k=i-\delta}^{i+\delta} \sum_{k=j-\delta}^{j+\delta} |C_{kl}|^2}, \tag{6.5}$$

where C_{ij} is the individual values of the correlation surface and δ is a small radius.

The **PCECorrelate** function shown in Code 6.6 computes the correlation and the PCE. These are accomplished by functions in the *correlate.py* module which are detailed in Appendix D. The output is a single matrix in which the high energy spikes are potential locations of the target.

Code 6.6 The **PCECorrelate** function.

```
# oi.py
import correlate

def PCECorrelate( edj, filt ):
    corr = correlate.Correlate( edj, filt )
    if corr.max() > 0.001:
        pce = correlate.PCE( corr, 2400 )
        return pce
    else:
        return corr
```

6.2.5 Peak Detection

There should be a small number of spikes in the PCE if there are potential targets. These spikes may be wider than a single pixel and some may be false positives. The peak detection algorithm recursively detects a the largest significant peak and then removes the surrounding area from further consideration.

The process is handled by the **Peaks** shown in Code 6.7. The inputs are the PCE surface and the binary valued target (from **LoadTarget**). Line 9 creates a mask that is 8 pixels larger than the target in all directions. This is used in Line 17 to remove from consideration any region that has detected a peak. The peak is detected in Line 12 and if the peak is too small the process is terminated. If the peak is significant then Line 16 archives the peak location and the Lines 17 and 18 prevent another peak being found in the general vicinity. The output is a list of (x, y) points were the PCE had spikes.

Code 6.7 The **Peaks** function.

```
# oi.py
from scipy.ndimage import shift, binary_dilation

def Peaks( corr, targ, gamma = 0.75):
    temp = corr + 0
    pks = []
    ok = 1
    V,H = temp.shape
    etarg = binary_dilation( targ, iterations=8 ).astype(int)
    while ok:
        v, h = divmod( temp.argmax(), H )
        if temp[v,h] < gamma:
            ok = 0
            break
        else:
            pks.append( (v,h) )
            mask = shift( etarg, (v-V/2,h-H/2) )
            temp *= (1-mask)
    return pks
```

6.2.6 Modifications to the Input and PCNN

Once the spikes have been detected it will be necessary to modify the input and PCNN through the RIG. Basically, the input is enhanced by the target shape at the location of the spikes and the threshold of the PCNN is lowered in the same regions. Formally,

$$J[n; i, j] = 0.9\, J[n-1; i, j] + 0.15\, K[i, j], \tag{6.6}$$

where n represents the iteration index and $K[ij]$ is the mask that contains the solid shapes of the target centered at each spike. Likewise,

$$\Theta[n; i, j] = \begin{cases} 0.9\,\Theta[n-1; i, j] & K[i, j] = 1 \\ \Theta[n-1; i, j] & \texttt{Otherwise} \end{cases}. \tag{6.7}$$

Function **Enhance** shown in Code 6.8 performs the first operation. The loop starting in Line 6 replicates the shape of the mask at each spike location. Line 11 modifies the input accordingly. The modification of the PCNN is performed in the next section.

Code 6.8 The **Enhance** function.

```
# oi.py
def Enhance( data, pks, targ ):
    N = len( pks )
    temp = np.zeros( data.shape )
    V,H = data.shape
    for i in range( N ):
        vs, hs = pks[i][0]-V/2, pks[i][1]-H/2
        temp += shift( targ, (vs,hs) )
    temp = (temp > 0.001).astype(int)
    if temp.sum() > 1:
        data = 0.9*data + 0.15*temp
    return data, temp
```

The result after the first spike detections is shown in Fig. 6.6. There are two locations in which the input has been enhanced and one is on the target. There is also a false positive on the lower left mainly due to the distance between the left edge of the frame and the edge of the concrete. This width matches the width of the shoulders of the target and both widths are delineated by vertical edges. However, subsequent iterations of the process will continually enhance the real target and the false targets will fade.

Fig. 6.6 The first enhancement of the target. Potential targets are enhanced, and in this case the enhancements include the actual target and one false positive

Fig. 6.7 The final result with the intensity reversed for viewing. Some false positives do exist but the true positive is very distinct from them

6.2.7 Drivers

There are two functions provided to drive the entire process. The first is the steps required for a single iteration and the second runs multiple iterations. The **SingleIteration** function shown in Code 6.9 performs the steps of a single iteration. It runs a single PCNN iteration in Line 3, enhances the pulse image in Line 4, and computes the correlation in Line 5. The PCE is computed in Line 6 and used to detect peaks in Line 7. This modifies the input in Line 8 and the threshold array in the PCNN in Line 9. The function returns the modified input.

Code 6.9 The **SingleIteration** function.

```
# oi.py
def SingleIteration( net, data, filt, targ ):
    net.Iterate( data )
    edj = EdgeEnhance( net.Y )
    corr = abs(correlate.Correlate( edj, filt ))
    pce = correlate.PCE( corr )
    pks = Peaks( pce, targ, 0.75 )
    data, mask= Enhance( data, pks, targ )
    net.T = mask*net.T*0.9 + (1-mask)*net.T
    return data
```

The final function is the **Driver** shown in Code 6.10. This loads in the original image and target and calls the functions necessary to normalise them. Then it runs 15 iterations of the process continually modifying the input. The call to the function in shown in Line 13 and the result generated by Line 14 is shown in Fig. 6.7. This image is reversed so that the highest energies are shown in dark pixels.

The target is continually enhanced by the process and the few false positives that appear are soon atrophied. This process isolated a dark target with inhomogeneous texture using the pulse images and the FPF.

6.3 Dynamic Object Isolation

Dynamic Object Isolation (DOI) is the ability to perform object isolation on a moving target. A system to do this has two alterations to the static object isolation system discussed above. The first is that it trains the filter on many differing views of the target and the second is that it must be optimised to recognise the target in frames that were not used in training. This second goal forces the system to exhibit generalisation abilities since the target may be presented differently in the non-training views. Using the example of the boy kicking the ball, the non-training views would show the boy in a different configuration (arms and legs and new angles), different orientations and scale (as he moves towards the camera).

Code 6.10 The **Driver** function.

```
# oi.py
def Driver( ):
    data = LoadImage( 'pingpong.png' )
    targ = LoadTarget( 'ping.jpg', data.shape)
    filt = EdgeEncourage( targ )
    filt = NormFilter( filt )
    net = CreatePCNN( targ.shape )
    for i in range( 15 ):
        print 'iteration ', i
        data = SingleIteration( net, data, filt, targ )
    return data

>>> data = Driver()
>>> mgconvert.a2i( data ).show()
```

The difference in the configuration of the system is that the filter is trained on several views of the target. A sequence of images are shown in Fig. 6.8 in which the target (small boy) is moving. This target is also dark and inhomogeneous. The FPF is a composite filter which can train on multiple images. Figure 6.9 shows the FPF trained on images from Fig. 6.8a,b,d,e but not c. The iterative process of Fig. 6.3 is then applied to the c image and the enhancing process is shown in Fig. 6.10.

6.4 Shadowed Objects

The PCNN relies heavily upon the intensity of the input pixels. Thus, shadowed objects tend to produce a radically different response in the PCNN. A shadowed object is one that has a diminished intensity over part of itself. Consider an object that has two segments A and B, which originally have very similar intensities and therefore would pulse in the same iteration. A shadow falls upon segment B and its

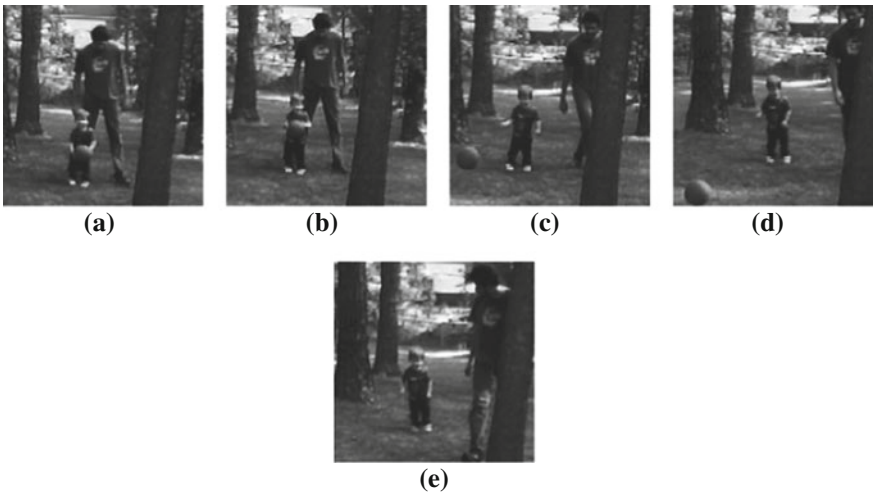


Fig. 6.8 A sequence of five input images

Fig. 6.9 The composite filter

Fig. 6.10 The progression of the dynamic input

intensity is reduced. Now, due to its lower intensity, the segment pulses in frames subsequent to that of A. The object now has separate pulsing activity.

For many PCNN-based architectures this can be a devastating effect and can destroy effective processing. In the object isolation architecture, however, the FPF has the ability to overcome the detrimental effects of shadows. Since the FPF has the fractional power set to include discriminatory frequencies and the pulse segments have sharp edges, a sufficient correlation occurs between the filter and the pulsing

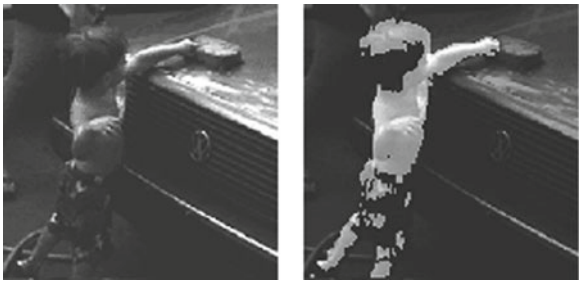
(a) **(b)**

Fig. 6.11 **a** The shadow mask. All pixels in the lower half of the image were used to decay the values of the original image. **b** The shadowed input

of only a part of the target. In other words, the filter can still provide a good enough correlation with segment A or B to progress the object isolation.

Consider the images in Fig. 6.11. In Fig. 6.11b is a shadowed image. This image was created from an original image in which the target (the boy) was cut-out and binarised (Fig. 6.11a). This binary image became the shadow mask and Fig. 6.11b was created by reducing all pixels in the lower half of the image that were ON in Fig. 6.11a. The effect is that the boy's shorts and legs were shadowed. The FPF filter and feedback mask were created with pulse images from the non-shadowed image.

The shadowed area intensity was sufficient to get the boy's shorts and legs to pulse in frames later than the torso and arms. However, the FPF filter was still able to find the partial target pulsing. The progression of the shadowed input is shown in Fig. 6.12.



(a) **(b)**

Fig. 6.12 As the system iterates the target, even though originally in shadows, it is progressively enhanced

Fig. 6.13 An input stimulus

6.5 Consideration of Noisy Images

Random noise is an enemy of the PCNN. Pulse segments are easily destroyed by random noise. Noise can enter the system in three basic ways. The first is input noise in which **S** has noise added, the second is system noise in which noise is added to **U**, and the third is a random start in which Θ is initially randomised. Any of these cases can destroy the PCNNs ability to segment. Consider the stimulus image shown in Fig. 6.13, which shows a boy (kicking a football), his father and some trees.

Using the input stimulus shown in Fig. 6.13, the original PCNN produces the temporal outputs of binary images shown in Fig. 6.14. Segmentation and edge enhance-

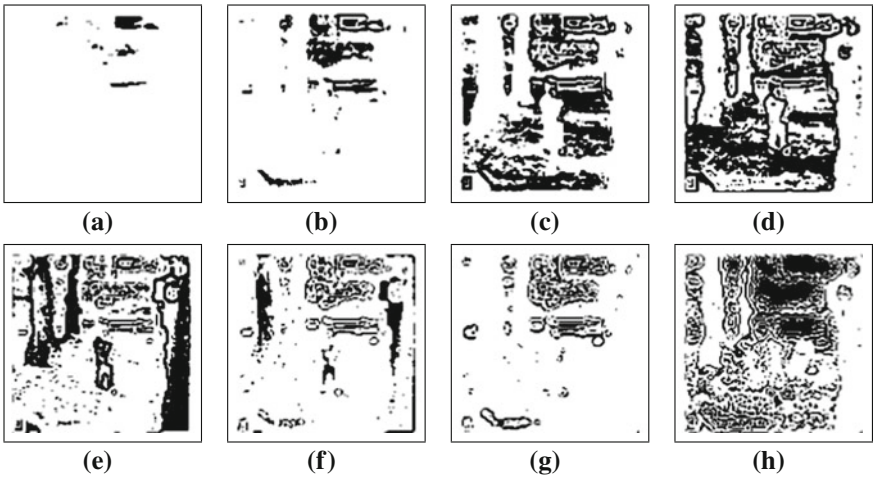


Fig. 6.14 Outputs of a PCNN with Fig. 6.13 as a stimulus

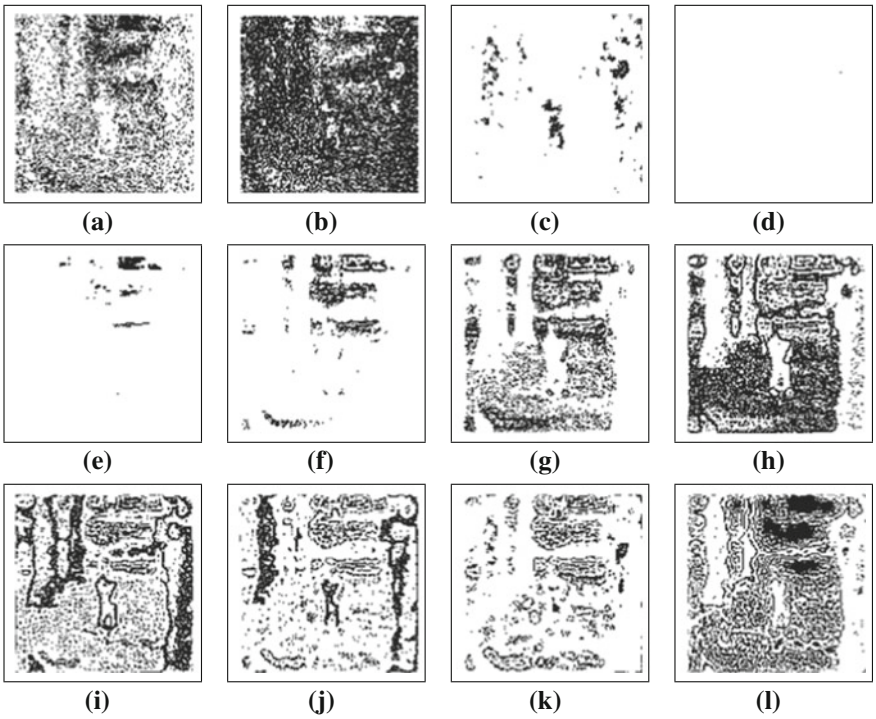


Fig. 6.15 Outputs of the PCNN with random initial threshold values

ment are evident in the outputs shown. Compare these outputs to a system that initialised the threshold to random values between 0.0 and 1.0. It should be noted that the initial values are less than 5 % of threshold values after a neuron pulses. The results of this experiment are shown in Fig. 6.15.

Certainly, the segments in the output are noisier than in the original case. This is expected. It should also be noted that the PCNN did not clean up the segments very well. There are methods by which this problem can be ameliorated.

The first method to be discussed for the reduction of noise uses a signal generator as a post-processor to the PCNN. This generator will produce small oscillations to the threshold values, which are in synchronisation with the natural pulsing frequency of stimulated neurons. The segments then tend to synchronise and noise is therefore significantly reduced.

A typical signal generator is the addition of a cosine term (where f is the design frequency) to the threshold during the creation of the output,

$$Y_{ij}[n] = \begin{cases} 1 & \texttt{if } U_{ij}[n] > Q_{ij}[n+1] + (\cos(fn/2p) + 1.0) \\ 0 & \texttt{Otherwise} \end{cases}. \qquad (6.8)$$

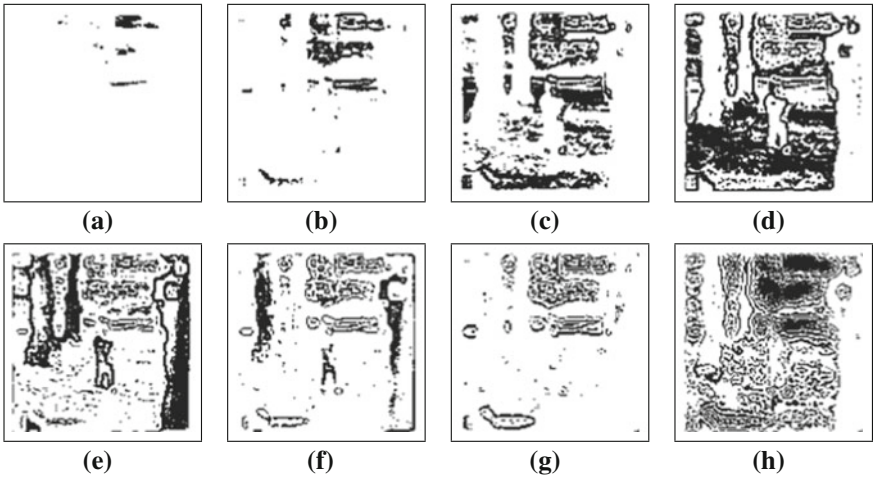


Fig. 6.16 Outputs of a PCNN with a signal generator

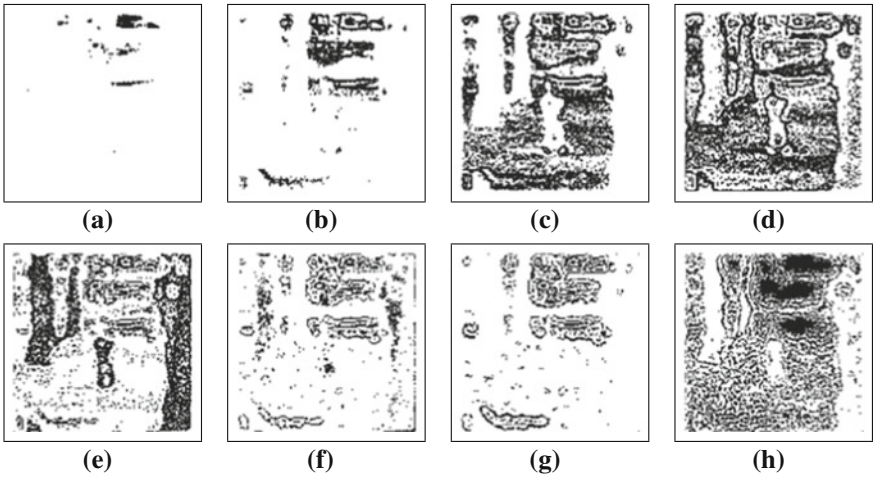


Fig. 6.17 Outputs of a PCNN with a signal generator and a noisy stimulus

The outputs are shown in Fig. 6.16.

The noise of the system is now almost eliminated. The threshold levels have been synchronised by the continual addition and then subtraction of small values to the comparison function. The function periodically delays the pulsing of some neurons as the generator produces a larger output. When the generator produces a smaller output the pent-up potential of the neurons begins pulsing.

Noise can occur in other parts of the system. For example the input stimulus can be noisy. In Fig. 6.17 the output of a PCNN with a signal generator is shown for

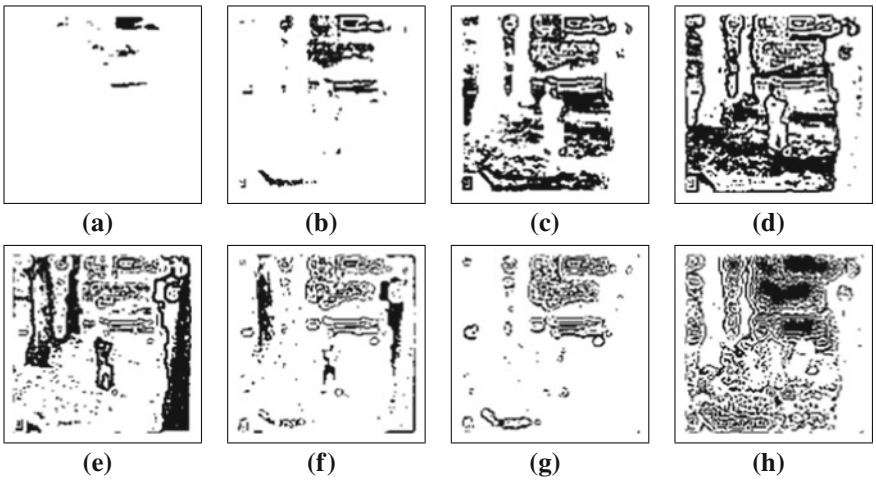


Fig. 6.18 Outputs of the PCNN after noise added to **U** for each iteration

the case where the stimulus has random values between −0.1 and 0.1 added to the elements.

This system virtually eliminates the noise for the early iterations. However, the noise returns in subsequent iterations. The reason is quite simple. Each iteration has the stimulus added to **F**. Thus, constant noise continually accumulates. Noise begins appearing in the output when the generator can no longer overcome the accumulation of the noise.

The last example adds dynamic noise to the system. In other words, our noise generator adds random zero mean noise is added to U each iteration. The values of the additional noise are $[-0.1, 0.1]$. The results of this test are shown in Fig. 6.18. As can be seen, the noise is considerably reduced. In this case the noise was different for each iteration. These cancelling effects allowed the system to operate in a similar way to that of Fig. 6.17.

Another method of reducing the noise is to employ the fast linking algorithm. This was demonstrated earlier in Sect. 4.1.7 using the same example.

6.6 Summary

Recursive or feedback systems allow the pulsing process of the PCNN to key on particular targets. This feedback alters the PCNN and/or the input image. This allows the PCNN to extract information that is rather difficult to obtain in the standard model. Recursive systems provide the advantage in that a system can accumulate knowledge rather than attempt a decision in a single process. For cluttered images this has the advantage of deciding the difference between false positives and the real target as demonstrated in an example.

Chapter 7
Recognition and Classification

7.1 Aircraft

Consider the image in the upper left corner of Fig. 7.1. This image gives a grey scale input to a PCNN and the subsequent frames show the temporal series of outputs from a PCNN. Close to perfect edge detection is obtained and there are no problems identifying the aircraft (e.g. from image number 3). Here the damaged wing tip is also easily seen.

However, this is not the case with a more complicated background such as mountains. Figure 7.2 shows a case where it is much harder to identify the aeroplane. A subsequent correlator is needed, and the selection is the fractional power filter (FPF) [13] discussed in Appendix C.

The PCNN has been used as the first stage in a hybrid neural network [112] for Automatic Target Recognition (ATR). The time series provided a series of unique patterns of each of four aeroplanes (F5XJ, MIG-29, YF24 and the Learjet) used in this work. The input image to the ATR system was a 256×256 pixel image using 8 bit grey scale. The input data used for training the networks was obtained from simulated move-sequences of each aeroplane. The sequence includes large variations in scale and 3D orientation of the aeroplanes. However, not all angles (and scale sizes) were included in the training data. This was done particularly in order to evaluate the generalisation capability of the system. Only the non-zero components of the first period of the PCNN 1D time series were used as input to the subsequent neural networks. The results [112] for several such 'conventional' neural network classifiers are shown in Table 7.1. The number of inputs was in all cases 43. Different neural networks were tested as final 'classificator' of PCNN output, The Logicon Projection Network™, LPN [113], the back propagation network, BP, The Radial Basis Function network, RBF, using two different algorithms, the Moody-Darken algorithm [73]. The numbers in Table 7.1 represent the mean value of correct classification of Yes/No, in percentage, for each of the four classes, together with the standard deviation, σ, of each class. The LPN and the BP get total average results that are nearly equal. However the LPN is always better classifying the signal (Yes),

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_7, © Springer-Verlag Berlin Heidelberg 2013

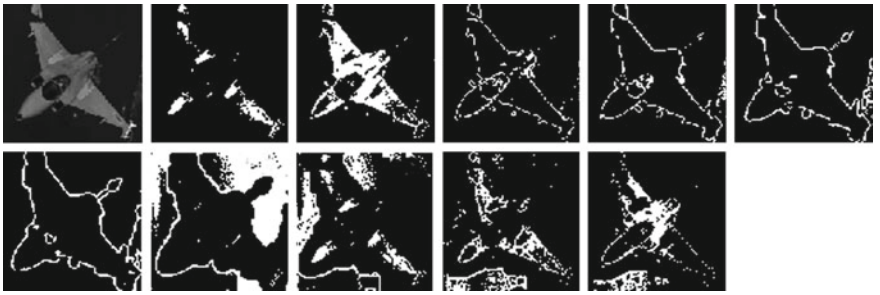


Fig. 7.1 The SAAB JAS 39 Gripen aircraft as an input to the PCNN. The initial sequence of temporal binary outputs are shown

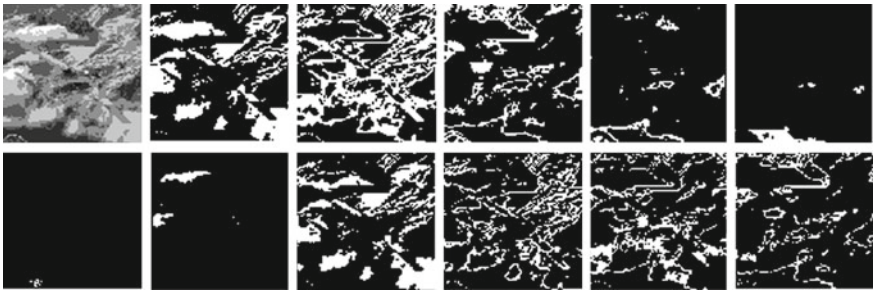


Fig. 7.2 The *upper left* image shows an aircraft as an input to the PCNN. However, this time an aeroplane is flying upside down in front of some Swiss Alps. It is hard to 'see' the aeroplane in the original input as well as in the temporal binary outputs

Table 7.1 Results of correct classification (in %)

Net	F5XJ		Mig29		YF24		LearJet	
	Yes	No	Yes	No	Yes	No	Yes	No
LPN	95	86	90	85	86	87	92	87
BP	85	93	81	89	82	92	88	93
MD	82	78	83	90	72	90	90	88

and BP is always better classifying the background (No). Moody-Darken network showed large standard deviations in several tests, especially for classifying the signal (Yes) for the F5XJ and YF24 aeroplanes.

7.2 Aurora Borealis

Auroras are spectacular and beautiful phenomena that occur in the auroral oval regions around the polar caps. Here geomagnetic field lines guide charged particles (mainly electrons and protons of magnetospheric or magnetosheath origin) down to

ionospheric altitudes. When precipitating, the particles lose their energy via collisions with the neutral atmosphere, (i.e. mainly oxygen and nitrogen atoms). At an altitude range of about 75–300 km, some of the atmospheric constituents can be excited to higher energy levels and this can lead to the formation of auroral light. The auroras during magnetospheric substorms and, especially, the great auroras during magnetospheric storms can create extremely impressive and beautiful spectacles. However, they are different (Fig. 7.3).

7.3 Target Identification: Binary Correlations

One application of the FPF is to find targets in an image. In this case a single image is used for the recognition of the target and the fractional power term is adjusted to increase the importance of edge information. Consider the first image in Fig. 7.4a

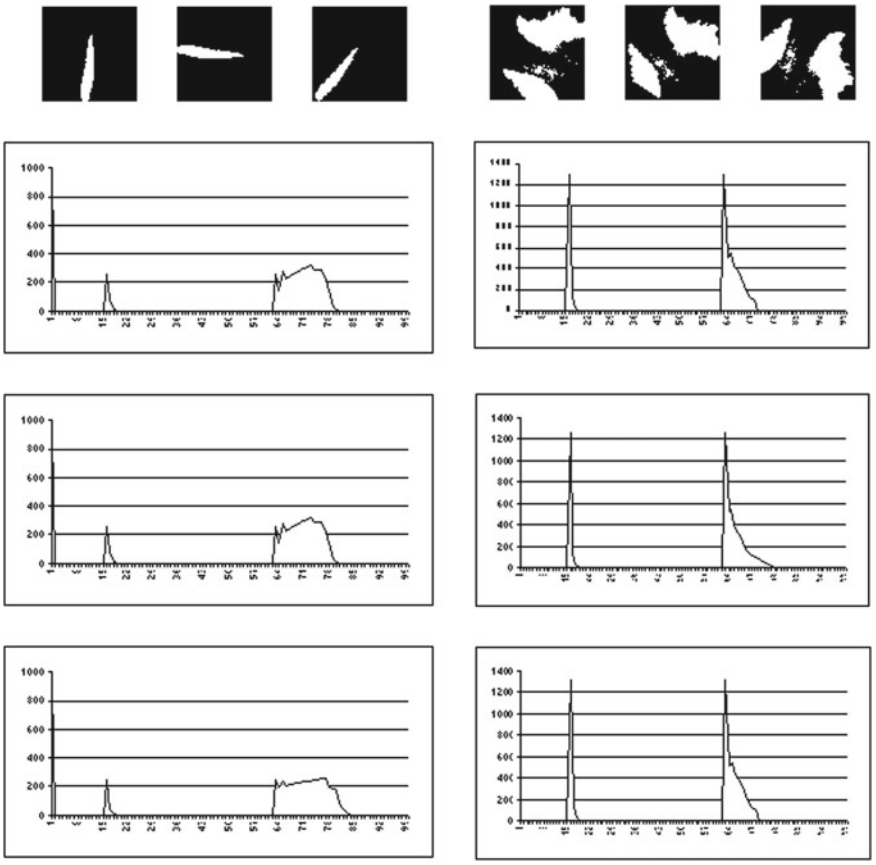


Fig. 7.3 Examples of Aurora Borealis and their resulting time signals when presented to a PCNN. A single arc (*left*) and a double arc (*right*) both retain their respective time signals when the images are rotated (*top*)

(a) **(b)**

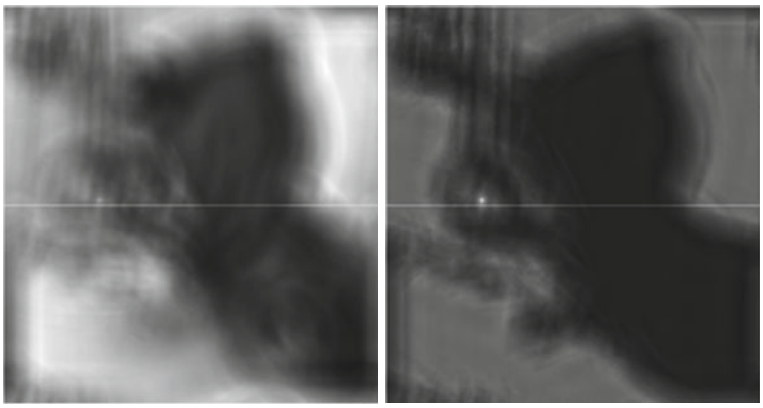
Fig. 7.4 **a** is the original input and **b** is the isolated fist to be used as the target

and the task of identifying his hand. The procedure would first mask the target region and place it in the centre of a frame as shown in Fig. 7.4b.

