Digital Signal Processing Fundamentals

Jin Mitsugi*

October 4, 2025

Contents

*Keio University, Faculty of Environment and Information Studies

1 Introduction

1.1 Objective of this class

Digital Signal Processing (DSP) is the central technology in image and voice recognition, sound processing, remote sensing and wireless communication. The penetrations of large scale integrated circuits (LSI), tiny and affordable sensors and computers and the high performance computing available with cloud services prevailing DSP everywhere.

However, because of the discrete nature of digital signal, DSP needs to be properly implemented and the results should be properly comprehended. Otherwise, DSP may lead us to a solution which is artificially produced by its discrete nature.

In this class, students learn the theory and principles of digital signal processing through hands on using `MATLAB`. This class proceeds with the hands-on-first, theory-later policy. On every topic, we basically start with the implementation of signal processing system or application , then later study how and why it works that way with the minimum set of theoretical study.

After taking this class, students can produce practical DSP applications using either existing library in `MATLAB` or custom made software which is backed up with solid DSP theory.

1.2 Advanced readings

This lecture note is designed to be self-complete. Because of the wide range of DSP topics in this introductory class, however, the explanation on each of the topics may be shallow. If you would like to deepen your understandings further, the followings are the recommended reading.

- fundamental digital signal processing. detailed theory derivation.: J.G.Proakis and D.G. Manolakis, *Digital Signal Processing, 4th edition*, Prentice Hall, (2007)[3].
- digital communication, detailed theory derivation: J.G.Proakis and D.G. Manolakis, *Digital Communications, 5th edition*, MacGraw-Hill, (2008). [4]
- digital signal processing. practical and implementation oriented.: R.G.Lyons, *Understanding Digital Signal Processing, 3rd edition*, Prentice-Hall, (2010). [5]
- image processing. implementation oriented: A.Kaehler and G.Bradski, *Learning Open CV3*, O'Reilly, 2017[2]
- pattern recognition. classic: C.Bishop, *Pattern Recognition and Machine Learning*, Springer, (2006). [1]

1.3 MATLAB Installation

Keio students are eligible to install and use `MATLAB` with site license student option. The link `https://slc.keio.jp/slc/getSoftwareLicense` explains in detail how to obtain a license key and the installation. Please go to the `Category` from the top page of `keio.jp`. Then, in `Software Licensing Center` there should be `MATLAB` license. After agreeing the term and conditions, you should see an activation key with valid expiration date. The code is needed when you activate `MATLAB`. In order to download and install MATLAB we need to have an account in Mathworks, the developer of MATLAB, by visiting `http://www.mathworks.co.jp/`. You must create an account with `keio.jp` mail address. After creating an account and confirm the account is successfully created by checking notification mail, you can login to `Mathworks` home page. If you go to the account page, you should see `Associate with a License` link in the License Center. Please register the activation key you have obtained earlier. After registering the activation key, download MATLAB software from the support page. If your PC has sufficient space, you can download all the toolbox but it is not recommended because MATLAB is a big application. MATLAB consumes about 20 GB in my PC. Please download at least the followings for the class.

- Audio Toolbox

- Communication Toolbox
- Computer Vision Toolbox
- Deep Learning Toolbox
- DSP System Toolbox
- Image Processing Toolbox
- Signal Processing Toolbox

In every year, one or two students have a hard time to squeeze their disk space for `MATLAB`. If you have a limited disk space, I recommend to have an external SSD drive as early as possible, or replace the internal SSD with a bigger one if you can[1].

1.4 Fundamentals of MATLAB/Simulink

MATLAB is a programming language and also a collection of gigantic libraries (functions and toolkits). We write scripts with MATLAB language, which is similar to `javascript` and `python` using many utilities that MATLAB provides in a `.m` file. As you learn in `MATLAB Onramp` there is a sophisticated way to combine scripts and output by using `MATLAB live script`, which is a `.mlx` file. Simulink is an environment with which we can do simulations without writing `.m` files but by connecting `block set` with visual interface as shown in Fig.1.1. In this class, we use `MATLAB` with `.m` files to avoid excessive dependence on MATLAB/Simulink.

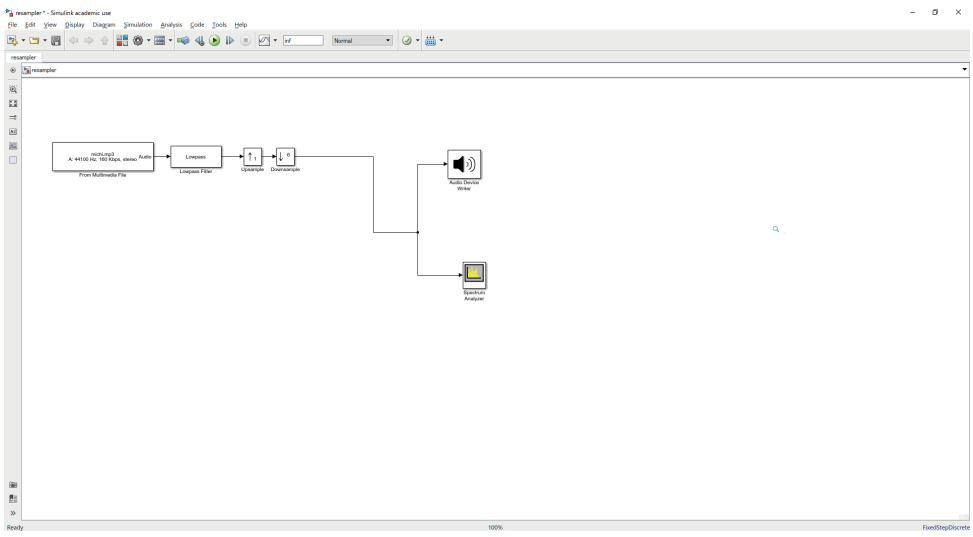

Figure 1.1: In Simulink, a model can be built by connecting blocksets

The first exercise of this class is to complete an online training course, `MATLAB Onramp`. After you successfully login in the `Mathworks` home page, go `Learn`/`Self-paced Online Courses`, `MATLAB Onramp` as in Fig. 1.2. You are encouraged to go through `How and Why of Writing Functions` too.

The training in `MATLAB Onramp` comprises 15 sections. Each section and subsection accompany "Tasks" for your training as shown in Fig.1.3. Final project is to calculate a star moving speed from a light spectrum Doppler shift, which is very interesting to learn.

[1]Mechanical replacement of SSD of Windows PC is relatively easy, but copying the contents of the old SSD to a bigger SSD may demand disk partitioning and master boot record rewriting. Unless you know what you are doing, asking help for an expert is, in general, a good idea.

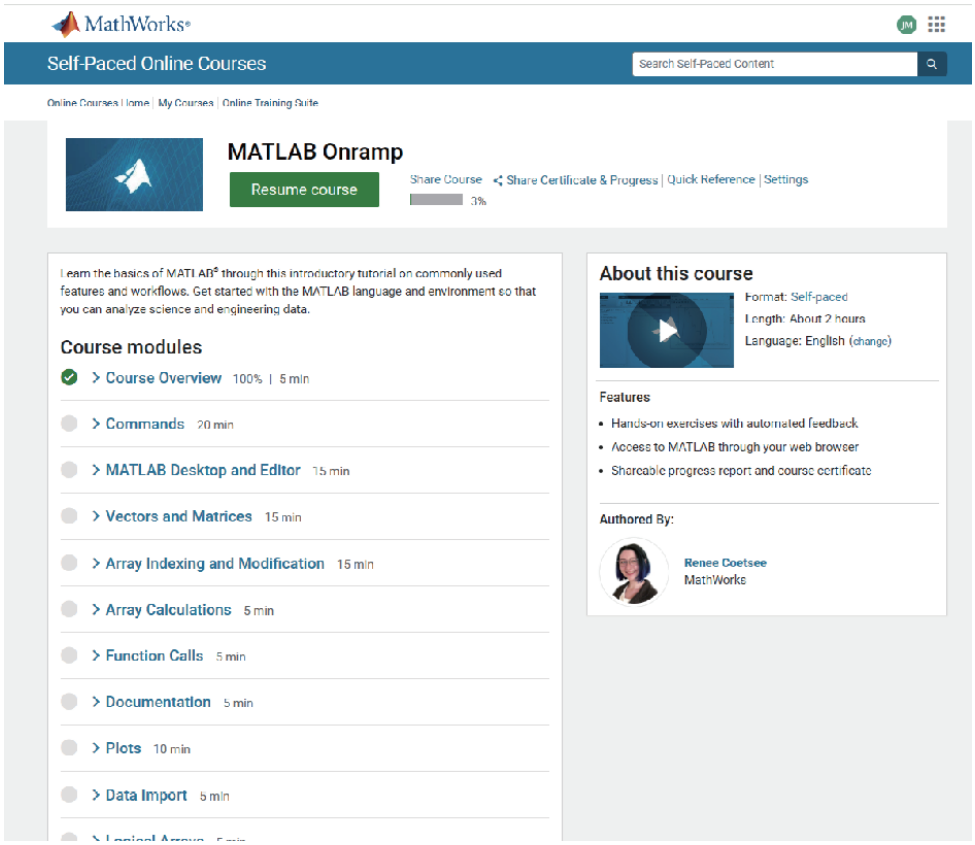

Figure 1.2: MATLAB online tutorials

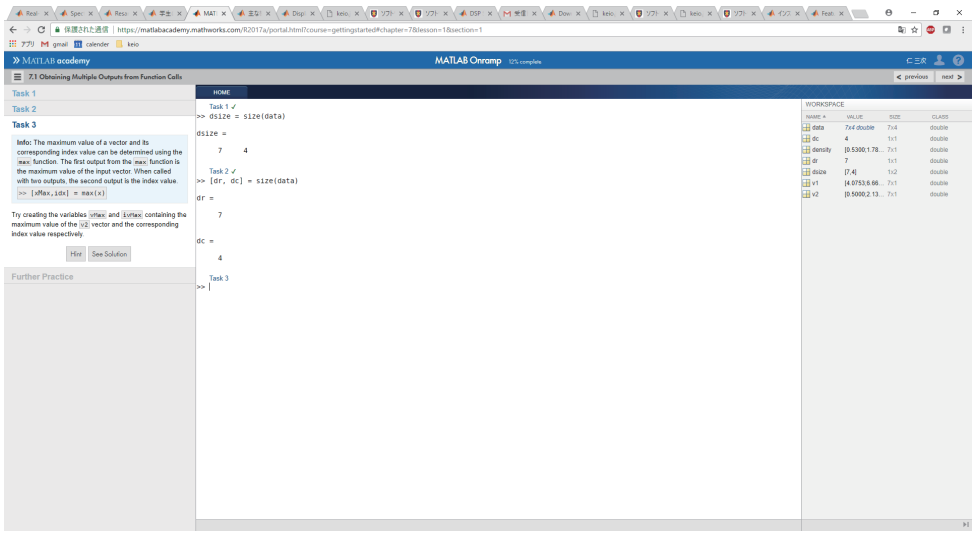

Figure 1.3: MATLAB training screen example

1.5 Tips and caveats

There are popular mistakes of `javascript` or `python` programmers who just start working with MATLAB.

- array index starts at 1 instead of 0 as is the case of most of the programming languages.
- if you define an external function (different `.m` file other than the script file you are using), the function name SHALL be the file name of the `.m` file.
- comment is `MATLAB` is % or three consecutive dots `...` to split one command into several lines. Upper and lower cases matter.
- there might be a language problem to use `MATLAB`. Try the followings.
  - When you encounter a language problem and `MATLAB` application, it should relate to the locale setting of your PC. The problem may be solved by setting the language `HOME/Preferences`
  - Language setting in Self-paced Online Courses can be changed in each of the page as . You also need to check the language setting of your browser.

2 Working with signals

2.1 Generate a tone

Sound is a wave which changes the air pressure (dense-thin-dense) in time and also propagates in the air. How many times a wave vibrates (change pressure) a second is denoted by "frequency" with unit Herz (Hz). The time for one cycle of vibration is referred to as period. The speed of sound in dry air is about 331 m/s. If the frequency

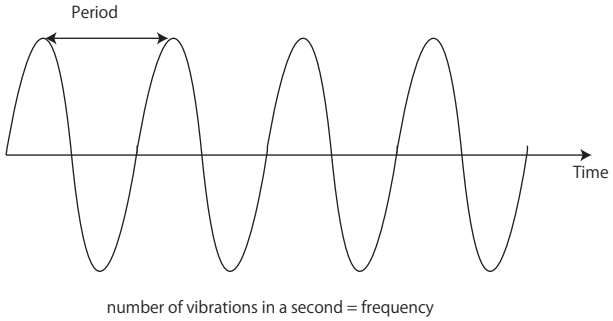

Figure 2.1: Frequency and period definitions.

of sound is 440Hz, the wavelength is 0.75 m (=331/440). This means the pressure fluctuates at every 0.75 m as well as in time.

A single tone whose frequency is fHz observed at a geographical point can be mathematically represented by a sine (or cosine) wave such that

$$y = \sin(2\pi ft). \tag{2.1}$$

With a microphone, a sound (or the change of air pressure) can be captured and converted to an electronic signal usually a voltage. To digitally process the signal with computer (processor), every signal has to be transfered to a digital form, which has two aspects. One is discretization which is a time-wise sampling. Usually discretization is performed with a constant time interval which is denoted as sampling period and its reciprocal is sampling rate. The other is a quantization. Every sample point has to be handled as a finite digit number as shown in Fig.2.2. After the discretization and quantization, a signal can be represented with a time series vector y_d

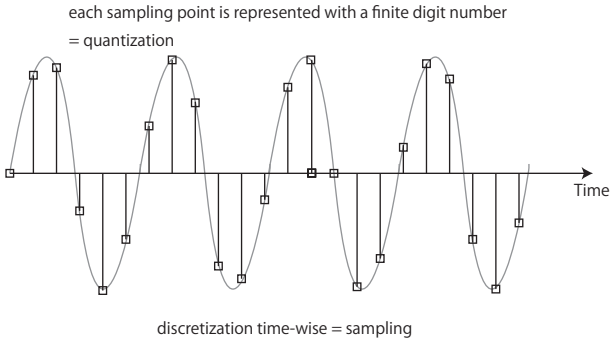

Figure 2.2: Signal can be processed by computers through discretization and quantization.

$$\boldsymbol{y_d} = \{\boldsymbol{y_0}, \boldsymbol{y_1}, \boldsymbol{y_2}, \boldsymbol{y_3}, \cdots\}, \tag{2.2}$$

each number represents a finite digit number of a sampling point. In this class, the sampling rate needs to be explicitly specified while the quantization is taken care almost automatically by the computer. We just need to specify how much memory space can be allocated to represent a sample. Usually it is 4bytes = 32bit floating point number.

Let us create a tone with 440Hz with MATLAB.

Example 2.1 *(tone440.m)*
The following script generates a sound of 440Hz for 2 seconds, with sampling rate 8000 Hz.

```
%%  tone440.m
%
%   generate a 440Hz tone
Fs = 8000; % sampling rate
f = 440; %tone frequency
t = (0:1/Fs:2); % from 0 to 2 second
y = sin(2*pi*f*t);
sound(y, Fs);  % generate sound
plot(t(1:100), y(1:100)); % we only draw first 100 samples
```

The first 100 samples plot is given as shown in Fig.2.3. `sound` function by default sends the audio data to the

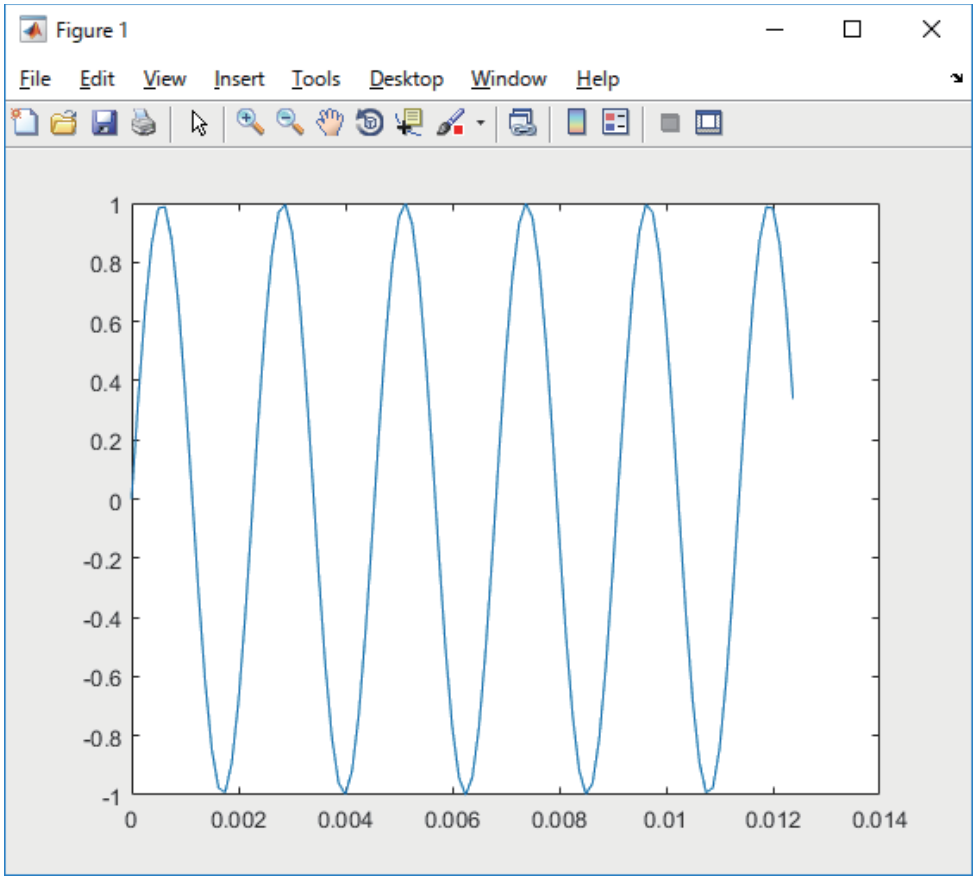

Figure 2.3: Wave shape of 440Hz tone.

sound card (speaker) with the default sampling rate of 8182 Hz. By specifying the sampling frequency such that `sound(y, Fs)`, the sampling frequency can be changed. Note that the available sampling frequency of MATLAB is from 1000 Hz to 384000 Hz and there might be limitation of your sound card, usually can handle from 5 to 48 kHz sampling rate. The audio data is a `m` row 1 column for monaural, `m` row 2 columns for stereo. The first column is the left channel and the second column is the right channel. y vector is supposed to be double array but they are converted to integer array with specified bit depth, default 16 bits.

One octave high tone doubles the frequency. 880 Hz is one octave high from 440Hz (A tone). The equal temperament divides the 12 half tones frequency-wise by

$$\sqrt[12]{2}. \tag{2.3}$$

Since the equal temperament involves irrational numbers, some say it does not produce good harmony. Just intonation is another principle to generate tones where the tone frequencies are rational numbers such that

Table 2.1: Just intonation

c	d	e	f	g	a	b	c
1	$\frac{9}{8}$	$\frac{5}{4}$	$\frac{4}{3}$	$\frac{3}{2}$	$\frac{5}{3}$	$\frac{15}{8}$	2

A simple harmony can be created with the following script.

Example 2.2 *(harmonyA.m)*
The following generate a harmony in equal temperament with three tones.

```
%%  harmonyA.m
%
%   genrate a 440Hz tone
Fs = 8000; % sampling rate
sep = power(2, 1/12);
A = 440; %tone A frequency
Cs= A*power(sep, 4); % tone C#
E = Cs*power(sep, 3);  % tone E
t = (0:1/Fs:2); % from 0 to 2 second
  % we reduce the amplitude by 0.2 to avoid distotion.
  % add the three tones
y = 0.2*(sin(2*pi*(A)*t)+ sin(2*pi*(Cs)*t)+sin(2*pi*(E)*t));
sound(y, Fs);  % generate sound
plot(t(1:100), y(1:100)); % we only draw firt 100 samples
```

The wave shape of the harmony is the sum of the three tones and still repetitive as shown in Fig.2.4.

