Appendix B: WebLib

This appendix describes the image processing routines available for use with the Adapt programming language. Each description includes the following:

- the purpose of the routine
- the call syntax
- the input and output parameters
- the routine code

The routines are arranged in alphabetical order.

Contents

PURPOSE: Add a constant to an image.

CALL SYNTAX: addc1b(*image1, constant, image2*)

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: the output image

PROCEDURE:

```
procedure addc1b(   image1      : in image byte,
                    constant    : in integer,
                    image2      : out image byte)
is
next begin
    image2 := image1 + constant;
end next;
end addc1b;
```

PURPOSE: Add a constant to a complex image.

CALL SYNTAX: addc1c(*image1r, image1i, constantr, constanti, image2r, image2i*)

INPUT PARAMETERS:

image1r: The real part of the input image.

image1i: The imaginary part of the input image.

constantr: The real part of a constant.

constanti: The imaginary part of a constant.

OUTPUT PARAMETERS:

image2r: The real part of the output image.

image2i: The imaginary part of the output image.

PROCEDURE:

```
procedure addc1c(  image1r    : in image float,
                   image1i    : in image float,
                   constantr  : in float,
                   constanti  : in float,
                   image2r    : out image float,
                   image2i    : out image float)

is

next begin

    image2r := image1r + constantr;
    image2i := image1i + constanti;

end next;

end addc1c;
```

PURPOSE: Add a constant to a real image.

CALL SYNTAX: addc1r(*image1, constant, image2)*

INPUT PARAMETERS:

image1: The input image.

constant: A real constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure addc1r(   image1      : in image float,
                    constant    : in float,
                    image2      : out image float)
is
next begin
    image2 := image1 + constant;
end next;
end addc1r;
```

PURPOSE: Add a constant to a signed_byte image.

CALL SYNTAX: addc1s(*image1, constant, image2)*

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: the output image

PROCEDURE:

```
procedure addc1s(   image1      : in image signed_byte,
                    constant    : in integer,
                    image2      : out image signed_byte)
is
next begin
    image2 := image1 + constant;
end next;
end addc1s;
```

PURPOSE: Add two images.

CALL SYNTAX: addp1b(*image1, image2, image3)*

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure addp1b(   image1    : in image byte,
                    image2    : in image byte,
                    image3    : out image byte)
is
next begin
    image3 := image1 + image2;
end next;
end addp1b;
```

PURPOSE: Add two complex images.

CALL SYNTAX: addp1c(*image1r, image1i, image2r, image2i, image3r, image3i)*

INPUT PARAMETERS:

image1r: The real part of the first input image.

image1i: The imaginary part of the first input image.

image2r: The real part of the second input image.

image2i: The imaginary part of the second input image.

OUTPUT PARAMETERS:

image3r: The real part of the output image.

image3i: The imaginary part of the output image.

PROCEDURE:

```
procedure addp1c(   image1r   : in image float,
                    image1i   : in image float,
                    image2r   : in image float,
                    image2i   : in image float,
                    image3r   : out image float,
                    image3i   : out image float)
is
next begin
    image3r := image1r + image2r;
    image3i := image1i + image2i;
end next;
end addp1c;
```

PURPOSE: Add two real images.

CALL SYNTAX: addp1r(*image1, image2, image3)*

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure addp1r(   image1   : in image float,
                    image2   : in image float,
                    image3   : out image float)
is
next begin
    image3 := image1 + image2;
end next;
end addp1r;
```

PURPOSE: Add two signed_byte images.

CALL SYNTAX: addp1s(*image1, image2, image3)*

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure addp1s(   image1   : in image signed_byte,
                    image2   : in image signed_byte,
                    image3   : out image signed_byte)
is
next begin
    image3 := image1 + image2;
end next;
end addp1s;
```

PURPOSE: AND a constant with an image.

CALL SYNTAX: andc1b(*image1, constant, image2)*

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure andc1b(image1      : in image byte,
                 constant    : in integer,
                 image2      : out image byte)
is
next begin
    image2 := image1 land constant;
end next;
end andc1b;
```

PURPOSE: AND two images.

CALL SYNTAX: andp1b(*image1, image2, image3)*

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure andp1b(   image1   : in image byte,
                    image2   : in image byte,
                    image3   : out image byte)
is
next begin
    image3 := image1 land image2;
end next;
end andp1b;
```

PURPOSE: Perform one iteration of local selective averaging for edge preserving smoothing.

CALL SYNTAX: asmt(*imagein*, *imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image (output values are rounded to the nearest integer).

PROCEDURE:

```
procedure asmt(     imagein     : in image array (–1..1, –1..1) of byte border 0,
                    imageout    : out image byte)
is
     gu, sum, sw : integer;
     Pu : float;
     d00, d01, d02, d10, d12, d20, d21, d22 : integer;
     i00, i01, i02, i10, i12, i20, i21, i22 : integer;
next begin
     i00 := imagein(–1,–1); i01 := imagein(–1,0); i02 := imagein(–1,1);
     i10 := imagein(0,–1); gu  := imagein(0,0); i12 := imagein(0,1);
     i20 := imagein(1,–1); i21 := imagein(1,0); i22 := imagein(1,1);

     d00 := abs(i00–gu); d01 := abs(i01–gu); d02 := abs(i02–gu);
     d10 := abs(i10–gu); d12 := abs(i12–gu);
     d20 := abs(i20–gu); d21 := abs(i21–gu); d22 := abs(i22–gu);
     Pu := 0.125 * (d00 + d01 + d02 + d10 + d12 + d20 + d21 + d22);
     sum := gu; sw := 1;

     if  d00 <= Pu  then
          sum := sum + i00;
          sw:= sw + 1;
     end if;

     if  d01 <= Pu  then
          sum := sum + i01;
          sw:= sw + 1;
     end if;

     if  d02 <= Pu  then
          sum := sum + i02;
          sw:= sw + 1;
     end if;
```

```
    if  d10 <= Pu  then
        sum := sum + i10;
        sw:= sw + 1;
    end if;
    if  d12 <= Pu  then
        sum := sum + i12;
        sw:= sw + 1;
    end if;
    if  d20 <= Pu  then
        sum := sum + i20;
        sw:= sw + 1;
    end if;
    if  d21 <= Pu  then
        sum := sum + i21;
        sw:= sw + 1;
    end if;
    if  d22 <= Pu  then
        sum := sum + i22;
        sw:= sw + 1;
    end if;
    imageout := sum/sw;
end next;
end asmt;
```

PURPOSE: Detect the borders in a binary image in terms of 4–connectedness.

CALL SYNTAX: bdr41(*imgin*, *mode*, *imgout*)

INPUT PARAMETERS:

imgin: The input image.

mode: If 1, detect inner borders; otherwise detect outer borders.

OUTPUT PARAMETERS:

imgout: The output image.

PROCEDURE:

```
procedure bdr41(   imgin    : in image array (–1..1, –1..1) of byte border 0,
                   mode     : in integer,
                   imgout   : out image byte)
is
    sum, prod : integer;
next begin
    imgout := 0;
    sum :=  imgin(–1, 0) + imgin( 0,–1) + imgin( 0, 1) + imgin( 1, 0);
    prod := imgin(–1, 0) * imgin( 0,–1) * imgin( 0, 1) * imgin( 1, 0);
    if mode = 1 then
        if imgin(0,0) = 0 then
            if sum > 0 then
                imgout := 1;
            end if;
        end if;
    else
        if imgin(0,0) = 1 then
            if prod = 0 then
                imgout := 1;
            end if;
        end if;
    end if;
end next;
end bdr41;
```

PURPOSE: Detect the borders in a binary image in terms of 8–connectedness.

CALL SYNTAX: bdr81(*imgin*, *mode*, *imgout*)

INPUT PARAMETERS:

imgin: The input image.

mode: If *mode* = 1, detect inner borders; otherwise detect outer borders.

OUTPUT PARAMETERS:

imgout: The output image.

PROCEDURE:

```
procedure bdr81(   imgin    : in image array (–1..1, –1..1) of byte border 0,
                   mode     : in integer,
                   imgout   : out image byte)
is
    sum, prod : integer;
next begin
    imgout := 0;
    sum :=  imgin(–1,–1) + imgin(–1, 0) + imgin(–1, 1) + imgin( 0,–1)
              + imgin( 0, 1) + imgin( 1,–1) + imgin( 1, 0) + imgin( 1, 1);
    prod := imgin(–1,–1) * imgin(–1, 0) * imgin(–1, 1) * imgin( 0,–1)
              * imgin( 0, 1) * imgin( 1,–1) * imgin( 1, 0) * imgin( 1, 1);
    if mode = 1 then
        if imgin(0,0) = 0 then
            if sum > 0 then
                imgout := 1;
            end if;
        end if;
    else
        if imgin(0,0) = 1 then
            if prod = 0 then
                imgout := 1;
            end if;
        end if;
    end if;
end next;
end bdr81;
```

PURPOSE: Detect the borders of regions in a labeled image.

CALL SYNTAX: blfp1(*imgin*, *rgn*, *lbl, connect, imgout*)

INPUT PARAMETERS:

imgin: A labeled input image.

rgn: If *rgn* = 0, all regions are processed, otherwise, only the region specified by *rgn* is processed

lbl: A label assigned to the border points. If *lbl* = 0, the label of the region is output.

connect: The neighbor-connectedness (4 or 8).

OUTPUT PARAMETERS:

imgout: The output image.

PROCEDURE:

```
procedure bflp1(   imgin      : in image array (–1..1, –1..1) of byte border 0,
                   rgn        : in integer,
                   lbl        : in integer,
                   connect    : in integer,
                   imgout     : out image byte)
is
    i00, temp1, temp2, temp3, temp : integer;
next begin
    i00 := imgin( 0, 0);
    imgout := 0;
    temp := 0;

    if rgn=0 then
        temp := 1;
    else
        if i00=rgn then
            temp := 1;
        end if;
    end if;

    if  temp /= 0 then
        temp1 := 4*i00;
        temp2 := temp1 – imgin(–1, 0) – imgin( 0,–1) – imgin( 0, 1) – imgin( 1, 0) ;
        if connect=8  then
            temp2 := temp2 + temp1 – imgin(–1,–1) – imgin(–1, 1) – imgin( 1,–1)
                           – imgin( 1, 1);
        end if;

        temp3 := temp2 * i00;
```

```
            if temp3 = 0 then
                  imgout := 0;
            else
                  if  lbl=0  then
                        imgout := i00;
                  else
                        imgout := lbl;
                  end if;
            end if;
      end if;

end next;

end bflp1;
```

PURPOSE: Convert an image from byte to real.

CALL SYNTAX: byrl(*imagein, imageout)*

INPUT PARAMETERS:

imagein: The input byte image.

OUTPUT PARAMETERS:

imageout: The output real image.

PROCEDURE:

```
procedure byrl(   imagein    : in image byte,
                  imageout   : out image float)
is
next begin
     imageout := imagein;
end next;
end byrl;
```

PURPOSE: Set the gray values within a range to zero.

CALL SYNTAX: clip(*imagein, gry1, gry2, imageout)*

INPUT PARAMETERS:

imagein: The input image.

gry1: The lower bound of the range.

gry2: The upper bound of the range.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure clip(   imagein     : in image byte,
                  gry1        : in integer,
                  gry2        : in integer,
                  imageout    : out image byte)

is

next begin

    imageout := imagein;
    if  imagein >= gry1  then
        if imagein <= gry2 then
            imageout:= 0;
        else
            imageout := imagein;
        end if;
    end if;

end next;

end clip;
```

PURPOSE: Average three color bands to produce a black and white image.

CALL SYNTAX: colortobw(*red, green, blue, gray)*

INPUT PARAMETERS:

red: A color image.

green: A color image.

blue: A color image.

OUTPUT PARAMETERS:

gray: A black and white image.

PROCEDURE:

```
procedure colortobw(   red     : in image byte,
                       green   : in image byte,
                       blue    : in image byte,
                       gray    : out image byte)
is
next begin
    gray := (red + green + blue)/3;
end next;
end colortobw;
```

PURPOSE: Compute the connectivity number of a binary image.

CALL SYNTAX: conc(*imagein*, *connectivity*, *imageout*)

INPUT PARAMETERS:

imagein: The input image.

connectivity: The neighbor connectedness (4 or 8).

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure conc(   imagein       : in image array (–1..1, –1..1) of byte border 0,
                  connectivity  : in integer,
                  imageout      : out image byte)
is
next begin
    if connectivity = 4 then
        imageout := imagein( 0, 1) – imagein( 0, 1)*imagein(–1, 1)*imagein(–1, 0)
                  + imagein(–1, 0) – imagein(–1, 0)*imagein(–1,–1)*imagein( 0,–1)
                  + imagein( 0,–1) – imagein( 0,–1)*imagein( 1,–1)*imagein( 1, 0)
                  + imagein( 1, 0) – imagein( 1, 0) * imagein( 1, 1) * imagein( 0, 1);
    else
        imageout :=

    (1–imagein( 0, 1))–(1–imagein( 0, 1))*(1–imagein(–1, 1))*(1–imagein(–1, 0))
  + (1–imagein(–1, 0))–(1–imagein(–1, 0))*(1–imagein(–1,–1))*(1–imagein( 0,–1))
  + (1–imagein( 0,–1))–(1–imagein( 0,–1))*(1–imagein( 1,–1))*(1–imagein( 1, 0))
  + (1–imagein( 1, 0))–(1–imagein( 1, 0))*(1–imagein( 1, 1))*(1–imagein( 0, 1));

    end if;
end next;
end conc;
```

PURPOSE: Compute the crossing number of a binary image.

CALL SYNTAX: cros(*imagein*, *connectivity*, *imageout*)

INPUT PARAMETERS:

imagein: The input image.

connectivity: The neighbor connectedness (4 or 8).

OUTPUT PARAMETERS:

imagout: The output image.

PROCEDURE:

```
procedure cros(   imagein        : in image array (–1..1, –1..1) of byte border 0,
                  connectivity   : in integer,
                  imageout       : out image byte)
is
next begin
     if connectivity = 4 then
          imageout := abs(imagein(–1, 0) – imagein(0, 1))
                    + abs(imagein( 0, 1) – imagein(1, 0))
                    + abs(imagein( 1, 0) – imagein(0,–1))
                    + abs(imagein( 0,–1) – imagein(–1,0));
     else
          imageout := abs(imagein(–1, 0) – imagein(–1,1))
                    + abs(imagein(–1, 1) – imagein( 0,1))
                    + abs(imagein( 0, 1) – imagein( 1,1))
                    + abs(imagein( 1, 1) – imagein( 1,0))
                    + abs(imagein( 1, 0) – imagein(0,–1))
                    + abs(imagein( 0,–1) – imagein(–1,–1))
                    + abs(imagein(–1,–1) – imagein(–1, 0));
     end if;
end next;
end cros;
```

PURPOSE: Divide an image by a constant.

