Informe de prácticas
1

Alejandro García Montoro
agarciamontoro@correo.ugr.es

28 de octubre de 2015

1. Análisis de la implementación

1.1. Función de convolución

La implementación de la función de convolución 2D se construye sobre funciones más simples; a saber:

- Cálculo de una máscara gaussiana 1D.
- Convolución 1D con una máscara general.

Sobre la convolución 2D se construye, además, la función lowPassFilter(), que implementa un filtro Gaussiano.

Cálculo de una máscara gaussiana 1D

La máscara gaussiana se construye con la función getGaussMask(), cuyo código se puede ver más abajo. Esta función devuelve un objeto Mat que representa un vector uni-dimensional generado a partir de la discretización de la función gaussiana en el intervalo $[-3\sigma, 3\sigma]$.

El tamaño de la máscara tiene que ser un número natural impar. Así, primero se redondea el resultado de triplicar el σ, luego se duplica lo anterior y por último se le añade uno.

Además, las máscaras de convolución tienen una restricción: la suma de sus valores tiene que ser igual a uno. Por tanto, primero se genera la máscara discretizando la función sin atender a esta restricción y luego se *normaliza*; esto es, se divide cada valor de la máscara por la suma de todos ellos.

```
1       /**
2        * Builds a gaussian mask given the parameter sigma. The
             gaussian function is
3        * sampled in the interval [-3*sigma, 3*sigma].
4        */
5       Mat Image::getGaussMask(double sigma){
```

```
6           // Gaussian function needs to be sampled between -3*sigma
                and +3*sigma
7           // and the mask has to have an odd dimension.
8           int mask_size = 2*round(3*sigma) + 1;
9
10          Mat gauss_mask = Mat(1,mask_size,CV_32FC1);
11
12          // It is necessary to normalize the mask, so the sum of
                its elements
13          // needs to be saved.
14          float values_sum = 0;
15
16          // Fills the mask with a sampled gaussian function and
                saves the sum of all elements
17          for (int i = 0; i < mask_size; i++) {
18              gauss_mask.at<float>(0,i) =
                    gaussianFunction(i-mask_size/2, sigma);
19              values_sum += gauss_mask.at<float>(0,i);
20          }
21
22          // Normalizes the gauss mask.
23          gauss_mask = gauss_mask / values_sum;
24
25          return gauss_mask;
26      }
```

Los valores de la función gaussiana se consiguen con la siguiente implementación, que da el valor de la función en un punto x con parámetro σ.

```
1       /**
2        * Samples the 1D gaussian function at point x with parameter
             sigma
3        */
4       double Image::gaussianFunction(double x, double sigma){
5           return exp(-0.5*(x*x)/(sigma*sigma));
6       }
```

