Image processing techniques in Fourier domain

1. Theoretical fundamentals

A Fourier Transform (FT) is a mathematical transform that decomposes functions depending on space or time into functions depending on spatial frequency or temporal frequency. An example application would be decomposing the waveform of a musical chord into terms of the intensity of its constituent pitches. The term Fourier Transform refers to both the frequency domain representation and the mathematical operation that associates the frequency domain representation to a function of space or time.

The Fourier Transform is used if we want to access the geometric characteristics of a spatial domain image. Because the image in the Fourier domain is decomposed into its sinusoidal components, it is easy to examine or process certain frequencies of the image, thus influencing the geometric structure in the spatial domain.

In most implementations the Fourier image is shifted in such a way that the DC-value is displayed in the center of the image. The further away from the center an image point is, the higher is its corresponding frequency.

Taking into consideration the above criteria, different kind of filters can be implemented using the Fourier transform. If we want to implement a Low-Pass Filter, we the frequencies that are further away from the center. The opposite is also true, if we want to implement a High-Pass Filter, we cut the frequencies that are closer to the center. The Band-Pass Filter and the Band-Stop Filter are combinations of those.

The definition of the 2D Fourier Transform is the following:

$$F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-j2\pi(ux+vy)}\,dx\,dy,$$
$$f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)e^{j2\pi(ux+vy)}\,du\,dv$$

Figure 1: The definition of the Fourier Transform

Where u and v are spatial frequencies. F(u, v) is complex in general:

$$F(u,v) = F_{\mathrm{R}}(u,v) + jF_{\mathrm{I}}(u,v)$$

- $|F(u,v)|$ is the magnitude spectrum
- $\arctan(F_{\mathrm{I}}(u,v)/F_{\mathrm{R}}(u,v))$ is the phase angle spectrum.
- Conjugacy: $f^*(x,y) \Leftrightarrow F(-u,-v)$
- Symmetry: $f(x,y)$ is even if $f(x,y) = f(-x,-y)$

Figure 2: Complex representation

For this project, I have chosen to use a Low-Pass Filter. This type of filter should erase the details of the image. It cuts the frequencies which are above a certain level which is called the cut-off frequency.

For a 2D filter we have M and N, which are the dimensions of the image. The coordinates of the filter are represented by u and v. Those have to be normalized using the following formulas:

m=(u-M/2)/M, n=(v-N/2)/N

Figure 3: Normalizing the frequencies

The values of m and n will be in the -0.5 and 0.5 range. We will notate the cut-off frequency with Fc, and define r as follows:

$$r(u,v) = \sqrt{m^2 + n^2}$$

Figure 4: The definition of r

Then, using the formula for the Los-Pass Filter, we will cut the details of the image:

$$H_{LP}(u,v) = \begin{cases} 1, & r(u,v) < F_c \\ 0, & otherwise \end{cases}$$

Figure 5: The definition of the LPF

2. Implementation

For the implementation I used a one-dimension array for the input and output image. Those were of type unsigned char. To realize the Fourier Transform of the image I used the predefined functions for the Blackfin BF533 rfft2d_fr32 and ifft2d_fr32. Using the fract32 format, the sub unitary values could be represented. The unsigned char values from the original image were converted in fract32 format using the float_to_fr32() function.

```
for(i=0;i<128;i++)
for(j=0;j<128;j++)
    in[i+j*128]= float_to_fr32((imIn[i+j*128]-128.0)/128.0);
```

Figure 6: Float to fract32

The twidfft2d_fr32() function was used to generate the twiddle factor which represents the coefficients of the sine and cosine. Once the transform was applied to the input image, the result was stored in a complex_fract32 variable.

Then the filtering part comes. Below is the code that does the filtering:

```
double cut_off = 0.685;
double val = 0;
double m = 0;
double n = 0;

double min = 0.5;
double max = 0;

for(i=0;i<128;i++)
for(j=0;j<128;j++)
{
    m = (i-64)/128.0;
    n = (j-64)/128.0;
    val = sqrt(pow(m,2)+pow(n,2));

    if(val>max)
    max=val;

    if(val<min)
    min=val;

    if(val>cut_off)
    {
        out[i+j*128].re = float_to_fr32(0);
        out[i+j*128].im = float_to_fr32(0);
    }
}
printf("%f\n",max);
printf("%f\n",min);
```

Figure 7: LPF

The cut_off variable represents the cut-off frequency. Val is the value of r from the Fourier Transform formula. I used the min and max variables to find out between which values the val variable was finding itself. I find out that its value was between 0 and 0. 707107. This is the reason for the value of the cut_off variable. The program works best if this variable takes values between 0.6 and 0.7. If val exceeds the max value, there is no filtering taking place.

We can see that if the val is bigger than cut-off, the respective frequency component takes the value 0 for both real and imaginary part.

