Chapter 9

Frequency Domain Processing

This chapter discusses simple forms of frequency domain transforms based on Fourier transforms. Normally images are processed using the spacial domain, which is the natural way of dealing with images. However, it is often convenient to transform the representation of an image to the frequency domain. The classical way of doing that is using Fourier Transforms.

Three common applications for Fourier Transforms are: filter out certain undesirable frequencies (pass-low or pass-high filters), unblurring (similar effects to the previously studied sharpening convolution filters, but more powerful), and feature extraction for object detection.

9.1 Fourier Transform

The Fourier transform has been known for a long time. The basic idea is that any cyclic function can be expressed as a sum of infinite series of sines or cosines of increasing frequency. The equations and derivation are available in many books (see for example [1]), so we are not repeating them here. Instead, we are going to focus on the Fourier transform for digital images, which are discrete and in two dimensions.

The special version of the Fourier transform that uses discrete data is called DFT. DFTs for two dimensions takes $O(N^2)$. Due to certain repeated operations it is possible to create an algorithm that computes the discrete transform more rapidly, in $O(NlogN)$. This form is called FFT (for Fast Fourier Transform). There are many versions of this algorithm in the form of open source software, including parallel versions. One of the best is called FFTW (available at http://www.fftw.org/). OpenCV implements it too, but the function cvDFT() is actually a customised form of an FFT.

The DFT general formula is:

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-i2\pi(ux/M+vy/N)} \tag{9.1}$$

The component with the Neperian number (e) is a complex number[1]. Therefore, for every DFT there is a real component and a complex component. These need to be stored in separate arrays (see the code example below).

For an illustration example, see figures 9.1 to 9.4.

The first image in figure 9.1 is formed by a single frequency along the axis x. Note that there are three bright points, representing the main frequency found in this image. The distance between the centre and one of the other points indicates the frequency.

[1]from Euler's identity: $e^{i\theta} = cos(\theta) + i \ sin(\theta)$

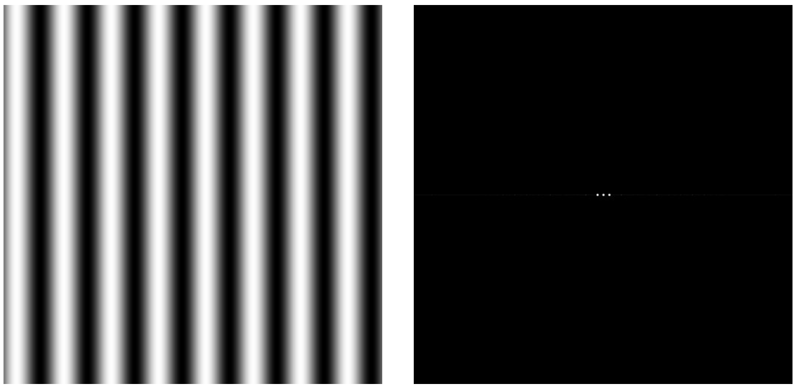

Figure 9.1: a) The original image. b) The FFT of a).

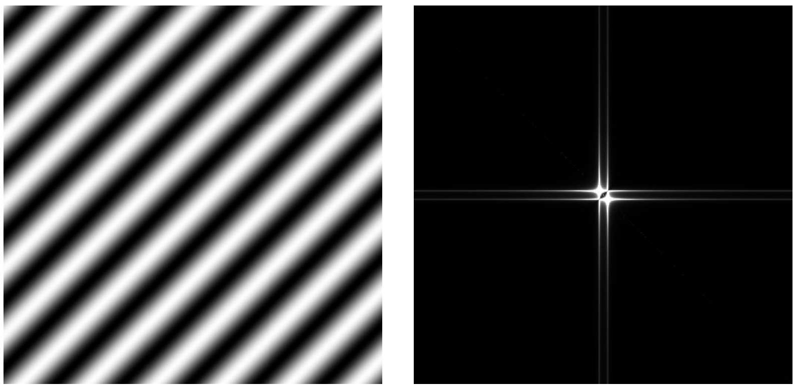

Figure 9.2: a) The original image. b) The FFT of a).

