# Synthesis of 2 Dimensional Image Filters by Cellular Automata

David Sinclair,

Dublin City University,

Glasnevin,

Dublin 9.

9th. August 1991

## Table of Contents

## Abstract

The filtering action of some type of cellular automata has been known about since the mid 1960's. These cellular filters, based on a combination of reducing and augmenting kernels operating on a series of thresholded images, are derived empirically and have fixed characteristics. This thesis presents a method for synthesising 2 dimensional grey level image filters using cellular automata. This method takes as its starting point the characteristic equation of a filter expressed in the Laplace domain, and derives from this equation the kernel of the cellular filter which has the same characteristics as the filter defined by the Laplacian equation. The advantage of using cellular automata for image filtering is that they are highly parallel in nature, and as such, are a natural algorithm for image filtering on a parallel computational engine.

## 1.0 Introduction

The application of cellular automata to the task of image filtering has been known about since the mid 1960's. To date, this has been limited to empirical low pass filters with fixed amplitude response for use mainly on binary images. This thesis proposes a method based in the formal methods of linear system analysis, for the synthesis of any 2 dimensional grey level image filter which can be expressed by a transfer function in the Laplace domain.

The remainder of section 1 of this thesis introduces the concept of a cellular automaton, the process of image filtering and the previous work on the applications of cellular automata to the spatial filtering of images. In section 2 the method for synthesising 2 dimensional filters using cellular automata is developed using both an analytical approach and a heuristic approach. Several examples of filters are worked through in section 3 showing the validity of the method, and its limitations. In section 4 a method which extends this technique to the synthesis of other types of filters, such as high pass and band pass filters, is illustrated. The parallel nature, and the suitability of cellular filters for the parallel processing of images, is shown by the description, in section 5, of one possible implementation of a cellular filter using a network of transputers. Finally, section 6 summarises the results of this technique, its limitations and possible areas for further research in the synthesis of filters using cellular automata.

## 1.1 What are Cellular Automata?

A Cellular Automaton is an array of processing elements (PEs). The functions computed by a PE is a function of its state and the states of its neighbouring elements. At its simplest, the role of a cellular operation is to transform an array of data s(i,j) as it exists at a time t, given by s(i,j,t), into an array of data d(i,j,t+1). At time (t+1), an element of the array of data d(i,j) has a value determined by its original value s(i,j), along with the values of its nearest neighbours.

It is also assumed that the array of PEs is ***tightly coupled*** in that each PE is connected to its immediate adjacent neighbouring PEs. A loosely coupled array of PEs would allow connections, through some form of switching network, to an arbitrary set of remote PEs.

A further restriction placed on the behaviour of the array of PEs is that all operations on a block of data in the array of PEs are independent of the position of the data in the array. If the data in the array is shifted to a new location in the array and operated upon, then the results are identical to the same operation conducted on the data in its original position in the array.

Operations on the array are assumed to be ***discrete and synchronous*** in time. By discrete in time, we mean that the operations are assumed to occur in discrete time with each step in time being a generation, iteration, or cycle. By synchronous in time, we mean that the state changes of the PEs (i.e. the change in the values of the PEs) occur simultaneously.

The *neighbourhood* of the PE p(i,j) is defined as all the PEs coupled to p(i,j). The state of p(i,j) at a time (t+1) is determined by its present state and the states of the PEs in its neighbourhood. The *kernel* of the PE p(i,j) includes both itself and the PEs in its neighbourhood. Consider the PE p(i,j) and its neighbouring PEs arranged in a square tessellation, as follows.

$$\begin{array}{lllll}
p(i-2,j-2), & p(i-1,j-2), & p(i,\ j-2) & ,\ p(i+1,j-2), & p(i+2,\ j-2) \\
p(i-2,j-1), & p(i-1,j-1), & p(i,\ j-1) & ,\ p(i+1,j-1), & p(i+2,\ j-1) \\
p(i-2,j) & ,\ p(i-1,j) & ,\ p(i,\ j) & ,\ p(i+1,j) & ,\ p(i+2,\ j) \\
p(i-2,j+1), & p(i-1,j+1), & p(i,\ j+1) & ,\ p(i+1,j+1), & p(i+2,\ j+1) \\
p(i-2,j+2), & p(i-1,j+2), & p(i,\ j+2) & ,\ p(i+1,j+2), & p(i+2,\ j+2)
\end{array}$$

If we give p(i,j) a 3x3 kernel, then the neighbourhood of p(i,j) includes p(i-1,j-1), p(i,j-1), p(i+1,j-1), p(i-1,j), p(i+1,j), p(i-1,j+1) p(i,j+1) and p(i+1,j+1). In general the neighbourhood of p(i,j) is

$$\{\ p(i+n,j+m)\ \mid\ -1<=n<=+1,\ -1<=m<=+1,\ (n,m) <> (0,0)\ \}$$

and the kernel of p(i,j) is

$$\{\ p(i+n,j+m)\ \mid\ -1<=n<=+1,\ -1<=m<=+1\ \}.$$

And for an NxM kernel the neighbourhood of p(i,j) is

$$\{\ p(i+n,j+m)\ \mid\ \frac{-N}{2}<=n<=\frac{n}{2},\ \frac{-M}{2}<=m<=\frac{M}{2},\ (n,m) <> (0,0)\ \}$$

and the kernel of p(i,j) is

$\{ p(i+n,j+m) \mid \frac{-N}{2}<=n<=\frac{n}{2}, \frac{-M}{2}<=m<=\frac{M}{2} \}$.

The kernels, neighbourhoods and tessellations (the physical arrangement of the PEs) do not have to be rectangular. The tessellation can be any arrangement as long as operations on the array are isotropic. This tends to limit the tessellations to homogeneous regular geometric arrangements such as square, hexagonal and triangular which are capable of completely covering a plane surface. The kernels can be any combination the PE and its nearest neighbouring PEs. For example, figure 1 illustrates some of the possible neighbourhoods for a square tessellation.

The configuration of the array (i.e. the size and tessellation of the array and the state of each PE in the array) is called the signal, denoted by s(r). Usually r is a 2-dimensional space in which case the signal is s(i,j). The symbol used for the kernel is k(r).

The operation of the kernel k(r) on an element of s(r) (i.e. s(i,j) a given PE in the array defined by s(r) ) is the weighted sum of the values held by the PEs in the neighbourhood of s(i,j) and s(i,j). The weights are defined by the kernel k(r).

For example, if we define k(r) to be a 3x3 kernel as follows

$$\begin{array}{lll} k(i-1,j-1) = 1, & k(i,j-1) = -1, & k(i+1,j-1) = 0 \\ k(i-1,j) = -1, & k(i,j) = 1, & k(i+1,j) = -1 \\ k(i-1,j+1) = 0, & k(i,j+1) = 0, & k(i+1,j+1) = 1 \end{array}$$

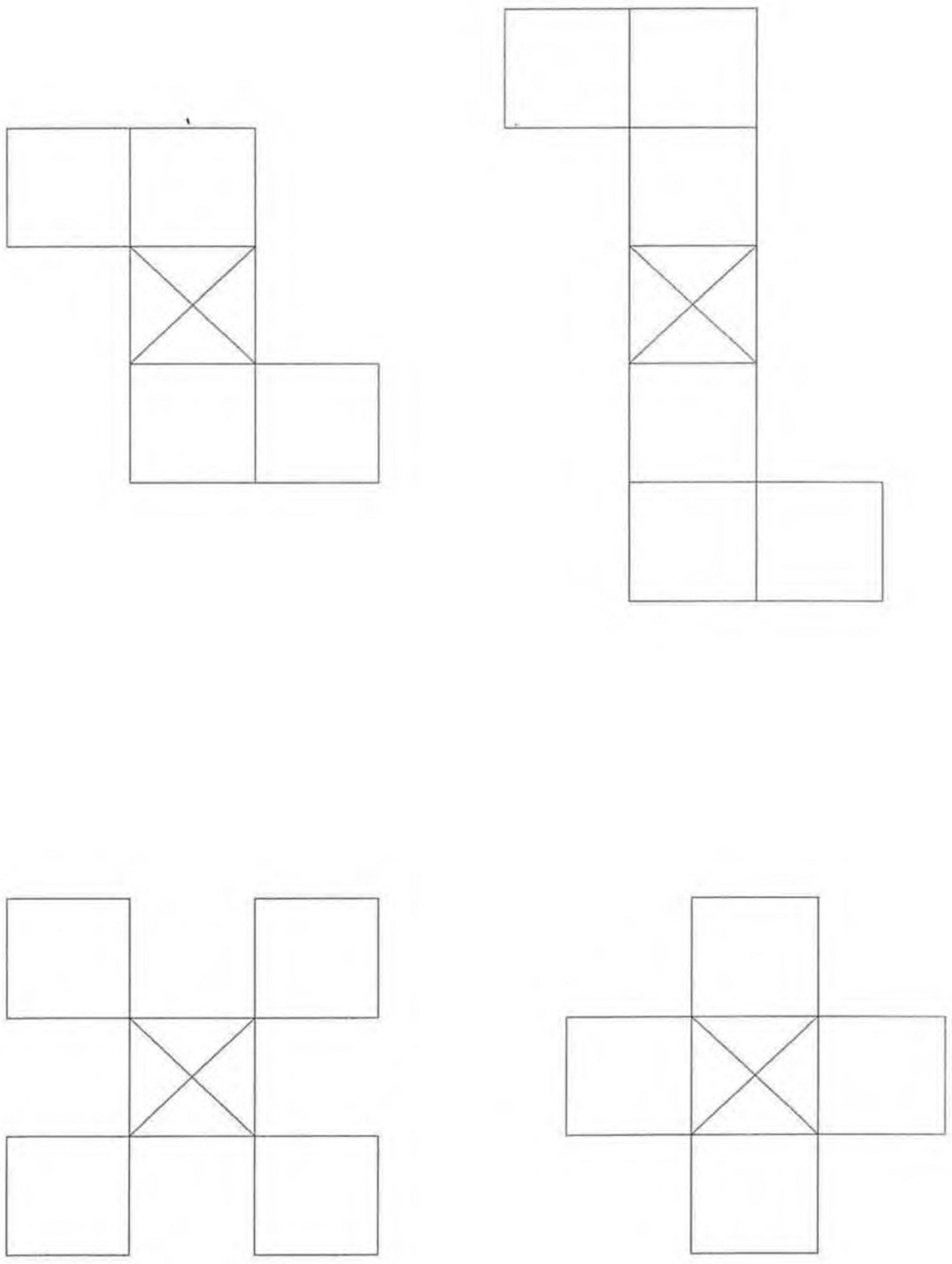


Figure 1 : Various neighbourhoods on a square tessalation

then the action of k(r) on a specific element of s(r), namely s(i,j) at time t is

$$s(i,j,t+1) = s(i-1,j-1,t) - s(i,j-1,t) - s(i-1,j,t) + s(i,j,t) - s(i+1,j,t) + s(i+1,j+1,t)$$

In general for 2-dimensional signal,

$$s(i,j,t+1) = \sum_{n,m}(k(i+n,j+m).s(i+n,j+m)) \qquad (1)$$

In the above example the kernel k(r) was a binary kernel, that is a kernel which only contains the values 0 and (+/-)1. Kernels are not limited to binary kernels and can contain any value. However binary kernels do have a particular interest since they are easily implemented in hardware.

This operation of the kernel k(r) is applied to all the PEs in s(r) simultaneously and after a single iteration the output of the 2-dimensional array s(r,t+1) is the discrete 2-dimensional convolution of the kernel k(r) with the signal s(r) at a time t.

## 1.2 Image Filtering

Image filtering is the process by which certain spatial frequencies are removed from an image. If the low spatial frequencies are removed from an image, then those regions of the image in which the signal (intensity) is constant and/or changing slowly, are reduced to a zero value. Only abrupt changes the image intensity remain. Such high pass filters, filters which attenuate low frequencies, are used for edge detection and enhancement because quick changes in image intensity is usually associated with the boundary, or edge, of a object and its background. High pass filters can also be used to remove shadows caused by diffuse or normal sunlight.

Low pass filters attenuate high spatial frequencies in an image and pass low spatial frequencies unaffected. These filters are very useful in removing speckling and snow effects from images. This type of high frequency noise in the image is generally caused by irregularities in the image capture system and changes in the environment. Such noise is expensive to completely eliminate since it would require high quality optics and a controlled environment.

A low pass filter can be combined with a high pass filter to produce a band pass filter which combines the operations of noise removal and edge enhancement. Band reject filters, that is filters which attenuate a specific band of frequencies, have applications in object recognition since certain textures can be characterised by a close band of frequencies localised in a region of the image.

## 1.3 Previous Work on Image filtering using Cellular Automata

To date image filters based on cellular automata have used the augmenting and reducing properties of binary multi-point kernels (kernels of 2 or more PEs).

For example consider the triple point kernel n(x,y) shown in figure 2. When this kernel is applied to the signal data s(x,y,t) the result is s(x,y,t+1) which is no longer a binary signal. In order to transform the signal s(x,y,t+1) back into a binary signal we need to threshold s(x,y,t+1). Figures 2(c), 2(d) and 2(e) are the results when s(x,y,t+1) is thresholded using the values 0, 1 and 2. In figure 2(c) the signal data has been augmented, i.e. is larger than the original signal data, when the threshold value was 0. When the threshold value was 1, as in figure 2(d), the signal data has been reduced. When the threshold value was 2, as in figure 2(e), the result was a "residue" of the original signal, i.e. a single element in a background of elements of opposite value.

In general, these results hold true for dual-point, triple-point and multi-point kernels, since mutli-point kernels can be generated by a sequence of dual-point or triple-point kernels. Augmentation and reduction by means of an 8 step sequence using dual-point kernels, or by means of a 4 step sequence using triple-point kernels, can cause the removal of certain high spatial frequencies and change the duty cycle of lower spatial frequencies.

When used to filter grey level images these sequences of kernels must be applied to the set of binary signal images formed by thresholding the original image at each grey level. For example if the signal image has 8 grey levels, then the sequence of kernels would have to be applied to the 8 signal images formed by thresholding the original image at values 0, 1, ..., 7 . Then the 8 resulting signals are recombined to produce the filtered grey level image.

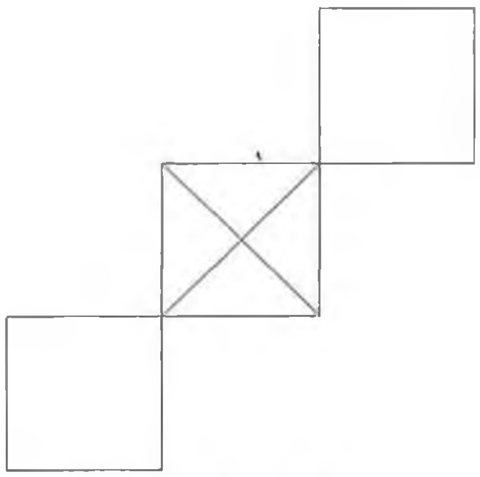


Figure 2(a): Kernel

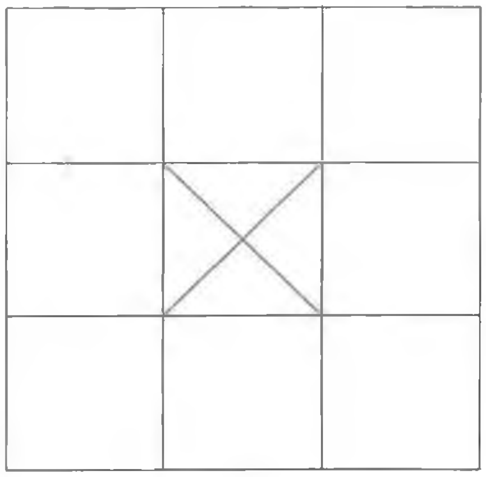


Figure 2(b): Signal data

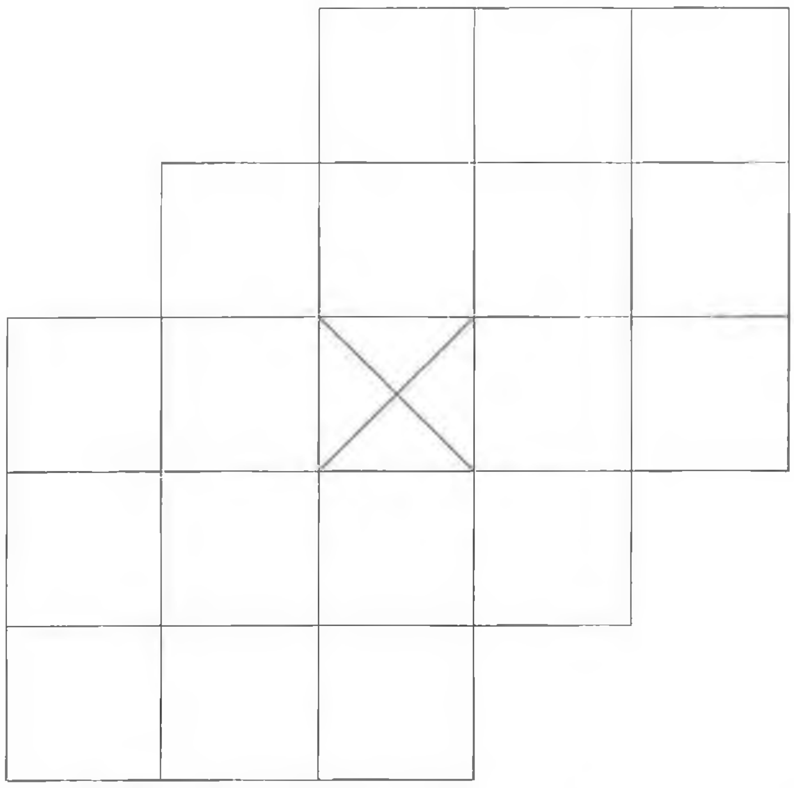


Figure 2(c): Augmentation with threshold = 0

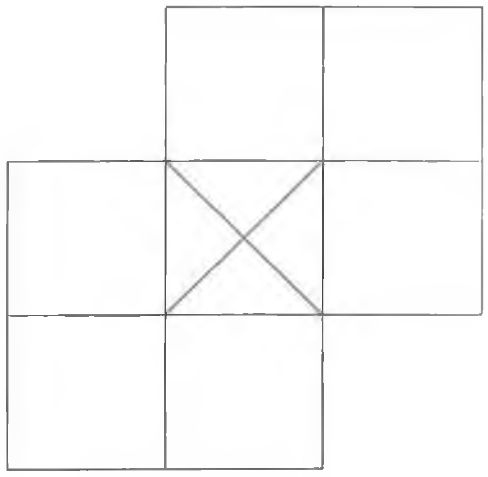


Figure 2(d): Reduction with threshold = 1

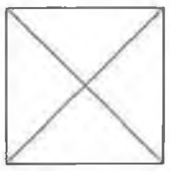


Figure 2(e): Residue with threshold = 2

The work of Preston et al. has shown, by analysising the boundary propagation of the signal, that if $n_i$ is the number of iterations of reduction are followed by $n_i$ iterations of augmentation, with a sampling interval of $\Delta x$, then for a signal function of period P, then that sinusoid and all sinusoids of higher frequencies will be annihilated when

$$n_i = \frac{P}{2(\Delta x)}$$

and therefore the cutoff frequency of the cellular filter is

$$f_{co} = \frac{1}{2n_i(\Delta x)}$$

Lower frequencies will not be annihilated but will be clipped at their crests. The peak to peak amplitude $A_{p-p}$ is given by

$$A_{p-p}(f) = \frac{1}{2[1 + \mathrm{Cos}\ (((\pi)f/f_{co}))]}$$

The advantage of this method of filtering a grey level image with N grey levels using cellular automata is that it uses binary kernels, and is therefore it is easier to implement in hardware than multi-value kernel. However this method does have 4 major disadvantages:

(1) It requires the processing a N thresholded images with $2n_i$ iterations for each thresholded image.

(2) The characteristics of the filter produced by the cellular automata are fixed, i.e. it is not possible to alter the attenuation suffered by frequencies less than the cutoff frequency.

(3) It only produces low pass filters. This technique does not provide any means of synthesising high pass, band pass or any other type of filter.

(4) Non linear group delay causing a change in the duty cycle of certain low frequency components and image distortion.

This thesis proposes a new technique for the synthesis of 2 dimensional image filters using kernels with floating point values which overcomes the disadvantages of previous methods. Since the kernel does contain floating point values, each iteration does require more processing, but this technique is applied to the original image only and does not require N thresholded images. Therefore it is more efficient when applied to high resolution grey level images.

# 2.0 Filter Synthesis using a 3x3 Cellular Filter

## 2.1 Assumptions

Throughout this thesis on filter synthesis using cellular automata, the following assumption are used.

(1) The image to be filtered is arranged in a square tessellation, and is represented by the 2-dimensional signal a(x,y).

(2) The image is a 8 bit gray level image which gives a range of -127 (black) to +127 (white).

(3) The filtered image is represented by the 2-dimensional signal b(x,y).

(4) The behaviour of the filter is limited to isotropic behaviour. Therefore the results of the filtering process are independent of direction.

(5) The square root of -1 ($\sqrt{-1}$) is represented by j.

## 2.2 The Filtering Operation

The action of a filter h(x,y) on a signal a(x,y) is defined by the equation

$$b(x,y) = \iint (a(x,y).h(x-X,y-Y))\, dY\, dX \qquad (2)$$

and hence is the convolution of the input signal a(x,y) with a filter characterised by h(x,y).

In the 2-dimensional s'-domain, the Laplace transform of equation (2) is written as

$$B(s') = H(s').A(s') \qquad (3)$$

where H(s') is called the transfer function of the filter. H(s') is the Laplace transform of h(x,y), and so we can write,

$H(s') = L\{h(x,y)\}$ ... Laplace transform

$h(x,y) = L^{-1}\{H(s')\}$ ... Inverse Laplace transform

An important property of the Laplace transform is the Laplace transform of the delta function $\delta(x,y)$.

$$L\{\delta(x,y)\} = 1$$

and hence, if we let the a(x,y) be the delta function,

$$B(s') = H(s').1 \qquad (4)$$

and

$$b(x,y) = h(x,y) \qquad (5)$$

This is why h(x,y) is called the impulse response of the filter, since it is the output of a filter characterised by the transfer function H(s') when a delta function, or impulse, is the input to the filter.

Since we have limited the behaviour of the filter H(s') to isotropic behaviour, this also places limitations on the form of the filter's impulse response. Since the filter's response to any input signal should be independent of direction, we have

$$L^{-1}\{H(s')\} = h(x,y) = h(r,\theta) = h(r) \qquad (6)$$

where $(r,\theta)$ are the polar coordinates of (x,y).

