# Python's eyes

Vision, imaging & visualization…

ß Menú

## Etiqueta: deteccion de bordes

## Filtros para la deteccion de bordes de una imagen con Python 3 (Parte 2)

28 julio, 201729 agosto, 2017 · Deja un comentario ·

*Consulta la primer parte aqui (https://pythoneyes.wordpress.com/2017/07/28/filtros-para-la-deteccion-de-bordes-de-una-imagen-con-python-3-parte-1/)

### Los operadores Prewitt y Sobel

Estos operadores representan dos de los metodos mas usados en la deteccion de bordes, los cuales son muy similares entre si.

# Los filtros

Ambos operadores utilizan como filtro una matriz de coeficientes de 3×3, que facilita la posibilidad de configurarlo de tal forma que el filtro no sea tan vulnerable al ruido propio de la imagen, en comparacion al filtro presentado en el post anterior.

El operador Prewitt utiliza el filtro definido por:

$$Px = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

$$Py = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

El cual es evidentemente aplicado sobre los diferentes vecinos del pixel en cuestion.

El operador de Sobel tiene un filtro practicamente identico al Prewitt con la unica diferencia que en este filtro se le da un mayor peso al renglon o columna central del filtro. La matriz de coeficientes para este operador es definida como:

$$Sx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$Sy = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Los resultados de los filtros de Prewitt y Sobel producen estimaciones del gradiente local para todos los pixeles de la imagen en sus dos direcciones, manteniendo la siguiente relacion:

$$\bigtriangledown I(x,y) \approx \frac{1}{6} \begin{bmatrix} Px \cdot I \\ Py \cdot I \end{bmatrix}$$

Para Prewitt

$$\bigtriangledown I(x,y) \approx \frac{1}{8} \begin{bmatrix} Sx \cdot I \\ Sy \cdot I \end{bmatrix}$$

Para Sobel

Lo anterior no es otra cosa mas que un sexto de la convolucion en sentido horizontal y vertical para Prewitt y un octavo de la convolucion en ambos sentidos para el caso de Sobel.

Para dar el resultado final ambos se relacionan igual que en el caso del metodo de la derivada (raiz cuadrada de la suma de los cuadrados de convoluciones horizontal y vertical pero en este caso divididas por su factor correspondiente como se mostro anteriormente):

$$|\bigtriangledown I| = \sqrt{(\frac{Px \cdot I}{6})^2 + (\frac{Py \cdot I}{6})^2}$$

Magnitud del gradiente para Prewitt

$$|\bigtriangledown I| = \sqrt{(\frac{Sx \cdot I}{8})^2 + (\frac{Sy \cdot I}{8})^2}$$

Magnitud del gradiente para Sobel

Como se menciono en el post anterior debido a que utilizaremos el modulo ImageFilter de PIL se aplicara dos veces el proceso, uno con los coeficientes tal y como fueron definidos anteriormente y otra vez para los mismos coeficientes pero con signo contrario, esto es necesario debido a la forma en la que trabaja internamente este modulo.