In the second image in figure 9.2 the image is rotated 45 degrees. Note that the bright points also were rotated by 45 degrees. However, the frequencies form a pattern where there are straight lines in the same direction of the edges of the square.

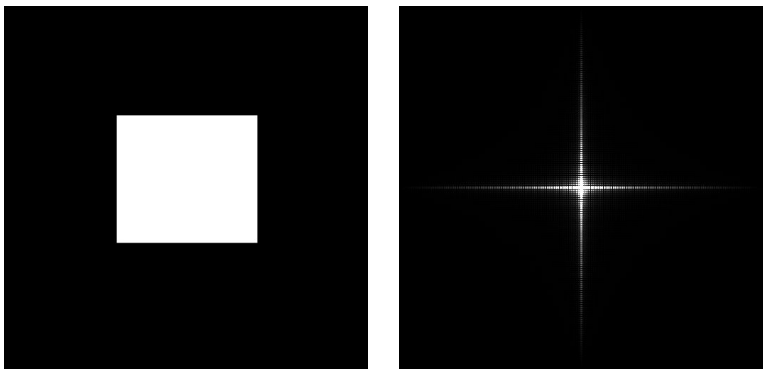

Figure 9.3: a) The original image. b) The FFT of a).

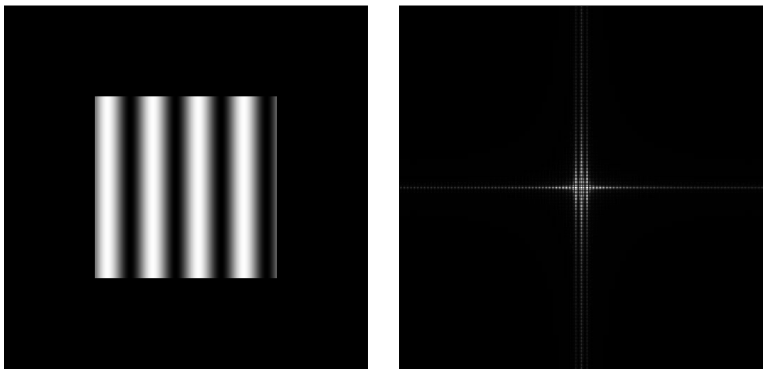

Figure 9.4: a) The original image. b) The FFT of a).

In the third image in figure 9.3, a square is subjected to the FFT. There are no single bright points, as there are many frequencies that forms this particular image.

In the forth image in figure 9.4, a square containing a single frequency figure is subjected to the FFT. Note the pattern formed by the FFT.

One important observation about the previous images is that the images of the FFTs are limited representations of the results. The actual resulting matrix contains complex numbers, as seen in equation 9.1. The only way to "show" the FFT is to either show the real part of the complex numbers, or to show the module.

There is also an inverse DFT, given by equation 9.2. One can recover an image from its DFT. It is interesting to note that original images obviously do not have a complex component. In practise, equation 9.1 produces coefficients that need to be rounded up. Therefore, when recovering the image, the complex part might not be zero (as it should). Due to these numerical problems the inverse DFT may not be completely accurate (lossy).

$$f(u,v) = \frac{1}{NM} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{-i2\pi(ux/M+vy/N)} \tag{9.2}$$

In the next example, Akiyo's image in figure 9.5 is subjected to the FFT and recovered via equation 9.2.

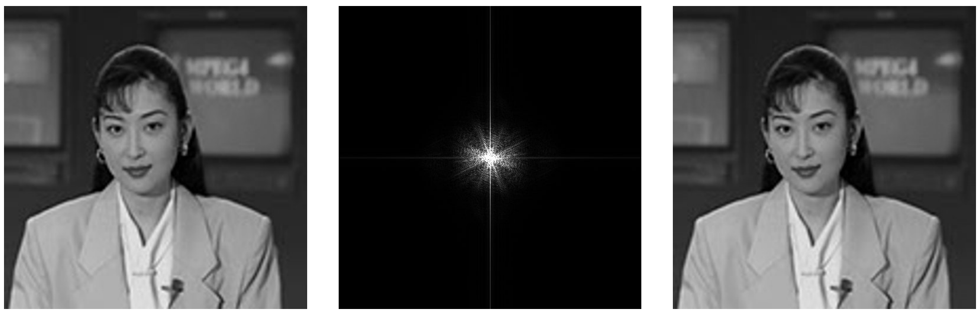

Figure 9.5: a) The original image. b) The FFT of a). c) The reconstructed image using the data from the FFT (not image b!))

