Chapter 1

Parallel Image Processing

1.1 Declaring an image data type

Vector Pascal does not have a pre-declared image data type. However one can readily declare one. There are two common approaches to representing full-colour image data. In both of them the colour is represented as 3 components each of 8 bit precision.

1. Display manufactures for PCs usually store the information as two dimensional arrays of 24 bit or 32 bit pixels, made up of red, green and blue fields with an optional alpha field for colour blending. The fields typically contain 8 bit unsigned numbers with 0 representing minimum brightness of the colour and 255 representing the maximum brightness. This approach simplifies display design but is not so suitable for image processing.

2. The alternative approach separates the colour information out into distinct planes, so that a colour picture is manipulated as three distinct monochrome images, one of which represents the red component, one the green and one the blue. This approach allows image processing procedures designed to operate on monochrome images to be applied un-modified to each of the planes of a colour image.

In what follows we use the colour plane model for images[1].

```
type
      image(maxplane,maxrow,maxcol:integer)=
        array[0..maxplane,0..maxrow,0..maxcol]of pixel;
```

This declares an image to be a parameterised data type with a variable number of image colour planes and a variable number of rows and columns. Whilst this definition will store also pixels in an 8 bit representation, it is as a signed 8-bit binary fraction in the range -1..1, instead of as 8-bit unsigned integers.

[1] The definition of the image type along with several of the functions over images are given in Unit Bmp, shown in chapter ??.

1.2 Brightness and contrast adjustment

The signed fractional representation of pixels lends itself well to image processing applications where arithmetic is done on pixels. We frequently want to subtract images from one another. Doing this can give rise to negative valued pixels. Using an unsigned format, negative pixels have no natural representation. Using signed pixels, 0 represents mid grey, -1 represents black and 1 white. This representation allows the contrast of an image to be adjusted simply by multiplying by a constant. Thus if we multiply an image by 0.5 we halve its contrast. If we multiply it by -1 we convert it to an negative image, etc.

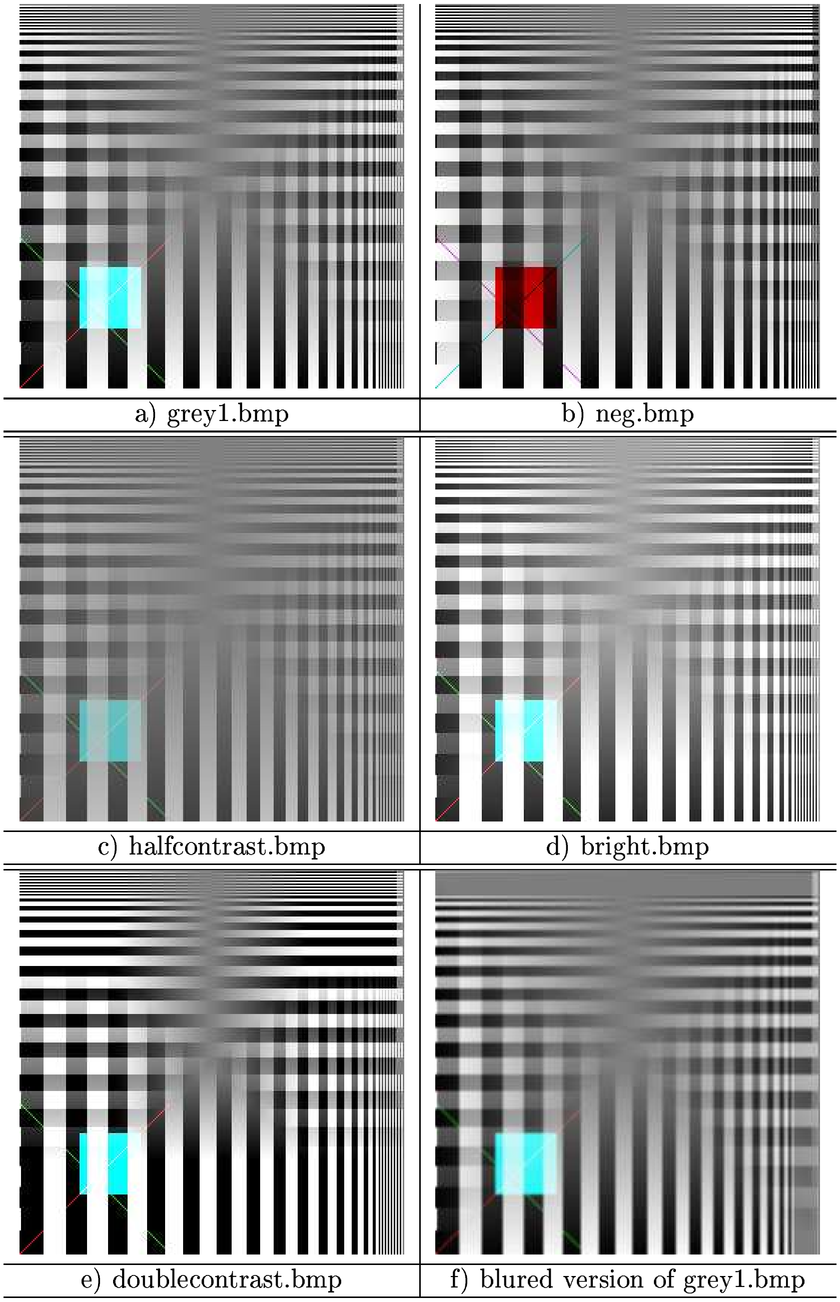
1.2.1 Efficiency in image code

Algorithm 1 illustrates how easy it is to alter the brightness/contrast of an image by adding/multiplying it with a real value. Although concise this does not necessarily produce the fastest code. The rules used in expression evaluation mean $im\uparrow\times 0.5$ is expanded out to $im\uparrow_{\iota_0,\iota_1\iota_2}\times 0.5$ which is a multiplication of a pixel by a real. Since reals are higher precision the pixel has to be promoted to a real before the multiplication. This effectively prevents the original array expression being vectorised.

Algorithm 1 Which shows simple manipulations of image contrasts and brightnesses. The type pimage used, is a pointer to an image.

```
program contrast ;
uses bmp ;
var
    Let im, outim ∈ pimage;
begin
    if loadbmpfile('grey1.bmp',im)then
    begin
        new ( outim ,im ^ .maxplane , im ^ .maxrow , im ^ .maxcol );
        outim↑← im↑ × - 1.0 ;                                negate image
        storebmpfile ( 'neg.bmp' , outim↑);
        outim↑← im↑ × 0.5 ;                                  halve contrast
        storebmpfile ( 'half.bmp' , outim↑);
        outim↑← im↑ + 0.3 ;                                  brighten
        storebmpfile ( 'bright.bmp' , outim↑);
    end
    else writeln( 'failed to load file' );
end .
```

A more efficient approach is seen in the procedure adjustcontrast shown in Algorithm 2, where a vector of pixels is initialised to hold the adjustment factor. By holding it as a vector of fixed point numbers the operation can be



a) grey1.bmp

b) neg.bmp

c) halfcontrast.bmp

d) bright.bmp

e) doublecontrast.bmp

f) blured version of grey1.bmp

Figure 1.1: Test images used to illustrate brightness, contrast adjustment and filtering. The images a..e are produced by the program graphio.

effectively vectorised on MMX based processors[2]. Since the fixed point pixel format only works for $|f| \leq 1$ it is necessary to use floating point multiplication when increasing the image contrast.

