Chapter 9: Frequency Domain

Andre L. C. Barczak

Computer Science
Massey University, Albany

Contents

The DFT general formula is:

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

Euler's identity:

$$e^{i\theta} = cos(\theta) + i \ sin(\theta) \quad (2)$$

The inverse DFT general formula is:

$$f(u,v) = \frac{1}{NM} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{-i2\pi(ux/M+vy/N)} \tag{3}$$

Let's take a look at what results from the DFT transforms.

Single frequency image

This image was created with a single sinoidal curve (single frequency).

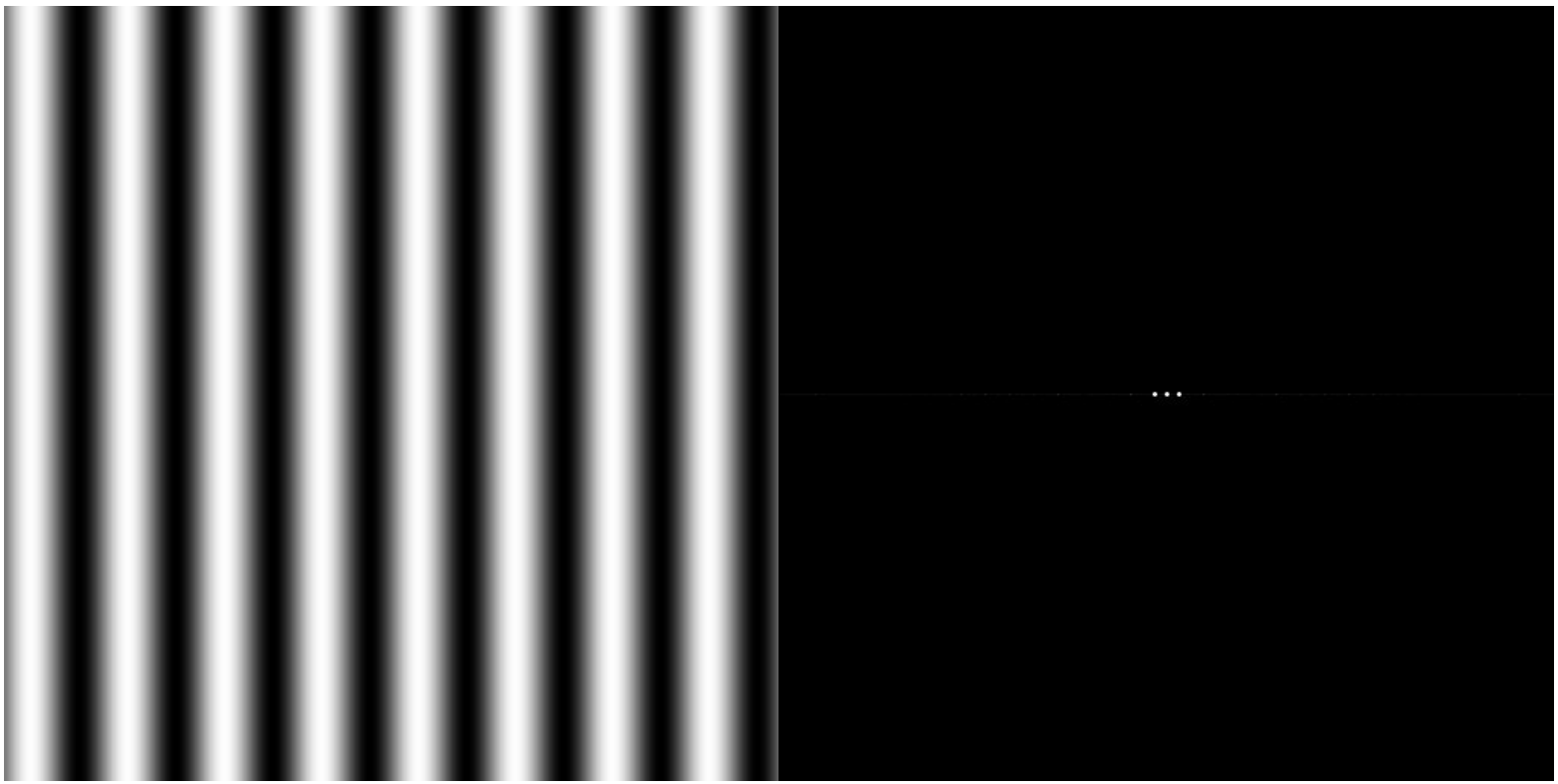

Figure 1: a) The original image. b) The FFT of a).