We can therefore represent H(s'), the 2-dimensional filter transfer function, by a 1-dimensional filter transfer function H(s).

Now consider a 3x3 kernel K, which is to act as a filter. From equation (1) we know that the action of one iteration of the kernel K on the input signal a(x,y) is the discrete convolution of K with a(x,y). This is already very similar to the filtering action of a filter h(x,y) as defined by equation (2). However, h(x,y) is not limited to a 3x3 domain, and therefore multiple iterations of the kernel K are necessary to approximate the action of h(x,y).

Since the filter has to be isotropic, the action of the kernel K must be isotropic, and K must be symmetrical about its 4 major axes.

$$\begin{matrix} k(0,0) & k(1,0) & k(2,0) \\ k(0,1) & k(1,1) & k(2,1) \\ k(0,2) & k(1,2) & k(2,2) \end{matrix}$$

Therefore the 9-element kernel K can be simplified to a kernel with 3 distinct elements.

$$\begin{array}{ccc} K_2 & K_1 & K_2 \\ K_1 & K_0 & K_1 \\ K_2 & K_1 & K_2 \end{array}$$

or formally as,

$$K = \{k(i,j) \mid 0<=i<=2,\ 0<=j<=2,\ k(1,1) = K_0,$$
$$k(1,0) = k(0,1) = k(2,1) = k(1,2) = K_1,$$
$$k(0,0) = k(2,0) = k(1,0) = k(2,2) = K_2\} \quad (7)$$

We will denote the action of one iteration of K to a signal a(x,y) as

$K*a(x,y)$

If we let the signal array a(x,y) be the impulse signal $\delta(x,y)$ where

$$\delta(x,y) = 1;\ x = 0,\ y = 0$$
$$0;\ x <> 0,\ y <> 0$$

Then,

$$K*\delta(x,y) \ -> \ \begin{array}{ccc} (K_2) & (K_1) & (K_2) \\ (K_1) & (K_0) & (K_1) \\ (K_2) & (K_1) & (K_2) \end{array}$$

where $K_0$ is located at (0,0).

The application of K to $K*\delta(x,y)$, denoted $K^2*\delta(x,y)$, is then

$K^2*\delta(x,y)$ ->

$$\begin{array}{ccccc}
(K_2{}^2) & (2K_1K_2) & (K_1{}^2+2K_2{}^2) & (2K_1K_2) & (K_2{}^2) \\
(2K_1K_2) & (K_1{}^2+2K_2{}^2) & (2K_0K_1+4K_1K_2) & (K_1{}^2+2K_2{}^2) & (2K_1K_2) \\
(K_1{}^2+2K_2{}^2) & (2K_0K_1+4K_1K_2) & (K_0{}^2+4K_1{}^2+4K_2{}^2) & (2K_0K_1+4K_1K_2) & (K_1{}^2+2K_2{}^2) \\
(2K_1K_2) & (K_1{}^2+2K_2{}^2) & (2K_0K_1+4K_1K_2) & (K_1{}^2+2K_2{}^2) & (2K_1K_2) \\
(K_2{}^2) & (2K_1K_2) & (K_1{}^2+2K_2{}^2) & (2K_1K_2) & (K_2{}^2)
\end{array}$$

Since K is isotropic, then $K^2*\delta(x,y)$ is symmetrical about its major axes (horizontal, vertical and diagonal). Repeated applications of K to $\delta(x,y)$ will also be isotropic, therefore $K^n*\delta(x,y)$ is isotropic. $K^n*\delta(x,y)$ is also non-zero for all $(x,y)$ where $\frac{-n}{2} <= x <= \frac{n}{2}$ and $\frac{-n}{2} <= y <= \frac{n}{2}$.

The impulse response of the filter $h(x,y)$ is an infinite continuous function, with 1 representing white and 0 representing black. However, since we have restricted the image intensity to 8 bits, we can consider $h(x,y)$ to be zero valued for all $(x,y)$ where $h(x,y)$ is less than the minimum discrete resolution, i.e.

$$h(x,y) = h(lR,mR) = 0 \text{ if } h(lR,mR) < \frac{1}{2^8}$$

where R = resolution of the signal array.

Therefore h(x,y) is also limited in the same manner as $K^n * \delta(x,y)$. In order to synthesise the filter we need to choose n, $K_0$, $K_1$ and $K_2$ such that $K^n * \delta(x,y)$ approximates h(x,y). Then n iterations of the kernel K to the delta function $\delta(x,y)$ will have the same effect as applying the filter to the delta function.

The value of n, the order of the cellular filter K, is easily determined. First, we derive the impulse response of the filter from the Laplacian transfer function of the filter. We then determine at which $\sqrt{2}$ multiple of the resolution ($\sqrt{2}$mR) of the signal array does the impulse response remain less than the minimum resolution of the image intensity. The impulse response will always decay to a value less than the minimum image intensity resolution. This is because in order for h(x,y) = h(r) to have any Laplace transform , h(r) must meet the following condition.

$$\int_0^{\text{infinity}} |h(r)| \exp(-ar)dr < \text{infinity}$$

for some real, positive a. Therefore |h(r)| < M(exp(ar)) for all positive r, for some real positive M.

For this multiple (m) of the signal array resolution to exist at the furthest point of $K^n$, an nxn response to an impulse function, from the origin, then the value of n is given by

$$n = \text{floor}\left(\frac{m}{\sqrt{2}}\right) \qquad (8)$$

where floor(x) = largest integer < x.

The values of $K_0$, $K_1$ and $K_2$ can be derived by two means, (a) analytically or (b) by a heuristic search.

## 2.3 Analytical Estimation of the Coefficients $K_0$, $K_1$ and $K_2$.

By induction, equations for the value of $K^n*\delta(x,y)$ at the different points, in terms of $K^0$, $K^1$, $K^2$ and n, will be derived. The actual value of h(x,y) at these points can be set equal to the derived equations, and the values of $K^0$, $K^1$ and $K^2$ can be solved for.

Consider the following 3 elements of $K^n*\delta(x,y)$,

$K^n*\delta(x,y)$ for $x = n$, $y = n$ (case 1)

$x = n-1$, $y = n$ (case 2)

$x = n-1$, $y = n-1$ (case 3)

Case 1: $x = n$, $y = n$

When $i = 2$, for $x = i$, $y = i$,

$$K^i*\delta(x,y) = K^2*\delta(x,y) = {K_2}^2 = {K_2}^i$$

If K is applied to $K^2*\delta(x,y)$ then

$$K*(K^2*\delta(x,y)) = {K_2}^2.K_2 \qquad \text{at } x = i+1,\ y = i+1$$

$$\Rightarrow K^{i+1}*\delta(x,y) = {K_2}^3 = {K_2}^{i+1}$$

Let j = i+1 and then

$$K^j * \delta(x,y) = K_2^j \qquad \text{at } x = j,\ y = j$$

This equation has the same form as $K^i * \delta(x,y)$ and therefore ,in general, $K^n * \delta(x,y)$ at x = n, y = n can be expressed as

$$K^n * \delta(x,y) = K_2^n \qquad \text{for } x = n,\ y = n$$

Case 2: $x = n-1,\ y = n$

When i = 2, for $x = i-1$, $y = i$,

$$K^i * \delta(x,y) = K^2 * \delta(x,y) = 2K_1K_2 = iK_1K_2$$

If K is applied to $K^2 * \delta(x,y)$ then

$$K*(K^2 * \delta(x,y)) = 2K_1K_2.K_2 + K_2^2.K_1 \qquad \text{at } x = i,\ y = i+1$$

$$\Rightarrow K^{i+1} * \delta(x,y) = 2K_1K_2^2 + K_1K_2^2 = 3K_1K_2^2$$

$$= (i+1)K_1K_2^i$$

Let j = i+1 and then

$$K^j * \delta(x,y) = jK_1K_2^{j-1} \qquad \text{at } x = j-1,\ y = j$$

This not the same form as the expression for $K^i*\delta(x,y)$ so we apply another iteration of K to $K^j*\delta(x,y)$.

$$K*(K^j*\delta(x,y)) = jK_1K_2{}^{j-1}.K_2 + K_2{}^j.K_1$$

$$\Rightarrow K^{j+1}*\delta(x,y) \quad = (j+1)K_1K_2{}^j$$

let l = j+1 and then

$$K^l*\delta(x,y) = lK_1K_2{}^{l-1} \qquad \text{at } x = l-1,\ y = l$$

This equation has the same form as $K^j*\delta(x,y)$ and therefore ,in general, $K^n*\delta(x,y)$ at $x = n-1$, $y = n$ can be expressed as

$$K^n*\delta(x,y) = nK_1K_2{}^{n-1} \qquad \text{for } x = n-1,\ y = n$$

Case 3: $x = n-1$, $y = n-1$

When $i = 2$, for $x = i-1$, $y = i-1$,

$$K^i*\delta(x,y) = K^2*\delta(x,y) = 2K_0K_2 + 2K_1{}^2 = iK_0K_1 + iK_1{}^i$$

If K is applied to $K^2*\delta(x,y)$ then

$$K*(K^2*\delta(x,y)) = (2K_0K_2 + 2K_1{}^2)(K_2) + 2K_1K_2.K_1 + 2K_1K_2.K_1 + K_2{}^2.K_0$$

$$\text{at } x = i,\ y = i$$

$$\Rightarrow K^{i+1}*\delta(x,y) = iK_0K_2^i + iK_1^iK_2 + iK_1^iK_2 + iK_1^iK_2 + K_0K_2^i$$
$$= (i+1)K_0K_2^i + 3iK_1^iK_2 \quad \text{at } x = i,\ y = i$$

Let $j = i+1$ and then

$$K^j*\delta(x,y) = jK_0K_2^{j-1} + X(j)K_1^2K_2^{j-2} \quad \text{at } x = j-1,\ y = j-1$$

since the applications of K to $K^2*\delta(x,y)$ did not affect the power of $K_0$ in the first term and did not affect the power of $K_1$ in the second term. $X(j)$ denotes that the coefficient of the $K_1^2K_2^{j-2}$ term is some, as yet unknown, function of the variable j.

Applying K to $K^j*\delta(x,y)$ gives, at $x = j$, $y = j$

$$\begin{aligned} K^{j+1}*\delta(x,y) &= (jK_0K_2^{j-1} + X(j)K_1^2K_2^{j-2})(K_2) \\ &\quad + jK_1K_2^{j-1}.K_1 \\ &\quad + jK_1K_2^{j-1}.K_1 \\ &\quad + K_2^j.K_0 \end{aligned}$$

$$\Rightarrow K^{j+1}*\delta(x,y) = jK_0K_2^j + X(j)K_1^2K_2^{j-1} + 2jK_1^2K_2^{j-1} + K_0K_2^j$$

$$= (j+1)K_0K_2^j + (2j + X(j))K_1^2K_2^{j-1}$$

Let $l = j+1$ and then

$$K^l*\delta(x,y) = lK_0K_2^{l-1} + (2(l-1) + X(l))K_1^2K_2^{l-2}$$
$$\text{at } x = l-1,\ y = l-1$$

But $(2(l-1) + X(l))$ is a function equal to its last value plus twice the last iteration number, which can be written

$$(2(l-1) + X(l)) = 2(\sum_{m=0}^{l-1} m)$$

And therefore,

$$K^l * \delta(x,y) = lK_0K_2^{l-1} + 2(\sum_{m=0}^{l-1} m)K_1^2K_2^{l-2}$$

Since $K^l * \delta(x,y)$ has the same form as $K^j * \delta(x,y)$, in general, $K^n * \delta(x,y)$ at $x = n-1$, $y = n-1$ can be expressed as

$$K^n * \delta(x,y) = nK_0K_2^{n-1} + 2(\sum_{m=0}^{l-1} m)K_1^2K_2^{n-2}$$

$$\text{for } x = n-1,\ y = n-1$$

In summary, the n repeated applications of the kernel K to the impulse function, yields:

$$K^n * \delta(x,y) = nK_0K_2^{n-1} + 2(\sum_{m=0}^{l-1} m)K_1^2K_2^{n-2} \qquad \text{at } x = n-1,\ y = n-1$$

$$K^n * \delta(x,y) = nK_1K_2^{n-1} \qquad \text{at } x = n-1,\ y = n$$

$$K^n * \delta(x,y) = K_2^n \qquad \text{at } x = n,\ y = n$$

If $K^n$ is to have the same filtering effects as the filter defined by the Laplacian transfer function H(s), then the response of H(s) to an impulse function should be the same at $(x,y) = \{(n-1,n-1), (n-1,n), (n,n)\}$ as $K^n*\delta(x,y)$ at the same points.

The spatial response of H(s) to an impulse function has already been shown to be (equation (6))

$$h(r) = L^{-1}\{H(s)\}$$

Hence, from the transfer function H(s) of the filter,

$$K^n*\delta((n-1)R,\ (n-1)R) = h((n-1)R,\ (n-1)R)) \qquad (9)$$

$$K^n*\delta((n-1)R,\ nR) = h((n-1)R,\ nR) \qquad (10)$$

$$K^n*\delta(nR,\ nR) = h(nR,\ nR) \qquad (11)$$

And therefore,

$$nK_0K_2^{n-1} + 2(\sum_{m=0}^{l-1} m)K_1{}^2K_2{}^{n-2} = h((n-1)R,\ (n-1)R) \qquad (12)$$

$$nK_1K_2{}^{n-1} = h((n-1)R,\ nR) \qquad (13)$$

$$K_2{}^n = h(nR,\ nR) \qquad (14)$$

Since n has already been calculated, the values of $K_0$, $K_1$ and $K_2$ can be determined.

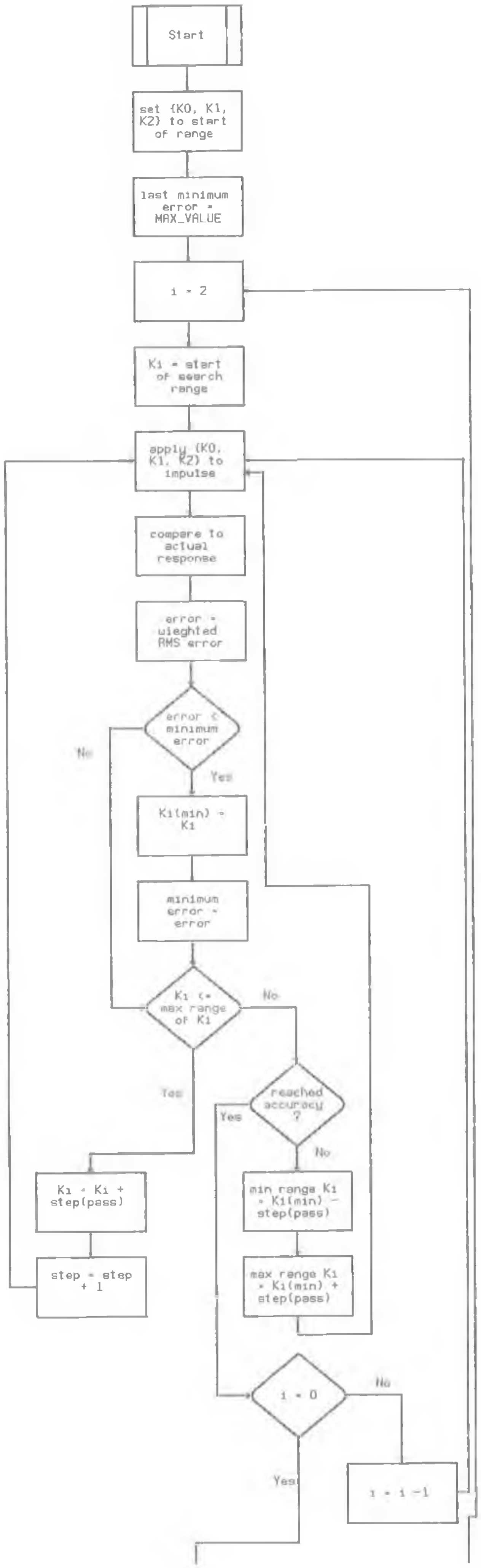
## 2.4 Heuristic Estimation of the Coefficients $K_0$, $K_1$ and $K_2$.

### 2.4.1 The Search Procedure

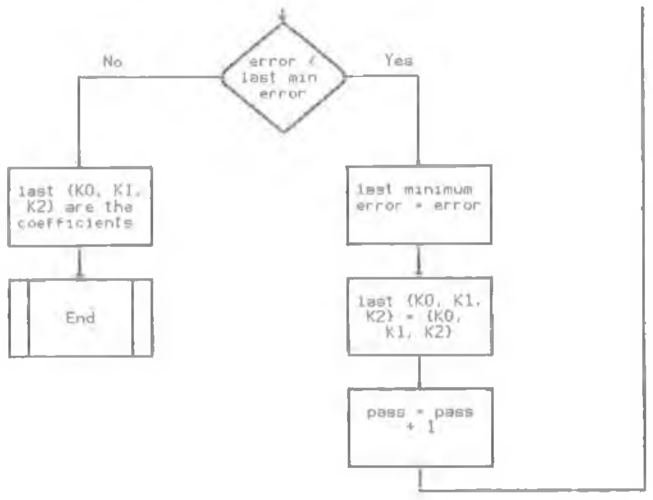
The coefficients of the cellular filter can be estimated by a heuristic search, as described in the flow chart in figure 3. For a given value of n (the order of the cellular filter) a range of values for $K_0$, $K_1$ and $K_2$ is specified. Initially the filter coefficients are set to the beginning of the search ranges. In turn, starting with $K_2$ and then progressing onto $K_1$ and $K_0$, each coefficient, $K_i$, is advanced in pre-defined steps, $d_k$, and at each triplet $\{K_0, K_1, K_2\}$, the response to the delta function is calculated. The value of the response at several points in the calculated impulse response for $\{K_0, K_1, K_2\}$ is compared with the values at the corresponding points in the actual impulse response. This error between the calculated impulse response and the actual impulse response at the different sample points is multiplied by a weighting function. The triplet $\{K_0, K_1, K_2\}$ with the minimum root mean square of the weighted error in the search range is recorded. The search range is then changed to $[K_i-d_k, K_i+d_k]$ and the step value is changed to $d_{k+1}$. The new interval $[K_i-d_k, K_i+d_k]$ is searched for a minimum root mean square weighted error. This process is repeated until $K_i$ reaches a desired accuracy. When $K_i$ reaches the desired accuracy this value of $K_i$ is recorded and is used for the searches involving the other coefficients.

When this procedure has been completed for $K_0$, $K_1$, and $K_2$, the current pass is finished. Then the triplet $\{K_0, K_1, K_2\}$, with the minimum root mean square weighted error, and the error are recorded. Another pass is started, but this time only $K_2$ is set to the beginning

Figure 3: Heuristic search algorithim for Cellular Filter coefficients



Start
set {K0, K1, K2} to start of range
last minimum error = MAX_VALUE
i = 2
Ki = start of search range
apply {K0, K1, K2} to impulse
compare to actual response
error = uieghted RMS error
error < minimum error
No
Yes
Ki(min) = Ki
minimum error = error
Ki <= max range of Ki
No
Yes
reached accuracy ?
Yes
No
Ki = Ki + step(pass)
step = step + 1
min range Ki = Ki(min) - step(pass)
max range Ki = Ki(min) + step(pass)
i = 0
No
Yes
i = i -1


error < last min error
No
Yes
last (K0, K1, K2) are the coefficients
End
last minimum error = error
last (K0, K1, K2) = (K0, K1, K2)
pass = pass + 1

of the specified search range. $K_0$ and $K_1$ remain at the values associated with the minimum root mean square weighted error during the last pass. When $K_2$ reaches a minimum, $K_1$, and then $K_0$, are searched.

The search for the final coefficient values is completed if the minimum root mean square error for a pass is within 1% of the minimum root mean square error for the previous pass.

The program which implements this algorithm, find_ks.c (appendix A), initially performs a search for the filter coefficients with a 1 decimal point accuracy. This is done in order to quickly locate the region in which the minimum of the error function exists before performing the search to the desired degree of accuracy.

## 2.4.2 Weighting Functions for Heuristic Search

The weighting function is a function of the distance of the point from the origin and is used because the delta function, being a mathematical entity, can only be approximated in reality. The approximation used for the delta function in the heuristic search algorithm is the value of actual impulse response at the origin. The weighting function reduces the error introduced by this approximation of the delta function.

Two different weighting functions were used in the heuristic search algorithm, $W_1(i)$ and $W_2(i)$, defined as follows.

$$W_1(i) = \begin{cases} 0.5 & , 0 <= i < 1 \\ 0.9 & , 1 <= i < 2 \\ 1.0 & , i >= 2 \end{cases}$$

$$W_2(i) = \begin{cases} 0.0 & , 0 <= i < 1 \\ 0.01 & , 1 <= i < 2 \\ 0.1 & , 2 <= i < 3 \\ 0.5 & , 3 <= i < 4 \\ 0.9 & , 4 <= i < 5 \\ 1.0 & , i >= 5 \end{cases}$$

Both weighting functions reduce the error between the calculated and actual impulse responses, but $W_2(i)$ allows less contribution to the total error number than $W_1(i)$ at points close to the origin.

## 3.0 Example of the Synthesis of a Filter By a 3x3 Kernel

### 3.1 Test Pattern

In order to evaluate the cellular filter we will apply it to test image. This test image is defined by the following equation.

$$T(x,y) = \mathrm{Cos}\left(\frac{5(\pi)(x^2+y^2)}{R}\right) \qquad (15)$$

where R is the grid resolution.

This generates a test image which consists of a sinusoid whose frequency varies linearly with distance.

$$\omega = 2(\pi)f = \frac{10(\pi)r}{R} = 10(\pi)m$$

$=> f = 5m$

since $r = mR$.

When this test image is passed through a threshold function, a function which maps the input to 1 if it is greater or equal to a given value or maps the input to 0 otherwise, then the test image appears as a series of concentric circles of different thicknesses. Figure 4 shows the test pattern with a threshold value of 0. Note that the resolution of the images in figures 4-10 is 72 dots per inch (dpi). Therefore when used to display 300 dpi images, a correction factor of 4.1667 should be used to calculate the actual frequencies from the images.

## 3.2 Synthesis of a 750Hz. 5th. Order Low Pass Bessel Filter

The aim of this example is to illustrate how a 3x3 kernel for a cellular filter can be derived from a transfer function for a common filter. In this example a 5th order low pass Bessel filter with 3dB cutoff frequency of 750Hz for use on a square tessellation of resolution $8.4667 \times 10^{-5}$m. (300 dpi) will be synthesised. The Bessel filter was selected because it has a constant group delay, and therefore will not introduce distortion into the filtered image.

