VC P1: Filtrado y muestreo

Alejandro Alcalde, *Universidad de Granada*

27 de octubre de 2015

Índice

1. Ejercicio A.1

Cálculo del vector máscara. Con $f(x) = e^{-0,5\frac{x^2}{\sigma^2}}$

Este apartado se ha resuelto con el siguiente código (El parámetro `bool highpass` se usará más adelante, en las imágenes híbridas):

```
/**
* Get a 1D Gaussian kernel for the given parameters
*
* @param sigma
* @param type Type of Mat
* @param highpass True if we want a high pass kernel
* @return The kernel
*/
Mat myGetGaussianKernel1D(double sigma, bool highpass) {
  // Kernel size [-sigma, +sigma]
  int ksize = 2 * 3 * floor(sigma) + 1;
  Mat kernel(1, ksize, CV_64F);
  // Compute the mask with the given sigma
  int middle = ksize/2;
  // Fill the kernel symmetrically
  for (int i = 0; i <= middle; i++) {
      double gaussian = (double) exp(-.5 * ((i * i) / (sigma * sigma)));
      kernel.at<double>(i+middle) = highpass ? 1 - gaussian : gaussian;
      kernel.at<double>(middle-i) = kernel.at<double>(i+middle);
  }

  // Normalize the kernel to sum 1, it is a smooth kernel
  kernel = kernel * (1 / *(sum(kernel).val));

  return kernel;
}
```

El resultado, con $\sigma = 1$ es el siguiente:

```
EXCERSICE ONE RESULT: (with sigma=1)
Kernel: [0.004433048175243745, 0.05400558262241448, 0.2420362293761143,
 ↪   0.3990502796524549, 0.2420362293761143, 0.05400558262241448,
 ↪   0.004433048175243745]
```

El tamaño del *kernel* estará comprendido en el intervalo $[-3\sigma, +3\sigma]$, ya que para rangos mayores no es representativo, como se puede apreciar en la figura 1 (Gráfica obtenida en dev.theomader.com), con el intervalo anterior cubrimos toda la Gaussiana. El *kernel* está normalizado, es decir, la suma de todos los elementos es 1. Para asegurar que el tamaño siempre es impar, se define el tamaño a partir de σ con `int ksize = 2 * 3 * floor(sigma) + 1;`, Es necesario que sea impar porque a la hora de realizar un cálculo sobre un píxel, se realiza sobre el central. A continuación se rellena el *kernel* con los valores de la función, desde el centro hacia afuera.

