VISIÓN POR COMPUTADOR
Trabajo-1

Convolución gaussiana de una imagen

Se ha implementado la función **gaussianFilter()** como la función principal para la obtención de la imagen convolucionada en función del parámetro sigma introducido.

```
/**
* Función general para el cálculo de la convolución de la imagen @img con el valor de @sigma,
* si no se pasa como parámetro valor a sigma, este tendrá por defecto el valor de 1.
*/
Mat gaussianFilter(Mat &img, double sigma = 1.0){
	Mat mask  = convMask(sigma);
	return gaussianConvFilter(img, mask);
}
```

Lo primero que debemos hacer es obtener el tamaño y los valores de la máscara a partir del valor de sigma, esto se realiza en la función **convMask()**

```
/**
* Obtiene los valores de la máscara de convolución en funcion del parámetro @sigma y normaliza
* los valores obtenidos, todo esto se guarda en mask.
*/
Mat convMask(double sigma){
	int size = getKernelSize(sigma);
	double normal = 0, tmp_value;
	vector<double> v;
	v.resize(size);

	for(int i=1; i<=size/2; i++){
		tmp_value = exp(-0.5*((i*i)/(sigma*sigma)));

		v[(size/2)-i] = tmp_value;
		v[(size/2)+i] = tmp_value;
		normal += tmp_value;
	}
	normal*=2;
	tmp_value = exp(-0.5*(0/(sigma*sigma)));
	v[(size/2)] = tmp_value;
	normal += tmp_value;

	Mat mask(v, true);
	mask /= normal;
```

```
	return mask;
}
/**
* Obtiene el tamaño del kernel en función de @sigma y el @value son las unidades sigma que
* se escogen de rango, por defecto 6 (-3,3)
*/
int getKernelSize(double sigma = 1.0, int value = 6){
	return round(value*sigma)+1;
}
```

Con una función auxiliar como es **getKernelSize()** generamos el tamaño de la máscara, que dependerá del valor de sigma y los valores que queramos tomar de la función gaussiana; cogeremos desde [-3,3] sigma que son 6 valores. Se establecerá esto por defecto y a continuación se suma 1, el valor restante, de esta forma tenemos la máscara impar.

Una vez que sabemos el tamaño creamos un vector y añadimos los valores a éste de los generados de la función gausiana **exp(-0.5*((x*x)/(sigma*sigma)))** guardándose a la vez en forma espejo (el recorrido comienza en el centro del vector y se mueve a los lados) y se normalizan los valores del vector máscara tras haberla pasado al tipo de dato **Mat**.

Ya tenemos nuestra máscara, ahora falta pasársela a una imagen, esto ocurre en la función **gaussianConvFilter()**

```
/**
* Para la imagen @img y con la máscara @mask se aplican los bordes a la imagen (para evitar
* problemas de desbordamiento), se coge una porción de la imagen y se llama a la función que
* aplica la convolución en función del canal que estemos.
*/
Mat gaussianConvFilter(Mat img, Mat &mask){
	int total_channels = img.channels();
	img.convertTo(img, CV_64F);
	Mat aux, img_edges = apply_edges(img, mask.rows, REFLECTION);
	vector<Mat> channels;
	for(int i=0; i < img.cols; i++){
		for(int j=0; j < img.rows; j++){
			aux = img_edges(Rect(i, j, mask.rows, mask.rows));

			if(aux.channels() > 1){
				split(aux, channels);
				for(int k=0; k<total_channels; k++)
convMaskChannel(channels[k], mask, img.at<Vec3d>(j, i)[k]);
			}
			else convMaskChannel(aux, mask, img.at<double>(j, i));
		}
```

```
        }
        img.convertTo(img, CV_8UC3);
        return img;
}
```

