INSTITUTO POLITECNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

“Practica 4 - Filtros”

-Tinoco Videgaray Sergio Ernesto

Grupo: 5BV1

Materia: Visión Artificial

03/11/22

- Introducción

En el área del procesamiento de digital de imágenes existen múltiples herramientas que nos permiten aplicar filtros a las imágenes para destacar algún elemento dentro de la imagen o incluso para mejorar la calidad de la imagen presentada, la cual podría verse perjudicada por algún tipo de ruido (sal y pimienta, gaussiano, etc.).

Muchos de estos filtros utilizan un elemento matemático conocido como Mascara de Filtro o Kernel, dicho Kernel se define como una matriz N dimensional que almacena los coeficientes que van a afectar a cada píxel dentro de la imagen. Este Kernel se va a definir según el tipo de filtro que se desee realizar.

En esta practica se va a implementar un Kernel Gaussiano de NxN en el lenguaje de programación C++.

Para ello se va a trabajar con una imagen en niveles de gris a la que se le van a aplicar bordes de tamaño N/2 con el fin de que el filtro se pueda aplicar a los bordes de la imagen contenida.

Utilizando la fórmula para el Kernel Gaussiano de 2 dimensiones (x, y) como se muestra en la imagen 1.1, se va a desarrollar un algoritmo que genere la matriz para el Kernel considerando un valor para sigma σ=1:

$$g(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Imagen 1.1

Ecuación del filtro Gaussiano

- Desarrollo

Se importan las bibliotecas correspondientes.

```
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <iostream>
#define _USE_MATH_DEFINES
#include <math.h>
```

Se definen las constantes a utilizar

```
const double e = 2.71828182845904523536;
const double pi =3.14159265358979311600;
const double c = 1 / (2 * pi);
```

Se programa la función para el Kernel Gaussiano considerando σ=1

```
double** crearKernel(int N, double sigma)
{
    double** kernel = new double*[N];   //Reservo memoria para el kernel
    int m = N/ 2;   //Mitad del tamano del kernel
    c /= (sigma * sigma);

    for (int i = 0;i < N;i++)
    {
        kernel[i] = new double[N];  //reservo memoria en la i-esima
posicion del kernel
        for (int j = 0;j < N;j++)
        {
           double x = j - m;
           double y = (i - m) * -1;
           kernel[i][j]= c * pow(e, (-((x * x) + (y * y)) / (2* (sigma *
sigma))));    //Aplico la funcion Gaussiana
            //cout << kernel[i][j] << "\t";
        }
        //cout << "\n";
    }
```

```
    return kernel;
}
```

Se define la función para aplicar los bordes a la imagen.

```
Mat aplicarBordes(Mat imagen, int N,int borde)
{

    int rows = imagen.rows; //Numero de filas
    int cols = imagen.cols; //Numero de columnas

    Mat imagenG(rows + borde, cols + borde, CV_8UC1);   //Imagen en niveles de gris
    Mat imagenB(rows + borde, cols + borde, CV_8UC1); //Imagen resultante con bordes

    cvtColor(imagen, imagenG, COLOR_BGR2GRAY);//Obtiene la matriz de niveles de gris

    double niv_gris;

    for (int i = 0; i < rows + borde; i++) {
        for (int j = 0; j < cols + borde; j++) {

     //En caso de que i o j supere el numero de filas o sea inferior al borde
     if ((i >= rows || i < borde) || (j >= cols || j < borde)) {
     //Se coloca un nivel 0 en la i,j-esima posicion de la matriz resultante
                imagenB.at<uchar>(Point(i, j)) = uchar(0);
            }
            else {  //En caso contrario
                niv_gris = imagenG.at<uchar>(Point(j - borde, i - borde));
     //Coloca el nivel de gris correspondiente a la imagen original
                imagenB.at<uchar>(Point(j, i)) = uchar(niv_gris);
            }

        }
    }

    return imagenB;
}
```