9.1.1 A C code for FFTs

The properties of the transform allows one to carry out the calculations separately for rows and columns. Performing one dimensional transform on each line and then on each column makes it relatively easy to implement.

Here there is a simple example of an FFT algorithm (based on Johnson, M.J., “Notes on Machine Vision”) that can be used for computing the transforms for images that are 512 x 512 pixels:

Listing 9.1: FFT function

```c
#include "imgio.c"
#include <math.h>
#include "cv.h"
#include "highgui.h"
//
// FFT example for 159731 adapted
// from M. Johnson's code 21/5/01
//
# define M_PIl
3.1415926535897932384626433832795029L
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr
// a complex pixel
typedef struct {
  double real;
  double imag;
} cpixel;
// a complex image
typedef struct {
  int x,y;
  cpixel **data;
} cimage;
// a pointer to a complex image
typedef cimage * cimg_ptr;
// convert d to a pixel
int topixel(double d) {
  if (d>254) return 255;
  if (d<1) return 0;
  return (int) d;
}
// create complex image
cimg_ptr create_cimage(int x,int y) {
  cimg_ptr im;
  int a;

  im=new(cimage,1);
  im->x=x;
  im->y=y;
  im->data=new(cpixel *,im->y);
  im->data[0]=new(cpixel,im->x*im->y);
  for( a=1;a < im->y;a++)
    im->data[a]=im->data[a-1]+im->x;
  return im;
}

// isign is 1 for forward transform
// and -1 for inverse
void fft(double *data,int nn,int isign,int w){
  int n,mmax,m,j,istep,i;
  double wtemp,wr,wpr,wpi,wi,theta;
  double tempr,tempi;
  data-=w;
  n=nn<<1;
  j=1;
  for (i=1;i<n;i+=2) {
    if (j>i) {
      SWAP(data[j*w],data[i*w]);
      SWAP(data[(j*w)+1],data[(i*w)+1]);
    }
    m=n>>1;
    while (m>=2 && j>m ) {
      j-=m;
      m >>= 1;
    }
    j+=m;
  }
  mmax=2;
  while (n>mmax) {
    istep=2*mmax;
    theta=(2*M_PIl)/(isign * mmax);
    wtemp=sin(.5*theta);
    wpr= -2.0 * wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    for(m=1;m<mmax;m+=2) {
      for(i=m;i<=n;i+=istep) {
        j=i+mmax;
        tempr=wr*data[j*w]-wi*data[(j*w)+1];
        tempi=wr*data[(j*w)+1]+wi*data[j*w];
        data[j*w]=data[i*w]-tempr;
        data[(j*w)+1]=data[(i*w)+1]-tempi;
        data[i*w]+=tempr;
        data[(i*w)+1]+=tempi;
      }
      wr=(wtemp=wr)*wpr-wi*wpi+wr;
      wi=wi*wpr+wtemp*wpi+wi;
    }
    mmax=istep;
  }
  for(i=1;i<n;i+=2) {
    data[i*w]=data[i*w]/sqrt(nn);
    data[(i*w)+1]=data[(i*w)+1]/sqrt(nn);
  }
}

// 2d FFT of a complex 512*512 image
void fft2d(cimg_ptr i,int dir) {
 int j;
 for(j=0;j<i->y;j++)
   fft((double *)(i->data[j]),512,dir,1);
 for(j=0;j<i->x;j++)
   fft((double *)(i->data[0])+j*2,512,dir,512);
}
```

Follows an example that uses the FFT function shown above with OpenCV. A forward FFT is carried out and the real component is saved as an image. As the pixels are represented by doubles, it is not possible to "see" them properly, unless they are transformed to *uchars*. In order to print the values, they need to be rounded up to the usual 0 to 255 range. Obviously this transformation loses information, and it is only used for printing purposes. The full precision numbers are needed if we want to recover the original image using an inverse FFT.