This image can be correlated with the original image and the FPF will force a positive response at the location of the target. At the centre of the target in the original image the correlation value will be close to 1 since that was the value set in the constraint vector **c**. However, none of the other correlation surface values are similarly constrained except that as the value of the fractional power term is increased the overall energy of the correlation surface is reduced. This is a trade-off and as the fractional power term is increased the filter has an increasingly difficult time identifying anything that is not exactly like the training image. This discrimination also includes any windowing operation. Even though the hand in the filter is exactly like the hand in the original image the fact that the filter has a window about the target would detrimentally affect the filter response if the fractional power term became too high.

Thus, the trade-off between generalisation and discrimination can prevent a good solution from being achieved. Figure 7.5a displays the correlation between the image in Fig. 7.4a and the filter made from Fig. 7.4b with $\alpha = 0.3$. There is a small white dot at the centre of the target which indicates a strong correlation between the image and the filter. However, the filter also provides significant responses to other areas of the target. The conclusion is that the filter is generalising too much. If the fractional power term is increased to lower the generalisation then it also loses the ability to find the target.

By employing the ICM this problem is solved. The ICM will create several pulse images and in one the target will pulse. Complicated targets may their pulses spread over a few iterations, but in this case the hand is a simple target. The reason that this is important is that the pulse image presents the target as a solid object with sharp lines. This is much easier for a correlation filter to detect. The image in Fig. 7.5b



(a) **(b)**

Fig. 7.5 **a** is the correlation with an FPF and $\alpha = 0.3$, and **b** is the correlation of a pulse image ($n = 1$) with $\alpha = 0.3$

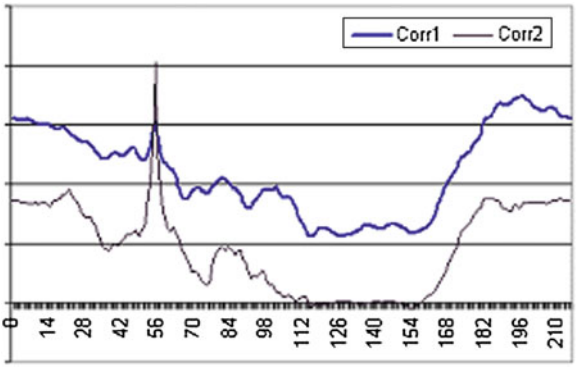


Fig. 7.6 Slices through the two correlation surfaces. Clearly in the second case the FPF can find the target from the ICM pulse image

displays the correlation of a pulse image and an FPF built from a binary version of the target.

The bright spot clearly indicates the detection of the target without interference from other objects in the image. To further demonstrate the detection ability using the ICM the image in Fig. 7.6 displays a slice through each correlation surface (Fig. 7.5a, b) through the target pixel. These are the values of the pixels along the horizontal row in Fig. 7.5b that passes through the target. Clearly, the method using the ICM provides a much more detectable response.

The employment of the ICM in this case is not a subtle affect. One of the unfortunate features of a correlation filter is that it prefers to find targets with higher energy. Darker targets are harder to find. In the ICM the darker objects will eventually create a pulse segment and in that iteration the target will be the bright object making it

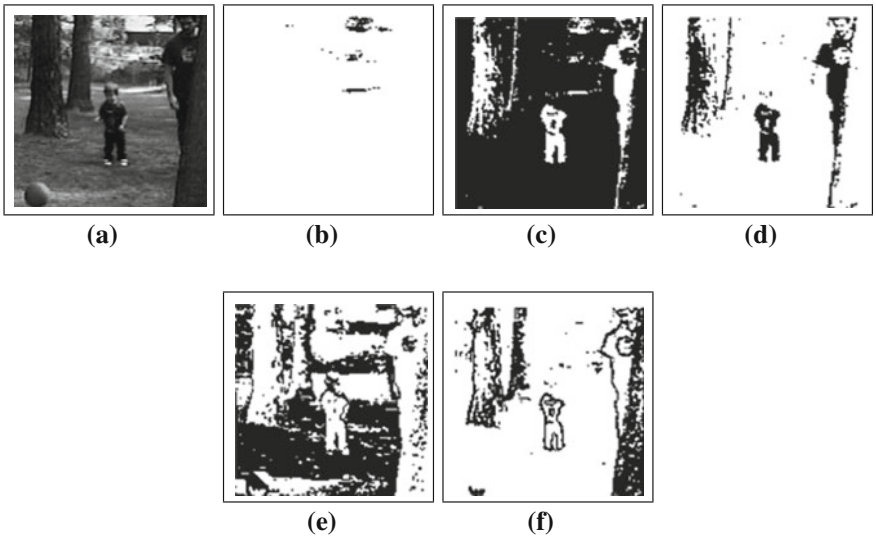


Fig. 7.7 An original image and some of the pulse images. In one of the pulse images the target is the bright object and in another the outline of the target is the bright object

easier for the filter to identify it. Consider the images in Fig. 7.7 in which there is an image with a young boy. In this case the boy is that target and the dark object. It would be different to build a filter to find the boy since he is so dark. Also in this figure are some of the pulse images. In the third pulse image the boy's pixels pulse. In this iteration he is now the high energy object and it much easier to find with a filter. For display purposes the neurons that pulse are shown in black.

Again, building an FPF from the pulse image the identification of the dark target is easily performed. The correlation is shown in Fig. 7.8 and the slice of the correlation surface through the target pixel is shown in Fig. 7.9. In this case the tree trunks pulsed in the same iteration and so they also correlate with the filter. Unfortunately, in this case the tree trunks are about the same width as the boy and so they provide strong correlations. However, the FPF has the ability to increase the discrimination and thus

Fig. 7.8 The correlation response of the pulse image and an FPF. The *darkest spot* depicts the highest peak

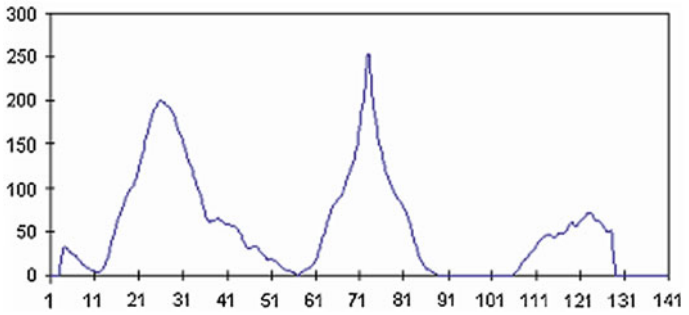


Fig. 7.9 A slice through the centre of the target of the correlation surface. The x-axis spans the horizontal dimension of the correlation surface and the y-axis represents the strength of the correlation

the correlation signal of the trees is not as sharp as that of the boy. The remainder of the process merely finds large, sharp peaks to indicate the presence of a target. The peak belonging to the boy in this case is definitely the sharpest peak and now a decision is possible.

It was the duty of the ICM to extract the target and present it as a high energy collective pulse segment. A Fourier filter such as the FPF could easily detect the target and produce a sharp correlation signal. Whereas, a filter operating on the original image would have a very difficult time in finding the target.

7.4 Galaxies

Contributed by Soonil Rughooputh, University of Mauritius

Astronomers predict that the universe may potentially contain over 100 billion galaxies. Classification of galaxies is an important issue in the large-scale study of the Universe. Understanding the Hubble sequence can be very useful in understanding the evolution of galaxies, cosmogony of the density-morphology relation, role of mergers, understanding how normal and barred spirals have been formed, parameters that do and do not vary along it, whether the initial star-formation rate is the principal driver, etc.. Our understanding of them depends on how good the sensitivity and resolving power of existing telescopes (X-ray, Optical, Infra-red, Radio, etc.). Database construction of these galaxies has only just begun. Recognition or classification of such large number of galaxies is not a simple manual task. Efficient and robust computer automated classifiers need to be developed in order to help astronomers derive the maximum benefit from survey studies. There are different ways of classifying galaxies, for example, one can look only at the morphology. Even in this classification scheme there exists different techniques. One can combine morphology with intrinsic properties of the galaxy such as the ratio of random to rotational velocities, amount of dust and gas, metallicity, evidence of young stars, spectral lines present and their widths, etc.

Nobody believes that galaxies look today as they did just after they were formed. Newly born galaxies probably did not fit the present-day classification continuum. The big puzzle is to find out how the galaxies evolved onto the present forms. Did the primeval galaxies "evolve" along basically the present Hubble sequence, or did they fall into the sequence, each from a more primitive state, 'landing' at their present place in the classification according to some parameter such as the amount of hydrogen left over after the initial collapse? [95]. The collapse occurred either with some regularity, [27], or chaotically [96] within a collapsing envelope of regularity, or, at the other extreme, in complete chaos. Present-day galaxies show variations of particular parameters that are systematic along the modern classification sequence. The obvious way to begin to search for the 'master parameter' that governs the formation process is to enumerate the variation of trail parameters along the sequence. Reviews by [87] and by [15] indicate somewhat different results that nevertheless are central to the problem. Within each morphological type, there is a distribution of the values of each parameter, and these distributions have large overlap from class to class. Hence, the dispersion of each distribution defines the vertical spread along the ridgeline of the classification continuum (i.e. the center line of Hubble's tuning fork diagram). Hence, besides a 'master parameter' that determines the gross Hubble Type (T), there are other parameters that spread the galaxies of a given type into the continuum of L values [11]. The fact that so many physical parameters vary systematically along the Hubble sequence is strong evidence that the classification sequence does have a fundamental significance.

The easiest property of a galaxy to discuss is its visual appearance. When Hubble introduced his classification, he thought it might represent an evolutionary sequence with galaxies possibly evolving from elliptical to spiral form but, this is not believed to be true today. Hubbles classification scheme, with some modifications, is still in use today. Galaxies are classified as spiral galaxies (ordinary spirals, barred spirals), lenticulars, elliptical galaxies and irregular galaxies; including other more specialized classifications such as as cD galaxies. The spirals are classified from Sa to Sc (ordinary spirals) and from SBa to SBc (barred spirals); a to c represent spiral arms that are increasingly more loosely wound. The elliptical galaxies are classified according to their ratio of their apparent major and minor axes; the classification is based on the perspective from Earth and not on the actual shape. The lenticulars are intermediate between spirals and ellipticals. There are other classification schemes like de Vaucouleurs, Yerkes, and DDO methods, which look into higher details.

The morphological classification of optical galaxies is done more or less visually. Better classification schemes would certainly help us to know more about the formation and evolution of galaxies. A technique that involves the use of robust software to do the classification is crucial, especially if we want to classify huge number of galaxies at one shot. Since there are billions of galaxies, a robust automated method would be desirable. Several authors have reported work along this line (see references 28–37 in [91]). The techniques studied include the use of statistical model fitting, fuzzy algebra, decision tree, PCA, and wavelet-based image analysis. Some work has been reported on the use of artificial neural networks for automatic morphological classification of galaxies; using feed-forward neural network, self-organizing

maps, computer vision technique—see references 32–34 in [91]. These techniques, however, require extensive training, hence are computationally demanding and may not be appropriate for the classification of a large number of galaxies. A galaxy classifier/identifier using PCNN has also been reported; [91, 101] initial results of which are promising. These authors have been able to classify galaxies according to an index parameter obtained from the time signature of the galaxies. The results reveal that this technique is fast and can be used for real-time classifications. The researchers have chosen a catalogue of digital images of 113 nearby galaxies [31] since these galaxies are all nearby, bright, large, well-resolved, and span the Hubble classification classes. Besides, Frei et al. photometrically calibrated all data with foreground stars removed and the catalogue is one of the first data set made publicly available on the web. Important data on these galaxies published in the "Third Reference Catalogue of Bright Galaxies" [111] are recorded in the FITS file headers. All files are available through anonymous FTP from "astro.princeton.edu"; they are also available on CD-ROM from Princeton University Press.

Binary barcodes corresponding to galaxies can be generated to constitute a databank that can be consulted for the identification of any particular galaxy (e.g. using the N-tuple neural network). The digital image of a galaxy is first presented as input to a PCNN to produce segmented version output of binary images. Figure 7.4 shows a set of original images of representative galaxies spanning over the Hubble classification classes; corresponding NGC values are given in Table 7.2. Figure 7.11 shows the original images of representative galaxies spanning over the Hubble classification classes (from top to bottom—NGC 4406, NGC 4526, NGC 4710, NGC 4548, NGC 3184, and NGC 4449) and the corresponding segmented images for the first five iterations (column-wise). Figure 7.12 shows the set of the third iteration images for a number of galaxies listed in Table 7.2 for producing the time signatures using a second PCNN. The segmented image version was used instead of the original image to minimize adverse effects of galactic halos.

A method was devised to mathematically compute an morphology index parameter (mip) from the first few iterations (mip $= G(3)^2/(G(2)G(4))$ since these are related to the image textures and hence retain useful information on the morphology of the galaxies [101]. We found that galaxies (except for NGC4472) with mip values less than 10 are spirals or irregulars otherwise ellipticals or lenticular (refer to Table 7.2). This exception may be due to the presence of halos. Figure 7.13 shows the corresponding barcoded images of the galaxies listed in Table 7.3 (obtained using the corresponding. 8-bit grey level version of the time signatures). We note that the 1:1 correspondence between the barcodes and the input NGC images.

Table 7.2 NGC values for galaxies in Fig. 7.10

3184	3726	4254	4374	4477	4636	5813
3344	3810	4303	4406	4526	4710	6384
3351	3938	4321	4429	4535	4754	4449
3486	4125	4340	4442	4564	4866	4548
3631	4136	4365	4472	4621	5322	5377

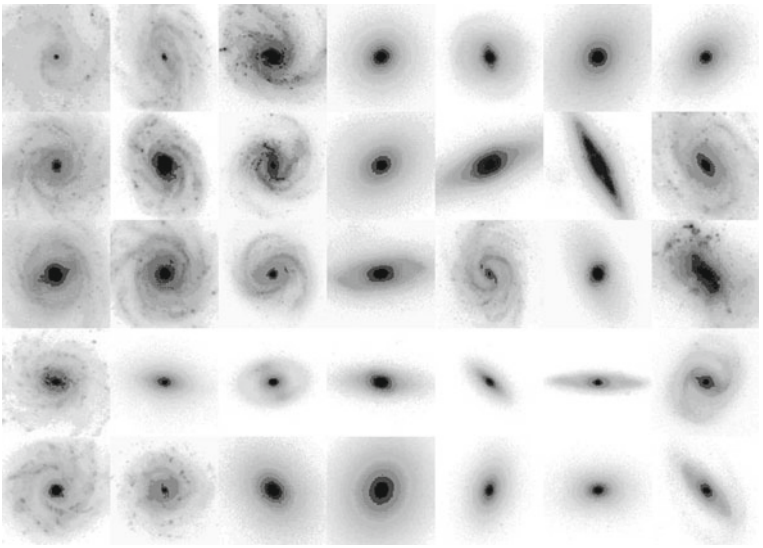


Fig. 7.10 Representative galaxies

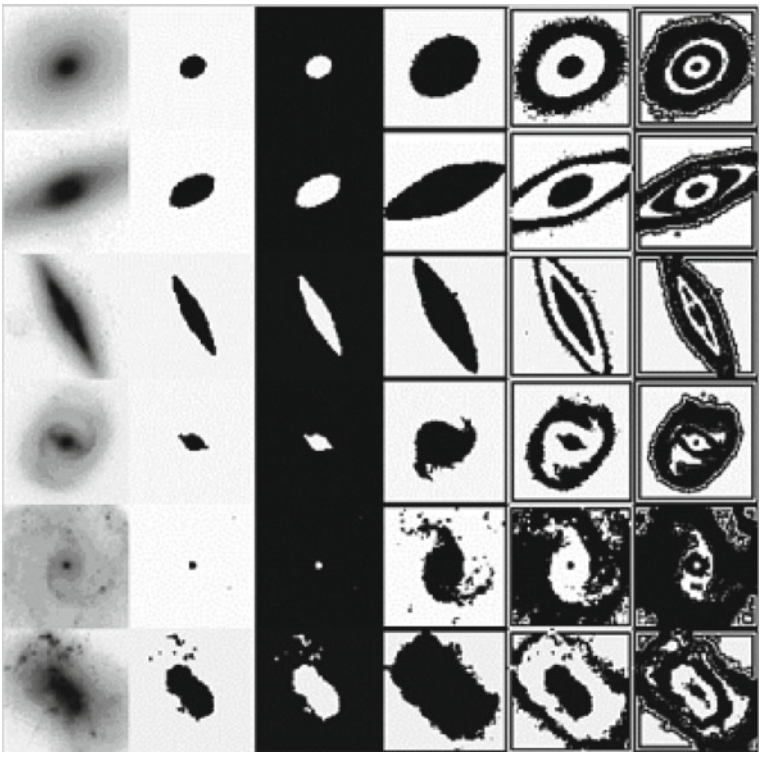


Fig. 7.11 Representative galaxies and their PCNN segmented images

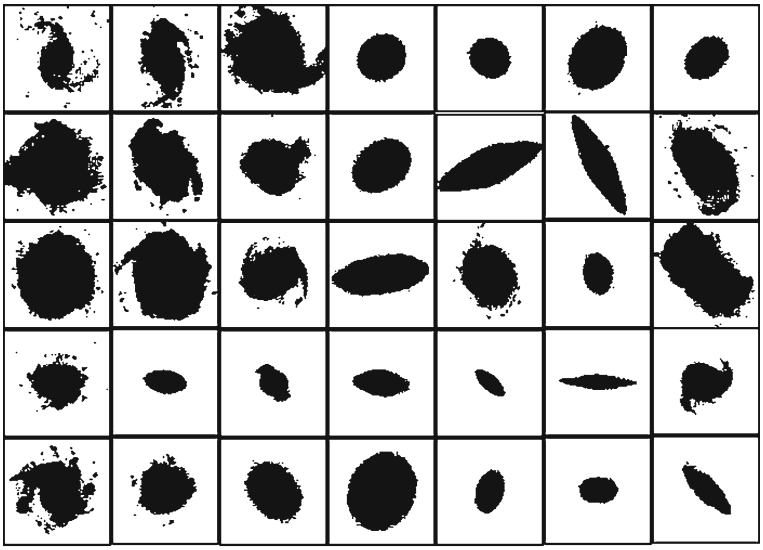


Fig. 7.12 Third iterated PCNN binary images of galaxies (see Table 6.3 for corresponding NGC values)

Table 7.3 Galaxies of different Hubble types (T) with the mip computed

NGC	T	Mip	NGC	T	Mip	NGC	T	Mip
3184	6	5.3	4303	4	9.2	4526	−2	11.1
3344	4	3.8	4321	4	7.2	4535	5	7.6
3351	3	4.9	4340	−1	16.3	4564	−5	18.1
3486	5	8.1	4365	−5	10.5	4621	−5	16.3
3631	5	4.6	4374	−5	13.9	4636	−5	10.9
3726	5	5.1	4406	−5	12.0	4710	−1	10.7
3810	5	6.0	4429	−1	10.7	4754	−3	15.9
3938	5	4.5	4442	−2	15.6	4866	−1	15.4
4125	−5	16.9	4449	10	5.1	5322	−5	17.3
4136	5	9.3	4472	−5	8.1	5813	−5	14.7
4254	5	4.5	4477	−3	14.9	6384	4	5.2

7.5 Hand Gestures

Contributed by Soonil Rughooputh, University of Mauritius

Hand gesture recognition provides a natural and efficient communication link between humans and computers for human computer interaction and robotics [28, 42]. For example, new generations of intelligent robots can be taught how to handle objects in their environments by watching human subjects (if not other robots) manipulating them. Unlike most modes of communication, hand gestures usually

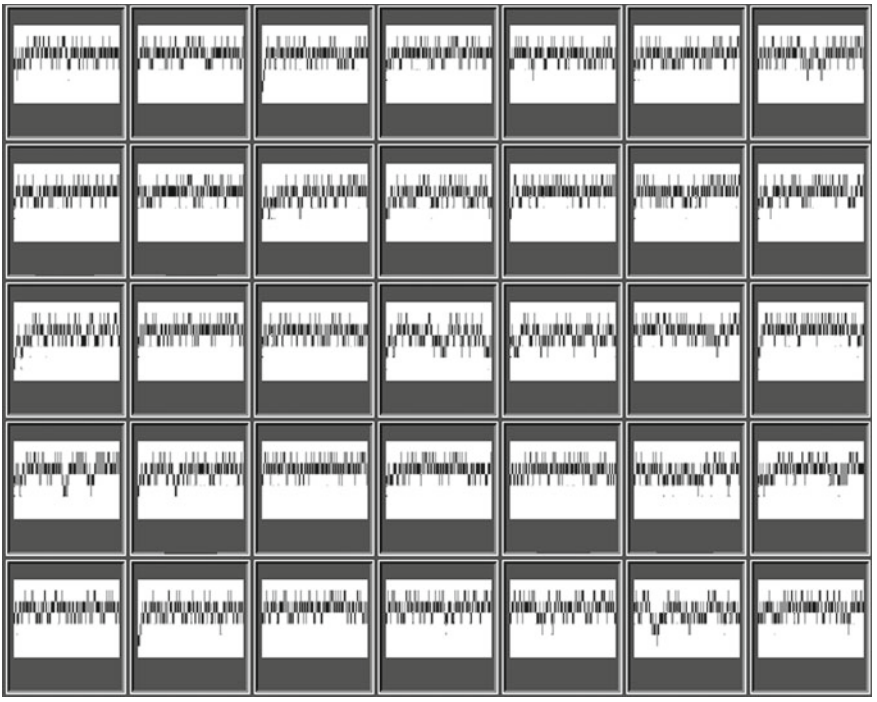


Fig. 7.13 Galaxies identified from their corresponding barcodes

possess multiple concurrent characteristics. Hand gestures can be either static, like a pose, or dynamic (over space and time) and include the hand gestures/hand signs commonly used in natural sign languages like the American Sign Language (ASL) or Australian Sign Language (AUSLAN). Although, there are many methods currently being exploited for recognition purposes, using both the static and dynamic characteristics of hand gestures, they are computationally time demanding, and therefore, not suitable for real-time applications. Recognition methods can be classified into two main groups, those requiring special gloves with sensors and those using computer vision techniques [23, 64, 114]. Recognition methods that fall under the first category can give very reliable information. Unfortunately, the connection cables in the gloves highly limit human movements in addition to being unsuitable for most real-world applications. Consequently, interests in computer vision techniques for hand gesture recognition have grown rapidly during the last few years.

Several researchers have devised hand gesture recognition systems in which marks are attached on fingertips, joints, and wrist [23]. Despite being suitable for real-time processing, it is however inconvenient for users. Another approach uses electromagnetic sensors and stereo-vision to locate the signer in video images [114]. To recognise ASL signs, Darrell [22] adopts a maximum a posteriori probability approach and uses 2D models to detect and tract human movements. Motion trajectories have

also been utilised for signer localisation [30]. However, these approaches require a stationary background with a certain predetermined colour or restrict the signer to wear specialised gloves and markers, which makes them unsuitable for most real-world applications. Researchers have also investigated the use of neural network based systems for the recognition of hand gestures. These systems should enable major advances in the fields of robotics and human computers interaction (HCI). Using artificial neural systems, Littmann [67] demonstrate the visual recognition of human hand pointing gestures from stereo pairs of video camera images and provide a very intuitive kind of man-machine interface to guide robot movements. Based on Johanssons suggestion that human gesture recognition rests solely on motion information, several researchers have carried out investigations on motion profiles and trajectories to recognise human motion [45]. Siskind [99] demonstrated gesture classification based on motion profiles using a mixture of colour based and motion based techniques for tracking. Isard [44] have come forward with the CONDENSATION algorithm as a probabilistic method to track curves in visual scenes. Furthermore, Yang [118] have used time-delay neural network (TDNN), specifically trained with standard error back propagation learning algorithm, to recognise hand gestures from motion patterns.

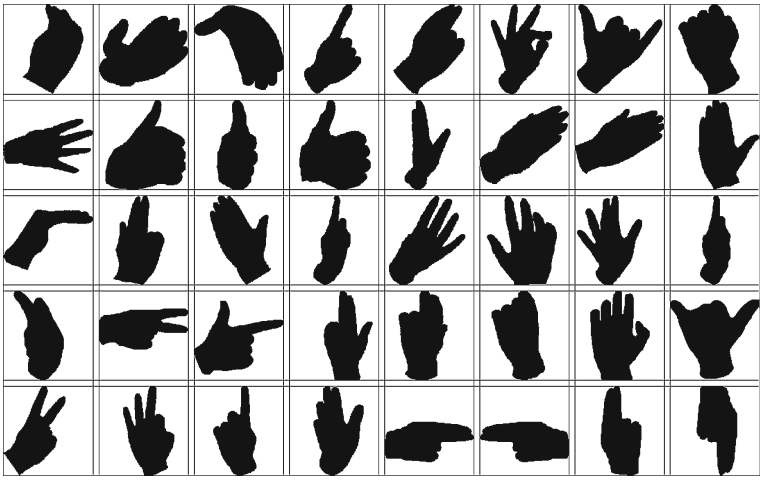
The one-to-one correspondence between each image and its corresponding binary barcode is shown in Fig. 7.14 [88]. Recognition of hand gestures is performed using an N-tuple weightless neural network.

7.6 Road Surface Inspection

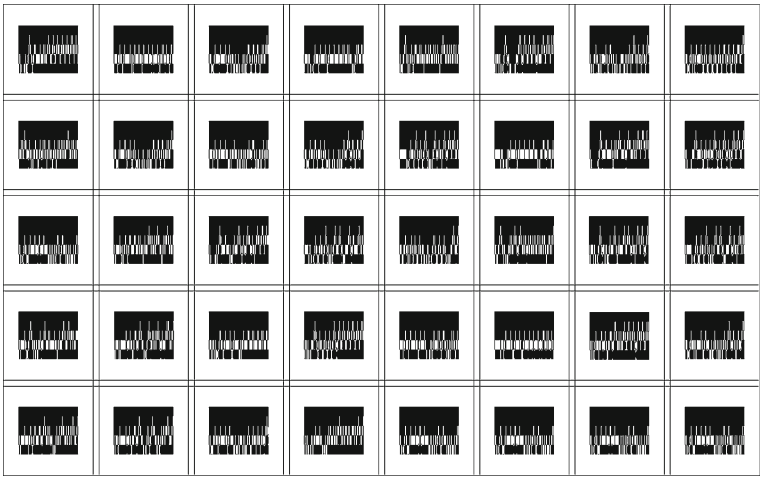
Contributed by Soonil Rughooputh, University of Mauritius

Inspections of road surfaces for the assessment of road condition and for locating defects including cracks in road surfaces are the traditionally carried out manually. As such they are time-consuming, subjective, expensive, and can prove to be hazardous and disruptive to both the road inspectors and the circulating traffic users. What is ideally required would be a fully equipped automated inspecting vehicle capable of high precision location (to the nearest cm) and characterization of road surface defects (say cracks of widths 1 mm or greater) over the width of the road at speeds (up to 80 kmh-1) over a single pass. The automated system could also be enhanced to store the type of cracks present.

Several studies on automated systems for the detection and characterization of road cracks have been reported recently [36, 85, 107]. In this spirit, Transport Research Laboratory Ltd. (UK) as recently proposed an automatic crack monitoring system, HARRIS [85]. In this system video images of the road surface are collected by three line scan cameras mounted on a survey vehicle with the resolution of the digital images being 2 mm of road surface per pixel in the transverse direction and a survey width of 2.9 m. The scanned image (256 KB) is preprocessed to 64 KB (through reduction of grey levels). Reduced images are then stored in hard disk together with the location information. The location referencing subsystem reported



(a) Gestures.



(b) Barcodes.

Fig. 7.14 2D hand gestures used in experiments and their corresponding barcodes

in HARRIS (± 1 m accuracy) requires extra cameras and other hardware. The image processing of HARRIS is carried out in two stages: the first one consists of cleaning and reducing the images (on-line operation aboard the vehicle) and the second stage consists of an off-line operation on the reduced images to characterize the nature of the cracks. A typical one day survey of 300 km of traffic lane would tantamount to 80 GB of data collected. Full details of HARRIS can be found elsewhere [85].

Several refinements to the HARRIS system for a more robust automatic inspection system are obvious. First, Global Positioning Systems (GPS) can be used (instead

of video-based subsystem) to provide a much better accuracy for position location (down to 1 mm with differential GPS). Second, there is no need to store large volumes of scanned images of 'acceptable' road surface conditions. In this respect, the PCNN technique can be used for preprocessing each scanned image to detect defects and a second PCNN to segment this image if any defect(s) is (are) identified [89]. The latter image is then stored as binary image along with the GPS data. A real-time-crack map can be displayed by combining the results of the individual cameras. Detailed characterization of the defects can be performed offline from the recorded binary images. This mode of data collection leads to a more accurate, less costly and faster automated system.

Since the reflective responses of the material road surface can differ from place to place, there is a need to calibrate the software with a sample of a good road surface condition. This can be done in real-time in two modes either once if the inspector is assured that for the whole length of the road has the same reflective responses in which case one sample image will suffice or periodically taking sample images at appropriate intervals. The overall philosophy behind the success of our method relies on the comparison of the input images from the camera to a reference (calibrated sample) image representing the good surface condition from defects. Our method does not compare the image directly since this will be very time-consuming; instead it involves the conversion of the input image into a binary barcode. The barcode generated from an image containing a defect will be different from that of a good road surface condition.

