

Informe de prácticas

1

Alejandro García Montoro
agarciamontoro@correo.ugr.es

28 de octubre de 2015

1. Introducción

Las prácticas de esta asignatura se implementarán siguiendo el paradigma de orientación a objetos. Por ser esta la primera práctica, se describe aquí la estructura general que se seguirá durante todo el cuatrimestre.

La implementación gira en torno a la clase `Image`, que no es más que un *wrapper* de la clase `Mat` de OpenCV. Sobre ella se definirán los métodos necesarios, siendo públicos sólo los que sean útiles de cara al usuario final.

Muchos de los métodos privados recibirán y devolverán objetos `Mat`, que son la base sobre la que se construye `Image`, mientras que los públicos trabajarán sobre el nivel de abstracción superior que añade esta clase. Además, se podrán definir como funciones *friend* aquellas que no pertenezcan a la clase pero interactúen con ella de una forma profunda.

Los métodos se declaran en el fichero `Image.hpp` y se definen en `Image.cpp`. Reservamos el fichero `consts.hpp` para declarar y definir tipos de objetos o constantes que se necesiten en el desarrollo de las prácticas.

El fichero principal es `main.cpp`, cuyo método `main()` se actualizará para satisfacer las necesidades de cada práctica. Normalmente, se colocará un `waitKey(0)` entre los diferentes apartados de la práctica para facilitar su visionado.

Como primer contacto con la estructura de las prácticas, se expone aquí el estado de la clase `Image` tras finalizar este primer trabajo:

```
1      class Image{
2      private:
3          static int num_images;
4
5          Mat image;
6          string name;
7          int ID;
8
```

```

9         void imageInit(string filename, string name, bool
              flag_color);
10
11         double gaussianFunction(double x, double sigma);
12         Mat getGaussMask(double sigma);
13         Mat convolution1D(const Mat& signal_vec, const Mat& mask,
              enum border_id border_type);
14         Mat convolution2D(const Mat& signal_mat, const Mat& mask,
              enum border_id border_type);
15         void copyTo(Mat dst);
16
17
18     public:
19         ~Image();
20
21         Image(string filename );
22         Image(string filename, string name );
23         Image(string filename, bool flag_color);
24         Image(string filename, string name, bool flag_color);
25         Image(Mat img, string name = "Image");
26         Image(const Image& clone);
27
28         const Image operator-(const Image rhs) const;
29         const Image operator+(const Image rhs) const;
30
31         int numChannels();
32         int rows();
33         int cols();
34         string getName();
35         void setName(string name);
36
37         Image lowPassFilter(double sigma);
38         Image highPassFilter(double sigma);
39         Image hybrid(Image high_freq, double sigma_low, double
              sigma_high);
40
41         Image reduceHalf();
42         Image pyramidDown(double sigma = 1.0);
43         Image makePyramidCanvas(int num_levels);
44
45         Image overlapContours(double low, double high, Scalar
              color = Scalar(0,0,255));
46
47         void draw();
48
49         friend Image makeHybridCanvas(Image low, Image high,
              double sigma_low, double sigma_high);
50     };

```

2. Análisis de la implementación

2.1. 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
3   * gaussian function is
4   * sampled in the interval  $[-3\sigma, 3\sigma]$ .
5   */
6  Mat Image::getGaussMask(double sigma){
7      // Gaussian function needs to be sampled between  $-3\sigma$ 
8      // and  $+3\sigma$ 
9      // and the mask has to have an odd dimension.
10     int mask_size = 2*round(3*sigma) + 1;
11
12     Mat gauss_mask = Mat(1,mask_size,CV_32FC1);
13
14     // It is necessary to normalize the mask, so the sum of
15     // its elements
16     // needs to be saved.
17     float values_sum = 0;
18
19     // Fills the mask with a sampled gaussian function and
20     // saves the sum of all elements
21     for (int i = 0; i < mask_size; i++) {
```