It is very important to understand every time series signal is a linear combination of fundamental tones. If we denote $y(t)$ is an arbitrary time series and $f_i(t)$ denotes the $i-th$ fundamental tone, the following is valid by choosing the coefficients a_i properly.

$$y(t) = a_0 f_0(t) + a_1 f_1(t) + \cdots \tag{2.4}$$

The choice of fundamental tone depends on the nature of the test signal. Most popular fundamental tones are the trigonometric functions such that

$$f_n(t) = \cos(2\pi nt/T) \tag{2.5}$$
$$f_n(t) = \sin(2\pi nt/T) \tag{2.6}$$

where T denotes the signal period. In some applications where signal involves abrupt changes in time as shown in Fig.2.5. In such case, we use an wavelet analysis [6] to obtain the time and frequency component simultaneously by scaling and shifting a mother wavelet as shown in Fig. 2.6). Piecewise Fourier transformation, commonly known as shoft-time Fourier transform, is another way to handle general time domain signals.

If we use a vector (discrete) form to represent a time sequence, Eq2.4 can be written as follows.

$$\begin{Bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{Bmatrix} = \begin{bmatrix} f_{00} & f_{01} & \cdots & f_{0(n-1)} \\ f_{10} & f_{11} & & \\ \vdots & & \ddots & \\ f_{(n-1)0} & f_{(n-1)1} & \cdots & f_{(n-1)(n-1)} \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{Bmatrix}, \tag{2.7}$$

where f_{ij} denotes the $i-th$ time component of $j-th$ component tone.

There may be situations where we need to consider nonlinear effects, such as distortion and inter-modulation. But they can be taken care in an advanced course.

Exercise 2.1 *(harmonyAjust) Compose a script to generate A minor (the combination of A + C + E) both in just intonation and equal temperament sequentially, and compare the difference.*

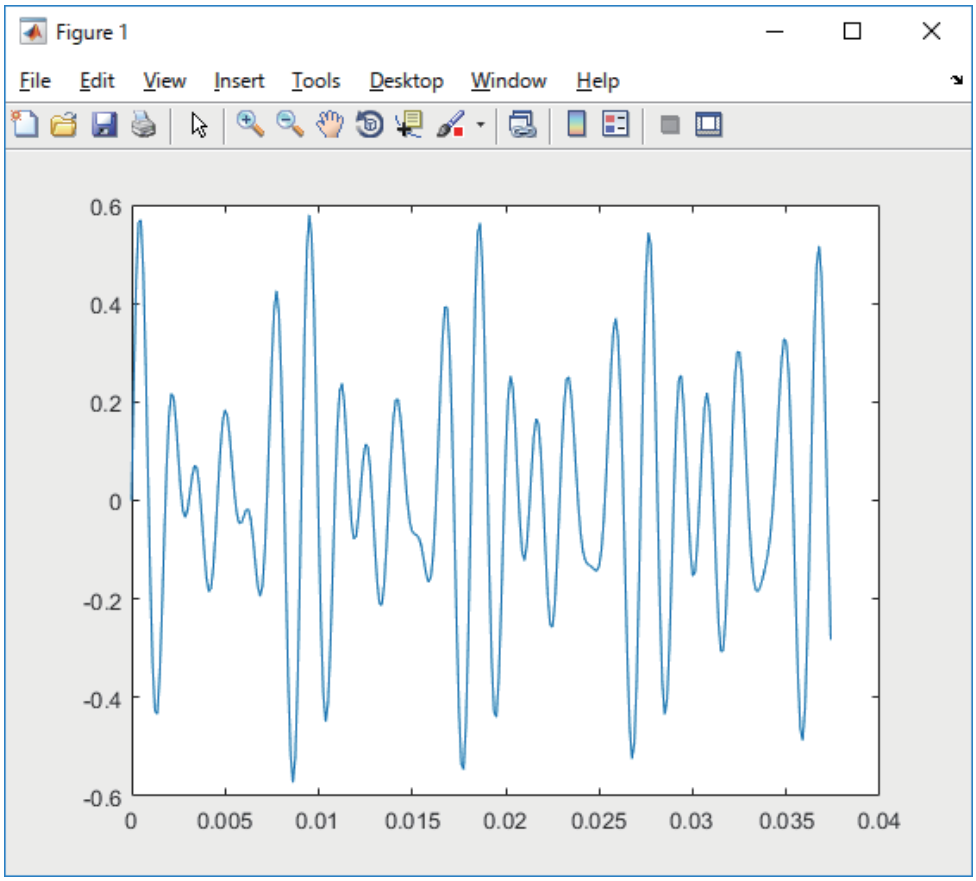

Figure 2.4: Wave shape of harmony A.

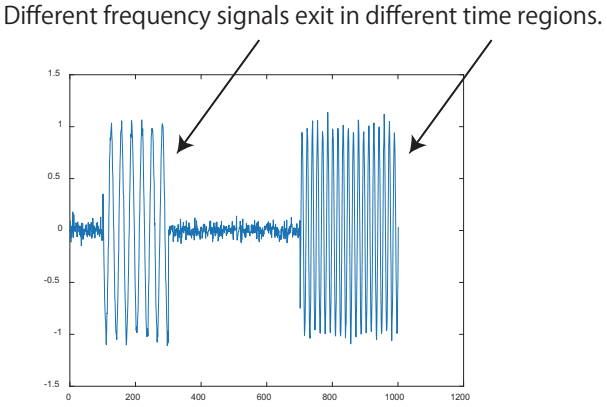

Figure 2.5: Abruptly changing signal example.

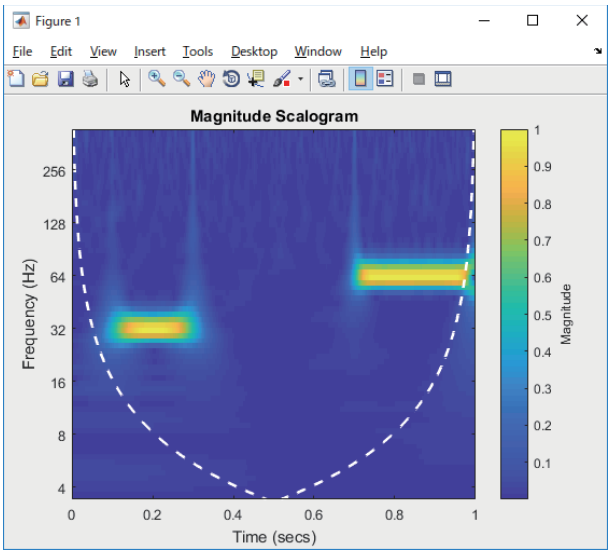

Figure 2.6: Time and frequency coefficients after Wavelet transformation.

2.2 Voice message record and playback

The next exercise is to record your voice in your PC and play back afterwards. Once a signal is captured as digital data. Signal processing can be applied regardless the original source of the signal whether it is a generated tone or human voice.

Example 2.3 *Create a new script file and name it* `recandplay.m`*. Type in the following*

```
% example of voice record
recObj = audiorecorder      % create a recorder object
disp("Start speaking");     % print out a start message
recordblocking(recObj, 5); % record the voice for 5 seconds
disp("Stop recording");     % print out a stop message after 5 second
play(recObj);               % play back the voice
```

Note that % in MATLAB script denotes a comment line.

Type in the script name (`recandplay`) in your command window as in Fig.2.7 to let it be executed.

Your message is recorded for 5 seconds and immediately playback. You see the following message in your command window, which describes how `MATLAB` records your voice.

```
 audiorecorder with properties:

          SampleRate: 8000
       BitsPerSample: 8
    NumberOfChannels: 1
            DeviceID: -1
       CurrentSample: 1
        TotalSamples: 0
             Running: 'off'
            StartFcn: []
             StopFcn: []
            TimerFcn: []
         TimerPeriod: 0.0500
```

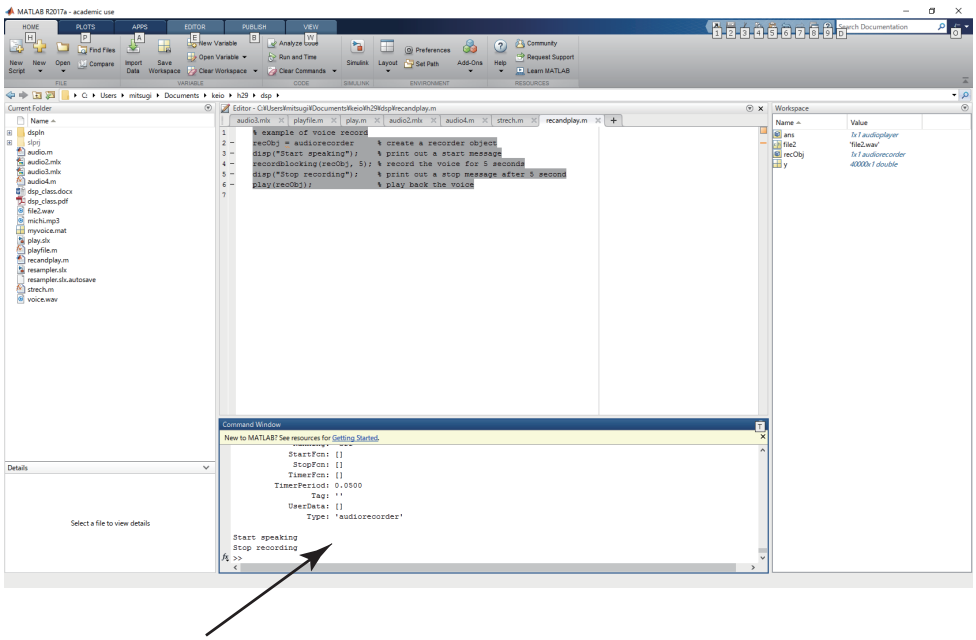

Figure 2.7: After creating the script. Type in the script name in the command window.

```
      Tag: ''
 UserData: []
     Type: 'audiorecorder'
```

The message says that your voice is discretized for 8000 times a second `SampleRate:8000`. Each time sample is digitally represented with 8 bit `BitsPerSample:8`. Since you recorded your message for five seconds, the total number of samples representing your voice is 40000 = 8000 × 5. Since voice is a sequence of air pressure change, voice can be represented by a sequence of pressure which is captured by the microphone of your PC. The time sequence of your voice can be extracted by adding the following script.

Example 2.4 *(recandplay_rev) Recorded voice data is visualized.*

```
 %% recandplay.m
 % example of voice record
recObj = audiorecorder      % create a recorder object
disp("Start speaking");     % print out a start message
recordblocking(recObj, 5); % record the voice for 5 seconds
disp("Stop recording");     % print out a stop message after 5 second
play(recObj);               % play back the voice
% obtain the pressure sequence as y array
y = getaudiodata(recObj);
plot(y) % visualize the array
audiowrite('voice.wav', y, 8000);  % save the audio data as a wave file.
```

The last line of Example 2.4 saves the voice file as 'voice.wav'. By doing so, you don't have to record your voice every time you process the signal. The following script `playfileplain.m` plays back the recorded voice.

Example 2.5 *(playfileplain) Function* `audioread` *reads an audio file and recover the time sequence data and its original sampling rate. Function* `sound` *plays the time sequence with the specified sampling rate.*

```
%% playfileplain.m
% play a recorded sound
[y, Fs] = audioread('voice.wav')
sound(y, Fs)
```

If you visualize the time sequence by adding `plot(y)` , it is like Fig.2.8 Since we have 40,000 samples (8,000

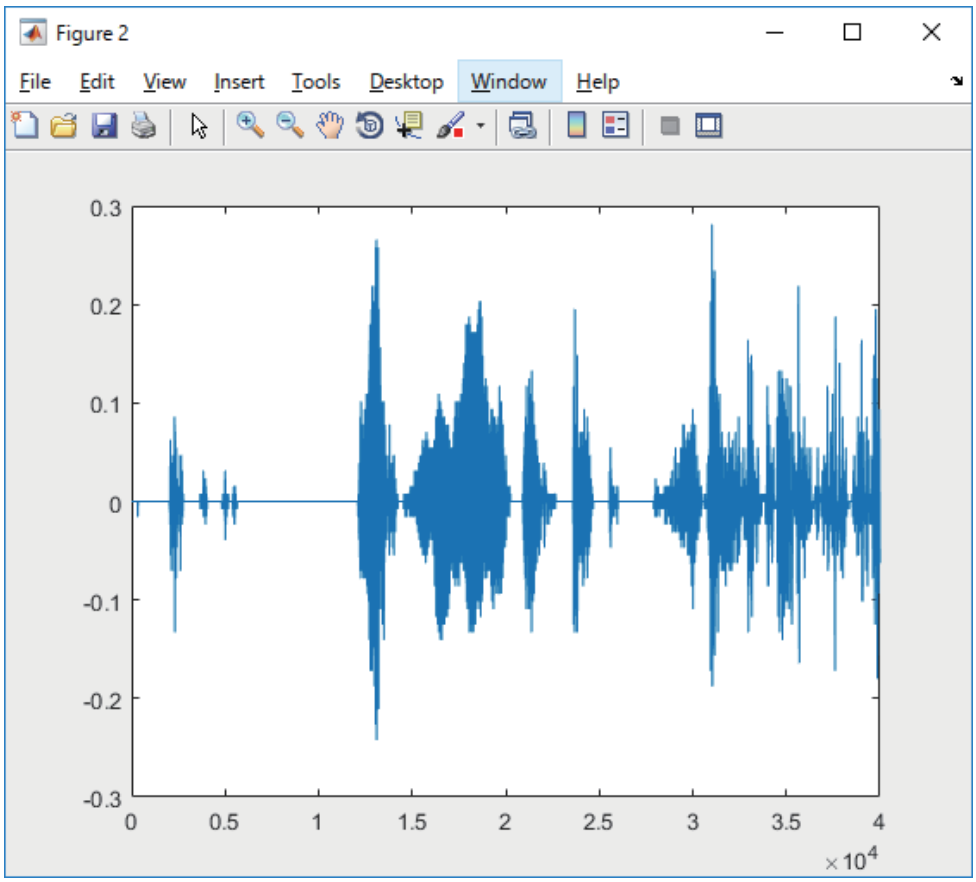

Figure 2.8: Example time sequence of a voice message.

sample a second × 5 seconds), the horizontal axis have 40,000 entries. Let us change the speed of play back by changing `Fs` in `sound`, for example

```
sound(y, 7000)
sound(y, 9000)
```

You should here the voice become lower at sampling rate 7000 and higher at sampling rate 9000. We can change the pitch (tone) of a voice by changing sampling rate. But with this simple method, we inevitably change the time duration of the message.

MATLAB tips: Capture a text from keyboard

The following prompts "input stretch" to the display and wait for keyboard input. The input string is captured as

```
coef = input('input stretch=');
```

3 Frequency domain signal

3.1 Discrete Fourier transformation

In the earlier example, we produce a signal at a specified frequency such as 440 Hz. In order to understand the distribution of the frequency components of a signal, we use Fourier Transformation. Fourier Transformation decomposes a fixed length time series of signal with a sampling period T_s

$$\boldsymbol{y} = \{\boldsymbol{y_0}, \boldsymbol{y_1}, \boldsymbol{y_2}, \boldsymbol{y_3}, \cdots, \boldsymbol{y_n}\} \tag{3.1}$$

into frequency components. The lowest frequency f_1 of the time series is determined by the duration of the signal, which is $L = T_s n$ irrespective to the nature of the signal as in Fig. 3.1.

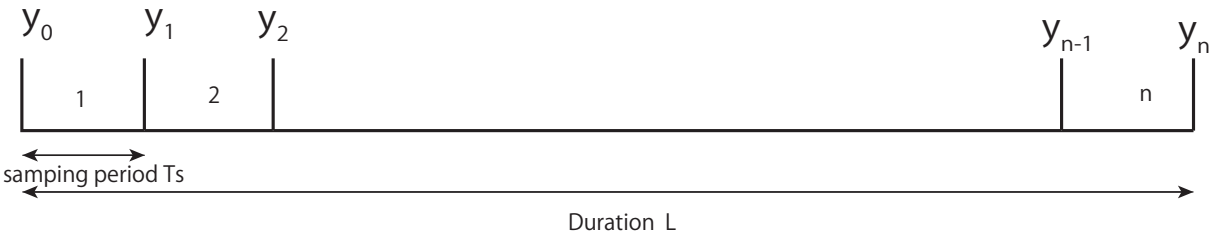

Figure 3.1: Sampling points, duration and sampling time in n+1 samples

The discrete frequency components are the multiple of the fundamental frequency such that

$$\begin{aligned} f_0 &= \frac{0}{nT_s} = \frac{0}{L} \\ f_1 &= \frac{1}{nT_s} = \frac{1}{L} \\ &\vdots \\ f_{n-1} &= \frac{n-1}{nT_s} = \frac{n-1}{L} \end{aligned} \tag{3.2}$$

Note that the first frequency f_0 is constant for the whole period L. Let us suppose we can decompose the given signal into a linear combination of sines and cosines of each frequency component such that

$$y = a_0 \cos 2\pi f_0 t + b_0 \sin 2\pi f_0 t + a_1 \cos 2\pi f_1 t + b_1 \sin 2\pi f_1 t + \cdots + a_{n-1} \cos 2\pi f_{n-1} t + b_{n-1} \sin 2\pi f_{n-1} t. \tag{3.3}$$

The cosines are illustrated in Fig. 3.2

For given y, the coefficients to each cosines and sines a_i, b_i can be computed such that

$$a_i = \frac{2}{L} \int_0^L y(t) \cos(2\pi \frac{i}{L} t) dt, \text{if } i \neq 0 \tag{3.4}$$

$$b_i = \frac{2}{L} \int_0^L y(t) \sin(2\pi \frac{i}{L} t) dt \tag{3.5}$$

$$a_0 = \frac{1}{L} \int_0^L y(t) \cos(2\pi \frac{i}{L} t) dt. \tag{3.6}$$

The coefficients $\frac{2}{L}$ and $\frac{1}{L}$ before the integral are derived as follows.

Suppose we have y, which exactly matches the waveshape of i-th Fourier component such that $y = \cos(2\pi \frac{i}{L} t)$ and $i \neq 0$, in this case a_i shall be equal to 1, but the integration of $\cos^2(2\pi \frac{i}{L} t)$ can be derived as follows

$$\int_0^L \cos^2(2\pi \frac{i}{L} t) dt = \int_0^L \frac{1 + \cos(4\pi \frac{i}{L})}{2} dt = \frac{L}{2}, \tag{3.7}$$

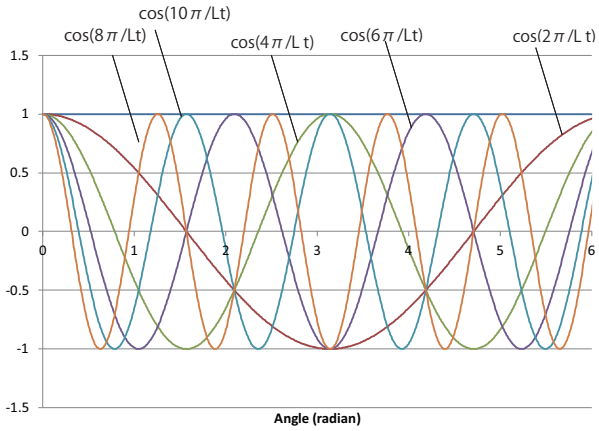

Figure 3.2: Cosine fundamental waves

where the double angle identify

$$cos^2(x) = \frac{1 + \cos(2x)}{2}, \tag{3.8}$$

is used.

