VC: Informe práctica 1

Gustavo Rivas Gervilla

Vamos a dividir el informe en distintas secciones dedicadas a explicar lo que hemos hecho para completar cada una de las tareas que contenía esta práctica:

A. La convolución.

calcularVectorMascara:

Lo primero que tenemos que hacer es crear la máscara. Vamos a aprovechar que estamos muestreando una Gaussiana que como sabemos es separable y simétrica. Por lo tanto lo que vamos a construir va a ser una máscara 1D que utilizaremos tanto para las filas como para las columnas.

El tamaño de la máscara vendrá en función del σ y será el siguiente: $round(3\sigma)2 + 1$ esto se debe a que queremos muestrear dentro del intervalo $[-3\sigma, 3\sigma]$ y ponemos así el redondeo para poder tener máscaras impares de cualquier tamaño. Si por ejemplo hiciésemos $3ceil(\sigma)2 + 1$ el tamaño mínimo de las máscaras sería de 7. Con lo cual hemos de hacer el cálculo como hemos dicho. Como vemos lo hacemos de modo que obtengamos máscaras de orden impar.

El siguiente cálculo importante que tenemos que hacer es el del paso de muestreo, pretendemos que el mayor peso recaiga sobre el píxel central con lo cual el tamaño del paso será: $\dfrac{6\sigma}{longitud - 1}$ y así también aseguramos muestrear en los extremos del intervalo relevante que hemos señalado antes.

Ya lo único que queda es muestrear la Gaussiana de párametro σ y dividir cada componente por la suma de todos para que la máscara sume uno, una condición de los filtros de alisamiento.

```
Mat calcularVectorMascara(float sigma, float(*f)(float, float)) {
        /*
        Vamos a calcular el numero de pixeles que conformaran el vector mascara.
        En primer lugar recordemos que necesitamos tener una mascara de orden impar, de modos
        Como sabemos para quedarnos con la parte significativa de la gaussiana tenemos que m
        Ahora bien, podemos obtener numero decimales con lo que tendremos que redondear. Si h
        Con lo cual vamos a optar por hacer un redondeo de la forma round(3*sigma)*2+1 y este
        estariamos en una situacion similar a la anterior, mascaras de 1, luego de 7, sin nin
        Dejamos el 2 fuera del round para asegurar de obtener una longitud impar, como querem
        */

        int longitud = round(3 * sigma) * 2 + 1;
        int centro = (longitud - 1) / 2; // <-- elemento del centro del vector

        /*
        Ahora vamos a calcular el tamanio del paso de muestre, como vamos a ir muestreando.
        Queremos que el mayor peso lo tenga el pixel central, con lo cual al este pixel le do
        En consecuencia el paso sera paso = 6sigma/(longitud-1).
        */

        float paso = 6 * sigma / (longitud - 1);

        // Creamos la imagen que contendra los valores muestreados.
```

```
        Mat mascara = Mat(1, longitud, CV_32F);

        //Cargamos los valores en la mascara
        for (int i = 0; i <= centro; i++) {
                mascara.at<float>(0, i) = f(-paso * (centro - i), sigma);
                mascara.at<float>(0, longitud - i - 1) = f(paso * (centro - i), sigma);
        }

        //Y ahora dividimos por la suma de todos para que los elementos sumen 1.

        float suma = 0.0;

        for (int i = 0; i < mascara.cols; i++)
                suma += mascara.at<float>(0, i);

        mascara = mascara / suma;

        return mascara;
}
```

Como vemos le pasamos como parámetro un puntero a una función que será de la que muestreemos, esto lo hemos hecho así para poder reutilizar el código para cuando tengamos que muestrear otras funciones, como haremos en el Bonus1. En nuestro caso la función es:

```
float f(float x, float sigma) {
        return exp(-0.5 * x * x / sigma / sigma);
}
```

obtenerVectorOrlado:

Como sabemos la convolución, al estar trabajando con un núcleo separable, se realiza en filas y en columnas por separado. Pues bien lo que hace esta función es tomar una matriz 1D y prepararla para realizar sobre ella la convolución. Es decir, la orlamos añadiendo a sus extremos tantos píxeles como sean necesarios para poder convolucionar correctamente.

En concreto si nos situamos en los extremos de la fila/columna a convolucionar sobrarán los píxeles de la máscara a uno de los lados del píxel central, con lo cual la cantidad de píxeles a añadir es: mascara.cols - 1.