Algorithm 2 A more efficient way of adjusting contrast. Note that in this example the type line refers to a vector of pixels.

```
procedure adjustcontrast ( f :real ; var src ,dest :image );
var
    Let l ∈ ^line;
    Let r ∈ real;
begin
    new ( l ,src .maxcol );
    {$r-}
    l↑← f;
    if (abs (f) < 1) then dest← src × l↑
    else dest← src × f;
    {$r+}
    dispose ( l );
end ;
```

Recall that pixels are represented as signed 8-bit numbers, with the conceptual value 1.0 being encoded as +127 and the conceptual value -1.0 being encoded as -128. Multiplication of pixels proceeds by :

1. Multiplying the 8-bit numbers to give a 16-bit result.
2. Shifting the result right arithmetically by 7 places.
3. Selecting the bottom 8 bits of the result

The 8-bit signed format contains 7 bits of significance plus the sign bit, the 16-bit result contains 14 bits of significance plus 2 replicated sign bits. It is clear that this format can not represent multiplication by a number greater than 1.

1.3 Image Filtering

As another practical example of Vector Pascal we will look at an image filtering algorithm. In particular we will look at applying a separable 3 element convolution kernel to an image. We shall initially present the algorithm in standard Pascal and then look at how one might re-express it in Vector Pascal. The entire program is shown in figure 5 and then developed in figures 4, 6 and **??**.

[2] It is a weakness of the Intel MMX instruction-set that it does not support scalar to vector operations. There are no instructions to operate between a signed byte and a vector of signed bytes. Motorola processors do not suffer this weakness.

Convolution of an image by a matrix of real numbers can be used to smooth or sharpen an image, depending on the matrix used. If A is an output image, K a convolution matrix, then if B is the convolved image

$$B_{y,x} = \sum_i \sum_j A_{y+i,x+j} K_{i,j}$$

A separable convolution kernel is a vector of real numbers that can be applied independently to the rows and columns of an image to provide filtering. It is a specialisation of the more general convolution matrix, but is algorithmically more efficient to implement. If $\mathbf{k}$ is a convolution vector, then the corresponding matrix K is such that $K_{i,j} = \mathbf{k}_i \mathbf{k}_j$.

Given a starting image A as a two dimensional array of pixels, and a three element kernel c_1, c_2, c_3, the algorithm first forms a temporary array T whose whose elements are the weighted sum of adjacent rows $T_{y,x} = c_1 A_{y-1,x} + c_2 A_{y,x} + c_3 A_{y+1,x}$. Then in a second phase it sets the original image to be the weighted sum of the columns of the temporary array: $A_{y,x} = c_1 T_{y,x-1} + c_2 T_{y,x} + c_3 T y, x + 1$. Clearly the outer edges of the image are a special case, since the convolution is defined over the neighbours of the pixel, and the pixels along the boundaries a missing one neighbour. A number of solutions are available for this, but for simplicity we will perform only vertical convolutions on the left and right edges and horizontal convolutions on the top and bottom lines of the image.

1.3.1 Blurring

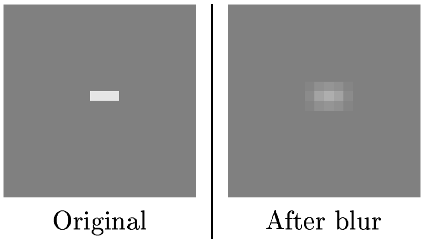
An image can be blurred using the separable filter (0.25,0.5,0.25). Consider that this implies: each row in the output image is formed by a mixture of itself and the rows above and below, with half the amplitude of the signal coming from the current row and half from the adjacent rows. Similarly, each column is made up of half from the current column and half from the adjacent column. The net result is that a pixel's influence spreads out over a 3×3grid. We can examine the effect of the filter on a point source. Here a single pixel that stands out against a uniform background in the initial image shows how the initial pixel spreads out to affect the region around. This is shown in Fig. 1.2.

Figure 1 shows the effect of using this filter on the classic 'Mandrill' test image.

1.3.2 Sharpening

If we use a filter that has negative weights away from the center, the effect is to sharpen an image. Suppose we apply the filter (-0.25,1.0,-0.25) to an image, what will be the result?

The first thing to note is that this filter is non-unitary, that is to say is coefficients do not add up to 1. If we use a unitary filter like the blur (0.25,0.5,0.25) , the mean contrast of the image is unchanged.



Original

After blur

Figure 1.2: The effect of a blurring filter on a finite impulse.

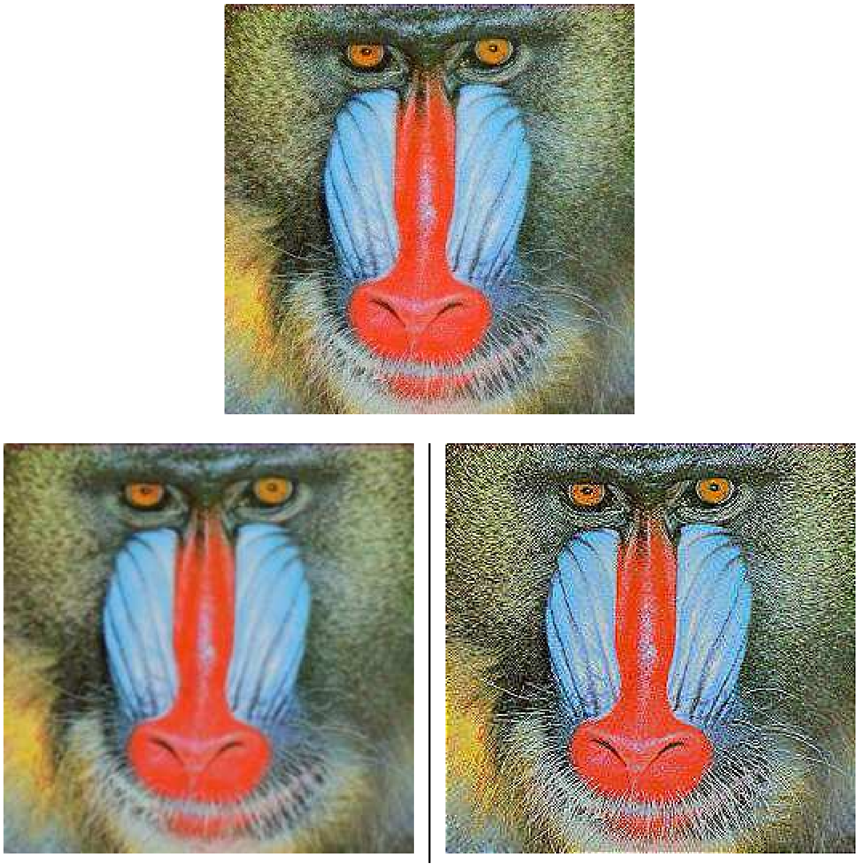


Figure 1.3: The image at the top is the original, the bottom left image has been subjected to a blurring filter (0.25,0.5,0.25), that on the right to a sharpening filter.