Therefore, a_i shall be adjusted by $\frac{2}{L}$. In case of $i = 0$, the integration is

$$\int_0^L 1dt = L \tag{3.9}$$

therefore, only in the case of a_0, the integration shall be adjusted by $\frac{1}{L}$.

y and $\cos(2\pi\frac{i}{L}t)$ are already discretized with the same sampling rate in our problem. The integration can be replaced with the inner product of y and $\cos(2\pi\frac{i}{L}t)$.

$$\int_0^L y(t)\cos(2\pi\frac{i}{L}t)dt = \sum_{j=0}^{n-1} y_j \cos(2\pi\frac{i}{L}jT_s)Ts \tag{3.10}$$

The summation is from 0 to $n-1$ because it is an approximation of integration with the summation of rectangular boxes as shown in Fig. 3.3.

Since $L = nT_s$, we can rewrite Eqs. 3.4-3.6 as

$$a_i = \frac{2}{n}\sum_{j=0}^{n-1} y_j \cos(2\pi\frac{i}{L}jT_s) \text{ if } i \neq 0 \tag{3.11}$$

$$b_i = \frac{2}{n}\sum_{j=0}^{n-1} y_j \sin(2\pi\frac{i}{L}jT_s) \tag{3.12}$$

$$a_0 = \frac{1}{n}\sum_{j=0}^{n-1} y_j \tag{3.13}$$

Please note that we need to double the amplitude of a_i and b_i with respect to a_0.

By increasing the number of trigonometric functions, we can approximate arbitrary time domain waveshape. For example, if the original signal is a rectangular shape as shown in Fig.3.4, it can be approximated with a collection of cosines and sines.

This is a straightforward application of Eq.2.7 with two sets of coefficients a and b. The reason we'd like to have both cosines and sines is to exactly decompose a signal which has intermediate phase ϕ such that

$$y(t) = A\sin(2\pi ft + \phi). \tag{3.14}$$

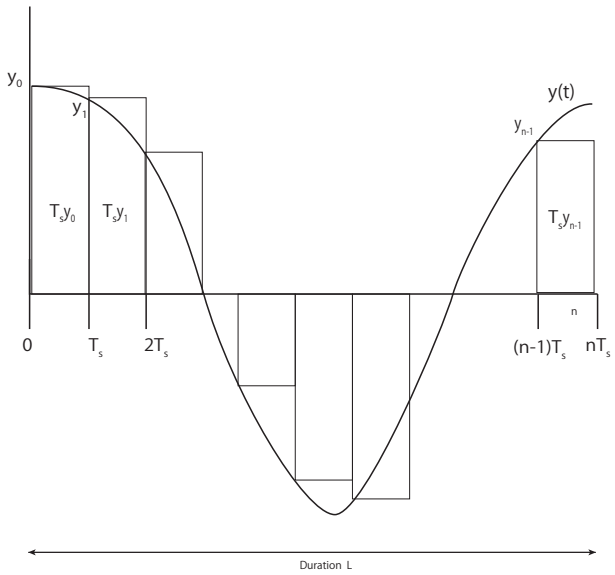

Figure 3.3: Converting an integration with a summation is equivalent to adding all the surfaces of $n-1$ rectangular boxes

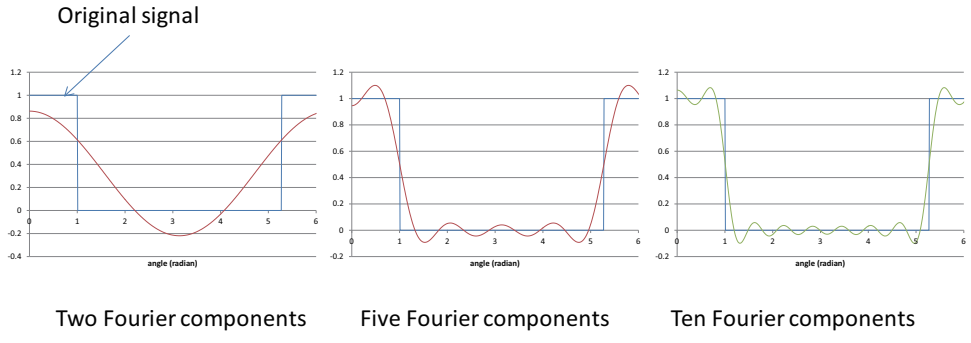

Figure 3.4: Linear combination of fundamental cosines approximate the original signal

By using the trigonometric addition formula, we can decompose the arbitrary phase signal into cosine and sine components.

$$\sin(2\pi ft + \phi) = \sin(\phi)\cos(2\pi ft) + \cos\phi \sin(2\pi ft) \tag{3.15}$$

Cosine components are referred to as In-phase and sine components are referred to as Quadrature particularly in radio communications field.

Example 3.1 *(basicFT)*

The following produces a 440Hz tone and decompose up to 8000 Hz cosines and sines. The resolution of the frequency is determined by the length of the sample. In the following case 1Hz. If L is 2 second, the resolution res = 0.5Hz. Note that in this case we don't have a_0 component.

```
%% basicFT
%  a tone is decomposed into res - 8000 Hz
%
clf; % clear figure
Fs = 8000; %sampling rate
L = 1; % length in second
t = (0:1/Fs:L-1/Fs);
res = 1/L; % frequency resolution
numF = Fs/res; % frequency components number
f440 = cos(2*pi*440*t);
fc = zeros(numF, L*Fs);
fs = zeros(numF, L*Fs);
a = zeros(numF,1);
b = zeros(numF,1);
f = (0:1/L:Fs-1/L);
for i=1:numF
    fc(i,:) = cos(2*pi*f(i)*t);
    fs(i,:) = sin(2*pi*f(i)*t);
end
for i=1:numF
    if i==1
        k=1;
    else
        k=2;
    end
    a(i) = k*fc(i,:)*f440'/L/numF;
    b(i) = k*fs(i,:)*f440'/L/numF;
end
hold off; % overwrite suppressed
plot(f, a);
hold on;  % overwrite enabled
plot(f, b);
xlabel('frequency(Hz)');
ylabel('intensity');
```

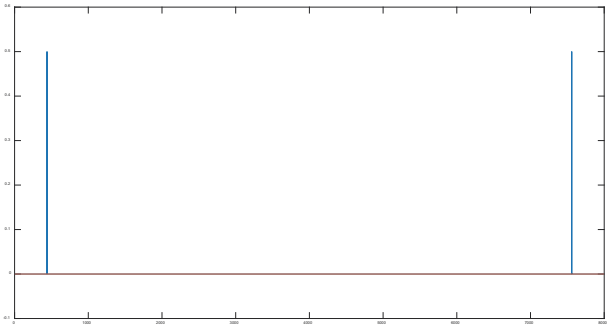

Figure 3.5: 440Hz cosine frequency domain signal at 8000 Hz sampling

This is the basic Discrete Fourier Transformation (DFT).

Exercise 3.1 *Calculate the frequency domain signal of* $\sin(2\pi 440t)$ *and compare that of* $\cos(2\pi 440t)$ *above.*

Fourier Transformation is to obtain the two sets components a, b of a given signal. In MATLAB, the function is `fft` and can be simply executed by

$$fy = fft(y) \tag{3.16}$$

You may wonder why you obtain a complex number as the result of `fft` of a real time series signal y later in the exercises. This is because MATLAB automatically decomposes the real signal into cosines and sines and represents the coefficients in a form of a complex number, $a_i + jb_i$, where j represents the imaginary unit $j^2 = -1$.

The use of FFT function not only eliminates the needs to analytically produce a_i and b_i in Eqs.3.4-3.6 but also is significantly fast. a_i and b_i can be described by

$$a_i = \frac{2}{n}\text{real}(fy_i) \text{ if } i \neq 0 \tag{3.17}$$

$$b_i = \frac{2}{n}\text{imag}(fy_i) \tag{3.18}$$

$$a_0 = \frac{1}{n}\text{real}(fy_0) \tag{3.19}$$

Let us start with a simple exercise to obtain `fft` of tone A.

Example 3.2 *fftToneA.m*

```
%%  fft of tone A.m
%
%   genrate a 440Hz tone
Fs = 8000; % sampling rate
A = 440; %tone frequency
t = (0:1/Fs:1-1/Fs);
  % we reduce the amplitude by 0.5 to avoid distotion.
y = 0.5*sin(2*pi*(A)*t+pi/4);
sound(y, Fs);
fy = fft(y);
plot(abs(fy)/8000);
xlabel('frequency (Hz)')
ylabel('gain')
```

The obtained image should be like Fig.3.6. If you take a look at the 441th element of the fft result, which

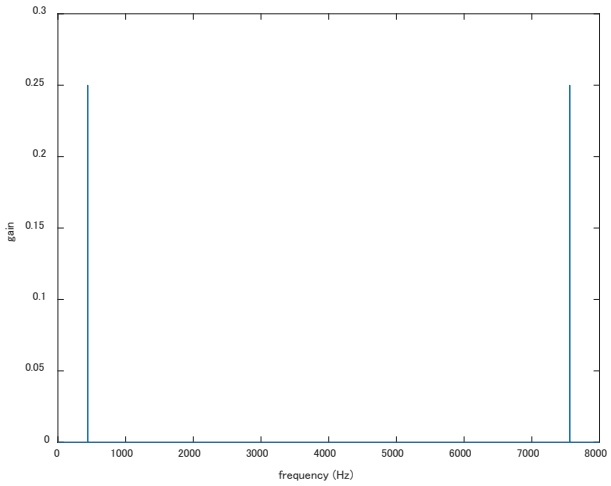

Figure 3.6: FFT of 440Hz monotone

corresponding to 440Hz, of `angle(fy)`, you should see

```
1.41e+03 + 1.4e+03i
```

which represents $\frac{\pi}{4}$ phase shift. If you delete $\frac{\pi}{4}$ in the `m file`, the coefficient should be

```
-5.2e-11 + 2.0e+03i
```

denoting the signal exclusively have a sine component. The marginal cosine component stems from the limited numerical computation resolution. The random angle in the other frequency is produced because of the small remnant of `fft` computations.

If you see the absolute value of fft, we see a 7560Hz signal although we only produce 440Hz tone. The 7560Hz signal is usually referred to as alias and is artificially produced because of the discretization of signal (sampling).

Exercise 3.2 *Try several sampling frequency and several tones and predict where the alias is produced.*

Since the mathematical proof of cause of alias is a little complex and even not necessary for application developers, a simple physical interpretation is given in the following.

Suppose we have a sine wave whose period is 2π second and apply a sampling with interval of four times a period, in this case $\frac{\pi}{2}$ second. The linear interpolation of the sampled signal is shown in Fig.3.7.

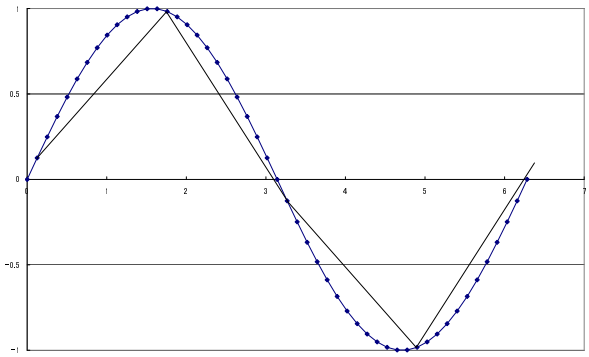

Figure 3.7: Four times sampling a signal period

The signal features can be approximated by the articulated line. Then, we halve the signal period to π keeping the sampling rate consistent to obtain Fig.3.8.

We still can observe the signal. We again halve the signal period to $\frac{\pi}{2}$ to obtain Fig.3.9,with which we no longer observe the existence of the signal. We futher increase the signal speed 5 times faster than the original signal, we observe is Fig.3.10, which produces an artificial signal whose shape is exactly the same to the original sampled data.

This is the famous Nyquist sampling theory, which states that we only capture the physical frequency component less than the half of the sampling frequency. The half of the sampling frequency is referred to as Nyquist frequency. Even more, we inevitably produce aliases above the Nyquist frequency by applying Fourier Transformation of a physical signal. The aliases are the image of physical signal in the frequency domain.

If you want to grasp the implication of Nyquist theorem and the aliases, image that a physical signal before sampling has a negative frequency component which is the mirror image of the positive frequency component as shown in Fig.3.11. We can actually suppress the imaginary part of frequency signal with an advanced sampling method, which discriminate the cosine component and sine component at sampling. But this is an advanced material.

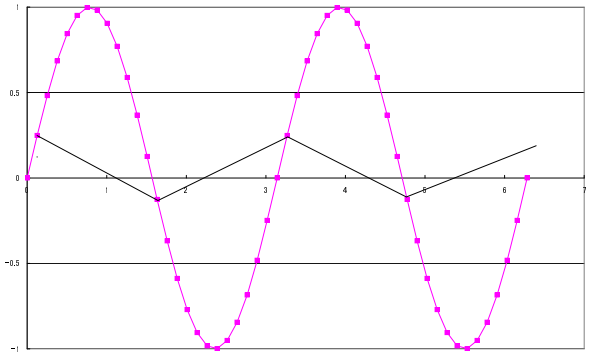

Figure 3.8: Two times sampling a signal period

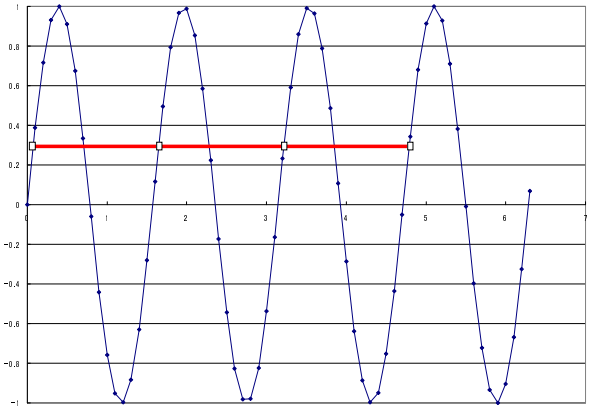

Figure 3.9: One time sampling a signal period

If we sample a physical signal, its images (aliases) are inevitably produced at every multiple (including negative) of sampling frequency. Figure 3.12 illustrates the repetition. Since a real signal inevitable has positive and negative spectrums, the sampling frequency should be two times larger than the maximum frequency of the signal. Otherwise, an alias overlaps the positive spectrum (Fig.3.13).

Exercise 3.3 *(fftTone7560) Let us examine from the opposite perspective. In the earlier example, we produce 440 Hz physical signal and observe 7650 Hz alias. Produce a 7560Hz tone and obtain frequency domain signal by applying fft and report what you observe and propose a method to observe 7560 Hz signal.*

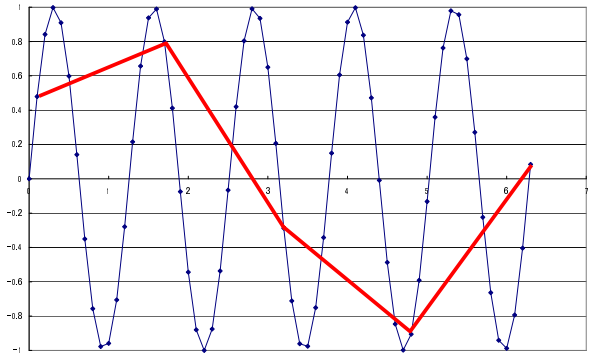

Figure 3.10: Artificially slow signal is observed when the sampling rate is not fast enough

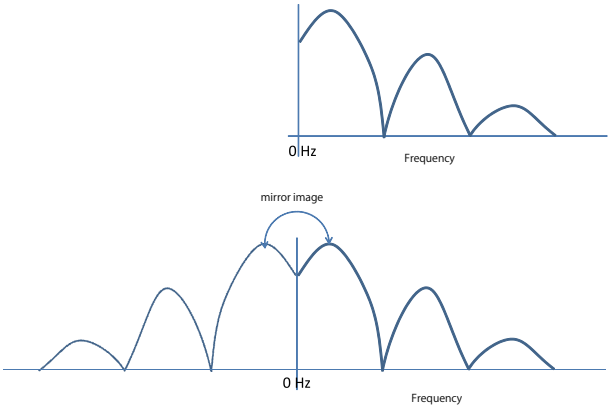

Figure 3.11: Physical signal in frequency domain

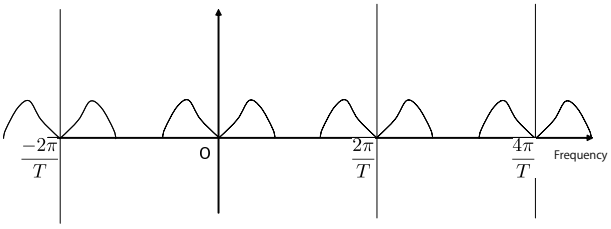

Figure 3.12: Alias of frequency domain signal appear in multiple of sampling frequency

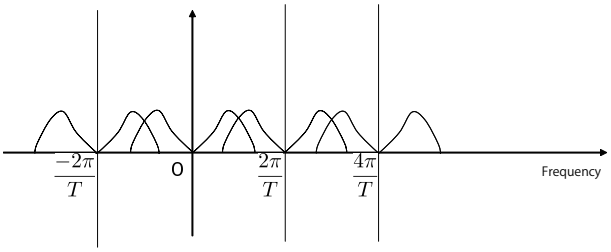

Figure 3.13: When sampling frequency is less than twice the maximum frequency, an alias interferes the real positive spectrum

As you may notice, the lowest frequency of fft is determined by the length of the signal. If we take shorter signal, we loose the resolution. The following `m file` provides the frequency domain signal of 440 Hz tone of 0.01 second.

Example 3.3 *fftShortToneA.m*

```
%%  fft of tone A.m
%%
%   genrate a 440Hz tone
Fs = 8000; % sampling rate
A = 440; %tone frequency
period = 0.01; %signal length in second
t = (0:1/Fs:period-1/Fs);
f = (1/period:1/period:Fs);
%sep = power(2, 1/12);
%Cs  = A*power(sep, 4)
E = Cs*power(sep, 3)
  % we reduce the amplitude by 0.5 to avoid distotion.
y = 0.5*sin(2*pi*(A)*t) ;
%y = 0.2*sin(2*pi*(A)*t) + 0.2*sin(2*pi*Cs*t) + 0.2*sin(2*pi*E*t);
sound(y, Fs);
fy = fft(y);
plot(f, abs(fy)/length(t));
xlabel('frequency (Hz)')
ylabel('gain')
```

The frequency domain signal looks like Fig.3.14 It is shown that the given physical tone includes frequency

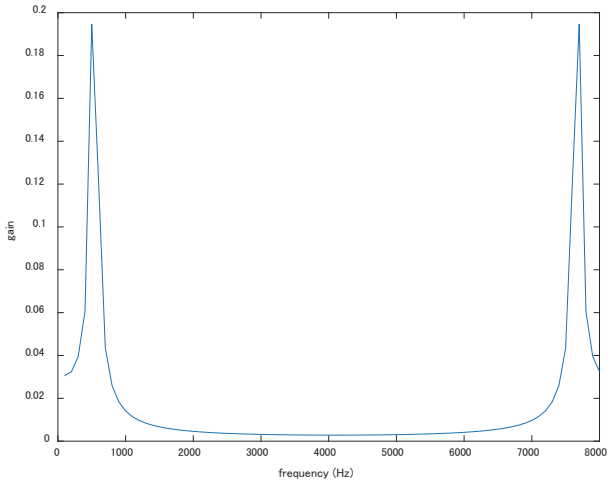

Figure 3.14: Frequency domain signal of 440 Hz tone of 0.01 second period

components other than 440Hz. This is the resolution problem we can only improve by increasing the signal period. Increasing sampling rate does not help us. In order to increase the bandwidth of the signal we need to increase the sampling rate to have larger Nyquist frequency. But increasing the sampling rate without increasing the number of sample does not contribute to increase the resolution because, for example, doubling the sampling rate merely halves the signal period.The resolution depends only on the period rather than the signal itself. We can increase the resolution by adding zeros.