The basis of the crack identification process is the fact the barcode of an image containing a crack (whatever the nature) is different from the barcode representing a good surface condition. Figure 7.15a shows examples of different road conditions—the input images being stored in the PGM format (256 levels). The binary images with defects collected at the end of the surveyed road can then be analysed both in real time and off-line. Figure 7.15b shows the binary images of the segmented images using PCNN. It is clear that the nature of the defects depicted from these images can be easily classified (crack widths, hole sizes, etc.) according to an established priority for remedial actions. Since cracks can be observed as dark areas on the digital images, a second order crack identification algorithm can be used to crudely classify the priority basis. In this case, a proper threshold level needs to be found after carefully omitting redundant parts of the images (for e.g. corner shadowing effects).

When compared with the performance of HARRIS, the technique reported offers a much higher success rate (100 %) for the crack identification process [89]. The high false-positive rates (i.e. low success rate of HARRIS) can be attributed to the poor image qualities arising from the fact that the number of grey levels in the cleaned images have been reduced in the primary processing stage (to 64 KB for storage purposes). Unlike HARRIS, there is no necessity to add specific criteria in our crack identification process, the need to add predetermined criteria to join crack fragments, and the need to store large volumes of scanned images of 'acceptable' road surface conditions. HARRIS also had to inbuilt a special algorithm based on predetermined criteria to join crack fragments. In short, we save a lot in terms of computing time. We note that road markings (such as yellow or white indicators) and artificially created

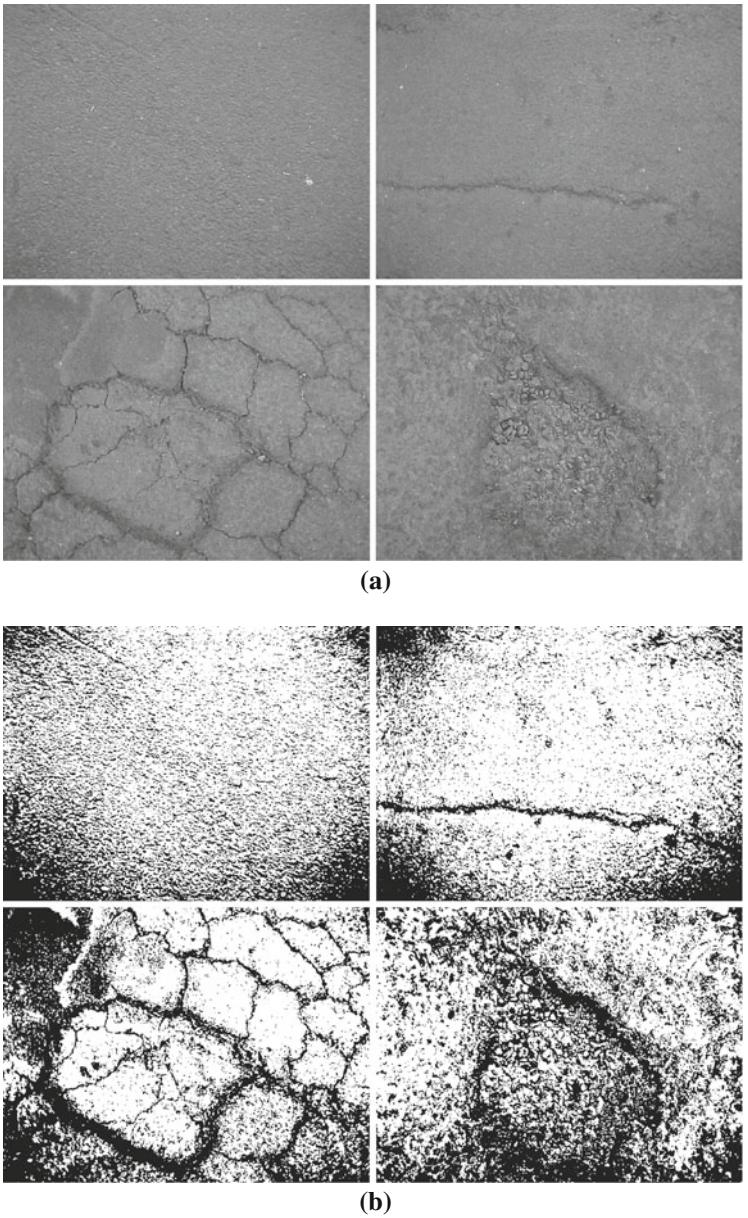


Fig. 7.15 **a** Typical road conditions. **b** Segmented images of typical road conditions shown in (**a**)

structures (such as manholes and construction plates, etc), would be initially treated as defects. In any case, most of such markings occur on either side of the lane so that the camera can be adjusted to reveal say around 80–90 % of the road width. In this configuration, most of the road markings will be ignored. Other markings or

structures can be easily rejected when the binary images are analysed manually. Using the GPS location data, the collected binary segmented images from each camera are laid side by side in multi-lane surveys to create a crack map which indicates not only the location, length and direction of each crack identified, but also identifies cracks or defects extending beyond the boundaries of the individual survey cameras. We note that it is possible to obtain crack maps in real-time in single passes using our technique.

7.7 Numerals

A system of handwritten numeral recognition using the PCNN and the FPF was proposed by Xue em et al. [115] in which the output of the PCNN was fed into the FPF for recognition. One of the features of the FPF is that it can train on only P vectors where $P < D$ and D is the dimensions of the vectors. While the proposed approach works well on images it will not do so well on PCNN signatures since the FPF will be severely limited in the number of training vectors. Thus, a modified version is presented here that shows that handwritten numeral recognition is possible use the PCNN signatures and a neural network for classification.

Each image will be processed by the PCNN and pulse images are generated and converted to an image signature using Eq. (4.8). In this case it is necessary that all numerals be of a similar size and that the all image frames be the same size. Such a database of handwritten numerals is readily available at http://yann.lecun.com/exdb/mnist. The neural network is then trained on these signatures and analysed for ability to learn the problem. This last statement is explained in the following Sect. 7.8.1.

7.7.1 Data Set

The data set is obtained from http://yann.lecun.com/exdb/mnist in which the images and labels of the images are contained in two large binary files. For example, one file contains 60,000 images. Thus, it is first necessary to unpack the data.

The image file *train-images.idx3-ubyte* starts with a very small header of 16 bytes. There are four elements in this header each consuming 4 bytes. These are a identification number, the number of images, the number of rows in a single image, and the number of columns in a single image. Starting with bytes 17 the data is stored sequentially as unsigned 8 bit pixels. The second file *train-labels.idx1-ubyte* contains the classification of each image. The header contains just the identification and number of labels. The data follows is simple a single byte value between 0 and 9 which corresponds to the image in the other file.

The **UnpackImages** function shown in Code 7.1 unpacks the data file and returns a list of images. Line 9 gathers the number of images and Lines 11 and 13 gather the horizontal and vertical size of the images. In this file there are 60,000 images and each is 28×28. The function returns a list `letts` which contains 60,000 matrices.

Reading the classifications is performed by the **UnpackLabels** function shown in Code 7.2. This function is similar to its predecessor and returns a list of integer classifications.

7.7.2 Isolating a Class for Training

The data set is in no particular order and so it is necessary to extract images according to their classification. Code 7.3 shows the **IsolateClass** function which extracts only those images that are associated with a particular class which is denoted by `targ`.

Code 7.1 The **UnpackImages** function.

```
# handwrite.py
import numpy as np

def UnpackImages( fname ):
    fp = file( fname, 'rb' )
    a = fp.read( 4 ) # magic number
    b = fp.read( 4 )
    a,b=np.fromstring(b[2],np.uint8),np.fromstring(b[3],np.uint8)
    N = int(a[0]) *256  + int(b[0])
    b = fp.read( 4 )
    rows = int(np.fromstring( b[3], np.uint8 )[0])
    b = fp.read( 4 )
    cols = int(np.fromstring( b[3], np.uint8 )[0])
    letts = []
    for i in range( N ):
        if i % 1000==0: print i,
        a = fp.read( rows*cols)
        a = np.fromstring( a, np.uint8 )
        letts.append( a.reshape( (rows,cols) ) )
    return letts
```

Code 7.2 The **UnpackLabels** function.

```
# handwrite.py
def UnpackLabels( fname ):
    fp = file( fname, 'rb' )
    a = fp.read( 4 ) # magic number
    # number of images
    b = fp.read( 4 )
    a,b=np.fromstring(b[2],np.uint8),np.fromstring(b[3],np.uint8)
    N = int(a[0]) *256  + int(b[0])
    work = fp.read( N )
    work = np.fromstring( work, np.uint8 )
    names = map( int, work )
    return names
```

Lines 7 and 8 shows the calls to load the data and Line 9 collects all of the images that are associated with the class '0'. These are the images of the handwritten '0'. In this data set there are 5,923 such images.

7.8 Generating Pulse Images

The next step is to generate the PCNN signatures for a set of data. Function **PulseOnNumeral** shown in Code 7.4 receives a list of images and computes the PCNN signature for each image. The PCNN loop starting on Line 6 considers each individual image. For this image a new PCNN is created (Line 7) with a default threshold of 1 (Line 8). The stimulus is normalized so that the maximum value is 1.0 in Line 9 and the PCNN iterations are computed in Line 12. The function returns a matrix in which the i-th row is the signature for the i-th training image.

Code 7.3 The **IsolateClass** function.

```
# handwrite.py
def IsolateClass( letts, names, targ ):
    hits = (np.array(names)==targ).nonzero()[0]
    data = map( lambda x: letts[x], hits )
    return data

>>> letts = UnpackImages( 'train-images.idx3-ubyte')
>>> names = UnpackLabels('/train-labels.idx1-ubyte')
>>> mgs0 = IsolateClass( letts, names, 0 )
>>> len( mgs0 )
5923
```

Code 7.4 The **PulseOnNumeral** function.

```
# handwrite.py
def PulseOnNumeral( datas, NITERS=12 ):
    N = len( datas )
    V,H = datas[0].shape
    gs = np.zeros( (N,NITERS) )
    for i in range( N ):
        net = pcnn.PCNN( (V,H) )
        net.T += 1
        stim = datas[i].astype(float)/datas[i].max()
        g = np.zeros( NITERS )
        for j in range( NITERS ):
            net.Iterate( stim, 2)
            g[j] = net.Y.sum()
        gs[i] = g + 0
    return gs
```

Code 7.5 The **RunAll** function.

```
# handwrite.py
def RunAll( letts, names, L=50 ):
    G = []
    for i in range( 10 ):
        print 'Generating Pulse Images', i
        datas = IsolateClass( letts, names, i )
        G.append( PulseOnNumeral( datas[:L] ))
    return G

>>> G = RunAll( letts, names, 100 )
```

This process is repeated for all 9 classes in **RunAll** shown in Code 7.5. Since there are so many images it is prudent to provide a limit of the number of images that are used which is `L` in Line 2 and to provide a print statement to monitor the computations. The call to the function shown in Line 10 computes the signatures for the first 100 images of each class. The output `G` is a list and each item is a matrix that is 100×12 for the 100 signatures each of length 12.

7.8.1 Analysis of the Signatures

The signatures are now generated and it is important to determine if there is any chance of classification. The first analysis is shown in Fig. 7.16 in which there are 12 groups of box plots. These correspond to the 12 units in the PCNN signature. Each group contains 10 boxes which correspond to the 10 different numeral classifications.

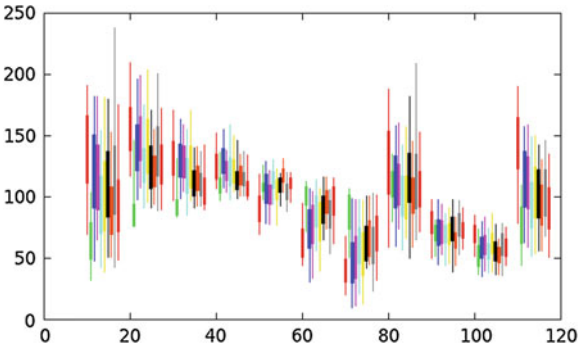


Fig. 7.16 The average and deviation of the different classes for each element in the signatures. Each *colour* represents a set of numeral images. Each *box* represents the average of the signature with a deviation of one standard deviation. The extent of the whiskers represent the max and min values for the signatures

The extent of the box shows the deviation from the average value and the extent of the lines shows the min and max values.

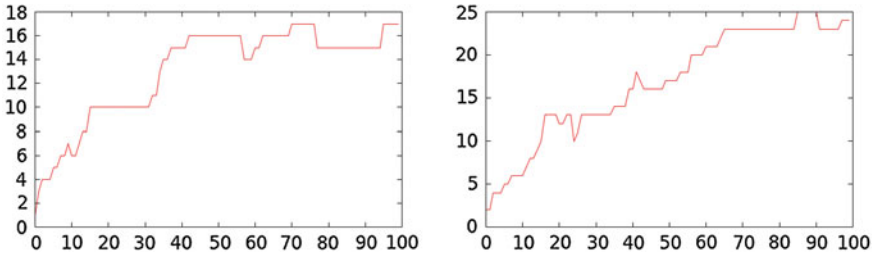
In order for first-classification to be plausible it is necessary that there be a difference between the classes. Thus, as a minimum, it is necessary that for any two classes that there be one element in the signatures that is vastly different. Thus, separation is possible when there are significant height differences between the boxes in a least one signature element.

The plots are shown in order and so for a single group the boxes shown are the classifications of 0–9 in that order. It is seen in several groups that the first two boxes have significant height differences. These correspond to the numerals '0' and '1' which are visually distinct. A much more difficult group would be to separate numerals ('0', '2', '3', '6', '8', and '9'). Even in these case there are differences at some elements. This indicates that first order separation is plausible but not guaranteed.

Thus, in this demonstration the classification of the signatures is provided by a higher-order network named FAAM (fast analog associative memory) which is detailed in Appendix E. The FAAM is a two-layered neural architecture that expands its internal nodes as the complexity of the training data is addressed. Neural networks acting as associative memories train on data that have associated classes. It is possible that the network trains on the data but can not correctly classify non-training data. This is merely a memorisation network. There are a few causes for this as perhaps the training data was insufficient or that the network was not capable of handling the complexity inherent in the data set.

Another possibility is that the network *learns the problem* in which case the network would be able to correctly classify non-training data. The FAAM provides information about this ability during the training session. Namely, when the network considers a particular vector for training it first determines if it is correctly recalled by the current system. If this is so then the network does not adjust its nodes. Thus, if the network begins to learn the problem the it does not adjust the nodes in the network. The FAAM tracks this by counting the number of hidden neurons it requires in order to learn the current training set. Thus, as more data is added a FAAM that learns the problem (instead of memorising the data) produces training curves as shown in Fig. 7.17. The x-axis is proportional to the number of training vectors and the y-axis is the number of hidden nodes that are required. The plot shows that the system is reaching an asymptote which indicates that as more data is considered training is not required to provide proper recalls.

Figure 7.17a shows the case in which a FAAM was trained to recognise the numeral '1' and reject all other numerals. In each training iteration one signature from each numeral was added to the training set and thus there are 100 training cycles. As seen a plateau is obtained near $x = 40$ which indicates that the FAAM is capable of recognising non-training data. Figure 7.17b shows the more difficult case of recognising the numeral '0' which visually has many similarities to '2', '3', '6', '8', and '9'. The plateau is reached at $x = 65$ but it is still reached. This indicates the handwritten numeral recognition using the PCNN signatures an the FAAM is plausible.



(a) Class '1'. **(b)** Class '0'.

Fig. 7.17 FAAM training for two cases. The x-axis represents the number of signatures that have been used in training the FAAM and the y-axis represents the number of decision surfaces the FAAM requires. The asymptotic behaviour indicates that the FAAM is learning the general nature of the problem rather than memorising specific data sets

7.9 Face Location and Identification

An application in face finding was proposed by Yamada et al. [116] in which the fast linking PCNN is slightly modified to assist in separating faces from other components of the image. The proposed method follows the fast linking protocol as outlined in Sect. 4.1.7 except that the computation of **L** is replaced by,

$$L_{ij}[n] = e^{\alpha_L \delta_n} L_{ij}[n-1] + V_L \sum_{k,l} W_{ijkl} Y_{kl}[n-1] - I \frac{1}{N} \sum_{i,j} Y_{ij}[n-1], \quad (7.1)$$

where I is an inhibition factor. The modification adds the third term which suppresses previously pulsed regions. The result is a much faster algorithm.

The protocol presented by Yamada et al. contains three major steps:

1. Extract skin-tone pixels,
2. Apply the PCNN, and
3. Detect oval shapes.

The test image used in this case is shown in Fig. 7.18 obtained from [5]. The first step in the protocol is to isolate the pixels that have a skin tone. The skin pixels may have a wide variety of intensity values even though the hue is relatively constant.

Therefore, selection of candidate skin pixels is performed by band thresholds in the YIQ colour space. The conversion from RGB colour space to YIQ is a linear transformation defined as,

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.595716 & -0.274453 & -0.321263 \\ 0.211456 & -0.522591 & 0.311135 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \quad (7.2)$$

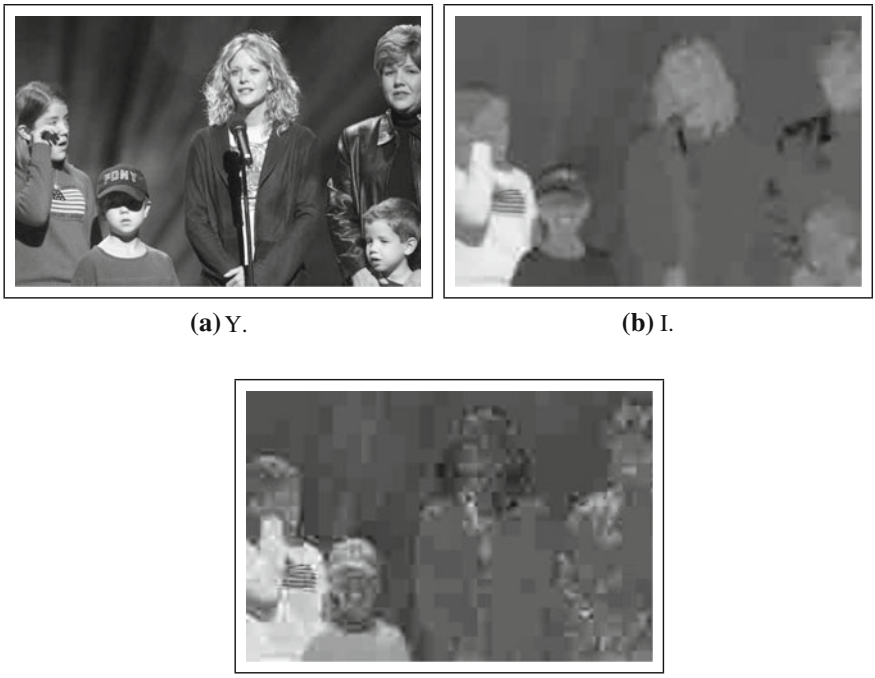
Fig. 7.18 Image from the database in [5]

The transformation of the test image is shown in Fig. 7.19. The Y channel displays intensity information and the I and Q channels display chromatic information. Of particular interest is the I channel which shows rather uniform intensities for the skin of all people in the image. However, items such as hair and a flag also have similar intensities. So, through this channel it is possible to isolate skin pixels but there will be clutter as well.

There are a couple of Python tools that can be used to perform the conversion to YIQ and the isolation of candidate skin pixels. The process is shown in Code 7.6 where Line 3 uses the Image module from PIL to load the colour image. Line 4 uses the **array** function from NumPy to convert the image into a data cube. In this case the cube has dimensions $300 \times 450 \times 3$ in which the three channels are the RGB (red, green, blue) information.. These three channels are separated into individual matrices in Lines 5–7. The *colorsys* module comes with tuhe standard Python installation and it contains routines to transform RGB data into other colour maps such as YIQ. Line 8 calls the **rgb_to_yiq** function to perform this conversion. The last line performs the threshold operation. Manually inspecting the I channel with an image viewer indicates that skin pixels were in the range of 10–40. These are isolate in Line 9 and the result is shown in Fig. 7.20. Here the white pixels indicate the pixels that have the correct I values and are candidate skin pixels. At this point it is necessary that all skin pixels are included.

The second step in Yamada's method is to use the modified PCNN on the input which is itself modified by the mask shown in Fig. 7.20. This process is replicated in Code 7.7 where Line 3 masks the grey scale version of the image with the mask and normalizes the result so that the maximum value is 1. Line 4 is the optional smoothing which reduces effects from random noise. Line 9 calls the **FastLYIterate** function which is shown in Code 7.8. This function is added to the *pcnn.py* module as part of the PCNN class.

Figure 7.21 shows the significant pulse outputs. As seen the faces are outline in pulses $n = 12$ and $n = 13$ which are in the second cycle.



(a) Y. **(b)** I.

(c) Q.

Fig. 7.19 YIQ transformation of the original image

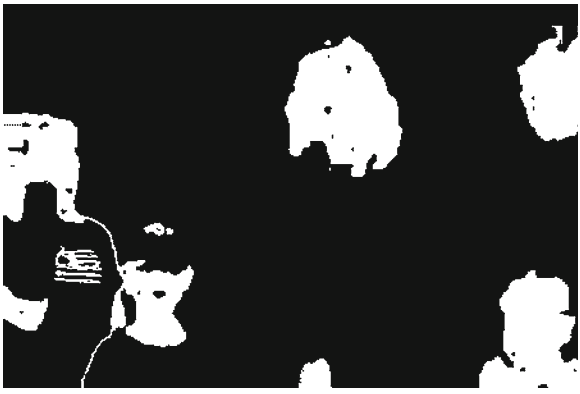
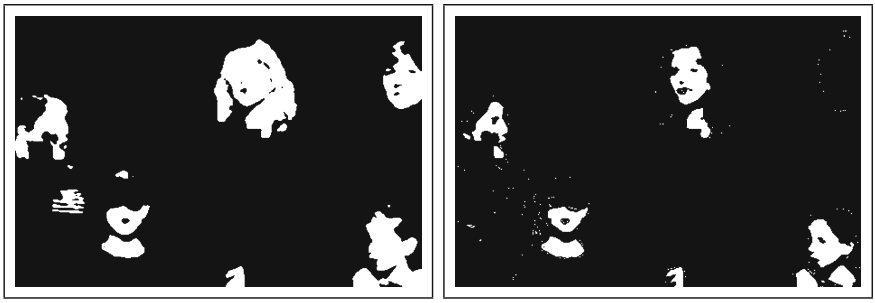
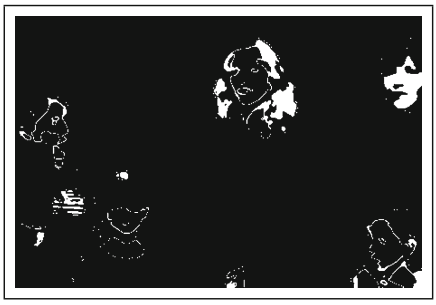


Fig. 7.20 The *white* pixels show the candidate skin pixels

Step three of Yamada's method is to find oval shapes in the output pulses. This, of course, only works if the faces are ovals. As seen in this case, the boy with the hat does not produce and oval shape. There are other methods of finding faces once they have been isolated. A simple method would be to isolate the individual segments from the pulse images and to perform a sum in the horizontal direction. Faces will have a



(a) $n = 1$. **(b)** $n = 12$.



(c) $n = 13$.

Fig. 7.21 The significant pulse images

Code 7.6 Isolating candidate skin pixels.

```
>>> import colorsys, Image
>>> import numpy as np
>>> mg = Image.open( 'benefit.jpg')
>>> a = np.array( mg )
>>> r = a[:,:,0].astype(float)
>>> g = a[:,:,1].astype(float)
>>> b = a[:,:,2].astype(float)
>>> yiq = colorsys.rgb_to_yiq( r,g,b )
>>> mask = (yiq[1]>10 ) * (yiq[1]<40 )
```

tri-modal function since the eyes and mouth regions tend to depress the values in the summation. This process is shown in Code 7.9 which uses the **scipy.ndimage.label** function to isolate the binary shapes. This function creates two outputs of which the second is a count of the number of individual shapes. The first output is `lbls` which is a matrix in which unique integers are assigned to the shapes. The output is shown in Fig. 7.22a. This image is show in the inverted manner. Each segment is identified

Code 7.7 Running the modified PCNN.

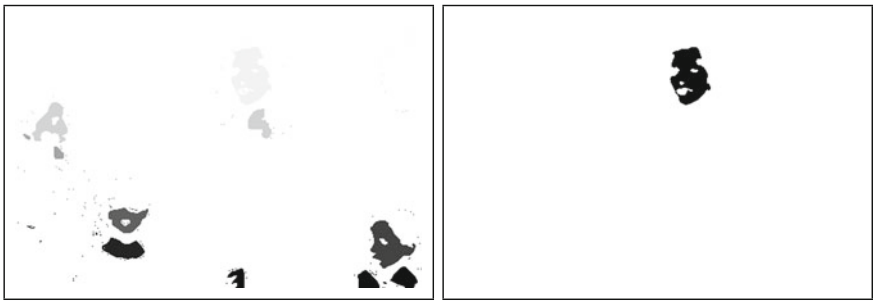
```
>>> from scipy.signal import cspline2d
>>> import pcnn
>>> stim = (mask * yiq[0]).astype(float) / 255.0
>>> stim = cspline2d( stim, 1 )
>>> Y = []
>>> net = pcnn.PCNN( stim.shape )
>>> net.T += 1
>>> for i in range( 15 ):
        net.FastLYIterate( stim, 0.5 )
        Y.append( net.Y + 0 )
```

Code 7.8 The **FastLYIterate** function.

```
# pcnn.py
    def FastLYIterate( self, stim, inhibit ):
        """Fast Linking Yamada's Method"""
        old = np.zeros( self.Y.shape )
        ok = 1
        V,H = stim.shape
        NN = float(V*H)
        self.Y = old + 0
        if self.Y.sum() > 0:
            work = cspline2d(self.Y.astype(float),90)
        else:
            work = np.zeros(self.Y.shape,float)
        self.F = self.f * self.F + stim + 8*work
        while ok:
            print '.',
            if self.Y.sum() > 0:
                work = cspline2d(self.Y.astype(float),90)
            else:
                work = np.zeros(self.Y.shape,float)
            self.L = self.l*self.L+work-inhibit/NN*self.Y.sum()
            U = self.F * (1 + self.beta * self.L )
            old = self.Y + 0
            self.Y = np.logical_or( U > self.T, self.Y )
            if abs(self.Y - old).sum() <100: ok = 0
        self.T = self.t1 * self.T + self.t2 * self.Y + 0.1
```

Code 7.9 Horizontal sums across a candidate shape.

```
>>> from scipy.ndimage import label
>>> lbls, cnt = label( Y[12] )
>>> hsum = (lbls==6).sum(1)
```



(a) (b)

Fig. 7.22 Results from the **label** function

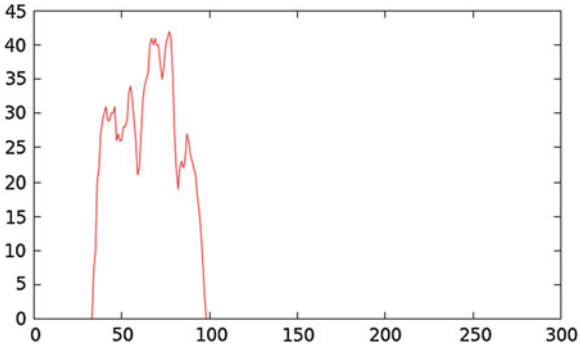


Fig. 7.23 Horizontal summation for a candidate shape

by pixels of a single value. In this case the actual face belongs to the region defined by `lbls==6` and is shown in an inverted manner in Fig. 7.22b.

Line 3 sums horizontally across this region and the result is shown in Fig. 7.23. There are two major dips in the plot which correspond to the eyes and the mouth regions. Non face regions will have a very different result from this summation. This region can therefore be considered as a face.

7.10 Summary

The PCNN and ICM models provide two avenues for target recognition. The first to directly employ the pulse images and the second is to use the image signatures. This chapter shows methods by which the pulse images can be used for target recognition. The systems were created by several groups and show a variety of applications.

Chapter 8
Texture Recognition

In many applications the information that is important is the textures within an image. There are many such applications and in this chapter the use of texture analysis using medical images will be considered. Regions in an image pulse in unison when their stimuli are the same. If the stimuli are varied (there exists a texture) then the synchronised behaviour of the pulse segments will disintegrate, and this desynchronisation increases and the system iterates. This desynchronisation is dependent upon the texture of the input and thus texture can be measured and used for segment classification.

The authors would like to acknowledge the significant contribution of Guisong Wang for the material in this chapter.

8.1 Pulse Spectra

Consider again the images shown in Fig. 5.5. The nucleus of the red blood cell contains a texture. In iteration $n = 1$ the nucleus pulses as a segment completing the first cycle. The second cycle occurs in iterations $n = 16$–19. The neurons of the nucleus have desynchronised and the pulses are separated according to the texture of the original segment. Thus, measurement of texture is performed over several iterations rather than in a single iteration.

Texture is an interesting metric in that it describes a property that spans several pixels but in that region those pixels differ. It is a segment described by dissimilarity. The size of this region, however, is defined by the user and cannot be set to a uniform distance. There have been several methods by which texture has been measured. Many of these rely on statistical measures but the ICM is different. The higher order system can also extract relational information. Figure 8.1 displays images with two distinct but typical textures [98]. Even though the pixel values vary on a local scale the texture is constant on a global scale.