CALL SYNTAX: divc1b(*image1, constant, image2)*

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure divc1b(   image1     : in image byte,
                    constant   : in integer,
                    image2     : out image byte)
is
next begin
    image2 := image1 / constant;
end next;
end divc1b;
```

PURPOSE: Divide a real image by a constant.

CALL SYNTAX: divc1r(*image1, constant, image2)*

INPUT PARAMETERS:

image1: The input image.

constant: A real constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure divc1r(   image1     : in image float,
                    constant   : const real,
                    image2     : out image float)
is
next begin
    image2 := image1 / constant;
end next;
end divc1r;
```

PURPOSE: Divide a signed_byte image by a constant.

CALL SYNTAX: divc1s(*image1, constant, image2)*

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure divc1s(   image1     : in image signed_byte,
                    constant   : in integer,
                    image2     : out image signed_byte)
is
next begin
    image2 := image1 / constant;
end next;
end divc1s;
```

PURPOSE: Divide the first image by the second image.

CALL SYNTAX: divp1b(*image1, image2, image3)*

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure divp1b(   image1  : in image byte,
                    image2  : in image byte,
                    image3  : out image byte)
is
next begin
    image3 := image1 / image2;
end next;
end divp1b;
```

PURPOSE: Divide the first real image by the second real image.

CALL SYNTAX: divp1r(*image1, image2, image3)*

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure divp1r(   image1    : in image float,
                    image2    : in image float,
                    image3    : out image float)
is
next begin
    image3 := image1 / image2;
end next;
end divp1r;
```

PURPOSE: Divide the first signed_byte image by the second signed_byte image.

CALL SYNTAX: divp1s(*image1, image2, image3)*

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure divp1s(   image1   : in image signed_byte,
                    image2   : in image signed_byte,
                    image3   : out image signed_byte)
is
next begin
    image3 := image1 / image2;
end next;
end divp1s;
```

PURPOSE: Perform edge detection using orthogonal 3x3 template.

CALL SYNTAX: egfc(*imagein*, *imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image

PROCEDURE:

```
procedure egfc(   imagein    : in image array (–1..1, –1..1) of byte,
                  imageout   : out image float)
is
    i00, i01, i02, i10, i11, i12, i20, i21, i22 : integer;
    sum1, sum2, isq : float;
next begin
    i00 := imagein(–1,–1);
    i01 := imagein(–1,0);
    i02 := imagein(–1,1);
    i10 := imagein(0, –1);
    i11 := imagein(0, 0);
    i12 := imagein(0, 1);
    i20 := imagein(1, –1);
    i21 := imagein(1, 0);
    i22 := imagein(1, 1);
    isq :=  i00*i00 + i01*i01 + i02*i02
            + i10*i10 + i11*i11 + i12*i12
            + i20*i20 + i21*i21 + i22*i22;
    sum1 := i00 + 1.41421356*i01 + i02 – i20 – 1.41421356*i21 – i22;
    sum2 :=  i00 – i02 + 1.41421356*i10 – 1.41421356*i12 + i20 – i22;
    imageout := (sum1*sum1 + sum2*sum2)/isq;
end next;
end egfc;
```

PURPOSE: Perform edge detection using the 3x3 Kirsch operator. EGKS1 outputs the edge magnitude only.

CALL SYNTAX: egks1(*imagein*, *imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image

PROCEDURE:

```
procedure egks1(    imagein     : in image array (–1..1, –1..1) of byte border 0,
                    imageout    : out image byte)
is
    mask        : array (0..7) of integer;
    maskmax     : integer;
    i           : integer;
next begin
    mask(0) :=  5*imagein(–1,–1) + 5*imagein(–1, 0) + 5*imagein(–1, 1)
               – 3*imagein( 0,–1) – 3*imagein( 0, 1) – 3*imagein( 1,–1)
               – 3*imagein( 1, 0) – 3*imagein( 1, 1);
    mask(1) :=  5*imagein(–1,–1) + 5*imagein(–1, 0) – 3*imagein(–1, 1)
               + 5*imagein( 0,–1) – 3*imagein( 0, 1) – 3*imagein( 1,–1)
               – 3*imagein( 1, 0) – 3*imagein( 1, 1);
    mask(2) :=  5*imagein(–1,–1) – 3*imagein(–1, 0) – 3*imagein(–1, 1)
               + 5*imagein( 0,–1) – 3*imagein( 0, 1) + 5*imagein( 1,–1)
               – 3*imagein( 1, 0) – 3*imagein( 1, 1);
    mask(3) := –3*imagein(–1,–1) – 3*imagein(–1, 0) – 3*imagein(–1, 1)
               + 5*imagein( 0,–1) – 3*imagein( 0, 1) + 5*imagein( 1,–1)
               + 5*imagein( 1, 0) – 3*imagein( 1, 1);
    mask(4) := –3*imagein(–1,–1) – 3*imagein(–1, 0) – 3*imagein(–1, 1)
               – 3*imagein( 0,–1) – 3*imagein( 0, 1) + 5*imagein( 1,–1)
               + 5*imagein( 1, 0) + 5*imagein( 1, 1);
    mask(5) := –3*imagein(–1,–1) – 3*imagein(–1, 0) – 3*imagein(–1, 1)
               – 3*imagein( 0,–1) + 5*imagein( 0, 1) – 3*imagein( 1,–1)
               + 5*imagein( 1, 0) + 5*imagein( 1, 1);
    mask(6) := –3*imagein(–1,–1) – 3*imagein(–1, 0) + 5*imagein(–1, 1)
               – 3*imagein( 0,–1) + 5*imagein( 0, 1) – 3*imagein( 1,–1)
               – 3*imagein( 1, 0) + 5*imagein( 1, 1);
    mask(7) := –3*imagein(–1,–1) + 5*imagein(–1, 0) + 5*imagein(–1, 1)
               – 3*imagein( 0,–1) + 5*imagein( 0, 1) – 3*imagein( 1,–1)
               – 3*imagein( 1, 0) – 3*imagein( 1, 1);
```

```
        maskmax := mask(0);
        for i in 1..7 loop
            if  mask(i) > maskmax  then
                maskmax := mask(i);
            end if;
        end loop;

        imageout := maskmax;

    end next;

    end egks1;
```

PURPOSE: Perform edge detection using the 3x3 Kirsch operator. EGKS2 outputs the edge magnitude and direction of gradient.

CALL SYNTAX: egks2(*imagein*, *magout, dirout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

magout: The output image magnitude.

dirout: The output image direction.

PROCEDURE:

```
procedure egks2(   imagein   : in image array (–1..1, –1..1) of byte border 0,
                   magout    : out image byte,
                   dirout    : out image byte)
is
    mask       : array (0..7) of integer;
    maskmax    : integer;
    i, dir     : integer;
next begin
    mask(0) :=  5*imagein(–1,–1) + 5*imagein(–1, 0) + 5*imagein(–1, 1)
              – 3*imagein( 0,–1) – 3*imagein( 0, 1) – 3*imagein( 1,–1)
              – 3*imagein( 1, 0) – 3*imagein( 1, 1);
    mask(1) :=  5*imagein(–1,–1) + 5*imagein(–1, 0) – 3*imagein(–1, 1)
              + 5*imagein( 0,–1) – 3*imagein( 0, 1) – 3*imagein( 1,–1)
              – 3*imagein( 1, 0) – 3*imagein( 1, 1);
    mask(2) :=  5*imagein(–1,–1) – 3*imagein(–1, 0) – 3*imagein(–1, 1)
              + 5*imagein( 0,–1) – 3*imagein( 0, 1) + 5*imagein( 1,–1)
              – 3*imagein( 1, 0) – 3*imagein( 1, 1);
    mask(3) := –3*imagein(–1,–1) – 3*imagein(–1, 0) – 3*imagein(–1, 1)
              + 5*imagein( 0,–1) – 3*imagein( 0, 1) + 5*imagein( 1,–1)
              + 5*imagein( 1, 0) – 3*imagein( 1, 1);
    mask(4) := –3*imagein(–1,–1) – 3*imagein(–1, 0) – 3*imagein(–1, 1)
              – 3*imagein( 0,–1) – 3*imagein( 0, 1) + 5*imagein( 1,–1)
              + 5*imagein( 1, 0) + 5*imagein( 1, 1);
    mask(5) := –3*imagein(–1,–1) – 3*imagein(–1, 0) – 3*imagein(–1, 1)
              – 3*imagein( 0,–1) + 5*imagein( 0, 1) – 3*imagein( 1,–1)
              + 5*imagein( 1, 0) + 5*imagein( 1, 1);
    mask(6) := –3*imagein(–1,–1) – 3*imagein(–1, 0) + 5*imagein(–1, 1)
              – 3*imagein( 0,–1) + 5*imagein( 0, 1) – 3*imagein( 1,–1)
              – 3*imagein( 1, 0) + 5*imagein( 1, 1);
```

```
      mask(7) := –3*imagein(–1,–1) + 5*imagein(–1, 0) + 5*imagein(–1, 1)
                 – 3*imagein( 0,–1) + 5*imagein( 0, 1) – 3*imagein( 1,–1)
                 – 3*imagein( 1, 0) – 3*imagein( 1, 1);

      maskmax := mask(0); dir := 0;
       for i in 1..7 loop
         if mask(i) > maskmax then
           maskmax := mask(i);
           dir := i;
         end if;
      end loop;

      magout := maskmax;
      dirout := dir;

   end next;

   end egks2;
```

PURPOSE: Perform edge detection using a 3x3 Laplacian operator.

CALL SYNTAX: eglp(*imagein*, *type, imageout*)

INPUT PARAMETERS:

imagein: The input image.

type: A constant to select the Laplacian operator.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure eglp(   imagein    : in image array (–1..1, –1..1) of byte border 0,
                  type       : in integer,
                  imageout   : out image signed_byte)

is

next begin
    if type = 1 then
        imageout :=  imagein(0, 0)*4 – imagein(–1, 0) – imagein(0, –1)
                   – imagein(0, 1) – imagein(1, 0);
    else
        if type = 2 then
            imageout := imagein(0, 0)*8 – imagein(–1, –1) – imagein(–1, 0)
                        – imagein(–1, 1) – imagein(0, –1) – imagein(0, 1)
                        – imagein(1, –1) – imagein(1, 0) – imagein(1, 1);
        else
            if type = 3 then
            imageout := imagein(–1, –1) + imagein(–1, 0)*–2 + imagein(–1, 1)
                        + imagein(0, –1)*–2 + imagein(0, 0)*4 + imagein(0, 1)*–2
                        + imagein(1, –1) + imagein(1, 0)*–2 + imagein(1, 1);
            end if;
        end if;
    end if;

end next;

end eglp;
```

PURPOSE: Filter edges for smoothing.

CALL SYNTAX: egpr(*img*, *imageout*)

INPUT PARAMETERS:

img: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure egpr(   img         : in image array (–2..2, –2..2) of byte border 0,
                  imageout    : out image byte)
is
    minavg, minvar, im      : float;
    i, j                    : integer;
    l,k                     : array(–2..2,–2..2) of float;
    a,v                     : array(0..8) of float;
next begin
    for i in –2..2 loop
        for j in –2..2 loop
            l(i,j) := img(i,j) * img(i,j);
        end loop;
    end loop;

    a(0):= 0.1111111*(img(–1,–1) + img(0,–1) + img(1,–1) + img(–1,0) + img(0,0)
        + img(1,0) + img(–1,1) + img(0,1) + img(1,1));

    v(0):= 0.7777778*(l(–1,–1) + l(0,–1) + l(1,–1) + l(–1,0) + l(0,0) + l(1,0)
        + l(–1,1) + l(–1,1) + l(1,1)) – 7.0*a(0)*a(0);

    minavg := a(0);
    minvar := v(0);

    a(1):= 0.1428571*(img(–1,–2) + img(0,–2) + img(1,–2) + img(–1,–1)
        + img(0,–1) + img(1,–1) + img(0,0));

    v(1):= l(–1,–2) + l(0,–2) + l(1,–2) + l(–1,–1) + l(0,–1) + l(1,–1)
        + l(0,0) –7.0*a(1)*a(1);

    if  v(1) < minvar  then
        minvar := v(1);
        minavg := a(1);
    end if;

    a(2):= 0.1428571*(img(1,–2) + img(2,–2) + img(0,–1) + img(1,–1)
        + img(2,–1) + img(0,0) + img(1,0));
```

```
v(2):= l(1,–2) + l(2,–2) + l(0,–1)+l(1,–1) + l(2,–1) + l(0,0) + l(1,0) –7.0*a(2)*a(2);
if  v(2) < minvar  then
      minvar := v(2);
      minavg := a(2);
end if;
a(3):= 0.1428571*(img(1,–1) + img(2,–1) + img(0,0) + img(1,0) + img(2,0)
      + img(1,1) + img(2,1));
v(3):= l(1,–1) + l(2,–1) + l(0,0)+l(1,0) + l(2,0)+l(1,1) + l(2,1) –7.0*a(3)*a(3);
if  v(3) < minvar  then
      minvar := v(3);
      minavg := a(3);
end if;
a(4):= 0.1428571*(img(0,0) + img(1,0) + img(0,1) + img(1,1) + img(2,1)
      + img(1,2) + img(2,2));
v(4):= l(0,0) + l(1,0) + l(0,1) + l(1,1) + l(2,1) + l(1,2) + l(2,2) –7.0*a(4)*a(4);
if  v(4) < minvar  then
      minvar := v(4);
      minavg := a(4);
end if;
a(5):= 0.1428571*(img(0,0) + img(–1,1) + img(0,1) + img(1,1) + img(–1,2)
      + img(0,2) + img(1,2));
v(5):= l(0,0) + l(–1,1) + l(0,1) + l(1,1) + l(–1,2) + l(0,2) + l(1,2) –7.0*a(5)*a(5);
if  v(5) < minvar  then
      minvar := v(5);
      minavg := a(5);
end if;
a(6):= 0.1428571*(img(–1,0) + img(0,0) + img(–2,1) + img(–1,1)
      + img(0,1) + img(–2,2) + img(–1,2));
v(6):= l(–1,0) + l(0,0) + l(–2,1) + l(–1,1) + l(0,1) + l(–2,2) + l(–1,2)
      –7.0*a(6)*a(6);
if  v(6) < minvar  then
      minvar := v(6);
      minavg := a(6);
end if;
a(7):= 0.1428571*(img(–2,–1) + img(–1,–1) + img(–2,0) + img(–1,0)
      + img(0,0) + img(–2,1) + img(–1,1));
v(7):= l(–2,–1) + l(–1,–1) + l(–2,0) + l(–1,0) + l(0,0) + l(–2,1) + l(–1,1)
      –7.0*a(7)*a(7);
if  v(7) < minvar  then
      minvar := v(7);
      minavg := a(7);
end if;
a(8):= 0.1428571*(img(–2,–2) + img(–1,–2) + img(–2,–1) + img(–1,–1)
      + img(0,–1) + img(–1,0) + img(0,0));
v(8):= l(–2,–2) + l(–1,–2) + l(–2,–1) + l(–1,–1) + l(0,–1) + l(–1,0)+l(0,0)
      –7.0*a(8)*a(8);
```

```
    if  v(8) < minvar  then
        minvar := v(8);
        minavg := a(8);
    end if;

    imageout := minavg;

end next;

end egpr;
```

PURPOSE: Perform edge detection using the Prewitt operator (differential type). EGPW1 outputs the edge magnitude only.