Single frequency image

This image was created with the same sin(), but rotated 45 degrees.

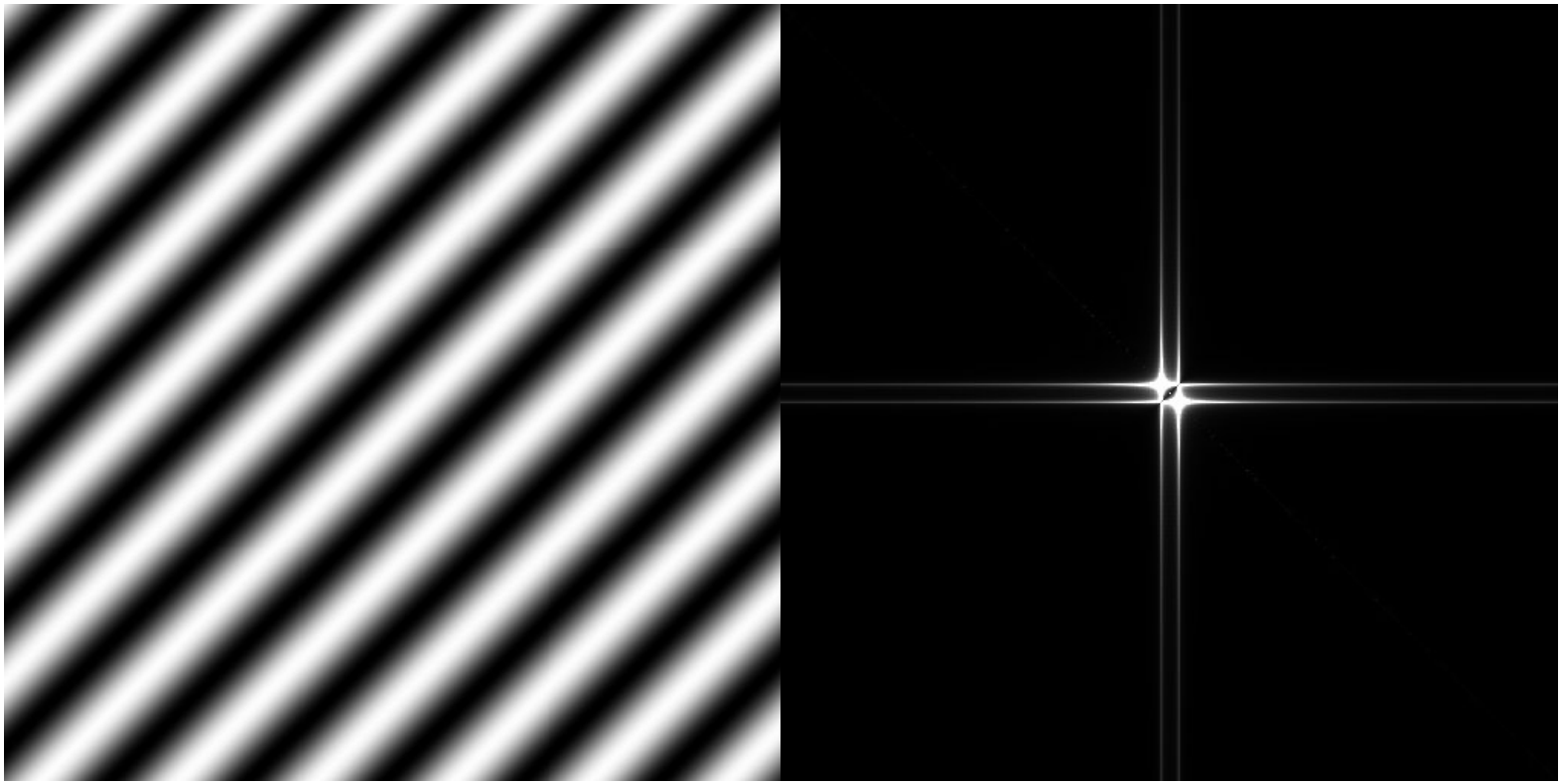

Figure 2: a) The original image. b) The FFT of a).