Exercise 3.4 *(threeTonesFft) If we produce a harmony A with the following script, we obtain frequency domain signal where the level of the three tones are different. If we generate three A tones such that A_{220}= 220 Hz, A_{440} = 440 Hz and A_{880} = 880 Hz, these three tones produce exactly the same level frequency components. Explain why the three levels are different only in the case of harmony A.*

```
%%  fft of harmony A.m
%
%   generate a 440Hz tone
Fs = 8000; % sampling rate
A = 440; %tone frequency
period = 0.02; %signal length in second
t = (0:1/Fs:period-1/Fs);
f = (1/period:1/period:Fs);
sep = power(2, 1/12);
Cs  = A*power(sep, 4)
E = Cs*power(sep, 3)
  % we reduce the amplitude by 0.2 to avoid distortion.
y = 0.2*sin(2*pi*(A)*t) + 0.2*sin(2*pi*Cs*t) + 0.2*sin(2*pi*E*t);
sound(y, Fs);
fy = fft(y);
plot(f, abs(fy)/length(t));
xlabel('frequency (Hz)')
ylabel('gain')
```

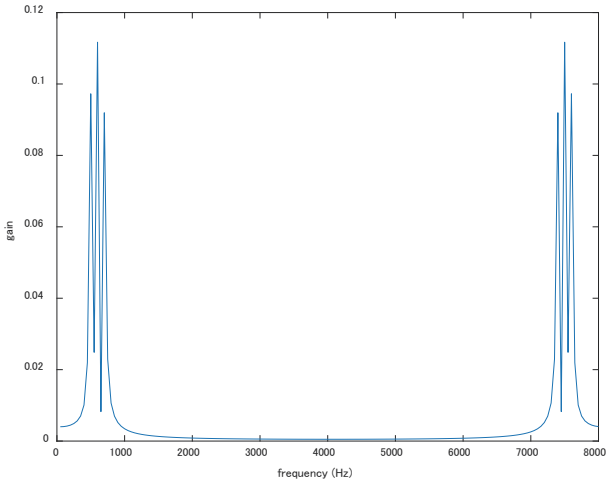

Figure 3.15: Frequency domain signal of harmony A for 0.02 second

3.2 Simple Music Player

Let us compose a simple mp3 player with `MATLAB`. The following program captures a group of samples by `frameLength` unit from an mp3 file usually in a stereo (2ch format), and feeds it to an audio device `deviceWriter` to play back.

Longer frame size entails longer play back delay, but the permissible time to process a frame before capturing the next frame increases, instead. Complex signal process on relatively slow computer demands a long frame size with the penalty of a long delay, which is usually not a big problem as a music player.

```
%
%  simple music play
%
clear;
frameLength = 1024; % length of a frame
mfile = uigetfile('*.mp3'); %mp3 file
fileReader = dsp.AudioFileReader(...
    mfile,...
    'SamplesPerFrame',frameLength);
Fs = fileReader.SampleRate;
deviceWriter = audioDeviceWriter(...
    'SampleRate',Fs);% audio system define
```

```
while ~isDone(fileReader)
    signal = fileReader(); % read frameLength samples from the mp3 file
    deviceWriter(signal); % write to audio device
end
release(fileReader);
release(deviceWriter);
```

The procedure can be depicted as Fig. 3.16.

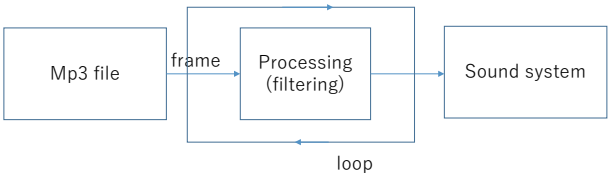

Figure 3.16: Capture audio samples by a frame from a specified mp3 file

Exercise 3.5 *(musicdsp_baser)*

Add dynamic spectrum window to `musicdsp_base` *while a music is playing as shown in Fig. 3.17. The spectrum display style can be other than the bar style shown. The separation unit of the frequency axis can be changed. Avoid any noise increase and slow down the play back speed because of the spectrum display option.*

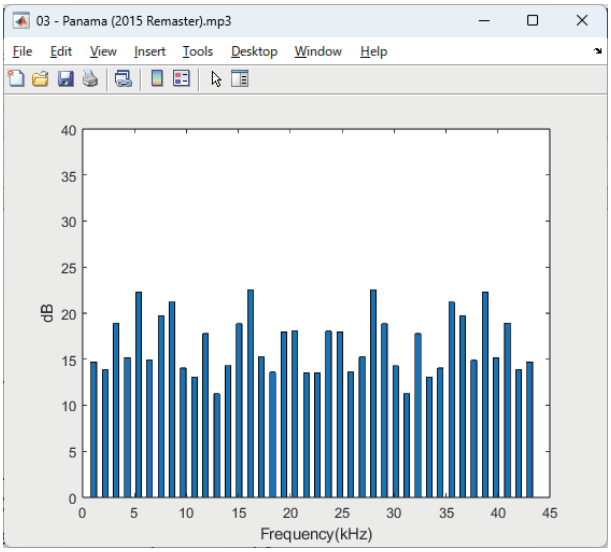

Figure 3.17: Dynamic spectrum display while a music file is playing.

4 Digital Filters

4.1 Linear Time Independent System

We can change the nature of input signal $\{x_0, x_1, x_2, \cdots\}$ by applying calculations to produce an output signal $\{y_0, y_1, y_2, \cdots\}$ where x_n and y_n is the finite digit (quantized) number of n-th sample time. The calculation we apply to a series x to produce another series y can be any operation but we usually apply linear operations and the operation does not depend on time. This is referred to as Linear Time Independent (LTI) system (Fig.4.1). In this class, we only focus on LTI system.

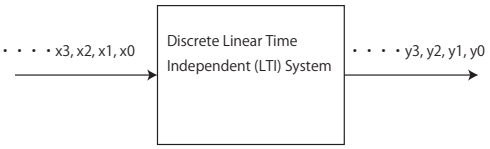

Figure 4.1: LTI system schematic

The general expression of LTI system can be expressed by introducing the two sets of coefficients a and b.

$$\begin{aligned} y(n) &= -a_1 y(n-1) - a_2 y(n-2) - a_3 y(n-3) + \cdots - a_N y(n-N) \\ &+ b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + \cdots + b_M x(n-M) \\ &= -\sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k) \end{aligned} \tag{4.1}$$

where N, M provide the limit of the contributions of past and present y, x, respectively. Since we cannot use future samples for real time processing, there is no coefficient for $a_{-1}, a_{-2}, \cdots$ and $b_{-1}, b_{-2}, \cdots$. The negative sign of a_n is for later convenience. It can be defined positive, if you like. Equation 4.1 only comprises the current and past samples. This is referred to as causal. If we can use future data as in the case of stored data processing, it is non-causal or anti-causal.

We already learn any time series can be converted to a frequency domain signal which comprises many cosines and sines such that

$$x(t) = c_0 \cos 2\pi f_0 t + s_0 \sin 2\pi f_0 t + c_1 \cos 2\pi f_1 t + s_1 \sin 2\pi f_1 t + \cdots + c_{N-1} \cos 2\pi f_{N-1} t + s_{N-1} \sin 2\pi f_{N-1} t \tag{4.2}$$

where $f_i = \frac{i}{T_s(N-1)}$ and T_s is the sampling period, which relates to the sampling frequency F_s with $T_s = \frac{1}{F_s}$ and N denotes the number of samples in the interested period. We use c_i and s_i to clearly discriminate these frequency components from filter coefficients a_i and b_i here. Denoting an angular velocity $\omega_i = 2\pi f_i$ and introducing the imaginary unit j to separate cosines and sines, the above can be rewritten as follows.

$$\begin{aligned} x(n) &= c_0 \cos \omega_0 T_s n + j s_0 \sin \omega_0 T_s n + c_1 \cos \omega_1 T_s n + j s_1 \sin \omega_1 T_s n + \cdots + c_{N-1} \cos \omega_{N-1} T_s n + j s_{N-1} \sin \omega_{N-1} T_s n \\ &= \sum_{i=0}^{N-1} c_i \cos \omega_i T_s n + j s_i \sin \omega_i T_s n. \end{aligned} \tag{4.3}$$

The discrete angular velocity spans $0 \leq \omega_i \leq \omega_s (= 2\pi F_s)$. Since we only handle LTI, a filter affects all the frequency components independently, $x(n)$ can be subdivided to each frequency component such that

$$x_i(n) = c_i \cos \omega_i T_s n + j s_i \sin \omega_i T_s n, \tag{4.4}$$

which can be simplified by eliminating the subscript and merging $\omega_i T_s$ as the normalized angular frequency $\omega = \omega_i T_s$

$$\begin{aligned} x(n) &= c \cos \omega n + j s \sin \omega n \\ &= \sqrt{c^2 + s^2} \left(\frac{c}{\sqrt{c^2 + s^2}} \cos \omega n - j \frac{s}{\sqrt{c^2 + s^2}} \sin \omega n \right) \\ &= A cos(\omega n + \theta). \end{aligned} \tag{4.5}$$

$A = \sqrt{c^2 + s^2}$ and $\cos\theta = \frac{c}{\sqrt{c^2+s^2}}$ are the amplitude and the phase offset determined by c and s. Note that ω spans from 0 to 2π because $\omega = 2\pi f_i T_s$ and the maximum value of f_i is equal to the sampling frequency $F_s = \frac{1}{T_s}$. One sample backward can be rewritten as

$$x(n-1) = c\cos\omega(n-1) + js\sin\omega(n-1), \tag{4.6}$$

and one sample forward is

$$x(n+1) = c\cos\omega(n+1) + js\sin\omega(n+1). \tag{4.7}$$

Example 4.1 *Let us give an example. For simplicity, we have the following wave with no phase offset*

$$x(t) = A\cos(2\pi t) \tag{4.8}$$

and the sampling period T_s is equal to $\frac{5}{12}$. Since $t = nT_s = \frac{5n}{12}$ where n is a positive integer, Eq.4.8 can be converted to the following sample based series.

$$x(n) = A\cos(\frac{5\pi}{6}n). \tag{4.9}$$

Since a trigonometric function is periodic, we can not distinguish the following two signals

$$x_0(n) = Acos(\omega n + \theta) \tag{4.10}$$
$$x_k(n) = Acos((\omega + 2\pi k)n + \theta). \tag{4.11}$$

By taking $k = -1$, the given cosine wave sample series matches a different sample series $A\cos(\frac{-7\pi}{6}n)$ as follows

$$x(n) = A\cos(\frac{5\pi}{6}n) = A\cos((\frac{5\pi}{6} - 2\pi)n) = A\cos(\frac{-7\pi}{6}n) = A\cos(2\pi\frac{7}{5}\frac{5}{12}n). \tag{4.12}$$

The last equation is derived from the even function nature of cosine. This means that a relatively slow wave $A\cos(2\pi\frac{5}{12}n)$ cannot be discriminated from a fast wave $A\cos(2\pi\frac{7}{5}\frac{5}{12}n)$ as sample series. In continuous signal, we have infinite resolution of T_s so that the two waves can be distinguished clearly.

Let us examine from the opposite perspective by taking

$$x(n) = A\cos(2\pi\frac{7}{5}\frac{5}{12}n). \tag{4.13}$$

This cosine wave time series matches a different time series $A\cos(\frac{5\pi}{6}n)$ as follows

$$x(n) = A\cos(2\pi\frac{7}{5}\frac{5}{12}n) = A\cos((2\pi\frac{7}{5}\frac{5}{12} - 2\pi)n) = A\cos(-2\pi\frac{5}{12}n) = A\cos(2\pi\frac{5}{12}n) \tag{4.14}$$

This indicates that we inevitably produce an alias by discretization either the sampling frequency is below or above Nyquist frequency. This is why we always have alias in FFT.

Figure 4.2 shows that a sample sequence produced with $T_s = \frac{5}{12}$ can be interpreted as $\cos(2\pi t)$ and $\cos(2\pi\frac{7}{5}t)$

Now that we can express a time series with the sampling count n rather than time t for any angular velocity ω as shown in Eq.4.5. From Euler equation, cosines and sines can be converted to the following form.

$$e^{j\omega n} = \cos\omega n + j\sin\omega n \tag{4.15}$$

The proof of Euler equation requires the knowledge on Taylor expansion.

.....advanced .....

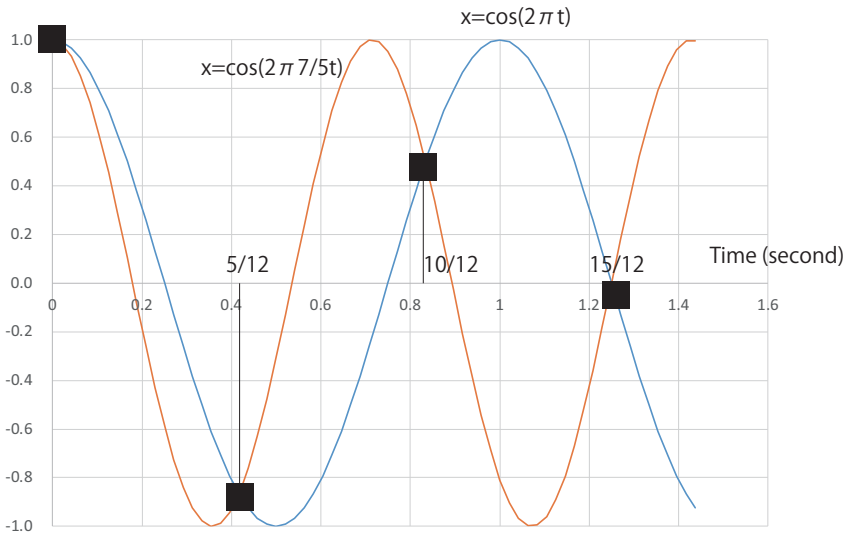

Figure 4.2: Discretization always can be interpreted differently

The derivation of this equation needs the knowledge of Taylor expansion, which is an approximation of a function with polynomials such that

$$f(x) = f(0) + f'(0)x + \frac{1}{2}f''(0)x^2 + \frac{1}{6}f'''(0)x^3 + \cdots . \tag{4.16}$$

where a prime $'$ denotes a derivative.

For example, If $f(x)$ is a quadratic function, $f(x) = ax^2 + bx + c$, each components of Taylor series are as followings which is the exact match of the original function.

$$\begin{aligned} &f(0) = c \\ &f'(0)x = 2ax + b\,|_{x=0}\, x = bx \\ &\frac{1}{2}f''(0)x^2 = ax^2 \end{aligned} \tag{4.17}$$

Likewise, if we take an infinite series of polynomial we can approximate any function.

Taylor series of $e^{j\phi}$ is as following, which proves Eq.4.15

$$e^{j\phi} = 1 + j\phi - \frac{1}{2}\phi^2 - j\frac{1}{6}\phi^3 + \frac{1}{24}\phi^4 + \cdots \tag{4.18}$$

Taylor series of $\cos\phi, \sin\phi$ are as following

$$cos\phi = 1 - \frac{1}{2}\phi^2 + \frac{1}{24}\phi^4 + \cdots \tag{4.19}$$

$$sin\phi = \phi - \frac{1}{6}\phi^3 + \cdots \tag{4.20}$$

This expression is very convenient because any increment of sampling period can be denoted as the multiplication of $e^{j\omega}$. One sample forward can be represented by multiplying $e^{j\omega}$ and one sample move backward is $e^{-j\omega}$

accordingly. Using this convention, we can express a discrete time series such that

$$\cdots, x_3, x_2, x_1, x_0, x_{-1}, x_{-2}, x_{-3}, \cdots = \sum_{n=-\infty}^{\infty} x(n) e^{-j\omega n} \tag{4.21}$$

when we define $z = e^{j\omega}$, the above equation can be rewritten as follows.

$$\cdots, x_3, x_2, x_1, x_0, x_{-1}, x_{-2}, x_{-3}, \cdots = \sum_{n=-\infty}^{\infty} x(n) z^{-n}. \tag{4.22}$$

This is Z transformation. Using Z transformation, time series $y(n), x(n)$ shown in Eq.4.1 can be concisely rewritten as follows.

$$Y(z) = \cdots, y_3, y_2, y_1, y_0, y_{-1}, y_{-2}, y_{-3}, \cdots = \sum_{k=-\infty}^{\infty} y(k) z^{-k} \tag{4.23}$$

$$X(z) = \cdots, x_3, x_2, x_1, x_0, x_{-1}, x_{-2}, x_{-3}, \cdots = \sum_{k=-\infty}^{\infty} x(k) z^{-k} \tag{4.24}$$

Note that we use $y(n)$ exclusively represents the value at nth sample, while $Y(z)$ represents the whole time series.

An LTI can be represented as a function $H(z)$ such that

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}} \tag{4.25}$$

As we learned earlier, one set of discrete signal input and output, $X(z)$ and $Y(z)$, can be related with another discrete signal $H(z)$. This independence is provided by the convolution calculations $-\sum_{k=1}^{N} a_k y(n-k)$ and $\sum_{k=0}^{M} b_k x(n-k)$ in Eq.4.1.

$$Y(z) = \sum_{n=-\infty}^{\infty} y(n) z^{-n} = -\sum_{n=-\infty}^{\infty} \sum_{k=1}^{N} a_k y(n-k) z^{-n} + \sum_{n=-\infty}^{\infty} \sum_{k=0}^{M} b_k x(n-k) z^{-n} \tag{4.26}$$

We replace the summation order to yield

$$\begin{aligned} &= -\sum_{k=1}^{N} a_k \sum_{n=-\infty}^{\infty} y(n-k) z^{-n} + \sum_{k=0}^{M} b_k \sum_{n=-\infty}^{\infty} x(n-k) z^{-n} \\ &= -\sum_{k=1}^{N} a_k Y(z) z^{-k} + \sum_{k=0}^{M} b_k X(z) z^{-k} \end{aligned} \tag{4.27}$$

We can extend the summation for k from $-\infty$ to ∞ by introducing zero coefficients such that

$$Y(z) = -A(z)Y(z) + B(z)X(z) \tag{4.28}$$

which leads to

$$Y(z) = H(z)X(z) \tag{4.29}$$

where $H(z) = \frac{B(z)}{1+A(z)}$. This outcome is very convenient because the contribution of filtering in terms of $H(z)$ onto $Y(z)$ is separated from the input discrete signal $X(z)$. This separation is possible because of the convolution multiplication in Eq.4.1. The frequency response of a filter is, therefore, only governed by $H(z) = H(e^{j\omega})$.

By defining the coefficients a_k, b_k we can change the characteristics of output $Y(z)$ subjected to the same $X(z)$. This process is, in general, referred to as filter. There are two types of filters, Fixed Impulse Response (FIR)

filters and Infinite Impulse Response (IIR) filters. FIR filters are subsets of general filters which only posses b_k coefficients such that

$$H(z) = \sum_{k=0}^{M} b_k z^{-k}. \tag{4.30}$$

An IIR filter has the general from of Eq.4.25 and demands less parameters, a_k and b_k, and yet exhibits better performance compared with FIR filter. Problems of IIR filter are the possible instability, the phase distortion for wide band signals and relatively complex filter design.

.....advanced .....

Let us see an example of instability. In general, Z transformation accompanies a region of convergence (ROC) which defines the stability region of Z transformation. In the earlier example $z = e^{j\omega}$ so that z transformation needs to be stable at the circumference $|z| = 1$ because $|e^{j\omega}| = \sqrt{\cos^2\omega + \sin^2\omega} = 1$. The stability depends on a_k and b_k. FIR filters are always stable at $|z| = 1$ but not the case in IIR filter. For example, consider $y(n) = x(n) + 2y(n-1)$ which yields

$$H(z) = \frac{1}{1 - 2z^{-1}} = \frac{1}{1 - \frac{2}{z}} \tag{4.31}$$

ROC is $|z| > 2$, thus, the system diverges as we progress time and we cannot derive frequency response in this case. The ROC can be derived by considering $H(z)$ as the infinite sum of a geometric series (等比級数) where the coefficient is $\frac{2}{z}$ and the initial value is equal to 1.