2. *Ejercicio A.2*

Calcular la convolución 1D

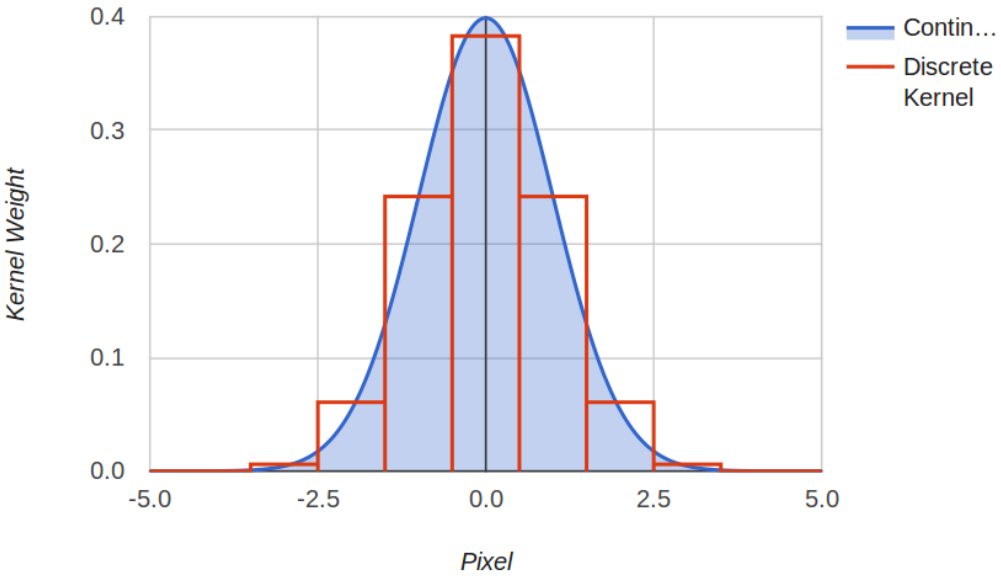

Figura 1: *Gráfica de la Gaussiana con $\sigma = 1$*

Para realizar el cálculo de la convolución 1D se ha escrito el siguiente código:

```
/**
* Computes the 1D convolution for the given signal vector, using the kernel
*
* @param signalVector Vector in which apply the convolution.
* @param kernel The mask to use
* @param border Only allows BORDER_CONSTANT|BORDER_REFLECT
* @return
*/
Mat convolutionOperator1D(Mat &signalVector, Mat &kernel, BorderTypes border)
 ↪  {
  Mat filtered;
  bool was1col = false;
  if (!signalVector.empty() || !kernel.empty()) {
      // If we receive a signalvector with one column, transpose it
      if (signalVector.cols == 1) {
          signalVector = signalVector.t();
          was1col = true;
      }
      int extraBorder = kernel.cols / 2;
      vector<Mat> signalVectorByChannels(signalVector.channels());
      split(signalVector, signalVectorByChannels);
      for (vector<Mat>::const_iterator it = signalVectorByChannels.begin();
 ↪   it != signalVectorByChannels.end(); ++it) {
          Mat m = *(it);
          // Create a new Mat with the extra borders needed
          Mat signalWithBorder;
```

```
            // Add extra borders to the vector to solve boundary issue
            copyMakeBorder(m, signalWithBorder, 0, 0, extraBorder, extraBorder,
↪   border, Scalar(0));
            // Vector to store the convolution result
            filtered = m.clone();
            // Create a ROI to pass along the vector and compute convolution
↪   with the kernel
            Mat roi(signalWithBorder, Rect(0, 0, kernel.cols, 1));
            for (int i = 0; i < m.cols; i++) {
                // Multiply the focused section by the kernel
                Mat r = roi.mul(kernel);
                // Sum the result of the above operation to the pixel at i
                filtered.at<double>(i) = (double) *(sum(r).val);
                // Move the Roi one position to the right
                roi = roi.adjustROI(0, 0, -1, 1);
            }
            filtered.copyTo(m);
        }
        // Merge the vectors into a multichannel Mat
        merge(signalVectorByChannels, filtered);
    }
    filtered = was1col ? filtered.t() : filtered;

    return filtered;
}
```

Una de las comprobaciones que se realizan es ver si se nos pasa un vector $1xN$ o $Nx1$, si se pasó como $Nx1$, se realiza la traspuesta para poder operar correctamente. Luego se calcula cuantos bordes extra se deberán añadir en función del tamaño del *kernel*. Una vez construida la matriz de destino, se itera sobre cada pixel del vector y se aplica la convolución usando un *roi* y desplazándolo por todo el vector. En esta función se realizan varias operaciones más, pero se verán en el siguiente apartado.

Para hacer uso de esta función, se ha escrito el siguiente código, usando el *kernel* calculado en el apartado A.1:

```
// Exercise two, compute convolution of a 1D signal vector and a kernel
// create and initialize 3 channel Mat
Mat vector(1, 7, CV_64FC3, CV_RGB(50, 150, 255));
Mat result = convolutionOperator1D(vector, kernel, BORDER_CONSTANT);
if (!debug) {
    cout << "EXCERSICE TWO RESULT: (with BORDER_CONSTANT)" << endl;
    cout << setw(15) << "Convol1D: " << result << endl;
    cout << "Press enter to show next example" << endl;
    cout << "EXCERSICE TWO RESULT: (with BORDER_REFLECT)" << endl;
    result = convolutionOperator1D(vector, kernel, BORDER_REFLECT);
    cout << setw(15) << "Convol1D: " << result << endl;
    cout << "Press enter to show next exercise result" << endl;
    cin.get();
}
```