Primero aplica los bordes a la imagen con **apply_edges()** (reflejados en este caso al tener la función con último parámetro REFLECTION) antes de realizar la convolución; si el parámetro hubiese sido ZEROS en lugar de REFLECTION los bordes aplicados serían negros:

```
/**
* En función del tamaño de la máscara @mask_size y el @type aplica unos bordes a la imagen
@img para evitar
* problemas durante la convolución.
*/
Mat apply_edges(Mat img, int mask_size, int type = ZEROS){
        if(type == ZEROS){
                Mat horizontal_edge(img.rows, mask_size/2, img.type(), 0.0);
                hconcat(horizontal_edge, img, img); hconcat(img, horizontal_edge, img);

                Mat vertical_edge(mask_size/2, img.cols, img.type(), 0.0);
                vconcat(vertical_edge, img, img); vconcat(img, vertical_edge, img);
        }
        else if(type == REFLECTION){
                Mat horizontal_edge_left, horizontal_edge_right, vertical_edge_down,
vertical_edge_up;

                horizontal_edge_left = img.rowRange(0, img.rows);
                horizontal_edge_left = horizontal_edge_left.colRange(0, mask_size/2);
                hconcat(horizontal_edge_left, img, img);

                horizontal_edge_right = img.rowRange(0, img.rows);
                horizontal_edge_right = horizontal_edge_right.colRange(img.cols - mask_size/2,
img.cols);
                hconcat(img,horizontal_edge_right, img);

                vertical_edge_up = img.rowRange(0, mask_size/2);
                vertical_edge_up = vertical_edge_up.colRange(0, img.cols);
                vconcat(vertical_edge_up, img, img);

                vertical_edge_down = img.rowRange(img.rows - mask_size/2, img.rows);
                vertical_edge_down = vertical_edge_down.colRange(0, img.cols);
                vconcat(img,vertical_edge_down, img);
        }

        return img;
}
```

Una vez aplicados los bordes se obtiene de la nueva imagen con bordes un cuadrado de las dimensiones de la máscara que será la que posteriormente se multiplique con la máscara 1D primero por columnas y luego por filas.

Posteriormente se llamará a otra función **convMaskChannel().** Ésta se llama una vez por cada canal que tenga la imagen, por lo que si tiene 3, se llamará 3 veces.

Aquí es donde se realiza la multiplicación entre el cuadrado (de un sólo canal) de valores de la imagen por la máscara 1D primero por columnas. Una vez realizada la multiplicación se vuelve a hacer pero por filas y guardamos el resultado obtenido en la variable **result** que será **la variable pasada por referencia que apunta a la posición del pixel** que se va a modificar**.**

```
/**
* Guarda en @result el valor final que tendrá el pixel concreto del canal que se encuentre
* en el momento de llamar a esta función. El parámetro @signal es la señal que recibe para
* realizar la convolución con la máscara 1D @mask.
*/
void convMaskChannel(Mat &signal, Mat &mask, double &result){
	vector<double> aux(signal.cols);
	result = 0;
	for (int i = 0; i < signal.cols; i++){
		aux[i] = 0.0;
		for (int j = 0; j < signal.rows; j++){
			aux[i] += signal.at<double>(j, i)*mask.at<double>(j,0);
		}
		aux[i] *= mask.at<double>(i,0);
		result += aux[i];
	}
}
```

Tras aplicar esta función por cada canal de la imagen ya tendríamos realizada la convolución de la máscara sobre la imagen en cuestión.

Dos ejemplos de la aplicación de la máscara de convolución con los valores de sigma 2 y 4 respectivamente para dos imagénes son los siguientes:

Imágenes híbridas

Para obtener una imagen híbrida tenemos que juntar dos imágenes donde en cada una de ellas predominen frecuencias bajas y la otra frecuencias altas. La función que realiza esta operación es **hybrid_images().**

```
/**
* Obtiene una imagen híbrida formada por la convolución de la imagen @img1 y la imagen @img2
* donde en la primera imagen se pretende eliminar las frecuencias altas de la imagen mientras
* que en la imagen 2 si obtener las frecuencias altas.
*/
vector<Mat> hybrid_images(Mat img1, Mat img2, double sigma1, double sigma2, int mode =
IMPLEMENTATION
        )
{
   Mat high_freq, high_freq_brigthness, low_freq, img1_smooth, img2_smooth, aux;
   vector<Mat> hybrid_image;

   resize(img2, img2, Size(img1.cols, img1.rows));
   img1_smooth = gaussianFilter(img1, sigma1);
   img2_smooth = gaussianFilter(img2, sigma2);
   img2_smooth.convertTo(img2_smooth, CV_64F);
   img1_smooth.convertTo(img1_smooth, CV_64F);
   img1.convertTo(img1, CV_64F);
   img2.convertTo(img2, CV_64F);

   low_freq = img2_smooth;
   high_freq = img1 - img1_smooth;

   if(mode == IMPLEMENTATION) suppressNegativesValues(high_freq);
   else if(mode == OPENCV) normalize(high_freq, high_freq, 0, 255, NORM_MINMAX, CV_64F);

   aux = (low_freq + high_freq)/2;

   low_freq.convertTo(low_freq, CV_8UC3);
   high_freq.convertTo(high_freq, CV_8UC3);
   aux.convertTo(aux, CV_8UC3);

   hybrid_image.push_back(low_freq);
   hybrid_image.push_back(high_freq);
   hybrid_image.push_back(aux);

   return hybrid_image;
}
```

Recibimos como parámetros dos imágenes, los sigmas respectivos y una variable entera, modo, que define el tipo de función de normalización que se llamará más adelante.