Listing 9.2: FFT sample

```
char* filename;
IplImage *cvimage = 0, *cvimage2 = 0;
unsigned char * cvpixel;
cimg_ptr fftdata;
int main( int argc, char** argv )
{
    if (argc == 2) { filename=argv[1];}
    else exit(0);
    if( (cvimage = cvLoadImage( filename, 1)) == 0 )
        return -1;
    if (cvimage->width!=512 || cvimage->height!=512) {
     exit(0); printf("The image should be 512x512 pixels\n");
    }
    cvimage2 = cvCreateImage(cvSize(cvimage->width,cvimage->height
        ), IPL_DEPTH_8U, 1);
    cvCvtColor(cvimage, cvimage2, CV_BGR2GRAY);
    fftdata=create_cimage(cvimage->width,cvimage->height);
    //access pixels of the grey-scale image
    for (int pos_y=0;pos_y<cvimage2->height;pos_y++){
      for (int pos_x=0;pos_x<cvimage2->width;pos_x++){
          cvpixel=&((uchar*)(cvimage2->imageData+cvimage2->
             widthStep*pos_y))[pos_x];
          fftdata->data[pos_y][pos_x].real=*cvpixel;
          fftdata->data[pos_y][pos_x].imag=0;
      }
    }
    fft2d(fftdata,1);//forward FFT
    for (int pos_y=0;pos_y<cvimage2->height;pos_y++){
      for (int pos_x=0;pos_x<cvimage2->width;pos_x++){
          cvpixel=&((uchar*)(cvimage2->imageData+cvimage2->
             widthStep*pos_y))[pos_x];
          *cvpixel=topixel(10*(fftdata->data[pos_y][pos_x].real)
             +128);
      }
    }
    cvSaveImage ("output.bmp", cvimage2);
    cvReleaseImage(&cvimage);
    cvReleaseImage(&cvimage2);
    return 0;
}
```

This particular implementation of FFT uses a square image of sizes 2^n. This is because it is easier to implement if the sizes are powers of 2. If different widths or heights need to be used, we can use padding, i.e., copy the image to a 512x512 size image filling up any spaces with zeros. The pixels with value zero will not contribute to the coefficients.

9.1.2 Some Fourier Transform Properties

FFTs can be used for filtering, feature extraction, coding etc.

We can use filters that can let certain frequencies pass, e.g., low pass filters and high pass filters. Then we apply the inverse transform and recover an image where some of the frequencies are no longer present.

Features can be extracted, as FFTs are invariant to certain modifications. For example, FFTs are invariant to translations of the original image.

One strong reason to use FFTs is that we can convolve images in a very efficient way.

According to the convolution theorem:

$$f(x,y) * g(x,y) \Leftrightarrow F(u,v).G(u,v) \tag{9.3}$$

The convolution between $f(x,y)$ and $g(x,y)$, both in spatial domain, is the equivalent of the product of the Fourier Transform of each function (in frequency domain). Therefore, if we want to find the convolution of the image $f(x,y)$ with a function $g(x,y)$:

$$f(x,y) * g(x,y) = FT^{-1}(F(u,v).G(u,v)) \tag{9.4}$$

where $F(u,v)$ and $G(u,v)$ are the Fourier Transform of $f(x,y)$ and $g(x,y)$ respectively.

It is also possible to carry out a “de-convolution”:

$$f(x,y) *^{-1} g(x,y) = FT^{-1}(F(u,v) \div G(u,v)) \tag{9.5}$$

9.1.3 Filters in frequency domain

Using the properties of the convolution, filters can be used. The following steps apply ([1]):

1. Compute the FFT from the original image. Care must be taken with the image sizes, as FFTs usually work with image sizes that are powers of 2.
2. *Multiply* $F(u,v)$ by the FFT of a filter function $H(u,v)$.
3. Compute the inverse FFT of the result.
4. Obtain the real part of the result (the complex parts should be discarded as explained before).
5. Save the new image.

An important application of FFTs in image processing is the ability to unblur images. Similarly, for deconvolution, the following steps should be carried out:

1. Compute the FFT from the original image $f(x,y)$.
2. *Divide* $F(u,v)$ by the FFT of a filter function $H(u,v)$.
3. Compute the inverse FFT of the result.

4. Obtain the real part of the result (the complex parts should be discarded as explained before).
5. Save the new image.