Cuyo resultado es:

```
EXCERSICE TWO RESULT: (with BORDER_CONSTANT)
Convol1D: [178.378910655688, 104.9287709739341, 34.97625699131137,
 ↪  240.0981491465972, 141.2342053803513, 47.07806846011709,
 ↪  253.8695727153129, 149.3350427737134, 49.77834759123781, 255, 150, 50,
 ↪  253.8695727153129, 149.3350427737135, 49.77834759123781,
 ↪  240.0981491465972, 141.2342053803513, 47.07806846011709,
 ↪  178.378910655688, 104.9287709739341, 34.97625699131137]
```

Si usamos `BORDER_REFLECT`, el resultado es:

```
EXCERSICE TWO RESULT: (with BORDER_REFLECT)
Convol1D: [255, 150, 50, 255, 150, 50, 255, 150, 50, 255, 150, 50, 255, 150,
 ↪  50, 255, 150, 50, 255, 150, 50]
```

3. *Ejercicio A.3*

Calcular la convolución 2D

Se ha diseñado una nueva función para resolver este problema:

```
/**
* Compute the convolution of an image using the given sigma
*
* @param m The image to which apply convolution
* @param sigma Sigma to compute the Gaussian kernel
* @param highpass True if we want a high pass kernel
* @return An image with the filter applied
*/
Mat computeConvolution(Mat &m, double sigma, bool highpass) {
    Mat result;
    if (!m.empty()) {
        result = m.clone();
        // Store type to restore it when the convolution is computed
        int type = result.channels() == 1 ? CV_64F : CV_64FC3;
        // Convert the image to a 64F type, (one or three channels)
        result.convertTo(result, type);
        Mat kernel = myGetGaussianKernel1D(sigma, highpass);
        // This kernel is separable, apply convolution for rows and columns
 ↪   separately
        for (int i = 0; i < result.rows; i++) {
            Mat row = result.row(i);
            row = convolutionOperator1D(row, kernel, BORDER_REFLECT);
            row.copyTo(result.row(i));
        }
        for (int i = 0; i < result.cols; i++) {
            Mat col = result.col(i);
            col = convolutionOperator1D(col, kernel, BORDER_REFLECT);
            col.copyTo(result.col(i));
```

```
        }
        result.convertTo(result, m.type());
    }
    return result;
}
```

Esta función, internamente usará la función `Mat convolutionOperator1D(Mat &signalVector, Mat &kernel, BorderTypes border)` del apartado A.2. Esto ha sido posible ya que la función *Gaussiana* es separable, y por tanto permite aplicar la convolución por filas y por columnas. Antes de realizar ningún cálculo, se convierten las matrices al mismo tipo de dato, en este caso se usan `doubles`, se permiten imágenes tanto de un canal como de tres. Una vez convertida la imagen al mismo tipo de dato del *kernel*, se procede a obtener un *kernel* con el correspondiente σ pasado como parámetro. `highpass` cobrará significado en ejercicios posteriores. Tras esto, se aplica la convolución por filas primero, y luego por columnas.

El código que usa esta función es la siguiente:

```
cout << "EXCERSICE THREE RESULT:" << endl;
Mat colorImage = imread("./images/dog.bmp", IMREAD_UNCHANGED);
Mat grayImage = imread("./images/dog.bmp", IMREAD_GRAYSCALE);

drawImage(colorImage, "Before Convolution | ColorImage");
drawImage(grayImage, "Before Convolution | GrayImage");

colorImage = computeConvolution(colorImage, sigma);
grayImage = computeConvolution(grayImage, sigma);

drawImage(colorImage, "After Convolution | ColorImage");
drawImage(grayImage, "After Convolution | GrayImage");

cout << "Press enter to show next exercise result" << endl;
cin.get();
```

Las imágenes originales, a color y escala de grises se muestran en las figuras 2 , 3. Aplicando un $\sigma = 1$ se obtienen las imágenes alisadas de las figuras 4 , 5, si aplicamos la convolución con $\sigma = 5$, la imagen está considerablemente más emborronada, como se aprecia en las figuras 6 , 7

