Visión por Computador (2016-2017)
Grado en Ingeniería Informática
Universidad de Granada

Informe Práctica 1: opción de 8 puntos

Laura Tirado López

20 de octubre de 2016

Índice

1. La convolución. **3**
1.1. Vector máscara 3
1.2. Convolución de un vector señal 1D 4
1.3. Convolución de una imagen 2D 6
1.4. Resultados 8

2. Imágenes híbridas **8**
2.1. Resultados 12

3. Pirámide Gaussiana **12**
3.1. Resultados 15

Índice de figuras

1.1. Convolución con valores de sigma 3.0 y 10.0 8
2.1. Imagen híbrida de un pájaro y un avión junto con sus frecuencias bajas (izquierda) y altas (derecha) 12
3.1. Pirámide Gaussiana 15

1. La convolución.

1.1. Vector máscara

Lo primero que tenemos que hacer es calcular la máscara. Dado que estamos muestreando una Gaussiana, que es separable y simétrica, vamos a construir una máscara 1D que utilizaremos tanto para las filas como para las columnas.

El tamaño de la máscara vendrá en función de σ y será el siguiente: $round(3\sigma)2+1$. Esto se debe a que vamos a muestrear dentro del intervalo $[-3\sigma, 3\sigma]$ y ponemos redondeamos para poder tener máscaras impares de cualquier tamaño. De este modo obtenemos máscaras de orden impar.

Lo siguiente que tenemos que calcular es el paso de muestreo. Como queremos que el mayor peso recaiga sobre el píxel central el tamaño del paso será $\frac{6\sigma}{longitud-1}$. De esta manera nos aseguramos de muestrear en los extremos del intervalo que hemos señalado antes.

Por último, muestreamos la Gaussiana de párametro σ y dividimos cada componente por la suma de todos para que la máscara sume uno, una condición de los filtros de alisamiento.

```
1  Mat calcularVectorMascara(float sigma)) {
2
3          //Calculamos la longitud de la mascara
4          int longitud = round(3*sigma)*2+1;
5          int centro = (longitud-1)/2; //elemento del centro del
               vector
6
7          //Calculamos el tamanio del paso
8          float paso=6*sigma/(longitud-1);
9
10         //Creamos la imagen que contendra los valores muestreados
11         Mat mascara = Mat(1,longitud,CV_32F);
12
13         //Cargamos los valores en la mascara
14         for(int i=0; i <=centro; ++i){
15                 mascara.at<float>(0,i) = exp(-0.5*(-paso*(centro
                       -1))*(-paso*(centro-1))/(sigma*sigma));
16                 mascara.at<float>(0,longitud-i-1) = exp(-0.5*(paso
                       *(centro-i))*(paso*(centro-i))/(sigma*sigma));
17         }
```

```
18
19          //Dividimos por la suma de todos para que los elementos
                sumen 1
20          float suma = 0.0;
21
22          for(int i=0; i < mascara.cols; ++i)
23                  suma += mascara.at<float>(0,i);
24
25          mascara = mascara /suma;
26
27          return mascara;
28  }
```

1.2. Convolución de un vector señal 1D

Antes de calcular la convolución del vector señal vamos a obtener el vector orlado, es decir, vamos a preparar el vector señal para poder realizar sobre él la convolución de forma correcta añadiendo a sus extremos tantos píxeles como sean necesarios.

Concretamente, si nos situamos en los extremos de la fila/columna a convolucionar sobrarán los píxeles de la máscara a uno de los lados del píxel central, con lo cual la cantidad de píxeles a añadir es mascara.cols - 1.