Square image

The square image has a different pattern:

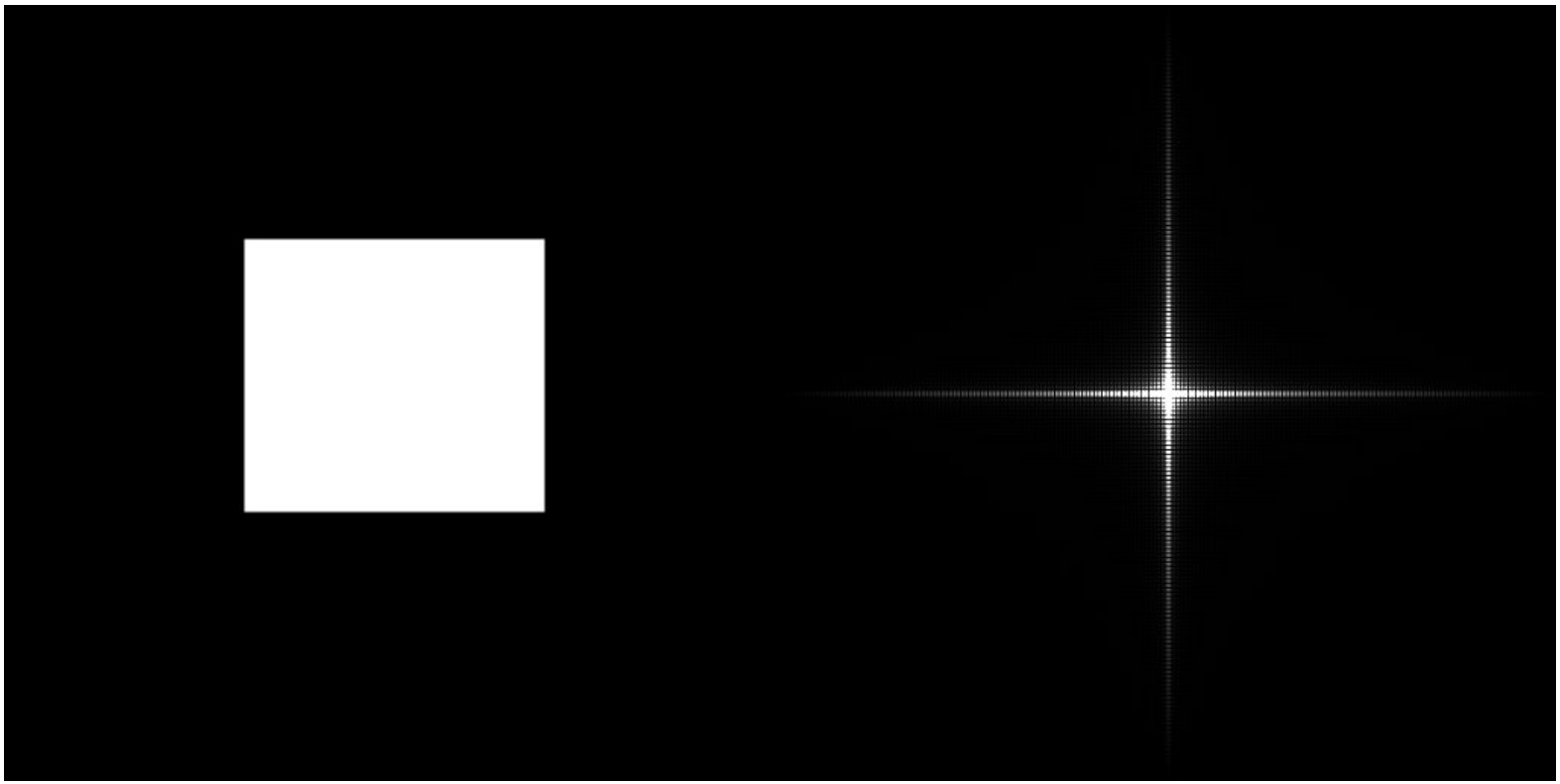

Figure 3: a) The original image. b) The FFT of a).

