# 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 $\sigma$ , 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 $3\sigma$ 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      }
```

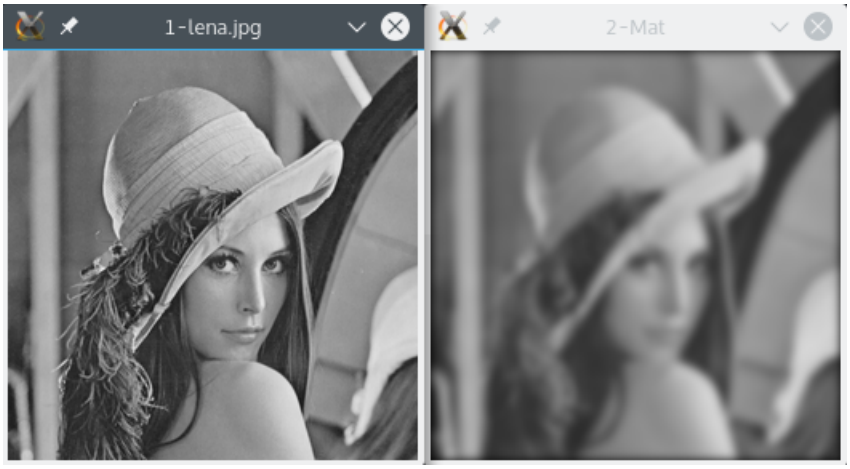
### 1.3. Convolución de una imagen 2D

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 *mezcla* de 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.

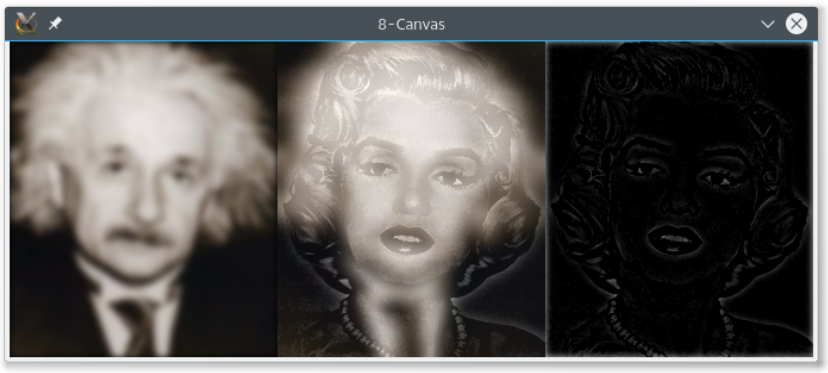
Utilizando el filtro Gaussiano conseguimos funciones para generar imágenes de alta y baja frecuencia. La de baja frecuencia es la de convolución 2D que hemos visto pero con un filtro Gaussiano, y la de alta frecuencia es la misma pero restando a la imagen original la suavizada.

```
1   Image Image::GaussConvolution(const float sigma, bool reflected)
2   {
3     Mat mask = gaussianMask(sigma);
4     Mat convolution = convolution2D(this->image, mask, reflected);
5
6     return Image(convolution);
7   }
8
9   Image Image::highFrecuencies(const float sigma, bool reflected)
10  {
11    Mat mask = gaussianMask(sigma);
12    Mat convolution = convolution2D(this->image, mask, reflected);
13
14    return Image(this->image - convolution);
15  }
16  Image Image::createHybrid(const Image &another, bool reflected, float ↩
        sigma_1, float sigma_2)
17  {
18    Mat low, high;
19    Mat output;
20
21    Mat mask_1 = gaussianMask(sigma_1);
22    Mat mask_2 = gaussianMask(sigma_2);
23
24    Mat copy_image = another.image.clone();
25
26    low = convolution2D(this->image, mask_1, reflected);
27    high = convolution2D(copy_image, mask_2, reflected);
28
29    high = copy_image - high;
30
31    output = low + high;
32
```