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_8,

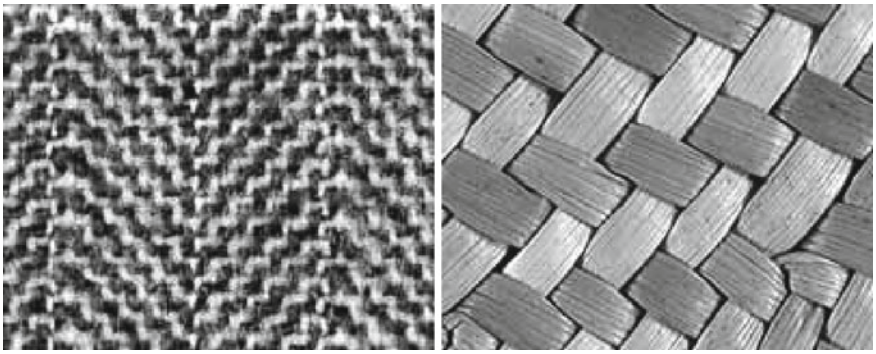


Fig. 8.1 Example textures

One of the simplest methods of measuring texture is to simply measure the statistics such as the mean, the variance (and the coefficient of variance), skewness and kurtosis. The mean for a vector of data is defined as,

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i, \tag{8.1}$$

where V and H are the vertical and horizontal dimensions of the image. The variance and the coefficient of variance are defined as,

$$\sigma = \frac{N\sum_{i=1}^{N} x_i^2 - \mu^2}{N(N-1)}, \tag{8.2}$$

and,

$$cv = \sigma/\mu. \tag{8.3}$$

The skewness and kurtosis are higher order measures and are defined as,

$$r = \frac{N}{(N-1)(N-2}\sum_{i=1}^{N}\left(\frac{x_i - \mu}{\sigma}\right)^3, \tag{8.4}$$

and,

$$k = \frac{N(N+1)}{(N-1)(N-2)(N-3)}\sum_{i=1}^{N}\left(\frac{x_i - \mu}{\sigma}\right)^4 - \frac{3(N-1)^2}{(N-2)(N-3)}. \tag{8.5}$$

Table 8.1 Statistics of textures

	Texture 1	Texture 2
Mean	0.492	0.596
Variance	0.299	0.385
Coefficient of var.	0.607	0.647
Skewness	0.032	−0.058
Kurtosis	0.783	0.142

For simple images like Fig. 8.1 it is possible to measure and distinguish the textures according to these measures. The values for the two sample images are shown in Table 8.1.

However, real problems generally do not fill the image frame with a single texture. Quite often it is desired to segment the image according to the texture implying that the texture boundaries are not known but rather need to be determined. The image in Fig. 8.2 displays a secretion cell with a variety of textures. One typical application would be to segment this image according to the inherent textures.

To employ the ICM to extract textures pulse activity of each pixel over several iterations is considered. Since texture is defined by a region of pixels rather than by a single pixel the pulse images are smoothed before the texture information is extracted. The information taken from a location (i, j) defines the pulse spectrum for that location and is defined by,

$$p_{i,j} = M\{\mathbf{Y}\}_{i,j}[n], \tag{8.6}$$

were the function $M\{\cdot\}$ is a smoothing operator over each pulse image. The goal is that all pulse spectra within a certain texture range will be similar. Figure 8.3a displays the original image and the rest of Fig. 8.3 displays selected pulse images.

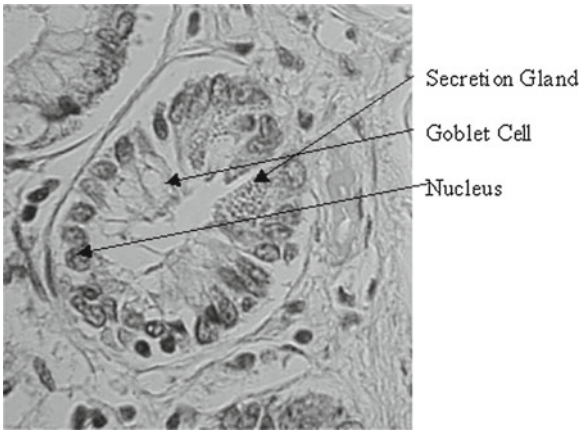
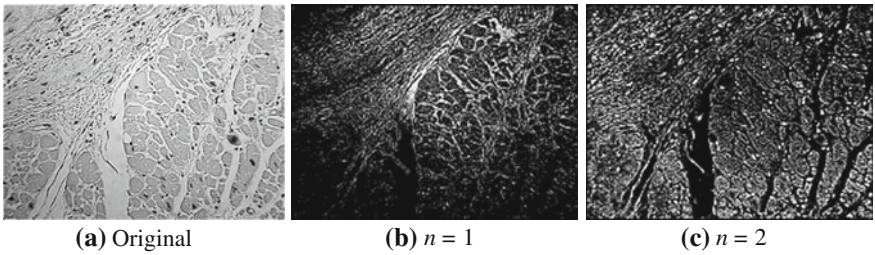
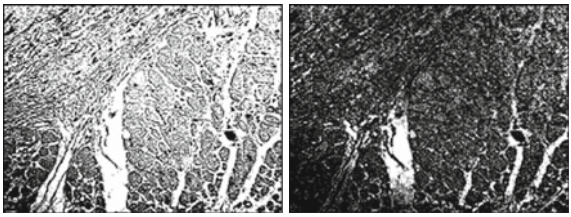


Fig. 8.2 Annotated image of a secretion cell



(a) Original **(b)** $n = 1$ **(c)** $n = 2$



(d) $n = 8$ **(e)** $n = 9$

Fig. 8.3 The original input image and selected pulse images

Common textures pulse in similar iterations. This displays only four of many pulse images. In this case the number of iterations was selected to be twenty.

For the case of standard textures (similar to Fig. 8.1) the method of using pulse spectra was compared to other methods. These methods are listed with their citations but their methods will not be reviewed here:

- Autocorrelation (ACF) [83, 109]
- Co-occurrence matrices (CM) [35]
- Edge frequency (EF) [83, 109]
- Laws masks (LM) [63]
- Run Length (RL) [83, 109]
- Binary stack method (BSM) [17, 18]
- Texture Operators (TO) [110]
- Texture Spectrum (TS) [37]

In [97] the performance of all of the methods in the above list were compared on a standardized database of textures. The tests consisted of a training on all but one of the images and then that image was used as a test. This test was repeated several times using each of the images as the one not used in training. Recall used the K-nearest neighbours algorithm and the results are shown in Table 8.2 for different values of K. At the top of this chart the texture recognition method using the ICM was added and it can be seen that it rivals the best performing algorithms.

Table 8.2 Texture analysis methods

Method	$K = 1$ (%)	$K = 3$ (%)	$K = 5$ (%)	$K = 7$ (%)	$K = 9$ (%)
ICM	94.8	94.2	93.9	92.1	91
ACF	79.3	78.2	77.4	77.5	78.8
CM	83.5	84.1	83.8	82.9	81.3
EF	69	69	69.3	69.7	71.3
LM	63.3	67.8	69.9	70.9	69.8
RL	45.3	46.1	46.5	51.1	51.9
BSM	92.9	93.1	93	91.9	91.2
TO	94.6	93.6	94.1	93.6	94
TS	68.3	67.3	67.9	68.5	68.1

8.2 Statistical Separation of the Spectra

The real task at hand is to measure the textures of a complicated image as in Fig. 8.2. A requirement for the accomplishment of this task is that the spectra discriminate between the different textures. This means that the spectra in one texture region must be similar and compared to another region they must be dissimilar. To demonstrate this three regions in Fig. 8.2 were selected for testing. This image is 700×700 pixels and each region selected was only 10×10 at the locations marked in the figure. The average and standard deviation of the spectra for each region are shown in Fig. 8.4.

The desire is to have each average signature differ significantly from the others and to also have small standard deviations. Basically, the error bars should not overlap. Clearly, this is the case and therefore discrimination of texture using the ICM is possible.

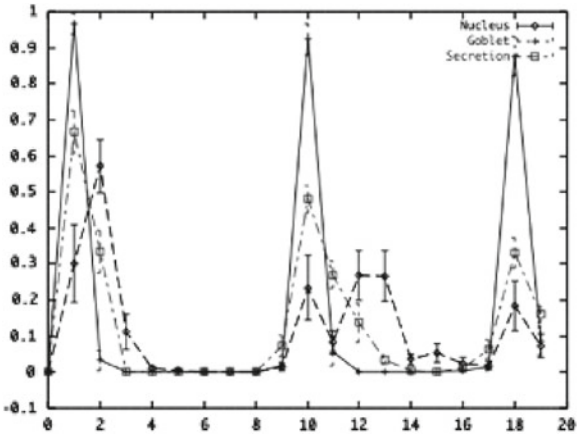


Fig. 8.4 Plots of the average and standard deviation of the three selected regions of Fig. 8.2

8.3 Recognition Using Statistical Methods

A simple method of classifying regions in an image by the texture is to simply compare a pulse spectrum to all of the average spectra in a library. The library consists of average spectra from specified training regions (as in Fig. 8.4). This is similar to the procedures practised in multi-spectral image recognition.

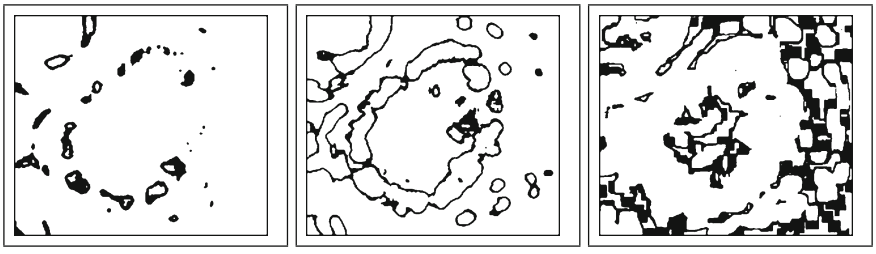
For each pixel in an image there is a pulse spectrum and this can be compared to those in a library. The pixel is then classified according to which member of the library is most similar to the pixel's spectrum. A pixel's spectrum can be classified as unknown if it is not similar to any of the members of the library. For this example, the elements of the spectrum needed to be within one standard deviation of the library spectrum in ordered to be considered close. This measure exclude spectrum members that were close to 0. More formally, the spectrum of the pixel in the image is defined as d_i where $i = 1, 2, \ldots, 20$ (the number of iterations in the ICM). The library consists of a set average spectra, m_i^k, where k is the index over the number of vectors in the library. For each member of the library there is also the standard deviation of the elements, σ_i^k. The pulse spectrum is considered close if for all $d_i > \epsilon$,

$$|d_i - m_i^k| < s_i^k, \tag{8.7}$$

where ϵ is a very small positive constant.

Using this measure the pixels in the image of Fig. 8.2 that were classified as belonging to the nucleus class are shown (black) in Fig. 8.5a. In Fig. 8.5b the pixels classified as secretion are shown and in Fig. 8.5c the pixels classified as goblet are shown.

In this example many of the pixels were classified correctly. However, there were a lot of false positives. Part of the problem is that the texture of some of these misclassified regions is very similar to that of the target. For example there are many nuclei outside of the large cell that have similar texture to the nuclei inside of the cell. Likewise, the goblet cells have similar texture to many regions outside of the cell. These strong similarities make it a difficult problem to solve.



(a) Nucleus. **(b)** Secretion. **(c)** Goblet.

Fig. 8.5 The classification by texture of the pixels as nucleus, secretion, and goblet

Another cause of these false positives is that the texture of regions of similar class are somewhat different. The texture of the individual nuclei inside of the large cell are different.

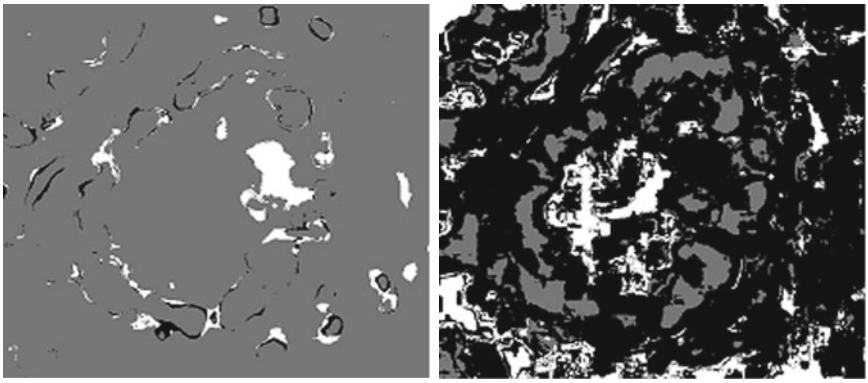
Unfortunately, it is quite common to have a problem in which the texture of target regions different more than the texture between non-target and target regions. If this wasn't the case then this would be an easy problem to solve. In this situation the classification system is insufficient. In other words, the use of statistical comparison between the spectra is incapable of discriminating between target spectra and non-target spectra. A much more powerful discrimination algorithm is required.

8.4 Recognition of the Pulse Spectra via an Associative Memory

The inability of the previous system to completely recognise a texture may be caused by either of two problems. The first may be that the texture extraction engine is inadequate. The second is that the process that converts extracted texture to a decision may be inadequate. In the previous case the decision was reached by simply comparing statistics of the texture. However, in a case in which one class may contain more than one texture this decision process will be inadequate. Thus, we attack the problem with a stronger decision making engine.

There are several types of associative memories that are available to be used and certainly a system that contends optimality would consider several of these memories. In the case here, the goal is to demonstrate that the ICM can extract sufficient information from an image, thus only an adequate associative memory need be considered. If the combination of the ICM and the chosen associative memory sufficiently classify the image then the contention is that the ICM sufficiently extracted the texture information from the image. The associative memory used here is the FAAM which is detailed in Appendix E.

The pixels used in training in the statistical example were also used here. Figure 8.6a displays that classification of pixels in the secretion class. All of the white pixels are declared to be in the class, all of the grey pixels are declared to be out of the class, and all of the black pixels are undefined. In this case the input vector to the associative memory produced a set of decisions that were not similar enough to any of the training vectors. Figure 8.6b contains the classification of the pixels for the goblet class. In this case many of the pixels are classified as not known, however, the goblet pixels are correctly classified.



(a) Secretion. **(b)** Goblet.

Fig. 8.6 Pixels shown in white are classified as the target, pixels shown in *grey* are classified as non-target, and pixels shown in *black* are not classified. **a** Secretion. **b** Goblet

8.5 Biological Application

Idiopathic pulmonary fibrosis (IPF) is a lethal disease of which their is no cure or even a consensus on treatments. Visual features of the disease include regions called *ground glass* and *honeycomb* which are regions in the lung that are no longer functioning. Often these regions begin at the bottom of the lung and along the perimeter and expand upwards. The honeycomb region is most significant in the upper right portion of the lung.

One goal of processing the images is to extract a measure of the volume of lung that is classified as fibrotic. For a single patient several images are taken of the lung at regular intervals and so the measure of volume is reduced to the summation of the measure of area for each image. The fibrotic region does not have a specific location, shape, or regular texture and so automated classification requires a robust system.

In Acharya et al. [6] the ICM was used to create a unique pixel signature in the honeycomb region. Figure 8.8 shows the first cycle of pulse images from the ICM. In this cycle the health and fibrotic regions of the lung do not separate very well.

The second cycle of pulses is shown in Fig. 8.9 which shows, for the most part, the same pixels pulsing but now the desynchronisation is conducive to the inherent texture. This desynchronisation is the key to the ability to classify the fibrotic regions differently from the healthy regions.

The pulse images create a data cube specified by $Y_{ij}[n]$ as from Eq. (4.18). This cube is smoothed slightly to reduce noise and make the pulse regions more cohesive. Thus, the pulse images now assume floating point values. This smoothing is small in extent so as to not significantly change the response of the ICM. The separation

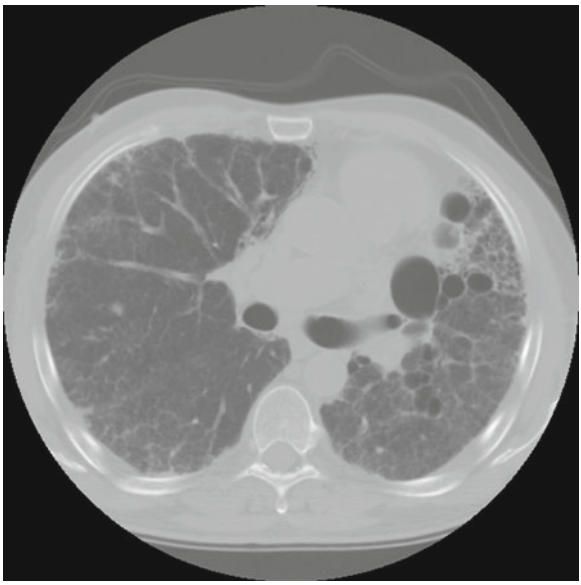
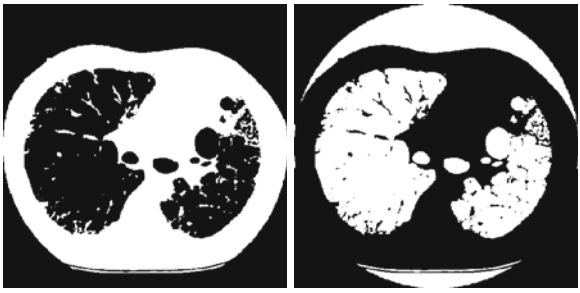


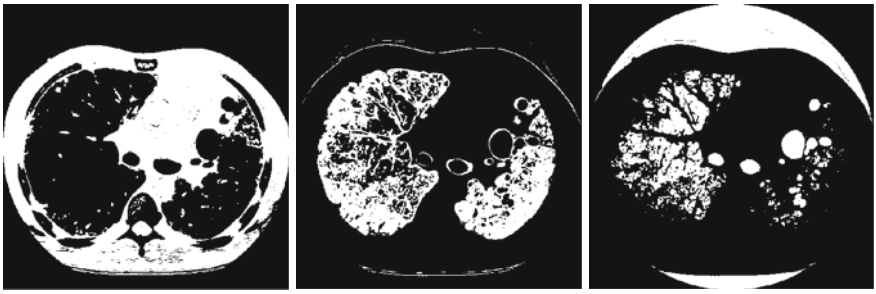
Fig. 8.7 A lung scan of a patient with IPF with the honeycomb region visible in the *upper right* portion of this image



(a) $n = 1$ **(b)** $n = 2$

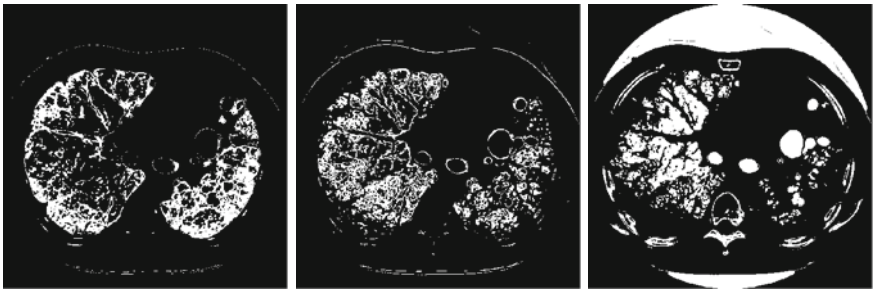
Fig. 8.8 The first cycle of ICM pulse images

of the pulse activity of the two different classifications is verified in Fig. 8.11. The x-axis represents the pulse iteration and the y-axis represents the distribution of pulse behaviour in the fibrotic (red) and healthy (green) regions. Each error bar represents a distribution of pulse values over a classified region. As seen the red bars have a shorter frequency that do the green bars. This indicates that as a whole the fibrotic regions start their second cycle earlier than do the healthy regions. This frequency trend continues into the third cycle as well. This indicates that the pulses are quite capable of separating healthy and fibrotic regions.



(a) $n = 10$ **(b)** $n = 12$ **(c)** $n = 13$

Fig. 8.9 The second cycle of ICM pulse images



(a) $n = 21$ **(b)** $n = 22$ **(c)** $n = 23$

Fig. 8.10 The third cycle of ICM pulse images

The next step was to train an associative memory such as the FAAM (see Appendix E) which is a supervised classification system that grows in complexity as training data is presented to the system. In this case the training data was selected pixels from small regions that are classified by a physician. These training vectors are $Y_{i,j}[n]$, $\forall n$ and selected i and j. This is shown in Fig. 8.12.

A small set of vectors (30 in total) were used to train the FAAM and all pixels inside of the lung were considered as queries. The FAAM produces three outputs which are 1 for the recognition of a class, 0 for the rejection of the class, and -1 for undecided. Thus, output images from the FAAM have three intensities with white being classified as the target, grey being a rejection of the target, and black being undecided. Figure 8.13 show an input from which training vectors were extracted and the classification of the entire image from two FAAMs. The first was trained to recognise fibrotic regions and the second was trained to recognise healthy regions.

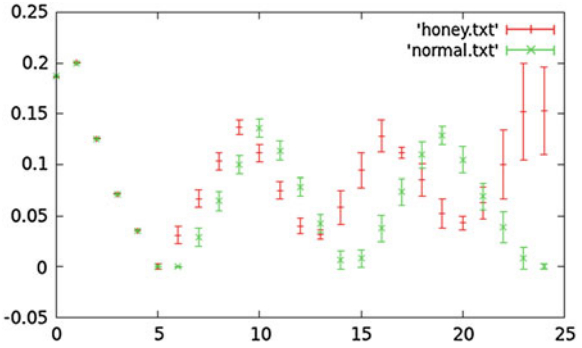


Fig. 8.11 Distributions of pulsing activity for two previously classified regions. The x-axis represents ICM iterations and the y-axis represents strength of pulse activity. The *red bands* coincide with the target regions and the *green bands* coincide with the non-target regions

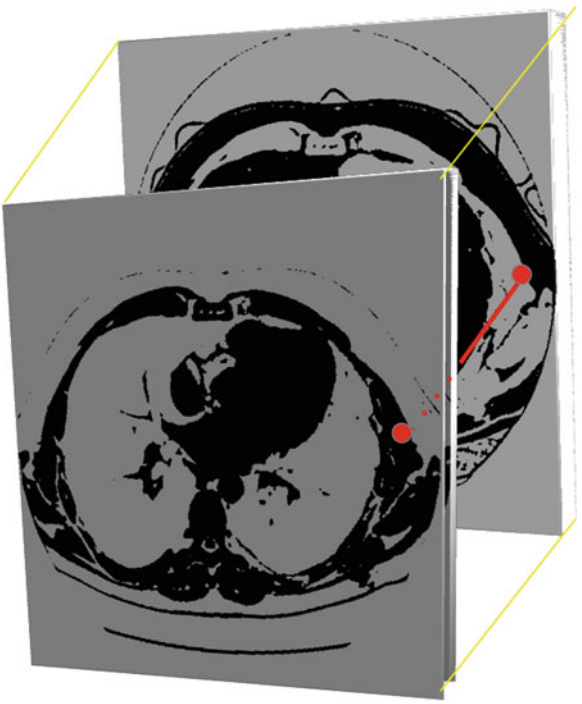


Fig. 8.12 Extraction of a training vector at a specified i, j location

(a) Input. **(b)** Fibrotic recall. **(c)** Healthy recall.

Fig. 8.13 The query image and the output of a FAAM trained to recognise fibrotic regions and a FAAM trained to recognise healthy regions

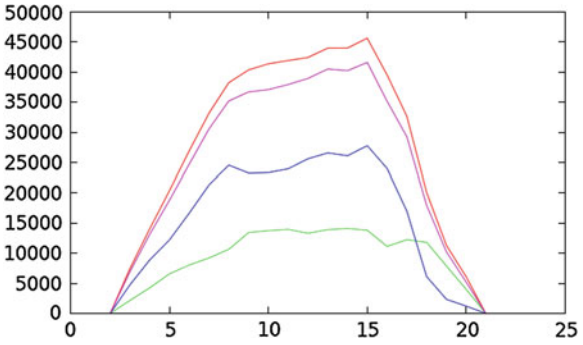


Fig. 8.14 Area measures for each scan for healthy and fibrotic regions. The text explains the axes

Now, that the regions are classified the final step is to measure the area classified by each FAAM. The patient's file contained several scans and each of these were used as queries to the FAAMs but not used in training. The volume of healthy and fibrotic regions were calculated for each scan and the results are shown in Fig. 8.14. The x-axis represents each scan with the top of the lung corresponding to low values of x. The y-axis is the area of each classification.

The top plot represents the total area of the lung from an independent measure. The lowest plot (green) represents the area classified as fibrotic in each scan, and the next lowest plot (blue) represents the area classified as healthy. The remaining plot (purple) represents the sum of the fibrotic and healthy regions. It difference between the top two plots is due to the pixels that were not classified (black in Fig. 8.13).

Clearly, the volume of fibrotic matter becomes dominant in the lower portion of the lung which corresponds to the fact that the disease tends to start at the bottom of the lung. This process now allows physicians to have an automated system of measuring fibrotic regions in patients who unfortunately have contracted IPF.

8.6 Texture Study

A recent study by Zhan et al. [122] consider the performance of the PCNN, the ICM, and their proposed SCM in a texture recognition test. The SCM is another simplified version of the PCNN which removes the Feeding equation and reduces the Linking equation to,

$$L_{ij}[n] = V_L \sum_{kl} W_{ijkl} Y_{kl}[n-1]. \tag{8.8}$$

The internal energy is,

$$U_{ij}[n] = S_{ij}\left(1 + \beta L_{ij}[n]\right). \tag{8.9}$$

The test consisted of several experiments based upon the time signatures produced by the networks. Recognition was performed using the original signals, entropy of the signals, standard deviation of the signatures, and frequency representation of the signatures (such as DCT or FFT). Performance of the three systems were compared and the ICM and SCM had similar recall rates for rotation and scale variations with both being superior to the original PCNN. Recall rates were also compared to systems using Gabor filters with markedly better results.

This study used the Brodatz image set which is a standard texture image set and readily available from many web sources. The Brodatz set has over 100 images which are each quite large. Thus, many studies chop up each image into sub-images and use some of these for training and others for testing. It is a very easy process to compute the signatures via the ICM for sub-images and the to use a method like PCA (principal component analysis) to determine if the signatures can uniquely describe a texture.

To replicate this study a few functions to convert each image into a set of ICM signatures are required. The first is **FileNames** which is shown in Code 8.1. This function receives a directory name that contains the Brodatz images. It loads all of the file names from that directory and then keeps only those which end in `'.gif'` or `'.GIF'`. Some operating systems will place other files in a directory if the user views the images in a file manager, thus it is necessary to prune out non-image files even if no other file was added to the directory by the user.

The second function in Code 8.1 is **LoadImage** which receives an image file name and returns a matrix which represents the grey scale values of the image. These values are scaled between 0 and 1. Code 8.2 shows the **Cutup** function which creates several small 128×128 non-overlapping matrices from the output of **LoadImage**. Line 8 is necessary when the images size is not exactly multiples of 128. If a submatrix is small then it may cause an error in the ICM during the smoothing process. The output is a list of matrices that are all 128×128 and from a single input image.

The **ManySignatures** function (Code 8.3) computes the ICM signature for each of the matrices in `cuts`. This function is hard coded for 15 iterations which gets into the third cycle of pulse images and the desynchronisation is significant. The function returns a list of signatures and each is a 15 element vector.

Code 8.1 The **FileNames** and **LoadImage** functions.

```
# texture.py
import os
import numpy as np
import Image

def FileNames( indir ):
    temp = os.listdir( indir )
    names = []
    for i in temp:
        if '.gif' in i or '.GIF' in i:
            names.append( indir + '/' + i )
    return names

def LoadImage( mgname ):
    mg = Image.open( mgname )
    data = np.array( mg.convert('L') )/255.0
    return data
```

Code 8.2 The **Cutup** function.

```
# texture.py
def Cutup( data, SZ=128):
    V,H = data.shape
    cuts = []
    for i in range( 0, V, SZ ):
        for j in range( 0, H, SZ ):
            vv,hh = data[i:i+SZ,j:j+SZ].shape
            if vv == 128 and hh == 128:
                cuts.append( data[i:i+SZ,j:j+SZ] + 0 )
    return cuts
```

Code 8.3 The **ManySignatures** function.

```
# texture.py
import icm
def ManySignatures( cuts ):
    N = len( cuts )
    sigs = []
    for i in range( N ):
        net = icm.ICM( cuts[i].shape )
        G = np.zeros( 15 )
        print 'ICM: ', i
        for j in range( 15 ):
            net.IterateLS( cuts[i] )
            G[j] = net.Y.sum()
        sigs.append( G + 0 )
    return sigs
```

The final function is **Driver** (Code 8.4) which runs all of the processes. Lines 4 through 10 gather the names of the files, loads the images, cuts each up into sub-matrices, and computes the signatures of all of the sub-matrices. This portion of the script can take more than an hour to process depending on the machine speed and memory. By Line 11 the `bigsigs` is a list and each item in this list is itself a list. Each inner list contains the vector signatures for a single image. Lines 13 through 18 convert the list of lists of vectors into a single large matrix that is $P \times 15$ where P is the total number of signatures.

Code 8.4 The **Driver** function.

```
# texture.py
import pca
def Driver( indir ):
    names = FileNames( indir )
    bigsigs = []
    for i in range( len( names )):
        print "Considering", names[i]
        data = LoadImage( names[i] )
        cuts = Cutup( data )
        bigsigs.append( ManySignatures( cuts ))
    lgs = np.array( map( len, bigsigs ) )
    L = lgs.sum()
    data = np.zeros( (L,15) )
    k = 0
    for i in range( len( bigsigs )):
        for j in range( len( bigsigs[i]) ):
            data[k] = bigsigs[i][j] + 0
            k += 1
    cffs, evecs = pca.PCA( data, 8 )
    return cffs, evecs, bigsigs
```

The final step of the process is to determine if these signatures are capable of distinguishing the different textures with the caveat that similar vectors should indicate similar textures. One approach is to use PCA which is detailed in Appendix F. PCA has the ability to create a new data space which is a rotation of the old space by minimizing the covariance between data elements. Data which has first order distinctions can often be shown to separate in PCA space. However, data that has purely higher-order relationships are not separated by PCA.

