Se incluye a continuación un pseudocódigo por cada uno de los procedimientos que se deben implementar en esta práctica (se ha añadido el pseudocódigo de "*suavizado*" para que sirva de referencia sobre la sintaxis utilizada). Se indican a continuación algunos aspectos generales a tener en cuenta para la implementación:

- Los datos que aparecen en el pseudocódigo que no se corresponden con operandos del bloque de código en ensamblador (parámetros del procedimiento o variables locales) deben representarse a través de registros. Se recuerda que en la arquitectura de 64 bits hay 8 nuevos registros de propósito general que pueden utilizarse con distintos tamaños (%r8, %r9, …, %r15).
- El pseudocódigo no incluye definición de tipos para los nuevos datos que aparecen. No obstante, todos ellos pueden deducirse a partir de su uso. Por ejemplo, el dato *dirImagenO* toma el valor del parámetro *imagenO*. Puesto que dicho parámetro es un puntero, *dirImagenO* almacenará una dirección de memoria (64 bits) y, por lo tanto, debe representarse mediante un registro de 64 bytes.
- Debe respetarse el tipo de cada dato en la traducción a ensamblador para evitar errores de ejecución. Por ejemplo, en el procedimiento *umbralizar*, en la sentencia "*Si([dirImagenO + offsetPixel] <= umbral)*", los dos valores a comparar son de tipo *unsigned char*, por lo que la comparación debe hacerse sobre operandos de tamaño byte y el salto posterior debe tener en cuenta que los valores comparados no tienen signo.
- Los accesos a los elementos de los arrays se especifican en el pseudocódigo mediante su posición en bytes, tal y como debe hacerse en ensamblador. Por ejemplo, en el procedimiento *suavizado*, la sentencia *[dirImagenO + offsetPixel]*, siendo *offsetPixel = (y+j)*W + (x+i)*, equivale a un acceso al elemento *imagenO[y+j][x+i]*.
- Para accesos a elementos de arrays cuyo tipo ocupa más de 1 byte, se ha especificado también el factor de escala que debe aplicarse a la posición del elemento en el acceso a memoria. Por ejemplo, en el procedimiento *calculoGradientes*, el acceso a un elemento del array *gradientes* se lleva a cabo multiplicando la posición del elemento (offsetPixel) por 8, puesto que cada elemento ocupa 8 bytes (la estructura *Gradient* contiene 2 elementos de tipo *int*).
- El acceso a los atributos de las estructuras *Gradient* y *Point* están indicados en el pseudocódigo a través de los nombres de dichos atributos. No obstante, en ensamblador, el acceso a cada atributo debe hacerse indicando los desplazamientos dentro de la estructura asociados con cada campo. Por ejemplo, para acceder a los atributos *dx* y *dy* de la estructura *Gradient*, los desplazamientos que deben añadirse a la posición de memoria del elemento son 0 y 4, respectivamente.
- Los procedimientos *detectaContorno* y *contornoAPoligono* devuelven un valor como retorno de función. En ambos casos, se retorna el contenido de una variable local que está incluida en la lista de operandos de salida del bloque de código en ensamblador. Para darle un valor a esa variable, basta con asignar dicho valor al operando (%0 en el caso de *detectaContorno* o %edx en el caso de *contornoAPoligono*).
- En el procedimiento opcional (*contornoAPoligono*), los operandos del bloque se han indicado en registros concretos para facilitar la implementación recursiva. Concretamente, los registros asociados con cada operando son los siguiente: %edx = nP; %rsi = contour; %eax = nPoints; %rdi = polygon. Para acceder a estos operandos, basta con acceder a los registros correspondientes. Además, en el pseudocódigo de este procedimiento, se ha marcado con fondo azul la parte del procedimiento que debe resolverse mediante operaciones de punto flotante. Dicha implementación puede realizarse tanto con FPU como con SSE. Para poder utilizar cualquiera de las dos alternativas, el parámetro de tipo *float* "*distancia*" se encuentra almacenado tanto en la pila FPU como en el registro %xmm0.

