Informe de trabajo 1

Antonio Álvarez Caballero
analca3@correo.ugr.es

1. Convolución

La primera parte de la práctica consiste en aplicar un filtro de convolución (en este caso, un filtro de alisamiento Gaussiano) a una imagen. Para ello, partiendo de la función Gaussiana en una sola variable (recordamos que este filtro es separable en filas y columnas), realizaremos una serie de pasos.

1.1. Creación del vector máscara

El primero de ellos será, a partir de la función proporcionada $f(x) = exp(-0{,}5\frac{x^2}{\sigma^2})$ y de un valor σ , generar un vector máscara representativo. Para conseguirlo, debemos recordar que para que la máscara sea significativa debe contener la región $[-3\sigma, 3\sigma]$. Sabiendo esto, la longitud de la máscara podremos conseguirla aplicando en $C++$ esta operación: es

```
1    float dimension = 2*round(3*sigma) + 1;
```

¿Por qué? Pues porque así conseguimos discretizar 3σ y que la máscara tenga en ambos lados dicha longitud. Luego sumamos 1 para contar también el centro.

Una vez tenemos la dimensión de la máscara, debemos aplicar $f(x)$ a los índices de nuestra máscara. Es decir, para $\sigma = 1$, la dimensión sería 7, y los índices de la máscara $[-3, -2, -1, 0, 1, 2, 3]$, pues debemos aplicar $f(x)$ a dichos valores para obtener nuestro vector máscara. No debemos olvidarnos de normalizar la máscara como último paso.

```
1    Mat Image::gaussianMask(float sigma)
2    {
3      int mask_size = 2 * round(3 * sigma) + 1; // +-3*sigma plus the zero
4      int mask_center = round(3 * sigma);
5
6      float value = 0, sum_values = 0;
7      int mask_index;
8
9      Mat mask = Mat::zeros(1, mask_size, CV_32FC1);
10
11     for (int i = 0; i < mask_size; i++)
12     {
13       mask_index = i - mask_center;
14       value = exp(-0.5 * (mask_index * mask_index) / (sigma * sigma));
15
16       mask.at<float>(Point(i,0)) = value;
17       sum_values += value;
18     }
19
20     mask *= 1.0 / sum_values;
21
```

```
22      return mask;
23    }
```

1.2. Aplicación del vector máscara sobre una fila

El siguiente paso es aplicar dicha máscara sobre un vector fila. Para ello, debemos expandir el vector de entrada (a partir de ahora, vector señal) para poder tomar algún píxel vecino a los de los extremos de la imagen, para poder pasar por ellos también la máscara. En este caso hemos trabajado con los modos *Reflejado* y *Constante*. Para ello hemos utilizado la función de OpenCV *copyMakeBorder*, que es muy versátil y nos permite copiar una imagen en otra con un determinado borde. Nos permite también decidir el modo de expansión que queremos, que en nuestro caso, como hemos indicado, son *Reflejado* y *Constante*.

La lógica es la siguiente: expandimos la imagen, y luego, vamos extrayendo un ROI del mismo tamaño de la máscara por todo el vector, realizando en cada iteración el producto escalar entre el ROI y la máscara. Este resultado lo vamos volcando en una imagen de salida.

```
1     Mat Image::convolution1D1C(Mat &input, Mat &mask, bool reflected)
2     {
3       // Expand the matrix
4       Mat expanded, copy_input;
5       int borderType = BORDER_CONSTANT;
6       int offset = (mask.cols - 1) / 2;
7
8       if (reflected)
9         borderType = BORDER_REFLECT;
10
11
12      copyMakeBorder(input,expanded,0,0,offset,offset,borderType,0);
13
14      // Convolution!
15      Mat ROI;
16      Mat output = Mat::zeros(1, input.cols, CV_32FC1);
17      expanded.convertTo(expanded,CV_32FC1);
18
19      for (int i = 0; i < input.cols; i++) // Index are OK
20      {
21        ROI = Mat(expanded, Rect(i,0,mask.cols,1));
22        output.at<float>(Point(i,0)) = ROI.dot(mask);
23      }
24
25      return output;
26    }
```

En el caso de que la imagen tenga más de un canal, lo tenemos controlado: La función antes definida sólo se llama desde otra más general, que si recibe una imagen de 3 canales, las separa y aplica la función a cada canal por separado, volviéndolos a unir al terminar.

```
1     Mat Image::convolution1D(Mat &input, Mat &mask, bool reflected)
2     {
3       Mat output;
4       if (input.channels() == 1)
5         output = convolution1D1C(input,mask,reflected);
6       else
7       {
8         Mat input_channels[3], output_channels[3];
9         split(input, input_channels);
```

```
10
11          for (int i = 0; i < input.channels(); i++)
12          {
13            output_channels[i] = convolution1D1C(input_channels[i],mask,reflected↩
                );
14          }
15          merge(output_channels, input.channels(), output);
16        }
17
18        return output;
19      }
```