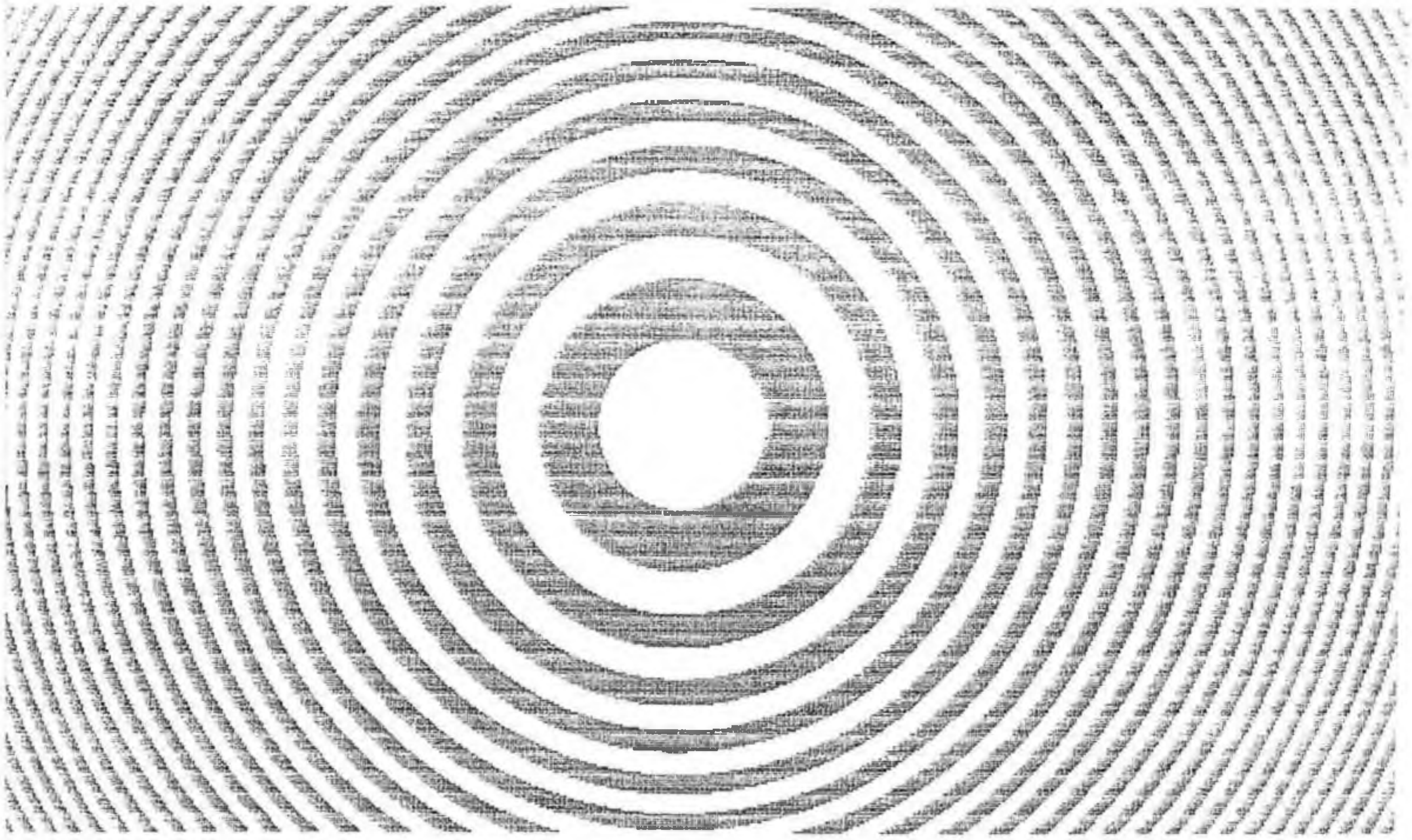


Figure 4: Test Pattern thresholded at 0

The normalised 5th order low pass Bessel transfer function is

$$H(s_n) = \frac{945}{s_n^5 + 15s_n^4 + 105s_n^3 + 420s_n^2 + 945s_n + 945}$$

$\Rightarrow H(s_n) =$

$$945\left\{\frac{1}{(s_n+3.6467)[(s_n+3.352)^2+1.7427^2][(s_n+2.3247)^2+3.571^2]}\right\}$$

$\Rightarrow H(s_n) =$

$$945\left\{\frac{1}{(s_n+3.6467)(s_n^2+6.704s_n+14.272)(s_n^2+4.649s_n+18.156)}\right\}$$

$\Rightarrow H(s_n) =$

$$945\left\{\frac{1}{(s_n+3.6467)(s_n-p_1)(s_n-p_1^*)(s_n-p_2)(s_n-p_2^*)}\right\}$$

where $p_1^*$ is the complex conjugate of p1

and $p_1 = -3.352 + j1.742$

$p_2 = -2.325 + j3.571$

Therefore,

$\Rightarrow H(s_n) =$

$$945\left\{\frac{k_1}{(s_n+3.6467)} + \frac{k_2}{(s_n-p_1)} + \frac{k_2^*}{(s_n-p_1^*)} + \frac{k_3}{(s_n-p_1)} + \frac{k_3^*}{(s_n-p_1^*)}\right\}$$

Calculating $k_1$:

$$k_1 = \frac{1}{(s_n^2+6.704s_n+14.272)(s_n^2+4.649s_n+18.156)}$$

at $s_n = -3.6467$

$$\Rightarrow k_1 = \frac{1}{45.286} = 0.0221$$

Calculating $k_2$:

$$k_2 = \frac{1}{(s_n+3.6467)(s_n-p_1^*)(s_n-p_2)(s_n-p_2^*)}$$

at $s_n = -3.352 + j1.742$

$$\Rightarrow k_2 = \frac{1}{(0.2947+j1.742)(j3.484)(-1.027-j1.829)(-1.027+j5.313)}$$

$$= \frac{1}{(-6.0691+j1.0267)(-1.027-j1.829)(-1.027+j5.313)}$$

$$= \frac{1}{(8.1108+j10.046)(-1.027+j5.313)} = \frac{1}{-61.7042+j32.7754}$$

$$= \frac{1}{(69.8687)\exp(-j0.4883)}$$

$$= (0.0143)\exp(j0.4883)$$

Calculating $k_3$:

$$k_3 = \frac{1}{(s_n+3.6467)(s_n-p_1^*)(s_n-p_2)(s_n-p_2^*)}$$

at $s_n = -2.325 + j3.571$

$$\Rightarrow k_3 = \frac{1}{(1.3217+j3.571)(1.027+j1.829)(1.027+j5.313)(j7.142)}$$

$$= \frac{1}{(-5.174+j6.085)(1.027+j5.313)(j7.142)}$$

$$= \frac{1}{(-37.6433-j21.2402)(j7.142)} = \frac{1}{151.6975-j268.8484}$$

$$= \frac{1}{(308.6934)\exp(-j1.0571)}$$

$$= (3.2394\text{x}10^{-3})\exp(j1.0571)$$

Therefore the normalised transfer function of a 5th order low pass Bessel filter is

$$H(s_n) = 945\{\frac{0.0221}{(s_n+3.6467)} + \frac{(0.0143)\exp(j0.4883)}{(s_n+(3.352-j1.742))}$$

$$+\frac{(0.0143)\exp(-j0.4883)}{(s_n+(3.352+j1.742))} + \frac{(3.2394\text{x}10^{-3})\exp(j1.0571)}{(s_n+(2.325-j3.571))}$$

$$+ \frac{(3.2394\text{x}10^{-3})\exp(-j1.0571)}{(s_n+(2.325-j3.571))}\} \qquad (16)$$

This transfer function is for a filter with a 3dB cutoff frequency of 1Hz. To scale the 3dB cutoff frequency to $\omega_c$ we transform $s_n$ to $(s/\omega_c)$.

Since the 3dB cutoff frequency of the filter is 750Hz. then

$\omega_c = 2(\pi)(f_c) = 4.7124\text{x}10^3$ rads/m.

Hence the transfer function for a 5th order low pass Bessel filter with a 750Hz. 3dB cutoff frequency is

$$H(s) = 945 \{ \frac{(1.0414\text{x}10^2)}{(s+1.7185\text{x}10^4)} + \frac{(6.7387\text{x}10^1)\exp(j0.4883)}{(s+(1.5796\text{x}10^4-j8.209\text{x}10^3))} + \frac{(6.7387\text{x}10^1)\exp(-j0.4883)}{(s+(1.5796\text{x}10^4+j8.209\text{x}10^3))} + \frac{(1.5265\text{x}10^1)\exp(j1.0571)}{(s+(1.0956\text{x}10^4-j1.6828\text{x}10^4)} + \frac{(1.5265\text{x}10^1)\exp(-j1.0571)}{(s+(1.0956\text{x}10^4+j1.6828\text{x}10^4)} \}$$

$$\Rightarrow H(s) = \frac{(9.8412\text{x}10^4)}{(s+1.7185\text{x}10^4)} + \frac{(6.3681\text{x}10^4)\exp(j0.4883)}{(s+(1.5796\text{x}10^4-j8.209\text{x}10^3))}$$

$$+ \frac{(6.3681\times10^4)\exp(-j0.4883)}{(s+(1.5796\times10^4+j8.209\times10^3))}$$

$$+ \frac{(1.4425\times10^4)\exp(j1.0571)}{(s+(1.0956\times10^4-j1.6828\times10^4)}$$

$$+ \frac{(1.4425\times10^4)\exp(-j1.0571)}{(s+(1.0956\times10^4+j1.6828\times10^4)}$$

Since $L^{-1}\{(s+a)^{-1}\} = \exp(-ar)$, then the impulse response is

$$\begin{aligned} h(r) = {} & (9.8412\times10^4)\exp-(1.7185\times10^4)r \\ & + (6.3681\times10^4)\exp(j0.4883)\exp(-(1.5796\times10^4-j8.209\times10^3)r) \\ & + (6.3681\times10^4)\exp(-j0.4883)\exp(-(1.5796\times10^4+j8.209\times10^3)r) \\ & + (1.4425\times10^4)\exp(j1.0571)\exp(-(1.0956\times10^4-j1.6828\times10^4)r) \\ & + (1.4425\times10^4)\exp(-j1.0571)\exp(-(1.0956\times10^4+j1.6828\times10^4)r) \end{aligned}$$

But since $\exp(ja)+\exp(-ja) = 2\mathrm{Cos}(a)$, the impulse response becomes

$$\begin{aligned} h(r) = {} & (9.8412\times10^4)\exp-(1.7185\times10^4)r \\ & + (6.3681\times10^4)\exp(-(1.5796\times10^4)r)2\mathrm{Cos}((8.209\times10^4)r+0.4883) \\ & + (1.4425\times10^4)\exp(-(1.0956\times10^4)r)2\mathrm{Cos}((1.6828\times10^4)r+1.0571) \end{aligned}$$

Therefore, the impulse response of a 5th. order low pass Bessel filter with a 3dB cutoff frequency of 750Hz. is

$$\begin{aligned} h(r) = {} & (9.8412\times10^4)\exp-(1.7185\times10^4)r \\ & + (1.2736\times10^5)\exp(-(1.5796\times10^4)r)\mathrm{Cos}((8.209\times10^4)r+0.4883) \\ & + (2.885\times10^4)\exp(-(1.0956\times10^4)r)\mathrm{Cos}((1.6828\times10^4)r+1.0571) \end{aligned}$$

and the envelope function is

$$h_e(r) = (9.8412x10^4)exp-(1.7185x10^4)r$$
$$+ (1.2736x10^5)exp(-(1.5796x10^4)r)$$
$$+ (2.885x10^4)exp(-(1.0956x10^4)r)$$

In order to calculate the order of the cellular filter which synthesises the 5th. order Bessel filter, the impulse response and the impulse response envelope are calculated at multiples of $\sqrt{2}$ times the resolution of the grid being used. In this case the resolution R = $8.4667x10^{-5}$m..

| nR | $\sqrt{2}$nR | h($\sqrt{2}$nR) | $h_e$($\sqrt{2}$nR) |
|---|---|---|---|
| 8R | 11.31370850R | -0.09077836 | 0.83978480 |
| 9R | 12.72792206R | 0.19838985 | 0.22113703 |
| 10R | 14.14213562R | -0.04136556 | 0.05882040 |
| 11R | 15.55634918R | -0.00516089 | 0.01573345 |
| 12R | 16.97056274R | 0.00419819 | 0.00422143 |
| 13R | 18.38477631R | -0.00058706 | 0.00113457 |
| 14R | 19.79898987R | -0.00016676 | 0.00030522 |
| 15R | 21.21320343R | 0.00008129 | 0.00008215 |
| 16R | 22.62741699R | -0.00000674 | 0.00002212 |

A resolution of 8 bits in image intensity gives a minimum image intensity value of 0.003906. The selection of the order of the cellular filter (n), as described in section 2.2, is to choose n such that it is the largest value for n where $|h(\sqrt{2}nR|> 0.003906$ and $h_e(\sqrt{2}(n+1)R) < 0.003906$. In this case n is 12.

### 3.2.1 Filter Coefficients Using Derived Equations

We have already determined that n = 12. Therefore, from equation (14), we have

$$K_2^n = h\ (nR,\ nR)$$

$$\Rightarrow K_2^{12} = h\ (12R,\ 12R) = 0.00419708$$

$$\Rightarrow K_2 = 0.63374$$

Using this value for $K_2$ in equation (13) gives

$$nK_1K_2^{n-1} = h\ ((n-1)R,\ nR)$$

$$\Rightarrow 12K_1(0.63374)^{11} = h\ (11R,\ 12R) = 0.00510807$$

$$\Rightarrow K_1 = \frac{0.00510807}{12(0.63374)^{11}} \approx 0.06428$$

Substituting the values for $K_1$ and $K_2$ into equation (12) gives

$$nK_0K_2^{n-1} + 2\left(\sum_{m=0}^{l-1} m\right)K_1{}^2K_2{}^{n-2} = h((n-1)R,\ (n-1)R)$$

$\Rightarrow 11K_0K_2^{11} + 2(66)K_1^2K_2^{10} = h(11R,\ 11R) = -0.00516913$

$\Rightarrow K_0(0.07947) + 0.0056995 = -0.00516913$

$$\Rightarrow K_0 = \frac{-0.0108686}{0.07947} = -0.13676$$

In summary,

$K_0 = -0.13676$

$K_1 = \ \ 0.06428$

$K_2 = \ \ 0.63374$

If we sum the elements of the kernel, we find that the sum is not unity.

$S_k = K_0 + 4K_1 + 4K_2 = 2.65532$

Because the sum of the filter kernel elements are not unity we will get a combination of amplification and inversion. This is avoided by normalising the kernel, such that the sum of the elements of the kernel is unity.

Therefore,

$K_{0N} = K_0/S_k = -0.0515$

$K_{1N} = K_1/S_k = \ \ 0.02421$

$K_{2N} = K_2/S_k = \ \ 0.23867$

and $n = 12$.

When this cellular filter was applied to test image the results were a isotropic low pass filter with a 3dB cutoff frequency of approximately 450Hz. (figure 5). The amplitude response at 900Hz. was approximately -13dB which is consistent with the response of a 5th. order low pass Bessel filter with a 3dB cutoff frequency of 450Hz. (figure 6).

## 3.2.2 Filter Coefficients Using Heuristic Search with $W_1(n)$

Using the heuristic search method, with weighting function $W_1(n)$, to determine the filter coefficients results in the following filter coefficients.

$K_0 = -0.40315$

$K_1 = -0.15082$

$K_2 = -0.00461$

which when normalised yields

$K_{0N} = 0.39337$

$K_{1N} = 0.14716$

$K_{2N} = 0.0045$

and $n = 12$.

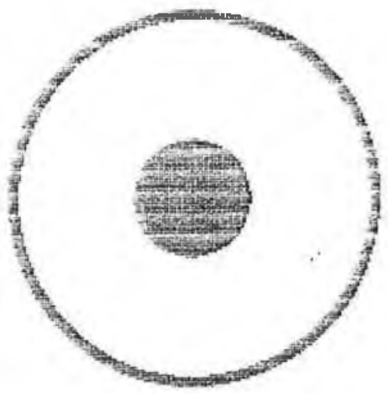


Figure 5: 750Hz Analytical Filter at -3dB

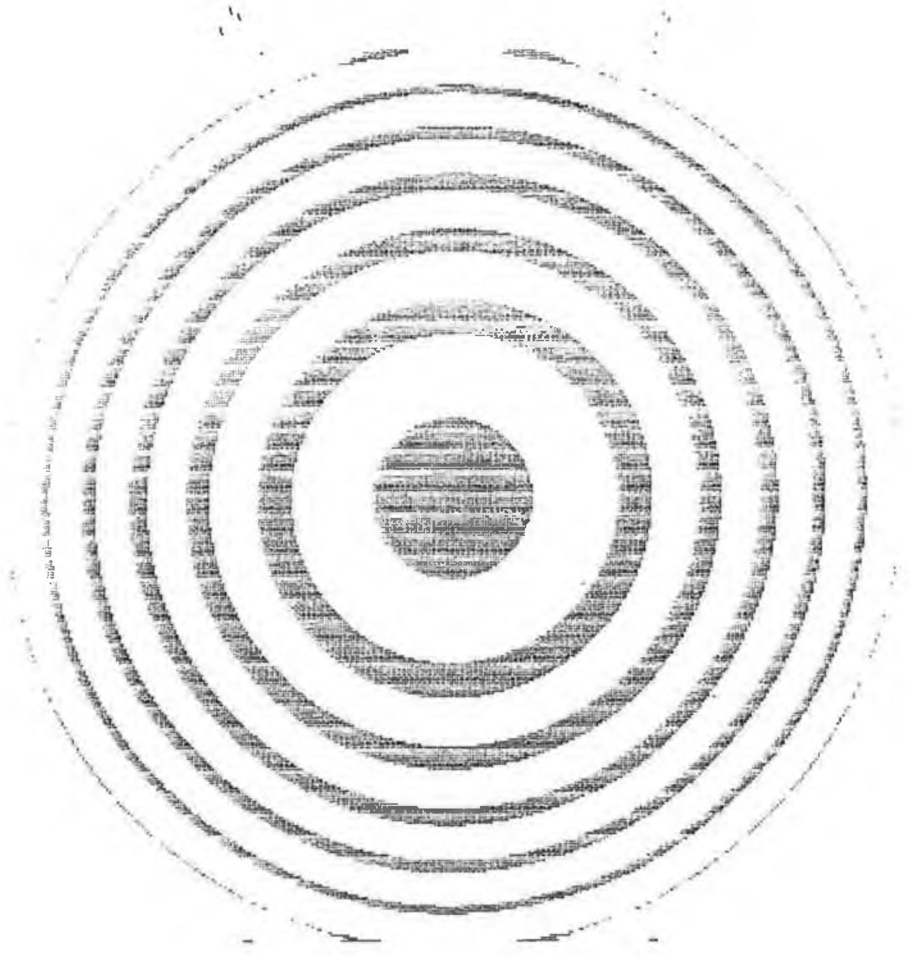


Figure 6: 750Hz. Analytical Filter at -13dB

When this cellular filter was applied to test image the results were an isotropic low pass filter with a 3dB cutoff frequency of approximately 750Hz. (figure 7). The amplitude response at 1500Hz. was approximately -13dB which is consistent with the response of a 5th. order low pass Bessel filter (figure 8).

### 3.2.3 Filter Coefficients Using Heuristic Search with $W_2(n)$

Using the heuristic search method, with weighting function $W_2(n)$, to determine the filter coefficients results in the following filter coefficients.

$K_0 = -0.0487$

$K_1 = 0.15081$

$K_2 = -0.03142$

which when normalised yields

$K_{0N} = -0.11356$

$K_{1N} = 0.35165$

$K_{2N} = -0.07326$

and n = 12.