Una vez orlada la fila/columna a convolucionar con esta cantidad de píxeles (la mitad a cada lado) no tenemos más que rellenarlos o bien poniéndolos a cero o en modo espejo (reflejado), según elijamos con los parámetros.

```
1   Mat calcularVectorOrlado(const Mat &senal, Mat &mascara, int
        cond_contorno){
2
3           //Aniadimos a cada lado del vector (longitud_senal -1)/2
                pixeles, porque es el maximo numero de pixeles que
                sobrarian
4           //al situar la mascara en la esquina.
5           Mat copia_senal;
6
7           //Trabajamos con vectores fila
8           if(senal.rows == 1)
9                   copia_senal = senal;
10          else if(senal.cols == 1)
11                  copia_senal = senal.t();
12          else
13                  cout << "El vector senal no es vector fila o
                        columna." << endl;
```

```
14
15          int pixel_copia = copia_senal.cols;
16          int pixel_extra = mascara.cols-1; //numero de pixeles
               necesarios para orlar
17          int cols_vector_orlado = pixel_copia + pixel_extra;
18
19          Mat vectorOrlado = Mat(1,cols_vector_orlado, senal.type())
               ;
20
21          int ini_copia, fin_copia; //posiciones donde comienza la
               copia del vector, centrada
22
23          ini_copia = pixel_extra/2;
24          fin_copia = pixel_copia+ini_copia;
25
26          //Copiamos senal centrado en vectorAuxiliar
27          for(int i=ini_copia; i < fin_copia; ++i)
28                  vectorOrlado.at<float>(0,i) = copia_senal.at<float
                       >(0,i-ini_copia);
29
30          //Ahora rellenamos los vectores de orlado
31          for(int i=0; i < ini_copia; ++i){
32
33                  vectorOrlado.at<float>(0,ini_copia-i-1) =
                       cond_contorno*vectorOrlado.at<float>(0,
                       ini_copia+i);
34                  vectorOrlado.at<float>(0,fin_copia+i) =
                       cond_contorno * vectorOrlado.at<float>(0,
                       fin_copia-i-1);
35          }
36
37          return vectorOrlado;
38  }
```

Una vez hemos calculado el vector orlado podemos calcular la convolución correctamente.

Para el caso de trabajar con imágenes a color utilizamos las funciones *split* y *merge* de OpenCV, las cuales nos permiten obtener por separado cada uno de los canales de la imagen y luego juntar varios canales en una sola imagen respectivamente.

Con el objetivo de poder reutilizar código, implementamos una función para calcular la convolución para imágenes con un solo canal. De esta forma si tenemos una imagen con tres canales, procesamos cada uno por separado con la función para un solo canal y luego las unimos con la función *merge*.

Para calcular la convolución del vector lo único que hacemos es aplicar la operación de convolución a una matriz 1D. Teniendo en cuenta que es un vector orlado, hacemos la convolución sólo en los píxeles que no son de la orla, aprovechando la función colRange

para fijar un ROI en cada paso.

La máscara está preparada para trabajar con vectores fila porque de esta forma simplemente con trasponer podemos trabajar también con las columnas con las columnas como si fuesen filas.

```
Mat calcularConvolucion1D1C(const Mat &senal, Mat &mascara, int
    cond_contorno){

        //Orlamos el vector para prepararlo para la convolucion
        Mat copiaOrlada = obtenerVectorOrlado(senal, mascara,
            cond_contorno);
        Mat segmentoCopiaOrlada;
        Mat convolucion = Mat(1,senal.cols, senal.type());

        int ini_copia, fin_copia, long_lado_orla;
        //Calculamos el rango de pixeles a los que tenemos que
            aplicar la convolucion, excluyen los vectores de orla
        ini_copia = (mascara.cols-1)/2;
        fin_copia = ini_copia + senal.cols;
        long_lado_orla = (mascara.cols-1)/2;

        for(int i=ini_copia; i < fin_copia; ++i){
                //Aplicamos la convolucion a cada pixel
                    seleccionado el segmento con el que
                    convolucionamos
                segmentoCopiaOrlada = copiaOrlada.colRange(i-
                    long_lado_orla, i+long_lado_orla+1);
                convolucion.at<float>(0,i-ini_copia) = mascara.dot
                    (segmentoCopiaOrlada);
        }

        return convolucion;
}
```

1.3. Convolución de una imagen 2D

Para el calcular la convolución de una imagen 2D reutilizamos las funciones anteriores.

De nuevo, implementamos una función para imágenes con un canal para hacer el código más reutilizable.

```
Mat calcularConvolucion2D1C(Mat &imagen, float sigma, int
    cond_bordes){