Lo primero que hay que hacer es aplicar los filtros de convolución con sus sigmas a cada imagen; a continuación una de ellas (en este caso la primera) será la de alta frecuencia, por lo que se resta la imagen original por la suavizada, llamemosla **high_freq_img** y la segunda imagen se queda únicamente con el filtro de convolución aplicado y será **low_freq_img.**

Tras haber obtenido **high_freq_img** éste contiene valores negativos. Para no trabajar con ellos vamos a normalizar esta imagen para tener valores comprendidos entre [0,255]. Para ello se han usado dos funciones, que al fin y al cabo son lo mismo:

Por ampliación se ha implementado **supressNegativeValues()** que recibe la imagen y obtiene el mínimo valor negativo que existe en dicha imagen para posteriormente sumárselo a los demás valores de la imagen independientemente del canal en el que se encuentre. De esta forma nos aseguramos que no existe ningún valor que sea negativo. Se obtendrá un tono más grisáceido en la imagen de alta frecuencia:

```
/**
* Aumenta la intensidad de los pixeles de todos los canales sumando el valor mínimo negativo existente  que se ha generado durante la obtención la imagen de alta frecuencia para mostrar de forma más clara en el canvas.
*/
void suppressNegativesValues(Mat &img){
	double min_value = 0, value;
	int total_channels = img.channels();
	for(int i=0; i < img.cols; i++){
		for(int j=0; j < img.rows; j++){
			if(total_channels > 1){
				for(int k=0; k < total_channels; k++){
					value = img.at<Vec3d>(j, i)[k];
					if (value < min_value) min_value = value;
				}
			}
			else
			{
				value = img.at<double>(j, i);
				if (value < min_value) min_value = value;
			}
		}
	}
	if(min_value < 0) img = sumValueToMat(img, min_value);
}
```

La función auxiliar para sumar el valor mínimo encontrado al resto de valores de la imagen es **sumValueToMat()**:

```
/**
* Devuelve la imagen resultante de la suma de todos los valores de la imagen @img y el valor establecido  en @value para todos los canales de dicha imagen.
*/
Mat sumValueToMat(Mat img, double value){
	Mat aux = img;
	int total_channels = aux.channels();
	for(int i=0; i < aux.cols; i++){
		for(int j=0; j < aux.rows; j++){
			if(total_channels > 1){
				for(int k=0; k < 3; k++){
					aux.at<Vec3d>(j, i)[k] += (-value);
				}
			}
			else
			{
				aux.at<double>(j, i) += (-value);
			}
		}
	}
	return aux;
}
```

Esta función simplemente accede al valor del pixel que hay en cada canal y suma su valor más lo que hay en la variable value.

La otra función, perteneciente a OpenCV es **normalize()** que devuelve una imagen igual a la que se obtiene con las funciones anteriores. La forma correcta de llamar a esta función es:

```
normalize(high_freq_img, high_freq_img, 0, 255, NORM_MINMAX, CV_64F);
```

donde el primer parámetro es la imagen que se va a normalizar, el segundo parámetro donde se guardará la imagen resultante, los dos siguientes parámetros el rango de valores donde queremos los datos de la imagen, en este caso de 0 a 255, el siguiente parámetro el tipo de normalización y el último el tipo de imagen.

(En el código se ha usado la función implementada supressNegativeValues() pero para comprobar que la de OpenCV tiene el mismo resultado puede cambiarse a la hora de llamar a la función hybrid_images() IMPLEMENTACION por OPENCV en el main() del programa).

Una vez normalizada la imagen de alta frecuencia se guarda en la imagen híbrida la suma de los valores de la imagen de baja y alta frecuencia y se divide entre dos, realmente se

hace una media de sus valores. Luego se convierte a un tipo de imagen que OpenCV pueda mostrar en la ventana.

Finalmente se guarda en un vector Mat la imagen de baja frecuencia, alta frecuencia y la imagen híbrida, que se pasará como parámetro en la función **showHybridImagesCanvas()** que es la que muestra el canvas de las tres imágenes:

```
/**
* Muestra el canvas de las imágenes de baja y alta frecuencia junto con la resultante de
* juntar ambas, es decir, la imagen híbrida.
*/
void showHybridImagesCanvas(vector<vector<Mat> > imgs){
	vector<Mat> hybridImgs;
	hybridImgs.resize(imgs.size());

	for(unsigned i=0; i<imgs.size(); i++){
		hconcat(imgs[i][LOW_FREQUENCY], imgs[i][HIGH_FREQUENCY], hybridImgs[i]);
	hconcat(hybridImgs[i], imgs[i][HYBRID_IMG], hybridImgs[i]);
	}
	showImages(hybridImgs);
}
```

A continuación se van a mostrar dos ejemplos de imágenes híbridas.