4. *Ejercicio B.1*

Escribir una función que genere las imágenes de alta y baja frecuencia

Para este ejercicio se ha implementado la siguiente función:

```
/**
* Computes both high frequency and low frequency with the given sigmas for
 ↪  the
* two images passed in
*
* @param highFreq The image use as high frequency
* @param lowFreq The image use as low frequency
* @param highSigma Sigma value for the high frequency
* @param lowSigma Sigma value for the low frequency
*
* @return A vector with the hybrid image, low and high freq images
*/
vector<Mat> hybridImage(Mat &highFreq, Mat &lowFreq, double highSigma, double
 ↪  lowSigma) {
    vector<Mat> result;
    Mat highBlurred = computeConvolution(highFreq, highSigma, true); // high
 ↪  pass filter
    Mat lowBlurred = computeConvolution(lowFreq, lowSigma); // low pass
 ↪  filter
    // Get the high frequencies of the image
    Mat highH = highFreq - highBlurred;
    // Generate the hybrid image
    Mat H = lowBlurred + highH;
    result.push_back(H);
    result.push_back(highH);
    result.push_back(lowBlurred);
    return result;
}
```

Con ella se pretende calcular la imagen híbrida a partir de dos imágenes, una en alta frecuencia y otra en baja. A cada una se le aplicará un σ distinto. La imagen híbrida H se calcula combinando dos imágenes I_1 y I_2, a una se le aplica un filtro de paso bajo G_1, en este caso a la imagen `lowBlurred`, y a otra un filtro de paso alto G_2, en este caso a `hithBlurred`. A la imagen de alta frecuencia se le aplica el filtro de la forma $1 - G_2$. Todo combinado da lugar a la imagen híbrida $H = I_1 \cdot G_1 + I_2 \cdot (1 - G_2)$. Para indicar que queremos pasar un filtro alto se llama a la función `computeConvolution()` con un tercer parámetro a `true`. Tras aplicar los filtros, se obtienen las altas frecuencias de la imagen `highFreq` restandole a ésta el resultado de alisarla (`Mat highH = highFreq - highBlurred;`) y se genera la híbrida combinando `lowBlurred` y `highH`. El resultado de ésta operación es el mostrado en la figura 8.

Ambos valores de σ se han obtenido de forma experimental, hasta obtener un resultado aceptable.

Esta función se usa como sigue:

```
double sigma1 = 7;
double sigma2 = 10;
```

```
Mat low = imread("./images/dog.bmp", IMREAD_UNCHANGED);
Mat high = imread("./images/cat.bmp", IMREAD_UNCHANGED);

std::vector<Mat> hybrid = hybridImage(high, low, sigma1, sigma2);
drawHybrid(hybrid);

cout << "Press enter to show the next hybrid image" << endl;

low = imread("./images/marilyn.bmp", IMREAD_UNCHANGED);
high = imread("./images/einstein.bmp", IMREAD_UNCHANGED);

sigma1 = 3;
sigma2 = 5.5;
std::vector<Mat> hybrid2 = hybridImage(high, low, sigma1, sigma2);
drawHybrid(hybrid2);

cout << "Press enter to show the next hybrid image" << endl;

sigma1 = 3;
sigma2 = 7;
low = imread("./images/fish.bmp", IMREAD_UNCHANGED);
high = imread("./images/submarine.bmp", IMREAD_UNCHANGED);

std::vector<Mat> hybrid3 = hybridImage(high, low, sigma1, sigma2);
drawHybrid(hybrid3);
cout << "Press enter to show the next hybrid image" << endl;

sigma1 = 3;
sigma2 = 7;
low = imread("./images/motorcycle.bmp", IMREAD_UNCHANGED);
high = imread("./images/bicycle.bmp", IMREAD_UNCHANGED);

std::vector<Mat> hybrid4 = hybridImage(high, low, sigma1, sigma2);
drawHybrid(hybrid4);

sigma1 = 5;
sigma2 = 3;
low = imread("./images/bird.bmp", IMREAD_UNCHANGED);
high = imread("./images/plane.bmp", IMREAD_UNCHANGED);

std::vector<Mat> hybrid5 = hybridImage(high, low, sigma1, sigma2);
drawHybrid(hybrid5);

cout << "Press enter to show the next hybrid image" << endl;

cout << "Press enter to show the next exercise result" << endl;
cin.get();
```

---

5. *Ejercicio B.2*

Escribir una función para mostrar las tres imágenes (alta, baja e híbrida) en una misma ventana