```

```
3           //Calculamos el vector mascara
4           Mat mascara = calcularVectorMascara(sigma);
5           Mat convolucion = Mat(imagen.rows, imagen.cols, imagen.
                type());
6
7           //Convolucion por filas
8           for(int i=0; i < imagen.rows; ++i)
9                   calcularConvolucion1D1C(imagen.row(i),mascara,
                        cond_bordes).copyTo(convolucion.row(i));
10
11          //Convolucion por columnas
12          convolucion = convolucion.t();//Trasponemos para poder
                operar como si fuesen filas
13
14          for(int i=0; i < convolucion.rows; ++i)
15                  calcularConvolucion1D1C(convolucion.row(i),mascara
                        ,cond_bordes).copyTo(convolucion.row(i));
16
17          convolucion = convolucion.t();//Deshacemos la trasposicion
18
19          return convolucion;
20  }
```

```
1   Mat calcularConvolucion2D(Mat &imagen, float sigma, int
        cond_bordes){
2
3           Mat convolucion;
4           Mat canales[3];
5           Mat canalesConvolucion[2];
6
7           //Si la imagen es 1C
8           if(imagen.channels() == 1)
9                   return calcularConvolucion2D1C(imagen, sigma,
                        cond_bordes);
10
11          //Si la imagen es 3C
12          else if (imagen.channels() == 3){
13
14                  split(imagen,canales);
15
16                  for(int i=0; i < 3; ++i)
17                          canalesConvolucion[i] =
                                calcularConvolucion2D1C(canales[i],
                                sigma,cond_bordes);
18
19                  merge(canalesConvolucion,3,convolucion);
20          }
21
```

```
22          else
23                  cout << "Numero de canales no valido. " << endl;
24
25          return convolucion;
26  }
```

1.4. Resultados

La siguiente imagen muestra los resultados de aplicar la convolución a la imagen con distintos valores de sigma. A la izquierda se encuentra la imagen original y a la derecha las imágenes resultado con los valores de sigma 3.0 y 10.0 respectivamente.

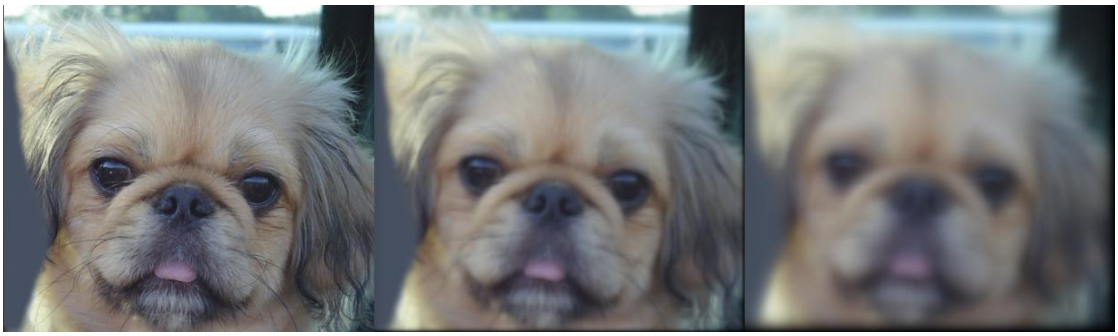

Figura 1.1: Convolución con valores de sigma 3.0 y 10.0

Como podemos ver cuánto más grande es el valor sigma más se difumina la imagen. Esto se debe a que cada vez influyen más píxeles (cada vez más alejados del central) en el valor de uno.

2. Imágenes híbridas

Para obtener una imagen híbrida lo único que tenemos que hacer es obtener las frecuencias bajas y altas de una imagen. Para calcular las frecuencias bajas necesitamos aplicar un filtro de alisamiento utilizando las funciones de convolución del apartado anterior y para obtener las frecuencias altas de una imagen le restamos a la imagen original sus frecuencias bajas.

```
1   Mat calcularHibrida(Mat &imagen1, Mat &imagen2, float sigma1,
        float sigma2, Mat &bajas_frec, Mat &altas_frec){
2
```

```
        bajas_frec = calcularConvolucion2D(imagen1,sigma1,0);
        altas_frec = imagen2 - calcularConvolucion2D(imagen2,
            sigma2,0);

        return bajas_frec + altas_frec;
}
```

Como se menciona en el guión de prácticas al obtener la imagen híbrida y la de altas frecuencias estas pueden tomar valores negativos, con lo cual si lo casteásemos directamente a 8U estos valores se truncarían a cero no observándose correctamente las altas frecuencias. Lo que hemos hecho para evitar tal efecto es reajustar el rango de la valores de la imagen para ponerlo dentro del intervalo [0,255].

```
Mat reajustarRango(Mat imagen){

        Mat canales_imagen[3];
        Mat imagen_ajustada;
        Mat canales_ajustada[3];

        //Si la imagen es 1C
        if(imagen.channels() == 1){

                float min = 0;
                float max = 255;

                //Calculamos el rango en el que se mueven los
                    valores de la imagen
                for(int i=0; i < imagen.rows; ++i){
                        for(int j=0; j < imagen.cols; ++j){
                                if(imagen.at<float>(i,j) < min)
                                    min = imagen.at<float>(i,j);
                                if(imagen.at<float>(i,j) > max)
                                    max = imagen.at<float>(i,j);
                        }
                }

                imagen.copyTo(imagen_ajustada);

                for(int i=0; i < imagen_ajustada.rows; ++i)
                        for(int j=0; j < imagen_ajustada.cols; ++j
                            )
                                imagen_ajustada.at<float>(i,j) =
                                    1.0*(imagen_ajustada.at<float>(
                                    i,j)-min)/(max-min)*255.0;


        }

