Version 12/11/18

Computer Architecture

Lab 4 : Gaussian filter

Fernando.Rincón@uclm.es
Serafin.Benito@uclm.es

Contents

- Gaussian blur
- Gaussian.cpp
- Separability
- Task: part 1
- Task: part 2

Gaussian Blur

- Filter that
  - removes noise from the image
  - Also blurs it
- Some advantages when compared to other filters

Gaussian Blur

- Obtained from the convolution product of the gaussian bidimensional function:

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

with image

$$I(x,y)$$

Discrete approximation

- Images are discrete sets of pixels
- Discrete approximation of the gaussian function:
  - Using a matrix
  - Convolution product of the matrix with the pixels in the image (just as in Sobel)
  - The size and coefficients of the matrix depend on standard deviation, σ
  - With higher σ:
    - Bigger size of the matrix
    - Bigger blur effect

Discrete approximation

- In this lab we'll use this matrix:

$$\frac{1}{256}\times\begin{pmatrix}1 & 4 & 6 & 4 & 1\\ 4 & 16 & 24 & 16 & 4\\ 6 & 24 & 36 & 24 & 6\\ 4 & 16 & 24 & 16 & 4\\ 1 & 4 & 6 & 4 & 1\end{pmatrix}$$

- Note how weight of the neighbor pixels decrease with distance

Gaussian.cpp

- In this program:
  - The *kernel* ***is also applied to pixels in the borders***
  - In these points we can't apply the full *kernel*
  - We'll apply only parts overlapping the image

```
double naive_matrix(QImage* image, QImage* result) {

  double start_time = omp_get_wtime();

  for (int y = 0; y < height; y++)
    for (int x = 0; x < width ; x++)
      convolution(image, result, x, y);

  return omp_get_wtime() - start_time;
}
```

Gaussian.cpp

- Ex.: consider pixel in row 0 column 1 of the image and its neighbors

I(0,0)	**I(0,1)**	I(0,2)	I(0,3)	
I(1,0)	I(1,1)	I(1,2)	I(1,3)	
I(2,0)	I(2,1)	I(2,2)	I(2,3)	

- We'll apply the following part of the *kernel* (the 36 in the center must match the pixel considered)

$$\frac{1}{256} \times \begin{pmatrix} 24 & \mathbf{36} & 24 & 6 \\ 16 & 24 & 16 & 4 \\ 4 & 6 & 4 & 1 \end{pmatrix}$$

Gaussian.cpp

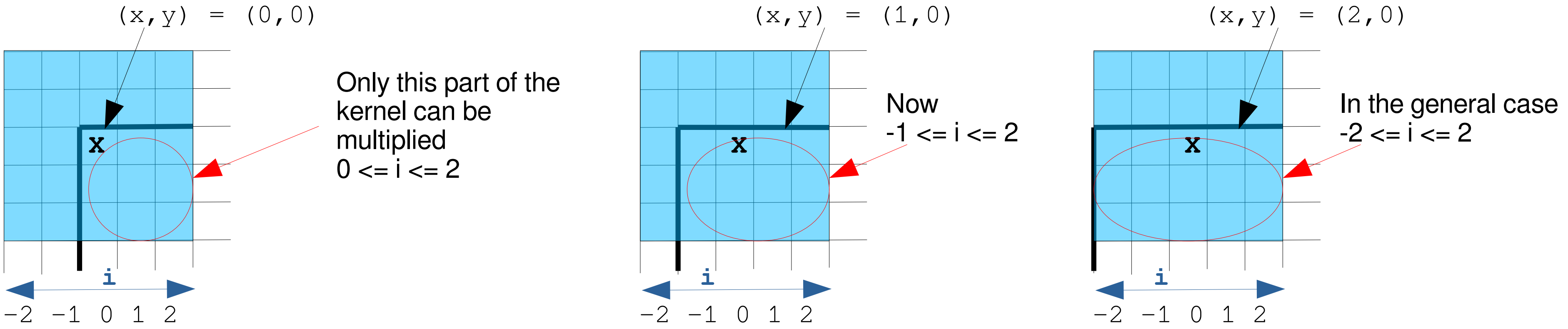
```
inline void convolution(QImage* image, QImage* result, int x, int y) {
  int i, j;
  int red, green, blue;
  int i_min, i_max, j_min, j_max;
  QRgb aux;

  red = green = blue = 0;

  i_min = max(-M, -x);
  i_max = min(M, width - x - 1);
  j_min = max(-M, -y);
  j_max = min(M, height - y - 1);

    for (j = j_min; j <= j_max ; j++)
      for (i = i_min; i <= i_max; i++) {

      int coef = getGaussCoefficient(i, j);
      aux = image->pixel(x + i, y + j);

      red += coef * qRed(aux);
      green += coef * qGreen(aux);
      blue += coef * qBlue(aux);
    };

  red /= 256; green /= 256; blue /= 256;
  result->setPixel(x, y, QColor(red, green, blue).rgba());
}
```

The limits are required for dealing with the borders

The loops perform the typical convolution product

Gaussian.cpp

- Why those limits?



Can you find the rule?

X=0 → 0<= i   X=1 → -1<= i   X>=2 → -2<= i

A similar reasoning can be applied to both the x superior limit and for the y's

Separability

- The following property

$$\frac{1}{256}\cdot\begin{pmatrix}1 & 4 & 6 & 4 & 1\\ 4 & 16 & 24 & 16 & 4\\ 6 & 24 & 36 & 24 & 6\\ 4 & 16 & 24 & 16 & 4\\ 1 & 4 & 6 & 4 & 1\end{pmatrix}=\frac{1}{256}\cdot\begin{pmatrix}1\\ 4\\ 6\\ 4\\ 1\end{pmatrix}\cdot\begin{pmatrix}1 & 4 & 6 & 4 & 1\end{pmatrix}$$

- Allows getting a faster version of the algorithm
  - Apply the vertical vector to each pixel in the image
  - Then apply the horizontal one to the result of the previous step

Task: part 1

- Add to Gaussian.cpp the following subprograms
  - convolution_1d_v. To apply the vertical convolution vector to one pixel
  - convolution_1d_h. To apply the horizontal one and divide by 256
  - gaussian_vectors. To call the previous one for each pixel and measure the resulting time
- **¡Be careful!** The output from convolution_1d_v can't be of type Qimage
  - Use a matrix per each color
  - Needs mallocs

Task: part 2

- Parallelize the vectorial convolution and compare the new alternative of the filter computation with the previous ones.