La función creada ha sido la siguiente:

```
/**
* Shows an image in the screen
*
* @param m Image to show
* @param wn Name of the window
*/
void drawImage(Mat &m, string windowName) {
    if (!m.empty()) {
        namedWindow(windowName, WINDOW_AUTOSIZE);
        imshow(windowName, m);
        waitKey(0);
        destroyWindow(windowName);
    }
}
////////////////////////////////////////////////////
/**
 * Draws an hybrid image and the two original images
 *
 * @param m Vector with images to draw
 */
void drawHybrid(const std::vector<Mat> &m) {
    if (!m.empty()) {
        int height = 0;
        int width = 0;
        // Get the size of the resulting window in which to draw the images
        // The window will be the sum of all width and the height of the
 ↪  greatest image
        for (std::vector<Mat>::const_iterator it = m.begin(); it != m.end();
 ↪  ++it) {
            Mat item = (*it);
            width += item.cols;
            if (item.rows > height) {
                height = item.rows;
            }
        }
        // Create a Mat to store all the images
        Mat result(height, width, CV_8UC3);
        // Black background
        result = 0;
        int x = 0;
        for (std::vector<Mat>::const_iterator it = m.begin(); it != m.end();
 ↪  ++it) {
            Mat item = (*it);
            // If a image is in grayscale or black and white, convert it to 3
 ↪  channels 8 bit depth
            if (item.type() != CV_8UC3) {
                cvtColor(item, item, CV_GRAY2RGB);
            }
            Mat roi(result, Rect(x, 0, item.cols, item.rows));
            item.copyTo(roi);
            x += item.cols;
        }
        drawImage(result, "Ventana");
    }
```

```
}
```

Básicamente se crea una matriz de tamaño suficiente para albergar las tres imágenes, y se pintan de izquierda a derecha. Se contempla el caso de que las imágenes que se pasen estén en otro tipo de dato (Blanco y negro, escala de grises, varios canales etc) y se convierten todas a un único tipo. Se usa de la siguiente forma:

```
std::vector<Mat> hybrid = hybridImage(high, low, sigma1, sigma2);
drawHybrid(hybrid);
```

El resultado de llamar a ésta función es el visto en la figura 8.

6. *Ejercicio C.1*

Crear una función que muestre una pirámide Gaussiana de al menos 5 niveles
Se ha creado una función simple que acomete este objetivo:

```
/**
* Draws a Gaussian pyramid with the given levels
*
* @param hybrid Hybrid images
* @param levels Levels of the pyramid
*/
void gaussianPyramid(Mat &hybrid, int levels){
  vector<Mat> pyramid;
  pyramid.push_back(hybrid);
  for (int i = 0; i < levels; i++) {
      Mat r;
      pyrDown(pyramid.at(i), r);
      pyramid.push_back(r);
  }
  drawHybrid(pyramid);
}
```

La función llama al método `pyrDown` de OpenCV que calcula la pirámide, el resultado es el mostrado en las figuras 9 y 10.