```
33      return Image(output);
34    }
```

Para la creación de la imagen híbrida no llamamos a las funciones anteriores por conseguir un poco de eficiencia y no liar mucho el código.

Las sumamos y conseguimos este resultado




## 3. Pirámide Gaussiana

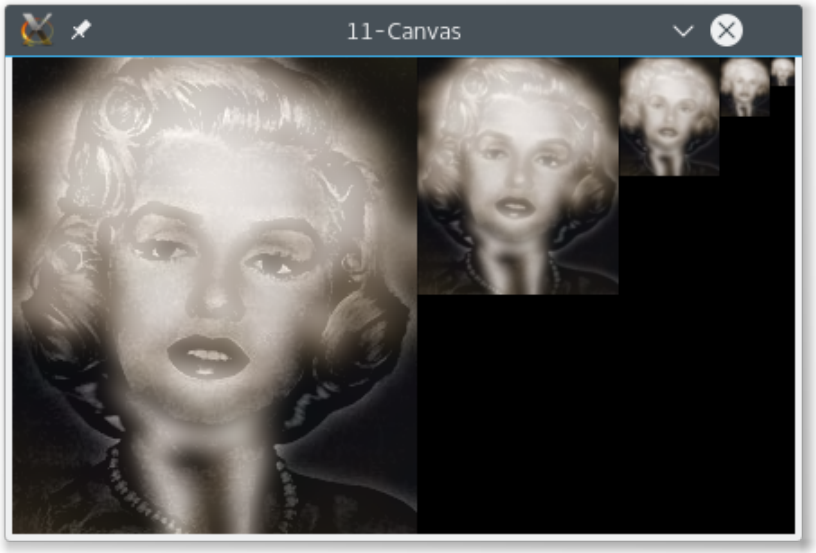
Para realizar la pirámide, no tenemos más que realizar un *downsampling* a la imagen. Para ello hemos realizado este pequeño bonus implementando el *downsampling*, y no usando *pyrDown()*.

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.




## 4. Bonus 1

Se ha implementado una función que calcula 2 máscaras para las parciales en $X$ e $Y$ de la Gaussiana. Recordando las parciales:

$$\frac{\partial G}{\partial x} = -\frac{xe^{-\frac{x^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma^2}\frac{e^{-\frac{y^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma^2}$$

Y para la derivada en $y$,

$$\frac{\partial G}{\partial y} = -\frac{e^{-\frac{x^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma^2}y\frac{e^{-\frac{y^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma^2}$$

Explico: la primera parte del cómputo es para el operando que tiene a la propia variable multiplicando (para la parcial en $X$, por ejemplo, el primer operando). La segunda es para el otro, claramente. Como son simétricos, calculamos ambas y al final comprobamos el eje: Si es el $X$, tomamos la parcial en $X$ (en el orden arriba indicado) y en otro caso la de $Y$. Análogamente definimos la de la segunda derivada, utilizando el mismo criterio.