Convolución 1D con una máscara general

El siguiente paso es la implementación de una convolución uni-dimensional con una máscara general. La función devuelve un objeto Mat, que representa el resultado de hacer la convolución del vector uni-dimensional signal_vec con la máscara mask. El código es el siguiente:

```
1       /**
2        * Returns the result of convolving the uni-dimensional
             signal_vec with the
```

```
3         * mask, applying one of two types of borders: REFLECT or
              ZEROS.
4         */
5        Mat Image::convolution1D(const Mat& signal_vec, const Mat&
             mask, enum border_id border_type){
6            assert(signal_vec.rows == 1 && mask.rows == 1 && mask.cols
                 < signal_vec.cols);
7
8            int num_channels = signal_vec.channels();
9
10           // Initialization of source vector with additional borders.
11           int border_size = mask.cols/2; // Number of pixels added
                 to each side
12           Mat bordered;
13           copyMakeBorder(signal_vec,bordered,0,0,border_size,border_size,border_type,0.0);
14
15           // Splitting of the bordered vector for making a
                 per-channel processing
16           vector<Mat> bordered_channels(num_channels);
17           split(bordered, bordered_channels);
18
19           // Declaration of the result vector -with same size and
                 type as the
20           // original signal vector- and its splitted channels.
21           Mat result = Mat(signal_vec.size(), signal_vec.type());
22           vector<Mat> result_channels(num_channels);
23           split(result, result_channels);
24
25           // The mask and the source/result channels need to have
                 the same type.
26           // They are all converted to CV_32FC1 in order not to lose
                 precision.
27           Mat converted_mask;
28           mask.convertTo(converted_mask,CV_32FC1);
29
30           // Per-channel processing: we need the source channels,
                 the masked channels;
31           // i.e., the source channel focused in a ROI of the same
                 size as the mask
32           // and the result channels.
33           Mat source_channel, masked_channel, result_channel;
34
35           for (int i = 0; i < num_channels; i++) {
36               // Channel type conversion
37               bordered_channels[i].convertTo(source_channel,
                     CV_32FC1);
38               result_channels[i].convertTo(result_channel, CV_32FC1);
39
40               // Actual processing
```

```
41              for (int j = 0; j < result.cols; j++) {
42                  // We focus on the zone centered at j+mask.cols/2
                        with mask width
43                  masked_channel =
                        source_channel(Rect(j,0,mask.cols,1));
44
45                  // Scalar product between the ROI'd source and the
                        mask
46                  result_channel.at<float>(0,j) =
                        masked_channel.dot(converted_mask);
47              }
48
49              // Backwards conversion: the result should have the
                    same type as the input image
50              result_channel.convertTo(result_channels[i],result_channels[i].type());
51          }
52
53          // Merging again the processed channels
54          merge(result_channels, result);
55
56          return result;
57      }
```

El tener que tratar todos los canales de la imagen por separado y después hacer de nuevo la unión hace el código algo más difícil de leer, pero la idea es sencilla:

1. Se genera un vector señal con bordes —replicados o a ceros, según la decisión del usuario—. El tamaño de estos bordes es igual a la mitad del tamaño de la máscara menos uno.

2. A cada píxel j del vector resultado —que tiene tamaño igual al vector señal— se le asigna el resultado del producto escalar de la máscara con la zona apropiada del vector señal; es decir, con una zona del vector señal de tamaño igual a la máscara y centrada en el píxel $j + mask.cols/2$.

Este sencillo algoritmo hay que hacerlo para cada canal, con lo que antes del procesamiento hay que separar el vector señal y el vector resultado y después del procesamiento unir los canales del resultado. Todo este trabajo se hace con las funciones split() y merge() de OpenCV.

Para la generación del vector señal con bordes se ha usado la función copyMakeBorder(), que hace justo lo que se necesita: generar una matriz con los bordes especificados y del tipo que se deseen. Como su penúltimo argumento recibe una constante de OpenCV especificando qué tipo de borde se desea, en esta implementación que permite sólo dos tipos se ha decidido declarar el siguiente tipo de dato, que es el que recibe la función implementada:

```
enum border_id{
    REFLECT = cv::BORDER_REFLECT,
    ZEROS = cv::BORDER_CONSTANT
};
```

Convolución 2D con una máscara general

Esta función simplemente aplica, por filas y columnas, la convolución 1D con la máscara uni-dimensional especificada. Así, devuelve un objeto Mat que representa el resultado de hacer la convolución 2D con la máscara bi-dimensional resultado de hacer el producto matricial de la máscara uni-dimensional consigo misma. El código es el siguiente:

```
/**
 * Returns the result of convolving the two-dimensional signal_vec with a
 * uni-dimensional mask, applied in both rows and columns with one of two types
 * of borders: REFLECT or ZEROS.
 */
Mat Image::convolution2D(const Mat& signal_mat, const Mat& mask, enum border_id border_type){
    Mat result = Mat(signal_mat.size(), signal_mat.type());

    // Row-processing: the uni-dimensional mask is applied to each row separately
    Mat result_row;
    for (int i = 0; i < result.rows; i++) {
        // Row i is replaced with its convolution
        convolution1D(this->image.row(i), mask, border_type).copyTo(result.row(i));
    }

    // Column-processing: the same uni-dimensional mask is applied to each column
    // separately
    Mat transposed_col;
    for (int j = 0; j < result.cols; j++) {
        // Column i is replaced with its convolution ---needs transposing, as convolution1D
        // works with row vectors---.
        transpose(result.col(j),transposed_col);
        transpose(convolution1D(transposed_col, mask, border_type), result.col(j));
    }

    // Returns the convoluted image
```

```
27          return result;
28      }
```