```
void suavizado(unsigned char * imagenO, unsigned char * imagenD, int W,
int H, int * kernel, int N)
{
    dirImagenO = imagenO;
    dirImagenD = imagenD;
    Para(y=0; y<H; y++)
    {
        Para(x=0; x<W; x++)
        {
            dirKernel = kernel;
            acum = 0;
            totalK = 0;
            Para(j=-N; j<=N; j++)
            {
                Para(i=-N; i<=N; i++)
                {
                    Si((x+i)>=0 y (x+i)<W y (y+j)>=0 y (y+j)<H)
                    {
                        offsetPixel = (y + j) * W + (x + i);
                        acum += [dirImagenO + offsetPixel]*[dirKernel];
                        totalK += [dirKernel];
                    }
                    dirKernel += 4;
                }
            }
            offsetPixel = y*W + x;
            [dirImagenD + offsetPixel] = acum / totalK;
        }
    }
}
```

```
void umbralizar(unsigned char * imagenO, unsigned char * imagenD,
unsigned char umbral, int W, int H)
{

    dirImagenO = imagenO;
    dirImagenD = imagenD;
    Para(y=0; y<H; y++)
    {
        Para(x=0; x<W; x++)
        {
            offsetPixel = y*W + x;
            Si([dirImagenO + offsetPixel] <= umbral)
                [dirImagenD +  offsetPixel] = 0;
            Sino
                [dirImagenD +  offsetPixel] = 255;
        }
    }

}
```

```
void calculoGradientes(unsigned char* imagen, Gradient* gradientes, int W, int H)
{
    dirImagenO = imagenO;
    dirGradientes = gradientes;
    Para(y=0; y<H; y++)
    {
        Para(x=0; x<W; x++)
        {
            dirSobelX = sobel_x;
            dirSobelY = sobel_y;
            gradX = 0;
            gradY = 0
            Para(j=-1; j<=1; j++)
            {
                Para(i=-1; i<=1; i++)
                {
                    Si((x+i)>=0 y (x+i)<W y (y+j)>=0 y (y+j)<H)
                    {
                        offsetPixel = (y + j) * W + (x + i);
                        gradX += [dirImagenO + offsetPixel]*[dirSobelX];
                        gradY += [dirImagenO + offsetPixel]*[dirSobelY];
                    }
                    dirSobelX += 4;
                    dirSobelY += 4;
                }
            }
            offsetPixel = y*W + x;
            [dirGradientes + offsetPixel*8].dx = gradX;
            [dirGradientes + offsetPixel*8].dy = gradY;
        }
    }

}
```

```
void supresionNoMaximo(int * gradNorm, unsigned char * gradDir, unsigned
char* imagenD, int W, int H)
{
    dirGradNorm = gradNorm;
    dirGradDir = gradDir;
    dirImagenD = imagenD;
    Para(y=1; y<H-1; y++)
    {
        Para(x=1; x<W-1; x++)
        {
            offsetPixel = y*W + x;
            Si([dirGradNorm + offsetPixel*4]<=255)
                [dirImagenD+offsetPixel]=[dirGradNorm+offsetPixel*4];
            Sino
                [dirImagenD+offsetPixel]=255;
            Si([dirGradDir+offsetPixel]==0)
            {
                vOffset1 = offsetPixel-1;
                vOffset2 = offsetPixel+1;
            }
            Sino Si([dirGradDir+offsetPixel]==1)
            {
                vOffset1 = offsetPixel-W-1;
                vOffset2 = offsetPixel+W+1;
            }
            Sino Si([dirGradDir+offsetPixel]==2)
            {
                vOffset1 = offsetPixel-W;
                vOffset2 = offsetPixel+W;
            }
            Sino Si([dirGradDir+offsetPixel]==3)
            {
                vOffset1 = offsetPixel-W+1;
                vOffset2 = offsetPixel+W-1;
            }

            Si([dirGradNorm+offsetPixel*4]<=[dirGradNorm+vOffset1*4]
               o [dirGradNorm+offsetPixel*4]<[dirGradNorm+vOffset2*4])
                [dirImagenD+offsetPixel]=0;
        }
    }

}
```

```
void dobleUmbralizacion(unsigned char* imagenO, unsigned char* imagenD,
unsigned char umbralMin, unsigned char umbralMax, int W, int H)
{
    dirImagenO = imagenO;
    dirImagenD = imagenD;
    Para(y=1; y<H-1; y++)
    {
        Para(x=1; x<W-1; x++)
        {
            offsetPixel = y*W + x;