CALL SYNTAX: egpw1(*imagein*, *type, imageout*)

INPUT PARAMETERS:

imagein: The input image.

type: A constant to select the magnitude computation equation.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure egpw1(   imagein    : in image array (–1..1, –1..1) of byte border 0,
                   type       : in integer,
                   imageout   : out image byte)
is
    sx, sy          : integer;
    mag1, mag2      : float;
next begin
    sy :=  imagein(–1,–1) + imagein(–1, 0) + imagein(–1, 1)
        – imagein( 1,–1) – imagein( 1, 0)–imagein( 1, 1);

    sx :=  imagein(–1,–1) – imagein(–1, 1) + imagein( 0,–1)
        – imagein( 0, 1) + imagein( 1,–1) – imagein( 1, 1);

    mag1 := sqrt(float(sx*sx + sy*sy));
    mag2 := abs(sx) + abs(sy);

    if type = 1 then
        imageout := mag1;
    else
        imageout := mag2;
    end if;

end next;

end egpw1;
```

PURPOSE: Perform edge detection using the Prewitt operator (differential type). EGPW2 outputs the edge magnitude and direction of gradient.

CALL SYNTAX: egpw2(*imagein*, *type, magout*, *dirout*)

INPUT PARAMETERS:

imagein: The input image.

type: A constant to select the magnitude computation equation.

OUTPUT PARAMETERS:

magout: The output image magnitude.

dirout: The output image direction.

PROCEDURE:

```
procedure egpw2(  imagein  : in image array (–1..1, –1..1) of byte border 0,
                  type     : in integer,
                  magout   : out image byte,
                  dirout   : out image float)

is

    sx, sy          : integer;
    temp1, temp2    : integer;

next begin

    temp1 := imagein(–1,–1) – imagein( 1, 1);
    temp2 := imagein(–1, 1) – imagein( 1,–1);

    sx :=  temp1 + temp2 + imagein(–1, 0) – imagein( 1, 0);
    sy :=  temp1 – temp2 + imagein( 0,–1) – imagein( 0, 1);

    if type = 1 then
        magout := sqrt(float(sx*sx + sy*sy));
    else
        magout := abs(sx) + abs(sy);
    end if;

    if (sx = sy) then
        dirout := 0.0;
    else
        dirout := atan2(float(sy), float(sx));
    end if;

end next;

end egpw2;
```

PURPOSE: Perform edge detection using the Prewitt operator (template type). EGPW3 outputs the edge magnitude only.

CALL SYNTAX: egpw3(*imagein*, *imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure egpw3(   imagein    : in image array (–1..1, –1..1) of byte border 0,
                   imageout   : out image byte)
is
    mask       : array (0..7) of integer;
    maskmax    : integer;
    i          : integer;
next begin
    mask(0) := imagein(–1,–1) + imagein(–1, 0) + imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0)+  imagein( 0, 1)
               – imagein( 1,–1) – imagein( 1, 0) – imagein( 1, 1);
    mask(1) := imagein(–1,–1) + imagein(–1, 0) + imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0) – imagein( 0, 1)
               + imagein( 1,–1) – imagein( 1, 0) – imagein( 1, 1);
    mask(2) := imagein(–1,–1) + imagein(–1, 0) – imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0) – imagein( 0, 1)
               + imagein( 1,–1) + imagein( 1, 0) – imagein( 1, 1);
    mask(3) := imagein(–1,–1) – imagein(–1, 0) – imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0) – imagein( 0, 1)
               + imagein( 1,–1) + imagein( 1, 0) + imagein( 1, 1);
    mask(4) := –imagein(–1,–1) – imagein(–1, 0) – imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0) + imagein( 0, 1)
               + imagein( 1,–1) + imagein( 1, 0) + imagein( 1, 1);
    mask(5) := –imagein(–1,–1) – imagein(–1, 0) + imagein(–1, 1)
               – imagein( 0,–1) – 2*imagein( 0, 0) + imagein( 0, 1)
               + imagein( 1,–1) + imagein( 1, 0) + imagein( 1, 1);
    mask(6) := – imagein(–1,–1) + imagein(–1, 0) + imagein(–1, 1)
               – imagein( 0,–1) – 2*imagein( 0, 0) + imagein( 0, 1)
               – imagein( 1,–1) + imagein( 1, 0) + imagein( 1, 1);
    mask(7) :=  imagein(–1,–1) + imagein(–1, 0) + imagein(–1, 1)
               – imagein( 0,–1)  –2*imagein( 0, 0) + imagein( 0, 1)
               – imagein( 1,–1) – imagein( 1, 0) + imagein( 1, 1);
```

```
    maskmax := mask(0);
    for i in 1..7 loop
        if  mask(i) > maskmax  then
            maskmax := mask(i);
        end if;
    end loop;

    imageout := maskmax;

end next;

end egpw3;
```

PURPOSE: Perform edge detection using the Prewitt operator (template type). EGPW4 outputs the edge magnitude and direction of gradient.

CALL SYNTAX: egpw4(*imagein*, *magout, dirout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

magout: The output image magnitude.

dirout: The output image direction.

PROCEDURE:

```
procedure egpw4(    imagein    : in image array (–1..1, –1..1) of byte border 0,
                    magout     : out image byte,
                    dirout     : out image byte)
is
    mask       : array (0..7) of integer;
    maskmax    : integer;
    i, dir     : integer;
next begin
    mask(0) := imagein(–1,–1) + imagein(–1, 0) + imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0)+  imagein( 0, 1)
               – imagein( 1,–1) – imagein( 1, 0) – imagein( 1, 1);
    mask(1) := imagein(–1,–1) + imagein(–1, 0) + imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0) – imagein( 0, 1)
               + imagein( 1,–1) – imagein( 1, 0) – imagein( 1, 1);
    mask(2) := imagein(–1,–1) + imagein(–1, 0) – imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0) – imagein( 0, 1)
               + imagein( 1,–1) + imagein( 1, 0) – imagein( 1, 1);
    mask(3) := imagein(–1,–1) – imagein(–1, 0) – imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0) – imagein( 0, 1)
               + imagein( 1,–1) + imagein( 1, 0) + imagein( 1, 1);
    mask(4) := –imagein(–1,–1) – imagein(–1, 0) – imagein(–1, 1)
               + imagein( 0,–1) – 2*imagein( 0, 0) + imagein( 0, 1)
               + imagein( 1,–1) + imagein( 1, 0) + imagein( 1, 1);
    mask(5) := –imagein(–1,–1) – imagein(–1, 0) + imagein(–1, 1)
               – imagein( 0,–1) – 2*imagein( 0, 0) + imagein( 0, 1)
               + imagein( 1,–1) + imagein( 1, 0) + imagein( 1, 1);
    mask(6) := – imagein(–1,–1) + imagein(–1, 0) + imagein(–1, 1)
               – imagein( 0,–1) – 2*imagein( 0, 0) + imagein( 0, 1)
               – imagein( 1,–1) + imagein( 1, 0) + imagein( 1, 1);
```

```
    mask(7) :=  imagein(–1,–1) + imagein(–1, 0) + imagein(–1, 1)
                – imagein( 0,–1)  –2*imagein( 0, 0) + imagein( 0, 1)
                – imagein( 1,–1) – imagein( 1, 0) + imagein( 1, 1);

    maskmax := mask(0);
    dir := 0;

    for i in 1..7 loop
        if  mask(i) > maskmax  then
            maskmax := mask(i);
            dir := i;
        end if;
    end loop;

    magout := maskmax;
    dirout := dir;

end next;

end egpw4;
```

PURPOSE: Perform edge detection using the 2x2 Roberts operator.

CALL SYNTAX: egrb(*imagein*, *type, imageout*)

INPUT PARAMETERS:

imagein: The input image.

type: A constant to select the magnitude computation equation.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure egrb(   imagein    : in image array (0..1, 0..1) of byte border 0,
                  type       : in integer,
                  imageout   : out image byte)
is
    s1, s2 : integer;
next begin
    s1 := imagein(0,0) – imagein(1,1);
    s2 := imagein(1,0) – imagein(0,1);
    if type = 1 then
        imageout := sqrt(float(s1*s1 + s2*s2));
    else
        imageout := abs(s1) + abs(s2);
    end if;
end next;
end egrb;
```

PURPOSE: Perform edge detection using the 3x3 Robinson operator. EGRS1 outputs the edge magnitude only.

CALL SYNTAX: egrs1(*imagein*, *imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure egrs1(    imagein     : in image array (–1..1, –1..1) of byte border 0,
                    imageout    : out image byte)
is
    mask        : array (0..7) of integer;
    maskmax     : integer;
    i           : integer;
next begin
    mask(0) := imagein(–1,–1) + 2*imagein(–1, 0) + imagein(–1, 1)
               – imagein( 1,–1) – 2*imagein( 1, 0) – imagein( 1, 1);
    mask(1) := 2*imagein(–1,–1) + imagein(–1, 0) + imagein( 0,–1)
               – imagein( 0, 1) – imagein( 1, 0) – 2*imagein( 1, 1);
    mask(2) := imagein(–1,–1) – imagein(–1, 1) + 2*imagein( 0,–1)
               – 2*imagein( 0, 1) + imagein( 1,–1) – imagein( 1, 1);
    mask(3) := –imagein(–1, 0) – 2*imagein(–1, 1) + imagein( 0,–1)
               – imagein( 0, 1) + 2*imagein( 1,–1) + imagein( 1, 0);
    mask(4) := –imagein(–1,–1) – 2*imagein(–1, 0) – imagein(–1, 1)
               + imagein( 1,–1) + 2*imagein( 1, 0) + imagein( 1, 1);
    mask(5) := –2*imagein(–1,–1) – imagein(–1, 0) – imagein( 0,–1)
               + imagein( 0, 1) + imagein( 1, 0) + 2*imagein( 1, 1);
    mask(6) := –imagein(–1,–1) + imagein(–1, 1) –2*imagein( 0,–1)
               + 2*imagein( 0, 1) – imagein( 1,–1) + imagein( 1, 1);
    mask(7) := imagein(–1, 0) + 2*imagein(–1, 1) – imagein( 0,–1)
               + imagein( 0, 1) – 2*imagein( 1,–1) – imagein( 1, 0);
    maskmax := mask(0);
    for i in 1..7 loop
        if  mask(i) > maskmax  then
            maskmax := mask(i);
        end if;
    end loop;
    imageout := maskmax;
end next;
end egrs1;
```

PURPOSE: Perform edge detection using the 3x3 Robinson operator. EGRS2 outputs the edge magnitude and direction of gradient.

CALL SYNTAX: egrs2(*imagein*, *magout, dirout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

magout: The output image magnitude.

dirout: The output image direction.

PROCEDURE:

```
procedure egrs2(  imagein   : in image array (–1..1, –1..1) of byte border 0,
                  magout    : out image byte,
                  dirout    : out image byte)
is
    mask        : array (0..7) of integer;
    maskmax     : integer;
    i, dir      : integer;
next begin
    mask(0) := imagein(–1,–1) + 2*imagein(–1, 0) + imagein(–1, 1)
               – imagein( 1,–1) – 2*imagein( 1, 0) – imagein( 1, 1);
    mask(1) := 2*imagein(–1,–1) + imagein(–1, 0) + imagein( 0,–1)
               – imagein( 0, 1) – imagein( 1, 0) – 2*imagein( 1, 1);
    mask(2) := imagein(–1,–1) – imagein(–1, 1) + 2*imagein( 0,–1)
               – 2*imagein( 0, 1) + imagein( 1,–1) – imagein( 1, 1);
    mask(3) := –imagein(–1, 0) – 2*imagein(–1, 1) + imagein( 0,–1)
               – imagein( 0, 1) + 2*imagein( 1,–1) + imagein( 1, 0);
    mask(4) := –imagein(–1,–1) – 2*imagein(–1, 0) – imagein(–1, 1)
               + imagein( 1,–1) + 2*imagein( 1, 0) + imagein( 1, 1);
    mask(5) := –2*imagein(–1,–1) – imagein(–1, 0) – imagein( 0,–1)
               + imagein( 0, 1) + imagein( 1, 0) + 2*imagein( 1, 1);
    mask(6) := –imagein(–1,–1) + imagein(–1, 1) –2*imagein( 0,–1)
               + 2*imagein( 0, 1) – imagein( 1,–1) + imagein( 1, 1);
    mask(7) := imagein(–1, 0) + 2*imagein(–1, 1) – imagein( 0,–1)
               + imagein( 0, 1) – 2*imagein( 1,–1) – imagein( 1, 0);
    maskmax := mask(0);
    dir := 0;
```

```
    for i in 1..7 loop
        if  mask(i) > maskmax  then
            maskmax := mask(i);
            dir := i;
        end if;
    end loop;

    magout := maskmax;
    dirout := dir;

end next;

end egrs2;
```

PURPOSE: Check the local connectivity of an images edges and delete the edges that do not meet specified conditions. EGRS3 uses the output of EGRS2 as input.