El código es claro. Lo único que hay que destacar es que para hacer la convolución 1D por columnas lo que se hace es, para cada columna:

1. Trasponer la columna.
2. Aplicar la convolución 1D con la función anterior.
3. Trasponer el resultado e insertarlo en el objeto Mat que será devuelto.

Filtro gaussiano

Con las funciones anteriores es entonces directo implementar el filtro gaussiano. Basta calcular la máscara gaussiana dado un σ y hacer la convolución 2D de la imagen original y la máscara. El código es el siguiente:

```
1       /*
2        * Returns the result of applying a 2D convolution with a
             gaussian mask that is
3        * built with the parameter sigma.
4        */
5       Image Image::lowPassFilter(double sigma){
6           Mat gaussMask = getGaussMask(sigma);
7
8           Mat result = convolution2D(this->image, gaussMask,
                REFLECT);
9
10          return Image(result);
11      }
```

1.2. Imágenes híbridas

Una imagen híbrida no es más que el resultado de sumar una imagen a la que se ha aplicado un filtro de paso bajo con otra a la que se le ha aplicado un filtro de paso alto.

La implementación del filtro de paso bajo es la anterior, así que sólo falta especificar la implementación del filtro de paso alto y de la generación de la imagen híbrida.

Filtro de paso alto

El filtro alto implementado es sencillo: se hace la diferencia entre la imagen original y una versión de ella misma a la que se le ha aplicado el filtro de paso bajo. Así, quedan las altas frecuencias de la imagen, que es lo que necesitamos. El código es el siguiente:

```
1       /*
2        * Returns the result of subtracting the low-pass-filtered
             source to the source
3        * itself, remaining the high frequencies.
4        */
5       Image Image::highPassFilter(double sigma){
6           return *this - this->lowPassFilter(sigma);
7       }
```

1.2.1. Generación de la imagen híbrida

La implementación escogida, siguiendo el paradigma de orientación a objetos que se seguirá durante todo el curso, permite llamar al método hybrid() sobre un objeto imagen pasándole como argumento otra imagen. El comportamiento de la función es entonces como sigue: el objeto sobre el que es llamado la función será el usado para generar las frecuencias bajas y, el pasado como argumento, el que se usará para las frecuencias altas.