Other images (e.g., Akiyo)

The Akiyo image has a more spread frequency pattern:

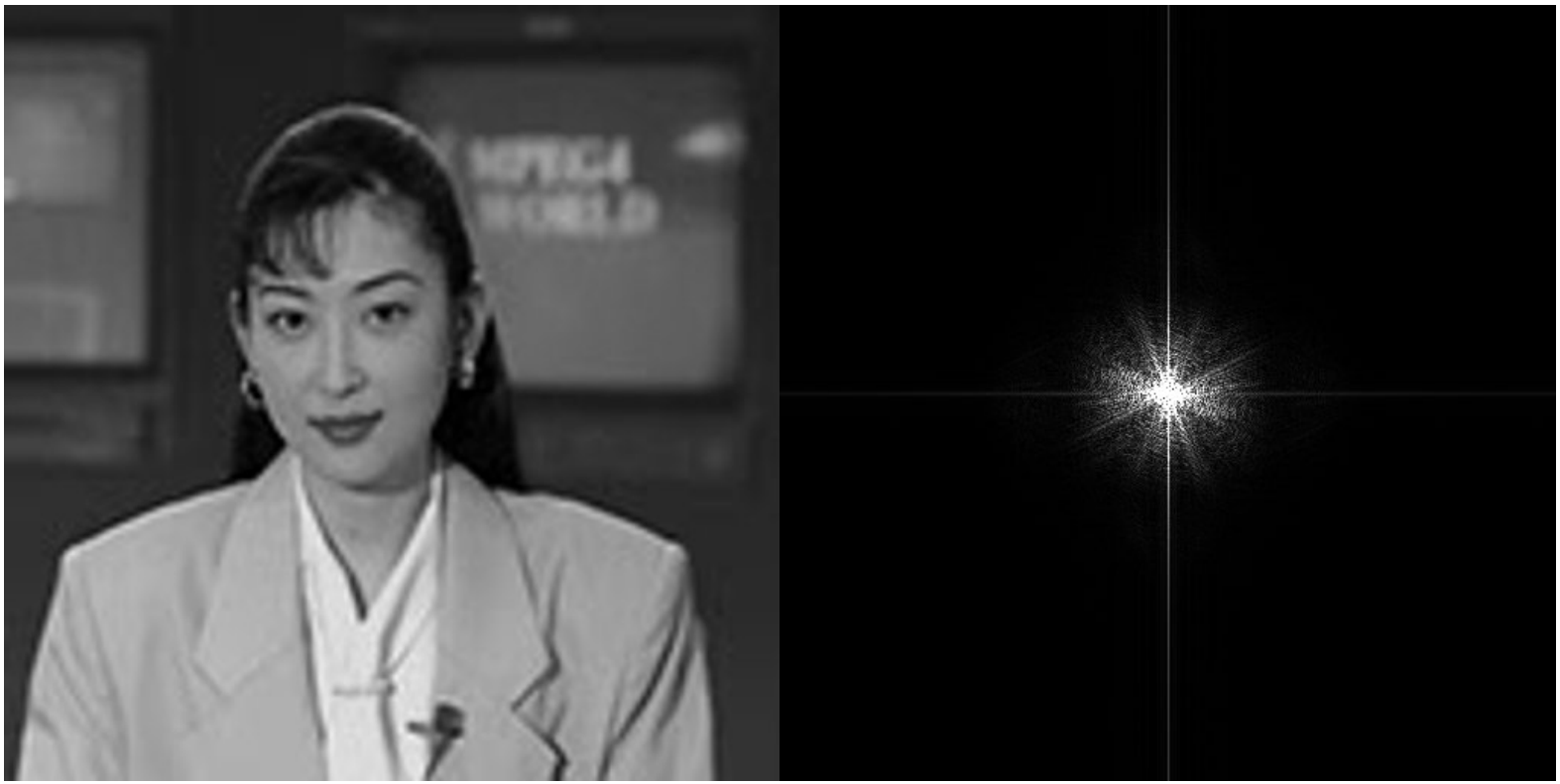

Figure 4: a) The original image. b) The FFT of a).