When this cellular filter was applied to test image the results were an anisotropic filter (figure 9). At low frequencies, up to 1kHz. the behaviour was low pass in nature, but still anisotropic. At higher

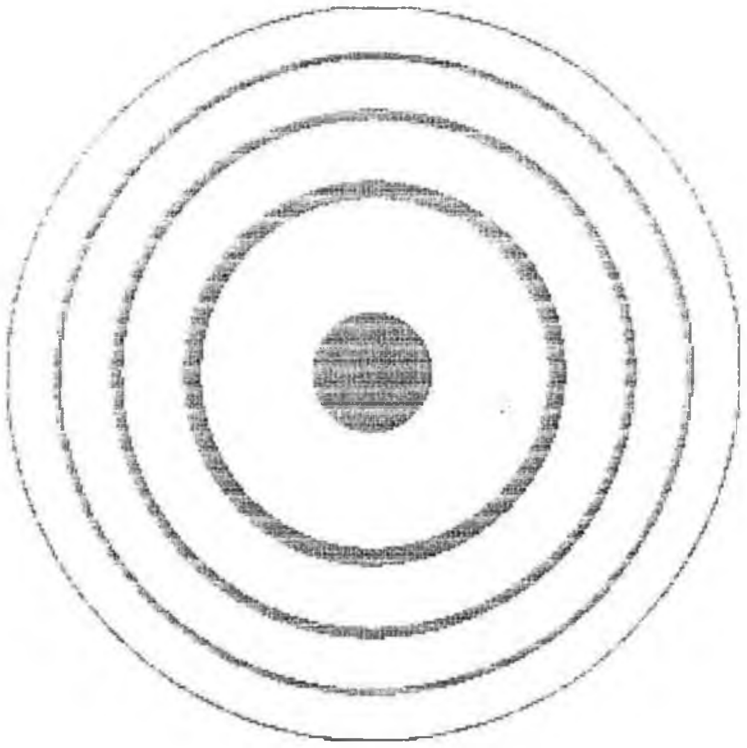


Figure 7: 750Hz W1 Heuristic Filter at -3dB

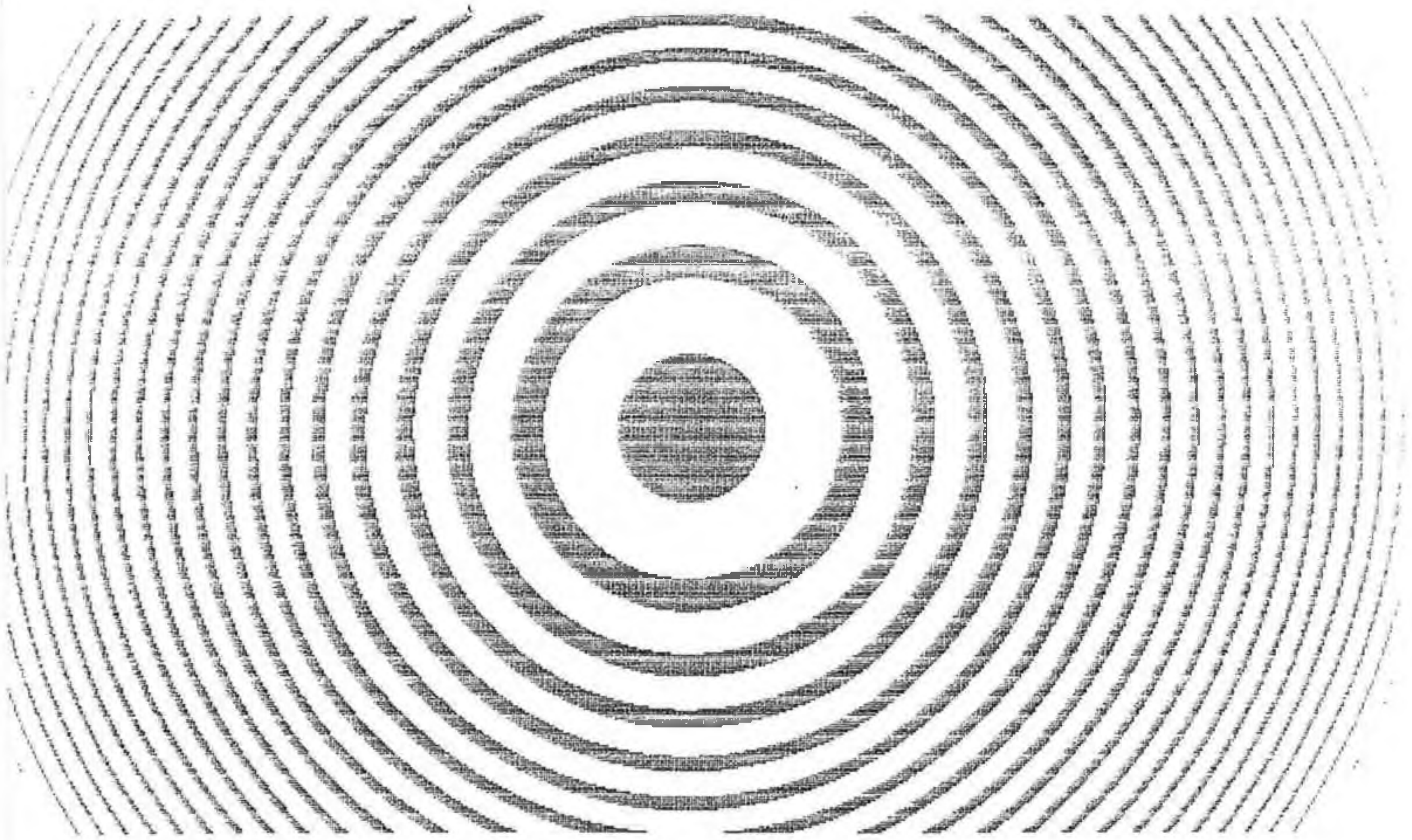


Figure 8: 750Hz W1 Heuristic Filter at -13dB

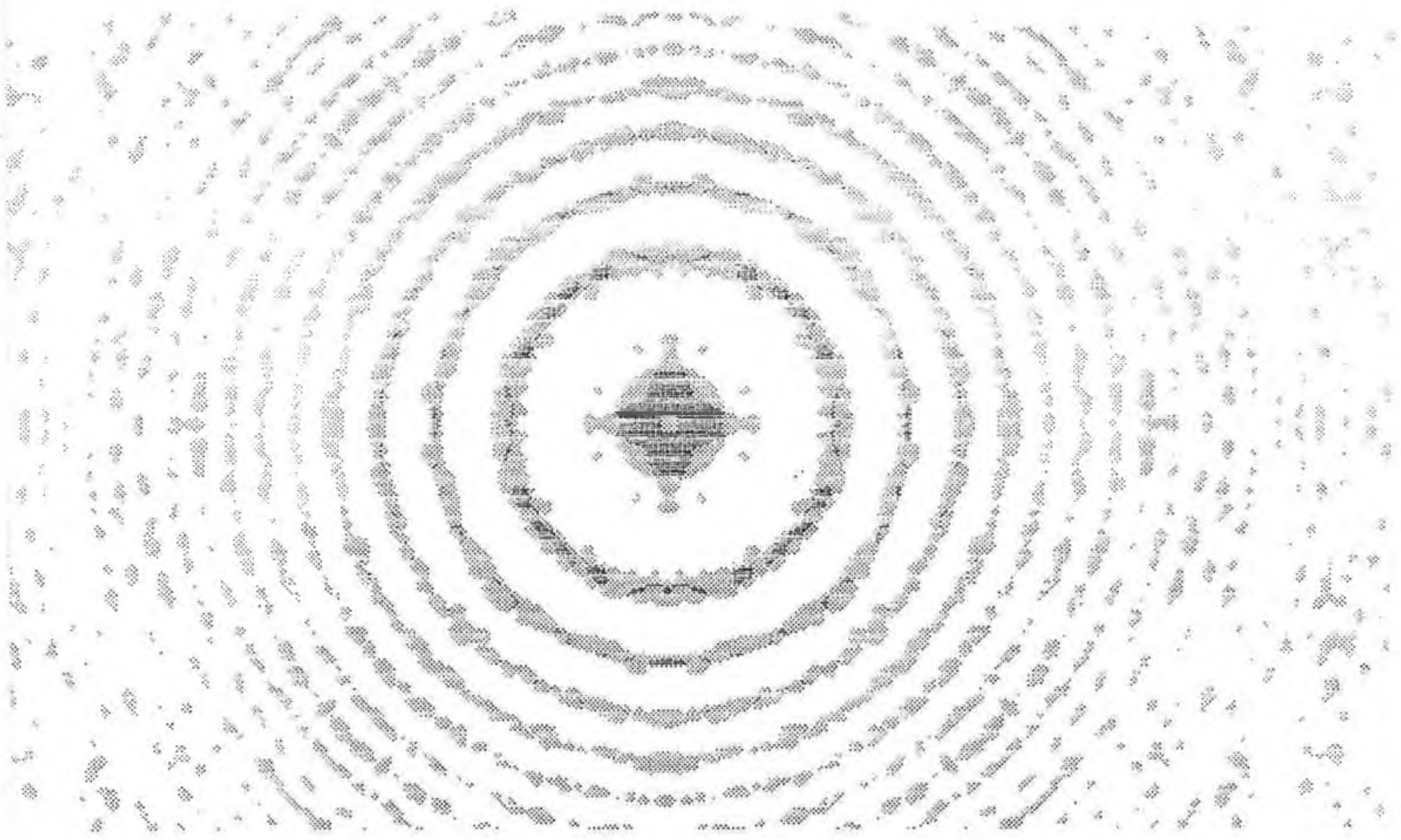


Figure 9: 750Hz W2 Heuristic Filter at -3dB

frequencies there was no discernible behaviour other than anisotropic behaviour.

### 3.2.4 Summary of Results

Deriving the cellular filter coefficients from the equations 12, 13 and 14 does not produce the expected results. This is due to the fact that we are trying to find a kernel which will match impulse response of the filter from only 3 points, far from the origin. If the procedure for deriving equations 12, 13 and 14 was applied to other points, producing equations for the response at further points then the response of the cellular filter may be closer to the expected response.

Using the heuristic search algorithm with weighting function $W_2(n)$ does not produce good results since $W_2(n)$ overcompensates for errors near the origin during the search.

Using the heuristic search algorithm with weighting function $W_1(n)$ produced excellent results. The filter has the correct cutoff frequency and the correct amplitude response.

## 3.3 Synthesis of a 2kHz. 5th. Order Low Pass Bessel Filter

From equation (16), section 3.2, the normalised transfer function for a 5th. order low pass Bessel filter is given by:

$$H(s_n) = 945\left\{ \frac{0.0221}{(s_n+3.6467)} + \frac{(0.0143)\exp(j0.4883)}{(s_n+(3.352-j1.742))} \right.$$

$$+\frac{(0.0143)\exp(-j0.4883)}{(s_n+(3.352+j1.742))} + \frac{(3.2394\text{x}10^{-3})\exp(j1.0571)}{(s_n+(2.325-j3.571))}$$

$$\left. + \frac{(3.2394\text{x}10^{-3})\exp(-j1.0571)}{(s_n+(2.325-j3.571))} \right\}$$

If the cutoff frequency is 2kHz., then $2\pi\omega_c = 1.2566\text{x}10^4$ rads/m. Applying the transform $s_n$ to $(s/\omega_c)$ to scale the 3dB. cutoff frequency to $f_c$ gives the transfer function of a 5th. order 500Hz. low pass Bessel filter as

$$H(s) = 945\left\{ \frac{(2.772\text{x}10^2)}{(s+4.5826\text{x}10^4)} \right.$$

$$+ \frac{(1.797\text{x}10^2)\exp(j0.4883)}{(s+(4.2122\text{x}10^4-j2.1891\text{x}10^4))}$$

$$+ \frac{(1.797\text{x}10^2)\exp(-j0.4883)}{(s+(4.2122\text{x}10^4+j2.1891\text{x}10^4))}$$

$$+ \frac{(4.0708\text{x}10^1)\exp(j1.0571)}{(s+(2.9217\text{x}10^4-j4.7136\text{x}10^4))}$$

$$\left. + \frac{(4.0708\text{x}10^1)\exp(-j1.0571)}{(s+(2.9217\text{x}10^4+j4.7136\text{x}10^4))} \right\}$$

$$\Rightarrow H(s) = \frac{(2.6245\text{x}10^5)}{(s+4.5826\text{x}10^4)}$$

$$+ \frac{(1.6982\text{x}10^5)\exp(j0.4883)}{(s+(4.2122\text{x}10^4-j2.1891\text{x}10^4))}$$

$$+ \frac{(1.6982\mathrm{x}10^5)\exp(-j0.4883)}{(s+(4.2122\mathrm{x}10^4+j2.1891\mathrm{x}10^4))}$$

$$+ \frac{(3.8469\mathrm{x}10^4)\exp(j1.0571)}{(s+(2.9217\mathrm{x}10^4-j4.7136\mathrm{x}10^4))}$$

$$+ \frac{(3.8469\mathrm{x}10^4)\exp(-j1.0571)}{(s+(2.9217\mathrm{x}10^4+j4.7136\mathrm{x}10^4))}$$

Using the inverse Laplace transform $L^{-1}\{(s+a)^{-1}\} = \exp(-ar)$, gives the impulse response as

$$\begin{aligned} h(r) = {} & (2.6245\mathrm{x}10^5)\exp(-(4.5826\mathrm{x}10^4)r) \\ & + (1.6982\mathrm{x}10^5)\exp(j0.4883)\exp(-(4.2122\mathrm{x}10^4-j2.1891\mathrm{x}10^4)r) \\ & + (1.6982\mathrm{x}10^5)\exp(-j0.4883)\exp(-(4.2122\mathrm{x}10^4+j2.1891\mathrm{x}10^4)r) \\ & + (3.8469\mathrm{x}10^4)\exp(j1.0571)\exp(-(2.9217\mathrm{x}10^4-j4.7136\mathrm{x}10^4)r) \\ & + (3.8469\mathrm{x}10^4)\exp(-j1.0571)\exp(-(2.9217\mathrm{x}10^4+j4.7136\mathrm{x}10^4)r) \end{aligned}$$

Since $\exp(ja)+\exp(-ja) = 2\mathrm{Cos}(a)$, the impulse response function of a 5th. order 2kHz. low pass Bessel filter is

$$\begin{aligned} h(r) = {} & (2.6245\mathrm{x}10^5)\exp(-(4.5826\mathrm{x}10^4)r) \\ & + (3.3964\mathrm{x}10^5)\exp(-(4.2122\mathrm{x}10^4))\mathrm{Cos}((2.1891\mathrm{x}10^4)r+0.4883) \\ & + (7.6938\mathrm{x}10^4)\exp(-(2.9217\mathrm{x}10^4))\mathrm{Cos}((4.7136\mathrm{x}10^4)r+1.0571) \end{aligned}$$

and the impulse response envelope function is

$$\begin{aligned} h_e(r) = {} & (2.6245\mathrm{x}10^5)\exp(-(4.5826\mathrm{x}10^4)r) \\ & + (3.3964\mathrm{x}10^5)\exp(-(4.2122\mathrm{x}10^4)) \\ & + (7.6938\mathrm{x}10^4)\exp(-(2.9217\mathrm{x}10^4)) \end{aligned}$$

To calculate the order of the cellular filter which we will use to synthesise the 5th. order 2kHz. low pass Bessel filter, we calculate the impulse response and the impulse response envelope at multiples of $\sqrt{2}$ times the resolution of the grid we are using. Since the resolution $R = 8.4667\text{x}10^{-5}$m., this gives:

| nR | $\sqrt{2}$nR | $h(\sqrt{2}nR)$ | $h_e(\sqrt{2}nR)$ |
|---|---|---|---|
| 2R | 2.82842712R | 85.19818548 | 89.01947656 |
| 3R | 4.24264069R | 1.36309284 | 2.23882155 |
| 4R | 5.65685425R | 0.00462502 | 0.06506179 |
| 5R | 7.07106781R | -0.00104613 | 0.00195191 |
| 6R | 8.48528137R | -0.00005509 | 0.00005894 |
| 7R | 9.89949493R | -0.00000171 | 0.00000178 |

A resolution of 8 bits in image intensity gives a minimum image intensity value of 0.003906. Therefore the selection of the order ,n , of the cellular filter is such that it is the largest value of n where $h(|h(\sqrt{2}nR)| > 0.003906$ and $h_e(\sqrt{2}(n+1)R) < 0.003906$. From the data in the table above the correct value for the order of the cellular filter is 4.

Using the heuristic search technique, with wieghting function $W_1(n)$, to calculate the cellular filter coefficients, the following coefficients were obtained.

$K_0$ = -0.18406

$K_1$ = -0.03752

$K_2$ = -0.00631

Since the sum of the kernel, $S_k$, is not unity (-0.35938), the kernel must be normalised to avoid unwanted inversion and attenuation. This gives a normalised kernel for a cellular filter to implement a 5th. order 2kHz. low pass Bessel filter of:

$K_{0N} = K_0/S_k = 0.51216$

$K_{1N} = K_1/S_k = 0.1044$

$K_{2N} = K_2/S_k = 0.01756$

and the order of the cellular filter n = 4.

When this cellular filter was applied to the test image it produced a low pass filter with an incorrect cutoff frequency which did not exhibit isotropic behaviour (figure 10).

This anisotropic behaviour and incorrect cutoff frequency highlight an important limitation of this method for synthesising 2-dimensional image filters by cellular automata. The central idea of this technique is find a cellular automaton, which after N iterations on the impulse signal, produces a result which is a sampled version of the desired filter's impulse response. The number of samples is equal to the number of iterations (order of the cellular filter) and the sampling rate of the impulse response is the resolution of the underlying grid. As in all sampling operations the Nyquist sampling criterion must be met for a set of samples to uniquely determine a

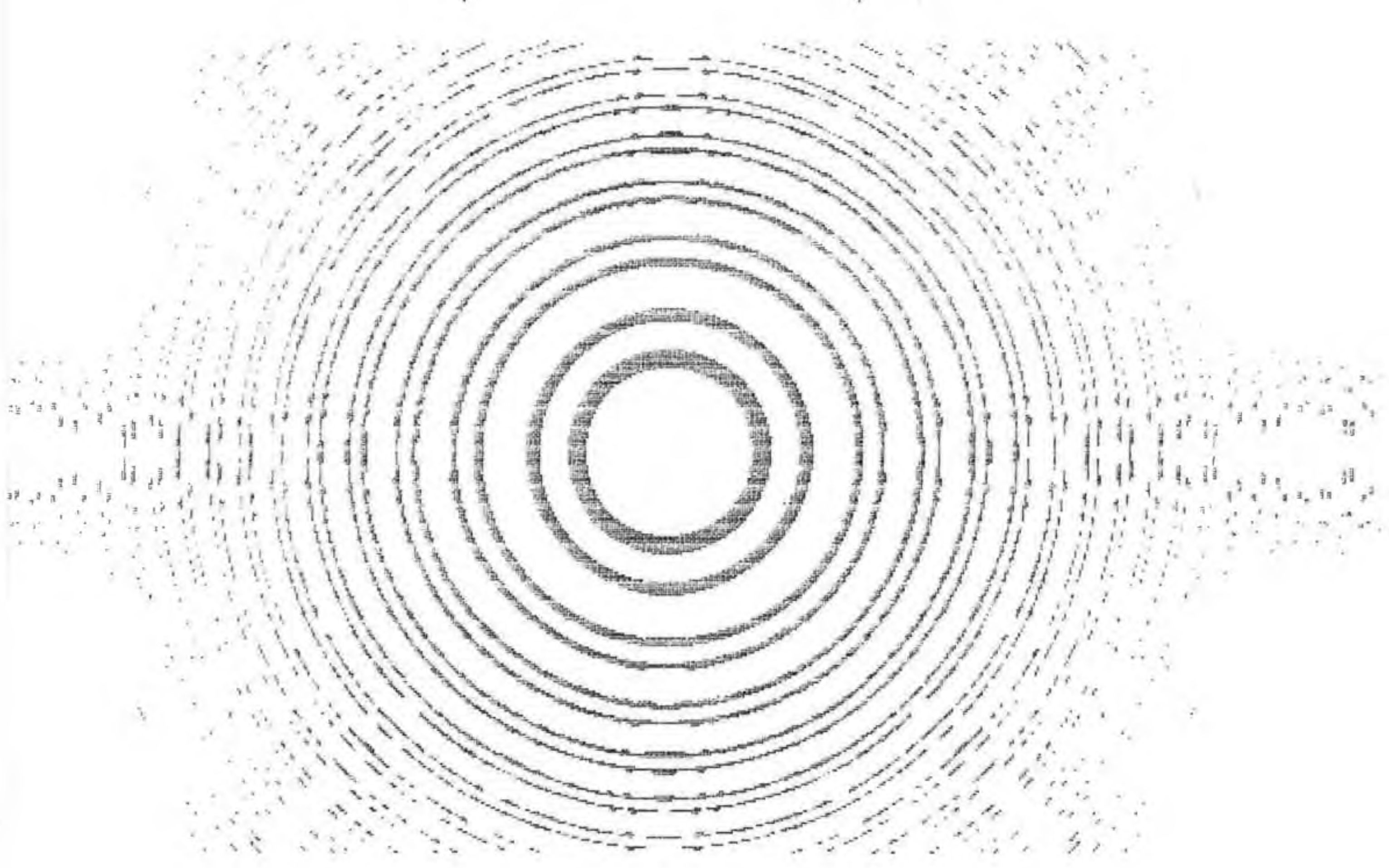


Figure 10: 2kHz W1 Heuristic Filter at -13dB

given function. The Nyquist sampling theorem states that in order for a set of samples to uniquely represent a given function, the sampling frequency must be greater than twice the highest frequency component in the function. In this technique the sampling rate is fixed by the resolution of the underlying grid and cannot be changed. Therefore, when the order of the cellular filter is low, not enough samples are obtained to uniquely represent the impulse response, and the cellular filter does not have the desired effects.

## 4.0 Synthesis of Other Filter Types

This technique for synthesising 2-dimensional image filters by cellular automata can be extended to other types of filters, such as high pass, band pass and band reject filters, as well as low pass filters. Since the basic idea of the technique is to find a cellular automaton which after N iterations on the impulse signal produces a sampled version of the impulse response of the filter, in order to synthesise other type of filters all that is needed is the transfer function of the filter to be synthesised. These transfer functions can be obtained by two methods:

(1) by consulting tables of transfer functions in references on filter design.

(2) by applying a frequency transform to a low pass filter transfer function

The use of frequency transformations on low pass filter transfer functions is a common way to derive the transfer functions of high pass, band pass, and other filters. Some of the common frequency transforms are:

Low Pass to High Pass transform

For a given low pass transfer function, the high pass transfer function can be obtained by substituting $s^{-1}$ for s in the transfer function.

Low Pass to Band Pass transform

For a given low pass transfer function, the equivalent band pass transfer function can be obtained by the transform

$$s \rightarrow \frac{s^2+\omega_0{}^2}{(\omega_{3dB})s} = \frac{Q(s_n{}^2+1)}{s_n}$$

where $\omega_{3dB}$ = 3dB pass band width

$\omega_0$ = the centre frequency defined by

$\omega_0 = \sqrt{\omega_u \omega_l}$ which approximates $\frac{\omega_u + \omega_l}{2}$ for $\frac{\omega_0}{\omega_{3dB}} >> 1$

Q = the quality factor $\frac{\omega_0}{\omega_{3dB}}$

and $s_n = \frac{s}{\omega_0}$

## Low Pass to Band Reject transform

For a given low pass transfer function, the equivalent band pass transfer function can be obtained by the transform

$$s \rightarrow \frac{(\omega_{3dB})s}{s^2+\omega_0{}^2} = \frac{s_n}{Q(s_n{}^2+1)}$$

where $\omega_0$, $\omega_{3dB}$, Q and $s_n$ have the same definitions as for the low pass to band pass frequency transform.