After the filtering, we perform the inverse transform in order to be able to visualize the effect of the filter. There is the part of the code that does that:

```
ifft2d_fr32(out1, tmp1, out2, won,1, 128);
printf("fftinv \n");

for(i=0;i<128;i++)
for(j=0;j<128;j++)
{
    conv_f=(fr32_to_float(out_inv[i+j*128].re))*16384; //1/(128*128)
    //printf("%f\n", conv_f*128+128);
    imOut[i+j*128]= conv_f*128+128;
    //imOut[i+j*128]= fr32_to_float(sp[i+j*128])*1600000.0+128;
}
```

Figure 8: The inverse transform

In the code, I also used the function clock() in order to be able to measure the time necessary for the program to compile. I defined two volatile clock_t variables. To one of them I gave value at the beginning of the program and to the other one at the end. I made the difference between them and then I divided the result to the number of clocks per cycle to give me the value in seconds.

```
clock_stop = clock();
secs = ((double) (clock_stop-clock_start)) / CLOCKS_PER_SEC;
printf("Time taken is %e seconds.\n",secs);
```

Figure 9: Measuring the time necessary

3. The results

I will begin be showing the time that was necessary to complete the compilation and the maximum and minimum values of the r.

```
0.707107
0.000000
fftinv
Time taken is 2.446024e-01 seconds.
```

Figure 10: Console output

As I said earlier, the first two values represent the maximum and minimum respectively values for r. The last row says the number of seconds that were necessary to complete the compilation of the program. It took approximatively 200ms.

Below is the image that I used as the input:

Figure 11: Lena

The following images represent the output image at different cut-off frequencies (the cut_off variable). The value of the cut_off is written in the caption of the image.

Figure 12: 0.6

Figure 13: 0.65

Figure 14: 0.68

Figure 15: 0.685

Figure 16: 0.69

Figure17: 0.7

4. The code

```
/*****************************************************************************
 * TPI_project.c
 *****************************************************************************/

#include <fract.h>
#include <cycle_count.h>

#include <complex.h>
#include <filter.h>
#include <math.h>
#include <stdio.h>

#include <time.h>

int contor,conv;
float conv_f;
//unsigned char imIn[128][128];
unsigned char imIn[16384];
//unsigned char imOut[128][128];
unsigned char imOut[16384];

//complex_fract32 out[128][128];
complex_fract32 out[16384];

//fract32 in[128][128];
fract32 in[16384];
fract32 sp[16384];

//complex_fract32 tmp[128][128];
complex_fract32 tmp[16384];

//complex_fract32 out_inv[128][128];
complex_fract32 out_inv[16384];
complex_fract32 won[128];

fract32 *in1 = (fract32*)&in[0];

complex_fract32 *out1=&out[0];
complex_fract32 *tmp1=&tmp[0];
complex_fract32 *out2=&out_inv[0];

```

```
46.int main( void )
47.{
48.    volatile clock_t clock_start;
49.    volatile clock_t clock_stop;
50.    double secs;
51.    clock_start = clock();
52.
53.    //cycle_t start_count;
54.    //cycle_t final_count;
55.    /* Begin adding your custom code here */
56.    int i,j;
57.    //START_CYCLE_COUNT(start_count);
58.    for(i=0;i<128;i++)
59.    for(j=0;j<128;j++)
60.        in[i+j*128]= float_to_fr32((imIn[i+j*128]-128.0)/128.0);
61.        //in[i+j*128]= float_to_fr32((imIn[i+j*128]*pow(-1,i+j)-
   128.0)/128.0);
62.
63.    //printf("scalare \n");
64.
65.    twidfft2d_fr32 (won, 128);
66.    //printf("twiddle \n");
67.
68.
69.    rfft2d_fr32(in1, tmp1, out1, won, 1, 128);
70.    //printf("fft \n");
71.
72.    ////////////////////////
73.
74.    /*
75.    val takes values between 0 and 0.707107
76.    recomanded values between 0.6 and 0.7
77.    */
78.    double cut_off = 0.685;
79.    double val = 0;
80.    double m = 0;
81.    double n = 0;
82.
83.    double min = 0.5;
84.    double max = 0;
85.
86.    for(i=0;i<128;i++)
87.    for(j=0;j<128;j++)
88.    {
89.        m = (i-64)/128.0;
```

```
90.         n = (j-64)/128.0;
91.         val = sqrt(pow(m,2)+pow(n,2));
92.
93.         if(val>max)
94.         max=val;
95.
96.         if(val<min)
97.         min=val;
98.
99.         if(val>cut_off)
100.             {
101.                 out[i+j*128].re = float_to_fr32(0);
102.                 out[i+j*128].im = float_to_fr32(0);
103.             }
104.         }
105.         printf("%f\n",max);
106.         printf("%f\n",min);
107.
108.         ////////////////////////
109.
110.         /*
111.         for(i=0;i<128;i++)
112.         for(j=0;j<128;j++)
113.         {
114.             sp[i+j*128]=add_fr1x32(mult_fr1x32x32(out[i+j*128].re,out[i+
    j*128].re), mult_fr1x32x32(out[i+j*128].im,out[i+j*128].im));
115.         }
116.         */
117.
118.         ifft2d_fr32(out1, tmp1, out2, won,1, 128);
119.         printf("fftinv \n");
120.
121.         for(i=0;i<128;i++)
122.         for(j=0;j<128;j++)
123.         {
124.             conv_f=(fr32_to_float(out_inv[i+j*128].re))*16384;
    //1/(128*128)
125.             //printf("%f\n", conv_f*128+128);
126.             imOut[i+j*128]= conv_f*128+128;
127.             //imOut[i+j*128]= fr32_to_float(sp[i+j*128])*1600000.0+128;
128.         }
129.         //printf("final  \n");
130.         //STOP_CYCLE_COUNT(final_count,start_count);
131.         //PRINT_CYCLES("Number of cycles:",final_count);
132.
```

```
133.          clock_stop = clock();
134.          secs = ((double) (clock_stop-clock_start)) / CLOCKS_PER_SEC;
135.          printf("Time taken is %e seconds.\n",secs);
136.
137.          return 0;
138.      }
139.
```