```

18         gauss_mask.at<float>(0,i) =
19             gaussianFunction(i-mask_size/2, sigma);
20         values_sum += gauss_mask.at<float>(0,i);
21     }
22
23     // Normalizes the gauss mask.
24     gauss_mask = gauss_mask / values_sum;
25
26     return gauss_mask;

```

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
3      * sigma
4      */
5     double Image::gaussianFunction(double x, double sigma){
6         return exp(-0.5*(x*x)/(sigma*sigma));

```

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
3      * signal_vec with the
4      * mask, applying one of two types of borders: REFLECT or
5      * ZEROS.
6      */
7     Mat Image::convolution1D(const Mat& signal_vec, const Mat&
8         mask, enum border_id border_type){
9         assert(signal_vec.rows == 1 && mask.rows == 1 && mask.cols
10             < signal_vec.cols);
11
12         int num_channels = signal_vec.channels();
13
14         // Initialization of source vector with additional borders.
15         int border_size = mask.cols/2; // Number of pixels added
16         to each side
17
18         Mat bordered;
19         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 $j + \text{mask.cols}/2$.

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 `copy-MakeBorder()`, 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:

```

1     enum border_id{
2         REFLECT = cv::BORDER_REFLECT,
3         ZEROS = cv::BORDER_CONSTANT
4     };

```

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:

```

1      /**
2      * Returns the result of convolving the two-dimensional
3      * signal_vec with a
4      * uni-dimensional mask, applied in both rows and columns with
5      * one of two types
6      * of borders: REFLECT or ZEROS.
7      */
8      Mat Image::convolution2D(const Mat& signal_mat, const Mat&
9      mask, enum border_id border_type){
10         Mat result = Mat(signal_mat.size(), signal_mat.type());
11
12         // Row-processing: the uni-dimensional mask is applied to
13         // each row separately
14         Mat result_row;
15         for (int i = 0; i < result.rows; i++) {
16             // Row i is replaced with its convolution
17             convolution1D(this->image.row(i), mask,
18                 border_type).copyTo(result.row(i));
19         }
20
21         // Column-processing: the same uni-dimensional mask is
22         // applied to each column
23         // separately
24         Mat transposed_col;
25         for (int j = 0; j < result.cols; j++) {
26             // Column i is replaced with its convolution ---needs
27             // transposing, as convolution1D
28             // works with row vectors---.
29             transpose(result.col(j), transposed_col);
30             transpose(convolution1D(transposed_col, mask,
31                 border_type), result.col(j));
32         }
33
34         // Returns the convoluted image
35         return result;
36     }

```

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
3      * gaussian mask that is
4      * built with the parameter sigma.
5      */
6      Image Image::lowPassFilter(double sigma){
7          Mat gaussMask = getGaussMask(sigma);
8
9          Mat result = convolution2D(this->image, gaussMask,
10                                     REFLECT);
11
12          return Image(result);
13      }
```

2.2. 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
3      * source to the source
4      * itself, remaining the high frequencies.
5      */
6      Image Image::highPassFilter(double sigma){
7          return *this - this->lowPassFilter(sigma);
8      }
```