As an example of how such a frequency transform can be used in this technique to synthesise a band pass filter consider the transfer function of a 1st order low pass Bessel filter.

$$H(s) = \frac{1}{s+1}$$

In order to transform this into the transfer function of a 1st. order band pass Bessel filter with lower and upper 3dB cutoff frequencies of 400Hz and 800Hz. respectively, the following transform is applied

$$s \rightarrow \frac{s^2+\omega_0{}^2}{(\omega_{3dB})s}$$

where $\omega_0 = \sqrt{2\pi(2800)2\pi(400)} = 3.5543\text{x}10^3$

and $\omega_{3dB} = 800 - 400 = 400$

Hence the transfer function of such a band pass filter is

$$H(s) = \frac{(\omega_{3dB})s}{s^2+(\omega_{3dB})s+\omega_0{}^2} \qquad (17)$$

The inverse Laplace transform of $\frac{s+\alpha}{(s+\beta)^2+\gamma^2}$ is given by

$$L^{-1}\{\frac{s+\alpha}{(s+\beta)^2+\gamma^2}\} = \frac{1}{\gamma}\,[\sqrt{(\alpha-\beta)^2+\gamma^2}]\exp(-\alpha r)\ \mathrm{Sin}(\gamma r+\phi) \qquad (18)$$

where $\phi = \mathrm{Tan}^{-1}(\frac{\gamma}{\alpha-\beta})$

But $\frac{s+\alpha}{(s+\beta)^2+\gamma^2} = \frac{s+\alpha}{s^2+2\beta s+(\beta^2+\gamma^2)}$

and comparing this to equation (17) gives the following equalities

$\alpha = 0$

$\beta = \frac{\omega_{3dB}}{2}$

$\gamma^2 = \omega_0{}^2 - \frac{\omega_{3dB}{}^2}{4}$

and therefore, by equation (18), the impulse response of the 1st. order band pass Bessel filter is

$$L^{-1}\{H(s)\} = h(r) = \frac{1}{\gamma}\,[\sqrt{(\alpha-\beta)^2+\gamma^2}]\exp(-\alpha r)\ \mathrm{Sin}(\gamma r+\phi)$$

$$\Rightarrow h(r) = \frac{\omega_{3dB}}{\sqrt{\omega_0{}^2-\frac{\omega_{3dB}{}^2}{4}}}\,\omega_0\ \mathrm{Sin}(\sqrt{\omega_0{}^2-\frac{\omega_{3dB}{}^2}{4}}\ r + \phi))$$

where $\phi = \mathrm{Tan}^{-1}\left(-\dfrac{\sqrt{\omega_0{}^2 - \dfrac{\omega_{3dB}{}^2}{4}}}{\dfrac{\omega_{3dB}}{2}}\right)$

$$\Rightarrow h(r) = (4.0063x10^2)\mathrm{Sin}((3.5487x10^3)r-1.5145)$$

Since this is a 1st. order band pass transfer function, it has no negative exponential term. Therefore the method for selecting the order of the cellular filter is no longer valid since there is no value of $n=n_o$ such that $|h(\sqrt{2}nR)|$ < minimum image intensity for $n>n_o$. However the frequency of h(r) is

$$3.5487x10^3 \text{ rads/m} = 2\pi f$$

$$\Rightarrow \quad f = 564.79 \text{ Hz.}$$

On a grid of 300 dpi ($8.4667x10^{-5}$m) the number of samples needed to characterise a full cycle is

$$((8.4667x10^{-5})(564.79))^{-1} = 20.91$$

Therefore the order of the cellular filter needed to synthesise the 1st. order band pass Bessel filter with lower and upper 3dB cutoff frequencies of 400Hz. and 800Hz. respectively is 21. The coefficients of the cellular filter, i.e. the elements of the cellular automaton can now be found by heuristic search.

## 5.0 Parallel Implementation of Cellular Filters

Cellular automata are an example of a highly symmetrical data-parallel computational system, in which the data decomposition between computational nodes is congruent. Within a given iteration, one cellular automaton does not depend on any other cellular automaton in order to produce its next state. All the cellular automata are independent of each other, and all of them can operate in parallel. This is the main advantage of using cellular filters for filtering 2-dimensional images. Given a network of processors, a set of cellular automata can be assigned to each processor. Since the cellular automata have a congruent data decomposition, by adding more processors, the number of cellular automata assigned to each processor can be reduced linearly, and therefore the time taken to filter an image can be reduced approximately linearly. Since these processors need only be capable of floating-point addition and multiplication to a limited accuracy (8 bits in a system with 256 shades of grey), this allows for real time image filtering at relatively low costs.

The occam programs in Appendices C and D are just one possible parallel implementation of cellular filters. Appendix C contains a listing of netfilt.tsr, an occam program designed to run on a network of four transputers connected in a pipeline. Appendix D contains a listing of host.tsr, an occam program which runs on the host transputer which can access the DOS file structure.

Each network transputer has two parallel processes running on it, a data.manager process and an image.filter process. The image.filter process receives a portion of the overall image, together with the elements of the cellular automaton which will be used to synthesise the desired filter, from the host transputer via the network of data.manager processes on the different transputers. It then starts M parallel processes, where M is the number of pixels in the image portion. Each process is a cellular automaton. While there is a slight performance disadvantage in implementing the M cellular automata on a single transputer in parallel rather then sequentially, it does emphasise the inherent parallel nature of cellular automata. After image.filter has performed one iteration of the cellular automata, the image portion is sent back the host transputer via the network of data.manager processes.

The data.manager process acts as a multiplexer for its associated image.filter process. The messages that the data.manager process receives from the image.filter process are reframed to include a node identification number. The messages are then sent to the host transputer via the data.manager process on the next transputer upstream on the pipeline of transputers. Messages which the data.manager process receives from the data.manager process on the next transputer downstream on the pipeline of transputers are passed on to the upstream data.manager process unaltered. When the data.manager process receives messages from the host transputer via the upstream data.manager processes, it checks the node identification number to determine if the message is destined for the image.filter process associated with it. If the message is destines for the image.filter process running on the same transputer as the data.manager process, the node identification number is stripped off and the message is passed on to the image.filter process. For all messages, except the *quit* message, if the data.manager process finds that the node identification number does not match this node's identification number, the message is passed on to the next downstream data.manager process. *Quit* messages are handled differently. If the *quit* message is destined for this node, the data.manager process passes the *quit* message on to the image.filter process, which then terminates. The data.manager process then checks to see if all downstream nodes have received *quit* messages. If they have, the data.manager process then terminates itself. If all the downstream nodes have not received a *quit* message, it records itself as terminated, but does not actually terminate itself just then. The data.manager process stays active. If the message is destined for another node, prior to being passed on downstream, the node the *quit* message is destined for is recorded as terminating by this node. After the message has been passed on downstream the data.manager process then checks to see if all downstream nodes, including itself, have received *quit* messages. If all downstream nodes and this node have received *quit* messages the data.manager process will terminate itself.

The process running on the host transputer acts as a file manager, interfacing the network of image.filter processes to the user and to the DOS file system. After it has determined, from the user, the file containing the image to be processed, and the coefficients and order of the cellular filter, the host process initialises the network of data.manager processes and kick starts the network of image.filter processes by sending each image.filter process a *request.something* message. Each image.filter process will then request a portion of the image to which it will apply one iteration of the cellular filter. After an image.filter process has processed the image portion, the image portion is sent back to the host process for filing. The host process will send an acknowledgement to the appropriate image.filter process to signal the image portion has been saved. The image.filter process will then request another image portion to process. When an image.filter process requests another image portion, and there is no more image portions left to be processed in this iteration of the cellular filter, the host process sends a *hold* message to the image.filter process. When the whole image has received an iteration of the cellular filter, another iteration is started by the host process sending *request.something* messages to the network of image.filter processes. This sequence of events continues until the required number of iterations has been carried out. The host process then shuts the transputer network down by sending *quit* messages to each data.manager process.

The following is the message structure between the host process and the data.manager process, and the data.manager process and the image.filter process.

Host process messages transmitted

| message name | parameters | final destination | response |
|---|---|---|---|
| initialise | node id | data.manager | none |
| quit | node id | image.filter | none |
| hold | node id | image.filter | none |
| request.something | node id | image.filter | request.image |
| filter.param | node id,<br>subimage id,<br>filter coefficients | image.filter | none |

| | | | |
|---|---|---|---|
| image.row | node id, subimage id, row of image | image.filter | none |
| image.saved | node id | image.filter | none |

Host process messages received

| message_name | parameters | initial source | response |
|---|---|---|---|
| request.image | node id | image.filter | filter.param image.row |
| image.filtered | node id subimage id | image.filter | none |
| filtered.row | node id, subimage id, filtered row of image | image.filter | image.saved |

The main disadvantage to this implementation of a cellular filter is the bottleneck introduced by the host process accessing the DOS file system. In a commercial implementation using a stand alone network of transputers, the host transputer would have enough memory to hold the entire image in memory, and the memory on each transputer in the network would only have enough memory to meet its requirements. Then, provided that the number of transputers in the network is an factor of the number of subimages in the total image, then the load balance is equal amoung all the transputers, and the performance of the filter increases linearly with the number of transputers.

## 6.0 Conclusions

The synthesis of a given 2-dimensional grey level image filter by cellular automata can be successfully carried out using the technique described in this thesis. The basic goal of the technique is to find a cellular automaton which when replicated M times (where M is the size of the image, and each cellular automaton is assigned to each pixel in the image) will produce a sampled version of the impulse response of the image filter to be synthesised, after N iterations on the impulse signal.

Two techniques for finding the kernel of such a cellular automaton were investigated, an analytical evaluation of the elements of the kernel and a heuristic search for the elements of the kernel.

The analytical technique did not produce good results, but this was due to the fact that only three points, at a distance from the origin of the impulse signal, were used to calculate the kernel. For this analytical technique to produce accurate results, further research is necessary to produce a general equation which describes the response, after N iteration, of a set of cellular automata using a given kernel at any point in the resulting signal array.

The heuristic search for the elements of the kernel was investigated using two different wieghting functions for the error between the impulse response and the kernel after N iterations. Wieghting function $W_2(n)$ was designed to mimic the analytical technique, by concentrating on matching the response of the kernel after N iteration on the impulse signal, to the impulse response at points distant from the origin of the impulse signal. Like the analytical technique, a heuristic search using wieghting function $W_2(n)$ does not produce good results. Wieghting function $W_1(n)$ was designed to give a closer fit to the impulse response at all points, while also allowing for misleading effects due to approximating the impulse signal. Heuristic searches using wieghting function $W_1(n)$ produced excellent results with the correct cutoff frequencies and accurate amplitude response characteristics.

Like all systems which seek to represent reality by a digitally sampled representation, this thesis has shown that cellular filters are also subject to the Nyquist sampling criterion. This limitation means that low order cellular filters are subject to aliasing effects and do not produce isotropic results.

In summary, the filtering behaviour of cellular automata on binary images has been known since the mid 1960's. In the late 1970's researchers, such as Preston and Duff, extended the use of cellular automata to the filtering of grey level images, by converting the grey level image into a series of binary images, applying cellular automata to these binary images and then reconstituting them back into a grey level image. These cellular filters were found empirically, and while the cutoff frequencies of these low pass filters could be altered, the response characteristics were fixed. In recent years, other researchers, such as Huang, Jenkins and Sawchuk, while working on digital cellular image processors and their associated algebra, have commented on the filtering characteristics on certain cellular automaton which have been found empirically. To date , no method has existed for deriving the kernel of a cellular filter which will have characteristics defined by the user. The technique described in this thesis, based in the formal methods of linear systems analysis, enables the user to successfully derive the kernel of a cellular filter to match the behaviour of a known filter described by its Laplacian transfer function.

## 7.0 References

[1] "Modern Cellular Automata - Theory and Applications"; K. Preston and M.J.B. Duff; Plenum; 1984.

[2] "Binary Image Algebra and Optical Cellular Logic Processor Design"; K.S. Huang, B.K. Jenkins and A.A. Sawchuk; Computer Vision, Graphics and Image Processing vol. 45 no. 3 p295-345; 1989.

[3] "Introduction to Linear System Analysis"; Matrix Publishers; 1976.

[4] "Modern Control Systems"; R.C. Dorf; Addison-Wesley Publishing Co.; 1980.

[5] "Manual of Active Filter Design"; J.L. Hilburn and D.E. Johnson; McGraw-Hill; 1973.

[6] "Filter Design Tables and Graphs"; E. Christian and F. Eisenman; 2nd. Ed.; Wiley; 1966.

[7] "Introduction to the Theory and Design of Active Filters"; L.P. Huelsman and P.E. Allen; McGraw-Hill; 1980.

## Appendix A

This appendix contains a listing of the C program, find_ks.c, which implements the heuristic search algorithm for the coefficients of the cellular filter which synthesises the filter whose impulse response is contained in the procedure impulse_response.

```
/****************************************************************************/
/*                                                                          */
/*       Filename:   find_ks.c                        Revision:   1.0       */
/*       Author:     David Sinclair                   Last Edit:  05Apr91   */
/*                                                                          */
/*       This program locates the coefficients of the kernal which has      */
/*       the minimum mean square error from the impulse response for the    */
/*       filter whose impulse response is defined by impulse_response ().   */
/*                                                                          */
/*       Command line (Microsoft compiler): cl /W3 response.c               */
/*                                                                          */
/****************************************************************************/
/*                                                                          */
/*                             Version History                              */
/*                                                                          */
/*       Revision                          Notes                            */
/*       --------                          -----                            */
/*                                                                          */
/*         1.00              Initial Development Version                    */
/*                                                                          */
/****************************************************************************/

/****************************** header  files ******************************/

£include "stdio.h"
£include "stdlib.h"
£include "conio.h"
£include "math.h"
£include "dos.h"

/**************************** declaration files ****************************/

/****************************** definitions ********************************/

/* ASCII characters */

£define   SPACE                    0x20


/* ASCII graphics characters */

£define  LEFT_TOP_COURNER          201
£define  RIGHT_TOP_COURNER         187
£define  LEFT_BOTTOM_COURNER       200
£define  RIGHT_BOTTOM_COURNER      188
£define  HORIZONTAL_LINE           205
£define  VERTICAL_LINE             186


/* screen dimensions, counting from 0 */

£define  SCREEN_WIDTH   79
£define  SCREEN_DEPTH   23


/* miscellaneous */

£define EPSILON                 (double) 1.0e-08
```

```
£define TRUE                0
£define FALSE               -1

£define SWEEP_TABLE_SIZE    11

£define MAX_ERROR           1.0e+300

/* response size definitions */

£define ROOT2               (double) 1.414213562e+00
£define RESOLUTION          (double) 8.4667e-05

£define RESPONSE_WIDTH   31
£define RESPONSE_DEPTH   31


/* BIOS video services */

£define VIDEO               0x10

£define SET_CURSOR_POS      0x02
£define READ_CURSOR_POS     0x03
£define DOS_SCROLL_WINDOW_UP 0x06
£define CURRENT_VIDEO_STATE  0x0f


/* DOS colours */

£define BLACK               0x00

/**************************** global  variables ****************************/

/**************************** module  variables ****************************/

int wieght_fn;

float filter_k0;
float filter_k1;
float filter_k2;

float delta_fn;

float response [RESPONSE_DEPTH+2][RESPONSE_WIDTH+2];
float resultant [RESPONSE_DEPTH+2][RESPONSE_WIDTH+2];

float step_table [SWEEP_TABLE_SIZE] = {1.0f, 5.0e-01f, 0.25f, 0.1f, 0.01f,
                                      0.001f, 0.0001f, 0.00001f, 0.000001f,
                                      0.0000001f, 0.00000001f};

/********************************* modules *********************************/

int main ()
{
   /* declare local variables */

   int iterations, sweep, end_sweep, precision, pass;
   int quit_flag = FALSE;
int low_resolution = TRUE;
float k0_at_min, k1_at_min, k2_at_min, step;
float best_k0, best_k1, best_k2;
float max_extent_k0, max_extent_k1, max_extent_k2;
float low_extent_k0, low_extent_k1, low_extent_k2;
float low_range, high_range;
double error, minimum_error, last_minimum_error;

/* function prototypes */

double root_mean_square_error (float, float, float, int);
double impulse_response (double);
float calculate_step (unsigned int);
void clear_screen (void);
void set_cursor_position (unsigned int, unsigned int);
void draw_border (void);
void filter_calc_data (char *, float, float);
void print_results (float, float, float, int, double);
void show_mean_square_error (double, double, float);
void get_filter_data (int *, int *, float *,  float *, float *, float *,
                      float *, float *, int *);


/* get cellular filter data boundaries */

get_filter_data (&iterations, &precision, &low_extent_k0, &max_extent_k0,
                 &low_extent_k1, &max_extent_k1, &low_extent_k2,
                 &max_extent_k2, &wieght_fn);

/* initialise variables */

delta_fn = (float) impulse_response ((double) 0.0);

best_k0 = low_extent_k0;
best_k1 = low_extent_k1;
best_k2 = low_extent_k2;

pass = 0;

end_sweep = 4;

filter_k0 = low_extent_k0;
filter_k1 = low_extent_k1;

last_minimum_error = MAX_ERROR;
step = 1.0f;
error = MAX_ERROR;



do
   {
   pass++;

   minimum_error = MAX_ERROR;
   filter_k2 = low_extent_k2;
   k2_at_min = filter_k2;
```

```
      /* find minimum root mean square error K2 with K0 and K1 constant */

      high_range = max_extent_k2;
      low_range = low_extent_k2;

      for (sweep = 1; sweep <= end_sweep; sweep++)
         {
         for (; filter_k2 <= high_range; filter_k2 += step)
            {
            /* calculate root mean square error between kernal */
            /* and impulse response */

            filter_calc_data ("K2", low_range, high_range);
            if (filter_k2 != low_range)
               show_mean_square_error (error, minimum_error, k2_at_min);

            set_cursor_position (20,15);
            printf ("Pass %s%d", (end_sweep == 4 ? "pre-": ""), pass);

            error = root_mean_square_error (filter_k0, filter_k1, filter_k2,
iterations);

            if (error < minimum_error)
               {
               minimum_error = error;
               k2_at_min = filter_k2;
               }

            step = calculate_step (sweep);
            }

         low_range = k2_at_min - step;
         filter_k2 = low_range;
         high_range = k2_at_min + step;
         }

      filter_k2 =k2_at_min;


      minimum_error = MAX_ERROR;
      filter_k1 = low_extent_k1;
      k1_at_min = filter_k1;

      /* find minimum root mean square error K1 with K0 and K2 constant */

      high_range = max_extent_k1;
      low_range = low_extent_k1;

      for (sweep = 1; sweep <= end_sweep; sweep++)
         {
         for (; filter_k1 <= high_range; filter_k1 += step)
            {
            /* calculate root mean square error between kernal */
            /* and impulse response */

            filter_calc_data ("K1", low_range, high_range);
            if (filter_k1 != low_range)
               show_mean_square_error (error, minimum_error, k1_at_min);
            set_cursor_position (20,15);
            printf ("Pass %s%d", (end_sweep == 4 ? "pre-": ""), pass);

            error = root_mean_square_error (filter_k0, filter_k1, filter_k2,
iterations);

            if (error < minimum_error)
               {
               minimum_error = error;
               k1_at_min = filter_k1;
               }

            step = calculate_step (sweep);
            }

         low_range = k1_at_min - step;
         filter_k1 = low_range;
         high_range = k1_at_min + step;
         }

      filter_k1 =k1_at_min;


      minimum_error = MAX_ERROR;
      filter_k0 = low_extent_k0;
      k0_at_min = filter_k0;

      /* find minimum root mean square error K0 with K1 and K2 constant */

      high_range = max_extent_k0;
      low_range = low_extent_k0;

      for (sweep = 1; sweep <= end_sweep; sweep++)
         {
         for (; filter_k0 <= high_range; filter_k0 += step)
            {
            /* calculate root mean square error between kernal */
            /* and impulse response */

            filter_calc_data ("K0", low_range, high_range);
            if (filter_k0 != low_range)
               show_mean_square_error (error, minimum_error, k0_at_min);

            set_cursor_position (20,15);
            printf ("Pass %s%d", (end_sweep == 4 ? "pre-": ""), pass);

            error = root_mean_square_error (filter_k0, filter_k1, filter_k2,
iterations);

            if (error < minimum_error)
               {
               minimum_error = error;
               k0_at_min = filter_k0;
               }

            step = calculate_step (sweep);
            }
         low_range = k0_at_min - step;
         filter_k0 = low_range;
         high_range = k0_at_min + step;
         }

      filter_k0 =k0_at_min;

                                       /* are we within 1% of last minumum error */
      if (minimum_error < (last_minimum_error * (double) 0.99))
         {
         last_minimum_error = minimum_error;
         best_k0 = filter_k0;
         best_k1 = filter_k1;
         best_k2 = filter_k2;
         }
      else
         {
         if (low_resolution == TRUE)
            {
            low_resolution = FALSE;
            end_sweep = precision + 3;
            pass = 0;
            }
         else
            quit_flag = TRUE;
         }

      }
   while (quit_flag == FALSE);

   /* print out the results */

   print_results (best_k0, best_k1, best_k2, pass, minimum_error);

   return (0);
}


/***************************************************************************/
/* This routine returns a measure of the root mean square error between    */
/* the impulse response defined by impulse_response () and the cellular    */
/* filter defined by K0, K1, K2 and N.                                     */

double root_mean_square_error (float k0, float k1, float k2, int n)
{
   int i, l, pass, x, y;
   float *p, *r;
   double actual, calculated, error;
   double error_squared_sum;

   double impulse_response (double);
   double weight (int);


   /* initialise response with delta function */

   for (l = (RESPONSE_WIDTH+2)*(RESPONSE_DEPTH+2), p = (float *) response;
        l != 0; l--, p++)
        *p = 0.0f;                      /* clear out response buffer */
   response [(RESPONSE_WIDTH+2)/2] [(RESPONSE_DEPTH+2)/2] = delta_fn;

   for (l = (RESPONSE_WIDTH+2)*(RESPONSE_DEPTH+2), p = (float *) resultant;
        l != 0; l--, p++)
        *p = 0.0f;                     /* clear out resultant buffer */


   /* apply filter to delta function */

   pass = 1;
   while (pass++ <= n)
      {

      for (y = 1; y <= ((RESPONSE_WIDTH+2)/2); y++)
         {
         for (x = y; x <= ((RESPONSE_WIDTH+2)/2); x++)
            {
            p = &(response [x][y]);
            r = &(resultant [x][y]);

            *r = (*p * k0)
                  + (*(p-(1)) * k1)
                  + (*(p+(1)) * k1)
                  + (*(p-(((RESPONSE_WIDTH+2)+1))) * k2)
                  + (*(p-((RESPONSE_WIDTH+2))) * k1)
                  + (*(p-(((RESPONSE_WIDTH+2)-1))) * k2)
                  + (*(p+(((RESPONSE_WIDTH+2)-1))) * k2)
                  + (*(p+((RESPONSE_WIDTH+2))) * k1)
                  + (*(p+(((RESPONSE_WIDTH+2)+1))) * k2);


            resultant [y][x] = resultant [x][y];
            resultant [((RESPONSE_WIDTH+2)-1) - x][y] = resultant [x][y];
            resultant [y][((RESPONSE_WIDTH+2)-1) - x] = resultant [x][y];
            resultant [x][((RESPONSE_WIDTH+2)-1) - y] = resultant [x][y];
            resultant [((RESPONSE_WIDTH+2)-1) - y][x] = resultant [x][y];
            resultant [((RESPONSE_WIDTH+2)-1) - x][((RESPONSE_WIDTH+2)-1) -
y] = resultant [x][y];
            resultant [((RESPONSE_WIDTH+2)-1) - y][((RESPONSE_WIDTH+2)-1) -
x] = resultant [x][y];
            }
         }

         /* copy resultant to response */

      for (p = (float *) response, r = (float *) resultant, i = 0;
           i < (RESPONSE_WIDTH+2)*(RESPONSE_DEPTH+2); i++)
               *(p++) = *(r++);
      }

   error_squared_sum = (double) 0.0;

   for (i = 1; i <= n; i++)
      {
      actual = impulse_response ((double) i * RESOLUTION);
      calculated = (double) response
[(RESPONSE_DEPTH+2)/2][((RESPONSE_WIDTH+2)/2)+i];
      error = (actual - calculated) * weight (i);
      error_squared_sum += error * error;

      actual = impulse_response ((double) i * ROOT2 * RESOLUTION);
      calculated = (double) response
[((RESPONSE_DEPTH+2)/2)+i][((RESPONSE_WIDTH+2)/2)+i];
      error = (actual - calculated) * weight ((int) ((double) i * ROOT2));
      error_squared_sum += error * error;
      }

   error_squared_sum = error_squared_sum / (double) (2 * n);

   return (sqrt (error_squared_sum));
}


/* This routine returns the interval to the next filter coefficient to be  */
/* checked.                                                                */

float calculate_step (unsigned int sweep_number)
{
   return (step_table [sweep_number - 1]);
}


/* This routine calculates the weighting factor for the error depending on */
/* its distance from the centre.                                           */

double weight (int distance)
{
   switch (distance)
      {
      case 0:
         if (wieght_fn == 1)
            return ((double) 0.5);
         else
            return ((double) 0.0);
         break;

      case 1:
         if (wieght_fn == 1)
            return ((double) 0.9);
         else
            return ((double) 0.01);
         break;

      case 2:
         if (wieght_fn == 1)
            return ((double) 1.0);
         else
            return ((double) 0.1);
         break;

      case 3:
         if (wieght_fn == 1)
            return ((double) 1.0);
         else
            return ((double) 0.5);
         break;

      case 4:
         if (wieght_fn == 1)
            return ((double) 1.0);
         else
            return ((double) 0.9);
         break;

      default:
         return ((double) 1.0);
         break;
      }
}

/* This routine returns the value of the impulse response for a specified  */
/* argument.                                                               */

double impulse_response (double arg)
{
   double answer;

   /* 500Hz 5th order low pass Bessel filter */
   answer = (double) 6.561e+04 * exp((double) -1.1456e+04 * arg)
            + (double) 8.4908e+04 * exp((double) -1.0531e+04 * arg) *
cos(((double) 5.4727e+03 * arg) + (double) 0.4883)
            + (double) 1.9235e+04 * exp((double) -7.3042e+03 * arg) *
cos(((double) 1.1219e+04 * arg) + (double) 1.0571);

   return (answer);
}

/****************************************************************************/
/* void draw_border (void)                                                  */
/*                                                                          */
/* This routine draws a border around the screen.                           */

void draw_border (void)
{
   int i;
   void set_cursor_position (unsigned int, unsigned int);


   set_cursor_position (0,0);
   putchar (LEFT_TOP_COURNER);

   set_cursor_position (SCREEN_WIDTH,0);
   putchar (RIGHT_TOP_COURNER);

   set_cursor_position (0,SCREEN_DEPTH);
   putchar (LEFT_BOTTOM_COURNER);

   set_cursor_position (SCREEN_WIDTH,SCREEN_DEPTH);
   putchar (RIGHT_BOTTOM_COURNER);

   set_cursor_position (1,0);
   for (i = 1; i < SCREEN_WIDTH; i++)
      putchar (HORIZONTAL_LINE);

   set_cursor_position (1,SCREEN_DEPTH);
   for (i = 1; i < SCREEN_WIDTH; i++)
      putchar (HORIZONTAL_LINE);

   set_cursor_position (0,1);
   for (i = 1; i < SCREEN_DEPTH; i++)
      {
      set_cursor_position (0,i);
      putchar (VERTICAL_LINE);
      }

   set_cursor_position (SCREEN_WIDTH,1);
   for (i = 1; i < SCREEN_DEPTH; i++)
      {
      set_cursor_position (SCREEN_WIDTH,i);
      putchar (VERTICAL_LINE);
      }

   return;
}


/* void clear_entry (unsigned int, unsigned int, unsigned int)            */
/*                                                                        */
/* This routine clears the screen with 31 spaces at the specified position */
/* and set the cursor position to the specified location.                 */

void clear_entry (unsigned int x, unsigned int y, unsigned int repeat)
{
   unsigned int i;

   void set_cursor_position (unsigned int, unsigned int);

   set_cursor_position (x,y);
   for (i = 0; i < repeat; i++)
      putchar (SPACE);
   set_cursor_position (x,y);

   return;
}


/* void clear_screen (void)                                               */
/*                                                                        */
/* This routine clears the entire screen.                                 */

void clear_screen (void)
{
   void up_scroll_region (unsigned char, unsigned char, unsigned char,
                          unsigned char, unsigned char, unsigned char);

   up_scroll_region (0,0,79,23,0,BLACK);

   return;
}
```

```
/* void up_scroll_region (unsigned char, unsigned char, unsigned char,       */
/*                         unsigned char, unsigned char, unsigned char)      */
/*                                                                           */
/* This routine uses DOS to clear a region of the screen.                    */

void up_scroll_region (unsigned char x1, unsigned char y1, unsigned char x2,
                       unsigned char y2, unsigned char up_scroll,
                       unsigned char attribute)
{
   union REGS reg86;

   reg86.h.al = up_scroll;
   reg86.h.cl = x1;
   reg86.h.ch = y1;
   reg86.h.dl = x2;
   reg86.h.dh = y2;
   reg86.h.bh = attribute;
   reg86.h.ah = DOS_SCROLL_WINDOW_UP;
   int86 (VIDEO, &reg86, &reg86);

   return;
}


/* void set_cursor_position (unsigned int, unsigned int)                     */
/*                                                                           */
/* This routine sets the cursor position to the specified co-ordinates.      */

void set_cursor_position (unsigned int x,unsigned int y)
{
   union REGS in_reg, out_reg;

   /* get page number */

   in_reg.h.ah = CURRENT_VIDEO_STATE;
   int86 (VIDEO, &in_reg, &out_reg);

   /* set cursor position */

   in_reg.h.ah = SET_CURSOR_POS;
   in_reg.h.bh = out_reg.h.bh;
   in_reg.h.dh = (unsigned char) y;
   in_reg.h.dl = (unsigned char) x;
   int86 (0x10, &in_reg, &out_reg);

   return;

}


/* This function displays the parameters of the cellular filter to be       */
/* examined.                                                                 */

void filter_calc_data (char *p, float low, float high)
{
   void set_cursor_position (unsigned int, unsigned int);
   void draw_border (void);
   void clear_screen (void);

   clear_screen ();
   draw_border ();

   set_cursor_position (20,3);
   printf ("Calculating cellular filter coefficients");

   set_cursor_position (20,5);
   printf ("Range for %s is [%9.8f, %9.8f]\n", p, low, high);
   set_cursor_position (20,7);
   printf ("K0 = %9.8f", filter_k0);
   set_cursor_position (20,8);
   printf ("K1 = %9.8f", filter_k1);
   set_cursor_position (20,9);
   printf ("K2 = %9.8f\n", filter_k2);

   return;
}


/* This function display the results for the programs search.             */

void print_results (float k0, float k1, float k2, int n, double error)
{
   int key;

   void set_cursor_position (unsigned int, unsigned int);
   void draw_border (void);
   void clear_screen (void);

   clear_screen ();
   draw_border ();

   set_cursor_position (30,3);
   printf ("Results:");
   set_cursor_position (25,7);
   printf ("K0 = %9.8f", k0);
   set_cursor_position (25,9);
   printf ("K1 = %9.8f", k1);
   set_cursor_position (25,11);
   printf ("K2 = %9.8f", k2);
   set_cursor_position (25,15);
   printf ("Number of passes = %d", n);
   set_cursor_position (10, 17);
   printf ("Minimum root mean square error = %20.8f\n", error);

   set_cursor_position (25,22);
   printf ("Hit any key to return to DOS");
   key  =getch ();
   clear_screen ();

   return;
}


/* This routine displays the mean square error and the minimum mean square */
/* error.                                                                    */

void show_mean_square_error (double error, double minimum_error, float k_min)
{
   void set_cursor_position (unsigned int, unsigned int);

   set_cursor_position (15,11);
   printf ("root mean square error = %20.8f\n", error);
   set_cursor_position (15,12);
   printf ("Minimum error %20.8f at %9.8f", minimum_error, k_min);

   return;
}


/* This routine asks the user for the parameters for the search for the    */
/* cellular filter coefficients and checks to see if they are reasonable.  */

void get_filter_data (int *n, int *accuracy, float *low_k0,  float *high_k0,
                      float *low_k1, float *high_k1, float *low_k2,
                      float *high_k2, int *wf)
{
   void clear_screen (void);
   void draw_border (void);
   void set_cursor_position (unsigned int, unsigned int);
   void clear_entry (unsigned int, unsigned int, unsigned int);


   clear_screen ();
   draw_border ();

   set_cursor_position (35,3);
   printf ("Enter Filter Data:");


   set_cursor_position (25,7);
   printf ("Enter order of cellular filter : ");
   scanf ("%d", n);
   while (*n > (RESPONSE_WIDTH/2))
      {
      set_cursor_position (22,20);
      printf ("Error - maximum cellular filter order is %d\n",
(RESPONSE_WIDTH/2));
      clear_entry (58,7,5);
      scanf ("%d", n);
      }
   clear_entry (22,20,45);

   set_cursor_position (25,9);
   printf ("Enter decimal point precision : ");
   scanf ("%d", accuracy);
   while (*accuracy > (SWEEP_TABLE_SIZE-3))
      {
      set_cursor_position (22,20);
      printf ("Error - maximum precision is %d\n decimal places",
SWEEP_TABLE_SIZE-3);
      clear_entry (57,9,5);
      scanf ("%d", accuracy);
      }
   clear_entry (22,20,55);

   do
      {
      set_cursor_position (25,11);
      printf ("Initial range of K0");
      clear_entry (20,12,50);
      set_cursor_position (20,12);
      printf ("Low : ");
      scanf ("%f", low_k0);
      set_cursor_position (40,12);
      printf ("High : ");
      scanf ("%f", high_k0);
      }
   while (*low_k0 >= *high_k0);

   do
      {
      set_cursor_position (25,14);
      printf ("Initial range of K1");
      clear_entry (20,15,50);
      set_cursor_position (20,15);
      printf ("Low : ");
      scanf ("%f", low_k1);
      set_cursor_position (40,15);
      printf ("High : ");
      scanf ("%f", high_k1);
      }
   while (*low_k1 >= *high_k1);

   do
      {
      set_cursor_position (25,17);
      printf ("Initial range of K2");
      clear_entry (20,18,50);
      set_cursor_position (20,18);
      printf ("Low : ");
      scanf ("%f", low_k2);
      set_cursor_position (40,18);
      printf ("High : ");
      scanf ("%f", high_k2);
      }
   while (*low_k2 >= *high_k2);

   do
      {
      clear_entry (25,20,50);
      set_cursor_position (25,20);
      printf ("Select wieghting function [1,2]: ");
      scanf ("%d", wf);
      }
   while ((*wf != 1) && (*wf != 2));


   return;
}
```