```
1   vector <Mat> Image::MaskFirstDerivative(float sigma, char axis)
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_x = 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 = -mask_index * exp(-0.5 * (mask_index * mask_index) / (2*sigma *↩
              sigma)) * sqrt(0.5);
15
16        mask_x.at<float>(Point(i,0)) = value;
17        sum_values += value;
18      }
19
20      mask_x *= 1.0 / sum_values;
21
22      // The another part
23
24      value = 0;
25      sum_values = 0;
26
27      Mat mask_1 = Mat::zeros(1, mask_size, CV_32FC1);
28
29      for (int i = 0; i < mask_size; i++)
30      {
31        mask_index = i - mask_center;
32        value = - exp(-0.5 * (mask_index * mask_index) / (2*sigma * sigma)) * ↩
              sqrt(0.5);
33
34        mask_1.at<float>(Point(i,0)) = value;
35        sum_values += value;
36      }
37
38      mask_1 *= 1.0 / sum_values;
39
40      vector <Mat> derivatives;
41      if (axis == "x")
42      {
43        derivatives.push_back(mask_x);
44        derivatives.push_back(mask_1);
45      }
46      else
47      {
48        derivatives.push_back(mask_1);
49        derivatives.push_back(mask_x);
50      }
51
52      return derivatives;
53    }
54    vector <Mat> Image::MaskSecondDerivative(float sigma, char axis)
55    {
56      int mask_size = 2 * round(3 * sigma) + 1; // +-3*sigma plus the zero
57      int mask_center = round(3 * sigma);
58
59      float value = 0, sum_values = 0;
60      int mask_index;
61
62      Mat mask_x = Mat::zeros(1, mask_size, CV_32FC1);
63
64      for (int i = 0; i < mask_size; i++)
65      {
66        mask_index = i - mask_center;
67        value = (mask_index - sigma)*(mask_index + sigma) * exp(-0.5 * (↩
              mask_index * mask_index) / (2*sigma * sigma)) * sqrt(0.5) / sigma*↩
              sigma*sigma*sigma*sigma*sigma;
68
```

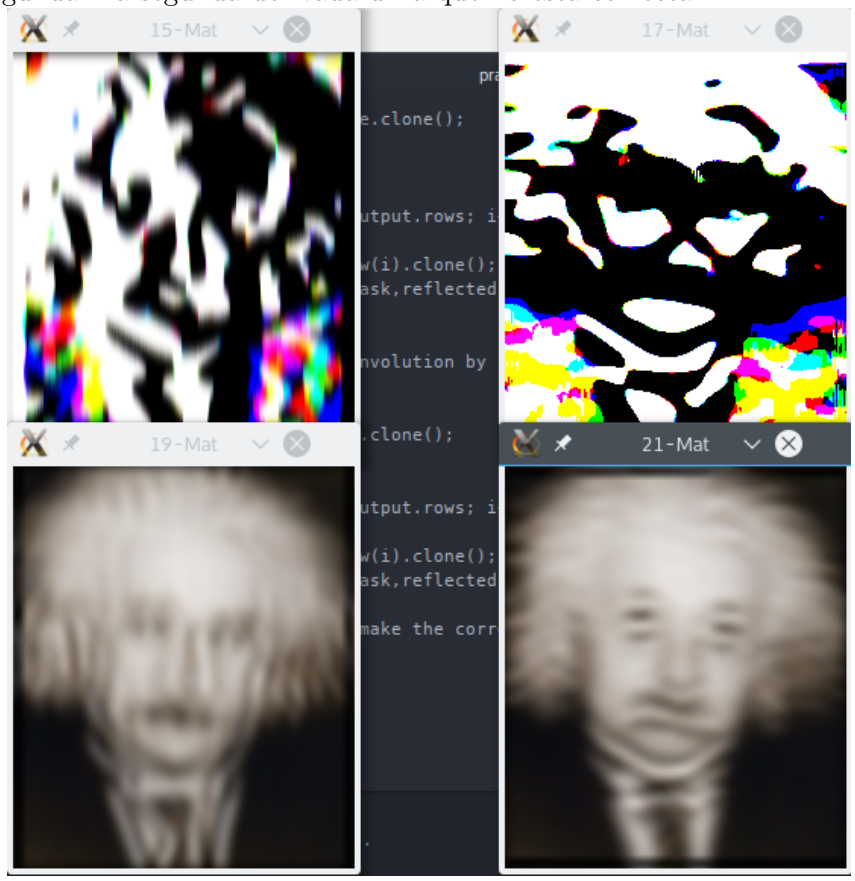
```
69          mask_x.at<float>(Point(i,0)) = value;
70          sum_values += value;
71        }
72
73        mask_x *= 1.0 / sum_values;
74
75        // The another part
76
77        value = 0;
78        sum_values = 0;
79
80        Mat mask_1 = Mat::zeros(1, mask_size, CV_32FC1);
81
82        for (int i = 0; i < mask_size; i++)
83        {
84          mask_index = i - mask_center;
85          value = exp(-0.5 * (mask_index * mask_index) / (2*sigma * sigma)) * ↩
                sqrt(0.5);
86
87          mask_1.at<float>(Point(i,0)) = value;
88          sum_values += value;
89        }
90
91        mask_1 *= 1.0 / sum_values;
92
93        vector <Mat> derivatives;
94        if (axis == "x")
95        {
96          derivatives.push_back(mask_x);
97          derivatives.push_back(mask_1);
98        }
99        else
100       {
101         derivatives.push_back(mask_1);
102         derivatives.push_back(mask_x);
103       }
104
105       return derivatives;
106     }
```

Para hacer la prueba se ha utilizado otra función (es exactamente igual para la segunda derivada, por lo que no la muestro).