CALL SYNTAX: egrs3(*magin, dirin, imageout)*

INPUT PARAMETERS:

magin: The input image magnitude. This magnitude indicates the edge strength.

dirin: The input image direction.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure egrs3(   magin      : in image byte,
                   dirin      : in image array (–1..1, –1..1) of byte border 0,
                   imageout   : out image byte)

is

next begin

    if  dirin(0,0) = 0  then
        if  dirin(–1, 0)=1 or dirin(–1, 0)=7  then
            imageout := magin;
        else
            imageout := 0;
        end if;

        if  dirin( 1, 0)=1 or dirin( 1, 0)=7  then
            imageout := magin;
        else
            imageout := 0;
        end if;
    end if;

    if  dirin(0,0)=1 or dirin(0,0)=5  then
        if  abs(dirin( 0, 0) – dirin(–1,–1))=1  then
            imageout := magin;
        else
            imageout := 0;
        end if;

        if  abs(dirin( 0, 0) – dirin( 1, 1))=1  then
            imageout := magin;
        else
            imageout := 0;
        end if;
    end if;
```

```
    if  dirin(0,0)=2 or dirin(0,0)=6  then
        if  abs(dirin( 0, 0)–dirin( 0,–1))=1  then
            imageout := magin;
        else
            imageout := 0;
        end if;
        if  abs(dirin( 0, 0) – dirin( 0, 1))=1  then
            imageout := magin;
        else
            imageout := 0;
        end if;
    end if;
    if  dirin(0,0) = 3  then
        if  abs(dirin( 0, 0) – dirin( 1,–1))=1  then
            imageout := magin;
        else
            imageout := 0;
        end if;
        if  abs(dirin( 0, 0) – dirin(–1, 1))=1  then
            imageout := magin;
        else
            imageout := 0;
        end if;
    end if;
    if  dirin(0,0) = 4  then
        if  abs(dirin( 0, 0) – dirin(–1, 0))=1  then
            imageout := magin;
        else
            imageout := 0;
        end if;
        if  abs(dirin( 0, 0) – dirin( 1, 0))=1  then
            imageout := magin;
        else
            imageout := 0;
        end if;
    end if;
    if  dirin(0,0) = 7  then
        if  dirin( 1,–1)=0 or dirin( 1,–1)=6  then
            imageout := magin;
        else
            imageout := 0;
        end if;
        if  dirin(–1, 1)=0 or dirin(–1, 1)=6  then
            imageout := magin;
        else
            imageout := 0;
        end if;
    end if;
end next;
end egrs3;
```

PURPOSE: Perform edge detection using the Sobel operator. EGSB1 outputs the edge magnitude only.

CALL SYNTAX: egsb1(*imagein*, *type, imageout*)

INPUT PARAMETERS:

imagein: The input image.

type: A constant to select the magnitude computation equation.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure egsb1(  imagein    : in image array (–1..1, –1..1) of byte border 0,
                  type       : in integer,
                  imageout   : out image byte)

is

    horz, vert : integer;

next begin

    horz := imagein(–1,–1) + 2*imagein(–1, 0) + imagein(–1, 1)
            – imagein( 1,–1) – 2*imagein( 1, 0) – imagein( 1, 1);

    vert := imagein(–1,–1) + 2*imagein( 0,–1) + imagein( 1,–1)
            – imagein(–1, 1) – 2*imagein( 0, 1) – imagein( 1, 1);

    if type = 1 then
        imageout := sqrt(float(horz*horz + vert*vert));
    else
        imageout := abs(horz) + abs(vert);
    end if;

emd next;

end egsb1;
```

PURPOSE: Perform edge detection using the Sobel operator. EGSB2 outputs the edge magnitude and direction of the gradient.

CALL SYNTAX: egsb2(*imagein*, *type, magout, dirout*)

INPUT PARAMETERS:

imagein: The input image.

type: A constant to select the magnitude computation equation.

OUTPUT PARAMETERS:

magout: The output image magnitude.

dirout: The output image direction.

PROCEDURE:

```
procedure egsb2(    imagein    : in image array (–1..1, –1..1) of byte border 0,
                    type       : in integer,
                    magout     : out image byte,
                    dirout     : out image float)

is

    horiz, vert, temp1, temp2, temp : integer;

next begin

    temp1 := imagein(–1,–1) – imagein( 1, 1);
    temp2 := imagein(–1, 1) – imagein( 1,–1);

    horiz :=  temp1 + temp2 + 2*(imagein(–1, 0) – imagein( 1, 0));
    vert  :=  temp1 – temp2 + 2*(imagein( 0,–1) – imagein( 0, 1));

    if  type = 1  then
        magout := sqrt(float(horiz*horiz + vert*vert));
    else
        magout := abs(horiz) + abs(vert);
    end if;

    if (vert = horiz) then
        dirout := 0.0;
    else
        dirout := atan2(float(vert), float(horiz));
    end if;

end next;

end egsb2;
```

PURPOSE: Perform one iteration of iterative edge detection using the Kasvand's method with a (2N+1) x (2M+1) window. The window parameters N and M are compiled into this procedure.

CALL SYNTAX: eikv1(*imagein, imageout)*

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
#define N 1
#define M 1

procedure eikv1(   imagein    : in image array (–N..N,–M..M) of byte border 0,
                   imageout   : out image byte)

is

    sum, i, j  :  integer;

next begin

    sum := 0;

    for i in  –N..N loop
        for j in  –M..M loop
            sum := sum + imagein(i,j);
        end loop;
    end loop;

    imageout := imagein(0,0) – sum/(2*N+1)*(2*M+1);

end next;

end eikv1;
```

PURPOSE: Perform one iteration of iterative line sharpening using the Kasvand's method with a 5x5 window.

CALL SYNTAX: eikv2(*img, imageout*)

INPUT PARAMETERS:

img: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure eikv2(    img         : in image array (–2..2, –2..2) of byte border 0,
                    imageout    : out image byte)
is
    mask        : array (0..3) of integer;
    maskmax     : integer;
    i           : integer;
next begin
    mask(0) := –img(–2,–2) – img(–2,–1) – img(–2, 0) – img(–2, 1) – img(–2, 2)
               + 2*img( 0,–2) + 2*img( 0,–1) + 2*img( 0, 0)
               + 2*img( 0, 1) + 2*img( 0, 2) – img( 2,–2)
               – img( 2,–1) – img( 2, 0) – img( 2, 1) – img( 2, 2);
    mask(1) := –img(–2,–1) – img(–2, 0) + 2*img(–2, 2) – img(–1,–2) – img(–1,–1)
               + 2*img(–1, 1) – img( 0,–2) + 2*img( 0, 0) – img( 0, 2)
               + 2*img( 1,–1) – img( 1, 1) – img( 1, 2)
               + 2*img( 2,–2) – img( 2, 0) – img( 2, 1);
    mask(2) := –img(–2,–2) + 2*img(–2, 0) – img(–2, 2) – img(–1,–2)
               + 2*img(–1, 0) – img(–1, 2) – img( 0,–2) + 2*img( 0, 0)
               – img( 0, 2) – img( 1,–2) + 2*img( 1, 0) – img( 1, 2)
               – img( 2,–2) + 2*img( 2, 0) – img( 2, 2);
    mask(3) := 2*img(–2,–2) – img(–2, 0) – img(–2, 1) + 2*img(–1,–1)
               – img(–1, 1) – img(–1, 2) – img( 0,–2) + 2*img( 0, 0)
               – img( 0, 2) – img( 1,–2) – img( 1,–1) + 2*img( 1, 1)
               – img( 2,–1) – img( 2, 0) + 2*img( 2, 2);
    maskmax := mask(0);
    for i in 1..3 loop
        if  mask(i) > maskmax  then
            maskmax := mask(i);
        end if;
    end loop;
    if  maskmax >= 0  then
        imageout := maskmax;
    else
        imageout := 0;
    end if;
end next;
end eikv2;
```

PURPOSE: Expand or contract a binary image.

CALL SYNTAX: epct(*imagein, connect, op, imageout*)

INPUT PARAMETERS:

imagein: The input image.

connect: The neighbor connectedness (4 or 8).

op: If *op*=0, contract the image, otherwise, expand the image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure epct(   imagein    : in image array (–1..1, –1..1) of byte border 0,
                  connect    : in integer,
                  op         : in integer,
                  imageout   : out image byte)
is
    i00, i01, i02, i10, i12, i20, i21, i22, temp : integer;
next begin
    i00 := imagein(–1,–1);   i01 := imagein(–1, 0);
    i02 := imagein(–1, 1);        i10 := imagein( 0,–1);
    i12 := imagein( 0, 1);        i20 := imagein( 1,–1);
    i21 := imagein( 1, 0);        i22 := imagein( 1, 1);

    if  op/=0  then
        i00:= 1–i00;              i01:= 1–i01;
        i02:= 1–i02;              i10:= 1–i10;
        i12:= 1–i12;              i20:= 1–i20;
        i21:= 1–i21;              i22:= 1–i22;
    end if;

    temp := i01 * i10 * i12 * i21;
    if  connect=8  then
        temp := temp * i00 * i02 * i20 * i22;
    end if;

    if  op/=0  then
        imageout := 1–temp;
    else
        imageout := temp;
    end if;

end next;

end epct;
```

PURPOSE: Erase regions whose area is less than a specified threshold from a labeled image.

CALL SYNTAX: ersr3(*imagein, area, athr, imageout*)

INPUT PARAMETERS:

imagein: The input image.

area: The areas of regions of the image.

athr: The area threshold.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure ersr3(   imagein    : in image byte,
                   area       : const array (0..255) of integer,
                   athr       : in integer,
                   imageout   : out image byte)

is

    img : integer;

next begin

    img := imagein;

    if  area(img)<athr  then
        imageout := 0;
    else
        imageout := img;
    end if;

end next;

end ersr3;
```

PURPOSE: Expand an image by averaging the image to double resolution.

CALL SYNTAX: expand(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure expand(    imagein     : in image array  (–1..1,  –1..1) of byte border 0,
                     imageout    : out image array (0..1, 0..1) of byte)
is
next begin
    imageout(0,0) := (imagein(–1,–1) + 3 * imagein(–1, 0)
                   + 3 * imagein( 0,–1) + 9 *  imagein( 0, 0)) / 16;
    imageout(0,1) := (3 * imagein(–1, 0) + imagein(–1, 1)
                   + 9 * imagein( 0, 0) + 3 *  imagein( 0, 1)) / 16;
    imageout(1,0) := (3 * imagein(0,–1) + 9 * imagein( 0, 0)
                   + imagein( 1,–1) + 3 *  imagein( 1, 0)) / 16;
    imageout(1,1) := (9 * imagein( 0, 0) + 3 * imagein( 0, 1)
                   + 3 * imagein( 1, 0) + imagein( 1, 1)) / 16;
end next;
end expand;
```

PURPOSE: Assign zero to an image.

CALL SYNTAX: fclib(*imageout)*

INPUT PARAMETERS:

none

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure fclib(imageout : out image byte)
is
next begin
    imageout := 0;
end next;
end fclib;
```

PURPOSE: Assign zero to a real image.

CALL SYNTAX: fclir(*imageout)*

INPUT PARAMETERS:

none

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure fclir(imageout : out image float)
is
next begin
    imageout := 0.0;
end next;
end fclir;
```

PURPOSE: Convert a real/imaginary complex representation to an amplitude/phase representation.

CALL SYNTAX: fcpl(*imginreal, imginimag, imgoutampl, imgoutphas)*

INPUT PARAMETERS:

imginfloat: The real part of the input image.

imginimag: The imaginary part of the input image.

OUTPUT PARAMETERS:

imgoutampl: The amplitude part of the output image.

imgoutphas: The phase part of the output image.

PROCEDURE:

```
#define C        6.283185307

procedure fcpl(   imginreal     : in image float,
                  imginimag     : in image float,
                  imgoutampl    : out image float,
                  imgoutphas    : out image float)

is

next begin

    imgoutampl := sqrt(imginreal*imginreal + imginimag*imginimag);
    imgoutphas := atan2(imginimag, imginreal);

    if  imginimag<0.0   then
        imgoutphas := imgoutphas + C;
    end if;

end next;

end fcpl;
```

PURPOSE: Compute the natural or base-10 log of a real image.

CALL SYNTAX: flog(*imagein, lntype, imageout)*

INPUT PARAMETERS:

imagein: The input image.

lntype: 1 for natural logarithm; 2 for base-10 logarithm.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure flog(   imgin    : in image float,
                  lntype   : in integer,
                  imgout   : out image float)

is

next begin

    imgout := log(imgin);
    if lntype = 2 then
        imgout := imgout*0.434294482;
    end if;

end next;

end flog;
```

PURPOSE: Perform linear filtering using an NxN weighting function. The window size, N, is compiled in the procedure.

CALL SYNTAX: flwl0(*imagein, weights, imageout*)

INPUT PARAMETERS:

imagein: The input image.

weights: The weighting function.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
#define N 1

procedure flwl0(   imagein    : in image array (0..N, 0..N) of byte border 0,
                   weights    : in array (0..N, 0..N) of float,
                   imageout   : out image byte)

is

    sum     : float;
    i, j    : integer;

next begin

    sum := 0;
    for i in 0..N loop
        for j in 0..N loop
            sum := sum + weights(i,j) * imagein(i,j);
        end loop;
    end loop;

    imageout := sum;

end next;

end flwl0;
```

PURPOSE: Perform linear filtering using an NxN weighting function. FLWL1 includes normalization. The window size, N, is compiled in the procedure.

CALL SYNTAX: flwl1(*imagein, weights, gain, imageout*)

INPUT PARAMETERS:

imagein: The input image.

weights: The weighting function.

gain: The normalization coefficient.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define (N, 1)

procedure flwl1(   imagein    : in image array (0..N, 0..N) of byte border 0,
                   weights    : in array (0..N, 0..N) of float,
                   gain       : in float,
                   imageout   : out image byte)

is

    sum     : float;
    i, j    : integer;

next begin

    sum := 0;
    for i in 0..N loop
        for j in 0..N loop
            sum := sum + weights(i,j) * imagein(i,j);
        end loop;
    end loop;

    imageout := gain*sum;

end next;

end flwl1;
```

PURPOSE: Perform local max filtering in a (2*N+1)x(2*N+1) window. N is compiled into the procedure.