```

```
30          //Si la imagen es 3C
31          else if(imagen.channels() == 3){
32                  split(imagen,canales_imagen);
33                  for(int i=0; i < 3; ++i)
34                          canales_ajustada[i] = reajustarRango(
                               canales_imagen[i]);
35                  merge(canales_ajustada,3,imagen_ajustada);
36          }
37          else
38                  cout << "Numero de canales no valido." << endl;
39
40          return imagen_ajustada;
41  }
```

Para poder mostrar las tres imágenes (altas frecuencias, bajas frecuencias e híbridas) he implementado una función auxiliar genérica para mostrar varias imágenes en una sola ventana y otra que solo muestra una imagen.

```
1   void mostrarImagen(string nombreVentana, Mat &imagen, int
        tipoVentana = 1){
2
3           //Comprobamos que la imagen no este vacia
4           if(imagen.data){
5                   namedWindow(nombreVentana,tipoVentana);
6                   //Mostramos la imagen con la funcion imshow
7                   imshow(nombreVentana,imagen);
8           }
9
10          else
11                  cout << "La imagen no se cargo correctamente." <<
                        endl;
12  }
13
14  void mostrarImagenes(string nombreVentana, vector<Mat> &imagenes){
15
16          //Primero calculamos el total de filas y columnas para la
                imagen que sera la union de todas las imagenes que
                queramos mostrar
17          int colCollage = 0;
18          int filCollage = 0;
19          int numColumnas = 0;
20
21          for(int i=0; i < imagenes.size(); ++i){
22                  //Cambiamos la codificacion del color de algunas
                        imagenes para evitar fallos al hacer el collage
23                  if (imagenes[i].channels() < 3) cvtColor(imagenes[
                        i], imagenes[i], CV_GRAY2RGB);
24                  //Sumamos las columnas
```

```
25                  colCollage += imagenes[i].cols;
26                  //Calculamos el maximo numero de filas necesarias
27                  if (imagenes[i].rows > filCollage) filCollage =
                        imagenes[i].rows;
28          }
29
30          //Creamos la imagen con las dimensiones calculadas y todos
                los pixeles a 0
31          Mat collage = Mat::zeros(filCollage, colCollage, CV_8UC3);
32
33          Rect roi;
34          Mat imroi;
35
36          //Unimos todas las imagenes
37          for(int i=0; i < imagenes.size(); ++i){
38
39                  roi = Rect(numColumnas, 0, imagenes[i].cols,
                        imagenes[i].rows);
40                  numColumnas += imagenes[i].cols;
41                  imroi = Mat(collage,roi);
42
43                  imagenes[i].copyTo(imroi);
44
45          }
46
47          //Mostramos la imagen resultante
48          mostrarImagen(nombreVentana,collage);
49
50  }
```

A la hora de mostrar las tres imágenes reajustamos el rango y hacemos las conversiones correspondientes dependiendo de si la imagen tiene un canal o tres canales.

```
1   void mostrarHibrida(Mat &imagen_hibrida, Mat &bajas_frec, Mat &
        altas_frec,string nombreVentana){
2
3           //Reajustamos el rango de las imagenes
4           imagen_hibrida = reajustarRango(imagen_hibrida);
5           altas_frec = reajustarRango(altas_frec);
6
7           //Hacemos la conversion para mostrar las imagenes
8           if(imagen_hibrida.channels()==3){
9
10                  imagen_hibrida.convertTo(imagen_hibrida,CV_8UC3);
11                  altas_frec.convertTo(altas_frec,CV_8UC3);
12                  bajas_frec.convertTo(bajas_frec,CV_8UC3);
13          }
14
```

```
15          else if(imagen_hibrida.channels() == 1){
16
17                  imagen_hibrida.convertTo(imagen_hibrida,CV_8U);
18                  altas_frec.convertTo(altas_frec,CV_8U);
19                  bajas_frec.convertTo(bajas_frec,CV_8U);
20          }
21
22          else
23                  cout << "Numero de canales no valido." << endl;
24
25          vector<Mat> imagenes;
26
27          imagenes.push_back(altas_frec);
28          imagenes.push_back(imagen_hibrida);
29          imagenes.push_back(bajas_frec);
30
31          mostrarImagenes(nombreVentana, imagenes);
32  }
```