A C implementation

```
1  ...
2  // a complex pixel
3  typedef struct {
4    double real;
5    double imag;
6  } cpixel;
7  // a complex image
8  typedef struct {
9    int x,y;
10   cpixel **data;
11 } cimage;
12 // a pointer to a complex image
13 typedef cimage * cimg_ptr;
14 ...
```

A C implementation

```
1  ...
2  // create complex image
3  cimg_ptr create_cimage(int x,int y) {
4    cimg_ptr im;
5    int a;
6
7    im=new(cimage,1);
8    im->x=x;
9    im->y=y;
10   im->data=new(cpixel *,im->y);
11   im->data[0]=new(cpixel,im->x*im->y);
12   for( a=1;a < im->y;a++)
13     im->data[a]=im->data[a-1]+im->x;
14   return im;
15 }
16 ...
```

A C implementation

```
1  ...
2  void fft(double *data,int nn,int isign,int w){
3  ...
4    while (n>mmax) {
5  ...
6      for(m=1;m<mmax;m+=2) {
7        for(i=m;i<=n;i+=istep) {
8  ...
9        }
10       wr=(wtemp=wr)*wpr-wi*wpi+wr;
11       wi=wi*wpr+wtemp*wpi+wi;
12     }
13     mmax=istep;
14 ...
```

A C implementation

```
1  ...
2  // 2d FFT of a complex 512*512 image
3  void fft2d(cimg_ptr i,int dir) {
4   int j;
5   for(j=0;j<i->y;j++)
6      fft((double *)(i->data[j]),512,dir,1);
7   for(j=0;j<i->x;j++)
8      fft((double *)(i->data[0])+j*2,512,dir,512);
9  }
10 ...
```

Reconstruction and Filtering

From the coefficients, we can reconstruct the original image perfectly.

If we take smaller coefficients away, the reconstruction will be slightly different than the original image.

Properties of FFTs

One strong reason to use FFTs is that we can do convolutions of images with functions:

$$f(x,y) * g(x,y) \Leftrightarrow F(u,v).G(u,v) \tag{4}$$

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 $g(x,y)$:

$$f(x,y) * g(x,y) = FT^{-1}(F(u,v).G(u,v)) \tag{5}$$

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

Properties of FFTs

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

$$f(x,y) *^{-1} g(x,y) = FT^{-1}(F(u,v) \div G(u,v)) \tag{6}$$

Using these properties for filters

Using the properties of the convolution, filters can be used. The following steps apply:

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.

Unblurring images

Another 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.

Deconvolution 1

The Akiyo image was blurred using a Gaussian:

Figure 5: a) The blurred image. b) The deconvolution of a).

Deconvolution 2

The Clown is blurred because the original photo was taken with the wrong adjustments:

Figure 6: a) The original image. b) The deconvolution of a).

Fourier Descriptors

Introduction

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 are complex number, it can be written as a sequence of pairs:

$$s(k) = x(k) + iy(k) \tag{7}$$

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

$$a(u) = \sum_{k=0}^{K-1} s(k) e^{-i2\pi uk/K} \tag{8}$$

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

$$s(k) = \frac{1}{K} \sum_{u=0}^{K-1} a(u) e^{i2\pi uk/K} \tag{9}$$

The descriptors

Any implementation has to compute the real and the imaginary parts separately:

$$a_{real}(u) = \sum_{k=0}^{K-1} x(k)cos(\theta) - y(k)sin(-\theta) \tag{10}$$

$$a_{imag}(u) = \sum_{k=0}^{K-1} x(k)sin(-\theta) + y(k)cos(\theta) \tag{11}$$

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

Reconstruction of the original contour

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

$$x(k) = \sum_{u=0}^{K-1} a_{real}(u)cos(\theta) - a_{imag}(u)sin(\theta) \tag{12}$$

$$y(k) = \sum_{u=0}^{K-1} a_{imag}(u)cos(\theta) + a_{real}(u)sin(\theta) \tag{13}$$