La implementación no tiene más misterios, aunque de nuevo su código es menos legible debido a todo el trabajo que hay que hacer por separado con cada canal, además de la división y posterior unión de estos canales. El código es como sigue:

```
1       /*
2        * Mixes a low-pass-filtered version of the source with a
             high-pass-filtered
3        * version of high_freq image, returning an hybrid image whose
             appeareance
4        * changes dependening on the distance at which the image is
             seen.
5        */
6       Image Image::hybrid(Image high_freq, double sigma_low, double
            sigma_high){
7           assert(this->image.size() == high_freq.image.size());
8
9           Mat result;
10
11          // Applies low-pass filter to the source and high-pass
                filter to high_freq.
12          Image low_passed = this->lowPassFilter(sigma_low);
13          Image high_passed = high_freq.highPassFilter(sigma_high);
14
15          // If the number of channels of both images is different,
                the image with the
16          // minimum number of channels (tested with 1) is expanded
                to an image with the
17          // maximum number of channels (tested with 3) copying the
```

```
            first channel.
18          if(low_passed.numChannels() != high_passed.numChannels()){
19              int max_channels = max(low_passed.numChannels(),
                    high_passed.numChannels());
20
21              //Both images are splitted in a vector with size =
                    maximum number of channels
22              vector<Mat> low_channels(max_channels);
23              vector<Mat> high_channels(max_channels);
24
25              split(low_passed.image, low_channels);
26              split(high_passed.image, high_channels);
27
28              // The image with less channels is expanded
29              if (low_passed.numChannels() < max_channels){
30                  int diff = max_channels - low_passed.numChannels();
31
32                  for (int i = diff-1; i < max_channels; i++) {
33                      low_channels[0].copyTo(low_channels[i]);
34                  }
35              }
36              else if (high_passed.numChannels() < max_channels ) {
37                  int diff = max_channels - high_passed.numChannels();
38
39                  for (int i = diff-1; i < max_channels; i++) {
40                      high_channels[0].copyTo(high_channels[i]);
41                  }
42              }
43
44              vector<Mat> result_channels(max_channels);
45
46              // Actual processing
47              for (int i = 0; i < max_channels; i++) {
48                  result_channels[i] = low_channels[i] +
                        high_channels[i];
49              }
50
51              merge(result_channels, result);
52          }
53          else{
54              // Actual processing if the number of channels is the
                    same.
55              result = low_passed.image + high_passed.image;
56          }
57
58          return Image(result);
59      }
```

Como se ve, el procesamiento real se reduce a una única línea:

```
1       result = low_passed.image + high_passed.image;
```

La imagen híbrida devuelta es entonces la suma de una imagen con un filtro de paso bajo y otra con un filtro de paso alto.

Dibujo de las tres imágenes

Se ha implementado, además, una función adicional que permite generar un *lienzo* con las tres imágenes —la de frecuencias bajas, la de frecuencias altas y la híbrida— en una misma imagen.

Se ha declarado como una función *friend* de la clase, pues no es un método propio del objeto Imagen pero necesita acceder a varios de sus atributos y métodos privados. El código es como sigue:

```
1       /**
2        * Returns an image object with the low frequencies image, the
             high frequencies image and the hybrid image
3        * all placed in the same canvas.
4        */
5       Image makeHybridCanvas(Image low, Image high, double
            sigma_low, double sigma_high){
6           assert(low.image.size() == high.image.size());
7
8           // Generates the low frequencies, high frequencies and
                hybrid images.
9           Image low_passed = low.lowPassFilter(sigma_low);
10          Image high_passed = high.highPassFilter(sigma_high);
11          Image hybrid = low.hybrid(high,sigma_low,sigma_high);
12
13          // Declare the final canvas
14          Mat canvas =
                Mat(hybrid.rows(),3*hybrid.cols(),hybrid.image.type());
15
16          // Obtain the three ROIs needed to place the images in the
                canvas
17          vector<Mat> slots(3);
18          for (int i = 0; i < 3; i++) {
19              slots[i] = canvas(
                    Rect(i*hybrid.cols(),0,hybrid.cols(),hybrid.rows())
                    );
20          }
21
22          // Places the images in the canvas ROIs
23          low_passed.copyTo(slots[0]);
24          high_passed.copyTo(slots[1]);
25          hybrid.copyTo(slots[2]);
26
27          return Image(canvas);
```

```
28      }
```

La función recibe dos imágenes y los valores de la generación de la función híbrida. Lo único que hace entonces es calcular la imagen de bajas frecuencias, la de altas frecuencias y la híbrida.

Una vez se han calculado estas imágenes se genera otra cuyo alto es el de la imagen original y cuyo ancho es de tres veces el ancho original. Después de insertar en esta nueva imagen las tres imágenes generadas, se devuelve un objeto Image con este canvas.

1.3. Pirámide gaussiana

2. Análisis de resultados