CALL SYNTAX: fmax(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define (N, 1)

procedure fmax(   imagein    : in image array (–N..N, –N..N) of byte border 0,
                  imageout   : out image byte)

is

     dmax    : array (–N..N) of integer;
     i, j    : integer;

next begin

     for i in –N..N loop
          dmax(i) := imagein(i, –N);
     end loop;

     for i in –N..N loop
          for j in –N..N loop
               if  imagein(i, j) > dmax(j)  then
                    dmax(j) := imagein(i, j);
               end if;
          end loop;
     end loop;

     imageout := dmax(N);

     for j in –N..N loop
          if dmax(j) > imageout  then
               imageout := dmax(j);
          end if;
     end loop;

end next;

end fmax;
```

PURPOSE: Perform local min filtering in a (2*N+1)x(2*N+1) window. N is compiled into the procedure.

CALL SYNTAX: fmin(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define (N, 1)

procedure fmin(     imagein     : in image array (–N..N, –N..N) of byte  border 0,
                    imageout    : out image byte)

is

    dmin    : array(–N..N) of integer;
    i, j    : integer;

next begin

    for i in –N..N loop
        dmin(i) := imagein(i, –N);
    end loop;

    for i in –N..N loop
        for j in –N..N loop
            if  imagein(i, j) < dmin(j)  then
                dmin(j) := imagein(i, j);
            end if;
        end loop;
    end loop;

    imageout := dmin(–N);

    for j in –N..N loop
        if dmin(j) < imageout  then
            imageout := dmin(j);
        end if;
    end loop;

end next;

end fmin;
```

PURPOSE: Compute the Fourier power spectrum for an image.

CALL SYNTAX: fpwc*(r, i, power)*

INPUT PARAMETERS:

r: The real part of the input image.

i: The imaginary part of the input image.

OUTPUT PARAMETERS:

power: The Fourier power spectrum

PROCEDURE:

```
procedure fpwc(  r       :  in image float,
                 i       :  in image float,
                 power   : out image float)
is
next begin
    power := r * r + i * i;
end next;
end fpwc;
```

PURPOSE: Convert an image to a complex image.

CALL SYNTAX: fsed*(imagein, imageoutreal, imageoutimag)*

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageoutreal: The float part of the output image.

imageoutimag: The imaginary part of the output image.

PROCEDURE:

```
procedure fsed(    imagein        : in image byte,
                   imageoutreal   : out image float,
                   imageoutimag   : out image float)

is

next begin

    imageoutreal := imagein;
    imageoutimag := 0.0;

en next;

end fsed;
```

PURPOSE: Multiply an image by a constant.

CALL SYNTAX: gmlt(*image1, constant, image2*)

INPUT PARAMETERS:

image1: the input image

constant: an integer constant

OUTPUT PARAMETERS:

image2: the output image

PROCEDURE:

```
procedure gmlt(  image1     : in image byte,
                 constant   : in integer,
                 image2     : out image byte)
is
next begin
    image2 := image1 * constant;
end next;
end gmlt;
```

PURPOSE: Add a constant to an image.

CALL SYNTAX: gsft(*image1, constant, image2*)

INPUT PARAMETERS:

image1: the input image

constant: an integer constant

OUTPUT PARAMETERS:

image2: the output image

PROCEDURE:

```
procedure gsft(   image1    : in image byte,
                  constant  : in integer,
                  image2    : out image byte)
is
next begin
    image2 := image1 + constant;
end next;
end gsft;
```

PURPOSE: Perform gray-scale transformation using a transformation table.

CALL SYNTAX: gtrn1(*image1, itable, image2*)

INPUT PARAMETERS:

image1: The input image.

itable: The transformation table.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure gtrn1(   image1   : in image byte,
                   itable   : in array (0..255) of integer,
                   image2   : out image byte)
is
    img : integer;
next begin
    img := image1;
    image2 := itable(img);
end next;
end gtrn1;
```

PURPOSE: Perform one iteration of an iterative enhancement for removing noise from an image.

CALL SYNTAX: iten1(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure iten1(    imagein    : in image array (–1..1, –1..1) of byte,
                    imageout   : out image byte)
is
    a1, a2, a3, a4, a5, a6, a7, a8, a9, Sall, imax, m, tmp, sum, c : integer;
    d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11 : integer;
    S, O, w             : array (0..11) of float;
    invs                : float;
    i                   : integer;
next begin
    a1 := imagein(–1,–1);   a2 := imagein(–1,0);    a3 := imagein(–1,1);
    a4 := imagein(0,–1);    a5 := imagein(0,0);     a6 := imagein(0,1);
    a7 := imagein(1,–1);    a8 := imagein(1,0);     a9 := imagein(1,1);
    Sall := a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9;
    S(0) := a1 + a2 + a3;   S(1) := a1 + a2 + a4;   S(2) := a1 + a4 + a7;
    S(3) := a4 + a7 + a8;   S(4) := a7 + a8 + a9;   S(5) := a6 + a8 + a9;
    S(6) := a3 + a6 + a9;   S(7) := a2 + a3 + a6;   S(8) := a4 + a5 + a6;
    S(9) := a3 + a5 + a7;   S(10) := a2 + a5 + a8;  S(11) := a1 + a5 + a9;
    d0 := Sall–S( 0);       d1 := Sall–S( 1);       d2 := Sall–S( 2);
    d3 := Sall–S( 3);       d4 := Sall–S( 4);       d5 := Sall–S( 5);
    d6 := Sall–S( 6);       d7 := Sall–S( 7);       d8 := S( 8);
    d9 := S( 9);            d10 :=  S(10);          d11 :=  S(11);
    for i in 0..11 loop
        O(i) := 3*S(i) – Sall;
    end loop;
    imax:=fabs(O(0));
    m:=O(0);
```

```
    for i in 1..11 loop
        tmp:=fabs(O(i));
        if tmp>imax then
            imax:=tmp;
            m:=O(i);
        end if;
    end loop;

    if m<0 then
        c := –1;
    else
        c:= 1;
    end if;

    for i in 0..11 loop
        O(i) := c*O(i);
    end loop;

    sum := 0;

    for i in 0..11 loop
        if O(i)<0 then
            O(i) := 0;
        else
            sum := sum + O(i);
        end if;
    end loop;

    invs := 1.0/sum;

    for i in 0..11 loop
        w(i) := invs*O(i);
    end loop;

    imageout := 0.16666666*( w( 0)*d0 + w( 1)*d1 + w( 2)*d2 + w( 3)*d3
                + w( 4)*d4 + w( 5)*d5 + w( 6)*d6 + w( 7)*d7 + w( 8)*d8 )
                + 0.33333333*(  w( 9)*d9 + w(10)*d10+ w(11)*d11);

end next;

end iten1;
```

PURPOSE: Perform one iteration of an iterative enhancement for removing noise from an image.

CALL SYNTAX: iten2(*imagein, exptable, imageout*)

INPUT PARAMETERS:

imagein: The input image.

exptable: A table for exp(x).

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure iten2(   imagein    : in image array (–1..1, –1..1) of byte,
                   exptable   : in array (0..999) of float,
                   imageout   : out image byte)
is
    aa, ab, ac, ad, ae, af, ag, ah, ai                               : integer;
    abc, ghi, abd, fhi, adg, cfi, bcf, dgh, def, ceg, beh, aei       : integer;
    ba, bb, bc, bd, be, bf, bg, bh, ca, cb, cc, cd, ce, cf, cg, ch, ci : integer;
    t, w, s                                                          : float;
next begin
    aa := imagein(–1,–1);   ab := imagein(–1, 0);   ac := imagein(–1, 1);
    ad := imagein( 0,–1);   ae := imagein( 0, 0);   af := imagein( 0, 1);
    ag := imagein( 1,–1);   ah := imagein( 1, 0);   ai := imagein( 1, 1);
    abc := aa + ab + ac;    ghi := ag + ah + ai;    abd := aa + ab + ad;
    fhi := af + ah + ai;    adg := aa + ad + ag;    cfi := ac + af + ai;
    bcf := ab + ac + af;    dgh := ad + ag + ah;    def := ad + ae + af;
    ceg := ac + ae + ag;    beh := ab + ae + ah;    aei := aa + ae + ai;

    s := –10.0/(–5.0);

    ba := exptable(integer(s * abs(abc–def) – 0.49999));
    bb := exptable(integer(s * abs(ghi–def) – 0.49999));
    bc := exptable(integer(s * abs(abd–ceg) – 0.49999));
    bd := exptable(integer(s * abs(fhi–ceg) – 0.49999));
    be := exptable(integer(s * abs(adg–beh) – 0.49999));
    bf := exptable(integer(s * abs(cfi–beh) – 0.49999));
    bg := exptable(integer(s * abs(bcf–aei) – 0.49999));
    bh := exptable(integer(s * abs(dgh–aei) – 0.49999));

    if  ba<0.0000000001  then
        ba := 0.0;
    end if;
```

```
    if  bb<0.0000000001  then
        bb := 0.0;
    end if;
    if  bc<0.0000000001  then
        bc := 0.0;
    end if;
    if  bd<0.0000000001  then
        bd := 0.0;
    end if;
    if  be<0.0000000001  then
        be := 0.0;
    end if;
    if  bf<0.0000000001  then
        bf := 0.0;
    end if;
    if  bg<0.0000000001  then
        bg := 0.0;
    end if;
    if  bh<0.0000000001  then
        bh := 0.0;
    end if;
    ca := ba*bc*be;    cb := ba*bc*bg;    cc := ba*bf*bg;    cd := bc*be*bh;
    cf := bd*bf*bg;    cg := bb*be*bh;    ch := bb*bd*bh;    ci := bb*bd*bf;
    ce := 0.1111111;
    t := ca + cb + cc + cd + cf + cg + ch + ci;
    w := 0.8888888/t;
    imageout := (aa*ca + ab*cb + ac*cc+ad*cd + af*cf +ag*cg + ah*ch + ai*ci)*w
                + ae*ce + 0.5;
    if  t < 0.0000000001  then
        imageout := imagein( 0, 0);
    end if;
end next;
end iten2;
```

PURPOSE: Compute the magnitude of an image.

CALL SYNTAX: mag(*gx, gy, mag)*

INPUT PARAMETERS:

gx: The float part of the input image.

gy: The imaginary part of the input image.

OUTPUT PARAMETERS:

mag: The magnitude of the output image.

PROCEDURE:

```
procedure mag(   gx     :  in image float,
                 gy     :  in image float,
                 mag    : out image float)
is
    sq : float;
next begin
    sq := gx * gx + gy * gy;
    mag := SQRT(sq);
end next;
end mag;
```

PURPOSE: Compute the magnitude and direction of an image.

CALL SYNTAX: magdir(*gx, gy, mag, dir)*

INPUT PARAMETERS:

gx: The float part of the input image.

gy: The imaginary part of the input image.

OUTPUT PARAMETERS:

mag: The magnitude of the output image.

dir: The direction of the output image.

PROCEDURE:

```
procedure magdir(  gx   : in image float,
                   gy   : in image float,
                   mag  : out image float,
                   dir  : out image float)
is
    sq : float;
    inv : float;
next begin
    sq := gx * gx + gy * gy;
    mag := SQRT(sq);
    dir := ATAN2( gy , gx);
end next;
end magdir;
```

PURPOSE: Perform a convolution for byte image magnification.

CALL SYNTAX: magnb(*imagein, weights, imageout)*

INPUT PARAMETERS:

imagein: The input image.

weights: The weighting function.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure magnb(   imagein    : in image array (–1..2, –1..2) of byte,
                   weights    : in array (–1..2, –1..2) of float,
                   imageout   : out image byte)
is
    sum : float;
    i, j : integer;
next begin
    sum := 0;
    for i in –1..2 loop
        for j in –1..2 loop
            sum := sum + weights(i,j) * imagein(i,j);
        end loop;
    end loop;
    imageout := integer(sum);
end next;
end magnb;
```

PURPOSE: Perform a convolution for float image magnification.

CALL SYNTAX: magnr(*imagein, weights, imageout)*

INPUT PARAMETERS:

imagein: The input image.

weights: The weighting function.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure magnr(   imagein    : in image array (–1..2, –1..2) of float,
                   weights    : in array (–1..2, –1..2) of float,
                   imageout   : out image float)
is
    sum     : float;
    i, j    : integer;
next begin
    sum := 0;
    for i in –1..2 loop
        for j in –1..2 loop
            sum := sum + weights(i,j) * imagein(i,j);
        end loop;
    end loop;
    imageout := sum;
end next;
end magnr;
```

PURPOSE: Perform median filtering on an image using a 3x3 window.

CALL SYNTAX: median1(*image, si)*

INPUT PARAMETERS:

image: The input image.

OUTPUT PARAMETERS:

si: The output image.

PROCEDURE:

```
procedure median1(   im      : in image array (–1..1, 0..0) of byte,
                     si      : out image array(–1..1, 0..0) of byte)
is
next begin
    if im(–1, 0) > im( 0, 0) then
        if im( 0, 0) > im( 1, 0) then
            si( 1, 0) := im(–1, 0);
            si( 0, 0) := im( 0, 0);
            si(–1, 0) := im( 1, 0);
        else
            if im(–1, 0) > im( 1, 0) then
                si( 1, 0) := im(–1, 0);
                si( 0, 0) := im( 1, 0);
                si(–1, 0) := im( 0, 0);
            else
                si( 1, 0) := im( 1, 0);
                si( 0, 0) := im(–1, 0);
                si(–1, 0) := im( 0, 0);
            end if;
        end if;
    else
        if im( 0, 0) > im( 1, 0) then
            if im(–1, 0) > im( 1, 0) then
                si( 1, 0) := im( 0, 0);
                si( 0, 0) := im(–1, 0);
                si(–1, 0) := im( 1, 0);
            else
                si( 1, 0) := im( 0, 0);
                si( 0, 0) := im( 1, 0);
                si(–1, 0) := im(–1, 0);
            end if;
        else
            si( 1, 0) := im( 1, 0);
            si( 0, 0) := im( 0, 0);
            si(–1, 0) := im(–1, 0);
        end if;
    end if;
end next;
end median1;
```

PURPOSE: Perform median filtering on an image using a 3x3 window.

CALL SYNTAX: median2(*si, median)*

INPUT PARAMETERS:

si: The input image.

OUTPUT PARAMETERS:

median: The output image.

PROCEDURE:

```
procedure median2(   si       : in image array(–1..1, –1..1) of byte sample (3, 1),
                     median   : out image byte)
is
    l, m, h  : integer;
    A, B     : float;
next begin
    if si(–1, 0) > si( 0, 0) then
        if si( 0, 0) > si( 1, 0) then
            h := –1;
            m := 0;
            l := 1;
        else
            if si(–1, 0) > si( 1, 0) then
                h := –1;
                m :=  1;
            else
                h :=  1;
                m := –1;
            end if;
            l := 0;
        end if;
    else
        if si( 0, 0) > si( 1, 0) then
            if si(–1, 0) > si( 1, 0) then
                m := –1;
                l :=  1;
            else
                m :=  1;
                l := –1;
            end if;
            h := 0;
```

```
        else
            h := 1;
            m := 0;
            l := –1;
        end if;
    end if;

    if si(l, –1) > si(m, 1) then
        A := si(l, –1);
    else
        A := si(m, 1);
    end if;

    if si(m, –1) < si(h, 1) then
        B := si(m, –1);
    else
        B := si(h, 1);
    end if;

    if A > si(m, 0) then
        if si(m, 0) > B then
            median := si(m, 0);
        else
            if  A > B  then
                median := B;
            else
                median := A;
            end if;
        end if;
    else
        if si(m, 0) > B then
            if  A > B  then
                median := A;
            else
                median := B;
            end if;
        else
            median := si(m, 0);
        end if;
    end if;
end next;

end median2;
```

PURPOSE: Multiply an image by a constant.

CALL SYNTAX: mulc1b(*image1, constant, image2*)

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure mulc1b(   image1     : in image byte,
                    constant   : in integer,
                    image2     : out image byte)
is
next begin
    image2 := image1 * constant;
end next;
end mulc1b;
```

PURPOSE: Multiply a complex image by a constant.