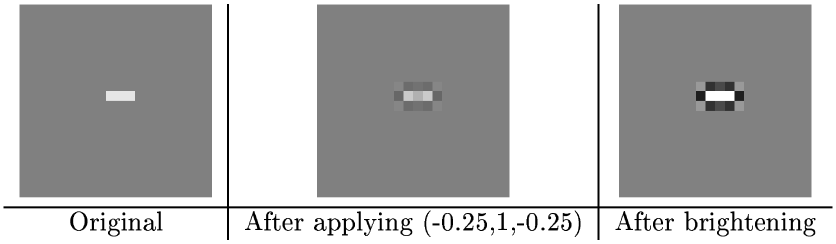

Original | After applying (-0.25,1,-0.25) | After brightening

Figure 1.4: Effect of a sharpening filter on a finite impulse.

Since the coefficients of our sharpening filter sum to 0.5 and since the filter is applied twice, once vertically and once horizontally the net effect is to reduce mean contrast to a quarter of what it was originally. This is shown in Fig 1.4. To compensate we must multiply the image by 4.0 to restore the original contrast as shown in Algorithm 3. Note the characteristic 'ringing' induced in the image by sharpening filters. Figure 1 shows how the picture of a Mandrill can be sharpened. Note that over the fur, the effect of sharpening is just to introduce noise. This is for two reasons:

1. This algorithm results in the loss of two bits of precision when the multiplication by 4 takes place, the effect is to introduce additional quantization noise.

2. Sharpening is only visually effective where an feature with high spatial frequency occurs against a background with lower spatial frequency. The hair area is all of high spatial frequency. In consequence, the ringing produced by sharpening overlaps with other hairs, occluding them.

Algorithm 3 The sharpening method.

```
procedure sharpen ( var im :image );
var
    Let $i \in$ integer;
begin
    $i \leftarrow$ 1;
    pconv (im, - 0.25 , 0.998, - 0.25 );
end ;
```

1.3.3 Comparing Implementations

Algorithm 4 shows `conv` an implementation of the convolution in Standard Pascal. The pixel data type has to be explicitly introduced as the subrange

-128..127. Explicit checks have to be in-place to prevent range errors, since the result of a convolution may, depending on the kernel used, be outside the bounds of valid pixels. Arithmetic is done in floating point and then rounded.

Algorithm 4 Standard Pascal implementation of the convolution

```
procedure conv (c1, c2, c3 : real);
var
    tim :array [0..m ,0..m ]of pixel ;
    Let quarter, half, temp $\in$ real;
    Let i, j $\in$ integer;
begin
    for i$\leftarrow$ 1 to m - 1 do
        for j$\leftarrow$ 0 to m do
        begin
            temp$\leftarrow$ $theim_{i-1,j} \times c1 + theim_{i,j} \times c2 + theim_{i+1,j} \times c3$;
            if temp > 127 then temp$\leftarrow$ 127 else
                    if temp < - 128 then temp$\leftarrow$ - 128 ;
            $tim_{i,j}$ $\leftarrow$ round(temp);
        end ;
        for j$\leftarrow$ 0 to m do
        begin
            $tim_{0,j}$ $\leftarrow$ $theim_{0,j}$;
            $tim_{m,j}$ $\leftarrow$ $theim_{m,j}$;
        end ;
        for i$\leftarrow$ 0 to m do
        begin
            for j$\leftarrow$ 1 to m - 1 do
            begin
                temp$\leftarrow$ $tim_{i,j-1} \times c1 + tim_{i,j+1} \times c3 + tim_{i,j} \times c2$;
                if temp > 127 then temp$\leftarrow$ 127 else
                        if temp < - 128 then temp$\leftarrow$ - 128 ;
                $tim_{i,j}$ $\leftarrow$ round(temp);
            end ;
            $theim_{i,0}$ $\leftarrow$ $tim_{i,0}$;
            $theim_{i,m}$ $\leftarrow$ $tim_{i,m}$;
        end ;
end ;
```

Because ISO Pascal does not support dynamic arrays the image sizes both in this version and the parallel version are statically declared.

Image processing algorithms lend themselves particularly well to data-parallel expression, working as they do on arrays of data subject to uniform operations. Algorithm 7 shows a data-parallel version of the algorithm, `pconv`, implemented in Vector Pascal. Note that all explicit loops disappear in this version, being replaced by assignments of array slices. The first line of the algorithm initialises three vectors `p1, p2, p3` of pixels to hold the replicated copies of the

Algorithm 5 The program dconv, a test harness for image convolution written to work under several Pascal compilers.

```
program dconv ;
const
    m =255;
    repeats =400;
type
    pixel = -128..127;
    tplain = array [0..m ,0..m ] of pixel ;
var
    Let theim, theres ∈ tplain;
    Let i ∈ integer;
    Let oldtime, ops ∈ real;
procedure showtime ; (see Figure 6 )
procedure conv ( c1 ,c2 ,c3 :real ); (see Figure 4 )

begin
    oldtime← secs;
    ops← $12 \times (m + 1) \times (m + 1) \times repeats$;
    for i← 1 to repeats do conv (0.2, 0.6, 0.2);
    showtime;
    writeln( ‘done’ , secs);
end .
```

Algorithm 6 The procedure showtime.

```
procedure showtime;
var
    Let sec, duration, rate ∈ real;
begin
    sec← secs;
    duration← sec - oldtime;
    write(duration, ‘ ’ );
    rate← $\frac{ops}{duration}$;
    write($\frac{rate}{1000000}$, ‘M ops per sec’ );
    oldtime← sec;
end ;
```

Table 1.1: Comparative Performance on Convolution

Algorithm	Implementation	Target Processor	Million Ops Per Second
conv	Borland Pascal	286 + 287	6
	Vector Pascal	Pentium + MMX	61
	DevPascal	486	62
	Delphi 4	486	86
pconv	Vector Pascal	486	80
	Vector Pascal	Pentium + MMX	820

Measurements done on a 1GHz Athlon, running Windows 2000.