```
1     Image Image::calcFirstDerivative(float sigma, char axis, bool reflected)
2     {
3       vector<Mat> derivatives = MaskFirstDerivative(sigma,axis);
4       Mat mask = derivatives[0].clone();
5
6       flip(mask,mask,-1);
7
8       // First we blur the image
9       Image tmp = GaussConvolution(3,true);
10
11      Mat output = tmp.image.clone();
12
13
14      // Convolution
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
21      // Transposing for convolution by cols
22      output = output.t();
23
24      mask = derivatives[1].clone();
25      flip(mask,mask,-1);
26
27      for (int i = 0; i < output.rows; i++)
28      {
29        Mat row = output.row(i).clone();
30        convolution1D(row,mask,reflected).copyTo(output.row(i));
31      }
32      // Re-transposing to make the correct image
33      output = output.t();
34
35      return Image(output);
36    }
```

Aquí muestro una salida con la foto de Einstein. La fila de arriba es la primera derivada y la de abajo la segunda. La segunda derivada diría que no está correcta.



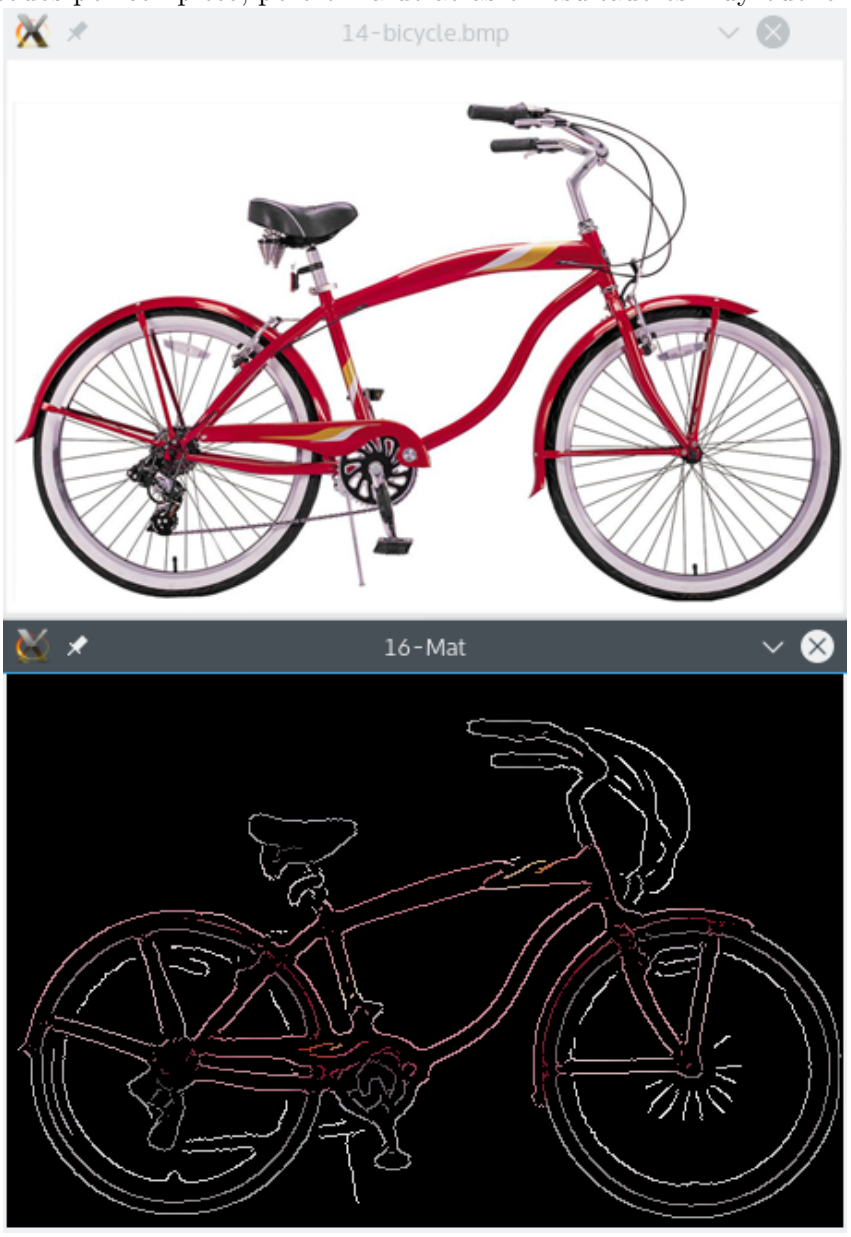

## 5. Bonus 2

El segundo bonus consiste en, usando las funciones de OpenCV, utilizar el filtro Canny en una imagen y mostrarlo sobre color sólido. Se muestra aquí la función implementada. Tomamos $\sigma = 3$ para el alisamiento ya que el filtro de Canny tiene un kernel de $3 \times 3$ por defecto.