## Appendix B

This appendix contains the listing of the C program, cfilter.c, which implements a specified cellular filter on an image file. This program supports Hercules monochreome graphics and 9 pin Epson printers.

```
/* $Header:   C:/msc/cfilter.c_v   1.1   25 Feb 1991 20:31:10   davids  $ */
/****************************************************************************/
/*                                                                          */
/*      Filename:   cfilter.c                    Revision:   1.2            */
/*      Author:     David Sinclair               Last Edit:  16May91        */
/*                                                                          */
/*      This module is the main module for the program which implements    */
/*      a specified 3x3 square tesselation cellular filter.                 */
/*                                                                          */
/*      This program requires a Hercules display to shows images on        */
/*      screen and produces printer output for an Epson 9 pin printer.     */
/*                                                                          */
/*      Command line (Microsoft compiler): cl /W3 /Ox /AL cfilter.c        */
/*                                                                          */
/****************************************************************************/
/*                                                                          */
/*                         Version History                                  */
/*                                                                          */
/*      Revision                          Notes                             */
/*      --------                          -----                             */
/*                                                                          */
/*        1.00                   Initial Development Version                */
/*                                                                          */
/****************************************************************************/

/****************************** File History ******************************/

/* $Log:   C:/msc/cfilter.c_v  $
 *
 *    Rev 1.1   25 Feb 1991 20:31:10   davids
 * Made show_image() data entry more compatible with the rest of the program.
*/

/****************************************************************************/

/****************************** header  files *****************************/

£include "stdio.h"
£include "stdlib.h"
£include "dos.h"
£include "conio.h"

/**************************** declaration files ***************************/

/****************************** definitions *******************************/

/* ASCII characters */

£define  BS                       0x08
£define  LF                       0x0a
£define  CR                       0x0d
£define  ESC                      0x1b
£define  SPACE                    0x20


/* ASCII graphics characters */

£define  LEFT_TOP_COURNER         201
```

```
£define  RIGHT_TOP_COURNER        187
£define  LEFT_BOTTOM_COURNER      200
£define  RIGHT_BOTTOM_COURNER     188
£define  HORIZONTAL_LINE          205
£define  VERTICAL_LINE            186


/* screen dimensions, counting from 0 */

£define  SCREEN_WIDTH   79
£define  SCREEN_DEPTH   23

£define  SHOW_IMAGE_X_OFFSET  0
£define  SHOW_IMAGE_Y_OFFSET  0


/* miscellaneous */

£define  TRUE        0
£define  FALSE       -1
£define  ERROR       -1

£define  BEGINNING   0

£define  STARTING    0
£define  RUNNING     1
£define  FINISHED    2

£define  MAX_TITLE   45

/* BIOS video services */

£define VIDEO                0x10

£define SET_CURSOR_POS       0x02
£define READ_CURSOR_POS      0x03
£define DOS_SCROLL_WINDOW_UP 0x06
£define CURRENT_VIDEO_STATE  0x0f


/* DOS colours */

£define BLACK                0x00


/* image size definitions */

£define RESOLUTION          300
£define PRINTER_RESOLUTION 72
£define AN_INCH             PRINTER_RESOLUTION

£define IMAGE_WIDTH         711
£define IMAGE_DEPTH         341
£define SLICE_DEPTH         10

£define PRINT_IMAGE_WIDTH  (IMAGE_WIDTH - AN_INCH)
£define PRINT_SLICE_DEPTH  8
```

```
/* graphics definitions */

£define WHITE_ON_BLACK 0x00
£define BLACK_ON_WHITE 0xff


/**************************** global  variables ****************************/

/**************************** module  variables ****************************/

unsigned int iterations = 1;
char show_threshold = 0;
char print_threshold = 0;

char input_file [16] = "input.img";
char output_file [16] = "output.img";
char show_file [16] = "input.img";
char image_file [16] = "input.img";
char print_file [16] = "print.img";
char *temporary_file = "temp.img";
FILE *in_fp, *out_fp, *temp_fp;                              /* file discriptors */

float filter_k0 = (float) 1.0;
float filter_k1 = (float) 1.0;
float filter_k2 = (float) 1.0;

char image [SLICE_DEPTH+2][IMAGE_WIDTH+2];
char resultant [SLICE_DEPTH+2][IMAGE_WIDTH+2];

char print_buffer [PRINT_IMAGE_WIDTH];
char raw_buffer [PRINT_SLICE_DEPTH][IMAGE_WIDTH];

char init_file [10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};


/* graphics variables */

char ttable[] =
{0x61,0x50,0x52,0x0f,0x19,0x06,0x19,0x19,0x02,0x0d,0x0b,0x0c};
char gtable[] =
{0x35,0x2d,0x2e,0x07,0x5b,0x02,0x57,0x57,0x02,0x03,0x00,0x00};
unsigned char colorglb;
int xmax = 719,xmin = 0,ymax = 348,ymin = 0;

/********************************* modules *********************************/

int main ()
{
   char quit_flag = FALSE;
   char ready_to_filter = FALSE;
   int key;

   void clear_screen (void), draw_screen (void), filter (void);
   void set_cursor_position (unsigned int, unsigned int);
   void show_image (void);
   void print_image (void);
   char define_options (void);
   int getche (void);
   int getch (void);

   while (quit_flag == FALSE)
      {
      clear_screen ();
      draw_screen ();

      switch (getche())
         {
         case '1':
            ready_to_filter = define_options ();
            break;

         case '2':
            if (ready_to_filter == TRUE)
               filter ();
            else
               {
               clear_screen ();
               set_cursor_position (15, 12);
               printf ("Filter Options must be defined before filtering");
               key = getch ();
               }
            break;

         case '3':
            show_image ();
            break;

         case '4':
            print_image ();
            break;

         case '0':
            quit_flag = TRUE;
            break;
         }
      }

   clear_screen ();

   return (0);

}


/****************************************************************************/


/* void draw_screen (void)                                                  */
/*                                                                          */
/* This routine draw the main program screen.                               */

void draw_screen (void)
{
   void set_cursor_position (unsigned int, unsigned int), draw_border (void);
   draw_border ();

   set_cursor_position (25,2);
   printf ("Cellular Automaton Filtering");
   set_cursor_position (20,8);
   printf ("1 ... Define Cellular Filter Parameters");
   set_cursor_position (20,10);
   printf ("2 ... Filter Image");
   set_cursor_position (20,12);
   printf ("3 ... Show Image");
   set_cursor_position (20,14);
   printf ("4 ... Print Image");
   set_cursor_position (20,16);
   printf ("0 ... Quit to DOS");

   set_cursor_position (9,20);
   printf ("Option .. [ ]");
   set_cursor_position (20,20);

   return;
}


/* void draw_border (void)                                                    */
/*                                                                            */
/* This routine draws a border around the screen.                             */

void draw_border (void)
{
   int i;
   void set_cursor_position (unsigned int, unsigned int);


   set_cursor_position (0,0);
   putchar (LEFT_TOP_COURNER);

   set_cursor_position (SCREEN_WIDTH,0);
   putchar (RIGHT_TOP_COURNER);

   set_cursor_position (0,SCREEN_DEPTH);
   putchar (LEFT_BOTTOM_COURNER);

   set_cursor_position (SCREEN_WIDTH,SCREEN_DEPTH);
   putchar (RIGHT_BOTTOM_COURNER);

   set_cursor_position (1,0);
   for (i = 1; i < SCREEN_WIDTH; i++)
      putchar (HORIZONTAL_LINE);

   set_cursor_position (1,SCREEN_DEPTH);
   for (i = 1; i < SCREEN_WIDTH; i++)
      putchar (HORIZONTAL_LINE);

   set_cursor_position (0,1);
   for (i = 1; i < SCREEN_DEPTH; i++)
      {
      set_cursor_position (0,i);
      putchar (VERTICAL_LINE);
      }

   set_cursor_position (SCREEN_WIDTH,1);
   for (i = 1; i < SCREEN_DEPTH; i++)
      {
      set_cursor_position (SCREEN_WIDTH,i);
      putchar (VERTICAL_LINE);
      }

   return;
}



/*****************************************************************************/
/* char define_options (void)                                                */
/*                                                                           */
/* This routine allows the user to define the filter coefficents, number     */
/* of iterations, input file and output file.                                */

char define_options (void)
{
   char quit_flag = FALSE;

   void clear_entry (unsigned int, unsigned int, unsigned int);
   void clear_screen (void);
   void set_cursor_position (unsigned int, unsigned int);
   void draw_border (void);
   int getche (void);

   while (quit_flag == FALSE)
      {
      clear_screen ();
      draw_border ();

      set_cursor_position (25,2);
      printf ("Define Cellular Filter Parameters");
      set_cursor_position (20,5);
      printf ("1 ... Number of Iterations = %d", iterations);
      set_cursor_position (20,7);
      printf ("2 ... Input File  = %s", input_file);
      set_cursor_position (20,9);
      printf ("3 ... Output File = %s", output_file);
      set_cursor_position (20,11);
      printf ("4 ... Filter Coefficient K0 = %8.7f", filter_k0);
      set_cursor_position (20,13);
      printf ("5 ... Filter Coefficient K1 = %8.7f", filter_k1);
      set_cursor_position (20,15);
      printf ("6 ... Filter Coefficient K2 = %8.7f", filter_k2);
      set_cursor_position (20,17);
      printf ("0 ... Return to Main Screen");

      set_cursor_position (9,20);
      printf ("Option .. [ ]");
      set_cursor_position (20,20);

      switch (getche())
         {
```

```
         case '1':
            clear_entry (49,5,5);
            scanf ("%d", &iterations);
            break;

         case '2':
            clear_entry (40,7,16);
            scanf ("%s", input_file);
            break;

         case '3':
            clear_entry (40,9,16);
            scanf ("%s", output_file);
            break;

         case '4':
            clear_entry (50,11,12);
            scanf ("%f", &filter_k0);
            break;

         case '5':
            clear_entry (50,13,12);
            scanf ("%f", &filter_k1);
            break;

         case '6':
            clear_entry (50,15,12);
            scanf ("%f", &filter_k2);
            break;

         case '0':
            quit_flag = TRUE;
            break;
         }
      }

   return (TRUE);
}


/* void clear_entry (unsigned int, unsigned int, unsigned int)             */
/*                                                                          */
/* This routine clears the screen with (repeat) spaces at the specified    */
/* position  and set the cursor position to the specified location.        */

void clear_entry (x, y, repeat)
unsigned int x, y, repeat;
{
   unsigned int i;

   void set_cursor_position (unsigned int, unsigned int);

   set_cursor_position (x,y);
   for (i = 0; i < repeat; i++)
      putchar (SPACE);
   set_cursor_position (x,y);

   return;
```

```
}


/*************************************************************************/
/* void filter (void)                                                    */
/*                                                                       */
/* This routine filters the input image with the cellular filter defined   */
/* by the define_options () routine.                                     */

void filter (void)
{
   unsigned char pass_state;
   char *p ,*r;
   unsigned int i, slice_number;
   unsigned long count, l, line_count;
   float centre_element;

   void cant_open (char *);
   unsigned long load_slice (unsigned int, unsigned int, unsigned char *);
   void write_slice (unsigned long, unsigned int);
   void set_cursor_position (unsigned int, unsigned int);
   void clear_screen (void);
   void draw_border (void);
   void fatal_error (char *, char *, char *);
   int getch (void);


   if ((out_fp = fopen (output_file, "w+b")) == (FILE *) ERROR)
      {
      cant_open (output_file);
      return;
      }
   else
      {
      fwrite (init_file, sizeof (char), sizeof (init_file), out_fp);
      fclose (out_fp);
      }

   if ((temp_fp = fopen (temporary_file, "w+b")) == (FILE *) ERROR)
      {
      cant_open (temporary_file);
      return;
      }
   else
      {
      fwrite (init_file, sizeof (char), sizeof (init_file), out_fp);
      fclose (temp_fp);
      }


   clear_screen ();
   draw_border ();
   set_cursor_position (30, 2);
   printf ("Filtering %s", input_file);

   for (i = 0; i < iterations; i++)
      {
      if ((in_fp = fopen (input_file, "r+b")) == (FILE *) ERROR)
```

```
         {
         cant_open (input_file);
         return;
         }

      if ((out_fp = fopen (output_file, "r+b")) == (FILE *) ERROR)
         {
         cant_open (output_file);
         return;
         }
      else                                                     /* reset to start
*/
         if (fseek (out_fp, (long) 0, BEGINNING) == ERROR)
            fatal_error ("FATAL ERROR- fseek on", output_file, "in filter ()
failed - Hard Luck!");

      if ((temp_fp = fopen (temporary_file, "r+b")) == (FILE *) ERROR)
         {
         cant_open (temporary_file);
         return;
         }
      else                                                     /* reset to start
*/
         if (fseek (temp_fp, (long) 0, BEGINNING) == ERROR)
            fatal_error ("FATAL ERROR- fseek on",temporary_file ,"in filter
() failed - Hard Luck!");


      slice_number = 0;
      pass_state = STARTING;
      do
         {
         set_cursor_position (25, 20);
         printf ("Iteration %2d of %2d : Slice %4d", (i+1), iterations,
slice_number);

         count = load_slice (i+1, slice_number, &pass_state);

                                                /* point p to image [1][1] */
         p = (char *) image + (IMAGE_WIDTH+3);
         r = (char *) resultant + (IMAGE_WIDTH+3);

         for (l = count, line_count = 1; l != 0; l--, line_count++)
            {
            centre_element = ((float) *p * filter_k0)
                              + ((float) *(p-(1)) * filter_k1)
                              + ((float) *(p+(1)) * filter_k1)
                              + ((float) *(p-(((IMAGE_WIDTH+2)+1))) *
filter_k2)
                              + ((float) *(p-((IMAGE_WIDTH+2))) * filter_k1)
                              + ((float) *(p-(((IMAGE_WIDTH+2)-1))) *
filter_k2)
                              + ((float) *(p+(((IMAGE_WIDTH+2)-1))) *
filter_k2)
                              + ((float) *(p+((IMAGE_WIDTH+2))) * filter_k1)
                              + ((float) *(p+(((IMAGE_WIDTH+2)+1))) *
filter_k2);
```

```
            if (centre_element > (float) 127.0)
               *r = (char) 127;
            else
               if (centre_element < (float) -127.0)
                  *r = (char) -127;
               else
                  *r = (char) centre_element;

            if (line_count < IMAGE_WIDTH)
               {
               p++;
               r++;
               }
            else
               {
               p++; p++; p++;
               r++; r++; r++;
               line_count = 0;
               }
            }

         write_slice (count, i+1);
         slice_number ++;
         }
      while (pass_state != FINISHED);

      fclose (in_fp);
      fclose (out_fp);
      fclose (temp_fp);
      }

   return;
}


/* unsigned long load_slice (unsigned int, unsigned int, unsigned char *)  */
/*                                                                          */
/* This routine loads a slice of the image into the image buffer and       */
/* returns the number of bytes in the buffer to be processed and whether   */
/* it has finished this iteration or not.                                   */
/* If the number of iterations is odd, then the slice comes from the       */
/* output file on even passes, and from the temporary file on odd passes   */
/* except for the first pass. If the number of iterations is even, then    */
/* the slice comes from the output file on odd passes except for the first */
/* pass, and from the temporary file on even passes.                        */

unsigned long load_slice (pass, slice_number, state)
unsigned int pass;
unsigned int slice_number;
unsigned char *state;
{
   char *p;
   long l;
   FILE *file_fp;
   long i, block_size, number_of_blocks, file_position;

   void set_cursor_position (unsigned int, unsigned int);
   void clear_screen (void);
   void fatal_error (char *, char *, char *);

   for (l = (IMAGE_WIDTH+2)*(SLICE_DEPTH+2), p = (char *) image;
        l != 0; l--, p++)
      *p = 0;                                 /* clear out image buffer */

   for (l = (IMAGE_WIDTH+2)*(SLICE_DEPTH+2), p = (char *) resultant;
        l != 0; l--, p++)
      *p = 0;                                 /* clear out resultant buffer
*/

                                             /* first get the right source */
   if ((iterations & 0x01) == 0x00 )         /* even number of iterations */
      if ((pass & 0x01) == 0x00)             /* even pass */
         file_fp = temp_fp;
      else                                   /* odd pass */
         if (pass != 1)                      /* not first pass */
            file_fp = out_fp;
         else                                /* first pass */
            file_fp = in_fp;
   else                                      /* odd number of iterations */
      if ((pass & 0x01) == 0x00)             /* even pass */
         file_fp = out_fp;
      else                                   /* odd pass */
         if (pass != 1)                      /* not first pass */
            file_fp = temp_fp;
         else                                /* first pass */
            file_fp = in_fp;

                                             /* and setup start position */
                                             /* in source file */

   file_position = (long) (((long) IMAGE_WIDTH * (long) SLICE_DEPTH *
slice_number) - (long) (IMAGE_WIDTH));
   file_position = (file_position < 0) ? (long) 0 : file_position;
   if (fseek (file_fp, file_position, BEGINNING) == ERROR)
      fatal_error ("FATAL ERROR- fseek in load_slice () failed - Hard
Luck!","","");

                                             /* then setup destination and */
                                             /* size of slice */
   if (*state == STARTING)