In this case, the data matrix is sent to the **PCA** function from the *pca.py* module (Appendix F). This function also receives an integer which is the number of dimensions to return from the PCA computation. The vectors in the original data have 15 dimensions and the call to the **PCA** function requests that only 8 dimensions be returned. This is more than will be used but since the PCA computation is sequential the computations of the first dimensions are not affected by the computations of the latter dimensions. The **Driver** returns the variable `cffs` which is a matrix whose

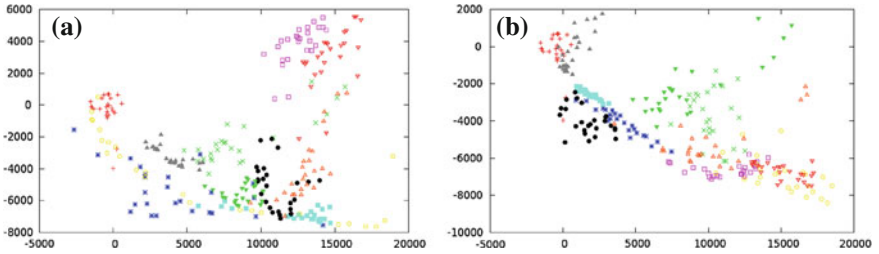


Fig. 8.15 Figures **a** and **b** show clear distinctions from two randomly selected sets of 10 sub-images extracted from the PCA space contrived from all 111 images.

rows represent the data in the new space. The variable `evecs` are the eigenvectors that convert data from the old space to the new space. These are not used in this example but are required if other data not used in creating the PCA space are to be mapped into the new space. The function also returns **bigsigs** which are the ICM signatures.

While there is a tremendous amount of analysis that is possible on the data in the PCA space this section will end with merely displaying the space in a colour coded fashion in which each colour marker represents data from a specific image. Furthermore, displaying the data for all 111 images is cumbersome and so the presentation of the results shows only part of the data. The PCA was computed for all 25 sub-images from all 111 images. The displays in Fig. 8.15 show two sets of 10 randomly selected images as they lie on the space computed for all 111 images. As seen the data does indicate a separation of the images which indicates that the ICM signatures are capable of describing texture.

8.7 Summary

The PCNN and ICM models create pulsing activity that begins as synchronised pulses but as the iterations increase the pulsing activity desynchronises in a manner that is quite sensitive to the texture in the image. Due to the autowave communications this de-synchronisation is both sensitive to variations within the immediate neighbourhood of a pixel as well as pixel activity beyond the distance of direct neural connections. This dependency on texture thus allows for the pulse images to be used in texture discrimination. Several approaches have been proposed and a few are outlined in this chapter.

Chapter 9
Colour and Multiple Channels

Previous chapters considered the PCNN and ICM operating on grey scale images. This chapter considers data that has multiple channels such as colour images and multi-spectral images. In these cases the neural model is expanded in order to handle the new dimension in the input space. The multi-spectral PCNN (ϵPCNN) is a set of parallel PCNNs each operating on a separate channel of the input with both inter- and intra-channel linking. This has a very interesting visual effect as autowaves in one channel cause the creation of autowaves in the other channels. The first autowave leads a progression of autowaves, but they all maintain the shape of the object.

9.1 The Model

Figure 9.1 depicts the new design which uses several PCNN's in parallel and allowing communications between them [53, 58].

The feeding component is now described by,

$$F(\mathbf{x}, n) = e^{\alpha_F \delta_n} F(\mathbf{x}, n-1) + S(\mathbf{x}) + V_F \mathbf{M} \otimes Y(\mathbf{x}, n-1), \tag{9.1}$$

where $\mathbf{x}$ represents the locations in the new space which could be three dimensions or higher. The tensor $\mathbf{M}$ represents the local connections and while it is commonly symmetric for intra-channel communications the symmetry does not necessarily extend into the new dimensions.

Likewise the rest of the system is described by,

$$L(\mathbf{x}, n) = e^{\alpha_L \delta_n} L(\mathbf{x}, n-1) + V_L \mathbf{W} \otimes Y(\mathbf{x}, n-1), \tag{9.2}$$

$$U(\mathbf{x}, n) = F(\mathbf{x}, n)\left(1 + \beta L(\mathbf{x}, n)\right), \tag{9.3}$$

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_9, © Springer-Verlag Berlin Heidelberg 2013

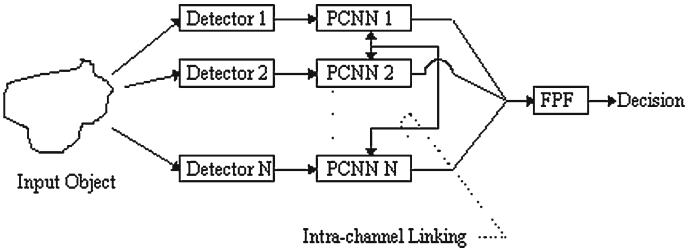


Fig. 9.1 Logic schematic of the multi-channel PCNN

$$Y(\mathbf{n}) = \begin{cases} 1 & \texttt{if } U(\mathbf{x}, n) > \Theta(\mathbf{x}, n-1) \\ 0 & \texttt{Otherwise} \end{cases}, \tag{9.4}$$

and,

$$\Theta(\mathbf{x}, n) = e^{\alpha_\Theta \delta_n} \Theta(\mathbf{x}, n-1) + V_\Theta Y(\mathbf{x}, n). \tag{9.5}$$

9.1.1 Colour Example

This is shown by the example of the original image in Fig. 9.2a and the pulse images shown in Figs. 9.2b–p. The original image is a $256 \times 256 \times 3$, three channel (colour) image of a boy eating ice cream. The other images are the colour-coded pulse outputs of the three channels. Segmentation and edge extraction is quite visible.

An example of the cross channel linking is evident in the boy's hair. The hair is brown and changes in intensity. At $n = 1$ through $n = 3$ and $n = 8$ through $n = 14$ the autowaves are seen travelling down the boy's hair. The red autowave leads the others since the hair has more red than green or blue. The other autowaves follow, but all waves follow the texture and shape of the boy's hair.

The intra-channel autowaves present an interesting approach to image fusion. The image processing steps and the fusion process are highly intertwined. Many traditional image fusion systems fuse the information before or after the image processing steps, whereas this system fuses during the image processing step. Furthermore, this fusion is not statistical. It is syntactical in that the autowaves, which contain descriptions of the image objects, are the portions of the image that cross channels. This method is significantly different from tradition in that it provides higher-order syntactical fusion.

There are several reasons to fuse the inputs of images of the same scene. One example is to fuse an infra red image with a visible image to see relations between items only seen in one of the inputs. The same detector image may also be filtered differently in order to enhance features of different, but related origin. Generally one fuses the signals from several sensors to get a better result.

Figure 9.1 also shows the pulse images being fed into an FPF (fractional power filter). The FPF needs to be modified slightly to handle multiple channels. This can

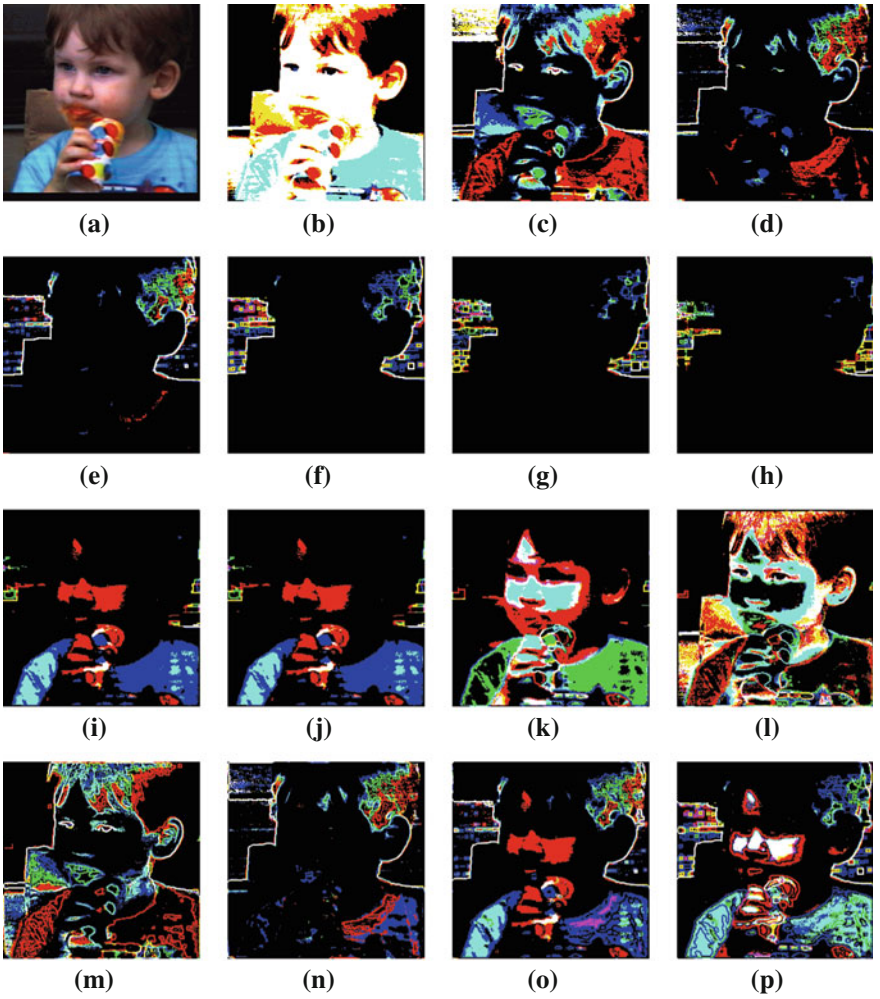


Fig. 9.2 An original image and the generated pulse outputs

be accomplished by phase encoding the constraint vector and treating each channel as a separate 2D input.

Thus, the final output of the multi-channel PCNN is a complex image that contains data from all channels,

$$\mathbf{Y}^T = \sum_{\epsilon} \mathbf{Y}^{\epsilon} e^{\iota 2\pi \epsilon / N}, \tag{9.6}$$

where ϵ represents each channel. The FPF is trained on a multi-channel target selected in the original images. For training views of the target are cut-outs from a few of the original images. Each training image is

$$\mathbf{x}_i^{\epsilon} = e^{\iota 2\pi \epsilon / N} \mathbf{S}^{\epsilon}, \tag{9.7}$$

and trained with the corresponding constraint,

$$c_i = e^{\iota 2\pi\epsilon/N}, \tag{9.8}$$

and the filter is trained according to Eqs. (C.1)–(C.3).

The final result is the correlation, $\mathbf{Z}$,

$$\mathbf{Z} = \mathbf{h} \otimes \mathbf{Y}^T. \tag{9.9}$$

The presence of a target will produce a large correlation signal at the location of the target. An exact match will produce a signal of height N. Stimuli that do not exactly match the training targets will have a lower correlation value depending upon the fractional power.

The data set and the defined task assist in determining the value of α. If the data set varies greatly then α can be lowered. If the targets consist mainly of low frequency data, the mid-point of the trade-off will move to a higher value of α. If the task requires a higher precision of discrimination the user should give a larger value of α. Generally, the best way to determine the proper value of α is to try several values. The entire fusion algorithm follows the following prescription:

1. Given a defined target training set, $\mathbf{X}$, the FPF filter is generated.
2. Given a stimulus, $\mathbf{S}$, with ϵ channels each channel is separately represented by $\mathbf{S}^\epsilon$. All arrays of the PCNNs are initialised to 0.
3. One ϵPCNN iteration is performed following Eqs. (9.1)–(9.4).
4. The outputs $\mathbf{Y}^\epsilon$ are phase-encoded to form $\mathbf{Y}^T$ as by Eq. (9.5).
5. The correlation $\mathbf{Z}$ is calculated to identify the presence of a target by Eq. (9.9). A large correlation spike is indicative of the presence of a target.
6. Steps 3–5 are repeated until a target is clearly evident or several iterations have produced no possible target identifications. The actual number of iterations is problem dependent and currently determined by trial and error. Generally, 10–20 iterations are sufficient.

The example below uses the colour-input image in Fig. 9.2a. This image is a three-channel image. The ϵPCNN responses are also shown. The target was selected to be the boy's ice cream. Cutting out and centring the ice cream created the FPF training image. In traditional filtering, creating a training image from a cut-out may be dangerous. This is because the cutting process induces artificial edges. These can constitute Fourier components. These generally differ significantly from the original image and may consequently change the correlation result. In other words, the filter may have large Fourier components that will not exist in the input. The PCNN, however, produces pulse images, which also contain very sharp edges, so in this case cutting out a target is not detrimental. Both components of the correlation contain sharp edges so the cutting-out process will not adversely affect the signal to noise of the correlation.

The FPF with $\alpha = 0.8$ is shown in Fig. 9.3. Each of the ϵPCNN pulses was correlated with the filter. The correlation surfaces for the first four non-trivial multi-

Fig. 9.3 The complex Fractional Power Filter (FPF) of the target

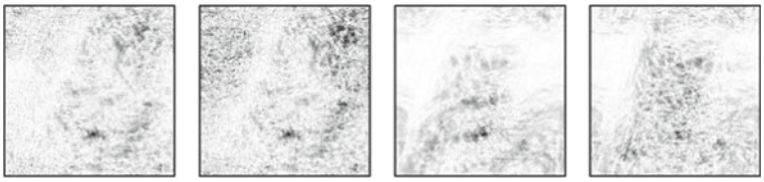


Fig. 9.4 Correlation surfaces of the first four non-trivial PCNN responses. Images shown are for $n = 1, 2, 7$ and 8

channel pulses are shown in Fig. 9.4 (where a trivial output is one in which very few neurons pulse).

The trivial and some of the non-trivial output pulse images do not contain the target. In these cases no significant correlation spike is generated. The presence or absence of a target is information collected from the correlation of several pulse images with the filter. The target will produce a spike in several of these correlation surfaces. The advantage is that the system tolerates a few false positives and false negatives. An occasional correlation spike can be disregarded if only one output frame in many produced this spike, and a low target correlation spike from one pulse image can be overlooked when many other pulse images produced significant spikes at the target location. This method of the accumulation of evidence has already been shown in [52].

The correlation surface for all shown iterations display a large signal that indicates the presence of the target. It should be noted that the target appears only partially in any channel. From the multi-channel input a single decision to indicate the presence of the target is reached. Thus, image fusion has been achieved.

As for computational speed, the PCNN is actually very fast. It contains only local connections. For this application $\mathbf{M} = \mathbf{W}$ so the costly computations were performed once. For some applications it has been noted that using $\mathbf{M} = 0$ decreases object crosstalk and also provides the same computational efficiency. Also quick analysis of the content of $\mathbf{Y}$ can add to the efficiency for the cases of a sparse $\mathbf{Y}$. In software simulations, the cost of using the FPF is significantly greater than using the ϵPCNN.

9.1.2 Python Implementation

Implementation of a 3D PCNN requires a few alterations from the original algorithm. Like the original the presented scripts encapsulate the algorithm in as an object for instances when an application requires multiple instances of the PCNN.

The initialisation is shown in Code 9.1. The constructor is very similar in nature to the original in Code 4.1.

Code 9.1 The constructor for *ucm3D*.

```
# ucm3D.py
from numpy import array, ones, zeros
from scipy.signal import cspline2d
import mgconvert, levelset

class UCM3D:
    "Datacube Unified Cortical Model"
    f,t1,t2 = 0.9,0.8,20.0

    # constructor
    def __init__ (self,dim):
        # dim = (N, vert, horz )
        self.F = zeros( dim )
        self.Y = zeros( dim )
        self.T = ones( dim )
```

A new function that is needed is **Image2Stim** function shown in Code 9.2 in which an image is converted to a data cube. Line 4 calls the **RGB2cube** function which is described in Appendix A. This function reads in the image and returns a list of matrices for the RGB channels. These are converted to a single array in Line 5 and then slightly smoothed in Lines 6 and 7 to eliminate random noise.

Code 9.2 The **Image2Stim** function.

```
# ucm3D.py
    def Image2Stim( self, mg, smooth=2 ):
        if mg.mode != 'RGB': mg = mg.convert('RGB')
        stim = mgconvert.RGB2cube( mg )
        self.stim = array( stim )/255.0
        for i in range( len( self.stim )):
            self.stim[i] = cspline2d( self.stim[i], smooth )
```

A single iteration is shown in Code 9.3. Again it is similar to the original except that the neural communications also come from other channels.

Code 9.3 The **Iterate** function.

```
# ucm3D.py
    def Iterate (self):
        N = self.F.shape[0] # Z dimension
        work = zeros( self.Y.shape )
        for i in range( N ):
            sc = 1.
            Y = self.Y[i]+0
            if i>1:
                Y=Y + 0.7* self.Y[i-1]
                sc = sc + 0.7
            if i>2:
                Y=Y + 0.3* self.Y[i-2]
                sc = sc + 0.3
            if i<N-1:
                Y=Y + 0.7* self.Y[i+1]
                sc = sc + 0.7
            if i<N-2:
                Y=Y + 0.3* self.Y[i+2]
                sc = sc + 0.3
            Y = Y / sc
            if Y.sum() >0:
                work[i] = cspline2d( Y, 25 )
            else:
                work[i] = zeros( Y.shape )
        self.F = self.f * self.F + self.stim + work
        self.Y = self.F > self.T
        self.T = self.t1 * self.T + self.t2 * self.Y + 0.1
```

The final function is **Y2Image** shown in Code 9.4 which converts a single pulse frame into a colour image. For cases in which the input has three channels (colour images) the conversion of **Y** to an RGB image is simple. For cases in which the input is more than three channels the conversion is a little more involved. Computer displays still have only three channels and therefore a mapping is required to convert more than 3 channels into the RGB display.

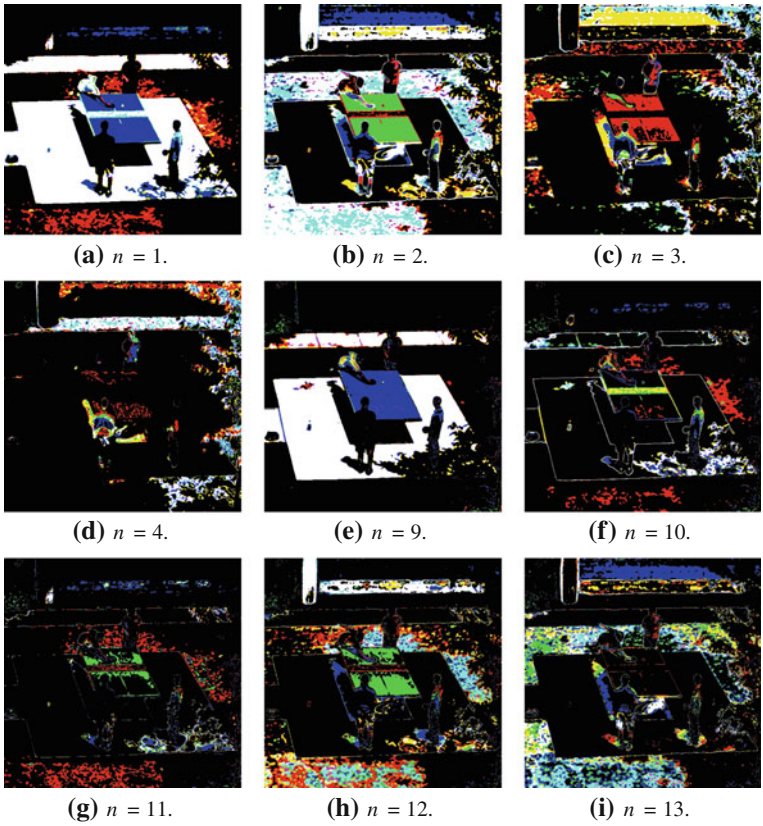
Code 9.4 The **Y2Image** function.

```
# ucm3D.py
    def Y2Image( self ):
        # converts the several Y's into an image
        N = self.F.shape[0]
        if N==3:
          mg=mgconvert.Cube2Image(self.Y[0],self.Y[1],self.Y[2])
        else:
            V,H = self.Y[0].shape
            r,g,b = zeros((V,H)),zeros((V,H)),zeros((V,H))
            for i in range( self.Y.shape[0] ):
                ii = float(i)/(self.Y.shape[0]-1)
                a = (-2. * ii + 1)*self.Y[i]
                r = r + clip( a,0.,a.max() )
                g = g +(-4*ii*ii + 4*ii) * self.Y[i]
                a = (2 * ii -1.)*self.Y[i]
                b = b + clip(a,0.,a.max() )
            # scale
            mn = r.min()
            a = g.min()
            if a < mn: mn = a
            a = b.min()
            if a < mn: mn = a
            r,g,b = r-a,g-a,b-a
            mx = r.max()
            a = g.max()
            if a > mx: mx = a
            a = b.max()
            if a > mx: mx = a
            if mx == 0: mx = 1  # prevents DIV0 errors
            r,g,b = r/mx,g/mx,b/mx
            r,g,b = r*255,g*255,b*255
            mgconvert.Cube2Image( r,g,b )
        return mg
```

Calls to the functions are shown in Code 9.5 and the results are shown in Fig. 9.5. In this example the image is from Fig. 4.15a and the pulse images are shown here with two cycles starting at $n = 1$ and $n = 9$. Segmentation is obvious and the desynchronisation according to texture can also be seen.

Code 9.5 Example implementation.

```
>>> mg = Image.open( fname)
>>> net = ucm3d.UCM3D( (3,500,514) )
>>> net.Image2Stim( mg )
>>> Y = []
>>> for i in range( 15 ):
        net.IterateLS( )
        Y.append( net.Y2Image() )
```

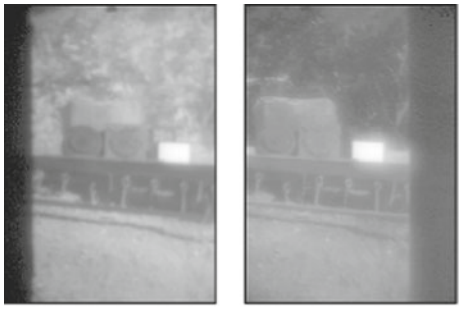


(a) $n = 1$. **(b)** $n = 2$. **(c)** $n = 3$.

(d) $n = 4$. **(e)** $n = 9$. **(f)** $n = 10$.

(g) $n = 11$. **(h)** $n = 12$. **(i)** $n = 13$.

Fig. 9.5 Pulse images for a colour input



(a) (b)

Fig. 9.6 4102dt0 (*long wavelength*) and 4130dt0 (*short wavelength*)

Fig. 9.7 The output for $n = 7$ and channels 18 and 24

(a) (b)

9.2 Multi-Spectral Example

This demonstration uses images collected by an experimental AOTF (acousto-optical tunable sensor) [19, 20]. An AOTF acts as an electronically tunable spectral bandpass filter. It consists of a crystal in which radio frequencies (RF) are used to impose travelling acoustic waves in the crystal with resultant index of refraction variations. Diffraction resulting from these variations cause a single narrowband component to be selected from the incoming broadband scene. The wavelength selected is a function of the frequency of the RF signal applied to the crystal. Multiple narrowband scenes are be generated by incrementally changing the RF signal applied to the transducer. The selected wavelength is independent of device geometry. Relative bandwidth for each component wavelength is set by the construction of the AOTF and the crystal properties. The sensor is designed to provide 30 narrow band spectral images within the overall bandpass of 0.48–0.76 μm.

Figure 9.6 contains examples of individual channel input images. The image designated as *4102dt0* is near the long wavelength end of the spectrum. The second example, *4130dt0* is the shortest wavelength image. Figure 9.7 contains examples

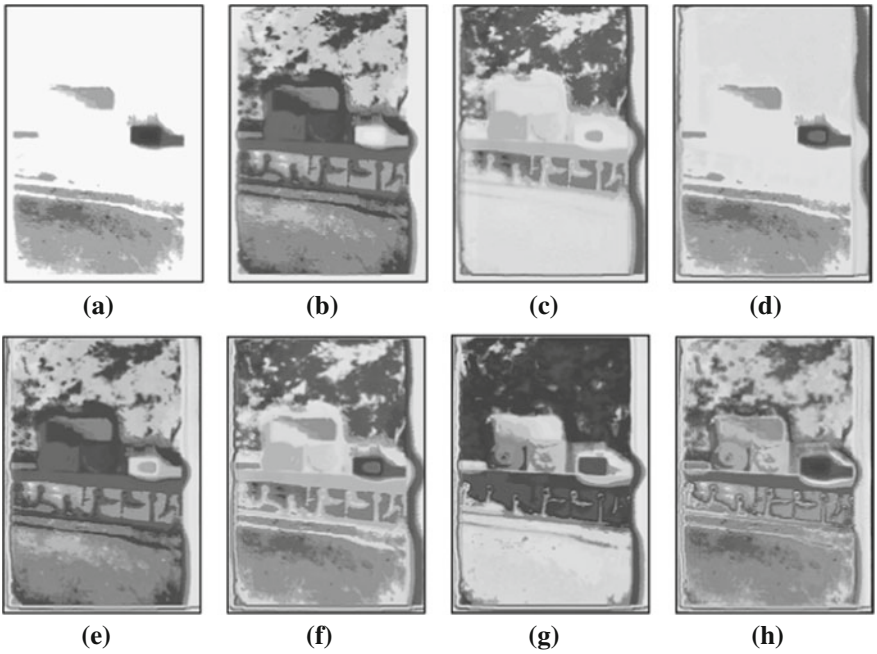


Fig. 9.8 *Grey* scale representations of the multi-channel pulse images

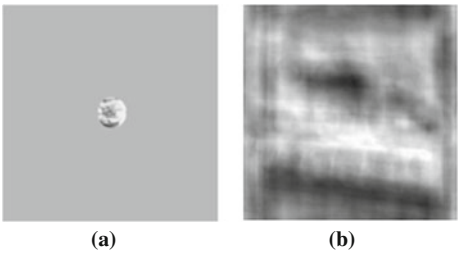


Fig. 9.9 **a** The amplitude of a spiral filter. **b** The correlation of the filter with a portion of the iteration 7. *Dark pixels* indicate a higher response

of individual channel binary outputs of the PCNN. These specific examples were chosen because they display features associated with the mines. Figure 9.8 displays a grey scale representation of all channels in particular iterations. The grey encoding coarsely displays phase. As can be seen the targets become quite visible in detail for some iterations (e.g. $n = 7$).

Figure 9.9a displays the amplitude of the spiral filter built to detect one of the targets by the FPF method. Figure 9.9b displays a 3-D plot of the correlation surface between the filter and iteration 7.

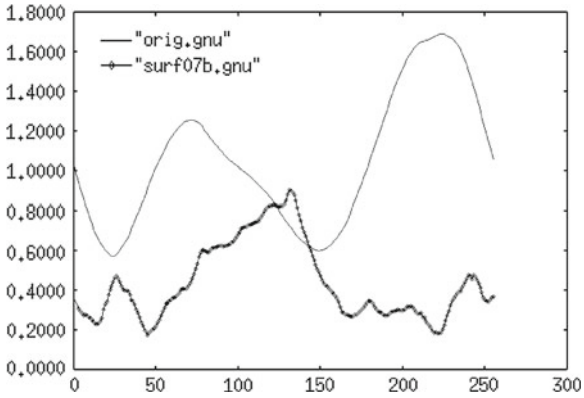


Fig. 9.10 Cross-sections of a correlation surface for the target and the original image (orig.gnu) and the spiral filter with pulse image $n = 7$

Figure 9.10 displays a cross-sectional slice of the correlation surface that passes through the peak of the correlation (labelled surf07b). As can be seen, the correlation produces a signal that is significantly greater than the noise. Similar performance was obtained from the filter built for the other target. The other plot in the figure is the correlation between the spoked land mine (immediately attenuated by a Kaiser window) and the original image of channel 20. As can be seen that this correlation function does little to indicate the presence of a target. The large signal is from the Halon plate. A drastic improvement is seen between this correlation and that produced through the spiral filter.

Not all iterations will contain the target as a segment. This is the inherent nature of the pulse images. The particular iteration in which the target appears is dependent upon scene intensity. It may coincidental that immediately neighbouring objects may pulse in the same iteration as the target making it difficult to distinguish the border between the two objects in the output. FPF correlations may still produce a significant correlation if a majority of targets edge is present and other iterations will separate these neighbouring objects. Figure 9.11 displays a 1D correlation slices (through the target region) for the first seven iterations. Iterations $n = 2$ and $n = 5$ produced the largest correlation signals, but the width of the signals (caused by the neighbouring location of the two land mines and their similarity in overall shape and size) prevents these iterations from being indicative of the target land mine. The Halon plate also causes significant signals due to its very high intensity which can be highly reduced by normalizing the correlation surface with a smoothed input intensity image. Iteration $n = 7$ (which is also displayed in Fig. 9.10) indicates that a target exists.

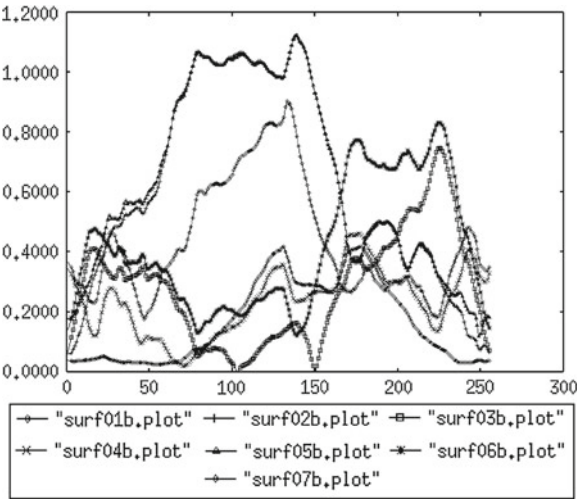


Fig. 9.11 Cross-sections of the spiral filter with coded pulse images from several iterations

9.3 Application of Colour Models

For the purposes of image processing the RGB colour model is usually a poor choice for representing information. There are several other models that separate intensity information from hue information which commonly provides better performance. All of the Python modules necessary to consider data in an alternate colour models have been presented in other sections.

The example shown here converts an image from the RGB colour format to the YUV image format where the Y channel represents intensity information and the U and V channels represent differences in the chroma information. The conversion is a linear system described as,

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.11 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & 0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{9.10}$$

Code 9.6 shows the Python commands that read in the image (Line 3) and converts it to a data cube (Line 4) using the **RGB2cube** function which is detailed in Appendix A. The RGB channels are normalized so that the values are floats between 0 and 1 and then smoothed slightly in Line 8. Line converts the RGB channels to YUV channels using the **RGB2YUV** function also described in Appendix A.

Code 9.6 Converting an image to the YUV format.

```
>>> import Image, mgconvert
>>> from scipy.signal import cspline2d
>>> mg = Image.open( fname)
>>> rgb = mgconvert.RGB2cube( mg.convert('RGB') )
>>> rgb = rgb.astype( float )
>>> rgb /= 255.0
>>> for i in range( 3 ):
        rgb[i] = cspline2d( rgb[i], 2 )
>>> yuv = mgconvert.RGB2YUV( rgb[0], rgb[1], rgb[2] )
```

Three ICM networks are constructed in Code 9.7 with the initialisation in Lines 1–4. A call to the iterator for each is in Lines 7 through 9. The results of each iteration is converted back to an RGB image in Line 10.

