EE-360 Assignment

Edge Detection Of Various Objects using FRDM-KL25z

Team Members:
Mandarapu Vaibhav - 150102032
Sagar Kohli - 150102058
Somya Parashar -150102066
Souradip Pal - 150102076
Akash Singh - 150108002
Mahim goyal -150108044
Sahil Thandra -150108046

Procedure

1. Input was taken in the .pgm format of 32px X 32px.
2. This was given to the python file(Appendix 1) which converted this image to an integer array.
3. This array was then supplied to FRDM-KL25z.
4. The code for the ARM platform was written on Mbed in C language.
5. FRDM-KL25z contains the code in C(Appendix 2) for the edge detection. Edge detection algorithm is elaborated later on. In the report
6. The output was received by a python receiver file. It uses the library pyserial to capture the Serial Bus data. The data was received for 32*32 = 1024 integer entries after which it was converted into a square matrix of 32 x 32. Upon its conversion it was passed to a ploting function which uses matplotlib to display the image.

Edge Detection Algorithm

Operators used in our edge detection algorithm:

- Sobel-Operator
- Laplacain Operator

Sobel-Operator:

The Sobel operator, sometimes called the Sobel-Feldman operator or Sobel filter, is used in image processing, particularly within edge detection algorithms where it creates an image emphasising edges.It is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function. At each point in the image, the result of the Sobel-Feldman operator is either the corresponding gradient vector or the norm of this vector. The operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives - one for horizontal changes, and one for vertical. If we define A as the source image, and Gx and Gy are two images which at each point contain the horizontal and vertical derivative approximations respectively, the computations are as follows:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

where * here denotes the 2-dimensional signal processing convolution operation.

Since the Sobel kernels can be decomposed as the products of an averaging and a differentiation kernel, they compute the gradient with smoothing.

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

The *x*-coordinate is defined here as increasing in the "right"-direction, and the *y*-coordinate is defined as increasing in the "down"-direction. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using:

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

Using this information, we can also calculate the gradient's direction

$$\mathbf{\Theta} = \operatorname{atan}\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right)$$

Since the intensity function of a digital image is only known at discrete points, derivatives of this function cannot be defined unless we assume that there is an underlying continuous intensity function which has been sampled at the image points. With some additional assumptions, the derivative of the continuous intensity function can be computed as a function on the sampled intensity function, i.e. the digital image. It turns out that the derivatives at any particular point are functions of the intensity values at virtually all image points. However, approximations of these derivative functions can be defined at lesser or larger degrees of accuracy.

The Sobel-Feldman operator represents a rather inaccurate approximation of the image gradient, but is still of sufficient quality to be of practical use in many applications. More precisely, it uses intensity values only in a 3×3 region around each image point to approximate the corresponding image gradient, and it uses only integer values for the coefficients which weight the image intensities to produce the gradient approximation.

Laplacian-Operator:

The Laplacian is a 2-D isotropic measure of the 2nd spatial derivative of an image. The Laplacian of an image highlights regions of rapid intensity change and is therefore often used for edge detection. The Laplacian is often applied to an image that has first been smoothed with something approximating a Gaussian Smoothing filter in order to reduce its sensitivity to noise. The operator normally takes a single gray level image as input and produces another gray level image as output.

The Laplacian L(x,y) of an image with pixel intensity values I(x,y) is given by:

$$L(x,y)= \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Since the input image is represented as a set of discrete pixels, we have to find a discrete convolution kernel that can approximate the second derivatives in the definition of the Laplacian. Three commonly used small kernels are shown below:

1	1	1
1	-8	1
1	1	1

-1	2	-1
2	-4	2
-1	2	-1

Because these kernels are approximating a second derivative measurement on the image, they are very sensitive to noise. To counter this, the image is often Gaussian Smoothed before applying the Laplacian filter. This pre-processing step reduces the high frequency noise components prior to the differentiation step. In fact, since the convolution operation is associative, we can convolve the Gaussian smoothing filter with the Laplacian filter first of all, and then convolve this hybrid filter with the image to achieve the required result. Doing things this way has two advantages: Since both the Gaussian and the Laplacian kernels are usually much smaller than the image, this method usually requires far fewer arithmetic operations. The LoG ('Laplacian of Gaussian') kernel can be pre-calculated in advance so only one convolution needs to be performed at run-time on the image.