Let us study a little more on the requirements on a filter. Those two are the fundamental requirements to $H(z) = H(e^{j\omega})$.

1. The absolute value of $H(e^{j\omega})$ in pass band is around 1,
2. The phase of $H(e^{j\omega})$ should be linear in terms of ω.

The first requirement is easy to understand. Any filter is not supposed to scale the magnitude of signal before and after a filter. The second requirement is to convey a signal, which has many frequency components with a same time delay before and after a filter. If a sinusoidal signal is incurred a phase delay ϕ by a filter such that $\sin(\omega_1 t + \phi_1)$, the absolute time delay Δt caused by the phase delay can be expressed as $\Delta t = \frac{\phi_1}{\omega_1}$. In order to incur the same absolute delay to another frequency component ω_2, the phase delay shall be

$$\Delta t = \frac{\phi_2}{\omega_2} = \frac{\phi_1 \frac{\omega_2}{\omega_1}}{\omega_2}. \tag{4.32}$$

Thus,

$$\phi_2 = \frac{\omega_2}{\omega_1}\phi_1 \tag{4.33}$$

In general, the following shall hold.

$$\frac{\partial \phi}{\partial \omega} = \text{const.} \tag{4.34}$$

The left-hand side of Eq. 4.34 denotes the group delay. If the group delay is constant, it is referred to as a linear phase (delay).

The characteristics of a filter can be analyzed by inspecting the poles (the z values that yield zero in the denominator of $H(z)$) and zeros (the z values that yield zeros in the numerator). Suppose we have $H_1(e^{j\omega}) = \frac{1-a}{1-ae^{-j\omega}}$. If $\omega = 0$, $H_1(e^{j\omega}) = 1$. If $\omega = \pi$, $H_1(e^{j\omega}) = \frac{1-a}{1+a}$. Thus this is a low pass filter (LPF). The addition of $1 + z^{-1}$ in the numerator of H_1 yields

$$H_2(e^{j\omega}) = \frac{1-a}{2}\frac{1+e^{-j\omega}}{1-ae^{-j\omega}} \tag{4.35}$$

In this case, at $\omega = 0$, $H_2(e^{j\omega}) = 1$ and at $\omega = \pi$, $H_2(e^{j\omega}) = 0$. This yields steeper LPF compared with H_1. But those filter does not conform to the linear phase requirement.

In practices with `MATLAB`, the phase distortion problem can be greatly mitigated by the filter design facilities such as `filterDesigner` or `firceqrip`, `butter`, `chevy1` in MATLAB. `filterDesigner` can be initiated by typing `filterDesigner` in the command window. To calculate a, b, we only need to specify the

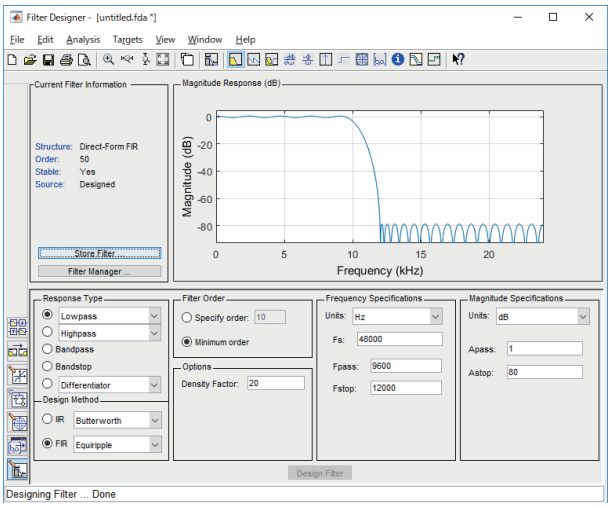

Figure 4.3: Filter Designer Window

filter parameters and to choose the type of filters. We can either design a filter with the `fileDesigner` or use `firceqrip`, `butter` function in a custom program. Practically we can design any kind of filter with this tool.

Let us demonstrate how to make filter and filter an audio stream. Typical filters are Low Pass Filter (LPF), High Pass Filter (HPF), Band Pass Filter (BPF) and Band Stop Filter(BSF). A LPF suppresses high frequency components. A HPF suppresses low frequency components. A BPF extracts a specified bandwidth. A BSF suppresses only specified frequency region as shown in Fig.4.4

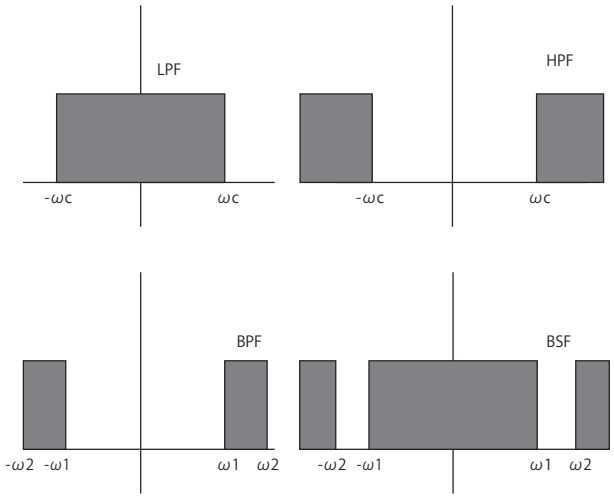

Figure 4.4: Popular Filters (shaded area is retrained and remaining is suppressed)

We have to bear in mind, in practice, there is no ideal filter like the ones in Fig.4.4 with which we can abruptly retain or suppress specified frequency region because we can only use causal filters. When we design filters we need to compromise at some characteristics such as fluctuation of output power. Typical characterization of filter are shown in Fig.4.5.

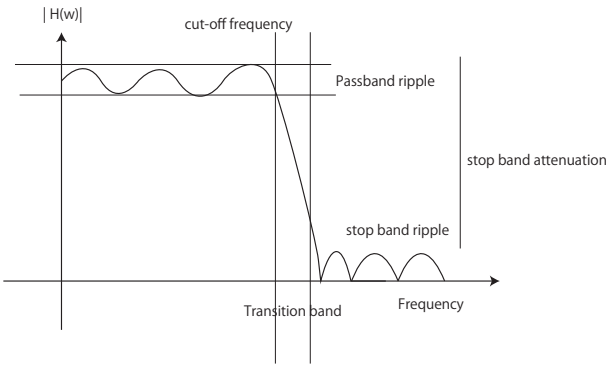

Figure 4.5: Practical Filter Performance

4.2 Digital filtering of audio streaming

Let us design and apply an LPF to an audio streaming. The following script designs a FIR filter specifying the frame length, pass band edge (cut off frequency) and stop band attenuation (level of stop band suppress). We later learn the theory of filter design.

Example 4.2 *(musicdsp)*

```
frameLength = 4096;
N   = 100;         % FIR filter order
Fp  = 1e3;        % 1 kHz passband-edge frequency
Rp  = 0.00057565; % Corresponds to 0.01 dB peak-to-peak ripple
Rst = 1e-4;       % Corresponds to 80 dB stopband attenuation
Fs = 44100;
% FIR filter design
eqnum = firceqrip(N,Fp/(Fs/2),[Rp Rst],'passedge'); % eqnum = vec of coeffs
% passband edge is normalized with sampling frequency
% create a filter
lowpassFIR = dsp.FIRFilter('Numerator', eqnum);
% show the characteristics of the filter
fvtool(lowpassFIR, 'Fs', Fs, 'Color', 'White');
% specify an audio file
fileReader = dsp.AudioFileReader(...
    '****.mp3',...  % replace with a mp3 file
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
scope = dsp.SpectrumAnalyzer(SampleRate', fileReader.SampleRate);
while ~isDone(fileReader)
       % acquire frame lenth audio stream
    signal = fileReader();
       % apply LPF by convolutional product
    yy = lowpassFIR(signal);
       % write to speaker
    deviceWriter(yy);
       % show
    scope([signal,yy]);
end
release(fileReader);
release(deviceWriter);
release(scope);
```

The flow of the script can be illustrated as Fig.4.6 The convolution product of the time samples and the filter

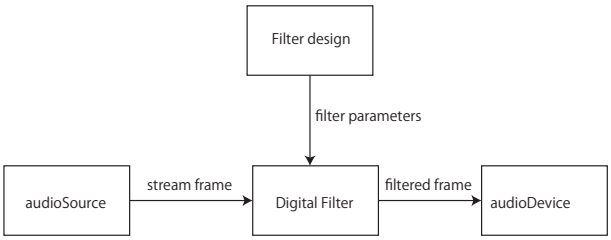

Figure 4.6: Processing flow in musicdsp.m

coefficient is automatically taken care of by `yy = lowpassFIR(signal);` *line. As the convolution demands past time samples, the* `lowpassFIR` *automatically stores the previous frame.*

The result of the script is shown in Fig,4.7. The two spectrum shows that of before and after the LPF.

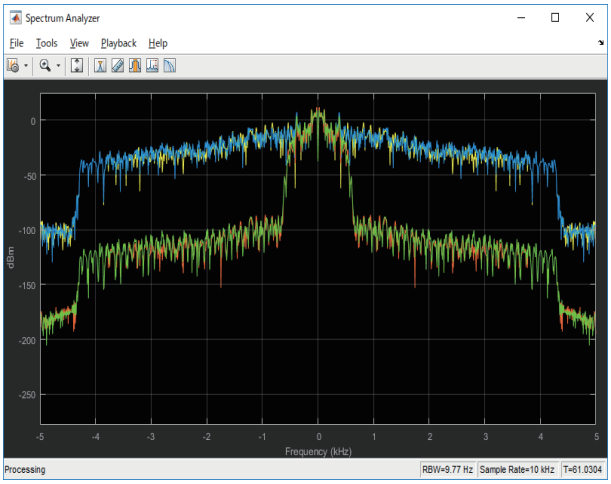

Figure 4.7: Before and after applying FIR filter

`firceqrip(N, Fo, dev)` *designs a FIR filter with* `N` *taps,* `Fo` *cutoff frequency spanning from 0 to 1.* `Fo=0.5` *means that cut off frequency is the half of Nyquist frequency.* `dev` *is an* `1x2` *array, which contains the passband ripple and the stopband attenuation. At the cut-off frequency, the amplitude becomes the half of passband. In the above filter design we allow 100 taps (samples) to be in the filter. If we reduce the number of samples for example to 20 taps. We can only produce a reduced performance filter. The comparison of the filter characteristics obtained from* `fvtool(lowpassFIR, 'Fs', Fs, 'Color', 'White');` *facility as shown in Fig.4.8.*

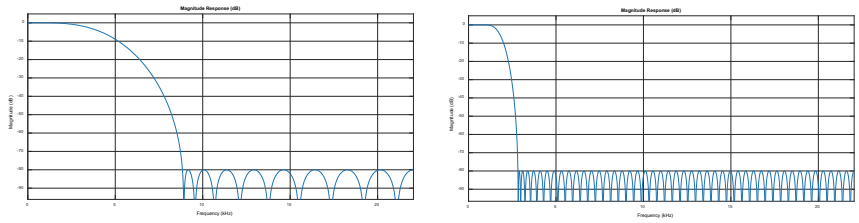

Figure 4.8: LPF characteristics of 100 and 20 taps FIR filters

Example 4.3 *(HPF filtering) The following is an example of high pass filter with 10 kHz stopband edge.*

```
frameLength = 1024;
mfile = uigetfile('*.mp3');
N   = 100;          % FIR filter order
Fp  = 10e3;         % 10 kHz stop band-edge frequency
Rp  = 0.00057565; % Corresponds to 0.01 dB peak-to-peak ripple
Rst = 1e-5;         % Corresponds to 80 dB stopband attenuation
eqnum = firceqrip(N,2*Fp/Fs,[Rp Rst],'high'); % eqnum = vec of coeffs
%fvtool(eqnum,'Fs',Fs,'Color','White') % Visualize filter
highpassFIR = dsp.FIRFilter('Numerator', eqnum);
%fvtool(lowpassFIR, 'Fs', Fs, 'Color', 'White');
fileReader = dsp.AudioFileReader(...
    mfile,...
    'SamplesPerFrame',frameLength, 'OutputDataType', 'int16');
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
scope = dsp.SpectrumAnalyzer('SampleRate', Fs);
%dRG = noiseGate(...
%    'SampleRate',fileReader.SampleRate,...
%    'Threshold',-25,...
%    'AttackTime',10e-3,...
%    'ReleaseTime',20e-3,...
%    'HoldTime',0);
%visualize(dRG);
%configureMIDI(dRG);
while ~isDone(fileReader)
    signal = fileReader();
    yy = highpassFIR(signal);
    %deviceWriter(yy);
    scope([signal, yy]);
end
release(fileReader);
release(deviceWriter);
release(scope);
```

Example 4.4 *(filterDesigner)* `filterDesigner` *utility significantly facilitates the filter design. This utility can be started by typing* `filterDesigner` *in the command prompt. After typing in the requirements on the filter are specified as shown in Fig.4.9, push* `Design Filter` *button to produce a filter (Fig.4.10).*

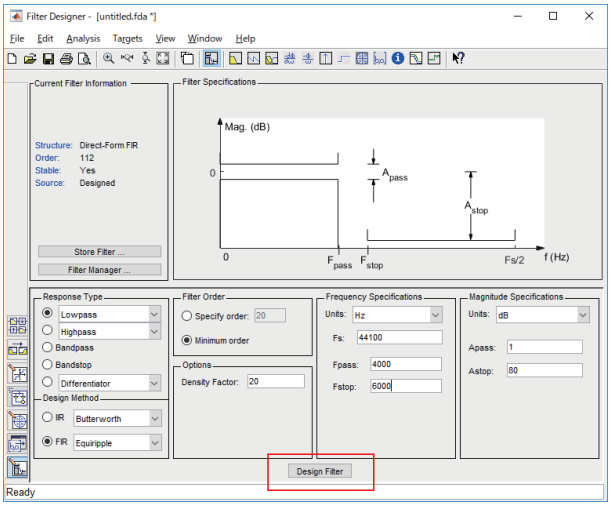

Figure 4.9: LPF design with filterDesigner

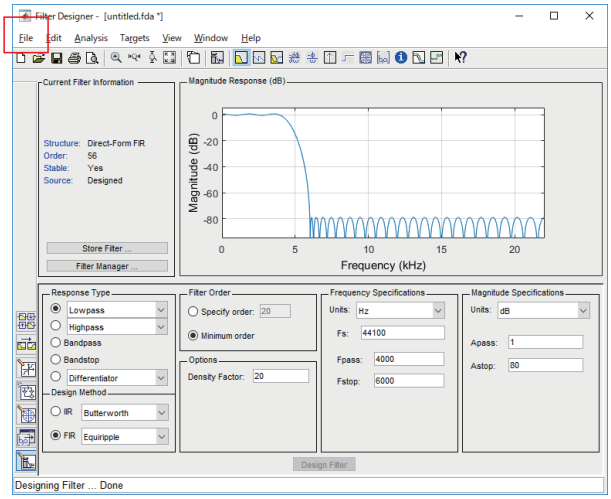

Figure 4.10: LPF produced by filterDesigner

In this case, we design FIR filter. The numerator of $H(z)$ *can be exported to workspace by* `File/Export`. *By default, the numerator is named as* `Num`. *We can create a FIRfilter by using the numerator such that*

```
lowpassFIR = dsp.FIRFilter('Numerator', Num);
```

Exercise 4.1 *(musicdsp_bpf) Create a* `m` *file to apply BPF (pass band edge = 10kHz, stop band edge = 4kHz) to a music source as shown in Fig.4.11.*

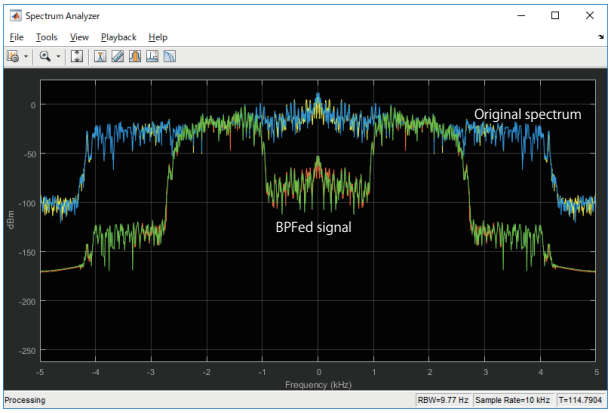

Figure 4.11: Band pass filtered audio signal

4.3 Design of FIR filters

There are two streams in designing FIR filter. One is window design method where we specify the ideal frequency response and apply inverse FFT to derive the filter numerators. The other method is optimal method such as Parks-McClellan filter where the filter numerators are derived by specifying the passband and stopband ripples (flatness). In the following, we use an window design method.

4.3.1 Inverse Fourier Transform

In the previous section, we learn how to obtain a frequency domain signal using Fourier Transform from a time domain signal. Since Fourier Transform is a linear operation, we can do the reverse operation, namely Inverse Fourier Transform, converting a frequency domain signal to time domain signal. This is to produce a time series data by specifying the frequency components.

Example 4.5 *(inverseA.m)*
The following script produce a time series of tone 440Hz for one second.

```
%% inverseA.m
%
%   genrate a 440Hz tone with inverse FFT.
%
Fs = 8000; % sampling rate
L =  8000; % sample length
f = zeros(L,1);
f(441) = L; % the first component is f=0
yt = ifft(f); % inverse FFT
sound(real(yt), Fs);
```

`ifft` *takes a real vector to represent the frequency components and produces a complex series of time signal. The real part represents the in-phase (cosine) component and the imaginary part represents the quadrature as is in Fourier Transform. In order to produce a sound we should take either the real part or the imaginary part. If we take the absolute value, it would produce no sound because the absolute value of the* `ifft` *result is constant because* $\cos^2 t + \sin^2 t = 1$.

Note that the lowest frequency after FFT is 0, rather than $\frac{1}{period}$*, 440 Hz frequency is located at* $f(441)$ *rather than* $f(440)$*. The alias of 440 Hz emerges at 7560 Hz when the sampling frequency is 8000 Hz as shown in the above example, The 7560 Hz component appears at* $f(7561)$*, accordingly.*

Exercise 4.2 *(inverseA2sec)*
Modify the script `inverseA` *to produce 440 Hz tone for 2 seconds.*

4.3.2 Produce filter coefficients with iFFT

Suppose we have a range of frequencies whose amplitude is finite and in the rest of the frequency components are zero as shown in Fig.4.12. Its inverse FFT is the time series coefficients of the filter with no time shift.

Example 4.6 *(ifftlpf_base)*
The following script produces coefficients of LPF with 1kHz pass band edge.

```
%% ifftlpf_base
%
Fs = 44100; % sampling rate
L =  44100; % sample length
N = 1000; % passband edge
f = zeros(L,1);
for i=1:N
    f(i) = L ;
end
yt = ifft(f); % inverse FFT
plot(real(yt));
sound(real(yt), Fs);
```

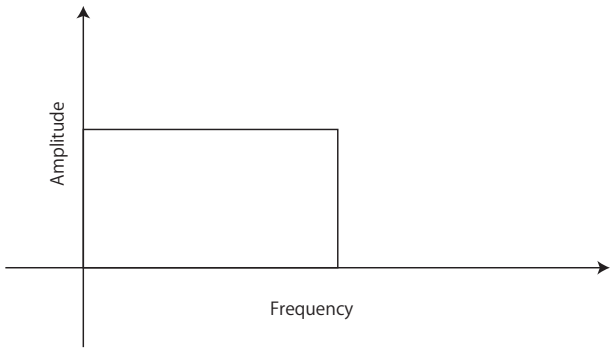

Figure 4.12: Allowing only a part of frequency components

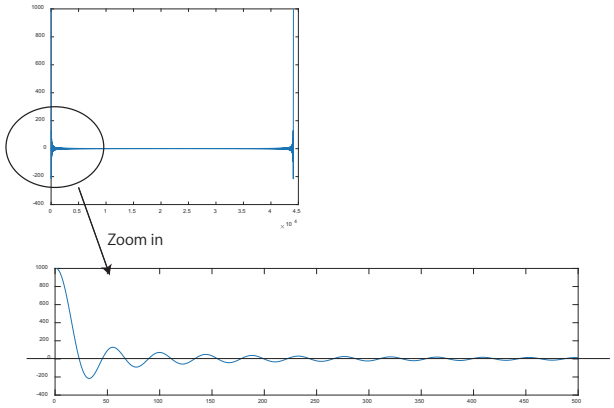

Figure 4.13: Ideal LPF coefficients

The produced LPF time series is as shown in Fig.4.13.