2.1. Resultados

Figura 2.1: Imagen híbrida de un pájaro y un avión junto con sus frecuencias bajas (izquierda) y altas (derecha)

Para cada pareja de imágenes tendremos que ajustar los valores de sigma de modo que el efecto de la imagen híbrida sea el mejor posible. Para el avión y el pájaro los valores de sigma que he utilizado son 19.0 y 1.0 respectivamente.

3. Pirámide Gaussiana

Para construir la pirámide Gaussiana partimos de la imagen original como primer nivel. Para obtener el siguiente nivel, alisamos la imagen del nivel anterior y la submuestreamos,

es decir, tomamos sólo las columnas y filas impares.

```
Mat submuestrear1C(const Mat &imagen){

        Mat submuestreado = Mat(imagen.rows/2, imagen.cols/2,
           imagen.type());

        for(int i=0; i < submuestreado.rows; ++i)
                for(int j=0; j < submuestreado.cols; ++j)
                        submuestreado.at<float>(i,j) = imagen.at<
                           float>(i*2+1,j*2+1);

        return submuestreado;
}
```

Tenemos que tener en cuenta el valor de sigma para el filtro de alisamiento. Dado que vamos saltanto las columnas pares no tiene sentido tener en cuenta lo que pase a más allá de un píxel de distancia del central. Por lo tanto vamos a tomar una máscara de tamaño tres y, en consecuencia, tomamos un σ menor que 1, mayor o menor en función del peso que queramos darle a cada uno de los píxeles que intervendrán en la convolución.

```
void calcularPiramideGauss(Mat &imagen, vector<Mat> &piramide, int
    niveles){

        Mat canales_imagen[3];
        Mat canales_nivel[3];
        vector<Mat> canales_piramide[3];

        //Reajustamos el rango de la imagen
        imagen = reajustarRango(imagen);

        //Si la imagen es 1C
        if(imagen.channels() == 1){

                piramide.push_back(imagen);

                for(int i=0; i < niveles-1; ++i){
                        piramide.push_back(submuestrear1C(
                           calcularConvolucion2D1C(piramide.at(i)
                           ,1.5,0)));
                }

        }

        //Si la imagen es 3C
        else if(imagen.channels() == 3){

```

```
24                  piramide.resize(niveles);
25                  split(imagen,canales_imagen);
26
27                  for(int i=0; i < 3; ++i)
28                          calcularPiramideGauss(canales_imagen[i],
                               canales_piramide[i],niveles);
29
30                  for(int i=0; i < niveles; ++i){
31                          for(int j=0; j < 3; ++j)
32                                  canales_nivel[j] =
                                       canales_piramide[j].at(i);
33
34                          merge(canales_nivel,3,piramide.at(i));
35                  }
36          }
37
38          else
39                  cout << "Numero de canales no valido." << endl;
40  }
```

Para mostrar la pirámide he implementado una función del mismo modo que en el apartado anterior. En este caso no he necesitado reajustar el rango de las imágenes:

```
1   void mostrarPiramide(vector<Mat> piramide, string nombreVentana){
2
3           if(piramide.at(0).channels() == 3){
4                   for(int i=0; i < piramide.size(); ++i)
5                           piramide.at(i).convertTo(piramide.at(i),
                                CV_8UC3);
6           }
7
8           else if(piramide.at(0).channels() == 1){
9                   for(int i=0; i < piramide.size(); ++i)
10                          piramide.at(i).convertTo(piramide.at(i),
                                CV_8U);
11          }
12
13          else
14                  cout << "Numero de canales no valido." << endl;
15
16          mostrarImagenes(nombreVentana,piramide);
17  }
```

3.1. Resultados

Figura 3.1: Pirámide Gaussiana

En esta imagen podemos ver el efecto que se busca en las imágenes híbridas. Considero que he conseguido un buen resultado tras probar con distintos valores de sigma para hacer que el pájaro desapareciese lo suficiente y las frecuencias altas del avión no fuesen demasiado marcadas de forma que se viese bien en los niveles superiores de la pirámide pero se perdiese en niveles inferiores.