The 2-D LoG function centered on zero and with Gaussian standard deviation σ has the form:

$$\text{LoG}(x,y) = -1/\pi\sigma^{4}\left[\,1-\left(\frac{x^{2}+y^{2}}{2\sigma^{2}}\right)\right]e^{-\frac{x^{2}+y^{2}}{2\sigma^{2}}}$$

and is shown:

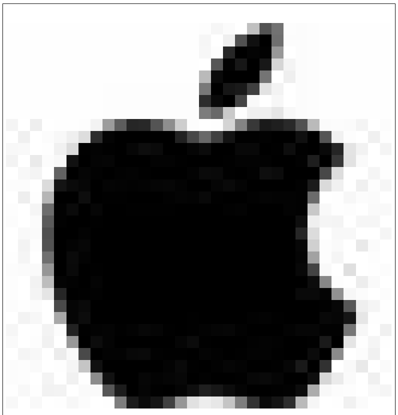
0	1	0
1	-4	1
0	1	0

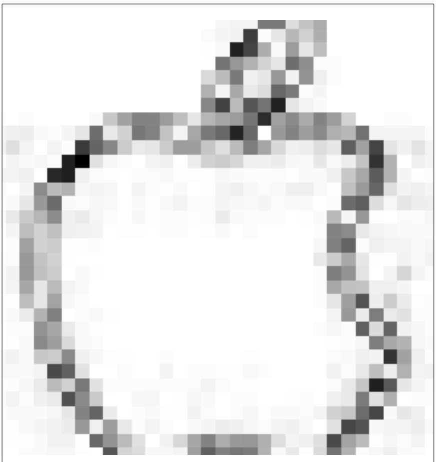
the Gaussian is made increasingly narrow, the LoG kernel becomes the same as the simple Laplacian kernels. This is because smoothing with a very narrow Gaussian ($\sigma < 0.5$ pixels) on a discrete grid has no effect.

Results

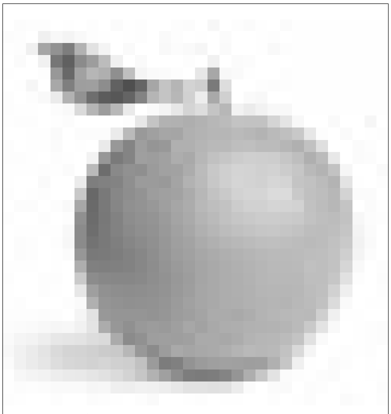
Following are some of the samples which were run and their outputs,