As is in the case of FFT, we have symmetric time series of the coefficients. These characteristics stem from the fundamental assumption of periodic signal in Fourier and inverse Fourier Transformations. The coefficients fluctuate and gradually decrease as the progress of time. It is interesting to see that the coefficients cross zero at every 22.05 (= 44100/1000/2) samples. This is because the filtered signal should be the linear combinations of multiple of the cut off frequency, 1000Hz.

Direct use the coefficients, after disposing the small values, as LPF coefficients apparently does not work, because there is an abrupt change of signal at the entry of filter resulting in distortion. To avoid the distortion, a mirror image of the filter is concatenated before the beginning of the coefficients as shown in Fig.4.14 allowing the processing delay, in this case 100 taps. Please avoid the duplication of the first coefficient a_o when you combine

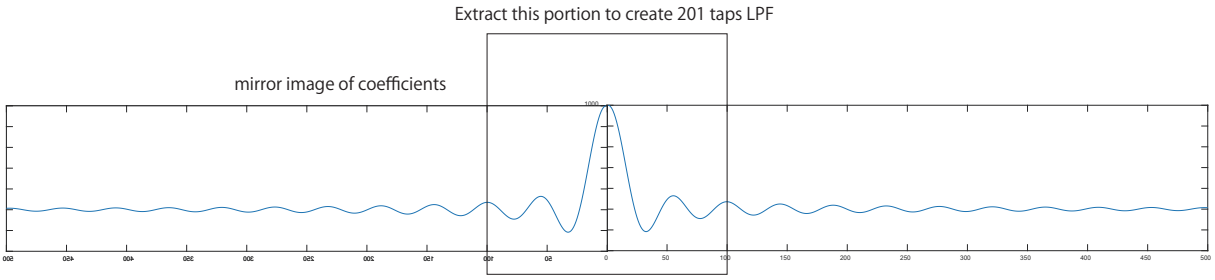

Figure 4.14: Extract LPF coefficients by concatenating mirror image

the mirror image of ifft coefficient. The number of FIR filter shall be an odd number.

The amplitude of signal in its passband before and after the filtering should be the same. This can be enforced by adjusting the amplitude of the FIR coefficients. In LPF FIR filter, the amplitude of the signal after the filter converges to $\sum_{i=0}^{m} a_i$ because $\lim_{\omega \to 0} z = \lim_{\omega \to 0} e^{j\omega} = 1$. The coefficients obtained by ifft shall be divided by $\sum_{i=0}^{m} a_i$. In HPF on the other hand, the highest frequency is the Nyquist frequency, where the amplitude converges to $\sum_{i=0}^{m} (-1)^i a_i$, because $\lim_{\omega \to \pi} z = \lim_{\omega \to \pi} e^{j\omega} = -1$. The coefficients shall be divided by $\sum_{i=0}^{m} (-1)^i a_i$.

When we design a filter with `ifft`, you may wonder how to handle the complex filter coefficients produced from `ifft`. In the above example, we only use the real part of the output of `ifft` of the frequency domain signal of an LPF below the Nyquist frequency as shown in Fig. 4.15.

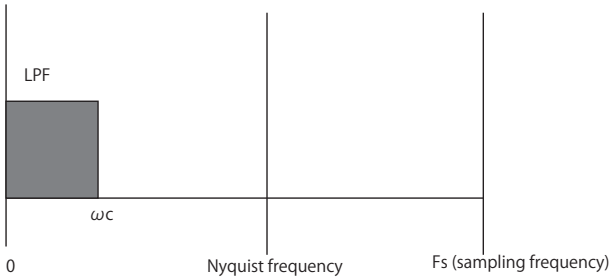

Figure 4.15: Only the frequency components below the Nyquist frequency are used to produced filter coefficients.

The comparison of the real and the imaginary coefficients and their frequency responses are shown in Fig. 4.16. In this example, the imaginary part does not produce a proper filter characteristics majorly because of the absence of the DC component.

If we define the frequency domain signal of an LPF to involve the frequency level in the alias component as in Fig. 4.17,

The complex filter coefficients and the filter characteristic are shown in Fig. 4.18. In this case, as the given spectrum is symmetric with respective to the Nyquist frequency, `ifft` produces only real part filter coefficients.

Exercise 4.3 *(ifflpf) Compose a script to create a set of LPF and HPF filter coefficients from the specified cutoff frequency response and applying iFFT as shown in Example 4.6 and replace the filter in the audio player in Example 4.2. The switch between LPF and HPF shall be implemented as a variable such that $hpf = logical(false)$; to take hpf as a boolean variable or $hpf = 1$; to handle hpf as an integer.*

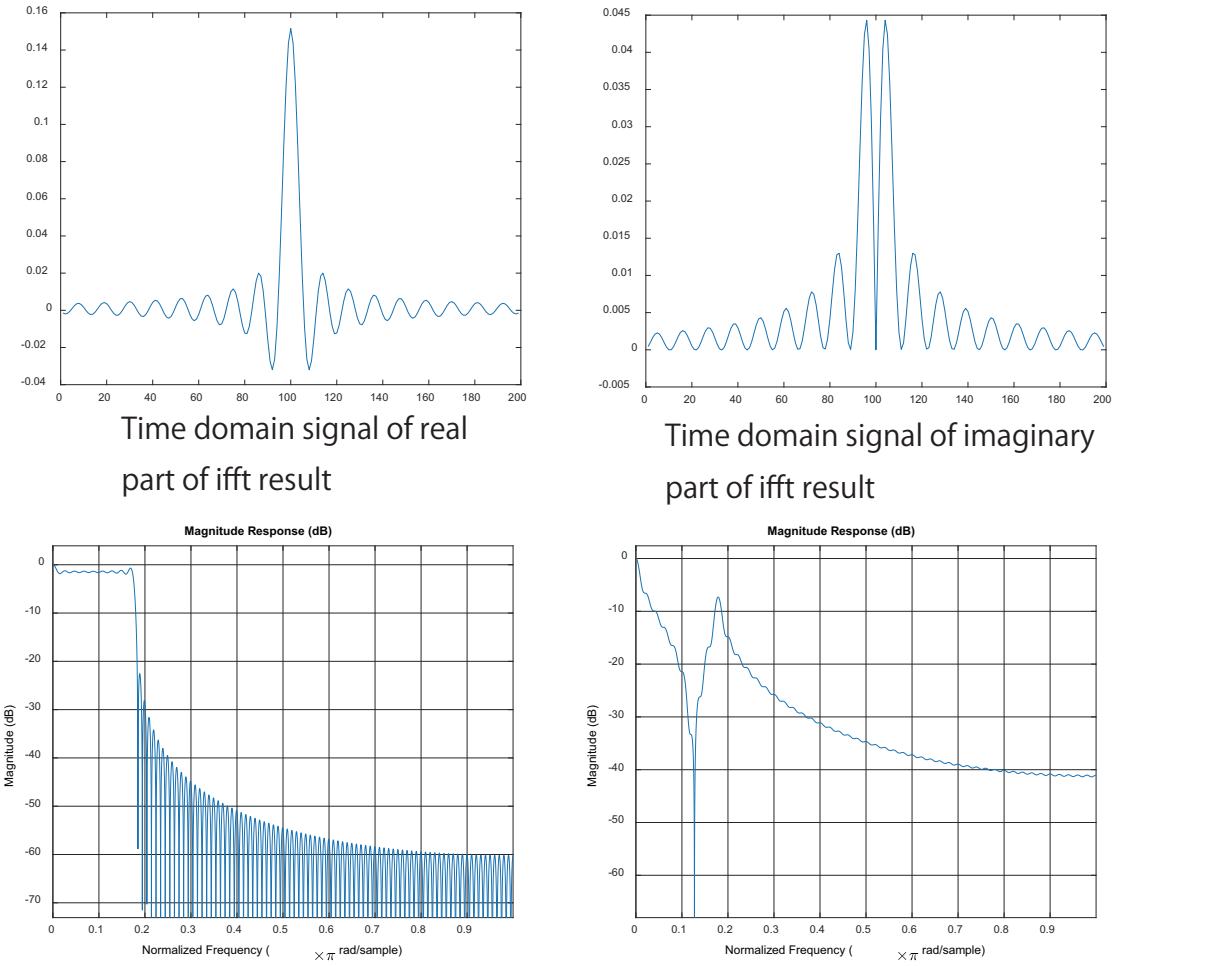

Figure 4.16: Impulse and frequency domain characteristics of LPF produced by the real and imaginary part off ifft.

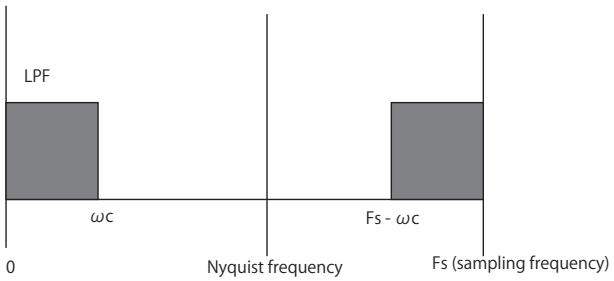

Figure 4.17: Frequency domain characteristic of filter is defined in the full spectrum

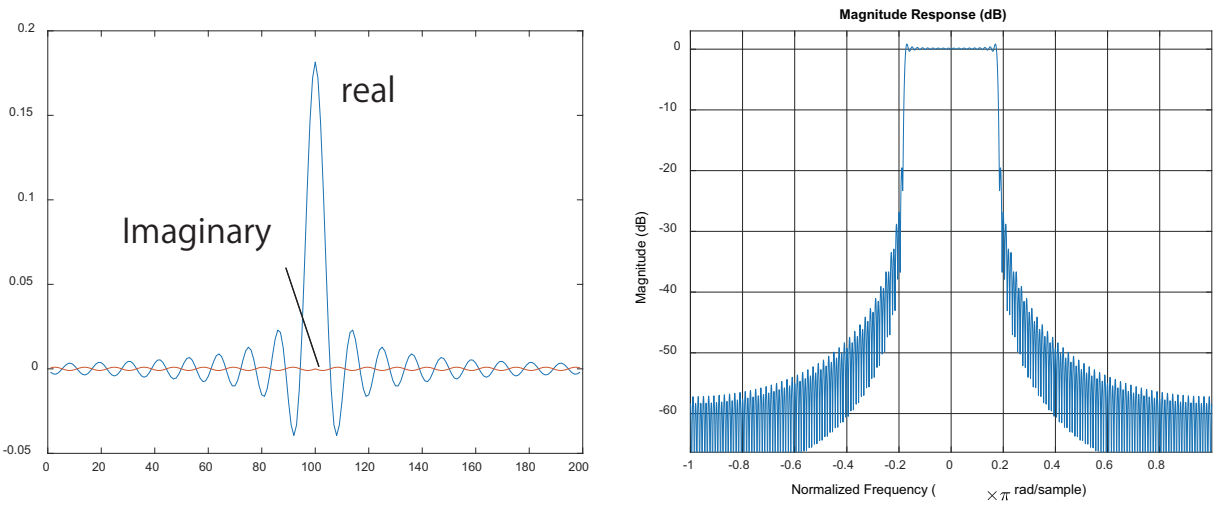

Figure 4.18: Impulse response and frequency domain characteristics of LPF produced by the complex ifft.

4.4 User interaction

In the filter exercise, we may want to dynamically turn on and off the filter to recognize the difference in the outcomes. This can be done by an event driven programming. The following is a simple example to dynamically control echo filter. An example to apply an echo filter as IIR filter is given as follows. In this example, the convolutional product is explicitly written as m script.

Example 4.7 *(musicdsp_base_echo)*

```
%
%  simple music play with graphical user interface
%
clear;
global echo;
global cont;
echo = false;
cont = true;
fig = uifigure("Name", "DSPlayer", "Position", [100 1000 300 50]);
echoButton = uibutton(fig, 'Text', 'normal', "Position", [100 20 50 20], ...
    'ButtonPushedFcn',@(btn,event)filButtonPush(btn));
stopButton = uibutton(fig, 'Text', 'cont', "Position", [180 20 50 20], ...
    'ButtonPushedFcn',@(btn,event)stopButtonPush(btn, fig));
playButton = uibutton(fig, 'Text', 'Play', "Position", [20 20 50 20], ...
    'ButtonPushedFcn',@(btn,event)playButtonPush(btn));
function playButtonPush(btn)
global cont;
global echo;
frameLength = 8192;
mfile = uigetfile('*.mp3');
fileReader = dsp.AudioFileReader(...
    mfile,...
    'SamplesPerFrame',frameLength);
Fs = fileReader.SampleRate;
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
scope = dsp.SpectrumAnalyzer('SampleRate', Fs);
psignal = zeros(frameLength, 2);
delay = floor(0.10*Fs); %100 msec delay
while ~isDone(fileReader) && cont
    pause(0.01) % this is to capture a button push
    signal = fileReader(); % read a audio file
    ss = [psignal;signal]; % use the previous output
    for i=1:frameLength
       ss(i+frameLength,:) = ss(i+frameLength,:) + 0.50*ss(i+frameLength-delay,:);
    end
    psignal = ss(1+frameLength:end, :);
    if echo
        deviceWriter(psignal);
    else
        deviceWriter(signal); % write to audio device
    end
    scope([psignal, signal]); % write to spectrum analyzer
end
release(fileReader);
release(deviceWriter);
release(scope);
end
function filButtonPush(btn)
global echo;
echo = xor(echo, true);
if echo
    btn.Text = "echo";
else
    btn.Text = "normal";
end
end
function stopButtonPush(btn, fig)
global cont;
cont = xor(cont, true);
pause(5);
close(fig);
```

```
end
```

Exercise 4.4 *(musicdsp_base_ave) Compose an m script of mp3 player with a dynamic control of averaging FIR filter of tap number up to 20. While an mp3 file is played, the averaging filter can be toggled between on and off everytime the user pushes a button as in Fig. 4.19.*

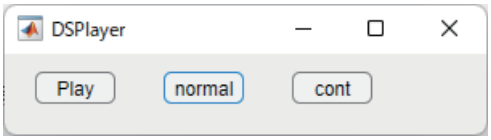

Figure 4.19: mp3 player with dynamical control of averaging filter example

5 Image processing

Similar to the case of digital filtering, image processing with MATLAB can be done easily with a number of libraries and toolkit. Still, the minimum theoretical knowledge is essential to properly exploit those tools.

5.1 Importing image

Any image file can be imported to MATLAB with `imread` function. The following instruction captures the image file `seesaa.jpg` from the current directoy to the workspace.

```
I = imread('seesaa.jpg');
```

`I` is an array contains image data. The dimension of `I` depends on the type of image. The information about the image data can be retrieved by `whos` command.

```
>> whos I
  Name         Size                    Bytes  Class    Attributes

  I         3648x5472x3            59885568  uint8
```

The above reveals the image data contains `3648x5472x3 uint8` (unsigned int 8bits) array. The leading `3648x5472` means the number of pixels in the vertical and horizontal axes. The last `x3` means that one pixel is represented by 3 samples, which is referred to as bit depth. Therefore, one pixel is represented with 24 bits (=16,777,216 color variants). The 24 bits comprises of three 8 bits for red, green and blue (RGB). Such information can be retrieved by `imfinfo`.

```
>> a = imfinfo('billi.jpg')
a =
  struct with fields:
           Filename: 'C:\Users\mitsugi\dsp\billi.jpg'
        FileModDate: '25-Mar-2017 21:08:48'
           FileSize: 4968844
             Format: 'jpg'
      FormatVersion: ''
              Width: 5472
             Height: 3648
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {}
   ImageDescription: '                                '
               Make: 'Canon'
              Model: 'Canon PowerShot G7 X'
        Orientation: 1
        XResolution: 180
        YResolution: 180
     ResolutionUnit: 'Inch'
           DateTime: '2017:03:25 21:08:49'
             Artist: ''
   YCbCrPositioning: 'Co-sited'
          Copyright: ''
```

The image can be shown to the computer screen by `imshow(I)`. If your image is large like the example above, MATLAB warns as follows.

```
>> imshow(I)
Warning: Image is too big to fit on screen; displaying at 17%
> In images.internal.initSize (line 71)
  In imshow (line 328)
```

We can change the image data size by `imresize` such that

Figure 5.1: Simple image by specifying the color depth

```
C = imresize(B, 0.2);   %this reduce the image data B with scale 0.2.
```

The sizes of an image can be retrieved with `size` command. The following returns the number of pixels in vertical and horizontal axis and the depth.

```
I = imread('seesaa.jpg'); % read seesaa.jpg from the current directory
[r, c, d] = size(I);           % obtain row, column and depth of the image
```

Exercise 5.1 *(resizeshow) Compose a script to load an image from the current directory and resize it to have 640 pixels in horizontal axis and show the resized image to the screen. The number of pixels in the vertical axis should be adjusted automatically.*

5.2 Creating image

A straightforward way to create an image in `MATLAB` is to define the pixel value in three dimensional array such that

Example 5.1 *(sfclogod)*

```
mm = zeros(8,8,3);
mm(:,:,1) = [ 0 0 0 0 0 0 0 0;...
              0 0 0 0 0 0 0 0;...
              0 0 255 255 255 255 0 0 ; ...
              0 0 255 0 0 0 0 0 ; ...
              0 0 255 255 255 255 0 0 ;...
              0 0 0 0 0 255 0 0 ;...
              0 0 255 255 255 255 0 0 ;...
              0 0 0 0 0 0 0 0];...
mm(:,:,2) =mm(:,:,1);
mm(:,:,3)= mm(:,:,1);
imshow(mm); .
```

to obtain the image shown in Fig.5.1.

There are two ways to draw an image in `MATLAB`. `imshow` is one of them, which is available in Image Processing Toolbox. `imshow` has many features such as specifying colormap. The other is `image` which simply produces an image from a matrix.