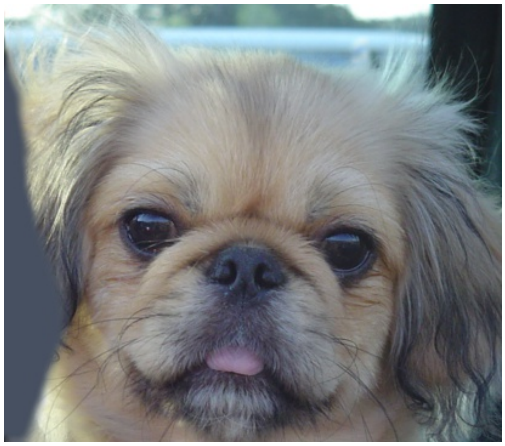

Figura 2: *Imagen original a color*

Figura 3: *Imagen original a escala de grises*

Figura 4: $\sigma = 1$

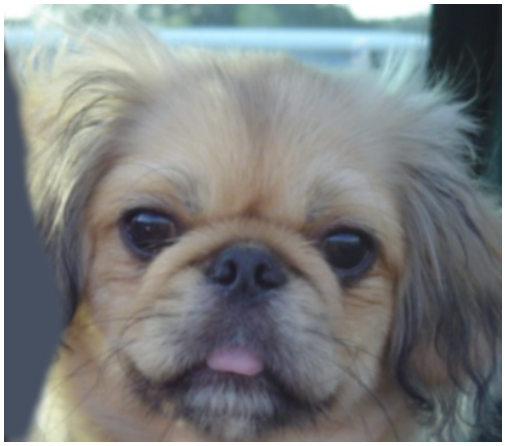

Figura 5: $\sigma = 1$

Figura 6: $\sigma = 5$

Figura 7: $\sigma = 5$

Figura 8: *Imagen híbrida*

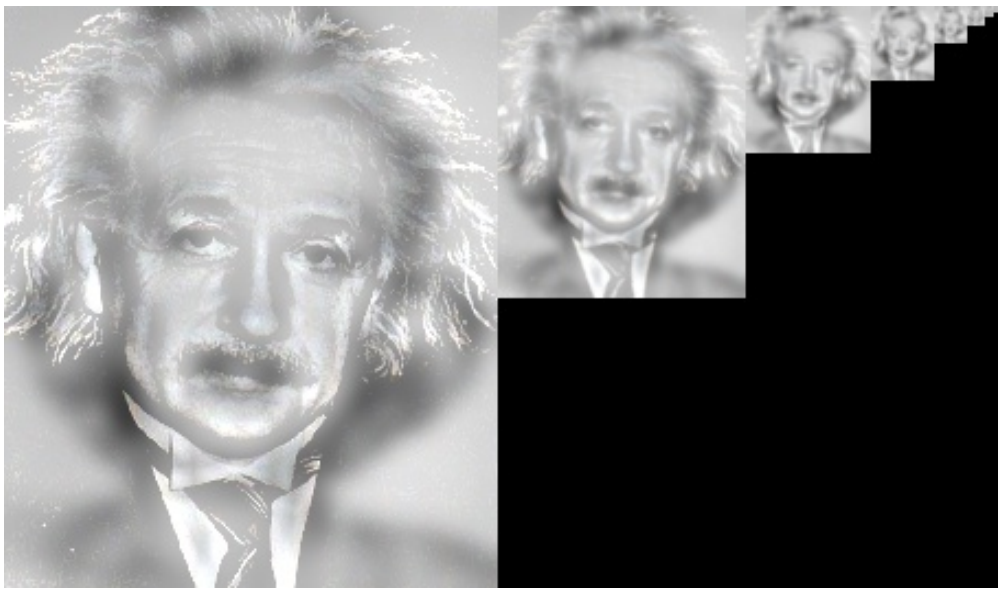

Figura 9: *Pirámide Einstein*

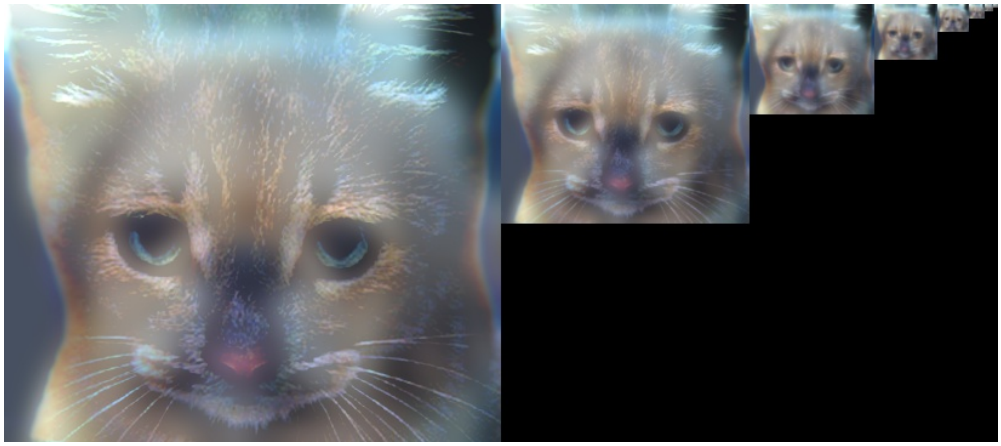

Figura 10: *Pirámide Gato*