Pájaro (sigma 3) y avión (sigma 5)

Einstein (sigma 3) y Marilyn (sigma 4)

Pirámide Gaussiana

Para la generación de la pirámide Gaussiana de una imagen se ha implementado la función **gaussianPyramid()** que recibe una imagen y el número de niveles de la pirámide como parámetro.

```
/**
* Dada una imagen híbrida @img y el número de niveles @levels obtener la pirámide Gaussiana
* de dicha imagen con los niveles dados. Se trata de un submuestreo y concatenación de
imágenes.
*/
Mat gaussianPyramid(Mat img, int levels){
	Mat pyramid = img, aux = img, tmp;
	for (int i = 0; i<levels; i++){
		pyramidDown(aux);
		tmp = Mat::zeros(img.rows - aux.rows, aux.cols, aux.type());
		vconcat(aux, tmp, tmp);
		hconcat(pyramid, tmp, pyramid);
	}
	return pyramid;
}
```

Primero realizamos un bucle de tantas iteraciones como niveles vaya a tener la pirámide, a continuación obtendremos la imagen una cuarta parte más pequeña, que esto se hace en la función implementada **pyramidDown()**.

```
/**
* Función que reemplaza a la de OpenCV pyrDown(). Primero genera la imagen @img alisada con
sigma=1
* y a ésta le realiza un subsampling guardandolo de nuevo en img.
*/
void pyramidDown(Mat &img){
	Mat img_smooth = gaussianFilter(img);
	img = subsampling(img_smooth);
}
```

La función lo primero que hace es alisar la imagen con la función **gaussianFilter()** explicada anteriormente con valor de sigma 1.0 (es el valor que define por defecto la función, por eso no aparece como parámetro) para después realizar el submuestreo con **subsampling()**.

```
/**
* Dada una imagen @img devuelve la misma con un tamaño 1/4 más pequeño
*/
Mat subsampling(Mat img){
	int total_channels = img.channels();
	Mat aux(img.rows/2, img.cols/2, img.type());
	img.convertTo(img, CV_64F);
```

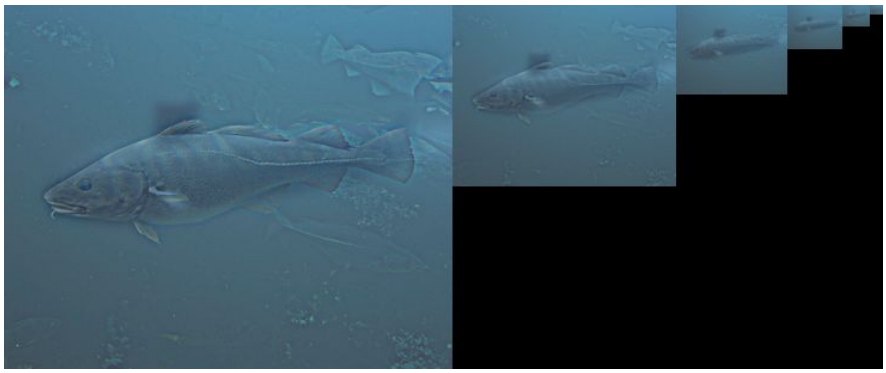
```
        aux.convertTo(aux, CV_64F);
        for(int i=0; i < aux.cols; i++){
                for(int j=0; j < aux.rows; j++){
                        if(total_channels > 1){
                                for(int k=0; k < total_channels; k++){
                                        aux.at<Vec3d>(j,i)[k] = img.at<Vec3d>(j*2, i*2)[k];
                                }
                        }
                        else aux.at<double>(j,i) = img.at<double>(j*2, i*2);
                }
        }
        aux.convertTo(aux, CV_8UC3);
        return aux;
}
```

Se crea una imagen una cuarta parte más pequeña y ésta se recorre y se le asigna valores a la posición de cada pixel eliminando las filas y columnas pares de la imagen original, lo que sería fila/columna si fila/columna no. Todo esto para cada canal que tenga la imagen.

Terminada la función **pyramidDown()** ya tenemos la nueva imagen que la juntamos a la derecha de otra que está generada únicamente por ceros, y a ésta resultante la juntamos con la generada anteriormente del submuestreo y así sucesivamente hasta la iteración final que depende de la variable **levels.**

Las siguientes imágenes son el resultado de aplicar una pirámide Gaussiana de nivel 5 a dos imágenes híbridas diferentes:

Marilyn (sigma 3) y Einstein (sigma 4)



Submarino (sigma 3) y pez (sigma 4)