```
1     Image Image::detectEdges(double lowThreshold, double highThreshonld)
2     {
3       // First of all we blur the image
```

```
4       Image tmp = GaussConvolution(3,true);
5
6       Mat edges;
7
8       // Apply Canny filter!
9       Canny(tmp.image, edges, lowThreshold, highThreshonld);
10
11      // Black background
12      Mat dest = Mat::zeros(tmp.image.rows, tmp.image.cols, tmp.image.type());
13
14      // Copy the input image to dest using the edges as mask
15      tmp.image.copyTo(dest, edges);
16
17      return Image(dest);
18    }
```

Aquí muestro un ejemplo con la foto de la bicicleta. Se han tomado los valores 40 y 60 como valores de threshold de manera empírica, teniendo en cuenta que en esta ocasión quería la silueta de la bicicleta, pero intentando eliminar los radios de las ruedas. En la rueda de delante no han desaparecido todos por completo, pero en la de atrás el resultado es muy bueno.



Y por último, aquí muestro el fichero main con las llamadas

```
#include "../inc/image.hpp"
#include "../inc/utils.hpp"
#include <iostream>

using namespace std;
using namespace cv;


int main(int argc, char const *argv[]) {


  Image lena(string("imagenes/lena.jpg"),false);
  Image lena_conv = lena.GaussConvolution(3);
  lena.paint();
  lena_conv.paint();

  waitKey(0);
  destroyAllWindows();

  Image einstein(string("imagenes/einstein.bmp"),true);
  Image marilyn(string("imagenes/marilyn.bmp"),true);

  Image low = einstein.GaussConvolution(3);
  Image hybrid = einstein.createHybrid(marilyn,true,6,6);
  Image high = marilyn.highFrecuencies(3);

  vector<Image*> secuencia;
  secuencia.push_back(&low);
  secuencia.push_back(&hybrid);
  secuencia.push_back(&high);

  Image tira(secuencia,1,3);
  tira.paint();

  waitKey(0);
  destroyAllWindows();

  vector<Image*> pyramid;
  Image hybrid_d1 = hybrid.downsample();
  Image hybrid_d2 = hybrid_d1.downsample();
  Image hybrid_d3 = hybrid_d2.downsample();
  Image hybrid_d4 = hybrid_d3.downsample();
  pyramid.push_back(&hybrid);
  pyramid.push_back(&hybrid_d1);
  pyramid.push_back(&hybrid_d2);
  pyramid.push_back(&hybrid_d3);
  pyramid.push_back(&hybrid_d4);

  Image pyramidImage(pyramid, 1,5);

  pyramidImage.paint();

  waitKey(0);
  destroyAllWindows();

  Image derivative_x = einstein.calcFirstDerivative(3,"x");
  Image derivative_y = einstein.calcFirstDerivative(3,"y");
  Image derivative2_x = einstein.calcSecondDerivative(3,"x");
  Image derivative2_y = einstein.calcSecondDerivative(3,"y");
  derivative_x.paint();
```

```
61      derivative_y.paint();
62      derivative2_x.paint();
63      derivative2_y.paint();
64
65      waitKey(0),
66      destroyAllWindows();
67
68      Image bicycle(string("imagenes/bicycle.bmp"),true);
69      Image edges_bicycle = bicycle.detectEdges(40,60);
70
71      bicycle.paint();
72      edges_bicycle.paint();
73
74      waitKey(0);
75      destroyAllWindows();
76
77      return 0;
78    }
```