Elliptic Fourier Descriptors

Another form of Fourier descriptors:

$$a_x(k) = \frac{1}{K}\sum_{u=0}^{K-1} x[k]cos(\theta); \tag{14}$$

$$b_x(k) = \frac{1}{K}\sum_{u=0}^{K-1} x[k]sin(\theta); \tag{15}$$

$$a_y(k) = \frac{1}{K}\sum_{u=0}^{K-1} y[k]cos(\theta); \tag{16}$$

$$b_y(k) = \frac{1}{K}\sum_{u=0}^{K-1} y[k]sin(\theta); \tag{17}$$

Elliptic Fourier Descriptors

The feature set is composed by the previous elements:

$$CE(k) = \sqrt{\frac{a_x^2(k) + a_y^2(k)}{a_x^2(0) + a_y^2(0)}} + \sqrt{\frac{b_x^2(k) + b_y^2(k)}{b_x^2(0) + b_y^2(0)}} \tag{18}$$

This form is called Elliptic Fourier descriptors, and they should be invariant to certain geometric transformations (rotation, scaling and translation).

A C implementation

Listing 1: EFDs function

```
1  void EllipticFourierDescriptors(vector<Point>&
       contour, vector<float> CE){
2   vector<float> ax, ay, bx,by;
3   int m=contour.size();
4   int n=20;//number of CEs
5   float t=(2*PI)/m;
6   for(int k=0;k<n;k++){
7     ax.push_back(0.0);      ay.push_back(0.0);
8     bx.push_back(0.0);      by.push_back(0.0);
9     for (int i=0;i<m;i++){
10      ax[k]=ax[k]+contour[i].x*cos((k+1)*t*(i));
11      bx[k]=bx[k]+contour[i].x*sin((k+1)*t*(i));
12      ay[k]=ay[k]+contour[i].y*cos((k+1)*t*(i));
13      by[k]=by[k]+contour[i].y*sin((k+1)*t*(i));
14    }
```

A C implementation

Listing 2: EFDs function

```
1        ax[k]=(ax[k])/m;
2        bx[k]=(bx[k])/m;
3        ay[k]=(ay[k])/m;
4        by[k]=(by[k])/m;
5     }
6     for(int k=0;k<n;k++){
7        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])) )
             ;
8     }
9  }
```

Contour example

This is an example of finding the contours to computing the CEs:

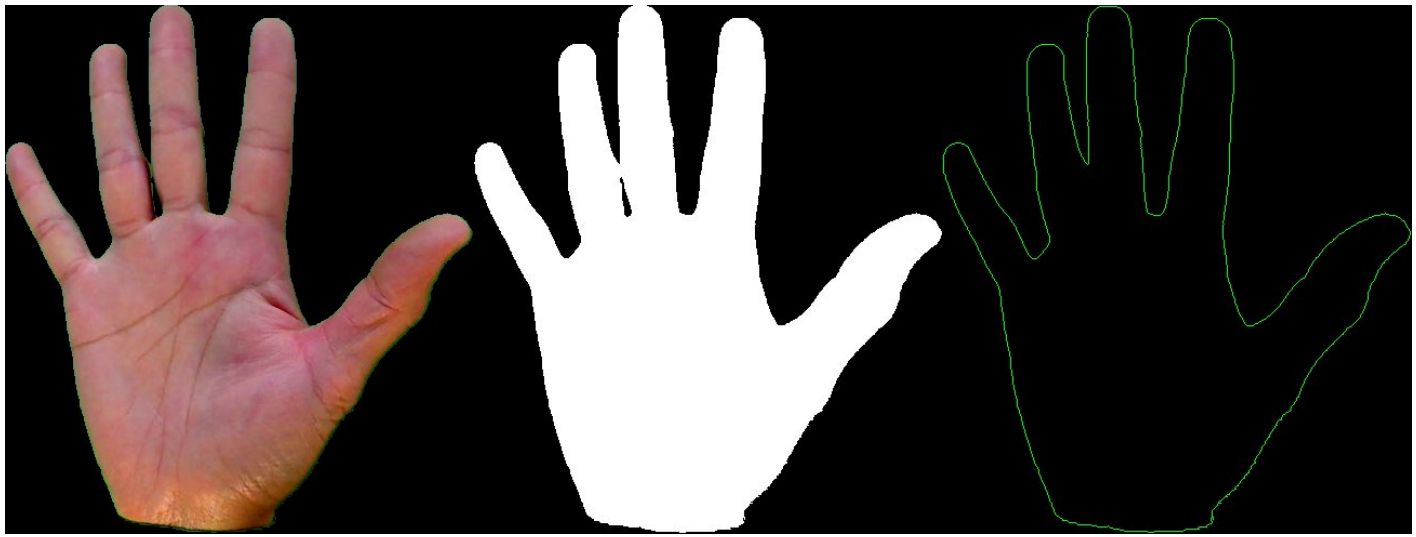

Figure 7: a) The original image. b) segmentation of a). c) the contour of b)

Contour example

The first 20 CEs are:

	CEs	a_x	a_y	b_x	b_y
0	2.000000	-26.867142	-83.252609	-64.881905	15.509949
1	0.740795	15.734508	23.033260	23.536472	-15.436429
2	0.314882	-5.785051	-5.617484	-10.328711	10.678829
3	0.410595	-3.864759	-0.312770	3.583835	-24.169739
4	0.173839	6.376639	-11.651113	0.019424	1.468244
5	0.323765	0.505321	-6.006077	-3.000287	16.735275
6	0.223076	-5.339873	-6.691420	4.404314	7.097655
7	0.156356	-0.294037	-9.179513	-3.371680	0.612619
8	0.042026	0.842521	0.775437	-1.666114	-0.974922
9	0.062318	1.230477	1.438030	-0.350696	-2.691195
10	0.055213	-0.319801	-3.423424	-1.055962	-0.106418
11	0.027776	-1.146091	-0.355667	-0.516705	-0.782705
12	0.044766	0.076189	1.358662	-0.107865	-1.945668
13	0.025255	-1.055612	1.034870	0.143197	-0.538780
14	0.021453	0.309870	-0.186862	-0.230818	1.131926
15	0.031525	0.419653	-1.488751	-0.913262	0.137147
16	0.021350	-0.556561	-0.887044	-0.435711	0.449094
17	0.020267	0.042636	-0.606309	-0.092211	0.883727
18	0.020407	0.301367	0.451174	-0.622980	0.714058
19	0.008485	0.229620	0.289056	0.284142	0.014188