Algorithm 7 Vector Pascal implementation of the convolution

```
procedure pconv ( var theim :tplain ;c1 ,c2 ,c3 :real );
var
    tim :array [0..m ,0..m ]of pixel ;
    Let p1, p2, p3 ∈ array[0..m]of pixel;
begin
    p1← c1;
    p2← c2;
    p3← c3;
    $tim_{1..m-1} \leftarrow theim_{0..m-2} \times p1 + theim_{1..m-1} \times p2 + theim_{2..m} \times p3$;
    $tim_{0} \leftarrow theim_{0}$;
    $tim_{m} \leftarrow theim_{m}$;
    $theim_{0..m,1..m-1} \leftarrow tim_{0..m,0..m-2} \times p1 + tim_{0..m,2..m} \times p3 + tim_{0..m,1..m-1} \times p2$;
    $theim_{0..m,0} \leftarrow tim_{0..m,0}$;
    $theim_{0..m,m} \leftarrow tim_{0..m,m}$;
end ;
```

kernel coefficients `c1, c2, c3` in fixed point format. These vectors are then used to multiply rows of the image to build up the convolution. The notation `theim[][1..maxpix-1]` denotes columns 1..maxpix-1 of all rows of the image. Because the built in pixel data type is used, all range checking is handled by the compiler. Since fixed point arithmetic is used throughout, there will be slight rounding errors not encountered with the previous algorithm, but these are acceptable in most image processing applications. Fixed point pixel arithmetic has the advantage that it can be efficiently implemented in parallel using multi-media instructions.instruction-set

It is clear that the data-parallel implementation is somewhat more concise than the sequential one, 12 lines with 505 characters compared to 26 lines with 952 characters. It also runs considerably faster, as shown in table 1.1. This expresses the performance of different implementations in millions of effective arithmetic operations per second. It is assumed that the basic algorithm requires 6 multiplications and 6 adds per pixel processed. The data parallel algorithm runs 12 times faster than the serial one when both are compiled using Vector Pascal and targeted at the MMX instruction-set. The `pconv` also runs a third faster than `conv` when it is targeted at the 486 instruction-set, which in effect, serialises the code.

For comparison `conv` was run on other Pascal Compilers[3], DevPascal 1.9, Borland Pascal and its successor Delphi[4]. These are extended implementations, but with no support for vector arithmetic. Delphi is a state of the art commercial compiler, as Borland Pascal was when released in 1992. DevPas is a recent free compiler. In all cases range checking was enabled for consistency with Vector Pascal. The only other change was to define the type pixel as equivalent to the system type shortint to force implementation as a signed byte. Delphi runs `conv` 40% faster than Vector Pascal does, whereas Borland Pascal runs it at only 7% of the speed, and DevPascal is roughly comparable to Vector Pascal.

1.4 genconv

The convolution algorithms presented so far use one dimensional kernels and work by being applied sucessively in vertical and horizontal directions. As such they are unable to deal with asymetrical kernels - ones which blur in one direction and sharpen in another for instance. They also, because they use 8-bit pixel multiplication, suffer from rounding errors when using sharpening convolutions.

We will now present

procedure *genconv* (**var** *p* :*image* ; **var** *K* :*matrix*);

which computes a general convolution on an image *p* producing a modified image

[3] In addition to those shown the tests were performed on PascalX, which failed either to compile or to run the benchmarks. TMT Pascal failed to run the convolution test.

[4] version 4

q such that if

$$q_{i,j,k} = \sum_x \sum_y p_{i,j+y-a,k+x-b} \times K_{x,y}$$

where $a = (K.rows)div2$ and $b = (K.cols)div2$. At the end p is updated with q.

Genconv allows an image to be convolved with an arbitrary two dimensional matrix of real numbers. If one performs this operation naively with an $n \times n$ matrix of reals against an image of dimensions $r \times c$ then the algorithmic complexity will be $\mathbf{O}rcn^2$, since each each output pixel is the result of multiplying n^2 input pixels by kernel components.

However it is worth observing that for most practical convolutions, there are repeated matrix elements in the kernel. A 9 element matrix might contain only 4 distinct values. We can take advantage of this by analysing the matrix to determine how many unique components it has and then forming pre-multiplied copies of the input image, one for each unique matrix element in the kernel. Appropriate selection from these premultiplied copies allows us to compute the convolution.

Let us define a couple of types and a variable to help with this:

```
type
    premult(rows,cols:integer )= array [1..rows ,1..cols ] of pimage ;
    tflag(rows,cols:integer )= array [1..rows ,1..cols ] of boolean ;
var
    Let f ∈ ^premult;
    Let a, b, i, j ∈ integer;
    Let flags ∈ ^tflag;
```

We will use f to hold the premultiplied versions of the image such that $f_{i,j} = p \times K_{i,j}$. The algorithm for constructing the premultiplied matrix of images will avoid carrying out redundant multiplications.

a,b store the steps away from the center of the kernel.

flags[i,j] is true if f[i,j] holds the first pointer to a premultiplied image.

1.4.1 dup

This function returns true if there exists a m, n such that

$$n + m \times K.cols < j + i \times K.cols$$

and

$$K_{m,n} = K_{i,j}$$

in other words, if the matrix element $K_{i,j}$ is preceeded in the matrix by an identical element. If that is true, then the element $K_{i,j}$ is a duplicate and this fact can be taken advantage of in reducing the amount of pre-multiplication required to perform the convolution

Algorithm 8 Main body of the generalised convolution

```
function dup ( i ,j :integer ):boolean ; (see Section 1.4.1 )
function prev ( i ,j :integer ):pimage ; (see Section 1.4.2 )
function pm ( i ,j :integer ):pimage ; (see Section 1.4.3 )
procedure doedges ; (see Section 1.4.4 )
procedure freestore ; (see Section 1.4.5 )

begin
    new ( f ,K . rows ,K . cols );
    f↑← nil;
    new ( flags ,K .rows ,K .cols );
    flags↑← false;

    for i← 1 to K.rows do
        for j← 1 to K.cols do
        else f↑[i, j]← pm (i, j);
```

The loops above perform the premultiplication of the input image to form the matrix of images. If item $K_{i,j}$ is a duplicate then we use a previous premultiply else we perform the premultiply now.

```
        a← $\frac{K.rows}{2}$;
        b← $\frac{K.cols}{2}$;
        p [][a ..p .maxrow -a ,b ..p .maxcol -b ]:=0;
        for i← 1 to K.rows do
            for j← 1 to K.cols do
                p [][a ..p .maxrow -a ,b ..p .maxcol -b ]:= p [][a ..p .maxrow -a ,b ..p .maxcol -b ] +f ^ [i ,j
                ]^ [iota 0, i + iota 1 -a ,j +iota 2 -b ];
```

The above line forms the convolution by replacing the central region of the image with the sum of the shifted premultiplied images.

```
    doedges;

    freestore;
end ;
```

Algorithm 9 The function which checks for duplicate kernel elements.

```
function dup ( i ,j :integer ):boolean ;

var
    Let $c$, $d$, $l$, $m \in$ integer;
    Let $b \in$ boolean;
begin
    $c \leftarrow K.cols$;
    $d \leftarrow j + i \times c$;
    $b \leftarrow false$;
    for $l \leftarrow 1$ to $c$ do for $m \leftarrow 1$ to $k.rows$ do
        $b \leftarrow b \vee (K_{i,j} = K_{m,l}) \wedge (m + c \times l < d)$;
    $dup \leftarrow b$
    { dup:=\or \or ((K[i,j]=K)and(iota 1 +c*iota 0<d));}
```

The Vector Pascal statement is more or less a direct translation of the mathematical formulation of the problem. We use or-reduction over both axes of the matrix to search for duplicates.

```
end ;
```

1.4.2 prev

For duplicated matrix elements $K_{i,j}$ function prev returns the pre-multiplied version of the image that was previously computed for this value of matrix element.