As an illustration of this process, a deconvolution code was used to improve the quality of two different images. In this example, the filter $H(u, v)$ is a Gaussian. Image 9.6 show Akiyo first frame that was blurred by a Gaussian filter. Image 9.7 shows another image that was originally blurred by a (probable) Gaussian as well, and it was unblurred using the same function as the previous example, but with different parameters.

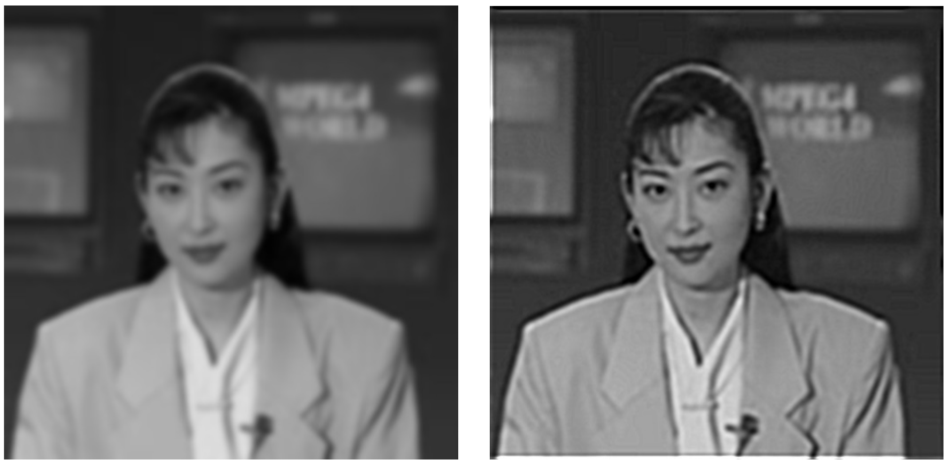

Figure 9.6: a) The original image. b) Deconvolution of a).

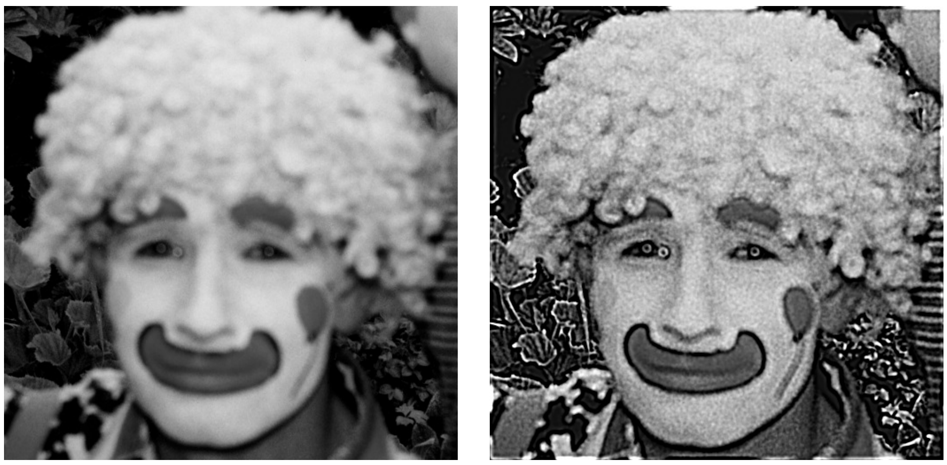

Figure 9.7: a) The original image. b) Deconvolution of a).

9.2 Fourier Descriptors

We have seen feature extraction methods that are invariant to certain transformations. There is a method used in object recognition that uses the profile of objects to compute the Fourier coefficients. Althought these features are not invariant, they can be computed in a way that they become translation invariant. The scaling issue can be dealt with easily and efficiently, as there is a limited number of points representing the object's profile. One advantage is that the profile can be reconstructed accurately from its Fourier coefficients. The coefficients computed by the DFT using a special form are losely known as *Fourier Descriptors.*

An arbitrary profile has a sequence of K coordinate pairs $(x_0, y_0), (x_1, y_1), ...(x_{K-1}, y_{K-1})$. Considering that these pair can be represented as a complex number, it can be written as a sequence of pairs:

$$s(k) = x(k) + iy(k) \tag{9.6}$$

The DFT can be re-written for the sequence $s(k)$ as follows:

$$a(u) = \sum_{k=0}^{K-1} s(k)e^{-i2\pi uk/K} \tag{9.7}$$

Moreover, the inverse transform recovers the original points of the profile:

$$s(k) = \frac{1}{K}\sum_{u=0}^{K-1} a(u)e^{i2\pi uk/K} \tag{9.8}$$

The lower frequencies of the profile are represented in the coefficients in the beginning of the sequence, while the higher frequencies are at the end of the sequence $a(u)$.