CALL SYNTAX: mulc1c(*image1r, image1i, constantr, constanti, image2r, image2i*)

INPUT PARAMETERS:

image1r: The real part of the input image.

image1i: The imaginary part of the input image.

constantr: The real part of a constant.

constanti: The imaginary part of a constant.

OUTPUT PARAMETERS:

image2r: The float part of the output image.

image2i: The imaginary part of the output image.

PROCEDURE:

```
procedure mulc1c(   image1r     : in image float,
                    image1i     : in image float,
                    constantr   : in float,
                    constanti   : in float,
                    image2r     : out image float,
                    image2i     : out image float)
is
next begin
    image2r := image1r*constantr – image1i*constanti;
    image2i := image1i*constantr + image1r*constanti;
end next;
end mulc1c;
```

PURPOSE: Multiply a float image by a constant.

CALL SYNTAX: mulc1r(*image1, constant, image2*)

INPUT PARAMETERS:

image1: The input image.

constant: A real constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure mulc1r(   image1     : in image float,
                    constant   : in float,
                    image2     : out image float)
is
next begin
    image2 := image1 * constant;
end next;
end mulc1r;
```

PURPOSE: Multiply a signed_byte image by a constant.

CALL SYNTAX: mulc1s(*image1, constant, image2*)

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure mulc1s(   image1      : in image signed_byte,
                    constant    : in integer,
                    image2      : out image signed_byte)
is
next begin
    image2 := image1 * constant;
end next;
end mulc1s;
```

PURPOSE: Multiply two images.

CALL SYNTAX: mulp1b(*image1, image2, image3*)

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure mulp1b(   image1    : in image byte,
                    image2    : in image byte,
                    image3    : out image byte)
is
next begin
    image3 := image1 * image2;
end next;
end mulp1b;
```

PURPOSE: Multiply two complex images.

CALL SYNTAX: mulp1c(*image1r, image1i, image2r, image2i, image3r, image3i*)

INPUT PARAMETERS:

image1r: The real part of the first input image.

image1i: The imaginary part of the first input image.

image2r: The real part of the second input image.

image2i: The imaginary part of the second input image.

OUTPUT PARAMETERS:

image3r: The real part of the output image.

image3i: The imaginary part of the output image.

PROCEDURE:

```
procedure mulp1c(   image1r   : in image float,
                    image1i   : in image float,
                    image2r   : in image float,
                    image2i   : in image float,
                    image3r   : out image float,
                    image3i   : out image float)
is
next begin
    image3r := image1r*image2r – image1i*image2i;
    image3i := image1i*image2r + image1r*image2i;
end next;
end mulp1c;
```

PURPOSE: Multiply the first complex image by the complex conjugate of the second complex image.

CALL SYNTAX: mulp1ccj(*image1r, image1i, image2r, image2i, image3r, image3i*)

INPUT PARAMETERS:

image1r: The real part of the first input image.

image1i: The imaginary part of the first input image.

image2r: The real part of the second input image.

image2i: The imaginary part of the second input image.

OUTPUT PARAMETERS:

image3r: The real part of the output image.

image3i: The imaginary part of the output image.

PROCEDURE:

```
procedure mulp1ccj(   image1r   : in image float,
                      image1i   : in image float,
                      image2r   : in image float,
                      image2i   : in image float,
                      image3r   : out image float,
                      image3i   : out image float)
is
next begin
    image3r := image1r*image2r + image1i*image2i;
    image3i := image1i*image2r – image1r*image2i;
end next;
end mulp1ccj;
```

PURPOSE: Multiply two real images.

CALL SYNTAX: mulp1r(*image1, image2, image3*)

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure mulp1r(   image1   : in image float,
                    image2   : in image float,
                    image3   : out image float)
is
next begin
    image3 := image1 * image2;
end next;
end mulp1r;
```

PURPOSE: Multiply two signed_byte images.

CALL SYNTAX: mulp1s(*image1, image2, image3*)

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure mulp1s(   image1   : in image signed_byte,
                    image2   : in image signed_byte,
                    image3   : out image signed_byte)
is
next begin
    image3 := image1 * image2;
end next;
end mulp1s;
```

PURPOSE: Generate a non-maxima suppressed image.

CALL SYNTAX: nonmax(*gx, gy, mag, edges)*

INPUT PARAMETERS:

gx: The float part of the input image.

gy: The imaginary part of the input image.

mag: The magnitude of the input image.

OUTPUT PARAMETERS:

edges: The output image.

PROCEDURE:

```
procedure nonmax(   gx      : in image float,
                    gy      : in image float,
                    mag     : in image array (–1..1, –1..1) of float,
                    edges   : out image float)
is
    mag1, mag2, fraction    : float;
    agx, agy                : float;
next begin
    agx := abs(gx);
    agy := abs(gy);

    if agx > agy  then
        fraction := agy / agx;
    else
        fraction := agx / agy;
    end if;

    if gx = 0.0 then
        mag1 := mag(1,0);
        mag2 := mag(–1,0);
    end if;

    if gy = 0.0 then
        mag1 := mag(0,1);
        mag2 := mag(0,–1);
    end if;
```

```
    if gx * gy > 0 then
        if agx > agy then
            mag1 := fraction * mag(1,1)  +  (1–fraction) * mag(0,1);
            mag2 := fraction * mag(–1,–1)  +  (1–fraction) * mag(0,–1);
        else
            mag1 := fraction * mag(1,1)  +  (1–fraction) * mag(1,0);
            mag2 := fraction * mag(–1,–1)  +  (1–fraction) * mag(–1,0);
        end if;
    end if;

    if gx * gy < 0 then
        if agx > agy then
            mag1 := fraction * mag(1,–1)  +  (1–fraction) * mag(0,–1);
            mag2 := fraction * mag(–1,1)  +  (1–fraction) * mag(0,1);
        else
            mag1 := fraction * mag(1,–1)  +  (1–fraction) * mag(1,0);
            mag2 := fraction * mag(–1,1)  +  (1–fraction) * mag(–1,0);
        end if;
    end if;

    edges := mag(0,0);

    if mag(0,0) < mag1 then
         edges := 0.0;
    end if;

    if mag(0,0) < mag2 then
        edges := 0.0;
    end if;

end next;

end nonmax;
```

PURPOSE: Generate a non-maxima suppressed image using a 3x3 window.

CALL SYNTAX: nonmax1(*mag, supp*)

INPUT PARAMETERS:

mag: The input image.

OUTPUT PARAMETERS:

supp: The output image.

PROCEDURE:

```
procedure nonmax1(   mag   : in image array (–1..1, –1..1) of byte,
                     supp  : out image byte)
is
    i, j : integer;
next begin
    supp := mag(0, 0);
    for i in –1..1 loop
        for j in –1..1 loop
            if mag(i, j) > supp then
                supp := 0;
            end if;
        end loop;
    end loop;
end next;
end nonmax1;
```

PURPOSE: Compute the logical NOT of an image.

CALL SYNTAX: not1(*image1, image2*)

INPUT PARAMETERS:

image1: The input image.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure not1(    image1   : in image byte,
                   image2   : out image byte)
is
next begin
    image2 := (lnot image1) land 255;
end next;
end not1;
```

PURPOSE: OR an image with a constant.

CALL SYNTAX: orc1b(*image1, constant, image2*)

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure orc1b(   image1     : in image byte,
                   constant   : in integer,
                   image2     : out image byte)
is
next begin
     image2 := image1 lor constant;
end next;
end orc1b;
```

PURPOSE: OR two images.

CALL SYNTAX: orp1b(*image1, image2, image3*)

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure orp1b(   image1   : in image byte,
                   image2   : in image byte,
                   image3   : out image byte)
is
next begin
    image3 := image1 lor image2;
end next;
end orp1b;
```

PURPOSE: Extract or delete points in an image which have values in a specified range.

CALL SYNTAX: pted(*imagein, mode1, lthd, hthd, mode2, labl, imageout*)

INPUT PARAMETERS:

imagein: The input image.

mode1: If 0, delete points; if 1, extract points.

lthd: The lower limit of the threshold.

hthd: The upper limit of the threshold.

mode2: If 0, extract or delete points within threshold range; if 1, extract or delete points outside threshold range.

labl: The label assigned to extracted points.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure pted(   imagein    : in image byte,
                  mode1      : in integer,
                  lthd       : in integer,
                  hthd       : in integer,
                  mode2      : in integer,
                  lbl        : in integer,
                  imageout   : out image byte)

is

next begin

    imageout := imagein;

    if  mode1/=0  then
        if lbl/=0 then
            imageout := lbl;
        end if;
    end if;

    if mode1=0 then
        if mode2=0 then
            if lthd<imagein then
                if imagein<hthd then
                    imageout := 0;
                end if;
            end if;
        else
            if lthd>imagein then
```

```
                imageout := 0;
            end if;

            if imagein>hthd then
                imageout := 0;
            end if;
        end if;

    else
        if mode2=0 then
            if lthd>imagein then
                imageout := 0;
            end if;

            if imagein>hthd then
                imageout := 0;
            end if;
        else
            if lthd<imagein then
                if imagein<hthd then
                    imageout := 0;
                end if;
            end if;
        end if;
    end if;

end next;

end pted;
```

PURPOSE: Create an image pyramid by image averaging.

CALL SYNTAX: pyramid(*imagein, imageout1, imageout2, imageout3, imageout4, imageout5, imageout6*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout1: A 1/2 size output image.

imageout2: A 1/4 size output image.

imageout3: A 1/8 size output image.

imageout4: A 1/16 size output image.

imageout5: A 1/32 size output image.

imageout6: A 1/64 size output image.

PROCEDURE:

```
procedure pyramid(   imagein      : in image array (0..63, 0..63) of byte border 0
                                                          sample (64, 64),
                     imageout1    : out image array (0..31, 0..31) of byte,
                     imageout2    : out image array (0..15, 0..15) of byte,
                     imageout3    : out image array (0..7,  0..7) of byte,
                     imageout4    : out image array (0..3,  0..3) of byte,
                     imageout5    : out image array (0..1,  0..1) of byte,
                     imageout6    : out image byte)

is
    i, j : integer;

next begin
    for i in 0..31 loop
        for j in 0..31 loop
            imageout1(i, j) := (imagein(2*i, 2*j) + imagein(2*i+1, 2*j)
                            + imagein(2*i, 2*j+1) + imagein(2*i+1, 2*j+1)) / 4;
        end loop;
    end loop;

    for i in 0..15 loop
        for j in 0..15 loop
            imageout2(i, j) :=(imageout1(2*i, 2*j) + imageout1(2*i+1, 2*j)
                            + imageout1(2*i, 2*j+1) + imageout1(2*i+1, 2*j+1)) / 4;
        end loop;
    end loop;
```

```
     for i in 0..7 loop
          for j in 0..7 loop
               imageout3(i, j) :=(imageout2(2*i, 2*j) + imageout2(2*i+1, 2*j)
                                   + imageout2(2*i, 2*j+1) + imageout2(2*i+1, 2*j+1)) / 4;
          end loop;
     end loop;

     for i in 0..3 loop
          for j in 0..3 loop
               imageout4(i, j) :=(imageout3(2*i, 2*j) + imageout3(2*i+1, 2*j)
                                   + imageout3(2*i, 2*j+1) + imageout3(2*i+1, 2*j+1)) / 4;
          end loop;
     end loop;

     for i in 0..1 loop
          for j in 0..1 loop
               imageout5(i, j) :=(imageout4(2*i, 2*j) + imageout4(2*i+1, 2*j)
                                   + imageout4(2*i, 2*j+1) + imageout4(2*i+1, 2*j+1)) / 4;
          end loop;
     end loop;

     imageout6 :=(imageout5(0, 0) + imageout5(1, 0) + imageout5(0, 1)
                    + imageout5(1, 1)) / 4;

end next;

end pyramid;
```

PURPOSE: Reduce an image by averaging to half resolution.

CALL SYNTAX: reduce(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure reduce(   imagein    : in image array (0..1, 0..1) of byte border 0
                                                        sample (2, 2),
                    imageout   : out image byte)
is
next begin
    imageout := (imagein(0,0) + imagein(0,1) + imagein(1,0) + imagein(1,1))/4;
end next;
end reduce;
```

PURPOSE: Convert an image from real to byte.

CALL SYNTAX: rlby(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure rlby(   imagein    : in image float,
                  imageout   : out image byte)
is
next begin
    imageout := imagein;
end next;
end rlby;
```

PURPOSE: Convert an image from real to byte with wraparound.

CALL SYNTAX: rlby1(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure rlby1(    imagein     : in image float,
                    imageout    : out image byte)
is
next begin
    imageout := imagein – integer(imagein / 256.0 – 0.49999) * 256.0;
end next;
end rlby1;
```

PURPOSE: Assign a constant to an image.