Una vez orlada la fila/columna a convolucionar con esta cantidad de píxeles (la mitad a cada lado) no tenemos más que rellenarlos o bien poniéndolos a cero o en modo espejo, según elijamos con los parámetros.

```
Mat obtenerVectorOrlado1C(Mat &senal, Mat &mascara, int cond_contorno) {
        //Añadiremos digamos a cada lado del vector (longitud_senal - 1)/2 pues son los pixe
        //Nosotros vamos a trabajar con vectores fila, pero no sabemos como sera senal con lo
        Mat copia_senal;

        //Vamos a trabajar siempre con vectores fila.
        if (senal.rows == 1)
                copia_senal = senal;
        else if (senal.cols == 1)
                copia_senal = senal.t();
        else
                cout << "Senal no es un vector fila o columna.\n";
```

```
	int colsParaCopia = copia_senal.cols;

	int pixels_extra = mascara.cols - 1; //<-- numero de pixeles necesarios para orlar.

	int colsVectorOrlado = colsParaCopia + pixels_extra;

	Mat vectorOrlado = Mat(1, colsVectorOrlado, senal.type());

	int inicio_copia, fin_copia; // <-- posiciones donde comienza la copia del vector, ce

	inicio_copia = pixels_extra/2;
	fin_copia = colsParaCopia + inicio_copia;

	//Copiamos senal centrado en vectorAuxiliar

	for (int i = inicio_copia; i < fin_copia; i++)
		vectorOrlado.at<float>(0, i) = copia_senal.at<float>(0, i-inicio_copia);

	// Ahora rellenamos los vectores de orlado segun la tecnica que hayamos elegido;
	// Hacemos el modo espejo solo que si la opcion elegida es cero entonces lo multiplic

	for (int i = 0; i < inicio_copia; i++) {
		vectorOrlado.at<float>(0, inicio_copia - i - 1) = cond_contorno * vectorOrlad
		vectorOrlado.at<float>(0, fin_copia + i) = cond_contorno * vectorOrlado.at<fl
	}

	return vectorOrlado;

}
```

calcularConvolucionVectores1C:

Una herramienta muy útil son las funciones *split* y *merge* de OpenCV que nos permiten obtener por separado cada uno de los canales de la imagen y luego juntar varios canales en una imagen (respectivamente). Con lo cual con el objetivo de reutilizar código y trabajar de la forma más estándar posible lo que haremos es escribir código para imágenes con un sólo canal, luego si tenemos una imagen con 3 canales en la función genérica los separamos, procesamos cada uno por separado (con la versión para un sólo canal) y los volvemos a unir. Por eso de ahora en adelante sólo nos centraremos en explicar el funcionamiento de las versiones 1C de las funciones.

En esta función lo único que hacemos es aplicar la operación de convolución a una matriz 1D. Teniendo en cuenta que es un vector orlado y con lo cual empezamos a hacer la convolución sólo en los píxeles que no son de la orla y aprovechando la función colRange para fijar un ROI en cada paso.

Observemos que la máscara está preparada para trabajar con vectores fila, esto es así porque simplemente con trasponer ya podemos trabajar con las columnas como si fuesen filas.

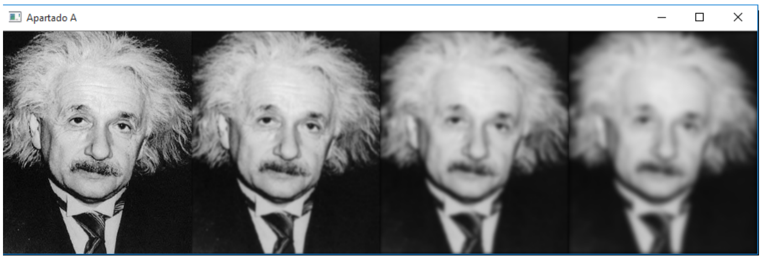
```
Mat calcularConvolucionVectores1C (Mat &senal, Mat &mascara, int cond_contorno) {
	//preparamos el vector para la convolucion orlandolo.
	Mat copiaOrlada = obtenerVectorOrlado1C(senal, mascara, cond_contorno);
	Mat segmentoCopiaOrlada;
	Mat convolucion = Mat(1, senal.cols, senal.type());
```

```
        int inicio_copia, fin_copia, long_lado_orla;
        //calculamos el rango de pixeles a los que realmente tenemos que aplicar la convoluci
        inicio_copia = (mascara.cols - 1)/2;
        fin_copia = inicio_copia + senal.cols;
        long_lado_orla = (mascara.cols - 1) / 2;

        for (int i = inicio_copia; i < fin_copia; i++) {
                //Vamos aplicando la convolucion a cada pixel seleccionando el segmento con e
                segmentoCopiaOrlada = copiaOrlada.colRange(i - long_lado_orla, i + long_lado_
                convolucion.at<float>(0, i - inicio_copia) = mascara.dot(segmentoCopiaOrlada)
        }

        return convolucion;
}
```

convolucion2D1C:

Aquí simplemente aplicamos la convolución a una imagen, primero a las filas y luego a las columnas. Con lo cual lo que hacemos es convolucionar las filas, trasponemos, convolucionamos las columnas (como si fuesen filas) y deshacemos la trasposición.

```
Mat convolucion2D1C(Mat &im, float sigma, int cond_bordes) {
        Mat mascara = calcularVectorMascara(sigma, f); //calculamos la mascara a aplicar
        Mat convolucion = Mat(im.rows, im.cols, im.type()); //matriz donde introducimos el re

        //Convolucion por filas
        for (int i = 0; i < im.rows; i++) {
                calcularConvolucionVectores1C(im.row(i), mascara, cond_bordes).copyTo(convolu
        }

        //Convolucion por columnas
        convolucion = convolucion.t(); //trasponemos para poder operar como si fuese por fila

        for (int i = 0; i < convolucion.rows; i++) {
                calcularConvolucionVectores1C(convolucion.row(i), mascara, cond_bordes).copyT
        }

        convolucion = convolucion.t(); //deshacemos la trasposicion para obtener el resultado

        return convolucion;
}

Mat convolucion2D(Mat &im, float sigma, int cond_bordes) {
        Mat convolucion;
        Mat canales[3];
        Mat canalesConvol[3];

        if (im.channels() == 1)
                return convolucion2D1C(im, sigma, cond_bordes);
        else if (im.channels() == 3) {
                split(im, canales);
                for (int i = 0; i < 3; i++)
                        canalesConvol[i] = convolucion2D1C(canales[i], sigma, cond_bordes);
                merge(canalesConvol, 3, convolucion);
        }
        else
                cout << "Numero de canales no valido." << endl;

        return convolucion;
}
```



Como podemos ver cuánto más grande es el sigma, y sobre todo cuando se aumenta el orden de la máscara la imagen se difumina más, esto se debe a que cada vez influyen más píxeles (cada vez más alejados del central) en el valor de uno.

B. Imágenes híbridas

El código es muy sencillo; como sabemos para obtener una imagen híbrida lo único que tenemos que hacer es obtener las frecuencias bajas de una imagen (aplicar un filtro de alisamiento, con las funciones de convolución de la parte A) y obtener las altas de una imagen, restándole a la original sus frecuencias bajas.

```
Mat calcularImHibrida(Mat &im1, Mat &im2, float sigma1, float sigma2) {
        Mat bajas_frecuencias = convolucion2D(im1, sigma1, 0);
        Mat altas_frecuencias = im2 - convolucion2D(im2, sigma2, 0);

        return bajas_frecuencias + altas_frecuencias;
}

Mat calcularImHibrida(Mat &im1, Mat &im2, float sigma1, float sigma2, Mat &bajas_frecuencias ,
        bajas_frecuencias = convolucion2D(im1, sigma1, 0);
        altas_frecuencias = im2 - convolucion2D(im2, sigma2, 0);

        return bajas_frecuencias + altas_frecuencias;
}
```

Hemos hecho dos versiones para elegir si queremos o no queremos recuperar las imagenes de altas y bajas frecuencias para futuros cálculos y procedimientos con ellas.

Para cada pareja de imágenes habrá que ajustar adecuadamente los sigmas de modo que el efecto sea el mejor posible.

C. Pirámide Gaussiana

Aquí lo único que hacemos es aplicar el mecanismo básico para obtener las pirámides Gaussianas. Partimos de la imagen original como primer nivel, entonces para obtener un nivel lo que hacemos es alisar el anterior y submuestrearlo, esto es tomamos sólo las columnas y filas impares.