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
3      *   high-pass-filtered
4      * version of high_freq image, returning an hybrid image whose
5      *   appeareance
6      * changes dependening on the distance at which the image is
7      *   seen.
8      */
9      Image Image::hybrid(Image high_freq, double sigma_low, double
10         sigma_high){
11         assert(this->image.size() == high_freq.image.size());
12
13         Mat result;
14
15         // Applies low-pass filter to the source and high-pass
16         // filter to high_freq.
17         Image low_passed = this->lowPassFilter(sigma_low);
18         Image high_passed = high_freq.highPassFilter(sigma_high);
19
20         // If the number of channels of both images is different,
21         // the image with the
22         // minimum number of channels (tested with 1) is expanded
23         // to an image with the
24         // maximum number of channels (tested with 3) copying the
25         // first channel.
26         if(low_passed.numChannels() != high_passed.numChannels()){
27             int max_channels = max(low_passed.numChannels(),
28                                     high_passed.numChannels());
29
30             //Both images are splitted in a vector with size =
31             // maximum number of channels
32             vector<Mat> low_channels(max_channels);
33             vector<Mat> high_channels(max_channels);
34
35             split(low_passed.image, low_channels);
36             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] +
49         high_channels[i];
50 }
51 merge(result_channels, result);
52 }
53 else{
54     // Actual processing if the number of channels is the
55     same.
56     result = low_passed.image + high_passed.image;
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
3      *   high frequencies image and the hybrid image
4      * all placed in the same canvas.
5      */
6      Image makeHybridCanvas(Image low, Image high, double
7      sigma_low, double sigma_high){
8      assert(low.image.size() == high.image.size());
9
10     // Generates the low frequencies, high frequencies and
11     //   hybrid images.
12     Image low_passed = low.lowPassFilter(sigma_low);
13     Image high_passed = high.highPassFilter(sigma_high);
14     Image hybrid = low.hybrid(high,sigma_low,sigma_high);
15
16     // Declare the final canvas
17     Mat canvas =
18         Mat(hybrid.rows(),3*hybrid.cols(),hybrid.image.type());
19
20     // Obtain the three ROIs needed to place the images in the
21     //   canvas
22     vector<Mat> slots(3);
23     for (int i = 0; i < 3; i++) {
24         slots[i] = canvas(
25             Rect(i*hybrid.cols(),0,hybrid.cols(),hybrid.rows())
26             );
27     }
28
29     // Places the images in the canvas ROIs
30     low_passed.copyTo(slots[0]);
31     high_passed.copyTo(slots[1]);
32     hybrid.copyTo(slots[2]);
33
34     return Image(canvas);
35 }
```

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.

2.3. Pirámide gaussiana

El algoritmo de generación de los niveles de una pirámide gaussiana es sencillo: para cada nivel, se toma el nivel anterior, se le aplica un filtro de paso bajo y se reduce su tamaño a la mitad.

Bajar de nivel en la pirámide gaussiana

En vez usar la función `pyrDown()` de OpenCV, que hace el paso descrito anteriormente, se ha implementado una función equivalente:

```
1      /*
2      * Returns the next level in a Gaussian pyramid: it simply
3      *   blurres the source
4      * and then downsamples it by half
5      */
6      Image Image::pyramidDown(double sigma){
7          return this->lowPassFilter(sigma).reduceHalf();
8      }
```

Esta función implementa el algoritmo descrito anteriormente: a la imagen inicial se le aplica un filtro de paso bajo y, después, se reduce su tamaño a la mitad. La implementación del filtro ya la hemos visto, así que sólo queda presentar la función `reduceHalf()`:

```
1      /**
2      * Downsamples an image reducing its size by half. The
3      *   returned image is
4      * a copy of the source with odd rows and columns removed.
5      */
6      Image Image::reduceHalf(){
7          Mat dst_rows = Mat(this->rows()/2,
8                             this->cols(),this->image.type());
9          Mat dst = Mat(this->rows()/2,
10                        this->cols()/2,this->image.type());
11
12          // First remove the odd rows
13          for (int i = 0; i < dst_rows.rows; i++) {
14              this->image.row(2*i).copyTo(dst_rows.row(i));
15          }
16
17          // Then, remove the odd columns from the previous output
18          for (int i = 0; i < dst.cols; i++) {
19              dst_rows.col(2*i).copyTo(dst.col(i));
20          }
21
22          return Image(dst);
23      }
```

El código anterior hace una copia de la imagen original y la devuelve con las filas y columnas impares eliminadas. Esto se consigue con dos pasos:

1. Se declara una imagen con la mitad de filas que la original y el mismo número de columnas; se copian en ella las filas impares de la imagen original.
2. Se declara una image con la mitad de filas *y* la mitad de columnas que la original; se copian en ella las columnas impares de la imagen resultado de 1.

Esto completa la implementación de la función propia `pyramidDown()`, que emula el comportamiento de la función `pyrDown()` de OpenCV.

Dibujo de la pirámide

Una vez tenemos esta función construida, la generación de una pirámide con un número arbitrario de niveles es trivial. La única *dificultad* es acoplar todos los niveles en un solo objeto para visualizarlos a la vez.

El código siguiente implementa esta funcionalidad:

```
1      /*
2      * Returns an image filled with a Gaussian pyramid. The number
3      * of levels of the
4      * pyramid is set by num_levels
5      */
6      Image Image::makePyramidCanvas(int num_levels){
7          // The size of the canvas is:
8          // width: image.width + image.width/2
9          // height: image.height
10         Mat canvas =
11             Mat::zeros(this->rows(),round(this->cols()*1.5),this->image.type());
12
13         // First pyramid level: the source.
14         Image pyramid_level = *this;
15
16         // Places the source in the first half of the canvas
17         Mat level_zero = canvas(
18             Rect(0,0,this->cols(),this->rows()) );
19         pyramid_level.copyTo(level_zero);
20
21         // The next levels of the pyramid are treated in the loop.
22         Mat level_i; //ROI where the pyramid level i will be placed
23
24         // ROI variables
25         int left, top, width, height;
26         left = this->cols(); // All levels (>0) are placed in the
27             second half of the canvas
```

```

25         top = 0;
26         height = 0;
27
28         for (int i = 1; i < num_levels; i++) {
29             // Blurred and downsampled image
30             pyramid_level = pyramid_level.pyramidDown();
31
32             top += height; // The top of the ROI is placed just
                           // below the previous level
33             width = pyramid_level.cols();
34             height = pyramid_level.rows();
35
36             // ROI
37             level_i = canvas( Rect(left,top,width,height) );
38
39             // Actual placing
40             pyramid_level.copyTo(level_i);
41         }
42
43         return Image(canvas);
44     }

```

La idea es sencilla: se crea una imagen con altura igual a la original y anchura igual a 1.5 veces la original. Entonces, el nivel cero de la pirámide, correspondiente a la imagen original sin modificar, se inserta en la parte izquierda del *lienzo*. Por último, se itera sobre el número de niveles especificado y:

1. Se especifica el ROI adecuado sobre el *lienzo*, de manera que todos los niveles tengan su lado izquierdo pegado a la derecha de la imagen original y su parte superior comience donde termina la parte inferior de la imagen correspondiente al nivel anterior.
2. Se inserta el nivel actual en el ROI creado.

Así, se devuelve un objeto Imagen que contiene todos los niveles de la pirámide gaussiana.

2.4. Bonus: detección de bordes

Para la consecución del bonus se ha implementado la siguiente función, que devuelve un objeto Image con una copia de la imagen original en la que se superponen los contornos encontrados, dibujados con el color especificado por el usuario —por defecto, rojo—.