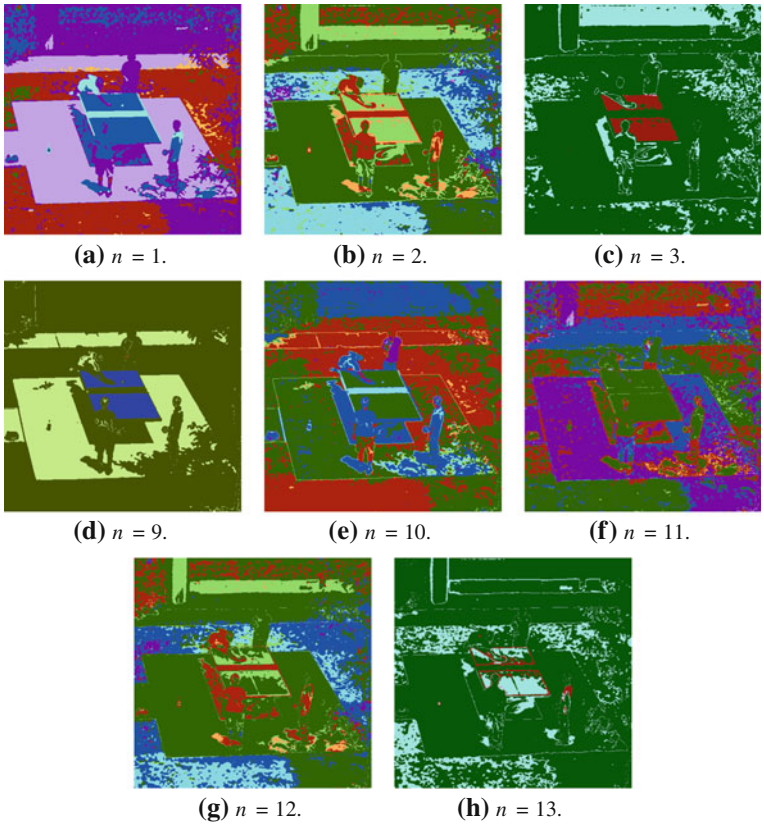
Code 9.7 Running 3 ICMs.

```
>>> VH = rgb[0].shape
>>> n1 = icm.ICM( VH )
>>> n2 = icm.ICM( VH )
>>> n3 = icm.ICM( VH )
>>> Y = []
>>> for i in range( 15 ):
>>>     n1.IterateLS( yuv[0] )
>>>     n2.IterateLS( yuv[1] + 0.5 )
>>>     n3.IterateLS( yuv[2] + 0.5 )
>>>     a = color.YUV2RGB( n1.Y, n2.Y-0.25, n3.Y-0.25 )
>>>     Y.append( a )
```

Finally, Code 9.8 converts the pulse data to image formats and stores them to disk. The pulse images that have significant pulse activity are shown in Fig. 9.12.

Code 9.8 Saving the pulse images as colour images.

```
>>> for i in range( 15 ):
>>>     mg = mgconvert.Cube2Image( Y[i][0], Y[i][1], Y[i][2] )
>>>     mg.save('yy' + str(i) + '.png')
```



(a) $n = 1$. **(b)** $n = 2$. **(c)** $n = 3$.

(d) $n = 9$. **(e)** $n = 10$. **(f)** $n = 11$.

(g) $n = 12$. **(h)** $n = 13$.

Fig. 9.12 Pulse images for an ICM using YUV format as inputs

9.4 Summary

This chapter considered colour and multi-spectral image formats with a variety of architectures. Each case showed crisp segmentation of the objects within the frame. Some architectures considered networks with inter-channel communications while the last architectures considered independent networks based but on data that separated intensity and hue information.

This chapter did not cover the wide variety of possible architectures or applications. Instead it demonstrated a few architectures and their performance to indicate the viability of PCNN and ICM as a processor of data cubes.

Chapter 10
Image Signatures

With the advent of the cheap digital camera we have the ability to overwhelm ourselves with digital images. Thus, there is a need to be able to describe the contents of images in a condensed manner. This description must contain information about the content of the images rather than just a statistical description of the pixels. Measurements of the activity in the brain of small mammals indicate that image information is converted to small one-dimensional signals. These signals are dependent upon the shapes contained in the input stimulus. This is a drastic reduction in the amount of information used to represent the input and therefore is much easier to process.

The goal is then to create a digital system that condenses image information into a signature. This signature must be dependent upon the contents of the image. Thus, two similar signatures would indicate that the two images had similar content. Once this is accomplished it would be fairly easy to construct an engine that could quickly find image signatures similar to a probe signature and thus find images that had content similar to the probe image.

The reduction of images to signatures using the PCNN and ICM have been proposed for some time. This chapter will explore the current state of this research.

10.1 Image Signature Theory

The idea of image signatures stems from biological research performed by McClurken et al. [70]. They measure the neural response of a macaque to checker board style patterns. The brain produced neural patterns that were small and indicative of the input stimulus. They also used colour as the input stimulus and measured the colour response. Finally, a colour pattern stimulus led to a signature that was the multiplication of the pattern signature and the colour signature.

Converting images to small signatures would be of great benefit to digital image searches for two reasons. The first reason is that images do consume memory

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_10,

resources. JPEG compression provides a reduction of about a factor of 10. While this is impressive it may be insufficient for large databases. For example, a database of 10,000 colour images that are 512×512 will still consume several gigabytes. This is manageable for today's hard drives, but it will still take time to read, decompress, and process this amount of information. Thus, image signatures would provide an efficient representation of the image data. The second reason is that image signatures would be extremely fast to process.

10.1.1 The PCNN and Image Signatures

The creation of signatures with the PCNN was first proposed by Johnson [46]. In this work two objects with equal perimeter lengths and equal areas were used. Several images were created by rotating, shifting, scaling and skewing the objects. Johnson showed that after several PCNN iterations the number of neurons firing per iteration became a repetitive pattern. Furthermore, the pattern for each shape was little changed by the input alterations. Thus, it was possible to determine which shape was in the input space by examining the repetitive integrated pulse activity.

This experiment worked well for single objects without a background, but problems awaited this approach. First, it required hundreds of PCNN iterations which were time consuming. Second, the signature changed dramatically when a background was used. It was no longer possible to determine the input object by examining the signature. The reason for this became clear when the problem of interference was recognised (see Sect. 4.2.3.1). Interference occurred when the neurons from one object dramatically changed the activity of neurons dedicated to another object. Thus, the presence of a background, especially a bright one, would significantly alter when the on-target neurons would pulse, and in turn, change the signature as demonstrated in Sect. 4.2.3.1. The pulsing activity of the neurons on the flower were significantly changed by the presence of a background.

The solution to the interference problem was to alter the inter-neuron connectivity scheme. The ICM therefore employs a more complicated scheme in which the connections between the neurons are altered with each iteration. Now, the signatures of on-target neurons are not altered and it is much easier to determine the presence of a target from the signature.

Johnson's signature was just the integration of the neurons that pulse during each iteration,

$$G[n] = \sum_{ij} Y_{ij}[n]. \tag{10.1}$$

There are still several concerns with this method. The first is that if the target only filled 20 % of the input space then only 20 % of the signature would be derived from the signature. Thus, the target signature could be lost amongst the larger background signature. The second concern is that it is still possible for objects of different

shape to produce the same signature. The second concern was addressed by adding a second part to the signature. The signature in Eq. (10.1) represented the area of the pulsing neurons. Area does not indicate shape and since shape is important for target recognition a second part of the signature was added that was highly dependent on the shape [57]. This additional component was,

$$G[n+N] = \sum_{ij} Z\{\mathbf{Y}[n]\}_{ij}, \tag{10.2}$$

where N is the total number of iterations and $Z\{\cdot\}$ is an edge enhancing function. Basically, this second component would count again the neurons that were on the edge of a collective pulsing segment.

Thus, the signature for a grey scale image was twice as long as the number of ICM iterations. Usually, N ranged from 15 to 25 so at most the length of the signature was 50 integers. This was a drastic reduction in the amount of information required to represent an image. The following sections will show results from using this type of signature.

10.1.2 Colour Versus Shape

Another immediate concern is that of colour. Most photo-images contain 3 colour bands (RGB), and it is possible to build a 3-channel ICM. However, it became apparent through trials that it was not necessarily better to process the colours in this manner. The question to be asked is what is the most important part of an image? Of course, this is based on specific applications, but in the case of building an image database from generic photos the most important information is the shapes contained within the image. In this case, shape is far more important than colour. Thus, one option is to convert the colour images to grey scale images before using the ICM. The logic is that the signature would be indicative of the shapes in the image and not the colour of the shapes. However, the debate as to use colour information or not is still ongoing.

10.2 The Signature of Objects

The ideal signature would be unique to object shape but invariant to its location in the image and to alterations such as in-plane rotation. This is a difficult task to accomplish since such a drastic reduction in data volume makes it more likely to have two dissimilar objects reduce to similar signatures. There are two distinct objects shown in Fig. 10.1a, b. The signatures for these two objects were computed independently using Eqs. (10.1) and (10.2) but they are plotted together in Fig. 10.2.

Fig. 10.1 **a** An input image and **b** a second image

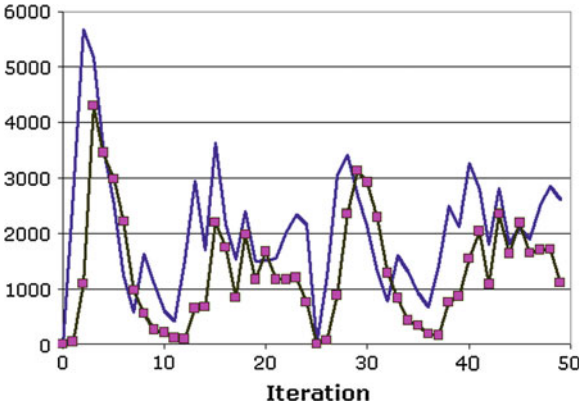


Fig. 10.2 The signatures of the two objects. The x-axis corresponds to the iteration index and the signature with the *boxes* corresponds to the image in Fig. 10.1b

The plot with the square boxes belongs to Fig. 10.1b. Since Eqs. (10.1) and (10.2) are independent of location or rotation then neither shifting the image nor rotating the image will alter the signature by an appreciable amount.

Combining the two objects into the same image will create a signature that is the summation of the signatures of the two objects. The plots in Fig. 10.3 display the summation of the two signatures in Fig. 10.1 and the signature from an image that contains the two objects. As can be seen, the signature of the two objects in the same image is the same as the summation of the two signatures. This may seem trivial but it is an important quality. For if this condition did not hold then attempts at target recognition would be futile. This was the case in the original PCNN signatures.

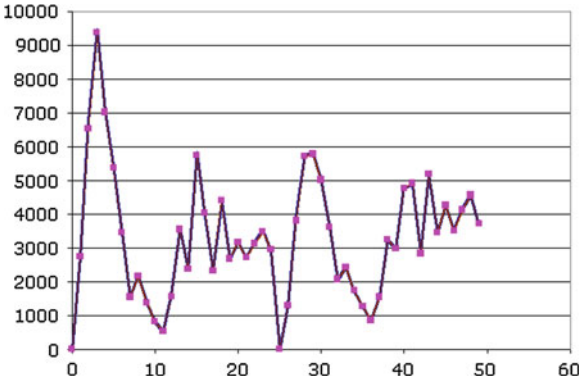


Fig. 10.3 The summation of the plots in Fig. 10.2 and the signature of a single image containing both objects of Fig. 10.1a, b

10.3 The Signatures of Real Images

The presence of a background was debilitating for the original signature method. So far the new signature method has eliminated the presence of interference, but it still remains to be seen if it will be possible to recognise a target in a real image. The image in Fig. 10.4a is the 'background' and the image in Fig. 10.4b displays the vendor pasted on top of the background.

To determine the capability of identifying a target the signature of these two images is considered. The philosophy is that the signatures of different targets are

(a) **(b)**

Fig. 10.4 **a** Background and **b** the vendor on the background

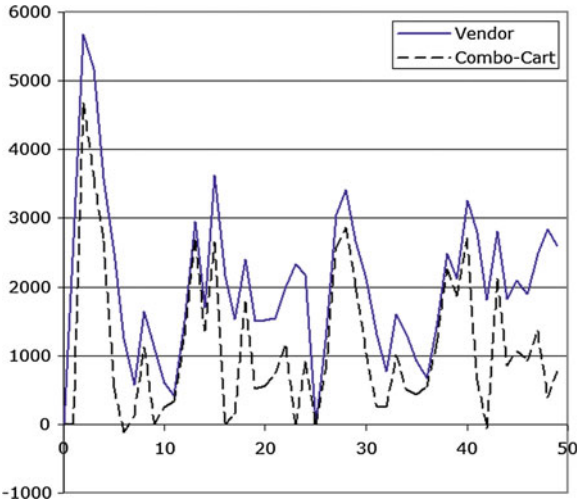


Fig. 10.5 The signature of the vendor (Fig. 10.1a) and the difference of the signatures of the images in Fig. 10.4a, b. The similarity of these two plots indicate that it is possible to identify the target. The *solid line* represents the signature from the vendor

additive. Thus, G[photo] = G[background] + G[vendor] G[occluded background]. It is expected that the signature of the vendor added to the signature of the background should almost be the same as the signature of the image in Fig. 10.4b. The difference will be the portion of the background that is occluded by the target.

The chart in Fig. 10.5 displays the signature of the vendor and the difference of the signatures of the two images in Fig. 10.4a, b. The target can be recognised if these two plots are similar. In cases where the background is predictable it is possible to estimate G[occluded background].

10.4 Image Signature Database

Another application of image signatures is to quickly search through a database of images. The signatures are far easier to compare to each other than the original images since the volume of data is dramatically reduced. Furthermore, the signatures can be compared by simple algorithms (such as subtraction) and the comparison is still independent of shift, in-plane rotation, and limited scaling. A comparison algorithm with the same qualities operating on the original images would be more complicated.

A small database of 1000 images from random sites was created. This provided a database with several different types of images of differing qualities. However, since some web pages had several images dedicated to a single topic the database did contain sets of similar images. There were even a few exact duplicates and several duplicates differing only in scale. The only qualifications were that the images had to be of sufficient size (more than 100 pixels in both dimensions) and sufficient variance

Table 10.1 Signature scores

Class	Scores
Different scales	11 examples above 0.9447, 1 each at 0.9408 and 0.9126
Different aspect	0.902
Similar objects	0.938, 0.918, 0.916, 0.912, 0.910, 0.909, 0.908
Somewhat similar objects	0.922, 0.912, 0.912, 0.910, 0.908, 0.908, 0.906, 0.906
High scoring mismatched	0.943, 0.920, 0.912, 0.909

in intensity (to prevent banners from being used). The database itself consisted of the signatures, the original URL and the data retrieved as it was not necessary to keep the original images. Each image thus required less than two hundred bytes except for cases of very lengthy URLs.

Comparison of the signatures was accomplished through a normalized subtraction. Thus, the scalar representing the similarity of two signatures $\mathbf{G}_q$ and $\mathbf{G}_p$ was computed by

$$a = 1.0 - \prod_n |\big(\|G_p[n]\| - \|G_q[n]\|\big)|. \tag{10.3}$$

The signatures were normalized to eliminate the effects of scale.

Comparisons of all possible pairings were computed. Since there were 1000 images in the database there were 499,000 different possible pairings (excluding self-pairings). The top scores were found and the images were manually compared.

In the database there are eight duplicate images (each pairing scored a perfect 1.0 by Eq. (10.3)). Most of the pairings scored below 0.9 and belonged to images that were dissimilar. Table 10.1 displays the results of the matches that scored above 0.9. There were 13 pairings in which but images were the same except for a scale factor. There was one pairing of the same image but the scale in the horizontal and vertical were different (1.29 and 1.43). There were several pairings of similar objects. Of these 499,000 different pairings, four scored high and contained images that did not appear to have similarity.

It was possible to mostly separate the perfect matches, the scaled pairings, and the similar images from the rest. This, of course, was not an exhaustive study since it was time consuming.

10.5 Computing the Optimal Viewing Angle

Contributed by Nils Zetterlund, Royal Inst. of Tech. (KTH), Stockholm

Another application is to select an optimal viewing angle for a 3D target. For example, if we wish to place a camera alongside of a road to take images of autos then we need to ask: what is the best placement for the camera? Should it look at the cars from a certain height and from a certain angle to the travelling direction?

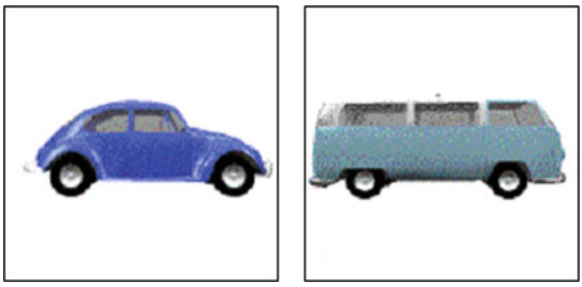


Fig. 10.6 Digital models of two vehicles

In order to answer this question we may wish to consider all possible angles of a few vehicles. We will then have to decide on a metric to define the 'best viewing angle'. The first problem encountered is that there can be a massive volume of data. Consider a case where the camera can take images that are 480×640 in three colours. There are 921,600 pixels per image. If we rotate this object about a single axis and take a picture every 5 degrees, then there will be 72 pictures each of 900 K pixels. If this object is rotated about three axes and take a picture for every 5 degrees of rotation then there will be about 3.4×10^{11} pixels of information. Obviously, we can not simply compare images from the different viewing angles. Image signatures have the ability to dramatically reduce the volume of information that needs to be processed and they can be used to determine the optimal viewing angle.

For this test two artificial objects will be considered. These two vehicles are shown in Fig. 10.6. Each object was rotated about a vertical axis and images for each 5 degrees of rotation were computed. For each image the signature was computed. In order for the signatures to be useful they must smoothly change from one angle of rotation to the next. That is, each element in the signature must not change in a chaotic fashion as the angle is increased. Thus, we should be able then to predict values of the elements of the signature given a sparse sampling.

A sampling set consisted of the signatures for every 30 degrees. This set then was $G_{b,\theta,n}$ where b is the class (bus or beetle), θ is the angle of viewing, and n is the element index. Given the G's for $\theta = 0, 30, 60$, is it possible to predict the values of **G** for $\theta \neq$'0, 30, 60,? Using a Gaussian interpolation function the prediction of the intermediate **G** values becomes feasible. The charts in Fig. 10.7 display the measured and estimated values of **G** for two fixed values of n. The estimation does indicate that the intermediate elements are somewhat predictable. This in turn validates the use of signatures for the representation of viewing angle information.

The next step in the process is to compare the signatures of two objects to find the view that is most distinguishing. In other words, we seek the viewing angle that best discriminates between the two objects. This angle will be the angle in which the signatures between the two objects differ the most. Figure 10.8 displays the first order difference between the respective signatures of the two objects. This is shown

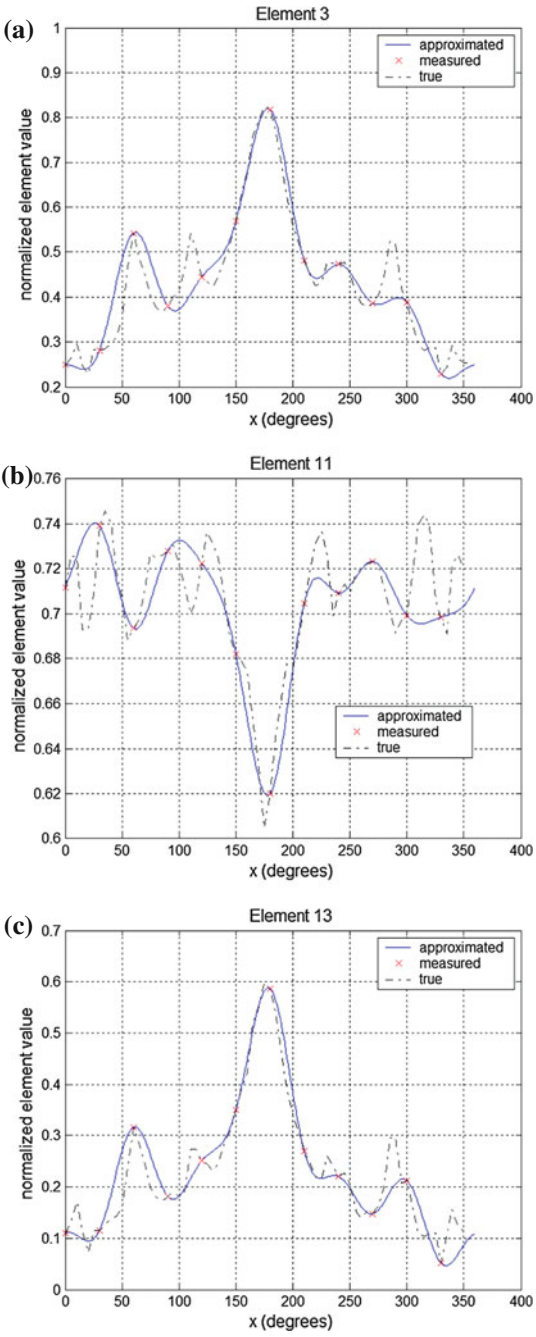


Fig. 10.7 **a** The actual and estimated values of element 3 of the signature. **b** The actual and estimated values of element 11 of the signature. **c** The actual and estimated values of element 13 of the signature

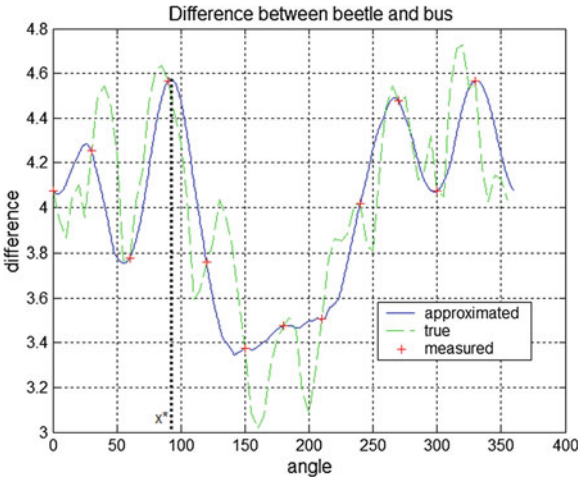


Fig. 10.8 The first order difference between the two targets

for both the 5 and 30 degree cases. It is seen that there are in fact four angles that greatly distinguish these two objects. The first is at 90 degrees which is a side view (Fig. 10.6). The second is at 270 degrees which is the other side view. The third and fourth correspond to views at the front corners of the vehicles. Likewise, views that are directly at the front or the back of the vehicles are shown to be poor discriminators.

10.6 Motion Estimation

Image signatures have also been employed to estimate the velocity of an object. A moving input will alter the signature of a static object. The overall characteristic of the signature is maintained but small alterations are indicative of motion. The current method of computing the signature is insensitive to opposing movement. For example, an object moving in the $-x$ direction is difficult to distinguish from the same object moving in the $+x$ direction. Thus, the signature calculation is altered to be sensitive to directionality,

$$g[n] = \sum_{i,j} = Y_{i,j}[n], \tag{10.4}$$

$$g[n+N] = \sum_{i,j} (\nabla_x Y)_{i,j}\,, \tag{10.5}$$

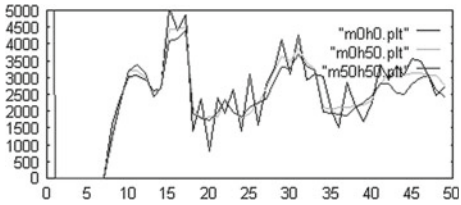


Fig. 10.9 The signature of a static object (*roughest curve*), the same object moving at velocity of (0,50) (*lightest curve*), and the same object moving at a velocity of (50,50)

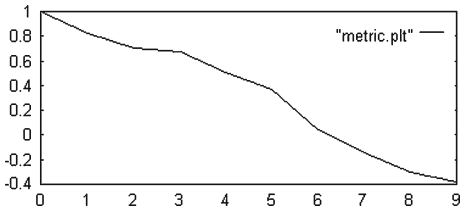


Fig. 10.10 The comparison of signatures at different velocities to the signature of the static case. The x-axis is increasing velocity and $x = 9$ is a velocity of (0,45)

and

$$g[n + 2N] = \sum_{i,j} \left(\nabla_y Y\right)_{i,j}, \tag{10.6}$$

where N is the number of iterations (in this study $N = 25$) and ∇_x is the spatial derivative in the x direction. Thus, the signature has three times as many elements as the number of iterations. A comparison score of two signatures (g_p, g_q) is computed by Eq. (10.3).

Figure 10.9 displays signatures from a static and moving object. The alterations to the image signature are dependent upon the speed, the direction of the motion, and the object shape. Signatures of a moving target can be computed by Eq. (10.3). Consider an object capable of moving in 2D space with a constant velocity. We can construct a velocity space $\mathcal{R}^2$ and compare the signature of all possible velocities to the signature of the static target. This comparison is shown in Fig. 10.10 where the curves depict similar values of the velocity difference Δv.

These curves are named iso-Δv since they depict constant values of the velocity difference. Of course, the *anchor velocity* does not have to be $v = 0$. We can compare all velocities in the space to any single velocity. If the iso-Δv values were solely dependent upon the difference in velocity then these curves would be circles. However, there is also a dependency upon the object shape. Thus, we expect the different iso-Δv to be similar in shape for a single target. As the Δv increases the iso-Δv curves lose their integrity, so there is an effective radius—or limit on how large Δv can be in order for Eq. (10.3) to be valid.

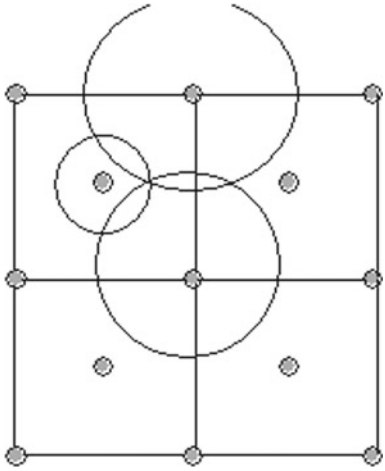


Fig. 10.11 A velocity grid with 13 anchor points. The middle point is $v = (0, 0)$. The *three circles* are the iso-Δv for an unknown. The *three circles* intersect at a single location. This location is the estimate of v

Now, consider the case of a target with an unknown velocity $v_?$. It is our task to estimate this velocity using image signatures. If we compare $v_?$ to a v_{x1} that is sufficiently close to $v_?$ then a value from Eq. (10.3) can be computed. However, this computation does not uniquely identify $v_?$. Rather, the computed value defines an iso-Δv curve surrounding v_{x1}. If we compare the signature from $v_?$ to two other knowns v_{x2} and v_{x3} then the three iso-Δv curves will intersect at a single location. This triangulation method is shown in Fig. 10.11. The estimate of $v_?$ is the point in $\mathcal{R}^v$ space where the three iso-Δv curves intersect.

The only caveat is that v_{x1}, v_{x2} and v_{x3} must be sufficiently close to $v_?$. Since $v_?$ is unknown it is not possible to define v_{x1}, v_{x2} and v_{x3}. To circumvent this problem many anchor velocities are considered. In the trial case 13 anchor points distributed evenly in $\mathcal{R}^v$ were used. The signature from $v_?$ was compared to the signatures from these anchor points. The three points with the highest comparison value (Eq.(10.3)) were then used for the triangulation method.

For the example case $\mathcal{R}^v$ was defined by 100×100 points with the static v_0 defined at (50,50). The maximum velocity was one pixel of target movement per iteration of the ICM. All 10,000 points in $\mathcal{R}^v$ were considered as $v_?$. In all cases this method correctly estimated $v_?$ with an accuracy of $(\pm 1, \pm 1)$. For the cases in which there was a single element error the signature of $v_?$ and the correct answer were identical. Thus, the velocity was accurately predicated for all cases.

10.7 Summary

The image signatures are an efficient method for reducing the volume required to represent pertinent image information. The signatures are unique to the shapes inherent in the image and are loosely based on biological mechanics. The reduction in

information volume allows for the construction of an image database that can be searched very quickly for matching images. Other uses include the determination of the optimal viewing angle and the estimation of motion.

Chapter 11
Logic

11.1 Maze Running and TSP

Running a maze such as the one shown in Fig. 11.1 can be accomplished via the autowave propagation of the PCNN. The goal of this section is to start at the leftmost location and work towards the rightmost location finding the shortest path along the way.

This application requires a couple of small changes to the PCNN. First, the autowave should not bridge gaps. In other words, if a wave is progressing along one path it should not activate the neurons in a nearby path just because the two paths are within a small vicinity. Therefore, the first modification is to limit neural communications to only their nearest neighbours. The second modification is that a neuron should only pulse once. To accomplish this the first term in Eq. 4.5 becomes

$$e^{\alpha_\Theta \delta_n} = 1.$$

These alterations warrant a modified iteration function in the *pcnn.py* module. Code 11.1 shows the new function **MazeIterate** which is part of the PCNN class. Line 3 creates a small kernel, `kern`, which is a 3×3 matrix will elements equal to 1. The correlation of this kernel with the previous pulse activity is now the neural connections. This is the neurons communicating with only the nearest neighbours.

The function also receives the maze which is a binary value matrix in which path elements are set to 1 and off path elements are set to 0. Line 6 multiplies the pulse field by the maze to prevent pulse activity outside of the maze.

