ECE/CSE 576

HW 1 Notes
Overview

• Assignment 1 is a big set of exercises to code functions that are basic and many of which are needed for future assignments.
• Sample functions are provided at the beginning of the code, so you get an idea how to work with the images in Qt.
• The required functions come from the lectures on filtering, edge finding, and segmentation.
QImage Class
in the QT package

• The QImage class provides a hardware-independent image representation
• Some of the useful methods
  – QImage() (and other forms with parameters)
  – copy(int x, int y, int width, int height) const
  – setPixel(int x, int y, uint index_or_rgb) can use function qRgb(int r, int g, int b)
  – width() const, height() const
• The QRgb class holds a color pixel.
• from http://doc.qt.io/qt-4.8/qimage.html
Double Arrays

• We’ve modified the original assignment, which had truncation problems when passing images around.
• Instead, you will pass around arrays of doubles.
• The function ConvertQImage2Double() that we provide will convert a QImage to a 2D matrix.
• The first dimension handles both columns (c) and rows (r), while the second one specifies the color channel (0, 1, 2).
• Position (c,r) maps to r*imageWidth + c.
• This will lead nicely in HW 2, which also uses doubles.
• You don’t have to convert back to QImage!
• You do have to copy any images that you are going to modify.
1. Convolution

• The first task is to code a general convolution function to be used in most of the others.

  • `void Convolution(double **image, double *kernel, int kernelWidth, int kernelHeight, bool add)`

  • `image` is a 2D matrix of class `double`
  • `kernel` is a 1D mask array with rows stacked horizontally
  • `kernelWidth` is the width of the mask
  • `kernelHeight` is the height of the mask
  • if `add` is true, then 128 is added to each pixel for the result to get rid of negatives.
Reminder: 2D Gaussian function with standard deviation $\sigma$

In 2-D, an isotropic (i.e. circularly symmetric) Gaussian has the form:

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

This distribution is shown in Figure 2.

Figure 2 2-D Gaussian distribution with mean (0,0) and $\sigma=1$
2. Gaussian Blur

• The second task is to code a Gaussian blur which can be done by calling the Convolution method with the appropriate kernel.

• void GaussianBlurImage(double **image, double sigma)

• Let the radius of the kernel be 3 times $\sigma$

• The kernel size is then $(2 \times \text{radius}) + 1$
3. Separable Gaussian Blur

• Now implement a separable Gaussian blur using separate filters for the horizontal blur and then the vertical blur. Call your Convolution function twice.

• void SeparableGaussianBlurImage(double **image, double sigma)

• The results should be identical to the 2D Gaussian Blur.
4. First and Second Derivatives of the Gaussian

- **void FirstDerivative_x(double **image, double sigma)** takes the image derivative in the x direction using a 1*3 kernel of \{-1.0, 0.0, 1.0\} and then does a standard Gaussian blur.

- **void FirstDerivative_y(double **image, double sigma)** takes the derivative in the y direction and then does a standard Gaussian blur.

- **void SecondDerivImage(double **image, double sigma)** computes the Laplacian function and then does a standard Gaussian. For the Laplacian, rather than taking the derivative twice, you may use the 2D kernel:

  \[
  \begin{bmatrix}
  0.0 & 1.0 & 0.0 \\
  1.0 & -4.0 & 1.0 \\
  0.0 & 1.0 & 0.0 \\
  \end{bmatrix}
  \]

- All of these add 128 to the final pixel values in order to see negatives. This is done in the call to Convolution().
5. Sharpen Image

• Sharpen an image by subtracting the Gaussian-smoothed second derivative image from the original. Will need to subtract back off the 128 that second derivative added on.

• void SharpenImage(double **image, double sigma, double alpha)

• Sigma as usual and alpha is the constant to multiply the smoothed 2nd derivative image by.
6. Sobel Edge Detector

- Implement the Sobel operator, produce both the magnitude and orientation of the edges, and display them.
- `void SobelImage(double **image)`
- Use the standard Sobel masks:
  -1, 0, 1,
-2, 0, 2,
-1, 0, 1
  
  1, 2, 1,
0, 0, 0
-1, -2, -1
7. Bilinear Interpolation

• Given an image and a real-valued point \((x,y)\), compute the RGB values for that point through bilinear interpolation, which uses the 4 closest pixel value.

• `void BilinearInterpolation(double **image, double x, double y, double rgb[3])`

• Put the red, green, and blue interpolated results in the vector rgb.
8. Find Peaks of Edge Responses

- This function finds the peaks of the edge responses perpendicular to the edges.
- `void FindPeaksImage(double **image, double thres)`
- It first uses Sobel to find the magnitude and orientation at each pixel.
- Then for each pixel, it compares its edge magnitude to two samples perpendicular to the edge at a distance of one pixel, which requires BilinearInterpolation().
- If the pixel edge magnitude is $e$ and these two are $e_1$ and $e_2$, a peak $e$ must be larger than “thres” and larger than or equal to $e_1$ and $e_2$.
- See next slide.
e1x = c + 1 \times \cos(\theta);

\text{example: } r=5, \ c=3, \ \theta=135 \ degrees
\sin \theta = .7071, \ \cos \theta = -.7071
\text{e1 } = (2.2929, 5.7071)

e1y = r + 1 \times \sin(\theta);

e2x = c - 1 \times \cos(\theta);

\text{e2 } = (3.7071, 4.2929)

e2y = r - 1 \times \sin(\theta);
9. Color Clustering

- Perform K-means clustering on a color image first with random seeds and then by selecting seeds from the image itself.

- **void RandomSeedImage(double **image, int num_clusters)**

- **void PixelSeedImage(double **image, int num_clusters)**

- Use the RGB color space, and the distance between two pixels with colors \((R_1,G_1,B_1)\) and \((R_2,G_2,B_2)\) is \(|R_1-R_2|+|G_1-G_2|+|B_1-B_2|\).

- Use \(\epsilon = 30\) or max iteration\# = 100