```

1     Image Image::overlapContours(double low, double high, Scalar
    color){
2         Mat canny_output;

```

```

3      vector< vector<Point> > contours;
4      vector<Vec4i> hierarchy;
5
6      // Blurs the source to get more accurate contours
7      Image source = this->lowPassFilter(1.0);
8
9      // Converts source to gray scale
10     Mat gray_source;
11     if (this->numChannels() > 1) {
12         cvtColor(source.image, gray_source, CV_RGB2GRAY);
13     }
14     else{
15         gray_source = source.image;
16     }
17
18     // Detects edges using Canny Filter
19     Canny( gray_source, canny_output, low, high );
20
21     // Finds contours and store the result in the contours and
22     // hierarchy variables.
23     findContours( canny_output, contours, hierarchy,
24                  CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE );
25
26     // Draws red contours
27     Image result(*this);
28
29     for( unsigned int i = 0; i< contours.size(); i++ ){
30         drawContours( result.image, contours, i, color, 2, 8,
31                     hierarchy, 0, Point() );
32     }
33
34     return result;
35 }

```

La implementación es trivial con las funciones `Canny()`, `findContours()` y `drawContours()` de OpenCV.

Lo primero que se hace es aplicar un filtro de paso bajo a la imagen original para mejorar el posterior filtro de Canny. Como este filtro acepta sólo imágenes de un canal, es necesario convertir la imagen original a escala de grises, lo que se consigue con la función propia de OpenCV `cvtColor()`.

El objeto devuelto por Canny es otra imagen; en particular, otro objeto `Mat`, pues no es más que el resultado de aplicar una convolución con una máscara específica. Sobre este objeto se ejecuta entonces la función `findContours()`, que devuelve un vector de contornos —implementados en OpenCV como un vector de puntos— y una estructura jerárquica, que especifica la relación entre los contornos anteriores. Así, para cada contorno i , las entradas de `hierarchy[i]` especifican:

1. *hierarchy*[*i*][0]: siguiente contorno con el mismo nivel jerárquico.
2. *hierarchy*[*i*][1]: contorno anterior con el mismo nivel jerárquico.
3. *hierarchy*[*i*][2]: primer contorno hijo en la estructura jerárquica.
4. *hierarchy*[*i*][3]: contorno padre en la estructura jerárquica.

Esta estructura da muchísima información sobre la estructura topológica de la imagen y, aunque en esta práctica no se explota su potencial, es de gran utilidad en aplicaciones de visión por computador donde se quiere sacar significado de los contornos.

Por último, para cada contorno devuelto por la función `findContours()`, se ejecuta la función `drawContours()` sobre la imagen resultado con el color especificado por el usuario. Esto finaliza la implementación pedida.

3. Análisis de resultados

3.1. Función de convolución

El filtro de paso bajo funciona de la forma esperada, como se aprecia en las figuras 1 y 2.



Figura 1: Imagen original.



Figura 2: Imagen tras aplicarle filtro de paso bajo.

3.2. Imágenes híbridas

La generación de imágenes híbridas, por otro lado, depende mucho de las imágenes originales y de los parámetros escogidos. La figura 3 es un primer ejemplo de su funcionalidad, donde se ven las imágenes intermedias y la imagen híbrida final.



Figura 3: Imagen híbrida.

En este caso, como en el artículo de Oliva, Torralba y Schyns, se ha decidido tomar la imagen de frecuencias bajas en blanco y negro, para que al observar de cerca la imagen híbrida, sus formas se asocien con sombras. Los parámetros usados hacen variar mucho el resultado, y un estudio de su elección queda pendiente.

Otro ejemplo quizás mejor conseguido es el siguiente, donde las frecuencias altas corresponden a la imagen de un pez y las frecuencias bajas a la de un submarino. En la siguiente sección se ve la pirámide gaussiana construida a partir de este ejemplo, y es claro cómo al disminuir el tamaño se identifica más el submarino y menos el pez.



Figura 4: Imagen híbrida.

3.3. Pirámide gaussiana

La solución de la pirámide gaussiana es tal y como se espera. Tanto la reducción de tamaño como la construcción de la imagen con toda la pirámide son triviales. Mostramos un ejemplo en la figura 5, donde se aprecia, además de la construcción de la pirámide, la bondad del resultado de la imagen híbrida vista anteriormente.



Figura 5: Pirámide gaussiana con imagen híbrida.

3.4. Bonus: detección de bordes

La implementación de la detección de los bordes es también muy dependiente de los parámetros que se escojan y de la imagen sobre la que se trabaje. La figura 6 refleja esta función en acción, donde se detectan los bordes de un avión de forma lejana a la óptima. Un estudio particular de las imágenes y de los parámetros necesarios para mejorar la detección de los bordes queda pendiente.



Figura 6: Detección de bordes.