CALL SYNTAX: rpla1b(*constant, imageout*)

INPUT PARAMETERS:

constant: An integer constant.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure rpla1b(    constant    : in integer,
                     imageout    : out image byte)
is
next begin
    imageout := constant;
end next;
end rpla1b;
```

PURPOSE: Assign a constant to a real image.

CALL SYNTAX: rpla1r(*constant, imageout*)

INPUT PARAMETERS:

constant: A float constant.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure rpla1b(     constant     : in float,
                      imageout     : out image float)
is
next begin
     imageout := constant;
end next;
end rpla1r;
```

PURPOSE: Requantize an image by reducing the graylevels.

CALL SYNTAX: rqnt(*imagein, coef, imageout*)

INPUT PARAMETERS:

imagein: The input image.

coef: The degree of reduction

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure rqnt(    imagein    : in image byte,
                   coef       : in integer,
                   imageout   : out image byte)
is
    pix     : float;
    dpix    : float;
next begin
    pix := imagein;
    dpix := coef * integer(float(imagein)/coef);
    if dpix > pix then
        dpix := dpix – coef;
    end if;
    imageout := dpix;
end next;
end rqnt;
```

PURPOSE: Binarize an image by setting nonzero grayvalues to 1.

CALL SYNTAX: sclp(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure sclp(    imagein     : in image byte,
                   imageout    : out image byte)
is
next begin
    if  imagein = 0  then
        imageout := 0;
    else
        imageout := 1;
    end if;
end next;
end sclp;
```

PURPOSE: Binarize an image using a single threshold value.

CALL SYNTAX: slth0(*imagein, thd, imageout*)

INPUT PARAMETERS:

imagein: The input image.

thd: The threshold value.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure slth0(   imagein    : in image byte,
                   thd        : in integer,
                   imageout   : out image byte)

is
next begin

    if imagein > thd then
        imageout:=1;
    else
        imageout:=0;
    end if;

end next;

end slth0;
```

PURPOSE: Binarize a real image using a single threshold value.

CALL SYNTAX: slth0r(*imagein, thd, imageout*)

INPUT PARAMETERS:

imagein: The input image.

thd: The threshold value.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure slth0r(    imagein    : in image float,
                     thd        : in float,
                     imageout   : out image byte)
is
next begin
    if imagein > thd then
        imageout:=1;
    else
        imageout:=0;
    end if;
end next;
end slth0r;
```

PURPOSE: Binarize an image using a single threshold value.

CALL SYNTAX: slth1(*imagein, thd, mode, imageout*)

INPUT PARAMETERS:

imagein: The input image.

thd: The threshold value.

mode: The binarization mode.

If *mode*=1, *imageout*(i,j) = 1 if *imagein*(i,j) > *thd*
If *mode*=2, *imageout*(i,j) = 1 if *imagein*(i,j) >= *thd*
If *mode*=3, *imageout*(i,j) = 1 if *imagein*(i,j) < *thd*
If *mode*=4, *imageout*(i,j) = 1 if *imagein*(i,j) <= *thd*

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure slth1(   input       : in image byte,
                   thd         : in integer,
                   mode        : in integer,
                   imageout    : out image byte)
is
next begin
    imageout := 0;
    if mode = 1 then
        if input > thd then
            imageout := 1;
        end if;
    else
        if mode = 2 then
            if input >= thd then
                imageout := 1;
            end if;
        else
            if mode = 3 then
                if input < thd then
                    imageout := 1;
                end if;
            else
                if mode = 4 then
                    if input <= thd then
                        imageout := 1;
                    end if;
                end if;
            end if;
        end if;
    end if;
end next;
end slth1;
```

PURPOSE: Binarize an image using two threshold values.

CALL SYNTAX: slth2(*imagein, thd1, thd2, mode, imageout*)

INPUT PARAMETERS:

imagein: The input image.

thd1: The upper threshold limit.

thd2: The lower threshold limit.

mode: The binarization mode.
If *mode*=0, *imageout* = 1 if *imagein* is inside the threshold range.
If *mode=1*, *imageout* = 1 if *imagein* is outside the threshold range.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure slth2(   imagein    : in image byte,
                   thd1       : in integer,
                   thd2       : in integer,
                   mode       : in integer,
                   imageout   : out image byte)
is
  d1, d2, pix : integer;
next begin
    pix := imagein;
    d1 := thd1 – pix;
    d2 := pix – thd2;
    imageout := 0;
    if d1>=0 then
        if d2>=0 then
            imageout := 1;
        end if;
    end if;
    if mode /= 0 then
        imageout := 0;
        if d1<=0 then
            imageout := 1;
        end if;
        if d2<=0 then
            imageout := 1;
        end if;
    end if;
end next;
end slth2;
```

PURPOSE: Binarize an image using two threshold values. The operation is performed only on pixels which correspond to nonzero values in the mask plane. Pixels which correspond to zero values in the mask plane are assigned 0 in the output image.

CALL SYNTAX: slth2m(*imagein, mask, thd1, thd2, mode, imageout*)

INPUT PARAMETERS:

imagein: The input image.

mask: The mask plane array.

thd1: The upper threshold limit.

thd2: The lower threshold limit.

mode: The binarization mode.
If *mode*=0, *imageout* = 1 if *imagein* is inside the threshold range.
If *mode*=1, *imageout* = 1 if *imagein* is outside the threshold range.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure slth2m(   imagein    : in image byte,
                    mask       : in image byte,
                    thd1       : in integer,
                    thd2       : in integer,
                    mode       : in integer,
                    imageout   : out image byte)

is

next begin

    if mode=0 then
        imageout := 0;
        if mask /= 0 then
            if ( thd1 <= imagein ) and  ( imagein <= thd2 ) then
                imageout := 1;
            end if
        end if
    else
        imageout := 0;
        if mask /= 0 then
            if ( imagein >= thd1 ) or ( imagein <= thd2 ) then
                imageout := 1;
            end if;
        end if;
    end if;

end next;

end slth2m;
```

PURPOSE: Binarize each pixel in an image using individual thresholds contained in a reference plane.

CALL SYNTAX: slth3(*imagein, reference, mode, imageout*)

INPUT PARAMETERS:

imagein: The input image.

reference: Reference plane containing the threshold value for each pixel.

mode: The binarization mode.
If *mode*=1, *imageout* = 1 if *imagein* > *reference*
If *mode*=2, *imageout* = 1 if *imagein* >= *reference*
If *mode*=3, *imageout* = 1 if *imagein* < *reference*
If *mode*=4, *imageout* = 1 if *imagein* <= *reference*

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure slth3(    imagein     : in image byte,
                    reference   : in image byte,
                    mode        : in integer,
                    imageout    : out image byte)
is
next begin
    imageout := 0;
    if mode = 1 then
        if imagein > reference then
            imageout := 1;
        end if;
    else
        if mode = 2 then
            if imagein >= reference then
                imageout := 1;
            end if;
        else
            if mode = 3 then
                if imagein < reference then
                    imageout := 1;
                end if;
            else
                if mode = 4 then
                    if imagein <= reference then
                        imageout := 1;
                    end if;
                end if;
            end if;
        end if;
    end if;
end next;
end slth3;
```

PURPOSE: Perform smoothing using Harwood's symmetric nearest neighbor algorithm. The computation window size is compiled in the procedure, and it must be an odd number.

CALL SYNTAX: snns(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define(windowsize, 15)
define(wb2, eval(windowsize/2))

procedure snns(   imagein    : in image array(–wb2..wb2, –wb2..wb2) of byte,
                  imageout   : out image byte)

is

    i,j     : integer;
    sum     : float;

next begin

    sum := 0;

    for i in –wb2..wb2 loop
        for j in i+1..wb2 loop
        if abs(imagein(0,0)–imagein(i,j)) < abs(imagein(0,0)–imagein(–i,–j)) then
            sum := sum + imagein(i,j);
        else
            sum := sum + imagein(–i,–j);
        end if;
        end loop;
    end loop;

    imageout := sum / eval(wb2 * windowsize);

end next;

end snns;
```

PURPOSE: Shrink a binary pattern using Levialdi's parallel algorithm.

CALL SYNTAX: srnk1(*imagein, connect, imageout*)

INPUT PARAMETERS:

imagein: The input image.

connect: The neighbor connectedness (4 or 8).

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure srnk1(    imagein    : in image array (0..1, 0..1) of byte,
                    connect    : in integer,
                    imageout   : out image byte)
is
    a, b, c, d : integer;
next begin
    a := imagein(0,0);
    b := imagein(0,1);
    c := imagein(1,0);
    d := imagein(1,1);
    if connect/=4 then
        a := 1–a;
        b := 1–b;
        c := 1–c;
        d := 1–d;
    end if;
    if  a=0  then
        imageout := b*c*d;
    else
        imageout := 1–(1–b)*(1–c);
    end if;
    if connect/=4 then
        imageout := 1–imageout;
    end if;
end next;
end srnk1;
```

PURPOSE: Shrink a binary pattern using the parallel algorithm of Rao et al (single iteration).

CALL SYNTAX: srnk3(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure srnk3(    imagein     : in image array (0..2, 0..2) of byte,
                    imageout    : out image byte)
is
    ia0, ia1, ia2, ia3, ia4, ia5, ia6, ia7 , ia8, temp : float;
next begin
    ia0 := imagein(1,1);        ia1 := imagein(2,1);        ia2 := imagein(2,0);
    ia3 := imagein(1,0);        ia4 := imagein(0,0);        ia5 := imagein(0,1);
    ia6 := imagein(0,2);        ia7 := imagein(1,2);        ia8 := imagein(2,2);

    imageout := ia0;
    temp := ia1+ia3+ia5+ia7;
    if temp/=4 then
        if (temp+ia2+ia4+ia6+ia8)/=0 then
            imageout := 0;
        end if;
    end if;
end next;
end srnk3;
```

PURPOSE: Subtract a constant from an image.

CALL SYNTAX: subc1b(*image1, constant, image2*)

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure subc1b(   image1      : in image byte,
                    constant    : in integer,
                    image2      : out image byte)
is
next begin
    image2 := image1 – constant;
end next;
end subc1b;
```

PURPOSE: Subtract a constant from a complex image.

CALL SYNTAX: subc1c(*image1r, image1i, constantr, constanti, image2r, image2i*)

INPUT PARAMETERS:

image1r: The real part of the input image.

image1i: The imaginary part of the input image.

constantr: The real part of a constant.

constanti: The imaginary part of a constant.

OUTPUT PARAMETERS:

image2r: The real part of the output image.

image2i: The imaginary part of the output image.

PROCEDURE:

```
procedure subc1c(   image1r    : in image float,
                    image1i    : in image float,
                    constantr  : in float,
                    constanti  : in float,
                    image2r    : out image float,
                    image2i    : out image float)
is
next begin
    image2r := image1r – constantr;
    image2i := image1i – constanti;
end next;
end subc1c;
```

PURPOSE: Subtract a constant from a real image.

CALL SYNTAX: subc1r(*image1, constant, image2*)

INPUT PARAMETERS:

image1: The input image.

constant: A real constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure subc1r(   image1      : in image float,
                    constant    : in float,
                    image2      : out image float)
is
next begin
    image2 := image1 – constant;
end next;
end subc1r;
```

PURPOSE: Subtract a constant from a signed byte image.

CALL SYNTAX: subc1s(*image1, constant, image2*)

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure subc1s(   image1     : in image signed_byte,
                    constant   : in integer,
                    image2     : out image signed_byte)
is
next begin
    image2 := image1 – constant;
end next;
end subc1s;
```

PURPOSE: Subtract the second input image from the first input image.

CALL SYNTAX: subp1b(*image1, image2, image3*)

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure subp1b(   image1   : in image byte,
                    image2   : in image byte,
                    image3   : out image byte)
is
next begin
    image3 := image1 – image2;
end next;
end subp1b;
```

PURPOSE: Subtract the second complex input image from the first complex input image.

CALL SYNTAX: subp1c(*image1r, image1i, image2r, image2i, image3r, image3i*)

INPUT PARAMETERS:

image1r: The real part of the first input image.

image1i: The imaginary part of the first input image.

image2r: The real part of the second input image.

image2i: The imaginary part of the second input image.

OUTPUT PARAMETERS:

image3r: The real part of the output image.

image3i: The imaginary part of the output image.

PROCEDURE:

```
procedure subp1c(   image1r    : in image float,
                    image1i    : in image float,
                    image2r    : in image float,
                    image2i    : in image float,
                    image3r    : out image float,
                    image3i    : out image float)
is
next begin
    image3r := image1r – image2r;
    image3i := image1i – image2i;
end next;
end subp1c;
```

PURPOSE: Subtract the second real image from the first real image.

CALL SYNTAX: subp1r(*image1, image2, image3*)

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure subp1r(   image1   : in image float,
                    image2   : in image float,
                    image3   : out image float)
is
next begin
    image3 := image1 – image2;
end next;
end subp1r;
```

PURPOSE: Subtract the second signed byte image from the first signed byte image.

CALL SYNTAX: subp1s(*image1, image2, image3*)

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure subp1s(   image1   : in image signed_byte,
                    image2   : in image signed_byte,
                    image3   : out image signed_byte)
is
next begin
    image3 := image1 – image2;
end next;
end subp1s;
```

PURPOSE: Thin one directional texture edges.

CALL SYNTAX: temx2(*imagein, cosvalue, sinvalue, imageout)*

INPUT PARAMETERS:

imagein: The input image.

cosvalue: The cosine of the edge direction.

sinvalue: The sine of the edge direction.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure temx2(   imagein    : in image array (–17..17, –17..17) of byte border 0,
                   cosvalue   : in float,
                   sinvalue   : in float,
                   imageout   : out image byte)
is
    fi, maxvalue, pix     : float;
    v1, v2                : array (0..17) of float;
    px, py, i             : integer;
next begin
    pix := imagein(0,0);
    maxvalue := pix;
    imageout := pix;
    for i in 0..17 loop
        fi := float(i);
        px := integer(sinvalue*fi);
        py := integer(cosvalue*fi);
        v1(i) := imagein(px, py);
        v2(i) := imagein(–px, –py);
    end loop;
    for i in 0..17 loop
        if  maxvalue < v1(i)  then
            maxvalue := v1(i);
        end if;
    end loop;
    for i in 0..17 loop
        if  maxvalue < v2(i)  then
            maxvalue := v2(i);
        end if;
    end loop;
    if  maxvalue > pix  then
        imageout := 0.0;
    end if;
end next;
end temx2;
```

PURPOSE: Smooth an image, preserving texture edges.