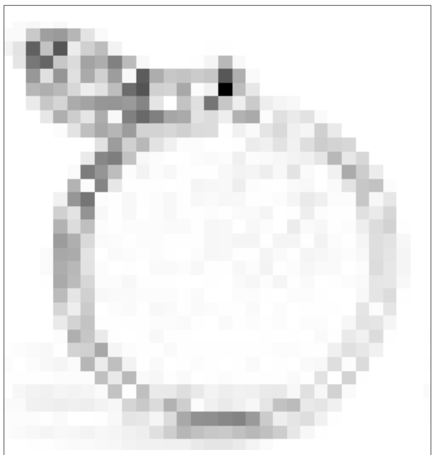
Test 1: Apple Inc. Logo


Input 1


Output 1

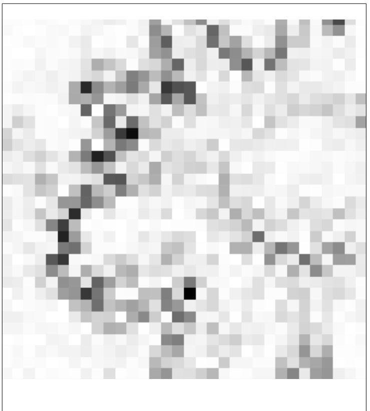
Test 2: Orange

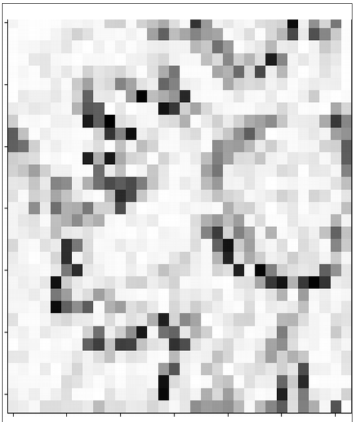

Input 2


Output 2

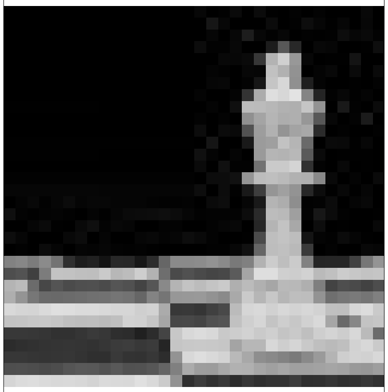
Test 3: Flower

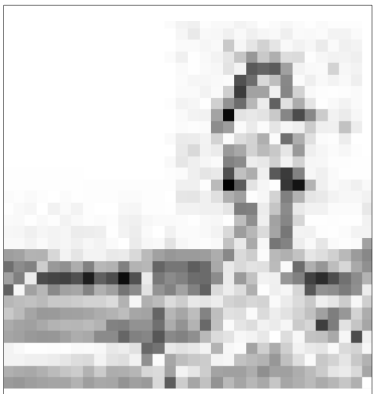
Input 3


Output 3

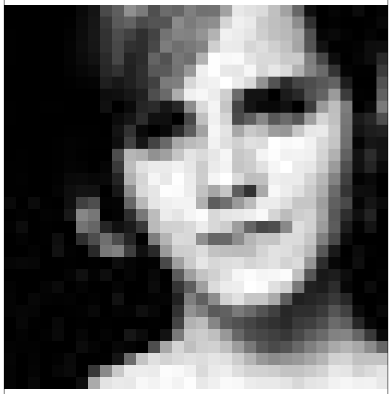

Output_using_Sobel 4

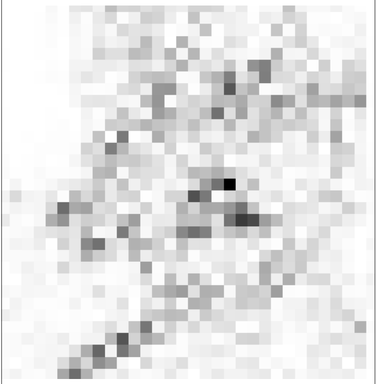
Test 4: King of Chess

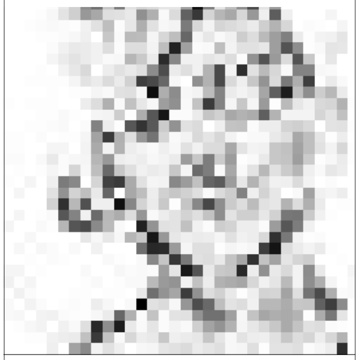

Input 4


Output 4

Test 5: Emma Watson


Input 5


Output_using_Laplacian 5


Output_using_Sobel 5

Conclusion

For simple images the output of the edge detection is as expected.
For complex images the output doesn't neccesarily show the required edges as the number of pixels in the given images are quite few.
If the number of pixels could be increased(which is not possible for FRDM-KL25z) the accuracy of the algorithm would be greatly visible in the output images.
When images of size 64px X 64px were tried the platform become unresponsive. It wasn't clear what the reason was for that, but it seems like there was some memory/stack overflow when the input image was put in flash memory.
We would be thrilled to try this on some other platform with better specifications and performance. We would like to thank the instructors for conducting these labs and to provide us with ARM processor to work on, which gave us the hands on experience.