This uses classical Pascal constructs to search the matrix for the position of the premultiplied duplicate and then assigns the duplicate to the return value of the function. Note that the function does not return when the assignment is made.

1.4.3 pm

The function pm, (shown in Algorithm 11) premultiplies the image by the real valued coefficient $K_{i,j}$ returning a new image. The fact that a new pre-multiplied image has been created is recorded in the flags matrix.

Algorithm 10 Function to find a previous instance of a kernel element.

```
function prev ( i ,j :integer ):pimage ;
var
    Let $m$, $n \in$ integer;
    Let $s \in$ real;
begin

    $s \leftarrow k_{i,j}$;
    for $m \leftarrow 1$ to $i$ - 1 do
        for $n \leftarrow 1$ to $K.cols$ do
            if $K_{m,n} = s$ then
                $prev \leftarrow f\uparrow$[m, n];
    for $n \leftarrow 1$ to $j$ - 1 do
        if $K_{i,n} = s$ then
            $prev \leftarrow f\uparrow$[i, n];
end ;
```

1.4.4 doedges

When performing a convolution on an image, the edges always pose a problem. The convolution operation determines the value of each output image from the corresponging neighbourhood in the input image. Around the edges only part of this neigbourhood exists. Some strategies that can be adopted here are:

1. One can treat the image as a being topologically equivalent to a torus so that upper the neighbourhood of pixel on the top line of the image continues onto bottom lines of the image. This approach is computationally

Algorithm 11 The premultiplication function.

```
function pm ( i ,j :integer ):pimage ;
var
    Let $x \in$ pimage;
begin

    new ( x ,p .maxplane ,p .maxrow ,p .maxcol );
    adjustcontrast ($K_{i,j}$, p, x↑);
    flags↑[i, j]← true;
    pm← x;
end ;
```

easy : when finding the neighbours of pixel $p_{i,j}$ we would normally do this by using the expression $p_{i+y,j+x}$ iterating over a range of values of x and y. To treat the image as a torus we substitute the indexing expression `p[(i+y)mod p.rows,(j+x)mod p.cols]`. Although this is computationally easy, it does not make a great deal of sense, since it allows output pixels to be influenced by input pixels in the parts of the picture that are furthest away from it.

2. One can mirror the original image around all four edges so that on, for instance, the top edge the upper neighbour of a pixel is the same as its lower neighbour. This makes more sense than using a toroidal topology, and will work well for where the edge of the image is intersected by a feature that runs a right angles to the edge.

3. One can assume that the edge pixels themselves are replicated to an arbitrary degree beyond the edge itself, and compute the edge convolution on this basis. This is the most parsimonious assumption, and is the one we use here.

If we have a 5 × 5 convolution matrix and a 100 × 100 image, then we will have a central subregion of the output image : `q[2..97,2..97]` which can be evaluated from the full convolution matrix. The 2-pixel wide vetical margins can be expressed a sum of columns of images within the premultiplied image matrix. Thus the 0th output column is the sum of the 0th image columns within the first three columns of the premultiplication matrix plus the 1st image column of the 4th column of the premultiplied image matrix and the 2nd image column of the 5th column of the premultiplied image matrix, etc. Processing the edges takes many more lines of code because it is a mass of special cases.

Algorithm 12 The edge processing algorithm

```
procedure doedges ;
var
    Let i, j, l, m, n, row, col ∈ integer;
    Let r ∈ pimage;
begin                                                        $r-
    j← $\frac{k.rows}{2}$;
    i← $\frac{k.cols}{2}$;
    p [][0..j -1]:=0;
    p [][][0..i -1]:=0;
    p [][1+p .maxrow -j ..p .maxrow ]:=0;
    p [][][1+p .maxcol -i ..p .maxcol ]:=0;
    for n← 0 to p.maxplane do                                iterate through the planes
        for l← 1 to k.rows do
            for m← 1 to k.cols do
            begin
                r← f↑[l, m];
                for row← 0 to j - 1 do
                    $p_{n,row}$← $p_{n,row}$ + r↑[n, (row + l - j - 1)];     top
```

The line above computes the convolution for the top edge, so that the neighbours above the top are replaced by the corresponding elements of the pre-multiplied top scan-line.

```
                for row← p.maxrow - j + 1 to p.maxrow do
                    $p_{n,row}$← $p_{n,row}$ + r↑[n, (row + l - j - 1)];     bottom
            for col← 0 to i - 1 do for row← 0 to p.maxrow do begin begin
                    $p_{n,row,col}$← r↑[n, row, (col + 1 + m - i)] + $p_{n,row,col}$;     left
                end ;
```

Using a similar technique we compute the convolution for the left edge. Note that the construct `p[n][][col]` means for plane n select the column col from all rows.

```
            for col← 1 + p.maxcol - i to p.maxcol do for row← 0 to p.maxrow do begin begin
                    $p_{n,row,col}$← $p_{n,row,col}$ + r↑[n];     right
                end ;
                {$r+}
            end ;
end ;
```

1.4.5 freestore

The first occurrence of of an image in the pre-multiplied image matrix is disposed of. The record in the flags matrix, initialised when pre-multiplication occured is used to keep track of this.

Algorithm 13 The release of temporary store.

```
procedure freestore ;
var
    Let $i, j \in$ integer;
begin
    for $i \leftarrow 1$ to K.rows do
        for $j \leftarrow 1$ to K.cols do
            if flags↑[i, j]
            then dispose(f↑[i, j]);
end ;
```

1.5 Digital Half-toning

Printing images on paper requires that they be converted into a dot pattern since it is not practical to print with ink of varying shades of grey. Since a digital image may have a range of grey values one has to map these to dots in such a way that the average darkness of the dots over a small area of the paper is the same as the average darkness of the corresponding area of the image. In this section we present two algorithms to achieve this, one is parallel and the other inherently sequential.

1.5.1 Parallel Halftone

Algorithm 14 is a parallel technique for half toning.