It is possible to partially reconstruct the profile by using a limited number of coefficients by imposing a threshold. However, this will cause the loss of some of the details of the profile. This allows for the representation of the "important" features of the profile, as probably the first few suffice to describe the object to be recognised.

9.2.1 Implementation of Fourier Descriptors

In order to implement Fourier Descriptors it is easier to separate the real from the imaginary components of the complex number in equations 9.7 and 9.8. For example, the real part of a descriptor $a(u)$ is:

$$a_{real}(u) = \sum_{k=0}^{K-1} x(k)cos(\theta) - y(k)sin(-\theta) \tag{9.9}$$

$$a_{imag}(u) = \sum_{k=0}^{K-1} x(k)sin(-\theta) + y(k)cos(\theta) \tag{9.10}$$

where: $\theta = (2\pi uk)/K$ and K is the number of points in the contour.

And the reconstruction (recovering $x(k)$ and $(y(k))$:

$$x(k) = \sum_{u=0}^{K-1} a_{real}(u)cos(\theta) - a_{imag}(u)sin(\theta) \tag{9.11}$$

$$y(k) = \sum_{u=0}^{K-1} a_{imag}(u)cos(\theta) + a_{real}(u)sin(\theta) \tag{9.12}$$

Equations 9.9 to 9.12 are fine for analysis and reconstruction, but they are not so useful as features if the original contour is rotated, scaled and translated. Often a different form of feature can be found that are invariant. For example, the following equations are going to compute a set of invariant features from the FDs [2]:

$$a_x(k) = \frac{1}{K}\sum_{u=0}^{K-1} x[k]cos(\theta); \tag{9.13}$$

$$b_x(k) = \frac{1}{K}\sum_{u=0}^{K-1} x[k]sin(\theta); \tag{9.14}$$

$$a_y(k) = \frac{1}{K}\sum_{u=0}^{K-1} y[k]cos(\theta); \tag{9.15}$$

$$b_y(k) = \frac{1}{K}\sum_{u=0}^{K-1} y[k]sin(\theta); \tag{9.16}$$

The features set is (for $k = 1$ to $k <= K$, so the CE sequence starts with $CE(1)$ to avoid divisions by zero):

$$CE(k) = \sqrt{\frac{a_x^2(k) + a_y^2(k)}{a_x^2(1) + a_y^2(1)}} + \sqrt{\frac{b_x^2(k) + b_y^2(k)}{b_x^2(1) + b_y^2(1)}} \tag{9.17}$$

This form is called Elliptic Fourier descriptors, and they should be invariant to certain geometric transformations (rotation, scaling and translation). A possible Opencv implementation of the Elliptic Fourier descriptors is listed in 9.3.

Listing 9.3: EFDs function

```
void EllipticFourierDescriptors(vector<Point>& contour, vector<
   float> CE){
 vector<float> ax, ay, bx,by;
 int m=contour.size();
 int n=20;//number of CEs we are interested in computing
 float t=(2*PI)/m;
 for(int k=0;k<n;k++){
   ax.push_back(0.0);
   ay.push_back(0.0);
   bx.push_back(0.0);
   by.push_back(0.0);
   for (int i=0;i<m;i++){
      ax[k]=ax[k]+contour[i].x*cos((k+1)*t*(i));
      bx[k]=bx[k]+contour[i].x*sin((k+1)*t*(i));
      ay[k]=ay[k]+contour[i].y*cos((k+1)*t*(i));
      by[k]=by[k]+contour[i].y*sin((k+1)*t*(i));
   }
   ax[k]=(ax[k])/m;
   bx[k]=(bx[k])/m;
   ay[k]=(ay[k])/m;
   by[k]=(by[k])/m;
 }
 for(int k=0;k<n;k++){
    CE.push_back( sqrt((ax[k]*ax[k]+ay[k]*ay[k])/(ax[0]*ax[0]+ay
        [0]*ay[0]))+sqrt((bx[k]*bx[k]+by[k]*by[k])/(bx[0]*bx[0]+by
        [0]*by[0])) );
  }
  for(int count=0;count<n && count<CE.size();count++){
    printf("%d CE %f   ax %f   ay %f   bx %f    by%f \n",count,CE[
        count],ax[count],ay[count],bx[count],by[count]);
  }
}
```