CALL SYNTAX: tepa(*imagein, ithd, imageout*)

INPUT PARAMETERS:

imagein: The input image.

ithd: The threshold value.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define(n, 7)

procedure tepa(    imagein     : in image array (eval(–(2*n))..eval(2*n),
                                            eval(–(2*n))..eval(2*n)) of byte,
                   ithd        : in integer,
                   imageout    : out image byte)

is

    a               : array(0..4, 0..4) of float;
    p, h, s         : array(0..4) of float;
    hmin, i, j      : integer;
    res, c          : float;

next begin

    c := 1.0/(eval(2*n+1)*eval(2*n+1));

    p(0) := imagein( 0, 0);
    p(1) := imagein(–n,–n);
    p(2) := imagein(–n, n);
    p(3) := imagein( n,–n);
    p(4) := imagein( n, n);

    for i in 0..4 loop
        for j in 0..4 loop
            a(i, j) := 0;
        end loop;
    end loop;

    for i in 0..n loop
        for j in 1..n loop
            a(0, 1) := a(0, 1) + imagein(–j, –i);
            a(0, 2) := a(0, 2) + imagein(–i,  j);
            a(0, 3) := a(0, 3) + imagein( i, –j);
            a(0, 4) := a(0, 4) + imagein( j,  i);
        end loop;
    end loop;
```

```
    for i in eval(–2*n)..eval(–(n+1)) loop
        for j in eval(–2*n)..–n loop
            a(1, 1) := a(1, 1) + imagein(i, j);
            a(2, 2) := a(2, 2) + imagein(j,–i);
            a(3, 3) := a(3, 3) + imagein(–j,i);
            a(4, 4) := a(4, 4) + imagein(–i,–j);
        end loop;
    end loop;

    for i in eval(–2*n)..n loop
        for j in eval(–(n–1))..0 loop
            a(1, 2) := a(1, 2) + imagein(i, j);
            a(2, 4) := a(2, 4) + imagein(j,–i);
            a(3, 1) := a(3, 1) + imagein(–j,i);
            a(4, 3) := a(4, 3) + imagein(–i,–j);
        end loop;
    end loop;

    for i in –n..0 loop
        for j in eval(–2*n)..eval(–(n+1)) loop
            a(1, 3) := a(1, 3) + imagein(i, j);
            a(2, 1) := a(2, 1) + imagein(j,–i);
            a(3, 4) := a(3, 4) + imagein(–j,i);
            a(4, 2) := a(4, 2) + imagein(–i,–j);
        end loop;
    end loop;

    a(1, 4) := a(0, 1);
    a(2, 3) := a(0, 2);
    a(3, 2) := a(0, 3);
    a(4, 1) := a(0, 4);

    for i in 0..4 loop
        h(i) := abs(a(i, 1) + a(i, 2) – a(i, 3) – a(i, 4))
                + abs(a(i, 1) + a(i, 3) – a(i, 2) – a(i, 4));
        s(i) := c*(p(i) + a(i, 1) + a(i, 2) + a(i, 3) + a(i, 4));
    end loop;

    hmin := h(0); res := s(0);

    for i in 1..4 loop
        if h(i)<hmin then
            hmin := h(i);
            res := s(i);
        end if;
    end loop;

    imageout := integer(res);

    if ithd>0 then
        if hmin<ithd then
            imageout := integer(p(0));
        end if;
    end if;

end next;

end tepa;
```

PURPOSE: Copy one image to another.

CALL SYNTAX: tfer1b(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure tfer1b(   imagein     : in image byte,
                    imageout    : out image byte)
is
next begin
    imageout := imagein;
end next;
end tfer1b;
```

PURPOSE: Copy one real image to another.

CALL SYNTAX: tfer1r(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure tfer1r(   imagein    : in image float,
                    imageout   : out image float)
is
next begin
    imageout := imagein;
end next;
end tfer1r;
```

PURPOSE: Threshold an input image. If the image is above the threshold, output another image. If the image is below the threshold, output zero.

CALL SYNTAX: thmult(*labels, thresh, thd, imageout*)

INPUT PARAMETERS:

labels: The input image that is output if *thresh* is above the threshold value.

thresh: The input image being thresholded.

thd: The threshold value.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
procedure thmult(   labels      : in image float,
                    thresh      : in image float,
                    thd         : in float,
                    imageout    : out image float)

is
next begin

    if thresh > thd then
        imageout:=labels;
    else
        imageout:=0;
    end if;

end next;

end thmult;
```

PURPOSE: Average the grayvalues in a 2^n x 2^n window. n is defined in the procedure.

CALL SYNTAX: txav(*imagein, imageout*)

INPUT PARAMETERS:

imagein: The input image.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define(expn, 8)

procedure txav(     imagein    : in image array (eval(–(expn/2))..eval(expn/2–1),
                                      eval(–(expn/2))..eval(expn/2–1)) of byte
                                                                   border 0,
                    imageout   : out image byte)

is
    sum, c   : float;
    i, j     : integer;

next begin
    c := 1.0/eval((expn)*(expn));
    sum := 0.0;

    for i in eval(–expn/2)..eval(expn/2–1) loop
        for j in eval(–expn/2)..eval(expn/2–1) loop
            sum := sum + imagein(i, j);
        end loop;
    end loop;

    imageout := c*sum;

end next;

end txav;
```

PURPOSE: Compute the edge value of a texture image. The size of the edge is defined in this procedure.

CALL SYNTAX: txdf1(*imagein, hv, imageout*)

INPUT PARAMETERS:

imagein: The input image.

hv: If 0, compute the horizontal edge; if 1, compute the vertical edge.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define(n, 7)

procedure txdf1(   imagein    : in image array (0..eval(n–1), 0..eval(n–1)) of byte
                                                                      border 0,
                   hv         : in integer,
                   imageout   : out image byte)

is

    horz, vert : float;

next begin

    horz := abs(imagein(eval((n–1)/2), 0) – imagein(eval((n–1)/2), eval(n–1)));
    vert := abs(imagein(0, eval((n–1)/2)) – imagein(eval(n–1), eval((n–1)/2)));

    if  hv = 0  then
        imageout := integer(horz);
    else
        imageout := integer(vert);
    end if;

end next;

end txdf1;
```

PURPOSE: Compute the edge value of a texture image of specified size and direction. The maximum separation between points is defined in the procedure.

CALL SYNTAX: txdf2(*imagein, size, cosvalue, sinvalue, imageout*)

INPUT PARAMETERS:

imagein: The input image.

size: The size of the edge.

cosvalue: The cosine of the edge direction.

sinvalue: The sine of the edge direction.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define(n, 7)

procedure txdf2(   imagein    : in image array (eval(–n/2)..eval(n/2),
                                          eval(–n/2)..eval(n/2)) of byte
                                                                border 0,
                   size       : in integer,
                   cosvalue   : in float,
                   sinvalue   : in float,
                   imageout   : out image byte)

is

   c, s : float;
   sz   : integer;

next begin

   sz := size;

   if  sz > n  then
      sz := n;
   end if;

   c := sz*cosvalue/2;
   s := sz*sinvalue/2;

   imageout := abs(imagein(integer(–c), integer(–s))
                  –imagein(integer(c),  integer(s)));

end next;

end txdf2;
```

PURPOSE: Compute the best edge value and direction for texture images.

CALL SYNTAX: txeg2(*edgevalue, oldbestedgevalue, oldbestedgesize, oldbestedgedir, edgesize, edgedir, lambda, newbestedgevalue, newbestedgesize, newbestedgedir)*

INPUT PARAMETERS:

edgevalue: Input image of edge value.

oldbestedgevalue: Input image of best edge value

oldbestedgesize: Input image of best edge size.

oldbestedgedir: Input image of best edge direction.

edgesize: The edge size.

edgedir: The edge direction.

lambda: A constant used to compute the best edge size.

OUTPUT PARAMETERS:

newbestedgevalue: The output image edge value.

newbestedgesize: The output image edge size.

newbestedgedir: The output image edge direction.

PROCEDURE:

```
procedure txeg2(  edgevalue              : in image byte,
                  oldbestedgevalue       : in image byte,
                  oldbestedgesize        : in image byte,
                  oldbestedgedir         : in image byte,
                  edgesize               : in integer,
                  edgedir                : in float,
                  lambda                 : in float,
                  newbestedgevalue       : out image byte,
                  newbestedgesize        : out image byte,
                  newbestedgedir         : out image byte)
is
    newval, newsiz, newdir : float;
next begin
    newval := oldbestedgevalue;
    newsiz  := oldbestedgesize ;
    newdir   := oldbestedgedir  ;

    if  oldbestedgesize = edgesize  then
        if  edgevalue > oldbestedgevalue  then
            newval := edgevalue;
```

```
            newdir   := edgedir  ;
            newsiz  := edgesize ;
        end if;
    else
        if  edgesize > oldbestedgesize then
            if  edgevalue >= lambda * oldbestedgevalue  then
                newval := edgevalue;
                newdir   := edgedir  ;
                newsiz  := edgesize ;
            end if;
        else
            if  edgesize < oldbestedgesize  then
                if  oldbestedgevalue < lambda * edgevalue  then
                    newval := edgevalue;
                    newdir   := edgedir  ;
                    newsiz  := edgesize ;
                end if;
            end if;
        end if;
    end if;

    newbestedgevalue := newval;
    newbestedgesize  := newsiz;
    newbestedgedir   := newdir;

end next;

end txeg2;
```

PURPOSE: Perform a convolution in the x direction.

CALL SYNTAX: xconv(*imagein, xfilter, imageout*)

INPUT PARAMETERS:

imagein: The input image.

xfilter: The convolution filter.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define(MAXSIZE, 20)

procedure xconv(   imagein    : in image array (–MAXSIZE..MAXSIZE, 0..0) of float
                                                                   border 0,
                   xfilter    : in array (–MAXSIZE..MAXSIZE) of float,
                   imageout   : out image float)

is
    sum1, sum2: float;
    sum3, sum4: float;
    sum5, sum6: float;
    sum7, sum8: float;
    i         : integer;

next begin
    sum1 := 0.0;   sum2 := 0.0;   sum3 := 0.0;   sum4 := 0.0;
    sum5 := 0.0;   sum6 := 0.0;   sum7 := 0.0;   sum8 := 0.0;

    for i in 0..eval((MAXSIZE–1)/4) loop
        sum1 := sum1 + xfilter(4*i+1) * imagein(0, 4*i+1);
        sum2 := sum2 + xfilter(4*i+2) * imagein(0, 4*i+2);
        sum3 := sum3 + xfilter(4*i+3) * imagein(0, 4*i+3);
        sum4 := sum4 + xfilter(4*i+4) * imagein(0, 4*i+4);
        sum5 := sum5 + xfilter(–(4*i+1)) * imagein(0, –(4*i+1));
        sum6 := sum6 + xfilter(–(4*i+2)) * imagein(0, –(4*i+2));
        sum7 := sum7 + xfilter(–(4*i+3)) * imagein(0, –(4*i+3));
        sum8 := sum8 + xfilter(–(4*i+4)) * imagein(0, –(4*i+4));
    end loop;

    for i in eval(4*eval((MAXSIZE–1)/4)+5)..MAXSIZE loop
        sum1 := sum1 + xfilter(i) * imagein(0, i);
        sum2 := sum2 + xfilter(–i) * imagein(0, –i);
    end loop;

    imageout := sum1 + sum2 + sum3 + sum4 + sum5 + sum6
              + sum7 + sum8 + xfilter(0) * imagein(0,0);

end next;

end xconv;
```

PURPOSE: XOR an image with a constant.

CALL SYNTAX: xorc1b(*image1, constant, image2*)

INPUT PARAMETERS:

image1: The input image.

constant: An integer constant.

OUTPUT PARAMETERS:

image2: The output image.

PROCEDURE:

```
procedure xorc1b(   image1     : in image byte,
                    constant   : in integer,
                    image2     : out image byte)
is
next begin
    image2 := image1 lxor constant;
end next;
end xorc1b;
```

XORP1B

PURPOSE: XOR two images.

CALL SYNTAX: xorp1b(*image1, image2, image3*)

INPUT PARAMETERS:

image1: The first input image.

image2: The second input image.

OUTPUT PARAMETERS:

image3: The output image.

PROCEDURE:

```
procedure xorp1b(   image1     : in image byte,
                    image2     : in image byte
                    image3     : out image byte)
is
next begin
    image3 := image1 lxor image2;
end next;
end xorp1b;
```

PURPOSE: Perform a convolution in the y direction.

CALL SYNTAX: yconv(*imagein, yfilter, imageout*)

INPUT PARAMETERS:

imagein: The input image.

yfilter: The convolution filter.

OUTPUT PARAMETERS:

imageout: The output image.

PROCEDURE:

```
define(MAXSIZE, 20)

procedure yconv(   imagein    : in image array (–MAXSIZE..MAXSIZE, 0..0) of float
                                                                   border 0,
                   yfilter    : in array (–MAXSIZE..MAXSIZE) of float,
                   imageout   : out image float)

is
    sum1, sum2: float;
    sum3, sum4: float;
    sum5, sum6: float;
    sum7, sum8: float;
    i         : integer;

next begin
    sum1 := 0.0;   sum2 := 0.0;   sum3 := 0.0;   sum4 := 0.0;
    sum5 := 0.0;   sum6 := 0.0;   sum7 := 0.0;   sum8 := 0.0;

    for i in 0..eval((MAXSIZE–4)/4) loop
        sum1 := sum1 + yfilter(4*i+1) * imagein(4*i+1, 0);
        sum2 := sum2 + yfilter(4*i+2) * imagein(4*i+2, 0);
        sum3 := sum3 + yfilter(4*i+3) * imagein(4*i+3, 0);
        sum4 := sum4 + yfilter(4*i+4) * imagein(4*i+4, 0);
        sum5 := sum5 + yfilter(–(4*i+1)) * imagein(–(4*i+1), 0);
        sum6 := sum6 + yfilter(–(4*i+2)) * imagein(–(4*i+2), 0);
        sum7 := sum7 + yfilter(–(4*i+3)) * imagein(–(4*i+3), 0);
        sum8 := sum8 + yfilter(–(4*i+4)) * imagein(–(4*i+4), 0);
    end loop;

    for i in eval(4*eval((MAXSIZE–4)/4)+5)..MAXSIZE loop
        sum1 := sum1 + yfilter(i) * imagein(i, 0);
        sum2 := sum2 + yfilter(–i) * imagein(–i, 0);
    end loop;

    imageout := sum1 + sum2 + sum3 + sum4 + sum5 + sum6
                + sum7 + sum8 + yfilter(0) * imagein(0,0);

end next;

end yconv;
```