It involves defining a mask of pixels of varying brightnesses and comparing the image with this mask. If a pixel is darker than the corresponding mask position it is printed as black and otherwise as white. The effect is shown in Fig. 1.6. The mask is chosen to be 8 bytes long to ensure that the operation will parallelise in the MMX registers. The mask is combined with the picture using modular arithmetic on the indices ι_0, ι_1.

1.5.2 Errordifuse

It is clear that simply masking, whilst quick, yields annoying artifacts since the human eye is well able to pick out the repetitive motifs embedded within the mask. Another disadvantage is that the mask will approximate the brightness

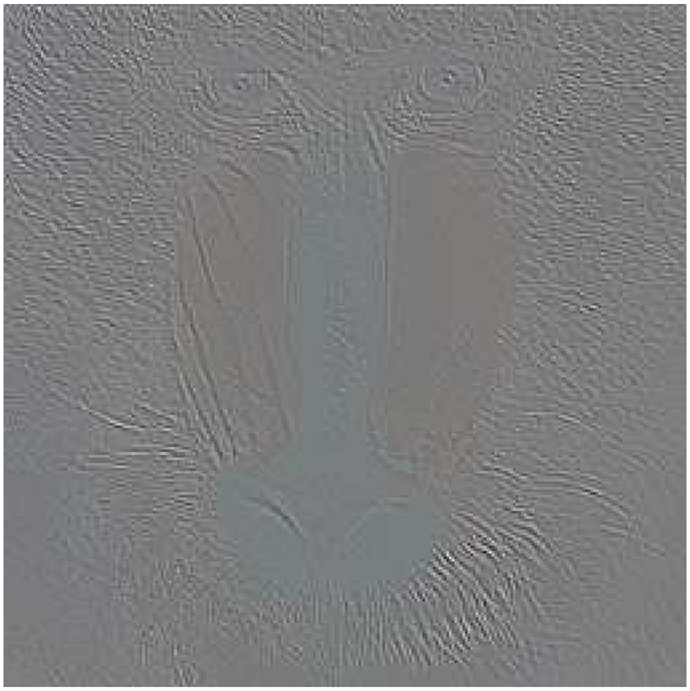


Figure 1.5: Effect of applying a diagonal edge detection filter to Mandrill.

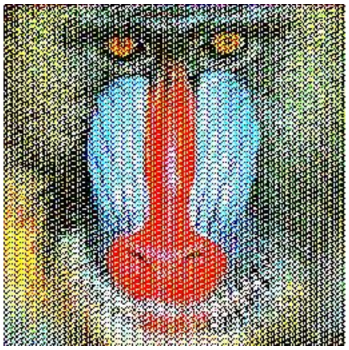


Figure 1.6: Mandrill rendered with a 4 × 8mask.

Algorithm 14 Parallel half toning using a fixed mask

```
procedure halftone ( var src ,dest :image );
const
    black :pixel =-1.0;
    white :pixel =1.0;
    pattern: array [0..3,0..7] of pixel =( ( 0.75,-0.95,0.0,0.5,-0.3,0.33,-0.2,-0.7),
    ( 0.62,-0.75,-0.1,-0.45,0.8,0.25,0.95,-0.6),
    ( -0.15,0.3,0.4,-0.8,-0.9,-0.5,0.15,0.17),
    ( -0.25,0.9,0.7,-0.33,-0.4,0.2,0.1,-0.82));

begin
    $dest \leftarrow pattern_{\iota_1 \,\mathbf{mod}\, 4, \iota_2 \,\mathbf{mod}\, 8}$;
    $dest \leftarrow \begin{cases} white & \text{if } src > dest \\ black & \text{otherwise} \end{cases}$ ;
end ;
```

of the picture with a spatial wavelength equivalent to twice the size of the mask itself. It thus responds poorly to sharp edges.

If one is willing to sacrifice parallelism, error diffusion techniques yield a much better result, as is shown in Fig. 1.7.

Algorithm 15 compensates for the quantization errors by adjusting the likelihood of using black or white for neighboring pixels. Once it has decided whether to render a pixel in black of white, it computes the quantization error in `e1`. This error term is then spread around the pixels to the right and below by subtracting weighted components of it to a temporary source image.

When the corresponding pixels in the temporary source come to be processed, the likelihood of their being rendered black or white is now biased away

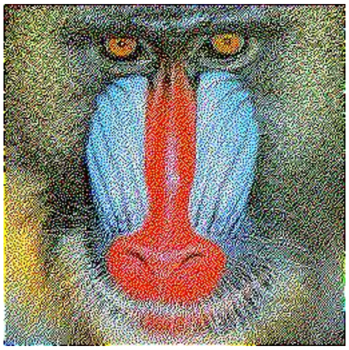


Figure 1.7: Mandrill rendered using error diffusion.

Algorithm 15 Classic error diffusion, non parallel code.

```
procedure errordifuse ( var src ,dest :image );
var
    Let temp ∈ ^image;
    Let i, j, k ∈ integer;
    Let black, white ∈ pixel;
    Let e1, e2, e3 ∈ real;
    Let r1, r2 ∈ integer;
begin
    black← - 1.0 ;
    white← 1.0;
    new ( temp , src .maxplane , src .maxrow ,src .maxcol );
    $dest \leftarrow \begin{cases} 1.0 & \text{if } src > 0 \\ -1.0 & \text{otherwise} \end{cases}$ ;
    temp↑← src;
    for k← 0 to src.maxplane do
        for i← 1 to src.maxrow - 1 do for j← 1 to src.maxcol - 1 do
        begin
            r1← random;
            r2← random;
            $e3 \leftarrow \begin{cases} 0.2 & \text{if } r1 > r2 \\ -0.2 & \text{otherwise} \end{cases}$ ;
            $dest_{k,i,j} \leftarrow \begin{cases} white & \text{if } temp\uparrow[k, i, j] > 0.0 \\ black & \text{otherwise} \end{cases}$ ;
            e1← $dest_{k,i,j}$ - temp↑[k, i, j];
            temp↑[k, i, j + 1]← temp↑[k, i, j + 1] - (0.45 - e3) × e1;
            temp↑[k, i + 1, j]← temp↑[k, i + 1, j] - (e3 + 0.375) × e1;
            temp↑[k, i + 1, j - 1]← temp↑[k, i + 1, j - 1] - (0.125) × e1;
        end ;
        dispose ( temp );
end ;
```

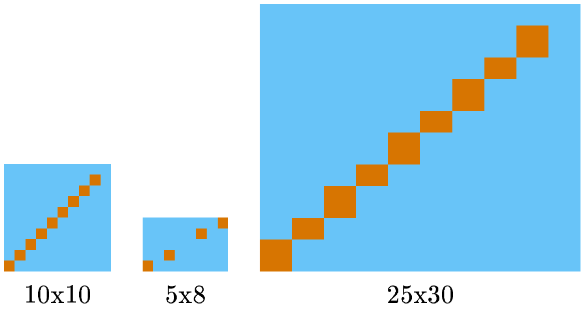


Figure 1.8: Naive re-sampling used to scale pictures introduces artifacts.

from its original value by this error term.