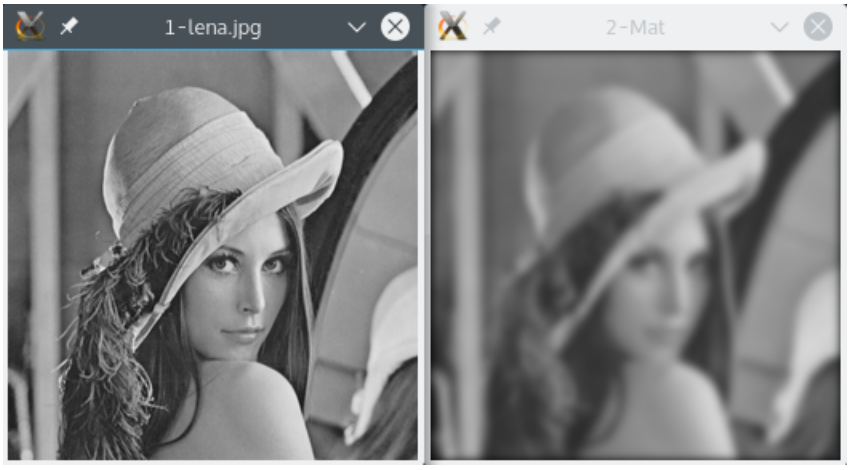
1.3. Convolución de una imagen 2D

Llegados a este punto, nos falta poco: ahora sólo tenemos que iterar por filas y columnas, utilizando las funciones antes definidas, para obtener la convolución de la imagen en su totalidad.

```
1     Mat Image::convolution2D(Mat &input, Mat &mask, bool reflected)
2     {
3       flip(mask,mask,-1);
4       Mat output = input.clone();
5
6       // Convolution
7       for (int i = 0; i < output.rows; i++)
8       {
9         Mat row = output.row(i).clone();
10        convolution1D(row,mask,reflected).copyTo(output.row(i));
11      }
12      // Transposing for convolution by cols
13      output = output.t();
14
15      for (int i = 0; i < output.rows; i++)
16      {
17        Mat row = output.row(i).clone();
18        convolution1D(row,mask,reflected).copyTo(output.row(i));
19      }
20      // Re-transposing to make the correct image
21      output = output.t();
22
23      return output;
24    }
```

Primero hacemos *flip* con la máscara en ambos ejes, es una función ya definida en OpenCV. Al recibir un número negativo da la vuelta en ambos ejes. Después de eso, itero sobre cada fila, llamamos a *Convolution1D*, y escribimos en la imagen de salida. Luego trasponemos la matriz y hacemos lo mismo. ¿Por qué trasponer la matriz? Así nos ahorramos comprobaciones dentro de los métodos: si es un vector fila, columna, ahora toma de abajo arriba en vez de izquierda a derecha... Es posible que sea algo menos eficiente, pero mucho más fácil de mantener y comprender.

Con esto ya tenemos la convolución de una imagen a través de un vector máscara. Aquí pongo un ejemplo con la imagen de *Lena*.



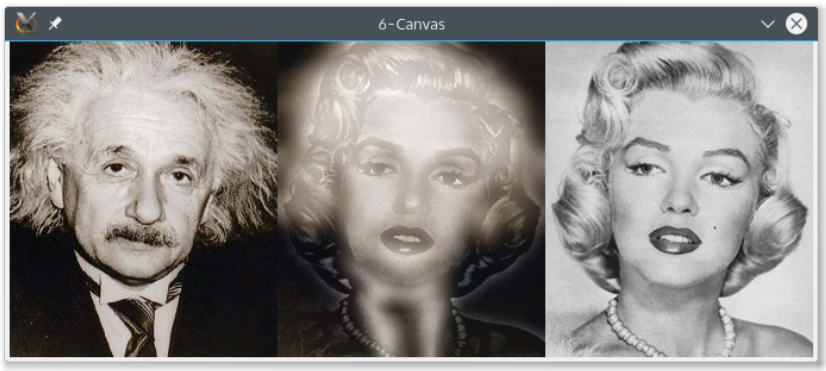
2. Imágenes Híbridas

Una imagen híbrida es una m̈ezclaẗde dos imágenes, donde de una tomamos las altas frecuencias y de la otra las bajas. Al unirlas, podemos lograr el efecto de ver una de ellas mientras estemos cerca y la otra si miramos desde lejos.

Utilizando el filtro Gaussiano conseguimos funciones para generar imágenes de alta y baja frecuencia,

[Insertar código, HAY QUE HACERLO, LO HE HECHO MAL]

Las sumamos y conseguimos este resultado



3. Pirámide Gaussiana

Para realizar la pirámide, no tenemos más que realizar un *downsampling* a la imagen. Para ello hemos realizado este pequeño bonus implementando el *downsampling*, y no usando *pyrDown()*.

Tomando 2 índices para cada imagen, es fácil quitarnos las filas y columnas pares. Es importante, como hemos visto en teoría, alisar antes la imagen. Tomamos $\sigma = 1$ para que sean 3 píxeles los que se miren. No tiene sentido hacerlo mayor, ya que vamos a quitar muchos píxeles vecinos.

```
1    Image Image::downsample()
2    {
3      // Classic technique: Blur and downsample (deleting odd cols and rows)
4      Mat output = Mat::zeros(image.rows / 2, image.cols / 2, image.type());
5
```

```
6       int i1, i2;
7       int j1, j2;
8
9       Mat mask = gaussianMask(1);
10      Mat image_blur = convolution2D(image,mask,false);
11
12      if (image.channels() == 1)
13      {
14        for (i1 = 0, i2 = 0; i1 < image_blur.rows && i2 < output.rows; i1+=2, ↩
              i2++)
15          for (j1 = 0,j2 = 0; j1 < image_blur.cols && j2 < output.cols; j1+=2,↩
                j2++)
16            output.at<float>(Point(j2,i2)) = image_blur.at<float>(Point(j1,i1))↩
                  ;
17      }
18      else
19      {
20        for (i1 = 0, i2 = 0; i1 < image_blur.rows && i2 < output.rows; i1+=2, ↩
              i2++)
21          for (j1 = 0,j2 = 0; j1 < image_blur.cols && j2 < output.cols; j1+=2,↩
                j2++)
22            output.at<Vec3b>(Point(j2,i2)) = image_blur.at<Vec3b>(Point(j1,i1))↩
                  ;
23      }
24
25      return Image(output);
26    }
```

Después de aplicar esto varias veces, insertamos los resultados en el mismo canvas y obtenemos el curioso resultado.