Rotation, translation, scaling

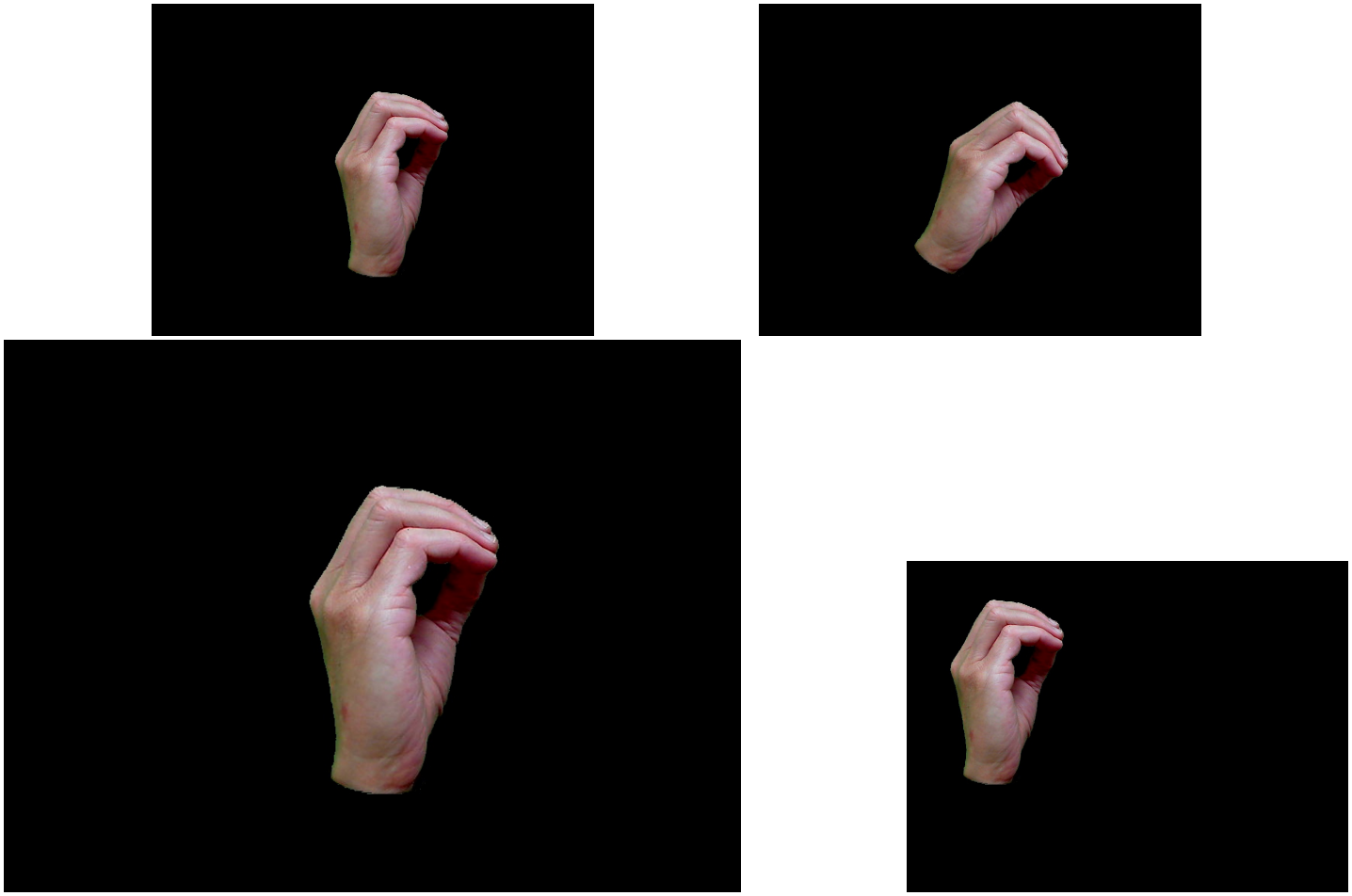

Figure 8: The same gesture was rotated, scaled and translated

Contour example

The first 20 CEs for different images of the same gesture:

	Original	rotated 45°	scale 2 x	translated
0	2.000000	2.000000	2.000000	2.000000
1	0.163645	0.176821	0.162909	0.158473
2	0.152705	0.159280	0.163765	0.167638
3	0.079536	0.078102	0.078379	0.084351
4	0.046756	0.043191	0.045626	0.048122
5	0.026473	0.034413	0.037960	0.037403
6	0.019169	0.026394	0.019026	0.021815
7	0.011909	0.016739	0.018974	0.016899
8	0.016845	0.009933	0.008501	0.014808
9	0.020322	0.014583	0.016915	0.015765
10	0.005140	0.004881	0.004200	0.006596
11	0.009187	0.004075	0.005674	0.008157
12	0.007175	0.009863	0.008454	0.010328
13	0.008993	0.006124	0.006408	0.007773
14	0.005090	0.003951	0.003585	0.003849
15	0.007007	0.005306	0.004392	0.005189
16	0.004254	0.005081	0.002545	0.002500
17	0.004344	0.004918	0.005087	0.004582
18	0.001941	0.003368	0.002813	0.002937
19	0.003016	0.002441	0.002410	0.002309

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?