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6_11, © Springer-Verlag Berlin Heidelberg 2013

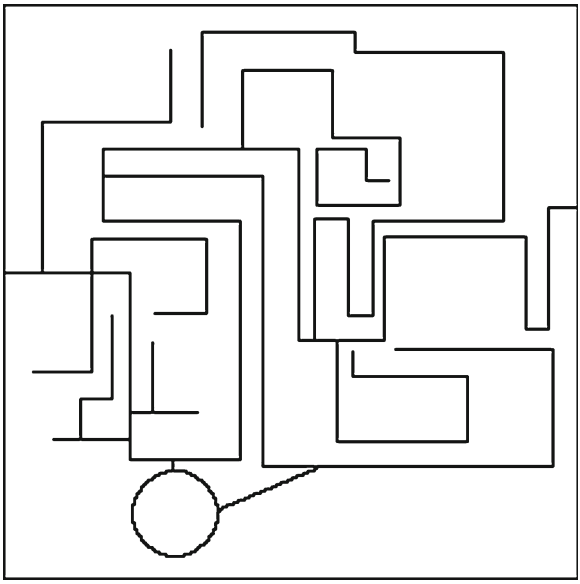


Fig. 11.1 The original maze which has a starting point on the left and a single ending point on the right. The goal is to find the shortest path from the start to the end

Code 11.1 The **MazeIterate** function.

```
# pcnn.py
from scipy.signal import cspline2d, correlate2d
    def MazeIterate (self, stim, maze):
        kern = np.ones( (3,3) )
        work = correlate2d( self.Y, kern, mode='same' )
        work *= maze
        self.F = self.f * self.F + stim + 8*work
        self.L = self.l * self.L + 8*work
        U = self.F * (1 + self.beta * self.L )
        self.Y = U > self.T
        self.T = self.t1 * self.T + self.t2 * self.Y
```

The **RunMaze** function shown in Code 11.2 is the driver for the process. The input is the matrix representing the maze with binary elements. Line 4 sets the coefficients for Eq. 4.5. Instead of just collecting pulse images this process will also accumulate the pulse images in a manner such that latter pulses are presented as larger values. This assists the viewer in seeing the path that was taken by the process. Line 7 creates a stimulus which is a matrix with all elements set to 0 except the starting point. Line 17 stops the process in this maze when the final pixel in the maze is pulsed. If a different maze is used then these lines will need to be altered.

Intermediate images of the progress are shown in Fig. 11.2a–g. As seen the autowave extends into all possible paths. However, at a junction, only the propagation that reached the junction first progresses forward. The final solution is shown in Fig. 11.2h. This is obtained by starting with the ending point in the maze and working backwards. Only the path that is monotonically decreasing is shown which is the shortest path between the starting point and ending point.

Maze running by the PCNN is not restricted to thin line mazes. Figure 11.3 shows a case in which the maze has thick paths and the autowave is allowed to expand in a more traditional manner.

Code 11.2 The **RunMaze** function.

```
# maze.py
def RunMaze( maze ):
    net = pcnn.PCNN( maze.shape )
    net.t1 = 1; net.t2 = 1e9
    v = maze[:,0].nonzero()[0]
    stim = np.zeros( maze.shape )
    stim[v,0] = 1
    Y = []
    accum = np.zeros( maze.shape )
    for i in range( 1000 ):
        net.MazeIterate( stim, maze )
        Y.append( net.Y )
        accum *= 0.99
        accum += net.Y
        stim = accum > 0
        print '.',
        if net.Y[90,-1] > 0:
            break
    return accum, Y

>>> acc,Y = RunMaze( maze )
```

11.2 Barcodes and Navigation

Contributed by Soonil Rughooputh, University of Mauritius. Considerable research has been conducted to improve safety and efficiency on navigational systems. For efficient control and increased safety, automatic recognition of navigational signs has become a major international issue. With the increasing use of semi-autonomous and autonomous systems (vehicles and robots), the design and integration of real-time operated navigational sign recognition systems have also gained in popularity. Systems that assist navigators or provide systems with computer vision-based navigational sign detection and recognition mechanisms have been devised. Manufacturers

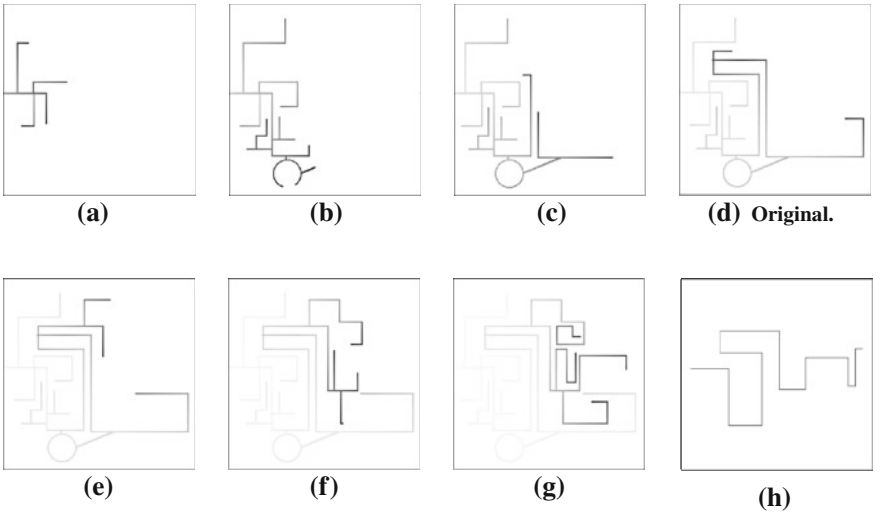


Fig. 11.2 Periodic progressions of the autowave and the final solution

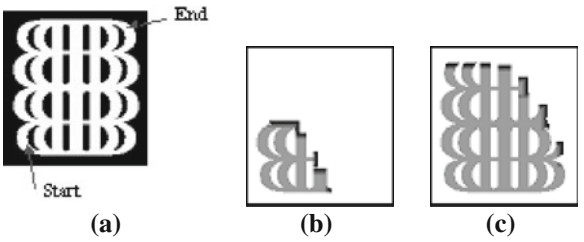


Fig. 11.3 **a** shows a thick maze and, **b** and **c** show autowaves traveling through a thick maze

in Europe, USA, and Japan and several universities even combined their efforts in this direction. The standards used in the design of navigational signs are typically according to size, shapes, and colour compositions. These signs form a very unique and easily visible set of objects within a scene. They always appear in a visible and fairly predictable region of an image. The only significant variables are the sizes of the signs in images (due to distance) and illumination of the scene (such as bright sunlight, overcast, fog, night). Two main characteristics of navigational signs are normally used for their detection in camera-acquired images, namely colour [9, 16, 25, 32, 38, 59, 77, 78, 81, 84 and 106] and shape [7, 12, 40, 49, 50, 82, 94, 104, 105]. Sign recognition is performed using sign contents such as pictograms and strings of characters. Normally, colour is employed in combination with shape for detection purposes first and then for sign recognition. Different types of image processing can be performed using colours. The three most widely used approaches

are neural network-based classifiers, colour indexing, and image segmentation based on colour.

Neural network-based classifiers involve the use of neural networks specifically trained to recognise patterns of colours. The use of multi-layer neural networks as experts for sign detection and recognition has been reported and applied a neural net as classifier to recognise signs within a region of interest [59]. Swain [106] has developed the 'colour indexing' technique that recognises signs by scanning portions of an image and then comparing the corresponding colour histograms with colour histograms of signs stored in a database. The technique has been improved by other researchers [32, 38]. Image segmentation based on colour uses algorithms to process an image and extract coloured objects from the background for further analysis. It remains the most widely used colour-based approach. Several authors have reported techniques for colour segmentation: including clustering in colour space, [108] region splitting, [25, 77, 78] colour edge detection, [16, 81] new parallel segmentation method based on 'region growing' or 'region collection' [84].

Shape-based sign detection relies largely on the significant achievements realised in the field of object recognition through research, such as techniques for scene analysis by robots, solid (3D) object recognition and part localisation in CAD databases. Almost all sign recognition systems process the colour information first to reduce the search for shape-based detection. Kehtarnavaz [93] extracted shapes from the image by performing edge detection and then applying the Hough transform to characterise the sides of the sign. Akatsuka [7] performed shape detection by template matching. de Saint-Blancard [94] used neural networks or expert systems as sign classifiers for a set of features consisting of perimeter (number of pixels), outside surrounding box, surfaces (inside/outside contour within surrounding box), centre of gravity, compactness ('aspect ratio' of box), polygonal approximation, Freeman code, histogram of Freeman code, and average grey level inside of box. Kellmeyer [50] trained a multi-layer neural net (with Back Propagation) to recognise diamond-shape warning signs in colour-segmented images. Piccioli [82] concentrated exclusively on geometrical reasoning for sign detection, detecting triangular shapes with Cannys algorithm and circles in a Hough-like manner. On the other hand, Priese [84] worked on a model-based approach where basic shapes of traffic sign components (circles, triangles, etc.) are predefined with 24-edge polygons describing their convex hulls. Segmented images are first scanned for 'objects', which are then encoded and assigned a probability (based on an edge-to-edge comparison between the object and the model) for shape classification. Besserer [12] used knowledge sources (a corner detector, a circle detector and a histogram-based analyser) to classify chain coded objects into shape classes. Other techniques, referred to as 'rigid model fitting' in [105], have also been used for shape-based sign detection. Stein [104], Lamdan [62] and Hong [40] use specific model representations and a common matching mechanism, geometric hashing, to index a model database.

The PCNN technique developed here does not necessitate any colour or shape processing. The automatic identification of signs is achieved simply through matching

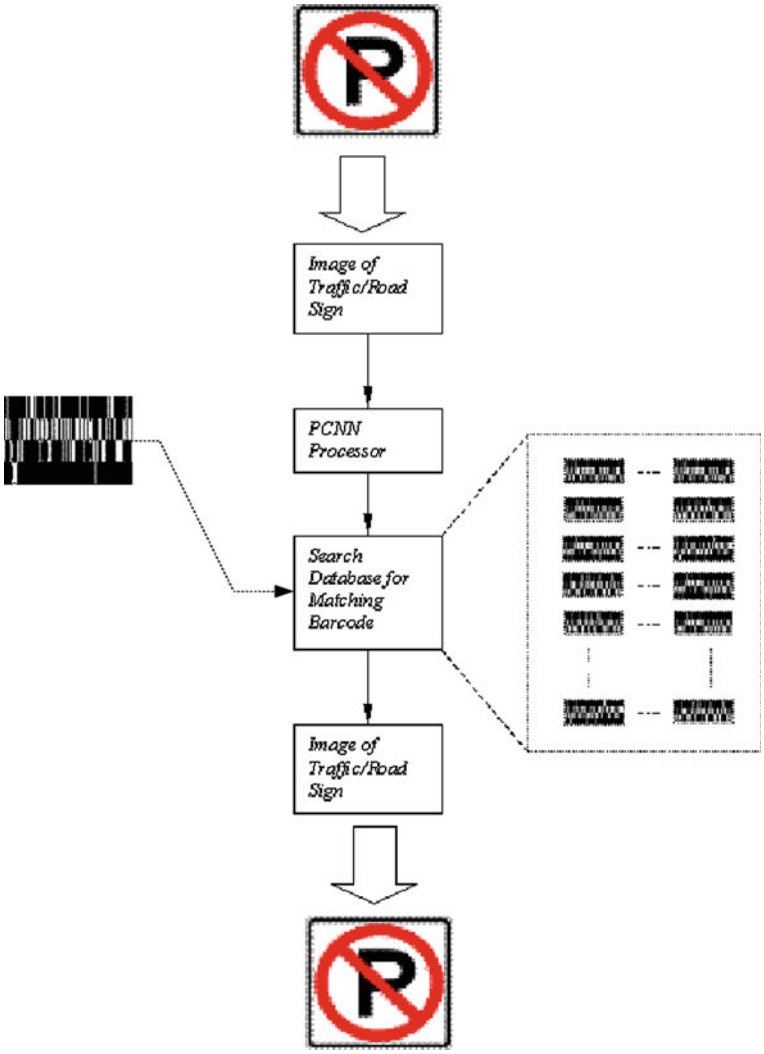


Fig. 11.4 Barcode generation from still road signs

of the barcodes of the images with the barcodes stored in the library (Fig. 11.4) [90]. An unknown sign can therefore be rapidly recognised using its unique barcode; a set of standard navigational signs is shown in Fig. 11.5 along with their respective barcodes.

Fig. 11.5 Typical road signs and their corresponding barcodes

11.3 Summary

The autowave propagation of the PCNN has some unique properties that can be used to perform logical operations. For example, the progression of a wave alters the activity of the locations it passes through. Therefore, it is easily known where a wave has transgressed. This, in a sense, is the nature of autowaves in that they do not pass through each other. Instead, colliding wave fronts annihilate the waves.

In applications such as running a maze this is a very useful feature. Waves can propagate through the system without replicating themselves. The first wave front to reach the destination has taken the shortest path, but the fact that the autowaves did not propagate more than once in any area allows the system to backtrack and determine which path was taken. Furthermore, this is not restricted to simple mazes but can be applied to wide path mazes as demonstrated.

Multiple versions of the maze running system can be used to solve the travelling salesman problem and the time of solution is dependent upon the size of the environment and linearly to the number of cities rather than to the total number of possible paths.

Appendix A
Image Converters

The *mgconvert.py* module contains functions to translate between matrices and images.

The functions are:

- **a2i**: converts a matrix to a grey scale image.
- **i2a**: converts a grey scale image to a matrix.
- **RGB2cube**: converts an RGB image to three integer matrices.
- **Cube2Image**: converts 3 matrices to an RGB image.
- **Cube2ImageF**: converts 3 matrices scaled between 0 and 255 to an RGB image.
- **RGB2YUV**: converts RGB images to the YUV colour model.
- **YUV2RGB**: converts YUV images into RGB images.

A.1 Transformation of Grey Scale Images and Matrices

The NumPy and Image modules offer functions to convert between matrices and images. Line 3 in Code A.1 loads an image and Line 4 converts this data to an array. The size of the array will be $V \times H \times N$ where V and H are the vertical and horizontal dimensions of the image and N is the number of channels. If the image is a grey scale image then $N = 1$ and if the image is colour then $N = 3$ or $N = 4$ depending on the presence of an alpha channel.

Code A.1 Functions for converting between images and arrays.

```
>>> import numpy as np
>>> import Image
>>> mg = Image.open( filename )
>>> mats = np.array( mg )
>>> mg2 = Image.fromarray( mats )
```

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6, © Springer-Verlag Berlin Heidelberg 2013

However, there are a few concerning the use of these functions that are addressed. The first is the conversion to and from a matrix to an image does not automatically scale. Thus, if the value of the matrix are between 0 and 1 then the image will have pixel values of 0 and 1. This will be an image in which all pixels are extremely dark.

The solution used here is to provide the function **a2i** which scales the matrix so that the lowest value becomes a black pixel and the highest matrix value becomes the white pixel. Code A.2 shows this function which receives a matrix and creates an automatically scaled grey scale image.

Code A.2 The **a2i** function.

```
# mgconvert.py
import Image

def a2i( indata ):
    mg = Image.new( 'L', indata.transpose().shape)
    mn = indata.min()
    a = indata - mn
    mx = a.max()
    a = a*256./mx
    mg.putdata(a.ravel())
    return mg
```

Conversion from an image to an array is possible in Line 4 in Code A.1 but this can create an image with a mode that is not conducive for saving in some formats. The **i2a** function shown in Code A.3 converts a grey scale image into an array of mode 'L'.

Code A.3 The **i2a** function.

```
# mgconvert.py
from numpy import array, fromstring, uint8
def i2a( mg ):
    H,V = mg.size
    d = fromstring( mg.tostring(), uint8)
    d = d.reshape( (V,H) )
    return d
```

Line 4 of Code A.1 can also convert a colour image into a three-dimensional data cube. The size of the cube is $V \times H \times N$ which is an arrangement that is not quite compatible with the scripts provided for the multi-channel PCNN and ICM. These routines expect the data to be in the form of $N \times V \times H$ which can be arranged by `matx = matx.transpose((2,0,1))`. However, to be congruent with the

previous functions the alternate function **RGB2cube** (Code A.4) can be used. This simply splits the image into the three channels and calls **i2a** for each.

Code A.4 The **RGB2cube** function.

```
# mgconvert.py
def RGB2cube( rgbmg ):
    r,g,b = rgbmg.split()
    r = i2a( r )
    b = i2a( b )
    g = i2a( g )
    return array( [r,g,b] )
```

The transformation from a set of three matrices with autoscaling is performed with **Cube2Image** as shown in Code A.5. For cases in which the user is responsible for the scaling of the three matrices the function **Image.fromarray** should be employed.

Code A.5 The **Cube2Image** function.

```
# mgconvert.py
def Cube2Image( r,g,b ):
    ri = a2i( r )
    gi = a2i( g )
    bi = a2i( b )
    mg = Image.merge( 'RGB', (ri,gi,bi) )
    return mg
```

A.2 Colour Conversion

The last two functions are used for conversions between RGB images and a colour format named YUV. There are several colour models that exist and most separate the intensity information from the colour information. The YUV model has three channels in which the Y channel displays the grey scale information and the U and V channels represent chroma differences.

The linear transformations between the two colour models is shown in Code A.6. The **RGB2YUV** function receives three matrices representing the RGB information and converts them to three matrices representing the YUV format by,

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.11 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & 0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{A.1}$$

The inverse transformation simply uses the inverse of the transformation matrix to reverse the process as in function **YUV2RGB**.

Code A.6 Functions for colour conversions.

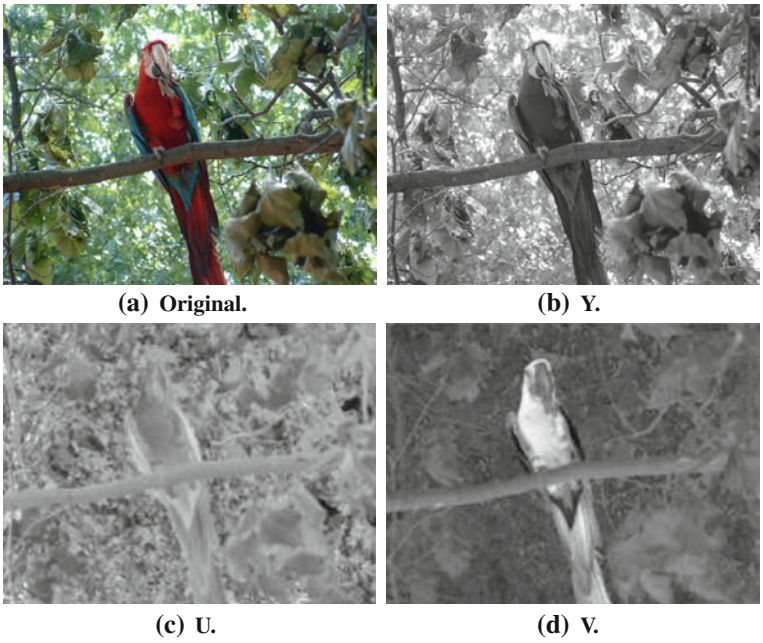
```
# mgconvert.py
def RGB2YUV(rr,gg,bb):
    y = 0.299* rr + 0.587*gg + 0.114*bb
    u = -0.147*rr - 0.289*gg + 0.436*bb
    v = 0.615*rr - 0.515*gg - 0.100*bb
    return y,u,v

def YUV2RGB( y, u, v ):
    r = y -3.94e-5*u +1.14 *v
    g = y -0.394*u -0.581*v
    b = y +2.032*u -4.81e-4*v
    return r,g,b
```

Usages are shown in Code A.7. Line 2 loads an image and Line 3 converts this image to grey scale and then converts it to a matrix. The matrix has integer values and usually it is wise to convert these to floats. Line 4 converts this matrix back to an image and shows it to the monitor. Line 5 converts an RGB image to three integer matrices corresponding to the three colour channels.

Code A.7 Functions from the *convert.py* module.

```
>>> import Image, mgconvert
>>> mg = Image.open( 'bird.png')
>>> r,g,b = mgconvert.RGB2cube( mg.convert('RGB') )
>>> y,u,v = mgconvert.RGB2YUV(r,g,b)
>>> mgconvert.a2i( y ).save ('ychannel.png')
>>> mgconvert.a2i( u ).save ('uchannel.png')
>>> mgconvert.a2i( v ).save ('vchannel.png')
```



(a) Original. **(b) Y.**

(c) U. **(d) V.**

Fig. A.1 An original colour image and the representations in YUV space. The Y channel represents the *grey* scale information and the U and V channels represent differences between two chroma channels

Appendix B
The Geometry Module

The *geometry* module provides a few generic routines useful in image processing.

B.1 Circle

The **Circle** function creates a matrix in which all values are initially 0. It sets element values to 1 if they are within a specified radius centred at a specified location. Code B.1 shows the function. It receives a tuple `size` which is the vertical and horizontal dimensions of the frame. The variable `loc` is the vertical and horizontal location of the center of the circle and the `rad` is the radius of the circle.

Code B.1 The **Circle** function.

```
# geometry.py
import numpy as np

def Circle( size, loc,rad):
    b1,b2 = np.indices( size )
    b1,b2 = b1-loc[0], b2-loc[1]
    mask = b1*b1 + b2*b2
    mask = mask <= rad*rad
    return mask.astype(int)
```

B.2 Plop

The **Plop** function shown in Code B.2 places data from one matrix into a larger one centered in the frame. The inputs are `data` which is the original data, and `vmax` and `hmax` which are the size of the new frame. The function finds the centre of mass of

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6, © Springer-Verlag Berlin Heidelberg 2013

the input and centres it into the output frame. Most of the function is dedicated to dealing with cases in which the boundaries of the frames or location of the centre of mass are not well behaved. The output is a matrix in which the input object is centred into the frame defined by `vmax`,`hmax`.

Code B.2 The **Plop** function.

```
# geometry.py
def Plop( data, vmax, hmax):
    # vmax, hmax are size of frame
    ans = zeros( (vmax,hmax), float )
    V,H = data.shape
    vctr, hctr = V/2, H/2  # center of frame
    vactr, hactr = vmax/2, hmax/2 # center of blob
    # compute the limits for the answ
    valo = vactr - vctr
    if valo<0: valo = 0
    vahi = vactr + vctr
    if vahi>=vmax: vahi = vmax
    halo = hactr - hctr
    if halo<0: halo = 0
    hahi = hactr + hctr
    if hahi>=hmax: hahi = hmax
    # compute limits of incoming
    vblo = vctr - vactr
    if vblo<=0: vblo = 0
    vbhi = vctr + vactr
    if vbhi>=V: vbhi= V
    hblo = hctr - hactr
    if hblo<=0: hblo = 0
    hbhi = hctr + hactr
    if hbhi>=H: hbhi = H
    if vahi-valo != vbhi-vblo:
        vbhi = vblo+vahi-valo
    if hahi-halo != hbhi-hblo:
        hbhi = hblo+hahi-halo
    ans[valo:vahi, halo:hahi] = data[vblo:vbhi, hblo:hbhi] + 0
    return ans
```

Appendix C
The Fractional Power Filter

The fractional power filter (FPF) [13] is a composite filter that can trade-off generalisation and discrimination. The composite nature of the filter allows for invariance to be built into the filter. This is useful when the exact presentation of the target can not be predicted or the filter needs to be invariant to alterations such as rotation, scale, skew, illumination, etc. The FPF also allows the same filter to be used to detect several different targets.

While the composite nature of the filter is very desirable for this application, it means some trade-offs are unavoidable. The binary objects on which the filter operates, can, for example, be confused with other objects. An increase in the discrimination ability of the filter cures this problem.

The FPF is a correlation filter built by creating a matrix $\hat{\mathbf{X}}$ whose columns are the vectorized Fourier transform of the training images. The filter $\hat{\mathbf{h}}$ is built by,

$$\hat{\mathbf{h}} = \mathbf{D}^{-1/2}\hat{\mathbf{Y}}\left[\hat{\mathbf{Y}}^T\hat{\mathbf{Y}}\right]^{-1}\mathbf{c}, \tag{C.1}$$

where

$$\hat{\mathbf{Y}} \equiv \mathbf{D}^{-1/2}\hat{\mathbf{X}}, \tag{C.2}$$

$$D_{ij} = \frac{\delta_{ij}}{N}\sum_k |\hat{X}_{ki}|^\alpha, \quad \alpha = [0, 2] \tag{C.3}$$

and vector $\mathbf{c}$ is the constraint vector.

A generalising FPF has $\alpha = 0$, which is the synthetic discriminant filter. The fully discriminatory filter has $\alpha = 2$, which is the minimum average correlation energy filter. A good review of this family of filters is found in [60]. Values of α between 0 and 2 trade-off generalisation and discrimination.

Code C.1 shows the **FPF** function from *fpf.py*. This function receives a matrix, `data`, in which the samples are stored as rows (contrary to the theory which expresses data in columns), a constraint vector, `c`, and the scalar fractional power term, `fp`. It returns a single vector `H` which is the filter.

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6, © Springer-Verlag Berlin Heidelberg 2013

Code C.1 The **FPF** function.

```
# fpf.py
import numpy as np
def FPF( data, c, fp ):
    (N,Dim )= data.shape
    D = ( np.pow( abs( data), fp )).sum(0)
    D = D / N
    ndx = (abs(D) < 0.001).nonzero()[0]
    D[ndx] = 0.001 * np.sign(D[ndx]+1e9)
    Y = (data / np.sqrt( D )).transpose()
    Yc = Y.conjugate().transpose()
    Q = np.dot( Yc, Y )
    if N == 1:
        Q = 1./Q
    else:
        Q = np.linalg.inv( Q )
    Rc = np.dot( Q,c)
    H = np.dot( Y, Rc ) / np.sqrt(D)
    return H
```

When the fractional term is non-zero it is required that the data be in frequency space. Furthermore, the script only allows data as vectors whereas most of the applications in the chapters are with image data. Code C.2 shows a sample case in which several matrices are stored as in a list `M`. Line 5 allocates a matrix `X` which will store each input image as a long row vector. Line 7 computes the Fourier transform of each image and uses the **ravel** command to convert the 2D matrix to a long 1D vector. Line 9 calls the **fpf.FPF** function with a fractional power term of 0.8, and it returns a long vector which is still in Fourier space. The last two lines convert the filter back to 2D space and then converts the filter back to image space.

Code C.2 Computing the FPF for multiple matrices

```
>>> from scipy.fftpack import fft2, ifft2
>>> import fpf
>>> N = len( M )
>>> V,H = M.shape
>>> X = np.zeros( (N,V*H), complex )
>>> for i in range( N ):
>>>     X[i] = (fft2(M[i])).ravel()
>>> cst = np.ones( N )
>>> H = fpf.FPF( X, cst, 0.8 )
>>> H = H.reshape((V,H) )
>>> H = ifft2( H )
```

Appendix D
Correlation

A correlation of two signals $f(x)$ and $g(x)$ is the dot product of all relative shifts between the two signals. Formally, the correlation is,

$$c(\omega) = \int_{-\infty}^{\infty} f(x) g^{\dagger}(x - \omega) dx, \tag{D.1}$$

where $g^{\dagger}$ represents the complex conjugate. The integral performs the inner product and the argument of g is performing the relative shifts.

The computations for the correlation could be performed by using (D.1) but a much faster implementation is to perform the computations in Fourier space. The Fourier transform of a signal is defined as,

$$\mathcal{F}\{c(\omega)\} = \int_{-\infty}^{\infty} c(\omega) \exp[-i\omega\alpha] \, d\alpha. \tag{D.2}$$

Consider the Fourier transform of Eq. (D.1),

$$\mathcal{F}\{c(\omega)\} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x) g^{\dagger}(x - \omega) \exp[-i\omega\alpha] \; dx \; d\alpha. \tag{D.3}$$

Set $z = x - \alpha$ and thus also $dx = -d\alpha$ and $\alpha = x - z$ to yield,

$$\mathcal{F}\{c(\omega)\} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x) g^{\dagger}(z) \exp[-i\omega(x - z)] \, dx \; dz. \tag{D.4}$$

This can be rewritten as,

$$\mathcal{F}\{c(\omega)\} = -\int_{-\infty}^{\infty} f(x) \exp[-i\omega x] \, dx \int_{-\infty}^{\infty} g^{\dagger}(x) \exp[-i\omega z] \, dz. \tag{D.5}$$

Each of the integrals are in fact Fourier transforms,

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6, © Springer-Verlag Berlin Heidelberg 2013

$$\mathcal{F}\{c(\omega)\} = F(\omega)G^{\dagger}(\omega). \tag{D.6}$$

The Fourier transform of the correlation is actually the multiplication of the Fourier transforms of the two signals. For a signal of length N the digital version of Eq. (D.1) would require two nested loops and so the number of operations (multiple-additions) is N^2. Recall that the FFT requires $N \log_2 N$ operations and thus the number of operations required to compute the correlation using Fourier space is $N + 3N \log_2 N$ which is a tremendous savings for large N. The savings is amplified for signals in two-dimensions.

The digital version of a correlation can be computed in the same manner by multiplying the Fourier transforms of the data. However, the digital Fourier transform places the high frequencies in the middle of the matrix which is often the opposite of the desired usage. Thus, the data requires a swapping function which divides that matrix into four equally size quadrants and swaps the position of the first quadrant with the fourth quadrant and swaps the second quadrant with the third quadrant.

A Python implementation of the quadrant swap is shown in Code D.1 with the **Swap** function. This function receives a matrix, performs the swap, and returns a new matrix with the swapped elements. The only caveat is that the dimensions of the matrix must be even numbers.

Code D.1 The **Swap** function.

```
# correlate.py
import numpy as np

def Swap( A ):
    (v,h) = A.shape
    ans = np.zeros(A.shape,A.dtype)
    ans[0:v/2,0:h/2] = A[v/2:v,h/2:h]
    ans[0:v/2,h/2:h] = A[v/2:v,0:h/2]
    ans[v/2:v,h/2:h] = A[0:v/2,0:h/2]
    ans[v/2:v,0:h/2] = A[0:v/2,h/2:h]
    return ans
```

The correlation is performed by the **Correlate** function shown in Code D.2. The function has an option which is controlled by the switch `sw`. In some cases the data may already be in Fourier space and so the correlation is not required to perform the conversions. For these cases the user should set `sw = 1`. The default setting is `sw = 0` and in this case Lines 4 and 5 convert the data to Fourier space, Line 6 performs the multiplications and Line 7 converts the data back to the original space. Line 8 performs the necessary swap.