Codificado en Python 3 para ambos casos:

```
import sys
from PIL import Image, ImageFilter

#apertura de la imagen original
imagen = Image.open('nombre_y_ruta_del_archivo).convert('L')

def detector_bordes(tipo):

    if tipo == 'Prewitt':

        factor = 6

        coeficientes_h = [-1, 0, 1, -1, 0, 1, -1, 0, 1]
        coeficientes_v = [-1, -1, -1, 0, 0, 0, 1, 1, 1]

        #coeficientes con signo contrario
        coeficientes_h1 = [1, 0, -1, 2, 0, -2, 1, 0, -1]
        coeficientes_v1 = [1, 2, 1, 0, 0, 0, -1, -2, -1]

    elif tipo == 'Sobel':

        factor = 8

        coeficientes_h = [-1, 0, 1, -2, 0, 2, -1, 0, 1]
        coeficientes_v = [-1, -2, -1, 0, 0, 0, 1, 2, 1]

        #coeficientes con signo contrario
        coeficientes_h1 = [1, 0, -1, 2, 0, -2, 1, 0, -1]
        coeficientes_v1 = [1, 2, 1, 0, 0, 0, -1, -2, -1]

    else:
        #en caso de no introducir el nombre correctamente se cierra el script
        sys.exit(0)

    datos_h = imagen.filter(ImageFilter.Kernel((3,3), coeficientes_h, factor)).getdata()
    datos_v = imagen.filter(ImageFilter.Kernel((3,3), coeficientes_v, factor)).getdata()

    datos= []


    for x in range(len(datos_h)):

        datos.append(round(((datos_h[x] ** 2) + (datos_v[x] ** 2)) ** 0.5))

    datos_h = imagen.filter(ImageFilter.Kernel((3,3), coeficientes_h1, factor)).getdata()
    datos_v = imagen.filter(ImageFilter.Kernel((3,3), coeficientes_v1, factor)).getdata()

    datos_signo_contrario = []


    for x in range(len(datos_h)):

        datos_signo_contrario.append(round(((datos_h[x] ** 2) + (datos_v[x] ** 2)) ** 0.5))


    datos_bordes = []


    for x in range(len(datos_h)):

        datos_bordes.append(datos[x] + datos_signo_contrario[x])

    return datos_bordes
```

```
datos_bordes = detector_bordes('Prewitt')

#linea para hacer la deteccion con Sobel:
#datos_bordes = detector_bordes('Sobel')

nueva_imagen = Image.new('L', imagen.size)
nueva_imagen.putdata(datos_bordes)

#guardar el resultado
nueva_imagen.save('ruta_y_nombre_del_archivo')


#cerrar los objetos de la clase Image
imagen.close()
nueva_imagen.close()
```

Imagen original

Bordes detectados por Prewitt

Bordes detectados por Sobel

Es recomendable realizar una etapa de pre-procesamiento o post-procesamiento como se menciono en la primer parte del post ya que la salida directa de los algoritmos tiende a ser muy tenue.

Pre-procesamiento con mejora de Nitidez + Linearizacion del Histograma y deteccion de bordes por Sobel

Por ultimo cabe mencionar que el operador Sobel es a razon de sus buenos resultados y facilidad de implementacion muy utilizado e implementado en la mayoria de los paquetes de software comerciales utilizados para el procesamiento digital de imagenes.

Peace

-Raziel

# Filtro de Laplace con Python 3 (realce de bordes en imagenes)

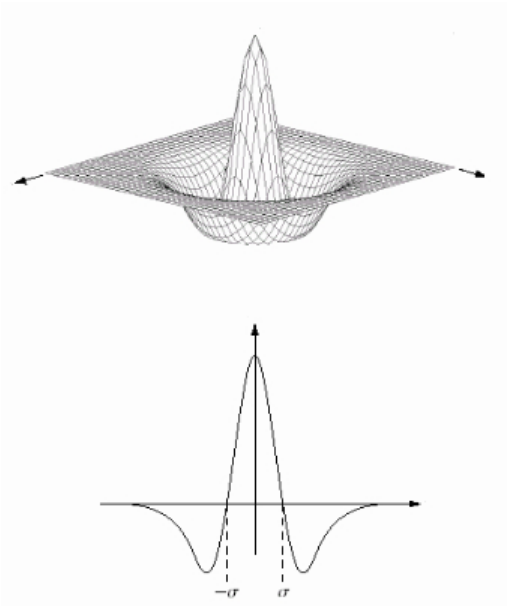
20 junio, 20178 agosto, 2017 · 2 comentarios ·

Si un filtro contiene como parte de sus coeficientes numeros negativos, su operacion puede interpretarse como la diferencia de dos diferentes sumas: La suma de todas las combinaciones lineales de los coeficientes positivos del filtro menos la suma de todas las combinaciones lineales debidas a los coeficientes negativos.