            Si([dirImagenO+offsetPixel] >= umbralMax)
            {
                [dirImagenD+offsetPixel] = 255;
            }
            Sino
            {
                esBorde = Falso;
                Si([dirImagenO+offsetPixel] >= umbralMin)
                {
                    Para(j=-1; j<=1 y no esBorde; j++)
                    {
                        Para(i=-1; i<=1 y no esBorde; i++)
                        {
                            vOffset = (y+j)*W + (x+i);
                            Si([dirImagenO+vOffset] >= umbralMax o
                               [dirImagenD+vOffset] == 255)
                            {
                                esBorde = Cierto;
                            }
                        }
                    }

                }
                Si(esBorde)
                    [dirImagenD+offsetPixel] = 255;
                Sino
                    [dirImagenD+offsetPixel] = 0;
            }

        }
    }

}
```

```
int detectaContorno(unsigned char * imagen, unsigned char * visitados,
int ini_x, int ini_y, Point direcciones[8], Point contorno[5000], int W,
int H)
{
    // Modifica la variable local nPuntos

    dirImagen = imagen;
    dirVisitados = visitados;
    dirContorno = contorno;
    dirDirecciones = direcciones

    sig_x = ini_x;
    sig_y = ini_y;
    iContorno = 0;
    hayPunto = cierto;

    Mientras(hayPunto y iContorno<4999)
    {
        [dirContorno + iContorno*8].x = sig_x;
        [dirContorno + iContorno*8].y = sig_y;
        offsetPixel = sig_y*W + sigx;
        [dirVisitados + offsetPixel] = 255;
        iContorno++;

        iDir = 0;
        hayPunto = falso;
        Mientras(iDir<8 y no hayPunto)
        {
            vec_x = sig_x + [dirDirecciones+iDir*8].x;
            vec_y = sig_y + [dirDirecciones+iDir*8].y;
            Si(vec_x>=0 y vec_x<W y vec_y>=0 y vec_y<H)
            {
                offset_vec = vec_y*W + vec_x;
                Si([dirVisitados + offset_vec]==0 y
                   [dirImagen + offset_vec]==255)
                {
                    hayPunto = cierto;
                    sig_x = vec_x;
                    sig_y = vec_y;
                }
            }
            iDir++;
        }
    }
    nPuntos = iContorno;
}
```

```
int contornoAPoligono(Point contorno[5000], int nPuntos, Point
poligono[5000], float distancia)
{
    // Modifica la variable local nP
    dirContorno = contorno;
    dirPoligono = poligono;
    nP = contornoAPoligonoRecursivo(dirContorno, nPuntos, dirPoligono,
                                    distancia);

    int contornoAPoligonoRecursivo(dirContorno, nPuntos, dirPoligono,
                                   distancia)
    {
        iF = nPuntos-1;
        // a, b, c, dAct y dMax deben tratarse con tipo float
        // Puede usarse tanto FPU como SSE
        a = [dirContorno + iF*8].y - [dirContorno].y;
        b = [dirContorno].x - [dirContorno + IF*8].x;
        c = -a*[dirContorno].x - b*[dirContorno].y;
        norm = sqrt(a*a + b*b);
        a = a/norm;
        b = b/norm;
        c = c/norm;
        dMax = 0;
        iCorte = 0;
        Para(i=0; i<nPuntos-1; i++)
        {
            dAct = abs(a*[dirContorno+i*8].x+b*[dirContorno+i*8].y+c);
            Si(dAct>dMax)
            {
                dMax = dAct;
                iCorte = i;
            }
        }
        Si(dMax>distancia)
        {
            nP1 = contornoAPoligonoRecursivo(dirContorno, iCorte+1,
                dirPoligono, distancia);
            nP2 = contornoAPoligonoRecursivo(&[dirContorno+iCorte*8],
                nPuntos-iCorte, &[dirPoligono+(nP1-1)*8], distancia);
            nP = np1+np2-1;
        }
        Sino
        {
            [dirPoligono].x = [dirContorno].x;
            [dirPoligono].y = [dirContorno].y;
            [dirPoligono+8].x = [dirContorno+iF*8].x;
            [dirPoligono+8].y = [dirContorno+iF*8].y;
            nP = 2;
        }
        devolver nP;
    }
}
```