Appendix 1

Python Code for converting image to an integer array:

```
import time
import serial
import re
import numpy
from PIL import Image
from matplotlib import pyplot

filename='apple.PGM'
byteorder='<'
with open(filename, 'rb') as f:
    buffer = f.read()
try:
    header, width, height, maxval = re.search(
        b"(^P5\s(?:\s*#.*[\r\n])*"
        b"(\d+)\s(?:\s*#.*[\r\n])*"
        b"(\d+)\s(?:\s*#.*[\r\n])*"
        b"(\d+)\s(?:\s*#.*[\r\n]\s)*)", buffer).groups()
    print(len(header))
except AttributeError:
    raise ValueError("Not a raw PGM file: '%s'" % filename)

image=numpy.frombuffer(buffer,
                        dtype='u1' if int(maxval) < 256 else
byteorder+'u2',
                        count=int(width)*int(height),
                        offset=len(header)
                        ).reshape((int(height), int(width)))

ser = serial.Serial('/dev/ttyACM0')
for i in range(0,32):
    for j in range(0,32):
        print(image[i][j],", ", end='')
print("################### Image Sent ####################")
```

Appendix 2a

Mbed code in C language (Laplacian operator):

```
#include "mbed.h"

DigitalOut myled(LED_GREEN);
Serial pc(USBTX, USBRX);

int main()
{
    int rows = 32, cols = 32;
    int row = rows, col = cols;
    char c;
    int i = 0, j = 0;
    int imag[32][32]={0 , 0 , 0 , 0 , 0 , 0 , 10 , 21 , 74 , 95 ,
144 , ........ , 242 , 243 , 233};
    int outImag[32][32];
    for(i = 0; i < row; i++) {
        for(int j = 0; j < col; j++) {
            int result = 0;
            if(i+1 < row) {
                result += ((-1)*imag[i+1][j] + imag[i][j])/2;
            }
            if(i-1 > 0) {
                result += ((-1)*imag[i-1][j] + imag[i][j])/2;
            }
            if(j+1 < col) {
                result += ((-1)*imag[i][j+1] + imag[i][j])/2;
            }
            if(j > 0) {
                result += ((-1)*imag[i][j-1] + imag[i][j])/2;
            }
            if(result < 0) result = result*(-1);
            outImag[i][j] = result;
        }
    }

    for(int j = 0; j < rows*cols; j++) {
        wait(0.01f); // wait a small period of time
        char cc = outImag[j/rows][j%cols];
        pc.printf("%c", cc); // print the value of variable i
        myled = !myled; // toggle a led
    }
}
```

Appendix 2b

Mbed code in C language (Laplacian operator):

```
#include "mbed.h"

DigitalOut myled(LED_GREEN);
Serial pc(USBTX, USBRX);

int main()
{
    int rows = 32, cols = 32;
    int row = rows, col = cols;
    char c;
    int i = 0, j = 0;
    int imag[32][32]={0 , 6 , 5 , 3 , 8 , 9 , 10 , 17 , 14 , 11 ,
11 , 32 , 33 , 30 , 12 , 138 , 180 , 122 , 49 , 26  , ....... , 28 ,
186 , 212 , 189 , 201 , 136 , 155 , 184 , 207 , 218 , 210 , 186,
221 };
    int outImag[32][32];
    for(i = 0; i < row; i++) {
        for(int j = 0; j < col; j++) {
            int result1 = 0, result2 = 0;
            if(i+1 < row) {
                result1 += ((2)*imag[i+1][j]);
            }
            if(i-1 > 0) {
                result1 += ((-2)*imag[i-1][j]);
            }
            if(j+1 < col) {
                result2 += ((-2)*imag[i][j+1]);
            }
            if(j > 0) {
                result2 += ((2)*imag[i][j-1]);
            }
            if(result1 < 0) result1 = result1*(-1);
            if(result2 < 0) result2 = result2*(-1);
            outImag[i][j] = result1 < result2 ? result1 : result2;
        }
    }

    for(int j = 0; j < rows*cols; j++) {
        wait(0.01f); // wait a small period of time
        char cc = outImag[j/rows][j%cols];
        pc.printf("%c", cc); // print the value of variable i
        myled = !myled; // toggle a led
    }
}
```

Appendix 3

Python code to receiver the output of FRDM-KL25z and convert that to image:

```
import serial
import numpy
from PIL import Image
from matplotlib import pyplot

temp = 0
ser = serial.Serial('/dev/ttyACM0')
size = 0

row = 32
col = 32
final2 = numpy.zeros([row*col])

while size < row*col:
    temp = ord(ser.read(1))
    final2[size] = 255 - temp
    print(temp, size    )
    size += 1

final2 = numpy.reshape(final2, (row, col))
print('/n',final2)
pyplot.imshow(final2, pyplot.cm.gray)
pyplot.show()
```