      {
                                                /* point p to image [1][1] */
      p = (char *) image + (IMAGE_WIDTH+3);
      number_of_blocks = SLICE_DEPTH+1;
      *state = RUNNING;
      }
   else
      {
                                                /* point p to image [0][1] */
      p = (char *) image + (1);
      number_of_blocks = SLICE_DEPTH+2;
      }

                                             /* read the slice from source */
                                             /* to destination.            */
   for (i = 0, block_size = (long) 0; i < number_of_blocks;
         i++, p += ((IMAGE_WIDTH+2)))
      {
      block_size += fread (p, sizeof (char), IMAGE_WIDTH, file_fp);
      }

   if (block_size != (IMAGE_WIDTH * number_of_blocks))
      {
      *state = FINISHED;
      block_size = (block_size - IMAGE_WIDTH)>0?
                   (long) (block_size - IMAGE_WIDTH): (long) 0;
      }
   else
      block_size = SLICE_DEPTH * IMAGE_WIDTH;

   return (block_size);
}


/* void write_slice (unsigned long, unsigned int)                           */
/*                                                                          */
/* This routine write the processed slice of the processed image            */
/* (resultant) to the appropriate file.                                     */
/* If the number of iterations is odd, then the slice is writen to          */
/* the output file on odd passes, and to the temporary file on even         */
/* passes. If the number of iterations is even, then the slice is writen    */
/* to the output file on even passes and to the temporary file on odd       */
/* passes.                                                                  */

void write_slice (length, pass)
unsigned long length;                        /* unreferenced at the moment */
unsigned int pass;
{
   char *p;
   int i;
   FILE *file_fp;

                                             /* first get the right source */
   if ((iterations & 0x01) == 0x00 )         /* even number of iterations */
      if ((pass & 0x01) == 0x00)             /* even pass */
         file_fp = out_fp;
      else                                   /* odd pass */
         file_fp = temp_fp;
   else                                      /* odd number of iterations */
      if ((pass & 0x01) == 0x00)             /* even pass */
         file_fp = temp_fp;
      else                                   /* odd pass */
         file_fp = out_fp;

   p = (char *) resultant + ((IMAGE_WIDTH+3));
   for (i = 0; i < SLICE_DEPTH;
         i++, p += ((IMAGE_WIDTH+2)))
      fwrite (p, sizeof (char), IMAGE_WIDTH, file_fp);

   return;
}


/* void cant_open (char *)                                                  */
/*                                                                            */
/* This routine displays the error message that the file could not be       */
/* opened.                                                                    */

void cant_open (file)
char *file;
{
   int debug_key;

   void clear_screen (void);
   void set_cursor_position (unsigned int, unsigned int);
   int getche (void);

   clear_screen ();
   set_cursor_position (15,12);
   printf ("ERROR - unable to open %s", file);
   set_cursor_position (15,13);
   printf ("Hit any key to continue");
   debug_key = getche ();

   return;
}

/* void fatal_error (char *, char *, char *)                                 */
/* This routine prints the message passed to it and exits to DOS.            */

void fatal_error (start_string, file, end_string)
char *start_string, *file, *end_string;
{
   void clear_screen (void);
   void set_cursor_position (unsigned int, unsigned int);

   clear_screen ();
   set_cursor_position (15,12);
   printf ("%s %s %s\n", start_string, file, end_string);
   fclose (in_fp);
   fclose (temp_fp);
   fclose (out_fp);
   exit (1);

   return;
}


/****************************************************************************/
/*                                                                            */
/* void show_image (void)                                                     */
/* This routine displays the image specified by the user.                    */

void show_image (void)
{
   unsigned char pass_state;
   char *p;
   char quit_flag = FALSE;
   char show_image_flag = FALSE;
   unsigned int key, slice_number, x_coord, y_coord;
   unsigned long count, l;
```

```
void clear_screen (void);
void draw_border (void);
void clear_entry (unsigned int, unsigned int, unsigned int);
void set_cursor_position (unsigned int, unsigned int);
unsigned long load_slice (unsigned int, unsigned int, unsigned char *);
void cant_open (char *);
int getch (void);
int getche (void);

void init_graphics (void), leave_graphics (void);
void set_pixel (unsigned int, unsigned int);
void clear_graphics_screen (void);


while (quit_flag == FALSE)
   {
   clear_screen ();
   draw_border ();

   set_cursor_position (35, 3);
   printf ("Display Image\n");
   set_cursor_position (20, 9);
   printf ("1 ... File to display = %s", show_file);
   set_cursor_position (20, 11);
   printf ("2 ... Set threshold = %hd", show_threshold);
   set_cursor_position (20, 13);
   printf ("3 ... Display %s", show_file);
   set_cursor_position (20,17);
   printf ("0 ... Return to Main Screen");

   set_cursor_position (9,20);
   printf ("Option .. [  ]");
   set_cursor_position (20,20);

   switch (getche())
      {
      case '1':
         clear_entry (44,9,16);
         scanf ("%s", show_file);
         break;

      case '2':
         clear_entry (42,11,5);
         scanf ("%hd", &show_threshold);
         break;

      case '3':
         show_image_flag = TRUE;
         quit_flag = TRUE;
         break;

      case '0':
         quit_flag = TRUE;
         break;
      }
   }
   if (show_image_flag == FALSE && quit_flag == TRUE)
      return;


   if ((in_fp = fopen (show_file, "r+b")) == (FILE *) ERROR)
      {
      cant_open (show_file);
      return;
      }


   init_graphics ();
   clear_graphics_screen ();

   x_coord = SHOW_IMAGE_X_OFFSET;
   y_coord = SHOW_IMAGE_Y_OFFSET;
   slice_number = 0;
   pass_state = STARTING;
   do
      {
                                     /* pass 1 always reads from input file */
      count = load_slice (1, slice_number, &pass_state);

                                                 /* point p to image [1][1] */
      p = (char *) image + (IMAGE_WIDTH+3);

      for (l = count; l != 0; l--)
           {
                                                           /* display image */
         if (*p >= show_threshold)
            set_pixel (x_coord, y_coord);

         if ((x_coord - SHOW_IMAGE_X_OFFSET) < (IMAGE_WIDTH-1))
            {
            p++;
            x_coord++;
            }
         else
            {
            p++; p++; p++;
            x_coord = SHOW_IMAGE_X_OFFSET;
            y_coord++;
            }
           }

      slice_number ++;
      }
   while (pass_state != FINISHED);

   key = getch ();
   leave_graphics ();

   return;
}


/***************************************************************************/
/*                                                                         */
/* void display_printing_options (char *, char, char *)                      */
/* This rouitne displays the printing options. This was seperated from the */
/* print_image routine to allow global optimisation.                         */

void display_printing_options (image_file, print_file, threshold, title)
char *image_file, *print_file, *title;
char threshold;
{
   void clear_screen (void);
   void draw_border (void);
   void clear_entry (unsigned int, unsigned int, unsigned int);
   void set_cursor_position (unsigned int, unsigned int);

   clear_screen ();
   draw_border ();

   set_cursor_position (35, 3);              /* display user print options */
   printf ("Print Image\n");
   set_cursor_position (20, 7);
   printf ("1 ... File to print = %s", image_file);
   set_cursor_position (20, 9);
   printf ("2 ... Set threshold = %hd", threshold);
   set_cursor_position (20, 11);
   printf ("3 ... Print File = %s", print_file);
   set_cursor_position (20,13);
   printf ("4 ... Title = %s", title);
   set_cursor_position (20,15);
   printf ("5 ... Print %s", image_file);
   set_cursor_position (20,17);
   printf ("0 ... Return to Main Screen");

   set_cursor_position (9,20);
   printf ("Option .. [ ]");
   set_cursor_position (20,20);

   return;
}


/* void print_image (void)                                                   */
/* This routine prints the image specified by the user to a 9 pin EPSON     */
/* printer attached to LPT1:.                                                */

void print_image (void)
{
   char *p, *rp;
   char c, quit_flag = FALSE, print_image_flag = FALSE;
   char title [MAX_TITLE];
   unsigned char mask;
   unsigned int i, j, slice_number, number_of_blocks, title_length;
   unsigned long l, file_position;
   FILE *file_fp, *print_fp;

   char line_feeds [8] = {LF, LF, LF, LF, LF, LF, LF, LF};
   char printer_init_seq [2] = {ESC, '@'};
   char line_step_and_unidirectional [6] = {ESC, 'A', 0x08, ESC, 'U', 0x01};
   char graphics_seq [5] = {ESC, '*', '5', 0x00, 0x00};
   char bold_and_underline [5] = {ESC, 'E', ESC, '-', 0x01};
   char cancel_bold_and_underline [5] = {ESC, 'F', ESC, '-', 0x00};
   char *spaces = "                                  ";
   char cr_lf [2] = {CR, LF};

   void clear_screen (void);
   void draw_border (void);
   void clear_entry (unsigned int, unsigned int, unsigned int);
   void set_cursor_position (unsigned int, unsigned int);
   void cant_open (char *);
   void display_printing_options (char *, char *, char, char *);
   void fatal_error (char *, char *, char *);
   int getch (void);
   int getche (void);


   title [0] = '\0';
   title_length = 0;


   while (quit_flag == FALSE)
      {
      display_printing_options (image_file, print_file, print_threshold,
title);

      switch (getche())                                    /* get user selection */
         {
         case '1':
            clear_entry (42,7,16);
            scanf ("%s", image_file);
            break;

         case '2':
            clear_entry (42,9,5);
            scanf ("%hd", &print_threshold);
            break;

         case '3':
            clear_entry (39,11,16);
            scanf ("%s", print_file);
            break;

         case '4':
            clear_entry (34,13,(MAX_TITLE-1));
            title_length = 0;
            p = title;
            while ( ((c = (char) getche ()) != '\r') &&
                     title_length < MAX_TITLE)
               {
               if ( c != BS )
                  {
                  *p = c;
                  p++;
                  title_length++;
                  }
               else
                  {
                  p--;
                  title_length--;
                  }
               }
            *p = '\0';
            break;

         case '5':
            print_image_flag = TRUE;
            quit_flag = TRUE;
            break;

         case '0':
            quit_flag = TRUE;
            break;
         }
      }


   if (print_image_flag == FALSE && quit_flag == TRUE)
      return;

   clear_screen ();
   draw_border ();
   set_cursor_position (15, 10);                   /* display user options */
   printf ("Printing %s thresolded at %hd to %s", image_file,
print_threshold, print_file);

                                                    /* open file to print */
   if ((file_fp = fopen (image_file, "r+b")) == (FILE *) ERROR)
      {
      cant_open (image_file);
      return;
      }

                                                    /* open output file */
   if ((print_fp = fopen (print_file, "wb")) == (FILE *) ERROR)
      {
      cant_open (print_file);
      return;
      }


   /* do a few line feeds */

   fwrite (line_feeds, sizeof (char), 8, print_fp);

   /* print escape sequence for 8 pin step line advance, unidirectional */

   fwrite (printer_init_seq, sizeof (char), 2, print_fp);
   fwrite (line_step_and_unidirectional, sizeof (char), 6, print_fp);

   graphics_seq [3] = (char) (PRINT_IMAGE_WIDTH % 256);
   graphics_seq [4] = (char) (PRINT_IMAGE_WIDTH / 256);
   slice_number = 0;

   do
      {
      /* clear out unformatted print buffer */
      for (l = (IMAGE_WIDTH)*(PRINT_SLICE_DEPTH), p = (char *) raw_buffer;
             l != 0; l--, p++)
          *p = 0;

       /* clear out print buffer */
       for (l = PRINT_IMAGE_WIDTH, p = (char *) print_buffer;
             l != 0; l--, p++)
          *p = 0;


       /* position file pointer */
       file_position = (long) ((long) IMAGE_WIDTH * (long) PRINT_SLICE_DEPTH *
slice_number);
       if (fseek (file_fp, file_position, BEGINNING) == ERROR)
          {
          fclose (file_fp);
          fatal_error ("FATAL ERROR- fseek in load_slice () failed - Hard
Luck!", "", "");
          }

       rp = (char *) raw_buffer;
       number_of_blocks = PRINT_SLICE_DEPTH;

       /* read section of image from file to unformatted print buffer */
       for (i = 0; i < number_of_blocks; i++, rp += IMAGE_WIDTH)
          fread (rp, sizeof (char), IMAGE_WIDTH, file_fp);

       rp = (char *) raw_buffer;
       p = (char *) print_buffer;

       mask = 0x80;
       for (j=0; j<PRINT_IMAGE_WIDTH; j++, rp++, p++)
          {
          for (i=0; i<8; i++)
             if (*(rp+(i*IMAGE_WIDTH)+AN_INCH) >= print_threshold)
                *p |= (mask >> i);
          }


          /* print graphics escape sequence */

          fwrite (graphics_seq, sizeof (char), 5, print_fp);


          /* print print_buffer */

          fwrite (print_buffer, sizeof (char), IMAGE_WIDTH, print_fp);


          /* print <cr><lf> */

          fwrite (cr_lf, sizeof (char), 2, print_fp);

       }
    while (++slice_number < (IMAGE_DEPTH/PRINT_SLICE_DEPTH));

    /* do a few line feeds */

    fwrite (line_feeds, sizeof (char), 8, print_fp);
```

```c
   /* add title */

   fwrite (spaces, sizeof (char), 20, print_fp);
   fwrite (bold_and_underline, sizeof (char), 5, print_fp);
   fwrite (title, sizeof (char), title_length, print_fp);
   fwrite (cr_lf, sizeof (char), 2, print_fp);
   fwrite (cancel_bold_and_underline, sizeof (char), 5, print_fp);

   /* and reset printer */

   fwrite (printer_init_seq, sizeof (char), 2, print_fp);

   /* clean up on exit */

   fclose (file_fp);
   fclose (print_fp);

   return;
}


/****************************************************************************/
/* void clear_screen (void)                                                 */
/*                                                                          */
/* This routine clears the entire screen.                                   */

void clear_screen (void)
{
   void up_scroll_region (unsigned char, unsigned char, unsigned char,
                          unsigned char, unsigned char, unsigned char);

   up_scroll_region (0,0,79,23,0,BLACK);

   return;
}


/* void up_scroll_region (unsigned char, unsigned char, unsigned char,      */
/*                          unsigned char, unsigned char, unsigned char)    */
/*                                                                          */
/* This routine uses DOS to clear a region of the screen.                   */

void up_scroll_region (x1, y1, x2, y2, up_scroll, attribute)
unsigned char x1, y1, x2, y2, up_scroll, attribute;
{
   union REGS reg86;

   reg86.h.al = up_scroll;
   reg86.h.cl = x1;
   reg86.h.ch = y1;
   reg86.h.dl = x2;
   reg86.h.dh = y2;
   reg86.h.bh = attribute;
   reg86.h.ah = DOS_SCROLL_WINDOW_UP;
   int86 (VIDEO, &reg86, &reg86);

   return;
```

```
}


/* void set_cursor_position (unsigned int, unsigned int)                        */
/*                                                                              */
/* This routine sets the cursor position to the specified co-ordinates.         */

void set_cursor_position (x,y)
unsigned int x,y;
{
   union REGS in_reg, out_reg;

   /* get page number */

   in_reg.h.ah = CURRENT_VIDEO_STATE;
   int86 (VIDEO, &in_reg, &out_reg);

   /* set cursor position */

   in_reg.h.ah = SET_CURSOR_POS;
   in_reg.h.bh = out_reg.h.bh;
   in_reg.h.dh = (unsigned char) y;
   in_reg.h.dl = (unsigned char) x;
   int86 (0x10, &in_reg, &out_reg);

   return;

}


/* void not_yet_implemented (void)                                              */
/*                                                                              */
/* this routine informs the user that the requested function has not yet       */
/* been implimented.                                                            */

void not_yet_implemented (void)
{
   int debug_key;

   void set_cursor_position (unsigned int, unsigned int);
   void clear_screen (void);
   int getche (void);

   clear_screen ();
   set_cursor_position (15,12);
   printf ("Not Yet Implemented ... Hit any key to continue");
   debug_key = getche ();

   return;
}

/*****************************************************************************/
/* void init_graphics (void)                                                    */
/*                                                                              */
/* This routine initialises the display into grpahics mode.                     */

void init_graphics (void)
{
   void clear_graphics_screen (void);

   int i;

   colorglb = (unsigned char) WHITE_ON_BLACK;
   outp (0x03bf,0x01);
   outp (0x03b5,0x02);
   for ( i=0; i<12; i++ )
      {
      outp (0x03b4,i);
      outp (0x03b5,gtable[i]);
      }
   outp (0x03b8,0x02+0x08);
   clear_graphics_screen ();
   return;
}


/* void leave_graphics (void)                                            */
/*                                                                       */
/* This routine leaves graphics mode and resets the display for text mode. */

void leave_graphics (void)
{
   int i;

   void clear_graphics_screen (void);

   outp (0x03bf,0x00);
   outp (0x03b8,0x20);
   for ( i=0; i<12; i++)
      {
      outp (0x03b4,i);
      outp (0x03b5,ttable[i]);
      }
   colorglb = 0;
   outp (0x03b8,0x20+0x08);
   clear_graphics_screen ();
   return;
}


/* void set_pixel (unsigned int, unsigned int)                           */
/*                                                                       */
/* This routine sets the pixel at the specified location.                */

void set_pixel (x, y)
unsigned int x, y;
{
   unsigned char *p,bit;

   FP_SEG(p) = 0xb000;
   FP_OFF(p) = ((y&3)<<13) + 90*(y>>2) + (x>>3);
   bit = (unsigned char) (0x80>>(x&7));
   if (colorglb == (unsigned char) WHITE_ON_BLACK)
      *p = *p ¦ bit;
   else
      *p = *p & bit;
   return;
}

/* void clear_graphics_screen (void)                                                  */
/*                                                                                    */
/* This routine clears a graphics screen.                                             */

void clear_graphics_screen (void)
{
   unsigned char *p;

   for ( FP_SEG(p) = 0xb000, FP_OFF(p) = 0x0000; FP_OFF(p) < 0x8000;
      *(p++) = colorglb );
   return;
}
```