Se define una función para aplicar la convolución con el Kernel

```
int aplicarConvolucion(Mat imagenBordes, double** kernel,int borde, int x, int y)
{
    int b = (2 * borde) + 1;
    int valor = 0;

    for (int i = 0; i < b; i++) {
        for (int j = 0; j < b; j++) {

     valor += int((imagenBordes.at<uchar>(Point(y - borde + i, x - borde + j)))*
(kernel[i][j]));

        }
    }

    return valor;
}
```

Se declara la función para aplicar el filtro gaussiano en la imagen con bordes.

```
Mat aplicarFiltro(Mat imagenBordes, double** kernel,int borde)
{
    int rows = imagenBordes.rows;   //Numero de filas
    int cols = imagenBordes.cols; //Numero de columnas
    Mat imagenR(rows, cols , CV_8UC1);  //Imagen Resultante


    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols ; j++) {
            //Si el valor del pixel es diferente de 0 se aplica la convolucion
            if (imagenBordes.at<uchar>(Point(i, j)) != uchar(0))
            {
             imagenR.at<uchar>(Point(i, j)) =
uchar(aplicarConvolucion(imagenBordes,kernel,borde,j,i));
            }
        }
```

```
    }

    return imagenR;
}
```

Se declaran las variables para las imágenes:

```
    char NombreImagen[] = "lena.png";
    int N=0,borde;
    double sigma;
    Mat imagen, imagenBordes,imagenResultante;
```

Se carga la imagen con la que se va a trabajar.

```
imagen = imread(NombreImagen);

    if (imagen.empty())
    {
        cout << "Error al cargar la imagen: " << NombreImagen <<
endl;
        return -1;
    }
```

Se solicita al usuario ingresar un valor para N, y en caso de que el valor sea par se solicitara al usuario ingresar un valor impar:

```
cout << "Ingrese el valor de N\n";
    cin >> N;

    while (N%2==0)
    {
        cout << "Por favor ingresa un valor impar para N\n";
        cin >> N;
    }
```

Se solicita al usuario ingresar un valor para sigma, y en caso de que el valor sea 0 se solicitara al usuario ingresar un valor distinto:

```
cout << "Ingrese el valor de Sigma\n";
    cin >> sigma;

    while (sigma == 0)
    {
  cout << "Por favor ingresa un valor distinto de 0 para sigma\n";
   cin >> sigma;
    }
```

Se define el ancho del borde y se crea el filtro gaussiano.

```
borde = N / 2;
double **kernel=crearKernel(N,sigma);   //Creo el Kernel Gaussiano
```

Se genera la imagen con bordes y se aplica el Kernel generado a esa imagen.

```
imagenBordes = aplicarBordes(imagen,N,borde);  //Genera el borde
imagenResultante = aplicarFiltro(imagenBordes, kernel, borde);//Aplica el
filtro
```

Se muestran las imágenes generadas.

```
    imshow("Imagen Original", imagen);
    imshow("Imagen con bordes", imagenBordes);
    imshow("Imagen resultante", imagenResultante);
```

- Resultados

Kernel de 7x7 con sigma = 1.



Kernel de 5x5 con sigma = 1.5



Imagen Original. *Imagen con bordes.* *Imagen filtrada*

Conclusiones.

El filtrado Gaussiano es una herramienta de mucha utilidad cuando se necesita mejorar la calidad de nuestra imagen en caso de que presente algún tipo de imperfección como el ruido Gaussiano. Lo que nos ayuda a eliminar este tipo de imperfecciones sin que se pierdan elementos importantes de la imagen con la cual vamos a trabajar en etapas posteriores del procesamiento. También es de mucha utilidad este filtrado en caso de querer aplicar algún tipo de difuminado a nuestra imagen.

También cabe destacar la utilidad de los bordes generados en la imagen puesto que resulta un factor importante a la hora de aplicar filtros que necesitan aplicar una convolución en las esquinas de la imagen.