In cases in which shapes are to be matched using a correlation a positive result should produce a spike in the correlation output. However, in dealing with pulse images it is possible and sometimes common that the correlation surface have values that equal or exceed the height of a spike at different locations. Thus, detection of a

Code D.2 The **Correlate** function.

```
# correlate.py
def Correlate( A, B, sw=0 ):
    if sw==0:
        a = np.fft.fft2( A)
        b = np.fft.fft2(B)
        c = a * b.conjugate()
        C = np.fft.ifft2( c )
        C = Swap(C)
    if sw==1:
        c = A * B.conjugate()
        C = np.fft.ifft2( c )
        C = Swap(C)
    return C
```

spike is more difficult than just performing a simple threshold. Horner [41] proposed a peak to correlation energy measure which divides the value of each pixel by the energy of the surrounding pixels. A region that has a large area of high correlation values will be suppressed by this method, whereas a pixel that represents a peak will be enhanced. Thus, the PCE (peak to correlation energy) is a simple tool to enhance peaks. The Python implementation of the PCE is shown in Code D.3. In this case the local energy is replaced by local averaging using the **csplined2d** function.

Code D.3 The **PCE** function.

```
# correlate.py
from scipy.signal import cspline2d

def PCE( corr, rad=12 ):
    acorr = abs( corr )**2
    temp = cspline2d( acorr, rad )
    mask = temp > 0
    nrg = np.sqrt( mask*temp )
    pce = acorr/(nrg+0.01)
    return pce
```

Appendix E
The FAAM

The FAAM (fast analog associative memory) [51] is a simple greedy system that grows in complexity in order to meet the demands of the training data. The logic of the system is shown in a series of images in Fig. E.1. The FAAM sequentially considers data from two different classifications (shown as circles and pluses). Figure E.1a after receiving two training vectors of opposing class. There must be a boundary that separates the two vectors but without any other knowledge the shape and exact location is not known. Therefore, a boundary that bisects the segment connecting the two data points is created.

In Fig. E.1b a third data point is considered and in this case it is on the correct side of the boundary. Thus, training is not required. This is one of the parts of the algorithm where the FAAM differs from a traditional neural network. A neural network would train to optimize the location of the boundary whereas the FAAM's logic requires no further calculations.

A fourth data point is considered in Fig. E.1c which is a plus class and it is not on the correct side of the boundary. So, a second boundary is created. Another deviation between the FAAM and a traditional neural network is that the FAAM creates boundaries as needed whereas in a neural network the user must define the number of hidden neurons before training commences often without sufficient knowledge as to how many hidden neurons are optimal.

In this case the first boundary is no longer necessary and so it is eliminated as shown in Fig. E.1d. A neural network optimizes the decisions by moving a specified number of boundaries. The FAAM optimizes by creating boundaries as needed and eliminating those that become redundant. The FAAM continues this process until all of the data is considered.

Another feature of the FAAM is that it provides information as to how well the network is learning a problem instead of memorising the data. As new data points are considered the network adds boundaries if needed. However, if the new data points are classified correctly by the current architecture then no training is performed. The metric for indication of the network's learning is the number of boundaries. As shown in Fig. 7.17a, b when the number of boundaries becomes asymptotic then the system is learning the problem. Thus, it is not necessarily required to separate the data into

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6, © Springer-Verlag Berlin Heidelberg 2013

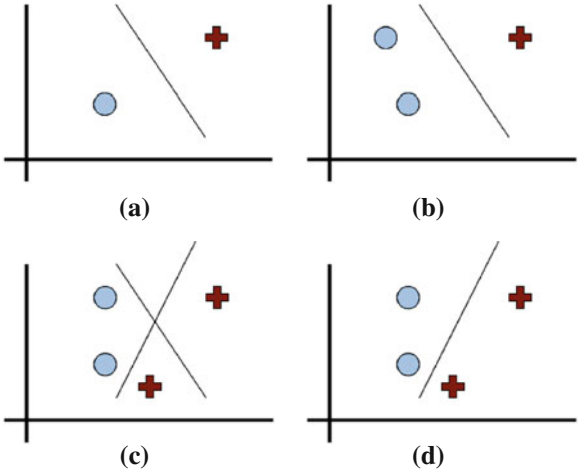


Fig. E.1 Growing boundaries to separate the data

training and testing sets since performance information is readily available during training.

The FAAM is the only program that is not included in the written text due to its size. However, it is available through the accompanying website (see the Preface to the Third Edition). Driving the FAAM is relatively easy. First the user needs to initialise the network. The second step is to add training data pairs and the third is to train. It is possible to add just a few trainers and train in a loop such that the progress of the system can be monitored. Code E.1 shows the commands. Line 2 initialises the network with three arguments. These are the maximum number of trainers to be used, the maximum number of decision surfaces, and the length of an input vector. Given a set of training data with vectors in a list `x` and classifications in a list `y`, Line 3 shows how one of the trainers is added to the network. This line would be repeated for all trainers with the proper index. Line 5 trains the network and Line 6 shows a typical recall operation with a query vector. There are two caveats to this system. The first is that there must be at least one vector from each class before training and the second is that there can not be two trainers with exactly the same input and two different output classes.

Code E.1 Running the FAAM.

```
>>> import faam
>>> net = faam.FAAM( mxtrainers, maxr, dim )
>>> net.SetTrainer( x[0], y[0] )
...
>>> net.Train( )
>>> y_out = net.Recall( queryx )
```

The recall can provide one of three answers. The first two are the possible choices for the classification (0 and 1) and the third is an undecided answer (-1). The third occurs when there are regions that are isolate from all other regions but do not have a training vector embedded therein.

Appendix F
Principal Component Analysis

Consider the data in Fig. F.1a which shows four data points in $\mathcal{R}^2$ space. Each point is described by (x, y) coordinates. However, there is a structure to the data and if the coordinate system is rotated and shifted as shown in Fig. F.1b the data need only be described by a single coordinate x. Principal component analysis (PCA) is a process that finds the optimal coordinate transformation that reduces the covariance amongst the data points. In Fig. F.1a the x and y coordinates are highly correlated. Simply, given any value of x it is possible to predict the value of y and vice versa. In Fig. F.1b, however, the covariance is eliminated. While it is possible to predict future values of y (all points are $y = 0$) it is not possible to relate this prediction to a given value of x. In other words, given a value of y it is not possible to predict a value of x.

Thus, the PCA begins with the computation of the covariance matrix,

$$\Sigma[i, j] = \mathbf{x}_i \cdot \mathbf{x}_j. \tag{F.1}$$

The diagonal elements are related to the variances of the individual elements in the data and the off-diagonal elements are the covariances. Minimization of the covariances is performed through the use of eigenvectors and eigenvalues. The eigenvectors are a set of orthonormal vectors that describe the new coordinate system and the eigenvalues indicate the importance of each coordinate. The PCA achieves a reduction

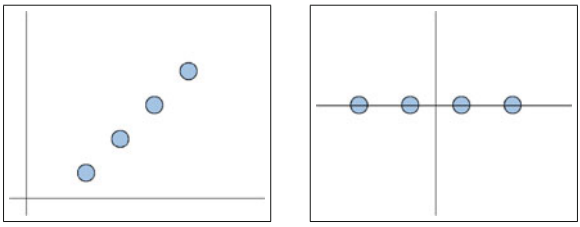


Fig. F.1 Rotating the coordinate system

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6, © Springer-Verlag Berlin Heidelberg 2013

in dimensionality by using only those eigenvectors that are associated with large eigenvalues.

Code F.1 displays **PCA** function which receives the data matrix as `mat` and the number of desired dimensions in the returned data. The function also maps the data into the new space and returns the new data points as `cffs` and the eigenvectors `vecs`. Line 5 removes the bias from the data and Line 6 computes the covariance using the **numpy.cov** function. Line 7 calls the **numpy.linalg.eig** function to compute the eigenvalues and eigenvectors. The rest of the function is dedicated to mapping the data into the new space.

Code F.1 The **PCA** function.

```
# pca.py
from numpy import array, cov, linalg, logical_and, zeros

def PCA( mat, D=2 ):
    a = mat - mat.mean(0)
    cv = cov( a.transpose() )
    evl, evc = linalg.eig( cv )
    V,H = mat.shape
    cffs = zeros( (V,D) )
    ag = abs(evl).argsort( )
    ag = ag[::-1]
    me = ag[:D]
    for i in range( V ):
        k = 0
        for j in me:
            cffs[i,k] = (mat[i] * evc[:,j]).sum()
            k += 1
    vecs = evc[:,me].transpose()
    return cffs, vecs
```

References

1. IBM: IBM microelectronics ZISC zero instruction set computer: Preliminary information. In: Supplement to Proceedings of World Congress on Neural Networks, International Neural Network Society. INNS Press, San Diego (1994).
2. http://c2.com/cgi/wiki?pythonide (2008)
3. http://www.scipy.org (2008)
4. http://www.youtube.com/watch?v=oSQJP40PcGI. Accessed August 2012
5. http://cse.msu.edu/pub/pnp/vincenUDB_FDC-MSU/DB-FDC-MSU_327_news ahoo.% zip Accessed 23 July 2008
6. Acharya, M., Kinser, J., Nathan, S., Albano, M.C., Schlegel, L.: An image analysis method for quantification of idiopathic pulmonary fibrosis. In: Proceedings of the AIPR (2012). In press.
7. Akatsuka, H., Imai, S.: Road signposts recognition system. In: Proceedings of the SAE vehicle highway infrastructure: safety compatibility, pp. 189–196 (1987).
8. Akay, M.: Wavelet applications in medicine. IEEE Spectr. **34**, 50–56 (1997)
9. Arens, J., Saremi, A., Simmons, C.: Color recognition of retroreflective traffic signs under various lighting conditions. Public Roads **55**, 1–7 (1991)
10. Balkarey, Y.I., Evtikhov, M.G., Elinson, M.I.: Autowave media and neural networks. Proc. SPIE **1621**, 238–249 (1991)
11. van de Bergh, S.: Luminosity classification of galaxies in the revised shapley-ames catalog. Publ. Astron. Soc. **94**, 745 (1982)
12. Besserer, B., Estable, S., Ulmer, B.: Multiple knowledge sources and evidential reasoning for shape recognition. In: Proceedings of IEEE 4th Conference on Computer Vision, pp. 624–631 (1993).
13. Brasher, J.D., Kinser, J.M.: Fractional-power synthetic discriminant functions. Pattern Recognit. **27**(4), 577–585 (1994)
14. Bryson, A.E., Ho, Y.C.: Applied Optimal Control. Blaisdell, New York (1969)
15. Buta, R., Mitra, S., de Vaucouleurs, G., Corwin, H.G.: Mean morphological types of bright galaxies. Astron. J. **107**, 118 (1994)
16. Carron, T., Lambert, P.: Color edge detector using jointly hue saturation and intensity. IEEE Int. Conf. Image Process. **3**, 977–981 (1994)
17. Chen, Y.Q.: Novel techniques for image texture classification. Ph.D. thesis, Department of Electronics and Computer Science, University of Southampton (1996).
18. Chen, Y.Q., Nixon, M.S., Thomas, D.W.: Statistical geometrical features for texture classification. Pattern Recognit. **28**(4), 537–552 (1995)

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6, © Springer-Verlag Berlin Heidelberg 2013

19. Cheng, L.J., Chao, T.H., Dowdy, M., LaBaw, C., Mahoney, C., Reyes, G., Bergman, K.: Multispectral imaging systems using acousto-optic tunable filter. In: Infrared and Millimeter Wave Engineering. Proc. SPIE **1874**, 224 (1993)
20. Cheng, L.J., Chao, T.H., Reyes, G.: Acousto-optic tunable filter multispectral imaging system. In: AIAA Space Programs and Technologies Conference, pp. 92–1439 (1992).
21. C1MK Data Sheet. CogniMem Technologies Inc., Folsom, CA 95630 USA. <!– Missing/Wrong Year –>
22. Darrell, T., Essa, I., Pentland, A.: Task-specific gesture analysis in real-time using interpolated views. IEEE Trans. Pattern Anal. Mach. Intell. **18**(12), 1236–1242 (1996)
23. Davis, J., Shah, M.: Recognizing hand gestures. In: ECCV94, pp. 331–340 (1994).
24. Duan, L., Miao, J., Liu, C., Lu, Y., Qiao, Y., Zou, B.: A PCNN based approach to image segmentation using size-adaptive texture features. In: Wang, R., Shen, E., Gu, F. (eds.) Advances in Cognitive Neurodynamics ICCN 2007, pp. 933–937. Springer, Netherlands (2008). https://dx.doi.org/10.1007/978-1-4020-8387-7_162,10.1007/978-1-4020-8387-7_162
25. Dubuisson, M.P., Jain, A.: Object contour extraction using color and motion. In: IEEE International Conference on Image Processing, pp. 471–476 (1994).
26. Eckhorn, R., Reitboeck, H.J., Arndt, M., Dicke, P.: Feature linking via synchronization among distributed assemblies: simulations of results from cat visual cortex. Neural Comput. **2**, 293–307 (1990)
27. Eggen, O.J., Lynden-Bell, D., Sandage, A.R.: Evidence from the motion of old stars that the galaxy collapsed. Astrophys. J. **136**, 748 (1962)
28. Fels, S.S., Hinton, G.E.: Glove-talk: a neural network interface between a data-glove and a speech synthesizer. IEEE Trans. Neural Netw. **4**, 2–8 (1993)
29. Fitzhugh, R.: Impulses and physiological states in theoretic models of nerve membrane. Biophys. J. **1**, 445–466 (1961)
30. Freeman, W.T., Weissman, C.D.: Television control by hand gestures. In: International Workshop on Automatic Face and Gesture Recognition, pp. 179–183 (1995).
31. Frei, Z., Guhathakurta, P., Gunn, J.E., Tyson, J.A.: A catalog of digital images of 113 nearby galaxies. Astron. J. **111**, 174–181 (1996)
32. Funt, B.V., Finlayson, G.D.: Color constant color indexing. IEEE Trans. Pattern Anal. Mach. Intell. **17**(5), 522–529 (1955)
33. Gernster, W.: Time structure of the activity in neural network models. Phys. Rev. E **51**(1), 738–758 (1995)
34. Grayson, M.A.: The heat equation shrinks embedded plane curves to round points. J. Differ. Geom. **26**, 285–314 (1987)
35. Haralick, R.M., Shanmugam, K., Dinstein, I.: Textural features for image classification. IEEE Trans. Syst. Man Cybern. **3**, 610–621 (1973)
36. Hawker, L.: The introduction of economic assessment to pavement maintenance management decisions on the United Kingdom using private finance. In: 13th IRF World Meeting, Toronto (1997).
37. He, D.C., Wang, L.: Texture features based on texture spectrum. Pattern Recognit. **25**(3), 391–399 (1991)
38. Healey, G., Slater, D.: Global color constancy: recognition of objects by use of illumination-invariant properties of color distributions. J. Opt. Soc. Am. A **11**(11), 3003–3010 (1994)
39. Hodgkin, A.L., Huxley, A.F.: A quantitative description of membrane current and its application to conduction and excitation in nerve. J. Physiol. **117**, 500–544 (1952)
40. Hong, J., Wolfson, H.: An improved model-based matching method using footprints. In: Proceedings of 9th International Conference on Pattern Recognition, pp. 72–78, IEEE (1988).
41. Horner, J.L.: Metrics for assessing pattern recognition. Appl. Opt. **31**(2), 165–166 (1992)
42. Huang, T.S., Pavlovic, V.I.: Hand modelling, analysis, and synthesis. International Workshop on Automatic Face and Gesture Recognition, Zurich, In (1995)
43. Intel 801NX ETANN Data Sheet. Intel, Corp., Santa Clara, CA (1993).
44. Isard, M., Blake, A.: Condensation: conditional density propagation for visual tracking. Int. J. Comput. Vis. **29**(1), 5–28 (1998)

45. Johansson, G.: Visual perception of biological motion and a model for its analysis. Percept. Psychophys. **73**(2), 201–211 (1973)
46. Johnson, J.L.: The signature of images. IEEE International Conference on Neural Network, In (1994)
47. Johnson, J.L., Padgett, M.L.: PCNN models and applications. IEEE Trans. Neural Netw. **10**(3), 480–498 (1999)
48. Johnson, J.L., Padgett, M.L., Friday, W.A.: Multiscale image factorization. In: Proceedings of the International Conference on Neural Networks (ICNN97), pp. 1465–1468 (1997). Invited Paper.
49. Kehtarnavaz, N., Griswold, N.C., Kang, D.S.: Stop-sign recognition based on color-shape processing. Mach. Vis. Appl. **6**(4), 206–208 (1993)
50. Kellmeyer, D., Zwahlen, H.: Detection of highway warning signs in natural video images using color image processing and neural networks. Proc. IEEE Int. Conf. Neural Netw. **7**, 4226–4231 (1994)
51. Kinser, J.M.: Fast analog associative memory. Proc. SPIE **2568**, 290–293 (1995)
52. Kinser, J.M.: Object isolation. Opt. Memories. Neural Netw. **5**(3), 137–145 (1996)
53. Kinser, J.M.: Pulse-coupled image fusion. Opt. Eng. **36**(3), 737–742 (1997)
54. Kinser, J.M.: Hardware: Basic requirements for implementation. Proc. SPIE **3728**, 222–229 (1998)
55. Kinser, J.M.: Image signatures: classification and ontology. In: Proceedings of the 4th IASTED International Conference on Computer Graphics and, Imaging (2001).
56. Kinser, J.M., Lindblad, T.: Detection of microcalcifications by cortical simulation. In: Bulsari, A.B., Kalli, S., (eds.) Neural Networks in Engineering Systems, pp. 203–206 (1997).
57. Kinser, J.M., Nguyen, C.: Image object signatures from centripetal autowaves. Pattern Recognit. Lett. **21**(3), 221–225 (2000)
58. Kinser, J.M., Wyman, C.L., Kerstiens, B.: Spiral image fusion: a 30 parallel channel case. Opt. Eng. **37**(2), 492–498 (1998)
59. Krumbiegel, D., Kraiss, K.F., Schreiber, S.: A connectionist traffic sign recognition system for onboard driver information. In: 5th IFAC/IFIP/IFORS/IEA Symposium on Analysis, Design and Evaluation of Man-Machine Systems, pp. 201–206 (1993).
60. Kumar, B.V.K.V.: Tutorial survey of composite filter designs for optical correlators. Appl. Opt. **31**(23), 4773–4801 (1992)
61. Labbi, A., Milanese, R., Bosch, H.: A network of fitzhugh-nagumo oscillators for object segmentation. In: Proceedings of International Symposium on Nonlinear Theory and Applications (NOLTA '97), pp. 581–584 (1997).
62. Lamdan, Y., Wolfson, H.: Geometric hashing: a general and efficient model-based recognition scheme. In: Proceedings of 2nd International Conference on Computer Vision, pp. 238–249, IEEE (1988).
63. Laws, K.I.: Textured image segmentation. Ph.D. thesis, Department of Electrical Engineering, University of Southern California (1980).
64. Lee, J., Kunii, T.: Model-based analysis of hand posture. IEEE Comput. Graphics Appl. **15**(5), 77–86 (1995). doi:10.1109/38.403831
65. Lindblad, T., Hultberg, S., Lindsey, C., Shelton, R.: Performance of a neural network for recognizing AC current demand signatures in the space shuttle telemetry data. Proc. Am. Control Conf. **2**, 1373–1377 (1995). doi:10.1109/ACC.1995.520975
66. Lindsey, C.S., Lindblad, T., Sekniaidze, G., Minersskjöld, M., Skékely, G., Eide, Å.: Experience with the IBM ZISC neural network chip. Int. J. Mod. Phys. C **6**(4), 579–584 (1995)
67. Littmann, E., Drees, A., Ritter, H.: Visual gesture recognition by a modular neural system. In: International Conference on Art in Neural Networks, Bochum, pp. 317–322 (1996).
68. Lu, Y., Miao, J., Duan, L., Qiao, Y., Jia, R.: A new approach to image segmentation based on simplified region growing PCNN. Appl. Math. Comput. **205**, 807–814 (2008)
69. Malladi, R., Sethian, J.A.: Level set methods for curvature flow, image enhancement, and shape recovery in medical images. In: Proceedings of Conference on Visualization and Mathematics, pp. 329–345. Springer, Heidelberg (1995).

70. McClurken, J.W., Zarbock, J.A., Optican, L.M.: Temporal codes for colors, patterns and memories. Cereb. Cortex **10**, 443–467 (1994)
71. McEniry, C., Johnson, J.L.: Methods for image segmentation using a pulse coupled neural network. Neural Netw. World **2**(97), 177–189 (1997)
72. Mirollo, R.E., Strogatz, S.H.: Synchronization of pulse-coupled biological oscillators. SIAM J. Appl. Math. **50**(6), 1645–1662 (1990)
73. Moody, J., Darken, C.: Fast learning in networks of locally tuned processing units. Neural Comput. **1**, 281–294 (1989)
74. Mornev, O.A.: Elements of the optics of autowaves. In: Krirsky, V.I., (ed.) Self-Organization Autowaves and Structures far from Equilibrium, pp. 111–118. Springer (1984).
75. Nagumo, J., Arimoto, S., Yoshizawa, S.: An active pulse transmission line stimulating nerve axon. Proc. IRE **50**, 2061–2070 (1962)
76. Neibur, E., Wörgötter, F.: Circular inhibition: A new concept in long-range interaction in the mammalian visual cortex. Proc. IJCNN **2**, 367–372 (1990)
77. Ohlander, R., Price, K., Reddy, D.: Picture segmentation using a recursive region splitting method. Comput. Graphics Image Process. **8**, 313–333 (1978)
78. Ohta, Y., Kanade, T., Sakai, T.: Color information for region segmentation. Comput. Graphics Image Process. **13**, 224–241 (1980)
79. Padgett, M.L., Johnson, J.L.: Pulse-coupled neural networks (PCNN) and wavelets: Biosensor applications. In: Proceedings of the International Conference on Neural Networks (ICNN97), pp. 2507–2512 (1997). Invited Paper.
80. Parodi, O., Combe, P., Ducom, J.C.: Temporal encoding in vision: Coding by spiking arrival times leads to oscillations in the case of moving targets. Neurocomputing **4**, 93–102 (1992)
81. Perez, F., Koch, C.: Toward color image segmentation in analog VLSI: Algorithm and hardware. Int. J. Comput. Vis. **12**(1), 17–42 (1994)
82. Piccioli, G., Michelli, E.D., Parodi, P., Campani, M.: Robust road sign detection and recognition from image sequence. In: Proceedings of Intelligent Vehicles '94, pp. 278–283 (1994).
83. Pratt, W.K.: Digital Image Processing, 3rd edn. Wiley-Interscience, New York (2001)
84. Priese, L., Rehrmann, V.: On hierarchical color segmentation and applications. In: Proceedings of CVPR, pp. 633–634 (1993).
85. Pynn, J., Wright, A., Lodge, R.: Automatic identification of road cracks in road surfaces. In: 7th International Conference on Image Processing and its Applications, vol. 2, pp. 671–675. Manchester (UK) (1999).
86. Raya, T.H., Bettaiah, V., Ranganath, H.S.: Adaptive pulse coupled neural network parameters for image segmentation. World Acad. Sci. Eng. Technol. **73**, 1046–1052 (2011)
87. Roberts, M.S., Haynes, M.P.: Physical parameters along the hubble sequence. Annu. Rev. Astron. Astrophys. **32**(1), 115–152 (1994)
88. Rughooputh, H.C.S., Bootun, H., Rughooputh, S.D.D.V.: Intelligent hand gesture recognition for human computer interaction and robotics. In: Proceedings of RESQUA2000: The First Regional Symposium on Quality and Automation: Quality and Automation Systems for Advanced Organizations in the Information Age, pp. 346–352, IEE (2000).
89. Rughooputh, H.C.S., Rughooputh, S.D.D.V., Kinser, J.: Automatic inspection of road surfaces. In: Kenneth, J., Tobin, W. (ed.) Machine Vision Applications in Industrial Inspection VIII. Proc. SPIE **3966**, 349–356 (2000)
90. Rughooputh, S.D.D.V., Bootun, H., Rughooputh, H.C.S.: Intelligent traffic and road sign recognition for automated vehicles. In: Proceedings of RESQUA2000: The First Regional Symposium on Quality and Automation: Quality and Automation Systems for Advanced Organizations in the Information Age, pp. 231–237, IEE (2000).
91. Rughooputh, S.D.D.V., Somanah, R., Rughooputh, H.C.S.: Classification of optical galaxies using a PCNN. In: Nasrabadi, N. (ed.) Applications of Artificial Neural Networks in Image Processing V. Proc. SPIE **3962**(15), 138–147 (2000)
92. Rybak, I.A., Shevtsova, N.A., Podladchikova, L.N., Golovan, A.V.: A visual cortex domain model and its use for visual information processing. Neural Netw. **4**, 3–13 (1991)

93. Rybak, I.A., Shevtsova, N.A., Sandler, V.A.: The model of a neural network visual processor. Neurocomputing **4**, 93–102 (1992)
94. de Saint Blancard, M.: Road Sign Recognition: A Study of Vision-Based Decision Making for Road Environment Recognition. Springer, New York (1992).
95. Sandage, A., Freeman, K.C., Stokes, N.R.: The intrinsic flattening of e, so, and spiral galaxies as related to galaxy formation and evolution. Astrophys. J. **160**, 831 (1970)
96. Searle, L., Zinn, R.: Compositions of halo clusters and the formation of the galactic halo. Astrophys. J. **225**, 357–379 (1978)
97. Singh, M., Singh, S.: Spatial texture analysis: A comparative study. In: Proceedings of 15th International Conference on Pattern Recognition (ICPR'02), Quebec (2002).
98. Singh, S., Singh, M.: Texture analysis experiments with meastex and vistex benchmarks. In: Singh, S., Murshed, N., Kropatsch, W. (eds.) Proceedings of International Conference on Advances in Pattern Recognition. Lecture Notes in Computer Science, vol. 2013, pp. 417–424. Springer, Rio (2001).
99. Siskind, J.M., Morris, Q.: A maximum-likelihood approach to visual event classification. In: Proceedings of the Fourth European Conference on Computer Vision, pp. 347–360 (1996).
100. Smith, S.M., Brady, J.M.: Susan: A new approach to low level image processing. Int. J. Comput. Vis. **23**(1), 45–78 (1997). doi:10.1023/A:1007963824710
101. Somanah, R., Rughooputh, S.D.D.V., Rughooputh, H.C.S.: Identification and classification of galaxies using a biologically-inspired neural network. Astrophys. Space Sci. **282**, 161–169 (2002)
102. Srinivasan, R., Kinser, J.: A foveating-fuzzy scoring target recognition system. Pattern Recognit. **31**(8), 1149–1158 (1998)
103. Srinivasan, R., Kinser, J., Schamschula, M., Shamir, J., Caulfield, H.J.: Optical syntactic pattern recognition using fuzzy scoring. Opt. Lett. **21**(11), 815–817 (1996)
104. Stein, F., Medioni, G.: Structural indexing: efficient 2-D object recognition. IEEE Trans. Pattern Anal. Mach. Intell. **14**(12), 1198–1204 (1992)
105. Suetens, P., Fua, P., Hanson, A.J.: Computational strategies for object recognition. ACM Comput. Surv. **24**(1), 5–61 (1992)
106. Swain, M.J., Ballard, D.: Color indexing. Int. J. Comput. Vis. **7**(1), 111–132 (1991)
107. Tomikawa, T.: A study of road crack detection by the meta-generic algorithm. In: Proceedings of IEEE African 99 International Conference, pp. 543–548, Cape Town (1999).
108. Tominaga, S.: A color classification method for color images using a uniform color space. In: IEEE CVPR, pp. 803–807 (1990).
109. Tuceryan, M., Jain, A.K.: Texture analysis. In: Chen, C.H., Pau, L.F. (eds.) The Handbook of Patten Recognition and Computer Vision, 2nd edn, pp. 207–248. World Scientific Publishing Company, Singapore (1998)
110. Vasquez, M.R., Katiyar, P.: Texture classification using logical operations. IEEE Trans. Image Anal. **9**(10), 1693–1703 (2000)
111. de Vaucouleurs, G., de Vaucouleurs, A., Corwin, H.G., Buta, R., Paturel, G., Fouque, P.: Third Reference Catalogue of Bright Galaxies. Springer, New York (1991)
112. Waldemark, J., Becanovic, V., Lindblad, T., Lindsey, C.S.: Hybrid neural networks for automatic target recognition. In: International Conference on Computational Cybernetics and, Simulation, pp. 4016–4021 (1997).
113. Wilensky, G., Manukian, N.: The projection neural network. Int. Jt. Conf. Neural Netw. **2**, 358–367 (1992)
114. Wilson, A.D., Bobick, A.F.: Recognition and interpretation of parametric gesture. In: Proceedings of the Sixth International Conference on Computer Vision, pp. 329–336 (1998).
115. Xue, F., Zhan, K., Ma, Y.D., Wang, W.: The model of numerals recognition based on PCNN and FPF. In: Proceedings of the 2008 International Conference on Wavelet Analysis and, Pattern Recognition (2008).
116. Yamada, H., Ogawa, Y., Ishimura, K., Wada, M.: Face detection using pulse-coupled neural network. In: SICE Annual Conference in Fukui, pp. 1210–1214 (2003).

117. Yamaguchi, Y., Ishimura, K., Wada, M.: Synchronized oscillation and dynamical clustering in chaotic PCNN. In: Proceedings of SICE, pp. 730–735 (2002).
118. Yang, M.H., Ahuja, N.: Extraction and classification of visual motion patterns for hand gesture recognition.In: Proceedings of the IEEE CVPR, pp. 892–897 (1998).
119. Yarbus, A.L.: The Role of Eye Movements in Vision Process. USSR, Nauka, Moscow (1968); Eye Movements and Vision. Plenum, New York (1965).
120. Zeki, S.: A Vision of the Brain. Blackwell Scientific Publications, Oxford (1993)
121. Zeki, S.: The visual image in mind and brain. In: Freeman, W.H. (ed.) Mind and Brain, pp. 27–39. W.H. Freeman and Company, New York (1993)
122. Zhan, K., Zhang, H., Ma, Y.: New spiking cortical model for invariant texture retrieval and image processing. IEEE Trans. Neural Netw. **20**(12), 1980–1986 (2009)

Index

T. Lindblad and J. M. Kinser, *Image Processing Using Pulse-Coupled Neural Networks*, Biological and Medical Physics, Biomedical Engineering,
DOI: 10.1007/978-3-642-36877-6, © Springer-Verlag Berlin Heidelberg 2013