Suppose a pixel had the value 0.2 and was rendered as 1.0. The error term `e1` would be 0.8, which would be subtracted from the surrounding pixels. Sufficient might be added to the pixel to the right to trip it from its original rendering as white to a rendering as black.

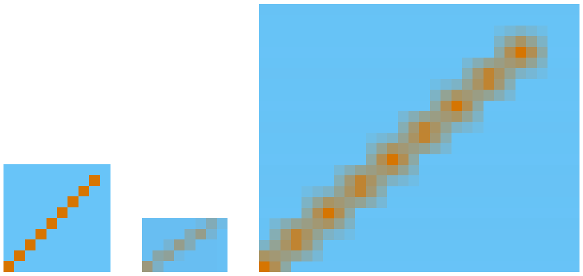
The way in which the errors are distributed is randomised using the term `e3`. In the absence of this random term one gets visually intrusive 'brain coral' patterns in the half toning.

1.6 Image Resizing

A very common operation in dealing with images is to resize them, making them larger or smaller. This may be done either uniformly - preserving their aspect ratio - or unevenly so that the shape as well as the size of the image changes.

In a naive resizing algorithm we simple scale the indices of the pixels in the source image by the ratio of the images sizes. Suppose we wanted to halve the size of an image, then we could simply select every second pixel. As can be seen in Fig. 1.8, a number of unpleasant artifacts occur with this method. When shrinking an image, thin lines can loose pixels, or even vanish. When enlarging an image, what were originally square pixels become oblong, something which is particularly disconcerting when looking at text. Collectively these errors are called aliasing.

The removal of these artifacts is termed anti-aliasing. The artifacts arise because of the spatial frequencies possible in pictures of different sizes. The notion of spatial frequency is illustrated by the test image shown in Fig. 1.1. These show horizontal and vertical gratings of varying spatial frequency. The Nyquist theorem states that the maximum spatial frequency, measured in oscillations per inch, that can be supported by an image is half the number of pixels per inch. The highest frequency in the images shown in Fig. 1.1 correspond to this limit. If we apply the blurring convolution [0.25,0.5,0.25] to the test image, Fig. 1.1.a to produce image Fig. 1.1.f, we have the effect of mak-



10 × 10 5 × 8 25 × 30

Figure 1.9: Anti-aliased rescaling using blurring and interpolation reduces artifacts.

ing the highest spatial frequency invisible. Thus the blurring convolution can be viewed as a subtractive spatial frequency filter that selectively removes the highest frequency information.

Now consider what happens when one increases the size of an image. The effect is to introduce a new spatial frequency bands into the image. Since these frequencies are higher than any that we have had up to now, what will occupy them?

If we use a naive sampling algorithm, simply replicating each original pixel, the higher frequency bands are populated with Moiré fringe noise, generated by the interference between the old Nyquist limit frequency and the new Nyquist limit frequency. What we want instead, is for these wavebands to be empty. We can achieve this by using an interpolation procedure which fills in new pixel positions as a weighted average of the neighboring pixel positions.

Conversely, if one reduces the size of an image, one removes certain possible spatial frequencies. but if one uses a naive approach, some of the original high frequency information is erroneously transfered to lower frequencies. The answer in this case is to apply a blurring filter first to remove the high frequency information before sampling. Figure 1.9 shows the effect of blurring before shrinking and of interpolating when expanding.

If we resize an image, we have to take into account the possibility that the scaling in the horizontal and vertical directions will differ, it is thus desirable to resize it in two steps, once in each direction. Consider first the problem of expanding an image. Horizontal interpolation involves the process shown in Fig 1.10.

Here we introduce a new sample point r between two existing sample points p, q. The value of r should be a weighted average of the values at the known points. If r is close to p then p should predominate and vice-versa for r. The simplest formula that achieves this is

$$r = p\frac{\delta(p,r)}{\delta(p,q)} + q(1 - \frac{\delta(p,r)}{\delta(p,q)})$$

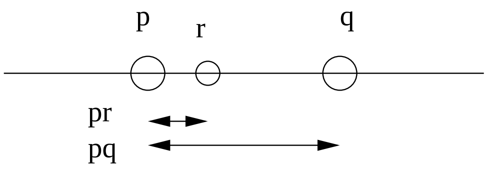


Figure 1.10: Horizontal interpolation of a new pixel position r between existing pixel positions p and q.

where $\delta(a, b)$ is the horizontal distance between points a, b.

It is clear that in the general case of horizontal resizing, the weights $\frac{\delta(p,r)}{\delta(p,q)}$ will differ for sequential pixels. As such horizontal rescaling lends itself poorly to SIMD parallelization. Vertical rescaling can be parallelized, since we can compute a complete new scan line as the weighted average of two original scan lines. It is therefore important to perform expansion in the horizontal direction first followed by rescaling in the vertical direction. This maximises the share of the work that can be run in parallel. Algorithm 16 illustrates this.

1.7 Horizontal Resize

This is done with the procedure

```
procedure resizeh ( var src ,dest :image );
```

This will change the size of an image in the horizontal direction. Dest must be same height as src. Its internal operation is shown in Algorithm 17.

1.8 Vertical Resizing

1.9 Horizontal Interpolation

This is performed by procedure

```
procedure Interpolateh ( var src ,dest :image );
```

This will interpolate an image in the horizontal direction. Src and dest must differ in size only in the horizontal direction.

This is an inherently serial procedure and as such used classical Pascal loops. It's internals are shown in Algorithm 19 .

Algorithm 16 Resize an image.

```
procedure resize ( var src ,dest :image );
```

This invokes the horizontal and vertical resize functions to do the effective work. Since vertical interpolation is run in parallel whereas horizontal interpolation must run sequentially, we want to do as much work as possible in the vertical resizing. If we are making a picture higher then it is quicker to resize horizontally and then resize vertically.
If we are reducing the height of a picture the reverse holds.

```
var
    Let $t \in$ pimage;
begin
    then
    begin
        new ( t ,src .maxplane ,src .maxrow ,dest .maxcol );
        resizeh (src, t↑);
        resizev (t↑, dest);
        dispose ( t );
    end
    else
    begin
        new ( t ,src .maxplane ,dest .maxrow ,src .maxcol );
        resizev (src, t↑);
        resizeh (t↑, dest);
        dispose ( t );
    end
end ;
```

Algorithm 17 Horizontal resize an image.

```
var
    Let $n \in$ real;
    Let $t$, $av \in$ pimage;
    Let $i \in$ integer;
begin

    $n \leftarrow \frac{1+src.maxcol}{1+dest.maxcol}$;
    if  $n < 1$
    else
        if  $n = 1$
        then  $dest \leftarrow src$
        else
            if  $n \leq 2$
            then
            begin
```