A simple example to extract contours of an image using threshold is listed in 9.4. There is a drawing contour function from OpenCV as well. In the new API the contour is implemented as a 2D vector (STL).

Listing 9.4: Contour function

```
int main( int argc, char** argv )
{
  Mat image;
  image=imread("image.jpg");
  cvtColor( image, image, CV_BGR2GRAY );
  threshold( image, image, 5, 255, CV_THRESH_BINARY );
  imshow("Binary image", image);
  vector<vector<Point> > contours;
  findContours( image, contours,CV_RETR_EXTERNAL,
      CV_CHAIN_APPROX_NONE);
  //drawing the largest contour
  Mat drawing = Mat::zeros( image.size(), CV_8UC3 );
  Scalar color = CV_RGB( 0, 255,0 );
  int largestcontour=0;
  long int largestsize=0;
  for(int i = 0; i< contours.size(); i++ )
  {
    if(largestsize < contours[i].size()) {
      largestsize=contours[i].size();
      largestcontour=i;
    }
  }
  drawContours( drawing, contours, largestcontour, color, 1, 8);
  namedWindow( "Contour", CV_WINDOW_AUTOSIZE );
  imshow( "Contour", drawing );
  waitKey(0);
  vector<float> CE;
  EllipticFourierDescriptors(contours[largestcontour], CE);
}
```

An example of the Elliptic Fourier Descriptors can be seen in figure 9.8.

It is important to understand that the Elliptic Fourier Descriptors are not able to reconstruct the contour as we did with equation 9.8. This is the case with invariant features. The information about rotation, scale and position of the original contour is lost in order to make

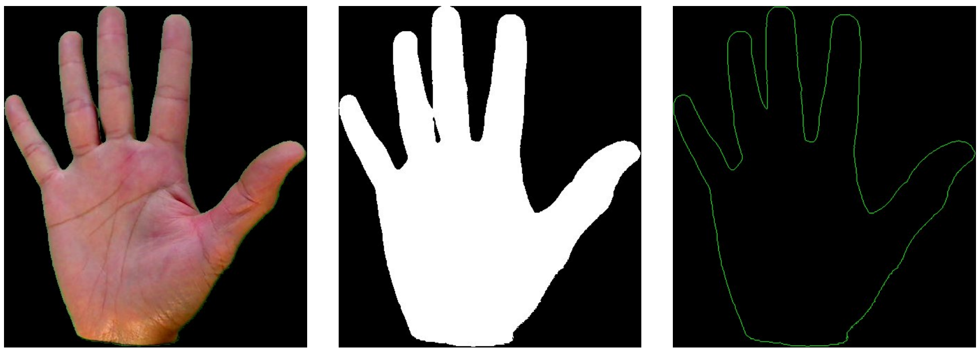

Figure 9.8: a) The original image. b) segmentation of a). c) contour of b)

the features invariant to these same transformations.

9.3 Exercises

1. Using the cvContour() function, write a simple Fourier descriptor function FD() for a certain contour.
2. Use the Fourier descriptors computed in the previous question to reconstruct the contour.
3. Implement the modified Fourier descriptors (the elliptic FDs). You can try some of the hand images as preparation for assignment 3.
4. Make a note of the first 20 elliptic FDs. Try the same contour rotated, then translated, then scaled. Are they the same descriptors?
5. Look at the first descriptors of the elliptic FDs. Does a pattern emerges?

Bibliography

[1] R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Prentice Hall, 2002.

[2] M. S. Nixon and A. S. Aguado, *Feature Extraction and image Processing for Computer Vision.* Elsevier, 2012.