## Appendix C

This appendix contains a listing of the occam program, netfilt.tsr, which implements a cellular filter on a network of transputers connected in a pipeline.

```
{{{  EXE netfilt
{{{F netfilt
--:::F netfilt.tsr
------------------------------------------------------------------------------
--                                                                          --
--    File:   netfilt.ocm                          Revision:  1.0           --
--    Author: David Sinclair                      Last Edit:  21Jul91       --
--                                                                          --
--    This program performs a cellular filter on an image on a network      --
--    of transputers.                                                       --
--                                                                          --
------------------------------------------------------------------------------

{{{  define constants
------------------------------------------------------------------------------
--                             define constants                             --
------------------------------------------------------------------------------

VAL image.width IS 714 :
VAL image.hieght IS 340 :

VAL block.width IS 42 :
VAL block.hieght IS 20 :

VAL number.of.processors IS 4 :

}}}

{{{  define channel protocols
------------------------------------------------------------------------------
--                         define channel protocols                         --
------------------------------------------------------------------------------

PROTOCOL data.mgr.msgs
  CASE
    quit
    hold
    request.something
    filter.params; REAL32; REAL32; REAL32; INT
    image.row; INT::[]BYTE
    image.saved
:


PROTOCOL image.filter.msgs
  CASE
    request.image
    image.filtered; INT
    filtered.row; INT::[]BYTE
:

PROTOCOL from.file.mgr.msgs
  CASE
    initialise; INT
    quit; INT
    hold; INT
    request.something; INT
    filter.param; INT; REAL32; REAL32; REAL32; INT
    image.row; INT; INT::[]BYTE
    image.saved; INT
:

PROTOCOL to.file.mgr.msgs
  CASE
    request.image; INT
    image.filtered; INT; INT
    filtered.row; INT; INT::[]BYTE
:

}}}

{{{  define channels
-----------------------------------------------------------------------------
--                              define channels                            --
-----------------------------------------------------------------------------

CHAN OF data.mgr.msgs from.data.mgr :
CHAN OF image.filter.msgs to.data.mgr :

CHAN OF to.file.mgr.msgs up.data.out :
CHAN OF to.file.mgr.msgs down.data.in :
CHAN OF from.file.mgr.msgs up.data.in :
CHAN OF from.data.mgr.msgs down.data.out :

}}}

{{{  define variables
-----------------------------------------------------------------------------
--                             define variables                            --
-----------------------------------------------------------------------------

}}}

-----------------------------------------------------------------------------
--                                 procedures                              --
-----------------------------------------------------------------------------

[number.of.processors] BOOL quit.table :

{{{  can.quit
-----------------------------------------------------------------------------
--    Procedure: can.quit                                                  --
--                                                                         --
--    This procedure updates the list of processors which have been told   --
--    to quit (terminate) by the file manager. This procedure is called    --
--    by the data manager when it sees a quit message sent from the file   --
--    manager to any node (processor running a data manager). When all     --
--    down stream nodes, include this node have been told to quit, then    --
--    this procedure will set the active flag to FALSE, signalling the     --
--    data manager to terminate.                                           --
-----------------------------------------------------------------------------

PROC can.quit (INT id, INT node.number, BOOL active)

  SEQ
    quit.table [node.number] := TRUE
    active := TRUE
    SEQ i = id FOR (number.of.processors - id)
      IF
        quit.table [i] = FALSE
          active := FALSE

        quit.table [i] = TRUE
          SKIP
:

}}}

{{{  data.manager
------------------------------------------------------------------------------
--    Task: data manager                                                    --
--                                                                          --
--    This task adds messages from the image filter task to the stream of --
--    messages being sent to the file manager. Messages which are sent     --
--    down stream from the file manager are examined by the data manager  --
--    to determine if the message should be sent to its assocated image    --
--    filter task. Quit messages for down stream tasks are recorded to     --
--    enable the data manager to terminate when it, and all down stream    --
--    processors have been told to terminate.                               --
------------------------------------------------------------------------------

PROC data.manager (VAL INT id,
                   CHAN OF from.file.mgr.msgs up.data.in, down.data.out
                   CHAN OF to.file.mgr.msgs up.data.out, down.data.in)

  [(block.width+2)]BYTE block :
  BOOL active :
  INT index, node.number, block.size :
  REAL k0, k1, k2 :

  SEQ
    up.data.in ? CASE initialise; node.number

    PAR i = 0 FOR number.of.processors
      quit.table [i] := FALSE

    IF
      node.number < id
        down.data.out ! initialise; node.number

      node.number >= id
        SKIP

    WHILE active
      ALT
        to.data.mgr ? CASE

          request.image
            up.data.out ! request.image; id

          image.filtered; index
            up.data.out ! image.filtered; id; index

          filtered.row; block.size::block
      up.data.out ! filtered.row; id; block.size::block

  down.data.in ? CASE

    request.image; node_number
      up.data.out ! request.image; node.number

    image.filtered; node.number; index
      SEQ
        up.data.out ! image.filtered; node.number; index
        SEQ i = 0 FOR (block.hieght+2)
          down.data.in ? CASE filtered.row; node.number;
                                            block.size::block
          up.data.out ! filtered.row; node.number; block.size::block

  up.data.in ? CASE

    quit; node.number
      PAR
        can.quit (id, node.number, active)
        IF
          node.number <> id
            down.data.out ! quit; node.number

          node.number = id
            from.data.mgr ! quit

    hold; node.number
      IF
        node.number <> id
          down.data.out ! hold; node.number

        node.number = id
          from.data.mgr ! hold

    request.something; node.number
      IF
        node.number <> id
          down.data.out ! request.something; node.number

        node.number = id
          from.data.mgr ! request.something

    image.saved; node.number
      IF
        node.number <> id
          down.data.out ! image.saved; node.number

        node.number = id
          from.data.mgr ! image.saved

    filter.param; node.number; k0; k1; k2; index
      IF
        node.number <> id
          SEQ
            down.data.out ! filter.param; node.number; k0; k1; k2;
                                          index
            SEQ i = 0 FOR (block.hieght+2)
          up.data.in ? image.row; node.number; block.size::block
          down.data.out ! image.row; node.number; block.size::block

       node.number = id
         SEQ
           from.data.mgr ! filter.param; k0; k1; k2; index
           SEQ i = 0 FOR (block.hieght+2)
             up.data.in ? image.row; node.number; block.size::block
             from.data.mgr ! image.row; block.size::block

:

}}}

{{{  image.filter
---------------------------------------------------------------------------
--     Task: image filter                                                --
--                                                                       --
--     This task applies the cellular automaton defined by k0, k1 and k2 --
--     to the image supplied to it.                                      --
---------------------------------------------------------------------------

PROC image.filter ()                           -- perform one pass of cellular
                                               -- filter

  VAL block.size IS (block.width * block.hieght) :

  INT size, block.number :
  REAL32 k0, k1, k2 :
  BOOL active :
  [(block.hieght + 2)][(block.width + 2)]BYTE raw.image :
  [(block.hieght + 2)][(block.width + 2)]BYTE filtered.image :

  SEQ
    active := TRUE
    WHILE active
      SEQ
        from.data.mgr ? CASE

          quit                                 -- shut down process neatly
            active := FALSE

          hold                                 -- don't do anything just yet
            SKIP

          request.something                    -- ask for an image
            to.data.mgr ! request.image        -- request image to process

                                               -- get cellular filter
                                              -- parameters
          filter.params; k0; k1; k2; block.number
            SEQ                                -- get data to process
              SEQ j = 0 FOR (block.width + 2)
                from.data.mgr ? CASE image.row; size::raw.image [j]

              PAR i = 0 FOR block.size         -- process each cell
                VAL x.offset IS ((i / block.width) + 1) :
                VAL y.offset IS ((i REM block.width) + 1) :
```

```
INT sum, b1, b2, b3, b4, b5, b6, b7, b8, b9 :

SEQ

  PAR                              -- work out intermediate values

    b5 := (INT TRUNC (((REAL32 TRUNC
          (INT (raw.image [x.offset][y.offset]))) -
          128.0 (REAL32)) * k0))

    IF
      x.offset > 0
        PAR
          b4 := (INT TRUNC (((REAL32 TRUNC
                (INT (raw.image [x.offset - 1][y.offset]))) -
                128.0 (REAL32)) * k1))

          IF
            y.offset > 0
              b1 := (INT TRUNC (((REAL32 TRUNC
                    (INT (raw.image [x.offset -1 ][y.offset -
                    1]))) - 128.0 (REAL32)) * k2))
            y.offset <= 0
              b1 := 0

          IF
            (y.offset + 1) < block.hieght
              b7 := (INT TRUNC (((REAL32 TRUNC
                    (INT (raw.image [x.offset - 1][y.offset +
                    1]))) - 128.0 (REAL32)) * k2))
            (y.offset + 1) >= block.hieght
              b7 := 0

      x.offset <= 0
        b4, b1, b7 := 0, 0, 0

    IF
      y.offset > 0
        PAR
          b2 := (INT TRUNC (((REAL32 TRUNC
                (INT (raw.image [x.offset][y.offset - 1]))) -
                128.0 (REAL32)) * k1))

          IF
            (x.offset + 1) < block.width
              b3 := (INT TRUNC (((REAL32 TRUNC
                    (INT (raw.image [x.offset + 1][y.offset -
                    1]))) - 128.0 (REAL32)) * k2))
            (x.offset + 1) >= block.width
              b3 := 0

      y.offset <= 0
        b2, b3 := 0, 0

    IF
      (x.offset + 1) < block.width
        PAR
          b6 := (INT TRUNC (((REAL32 TRUNC
                                (INT (raw.image [x.offset + 1][y.offset])) -
                                128.0 (REAL32)) * k1))

                          IF
                            (y.offset + 1) < block.hieght
                              b9 := (INT TRUNC (((REAL32 TRUNC
                                    (INT (raw.image [x.offset + 1][y.offset +
                                    1]))) - 128.0 (REAL32)) * k2))
                            (y.offset + 1) >= block.hieght
                              b9 := 0

                      (x.offset + 1) >= block.width
                        b6, b9 := 0, 0

                    IF
                      (y.offset + 1) < block.hieght
                        b8 := (INT TRUNC (((REAL32 TRUNC
                              (INT (raw.image [x.offset][y.offset + 1]))) -
                              128.0 (REAL32)) * k1))
                      (y.offset + 1) >= block.hieght
                        b8 := 0


                  sum := (b1 + (b2 + (b3 + (b4 + (b5 + (b6 + (b7 + (b8 +
                          b9))))))))
                  IF
                    sum > 127
                      sum := 127

                    sum < (-127)
                      sum := -127

                    (sum >= (-127)) AND (sum <= 127)
                      SKIP

                  filtered.image [x.offset][y.offset] := BYTE (sum + 128)

              to.data.mgr ! image.filtered; block.number
              SEQ j = 0 FOR (block.hieght + 2)
                to.data.mgr ! filtered.row;
                              (block.width + 2)::filtered.image [j]

              from.data.mgr ? CASE image.saved
              to.data.mgr ! request.image       -- request image to process
:

}}}

----------------------------------------------------------------------------
--                              Configuration                             --
----------------------------------------------------------------------------

-- Links

VAL link0out IS 0 (INT) :
VAL link0in  IS 4 (INT) :
VAL link1out IS 1 (INT) :
VAL link1in  IS 5 (INT) :
```

```
VAL link2out IS 2 (INT) :
VAL link2in  IS 6 (INT) :
VAL link3out IS 3 (INT) :
VAL link3in  IS 7 (INT) :
VAL in.event IS 8 (INT) :


-- Logical Channels

CHAN OF from.file.mgr.msgs app.in :
CHAN OF to.file.mgr.msgs app.out :
[number.of.transputers] CHAN OF from.file.mgr.msgs down.links :
[number.of.transputers] CHAN OF to.file.mgr.msgs up.links :


-- Configuration

PLACED PAR
  PROCESSOR 0 T4
    PLACE app.in AT link1in :
    PLACE app.out AT link1out :
    PLACE down.links[0] AT link2out :
    PLACE up.links[0] AT link2in :

    data.manger (0, app.in, down.links[0], app.out, up.links[0])
    image.filter ()

  PLACED PAR i = 1 FOR (number.of.transputers - 1)
    PROCESSOR i T4
      PLACE down.links[i-1] AT link1in :
      PLACE up.links[i-1] AT  link1out :
      PLACE down.links[i] AT link2out :
      PLACE up.links[i] AT link2in :

      data.manager (i, down.links[i-1], down.links[i],
                       up.links[i-1], up.links[i])
      image.filter ()

}}}F
...F code HT8
--:::F netfilt.dcd
...F descriptor
--:::F netfilt.dds
...F link
--:::F netfilt.dlk
...F debug
--:::F netfilt.ddb
.
```

# Appendix D

This appendix contains a listing of the occam program, host.tsr, which interfaces between the cellular filter program, netfilt.tsr, running on a network of transputers, the user and the DOS file system.

```
{{{  EXE host
{{{F host
--:::F host.tsr
--------------------------------------------------------------------------------
--                                                                            --
--     File:   host.ocm                                Revision:   1.0        --
--     Author: David Sinclair                          Last Edit:  19Jul91    --
--                                                                            --
--     This program is the host for the network version of the cellular       --
--     filter.                                                                --
--                                                                            --
--------------------------------------------------------------------------------

{{{  define constants
--------------------------------------------------------------------------------
--                              define constants                              --
--------------------------------------------------------------------------------

VAL image.width IS 714 :
VAL image.hieght IS 340 :

VAL block.width IS 42 :
VAL block.hieght IS 20 :

VAL number.of.blocks IS ((image.width / block.width) *
                         (image.hieght / block.hieght)) :

}}}

{{{  define channel protocols
--------------------------------------------------------------------------------
--                          define channel protocols                          --
--------------------------------------------------------------------------------

PROTOCOL from.file.mgr.msgs
  CASE
    initialise; INT
    quit; INT
    hold; INT
    request.something; INT
    filter.param; INT; REAL32; REAL32; REAL32; INT
    image.row; INT; INT::[]BYTE
    image.saved; INT
:

PROTOCOL to.file.mgr.msgs
  CASE
    request.image; INT
    image.filtered; INT; INT
    filtered.row; INT; INT::[]BYTE
:

}}}

{{{  define channels
--------------------------------------------------------------------------------
--                              define channels                               --
--------------------------------------------------------------------------------
```

```
CHAN OF from.file.mgr.msgs to.workers :
CHAN OF to.file.mgr.msgs from.workers :

}}}

-----------------------------------------------------------------------------
--                              host process                               --
-----------------------------------------------------------------------------
£USE userio
£USE afhdr
£USE afiler

[number.of.blocks] BYTE image.map :
INT iterations:
INT in.file, out.file :
INT result, char, in.file.len, out.file.len :
[number.of.transputers] BOOL active.table :
[number.of.transputers] BOOL busy.processor :
[16] BYTE in.file.name :
[16] BYTE out.file.name :
[16] BYTE temp.file.name :
REAL k0, k1, k2 :

SEQ

  temp.file.name = "temp.img"

{{{  get user data

  goto.xy (screen, 0, 0)
  clear.eos (screen)
  goto.xy (screen, 0, 0)
  write.full.string (screen, "Cellular Filter Parameters")
  newline (screen)
  newline (screen)

  write.full.string (screen, "Number of iterations = ")
  char := 0
  read.echo.int (keyboard, screen, iterations, char)
  newline (screen)

  write.full.string (screen, "Filter Constant K0 = ")
  char := 0
  read.echo.real32 (keyboard, screen, k0, char)
  write.full.string (screen, "Filter Constant K1 = ")
  char := 0
  read.echo.real32 (keyboard, screen, k1, char)
  write.full.string (screen, "Filter Constant K2 = ")
  char := 0
  read.echo.real32 (keyboard, screen, k2, char)
  newline (screen)

  write.full.string (screen, "File to be filtered = ")
  char := 0
  read.echo.text.line (keyboard, screen, in.file.len, in.file.name, char)
  newline (screen)
  write.full.string (screen, "Filtered filename = ")
  char := 0
  read.echo.text.line (keyboard, screen, out.file.len, out.file.name, char)
  newline (screen)

}}}

{{{  open image, temporary and filtered image files

  open.file (from.filer, to.filer, in.file.name, BinaryByteStream.Access,
             Update.Mode, Old.File, 0, image.file, out.result)

  open.file (from.filer, to.filer, out.file.name, BinaryByteStream.Access,
             Update.Mode, New.File, 0, filtered.file, out.result)

  seek (from.filer, to.filer, image.file, 0, out.result)
  in.result := OperationOk
  WHILE (in.result = OperationOk)          -- need filtered file to be same
    SEQ                                    -- size as image file
      read.block (from.filer, to.filer, image.file, SIZE buffer, bytes.read,
                  buffer, in.result)
      write.block (from.filer, to.filer, filtered.file, buffer, bytes.read,
                   out.result)

  open.file (from.filer, to.filer, temp.file.name, BinaryByteStream.Access,
             Update.Mode, New.File, 0, temp.file, out.result)

  seek (from.filer, to.filer, image.file, 0, out.result)
  in.result := OperationOk
  WHILE (in.result = OperationOk)          -- need temporary file to be same
    SEQ                                    -- size as image file as well
      read.block (from.filer, to.filer, image.file, SIZE buffer, bytes.read,
                  buffer, in.result)
      write.block (from.filer, to.filer, temp.file, buffer, bytes.read,
                   out.result)

}}}

  to.workers ! initialise; number.of.transputers

  SEQ i = 0 FOR iterations

    PAR j = 0 FOR number.of.blocks         -- initialise the map of the
      image.map [j] := FALSE               -- filtered parts of the image

    PAR j = 0 FOR number.of.transputers    -- initialise table of active
      busy.processor [j] := FALSE          -- processor table

{{{  assign input and output file for current pass

      IF
        ((iterations /\ 1) = 1)               -- odd number of iterations
          IF
            ((i /\ 1) = 1)                    -- odd pass number
              PAR
                out.file := filtered.file
                IF
                  (i <> 1)
                    in.file := temp.file
        (i = 1)
          in.file := image.file

      ((i /\ 1) = 0)                      -- even pass number
        PAR
          out.file := temp.file
          in.file := image.file

    ((iterations /\ 1) = 0)               -- even number of iterations
      IF
        ((i /\ 1) = 1)                    -- odd pass number
          PAR
            out.file := temp.file
            IF
              (i <> 1)
                in.file := filtered.file
              (i = 1)
                in.file := image.file

        ((i /\ 1) = 0)                    -- even pass number
          PAR
            out.file := filtered.file
            in.file := temp.file

}}}

-- this is where the work is done

    SEQ j = 0 FOR nomber.of.processors
      to.workers ! request.something; j

    WHILE active
      from.workers ? CASE
        request.image; node.number
          SEQ
            index := 0
            WHILE (image.map [index] = FALSE) AND (index < number.of.blocks)
              index := index + 1
            IF
              index < number.of.blocks
                SEQ
                  to.workers ! filter.params; node.number; k0; k1; k2; index
                  offset.x := (block.width * (index REM (INT(image.width /
                                block.width)))) - 1
                  offset.y := ((index / (INT(image.width / block.width))) *
                                block.hieght) - 2

                  SEQ j = 0 FOR (block.hieght + 2)
                    SEQ
                      offset.y := offset.y + 1
                      start := 0
                      length := block.width + 2

                      IF
                        offset.x < 0
                          PAR
                            length := block.width + 1
                            start := 1
                    adjust := 1

                offset.x > (image.width - (block.width + 2))
                  PAR
                    block [block.width + 1] := BYTE (0)
                    length := block.width + 1

                (offset.x >= 0) AND (offset.x <= (image.width -
                                    (block.width + 2)))
                  SKIP

              IF
                (offset.y < 0) OR (offset.y >= image.hieght)
                  PAR l = 0 FOR (block.width + 2)
                    block [l] := BYTE (0)

                (offset.y >= 0) AND (offset.y < image.hieght)
                  SEQ
                    offset := ((offset.y * image.width) + offset.x) +
                              adjust
                    seek (from.filer, to.filer, in.file, offset,
                          result)
                    read.block (from.filer, to.filer, in.file,
                                length,byte.count, block, result)
                    IF
                      start = 1
                        SEQ
                          SEQ l = 0 FOR (block.width + 1)
                            block [(block.width + 2) - l] :=
                                    block [(block.width + 1) - l]
                          block [0] := BYTE (0)

              to.workers ! image.row; node.number; SIZE block::block


          busy.processor [node.number] := TRUE


      index >= number.of.blocks
        SEQ
          to.workers ! hold; node.number
          active := FLASE
          SEQ j = 0 FOR number.of.processors
            IF
              busy.processor [j] = TRUE
                active := TRUE

              busy.processor [j] = FALSE
                SKIP


image.filtered; node.number; index
  SEQ
                                     -- ignore the first row
    from.workers ? CASE filtered.row; node.number; byte.count::block
    offset.x := block.width * (index REM (INT(image.width /
                block.width)))
    offset.y := ((index / (INT(image.width / block.width)) *
                        block.hieght) - 1

            SEQ j = 0 FOR block.hieght
              SEQ
                offset.y := offset.y + 1
                from.workers ? CASE filtered.row; node.number;
                                                  byte.count::block
                offset := (offset.y * image.width) + offset.x
                length := block.width
                seek (from.filer, to.filer, out.file, offset, result)
                short.block IS [block FROM 1 FOR block.width] :
                write.block (from.filer, to.filer, out.file, short.block,
                             length, result)

                                             -- ignore the last row as well
            from.workers ? CASE filtered.row; node.number; byte.count::block


            to.workers ! image.saved; node.number
            busy.processor [node.number] := FALSE


{{{  shut down the processors tasks

  SEQ i = 0 FOR number.of.processors
    to.workers ! quit; i

}}}

{{{  close image, temporary and filtered image files

  close.file (from.filer, to.filer, in.file, Close.Option, result)
  close.file (from.filer, to.filer, out.file, Close.Option, result)
  close.file (from.filer, to.filer, temp.file, CloseDel.Option, result)

}}}

}}}F
...F code HT8
--:::F host.dcd
...F descriptor
--:::F host.dds
...F link
--:::F host.dlk
...F debug
--:::F host.ddb
```