Lo que hay que tener en cuenta es el sigma para el filtro de alisamiento, dado que vamos saltanto las columnas pares no tiene sentido tener en cuenta lo que pase a más allá de un píxel de distancia del central. Por lo tanto vamos a tomar una máscara de tamaño tres y en consecuencia por los cálculos que hemos descrito anteriormente, tomamos un σ menor que 1, mayor o menor en función del peso que queramos darle a cada uno de los píxeles que intervendrán en la convolución.

```
Mat submuestrear1C(Mat &im) {
	int colOriginal = im.cols;
	int filOriginal = im.rows;

	Mat submuestreado = Mat(filOriginal / 2, colOriginal / 2, im.type());

	for (int i = 0; i < submuestreado.rows; i++)
		for (int j = 0; j < submuestreado.cols; j++)
			submuestreado.at<float>(i, j) = im.at<float>(i*2+1, j*2+1);

	return submuestreado;
}

void calcularPirGaussiana1C(Mat &im, vector<Mat> &piramide, int numNiveles) {

	piramide.push_back(im);

	for (int i = 1; i < numNiveles; i++)
		piramide.push_back(submuestrear1C(convolucion2D1C(piramide.at(i-1), 0.5, 0)))
}

void calcularPirGaussiana(Mat &im, vector<Mat> &piramide, int numNiveles) {
	Mat canalesIm[3];
	Mat canalesNivel[3];
	vector<Mat> canalesPiramide[3];

	if (im.channels() == 1)
		calcularPirGaussiana1C(im, piramide, numNiveles);
	else if (im.channels() == 3) {
		piramide.resize(numNiveles);
		split(im, canalesIm);

		for (int i = 0; i < 3; i++)
			calcularPirGaussiana1C(canalesIm[i], canalesPiramide[i], numNiveles);

		for (int i = 0; i < numNiveles; i++) {
			for (int j = 0; j < 3; j++)
				canalesNivel[j] = canalesPiramide[j].at(i);

			merge(canalesNivel, 3, piramide.at(i));
		}

	}
	else
		cout << "Numero de canales no valido." << endl;
```

```
}
```

Aquí podemos ver el efecto que se busca en las imágenes híbridas. En esta ocasión considero que he conseguido un efecto bastante bueno tras probar varios sigmas y decidir cuál de las dos imágenes usar para las frecuencias bajas y cuales las altas. Este ha sido el resultado final. En cambio por ejemplo para la híbrida entre Einstein y Marilyn ha no he logrado que quedase bien, siempre había algunas frecuencias que resaltaban y no debían resaltar.

Bonus 1

Como ya hemos visto en teoría la forma más eficiente de implementar estas máscaras es usar la separabilidad y convolucionar por separado las filas y las columnas. Entonces lo que hemos hecho es simplemente funciones para obtener estas máscaras, evidentemente las parciales son simétricas, esto es, por ejemplo la primera parcial respecto x e y son iguales intercambiando los papeles de x e y.

Por lo tanto solo hemos hecho una versión de cada función, si queremos usar estas máscara lo único que tendríamos que hacer es modificar un poco el código de Convolucion2D para que admita dos máscaras en los párametros, una para las filas y otra para las columnas.

Como vemos obtenemos matrices filas pero ya hemos visto cómo usar matrices filas para todo sin más que trasponer.

```
//Declaramos las funciones que intervienen en las mascaras y de las que por tanto muestrearem

float parte1PrimeraDerivada (float x, float sigma) {
        return 1/(2*M_PI*sigma*sigma)*(-x/(sigma*sigma))*exp(-(x*x)/(2*sigma*sigma));
}

float parte2PrimeraDerivada(float x, float sigma) {
        return exp(-(x*x) / (2 * sigma*sigma));
}

float parte1SegundaDerivada(float x, float sigma) {
        return (1 / (2 * M_PI*sigma*sigma)) * (((-1 / (sigma*sigma))*exp(-(x*x) / (2 * sigma*
```

```
}

float parte2SegundaDerivada(float x, float sigma) {
        return exp(-(x*x) / (2 * sigma*sigma));
}

void calcularMascarasPrimeraDerivada(float sigma, Mat &parte1, Mat &parte2) {
        parte1 = calcularVectorMascara(sigma, parte1PrimeraDerivada);
        parte2 = calcularVectorMascara(sigma, parte2PrimeraDerivada);
}

void calcularMascarasSegundaDerivada(float sigma, Mat &parte1, Mat &parte2) {
        parte1 = calcularVectorMascara(sigma, parte1SegundaDerivada);
        parte2 = calcularVectorMascara(sigma, parte2SegundaDerivada);
}
```