Because we don't need to have full color (24 =8x3 bits) representation in many case, we can define a color map, which is a set of aliases of limited set of full color and simplify the description of images. Let us define an `4x3` array `a` and using `image` function to produce an image such that

Example 5.2 *(createImage)*

```
a = [1 2 3 4 ; 5 6 7 8; 1 2 3 4];
image(a);
```

This scripts produces an image as Fig.5.2.
The values in the `a` *and the color in the image is interrelated with a table called* `colormap`*. The default colormap is* `parula`*. MATLAB has built in colormaps as shown in Fig.5.3.*

The colormap can be change by `colormap` *command. To change the colormap to* `flag`

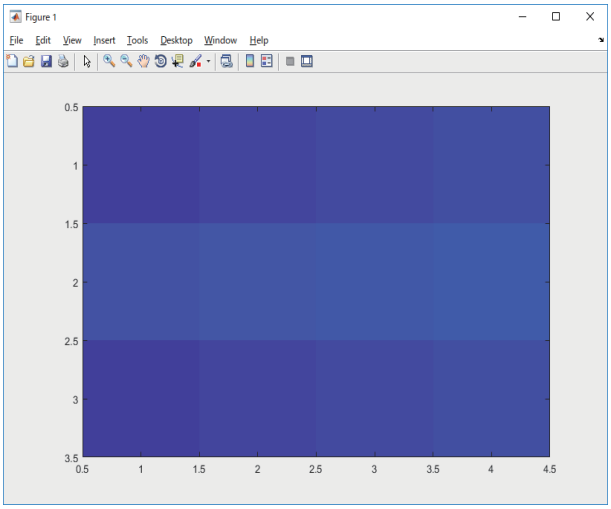

Figure 5.2: Create an image from an array example

```
colormap(flag);
```

The image on the screen automatically changes as Fig.5.4 although the array `a` *is intact. There are, in default, 256 colors in a colormap.*

We can create our own colormap too. The following script creates an original colormap by specifying the RGB combination in an array such that

```
>> mymap =[ 1,1,0 %yellow
1,0,1   %magenta
0,1,1   %cyan
1,0,0   %red
0,1,0   %green
0,0,1   %blue
1,1,1   %black
0,0,0]; %white
```

and apply this original map by

```
>> colormap(mymap).
```

This changes the image on the screen as Fig.5.5. In this colormap, there 8 colors. The created image can be saved as a specified image file format with `imwrite`*.*

imwrite(a, mymap, 'im.jpg');

In many cases, the range of probable data values, each of which shall be classified into a colormap, are different from the number of colormap[2]. The scaling of colormap can be specified by `CDataMapping` property of the current image. The property can be either default `direct` or `scaled`. With the `direct` option, the color distribution in the image is mapped onto the colormap one by one while `scaled` automatically scale the color distribution to the colormap. In the following `m` script, we have 16 data values, from -8 to 7, to be mapped with 64 gray colormap. The `direct` option maps -8 to the first colormap and data 7 to 16th colormap, while the `scale` option maps -8 to the first colormap and 7 to 64th colormap. We can check the relation with `colorbar` facility as shown in Fig. 5.6.

[2]This text may be difficult to follow. Don't worry, we will show a concrete example soon.

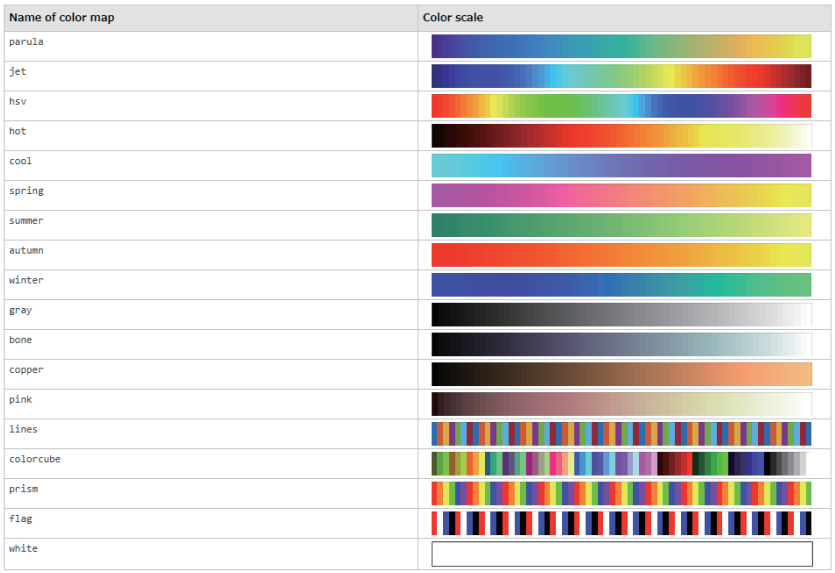

Figure 5.3: MATLAB built in colormaps

```
%
%  testcolor.m
%
a = (-8:1:7); % create data from -8 to 7 = 16 variations
colormap(gray(64)); % gray scale from 1 to 64
subplot(1,2,1);
im = image(a);
ax = gca;
ax.Title.String='scaled';
% data -8 mapped to colormap 1,  data 7 to colormap 64
im.CDataMapping = 'scaled';
colorbar;
subplot(1,2,2);
im = image(a);
ax = gca;
ax.Title.String='direct';
%  data -8 mapped to colormap 1, data 7 to colormap 16
im.CDataMapping = 'direct';
colorbar;
%caxis([]); % use from 10 to 20
```

The result is given as in Fig. 5.6.

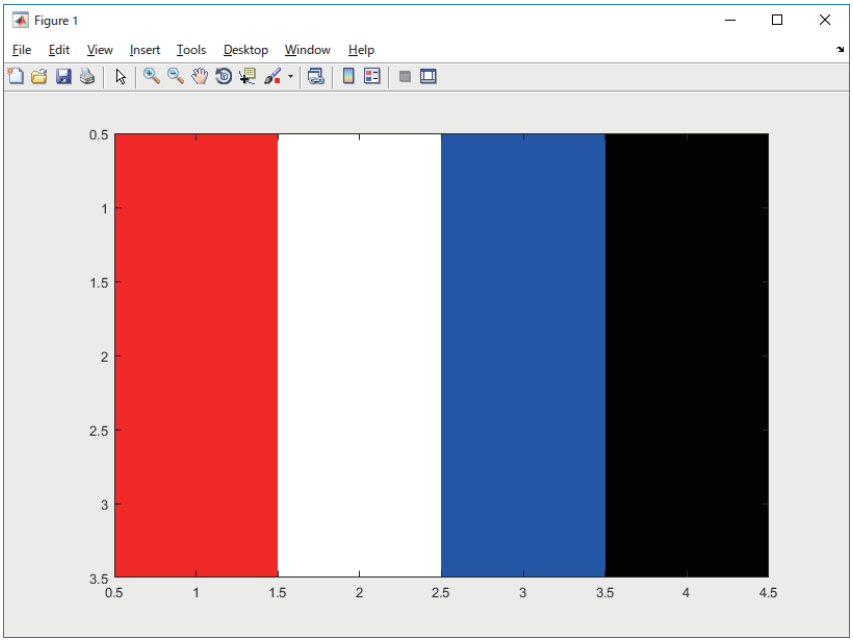

Figure 5.4: Example image with flag colormap

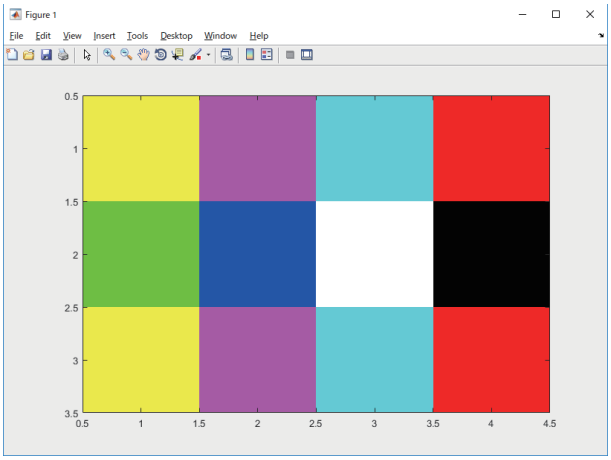

Figure 5.5: Example image with an original colormap

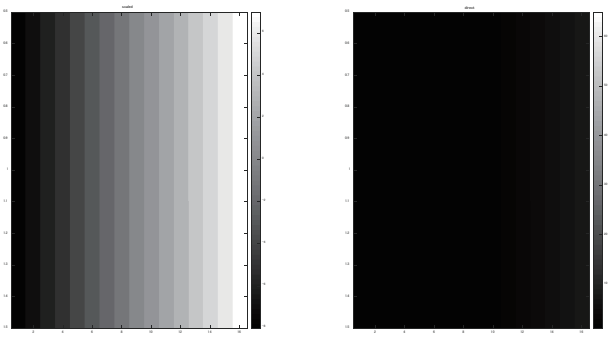

Figure 5.6: CDataMapping defines the scaling of colormap and data

Figure 5.7: Image created from an array

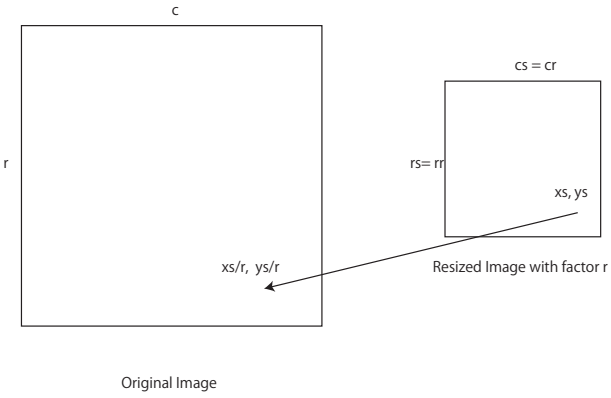

Figure 5.8: Resizing an image with factor r

Exercise 5.2 *(sfclogo) Compose a script to produce a jpeg image file (sfc.jpg) including three alphabets, sfc, from a* `8x24` *array. The font color is black and the background is yellow as shown in Fig.5.7.*

Exercise 5.3 *(scaleimage) In the previous example, we use* `imresize` *to change an image size. But a resizing can be performed with a simple procedure. Let us suppose we resize an original image with scale r. The original row and column numbers r, c are resized to r_s, c_s, respectively as shown in Fig.5.8. Any pixel pixel value at x_s, y_s in the resized image can be related to that of a pixel in the original image with the following relation.*

$$P_r(x_s, y_s) = P_o(x_s/r, y_s/r) \tag{5.1}$$

where P_r and P_o represent the pixel value at resized and original images, respectively. Compose a script to resize a given image with a specified resize scale. The following specifications shall be met.

- *The jpeg file to be resized shall be picked from the list of jpeg files in the current directory.*
- *The resize factor shall be specified by using* `input` *function*

You can create a new canvas for the scaled image by `figure` *command. As the default setting of new* `figure` *is to automatically adjust the size of image, you may want to specify non-resizing such that*

```
figure('Name', 'Original', 'NumberTitle', 'off', 'Resize', 'off') ;
```

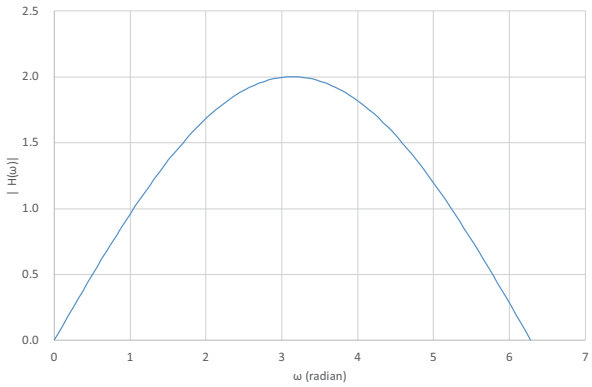

Figure 5.9: Edge detection is equivalent to a HPF

5.3 Edge detection

Edges in an image are where the pixel values change abruptly. Suppose we have an image which has r rows and c columns pixels and we focus on (x, y) pixel. The change d_x at x, y pixel whose pixel value is $v(x, y)$ with the neighbor pixel in the vertical axis can be defined as

$$d_v(x, y) = v(x, y) - v(x - 1, y) \tag{5.2}$$

and

$$d_h(x, y) = v(x, y) - v(x, y - 1) \tag{5.3}$$

in the horizontal axis. Since pixels are counted as integer, these equation can be understood as spacial sampling with sampling period (interval) is 1. The discrete series $d_v(1, y), d_v(2, y), \cdots, d_v(c, y)$ and $v(1, y), v(2, y), \cdots, v(c, y)$ can be related with a filter series H such that

$$D_v(z) = H(z)V_v(z) \tag{5.4}$$

where

$$H(z) = b_0 z^0 + b_1 z^{-1} = 1 - z^{-1}, \tag{5.5}$$

and we rewrite d_v and v to D_v and V_v since they represent the whole series in the vertical axis.

Similarly in the horizontal axis, we have

$$D_h(z) = H(z)V_h(z). \tag{5.6}$$

The frequency response of the filter can be obtained by substituting $z = e^{j\omega}$ such that

$$H(\omega) = 1 - e^{-j\omega} \tag{5.7}$$

The gain of the response is zero in low frequency region and the maximum is 2 as shown in Fig.5.9. This is a high pass filter. To apply this filter to an image, there is no pixel to calculate the edge in the boundary as shown in Fig.5.10. To avoid this lack of data, we assume that there is additional column beyond (below) the boundary which has the symmetric pixel values. Thus, the first column of $D_v(z, y) = 0$.

Example 5.3 *(singlediffilter)*

The following script applies the edge detection filter to an image file. Note that d may exceed the maximum value 8, of gray(8) colormap, or below the minimum value 1, of gray(8). In such case, the maximum and the minimum values are used to draw the image.

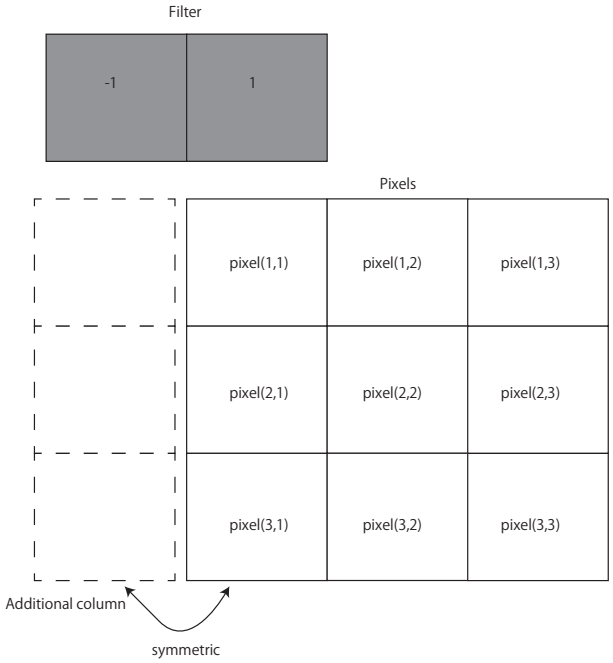

Figure 5.10: Empty pixel at the boundary

```
%% sigldiffilter
%
I = uigetfile({'*.jpg'; '*.bmp'}, 'File Selector');
b = imread(I);
bw = double(rgb2gray(b));
[r,c,d] = size(bw);
colormap(gray(8));
%colormap('flag');
dv = double(zeros(r,c));
dh = double(zeros(r,c));
for i=1:r
    for j=1:c
        if (i == 1)
          imnus1 = i+1;
        else
          imnus1 = i-1;
        end
        if (j == 1)
          jmnus1 = j+1;
        else
          jmnus1 = j-1;
        end
        dv(i,j) = ceil(9-abs(bw(i,j) - bw(imnus1,j)));
        dh(i,j) = ceil(9-abs(bw(i,j) - bw(i,jmnus1)));
    end
end
d = (dv+dh)/2.0;
subplot(1,2,1), image(d);
subplot(1,2,2), image(b);
```

The mapping between the pixel number and the effective color map can be automatically scaled with `CDataMapping` *as follows.*

```
%% sigldiffilter
%
I = uigetfile({'*.jpg';'*.bmp'}, 'File Selector');
b = imread(I);
bw = double(rgb2gray(b));
```

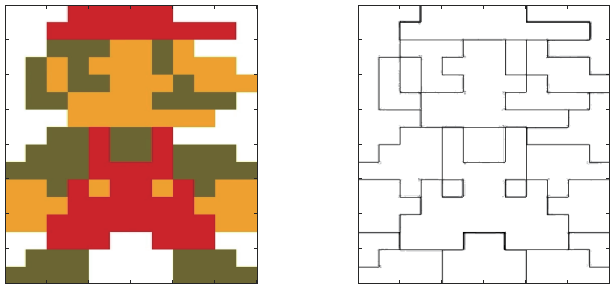

Figure 5.11: Edge detection with an image filter

```
[r,c] = size(bw);
colormap(gray(8));
colorbar;
%colormap('flag');
dv = double(zeros(r,c));
dh = double(zeros(r,c));
for i=1:r
    for j=1:c
        if (i == 1)
          imnus1 = i+1;
        else
          imnus1 = i-1;
        end
        if (j == 1)
          jmnus1 = j+1;
        else
          jmnus1 = j-1;
        end
        dv(i,j) = (9-abs(bw(i,j) - bw(imnus1,j)));
        dh(i,j) = (9-abs(bw(i,j) - bw(i,jmnus1)));
    end
end
d = uint8((dv+dh)/2.0);
%f = gcf;
subplot(1,2,1);
im=image(d);
im.CDataMapping = 'scaled';
subplot(1,2,2), image(b);
```

An example of edge detection with the above filter is shown in Fig.5.11 The edges are successfully detected.

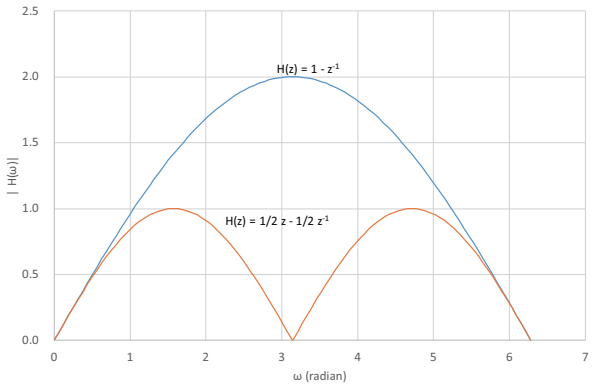

Figure 5.12: Prewitt filter frequency response

The difference with the neighbor pixel can be modeled as a derivative operation. For this reason above $H(z) = 1 - z^{-1}$ is referred to as a derivative filter. The definition of derivative $\frac{df}{dx}$ of a continuous function $f(x)$ with respect to x is given by the following equation.

$$\frac{df}{dx} = \lim_{\Delta x \to \infty} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{5.8}$$

This can be rewritten in the following form with a finite length Δx

$$f(x + \Delta x) \approx f(x) + \frac{df}{dx}\Delta x + O(\Delta x^2) \tag{5.9}$$

where $O(\Delta x^2)$ represents the error of this approximation may include order of Δx^2. If we take the following alternative

$$f(x - \Delta x) \approx f(x) - \frac{df}{dx}\Delta x + O(\Delta x^2) \tag{5.10}$$

Subtracting Eq.5.10 from Eq.5.9 yields

$$f(x + \Delta x) - f(x - \Delta x) = 2\frac{df}{dx}\Delta x + O(\Delta x^3). \tag{5.11}$$

This means that we can approximate the derivative more accurately with the following formula.

$$\frac{df}{dx} = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} \tag{5.12}$$

Therefore, an alternative derivative filter series can be obtained as follows.

$$H(z) = \frac{1}{2}z - \frac{1}{2}z^{-1} \tag{5.13}$$

The frequency domain response of this filter, Prewitt filter, is shown in Fig.5.12. Prewitt filter has similar filter characteristics in low frequency and also cuts high frequency region, which works to eliminate noises.

Exercise 5.4 *(prewitt) Compose a script to apply Prewitt filter to detect edges by modifying Example 5.3.*

Example 5.4 *(mariolaplace) Adding Eq. 5.9 and Eq. 5.10 yields the second derivative such that*

$$\frac{d^2f}{dx^2} = \frac{f(x + \Delta x) + f(x - \Delta x) - 2f(x)}{(\Delta x)^2}. \tag{5.14}$$

Arranging in two dimensional as shown in Fig.5.13 produces a filter called Laplace filter.

Since Laplace filter is the second derivative, it captures the change of the first derivative. Because a peak value of the first derivative becomes zero in the second derivative, Laplace filter for edge detection needs to reverse the color map as shown the following example.