We can not simply select every nth pixel on a row, since this would allow high frequency noise to penetrate the reduced image. We have to filter out this noise first. The way we do it is by first forming a new image each of whose pixels is is the average of the corresponding two horizontally adjacent pixels in the source.

```
                new ( t ,src .maxplane ,src .maxrow ,src .maxcol );
                new ( av ,src .maxplane ,src .maxrow ,src .maxcol );
                adjustcontrast (0.5, src, t↑);

                av↑← t↑;
                av ^ [][][src .maxcol ]:=src [][][src .maxcol ];
```

av now contains an horizontally blured version of the source.

```
                dispose ( t );
                interpolateh (av↑, dest);
                dispose ( av );
            end
            else
            begin
                ;
```

Apply the shrinking recursively to get down to a shrinkage factor < 2

```
                new ( t ,src .maxplane ,src .maxrow , ( 1+src .maxcol ) div 2 + -1 );
                resizeh (src, t↑);        by 2
                resizeh (t↑, dest);       by n/2
                dispose ( t );
            end
end ;
```

Algorithm 18 Vertical Resize Routine

```
procedure resizev ( var src ,dest :image );
```

Change the size of an image in the vertical direction. Dest must be same width as src.

```
var
    Let $n \in$ real;
    Let $t$, $av \in$ pimage;
    Let $rows \in$ integer;
begin
    $n \leftarrow \frac{1+src.maxrow}{1+dest.maxrow}$;
    else
        if $n = 1$ then dest← src
        else
            if $n \leq 2$
            then
            begin
```

this filters in the vertical direction

```
                new ( t ,src .maxplane ,src .maxrow ,src .maxcol );
                new ( av ,src .maxplane ,src .maxrow ,src .maxcol );
                adjustcontrast (0.5, src, t↑);
                for rows← 0 to src.maxrow - 1 do
                    av↑← t↑;
                av ^ [][src .maxrow ]:=src [][src .maxrow ];
```

av now contains a vertically blurred version of the source.

```
                dispose ( t );
                interpolatev (av↑, dest);
                dispose ( av );
            end
            else
            begin
```

Apply the shrinking recursively to get down to a shrinkage factor < 2

```
                rows← $\frac{src.maxrow}{2}$;
                new ( t ,src .maxplane ,rows , ( src .maxcol ) );
by 2            resizev (src, t↑);
by n/2          resizev (t↑, dest);
                dispose ( t );
            end
end ;
```

Algorithm 19 Horizontal Interpolation routine.

```
var
    Let $ratio$, $p$, $q \in$ real;
    Let $i$, $j$, $k$, $l \in$ integer;
begin
    $ratio \leftarrow \frac{1+src.maxcol}{1+dest.maxcol}$;
    for $j \leftarrow 0$ to $dest.maxrow$ do
    begin
        for $k \leftarrow 0$ to $dest.maxcol$ do
        begin
            $p \leftarrow k \times ratio$;
```

P holds the horizontal position in the source that the data must come from.

```
            $l \leftarrow trunc\ (p)$;
```

l holds the sample point below p and l+1 holds the position above it

```
            $q \leftarrow p - l$;
```

q holds the distance away from l, that p was.

```
            if $l + 1 > src.maxcol$ then $dest$ [][$j$ ,$k$ ]:=$src$ [][$j$ ,$l$ ]
            else
                $dest$ [][$j$ ,$k$ ]:=$src$ [][$j$ ,$l$ ]*( 1-$q$ )+$src$ [][$j$ ,1+$l$ ]*$q$ ;
```

Interpolate in the horizontal direction using linear weighting.

```
        end ;
    end ;
end ;
```

1.10 Interpolate Vertically

This is performed by the procedure

procedure *Interpolatev* (**var** *src* ,*dest* :*image*);

Interpolates in the vertical direction. Src, and dest must differ in size only in the vertical direction. This is parallel code, and uses array expressions. The internals of the procedure are given in Algorithm 20.

Algorithm 20 Vertical interpolation of image lines.

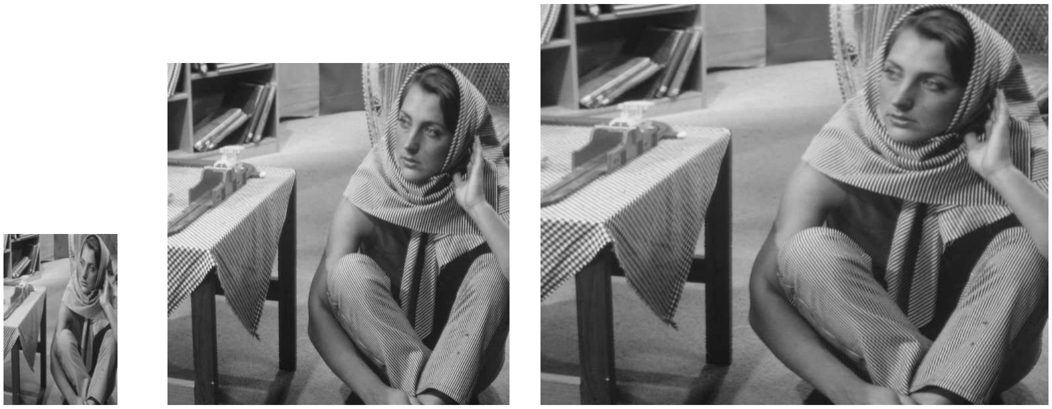
```
var
    Let $l \in$ ^line;
    Let $pp \in$ pixel;
    Let $i, j, k \in$ integer;
    Let $ratio, p, q \in$ real;
begin
    new ( $l$ ,$dest.maxcol$ );
    $ratio \leftarrow \frac{1+src.maxrow}{dest.maxrow+1}$;
    for $j \leftarrow 0$ to $dest.maxrow$ do
    begin
        $p \leftarrow j \times ratio$;
        $k \leftarrow trunc(p)$;
        $q \leftarrow p - k$;
        $pp \leftarrow q$;
```

convert weight to pixel

```
        $l\uparrow \leftarrow pp$;
```

replicate to a line to allow efficient vectorisation

```
        for $i \leftarrow 0$ to $src.maxplane$ do  if $k + 1 > src.maxrow$ then $dest_{i,j} \leftarrow src_{i,k} \times l\uparrow$
        else
            $dest_{i,j} \leftarrow src_{i,1+k} \times l\uparrow$;
        $pp \leftarrow 1 - q$;
        $l\uparrow \leftarrow pp$;
        for $i \leftarrow 0$ to $src.maxplane$ do $dest_{i,j} \leftarrow dest_{i,j} + src_{i,k} \times l\uparrow$;
    end ;
    dispose ( $l$ );
end ;
```



At 253 × 169 Original 512 × 512 At 598 × 756

Figure 1.11: Effect of applying resize to barbara.bmp.