La funcion del filtro Laplaciano queda definida por:

$$M_\sigma(x,y) = \frac{1}{\sqrt{2\pi\sigma^3}}\left(1 - \frac{x^2+y^2}{\sigma^2}\right)e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Curiosamente esta funcion es conocida como «Mexican hat» (o sombrero Mexicano) dada la forma tan parecida de su grafica resultante con el sombrero tipico Mexicano:




El siguiente kernel 5×5 que se presenta pertenece al filtro de Laplace, el cual, realiza la diferencia entre el punto central (unico coeficiente positivo con valor de 16) y la suma negativa de 12 coeficientes con valores de -1 y -2. Los coeficientes restantes valen cero y no son considerados en el procesamiento:

$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

Mientras que los filtros con coeficientes positivos tienen el efecto de suavizado, en los filtros de diferencia (que tienen coeficientes positivos y negativos) el efecto es el contrario, donde las diferencias entre niveles de intensidad de pixel son realzadas. Por consiguiente, este tipo de filtros son utilizados normalmente para el realce de bordes, para su deteccion.

Codigo para implementar el filtro de Laplace mediante la definicion del kernel:

```
from PIL import Image, ImageFilter

tamaño = (5,5)

coeficientes = [0, 0, -1, 0, 0, 0, -1, -2, -1, 0, -1, -2, , 16, -2, -1, 0, -1, -2, -1, 0, 0, 0, -1,
0, 0]

factor = 1

imagen_original = Image.open('ruta_y_nombre_del_archivo')

imagen_procesada = imagen_original.filter(ImageFilter.Kernel(tamaño, coeficientes, factor))

#se graba el resultado

imagen_procesada.save('ruta_y_nombre_del_archivo')

#se cierran ambos objetos creados de la clase Image

imagen_original.close()

imagen_procesada.close()
```

Utilizando como imagen original el logo del blog:

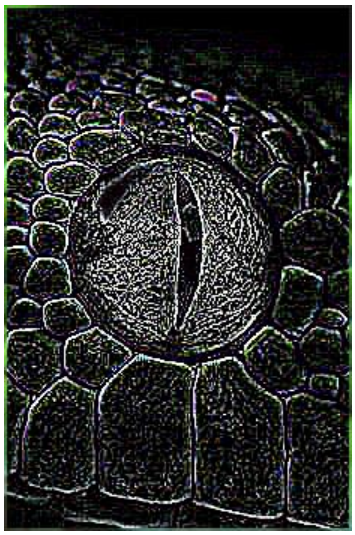


Imagen resultante del filtro
Laplaciano en imagen a color

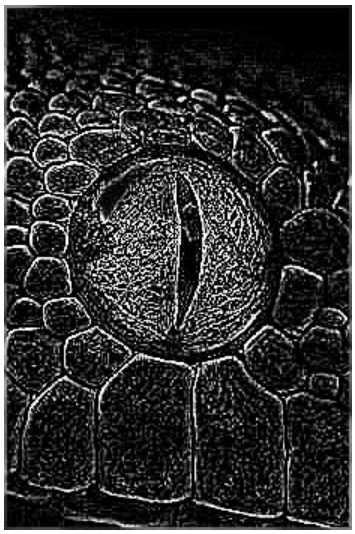


Imagen resultante de filtro Laplaciano en imagen a escala de grises

Resulta interesante ver que al aplicar este algoritmo en una imagen a color el resultado sea practicamente el mismo que el que se obtiene a partir de una imagen a escala de grises por lo que como regla general se debe calcular este filtro siempre con la imagen ya convertida a escala de grises (consulta aqui (https://pythoneyes.wordpress.com/2017/05/22/conversion-de-imagenes-rgb-a-escala-de-grises/) como hacer la conversion) para descartar la informacion del color y ahorrar tiempo al disminuir el costo de procesamiento.

Peace

-Raziel