0	1	0
1	-4	1
0	1	0

Figure 5.13: Laplacian Filter

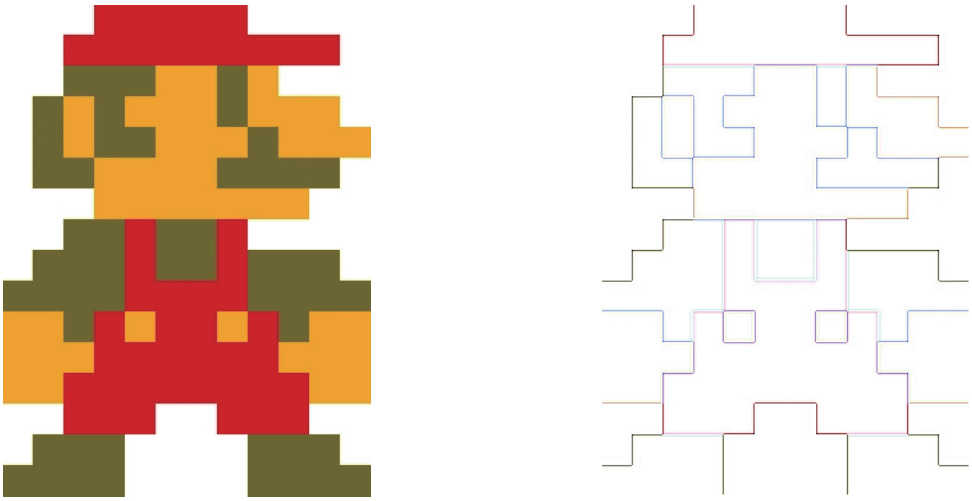

Figure 5.14: Laplacian Filter to detect edge

```
%% laplace edge detection
%  mariolaplace.m
I = imread('mario.bmp');
[r,c,d] = size(I);
R = uint8(zeros(r,c,d));
for i=2:r-1
    for j=2:c-1
       for k=1:d
          a = ceil((double(I(i-1,j,k))-4.0*double(I(i,j,k))+ ...
               double(I(i+1,j,k))+double(I(i,j-1,k))+double(I(i,j+1,k))));
          R(i,j,k) = a;
       end
    end
end
% reverse the colormap
Rr = imcomplement(R);
imshow(Rr);
```

The filter produces sharp edge such as shown in Fig.5.14.

Figure 5.15: Artificially noises are added to an image

5.4 Noise Rejection

5.4.1 Noise Addition

Noise can be conceived as unwanted high frequency components in an image.Let us first artificially add noises to an image. For the clarity of the outcome, we convert a color image to a gray scale image by using `rgb2gray` function and add randomly generated white spots. The noise added image is saved under the filename with `_n` in the current directly.

Example 5.5 *(add_noise.m)*

```
%% add_noise
filename=uigetfile('./*.jpg');
I = imread(filename);
g = rgb2gray(I);
J = addnoise(g, 0.999);
subplot(1,2,1), imshow(I),title("oiginal");
subplot(1,2,2), imshow(J), title("noise");
% save the noise added image with _n
[filepath,name,ext] = fileparts(filename);
imwrite(J, strcat(name,'_n',ext));
function J =  addnoise(g, p)
[row, col, c] = size(g);
J = uint8(zeros(row,col,c));
for i=1:row
    for j=1:col
        if rand > p
            n = 255;
        else
            n = 0;
        end
        for k=1:c
           J(i,j,k) = g(i,j) + n;
        end
    end
end
end
```

We can add white dots to an image as shown in Fig.5.15.

MATLAB tips: Parse file name and extension

`fileparts` function parses a filename into path, filename and extension. The extension includes a full stop at the beginning of the string.

```
[filepath,name,ext] = fileparts(filename);
```

`strcat` concatenates strings

```
imwrite(J, strcat(name,'_n',ext));
```

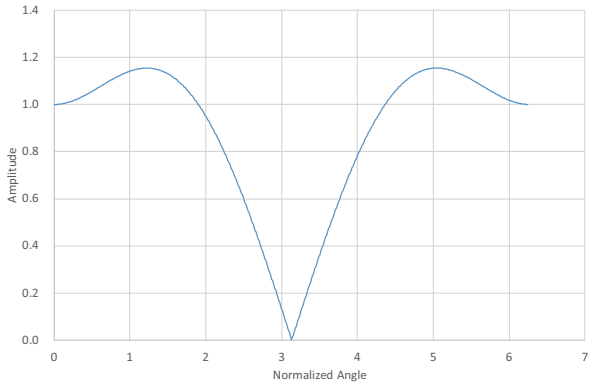

Figure 5.16: Average filter frequency response

5.4.2 Noise Removal

Low pass filter works to suppress noise. A simple low pass filter is an averaging filter. A simple average filter for the vertical direction can be represented by the following equation

$$a_v(x,y) = \frac{v(x,y) + v(x-1,y)}{2}, \tag{5.15}$$

and

$$a_h(x,y) = \frac{v(x,y) + v(x,y-1)}{2} \tag{5.16}$$

in the horizontal axis. Since pixels are counted as integer, these equation can be understood as spacial sampling with sampling period (interval) is 1. The discrete series $a_v(1,y), a_v(2,y), \cdots, a_v(c,y)$ and $v(1,y), v(2,y), \cdots, v(c,y)$ can be related with a filter series H such that

$$A_v(z,y) = H(z)V(z,y) \tag{5.17}$$

where

$$H(z) = b_0 z^0 + b_1 z^{-1} = 0.5 + 0.5z^{-1}. \tag{5.18}$$

This provides the frequency response as shown in Fig.5.16. While the low frequency region is retained, the high frequency region is suppressed. The above average filter is a two tap LPF, accordingly. If we increase the number of taps to five yielding the following five tap averaging filter.

$$H(z) = \frac{b_2 z^2 + b_1 z^1 + b_0 z^0 + b_1 z^{-1} + b_2 z^{-2}}{5} = 0.2z^2 + 0.2z^1 + 0.2 + 0.2z^{-1} + 0.2z^{-2} \tag{5.19}$$

Increasing the number of taps provides a steep LPF such as in Fig.5.17

Example 5.6 *(noise_lpf)*

The following script produces an average filtered image.

```
%% noise removal with LPF.m
%
I = imread('seesaa_n.jpg');
R = average(I);
subplot(1,2,1), imshow(I);
subplot(1,2,2), imshow(R);
%% average filter
%
function R = average(Ig)
  [r, c, d] = size(Ig);
  R = double(zeros(r,c,d));
```

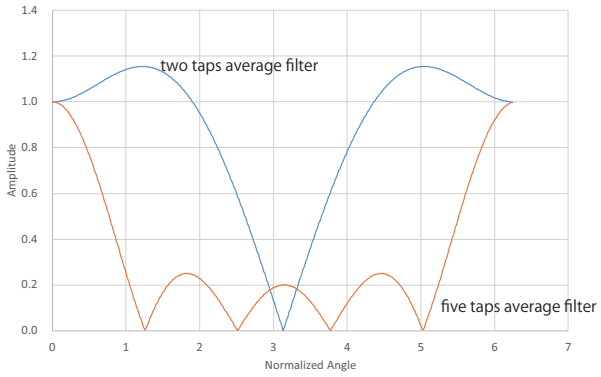

Figure 5.17: Five taps averaging filter response

Figure 5.18: Noise rejected with LPF

```
  for i=2:r-1
    for j=2:c-1
        for jj=1:d
          R(i,j,jj) = (double(Ig(i-1, j,jj))+double(Ig(i,j,jj))+double(Ig(i+1,j,jj))...
          + double(Ig(i, j-1,jj))+double(Ig(i,j+1,jj)))/5.0;
        end
    end
  end
  R = uint8(R);
end
```

The example output of the noise rejection filter is as shown in Fig.5.18.

The noise removal works but it only blurs the noises. This is because the LPF only suppress a part of high frequency caused by the noise. When an image has outliers noise, a mere LPF is not sufficient. In order to remove outliers, median filter is a good choice. Median is the middle value in a group of numbers. Suppose we have the distribution of pixel values as in the left figure of Fig.5.19. The thick cells are the ”kernel” with which we replace the pixel value at the center. Applying a median filter, the outlier (noise) of 255 is replaced with 127 in this case.

Exercise 5.5 *(noise_median) Compose a script to remove noises with median filter.*

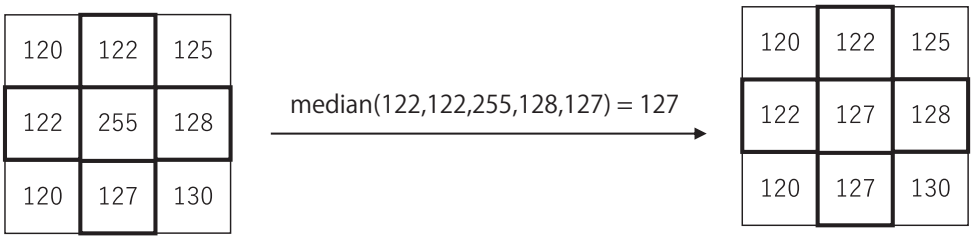

Figure 5.19: Median filter example

Figure 5.20: Noise removal with median filter

Example 5.7 *(createczp) Circular zone plate (CZP) is to check if there is sufficient resolution in an image. If the resolution is not high enough, we see aliases of high frequency component in an image as is in the case of tones. Suppose we have an image of $H \times V$ pixels and the number of colormap is n_c in grayscale, a CZP is a concentric circle with a varying color as we scan from the center to the peripheral. The colormap of x, y coordinate $c(x, y)$ is given by the following equation.*

$$c(x,y) = \frac{n_c}{2}\sin(\frac{\pi}{\alpha H}x^2 + \frac{\pi}{\alpha V}y^2) + \frac{n_c}{2} \tag{5.20}$$

where α is a parameter to control the varying color rate. Let us example the CZP equation. In case of $\alpha = 1$, at the boundary $x = \frac{H}{2}$, we have the following sinusoidal in the x axis.

$$\sin(\frac{H^2\pi}{4H}) = \sin(\frac{H\pi}{4}) \tag{5.21}$$

If we move one more pixel farther, the phase change will be

$$\frac{(H/2+1)^2\pi}{H} - \frac{(H/2)^2\pi}{H} = \frac{(H+1)\pi}{H} \tag{5.22}$$

Because $H \gg 1$, the phase change can be approximated as π, which is equivalent to the Nyquist frequency. An example of CZP creation in MATLAB is as follows.

```
% create czp
%
row=600;
col=600;
alpha = 1;
nmap = 64;
colormap(gray(nmap));
czp = zeros(row,col);
for i=1:row
    for j=1:row
       dep = nmap/2*sin(pi()*(i-row/2)*(i-row/2)/alpha/row ...
           +pi()*(j-col/2)*(j-col/2)/alpha/col) + nmap/2;
       czp(i,j) =dep;
    end
end
image(czp);
imwrite(czp, 'czp.bmp');
```

This script produces an image as shown in Fig.5.21 and save it as `czp.bmp`*. Since the figure is resized to insert in a latex document, there might be aliases appeared.*

Exercise 5.6 *(createczpsmall) Open a simple paint software like* `paint` *on windows and change the size of* `czp.bmp` *by half to observe how the image changes. Compose a* `m` *file which produces a half size (300x300 pixels) of CZP by resizing the previous CZP image.*

When we resize (particularly shrink) an image, the loss of resolution results in the generation of aliases. The image of high frequency components emerge as its image in low frequency. In order to avoid this alias, we first need to suppress high frequency components before the resizing.

Example 5.8 *(fir2d) We apply a two dimensional FIR filter to suppress high frequency component in an image. Different from time series, an image have relatively small number of pixels. Therefore, the tap number of FIR filter to an image should be relatively small. We lend ourselves to use 6 taps FIR filter to suppress the half of Nyquist frequency. In* `filterDesigner`*, we specify the parameters as shown in Fig.5.22. The obtained numerators are as follows.*

$$b = \left\{ \begin{array}{cccccc} -0.1513 & 0.1926 & 0.5501 & 0.5501 & 0.1926 & -0.1513 \end{array} \right\} \tag{5.23}$$

Using the numerator, the two dimensional filter can be established. For simplicity, we don't apply the filter to the peripheral pixels.

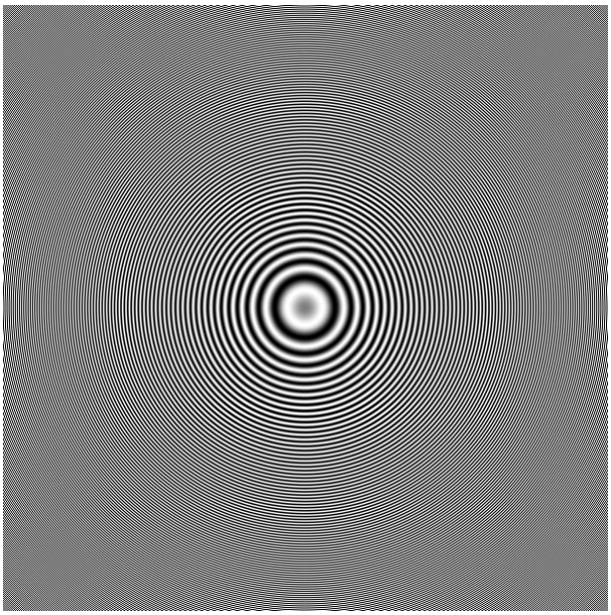

Figure 5.21: Circular Zone Plate example

```
%% fir2d
%
filename = uigetfile('./*.jpg,./*.bmp');
b = imread(filename);
colormap(gray(64));
imshow(b, gray(64));
title('original cpz');
truesize;
imwrite(b, gray, 'czpbefore.bmp');
[r,c,d] = size(b);
fir = [-0.1513 0.1926 0.5501 0.5501 0.1926 -0.1513];
dv = double(zeros(r,c,d));
dh = double(zeros(r,c,d));
for i=1:r
    for j=1:c
        for k=1:d
          dv(i,j,k) = 0.0;
          dh(i,j,k) = 0.0;
          for rr = -3:2
            [ci, cj] = boundaryCheck(rr, i, j, r, c);
            dv(i,j,k) = dv(i,j,k)+fir(rr+4)*double(b(i,cj, k));
            dh(i,j,k) = dh(i,j,k)+fir(rr+4)*double(b(ci,j, k));
          end
        end
    end
end
d = uint8((dv+dh)/2);
figure;
%colormap(gray);
imshow(d, gray(64));
title('filtered cpz');
truesize;
imwrite(d, gray(64), 'czpafter.bmp');
function [ci, cj] = boundaryCheck(rr, i, j, r, c)
if j+rr > c
    cj = c;
elseif j+rr < 1
    cj = 1;
else
    cj = j+rr;
end
```

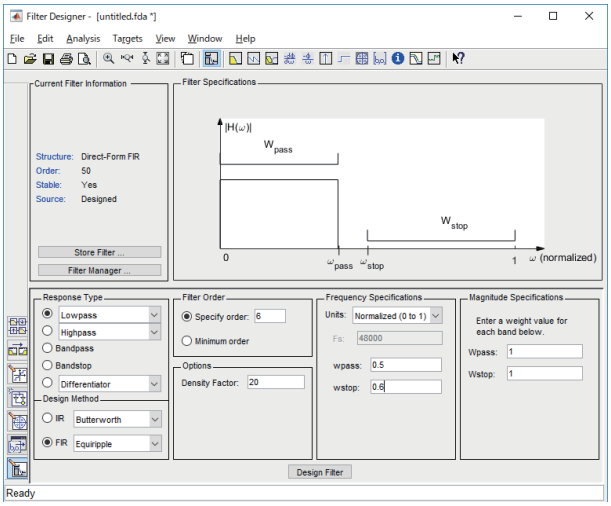

Figure 5.22: FIR filter design parameters

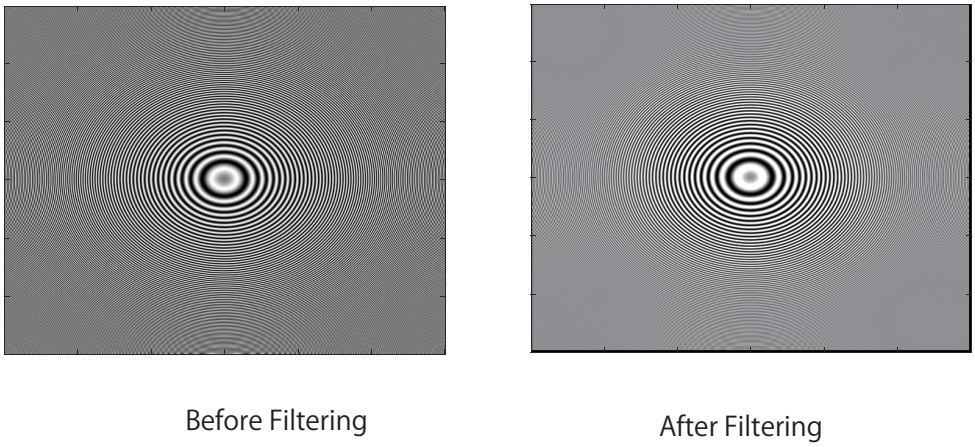

Figure 5.23: FIR filtered CZP

```
if i+rr > r
    ci = r;
elseif i+rr<1
    ci = 1;
else
    ci = i+rr;
end
end
```

This program produces a filtered image as shown in Fig.5.23.

Exercise 5.7 *(firimage02) Compose an m script, which takes a scale factor and produces three figures.*

- *original Nyquist CPZ*
- *Scaled CPZ using your own scaleimage (assignment #8)*
- *Scaled CPZ after applying a low pass filter to eliminate alias associated with the scaling. The coefficients of the low pass filter should better be generated automatically.*

5.5 Morphological Transformation

Morphology[3] is a set of image processing operations that process images based on shapes. The basic morphological transformations are dilation and erosion[2]. Dilation adds pixels to the boundary of images to provide inflating (dilating) effect. On the other hand, erosion deletes pixels from the boundary to provide shrinking effect. Morphological transformation is performed with a basic shape referred to as kernel. A kernel has an anchor pixel usually at the center pixel of kernel. If the anchor pixel is replaced with the brightest color in the kernel, it provides an dilating effect. On the other hand, the anchor pixel is replaced with the darkest color, it yields an eroding effect. Typical kernel is a square but it is not limited to.

Example 5.9 *(morphology)*

The following script produces either an erode or a dilate image by using `morph` *function with three times three square kernel. Erode or dilate can be specified by the second variable of the* `morph` *function. In order to avoid the unprocessed pixels in the peripheral, we assume that we have an mirror image of pixels beyond the negative boundary as shown in Fig.5.10. By including a function and invoking application in a file, we can repeatedly use the function.*

```
%% morphology
%
Is = imread('mario.bmp');
Ig = rgb2gray(Is);
imshow(Ig);
 % the second argument is 'erode' for eroding.
P = morph(Ig, 'dilate');
% we can repeat the morphological transformation.
P = morph(P, 'dilate');
P = morph(P, 'dilate');
imshow(P);
%
% morphology transformation
%
function R = morph(Ig, op)
  [r, c] = size(Ig);
  R = uint8(zeros(r,c));
  for i=1:r
    for j=1:c
        mc = Ig(i,j);
        for k = -1:1
            for s = -1:1
               cr = i+k;
               cc = j+s;
               if (i+k<1)
                   cr = abs(i+k)+1;
               elseif (i+k>r)
                   cr = 2*r-i-k;
               end
               if (j+s<1)
                   cc = abs(j+s)+1;
               elseif (j+s>c)
                   cc = 2*c - j-s;
               end
               if Ig(cr,cc) < mc && strcmp(op, 'erode')
                   mc = Ig(cr,cc);
               elseif Ig(cr,cc) > mc && strcmp(op, 'dilate')
                   mc = Ig(cr,cc);
               end
            end
        end
        R(i,j) = mc;
    end
  end
end
```

The script can produce the transformation as shown in Fig.5.24.

[3]Morphology is a branch of bilology that deals with the form and structure of animals and plants, or an study of structure or form

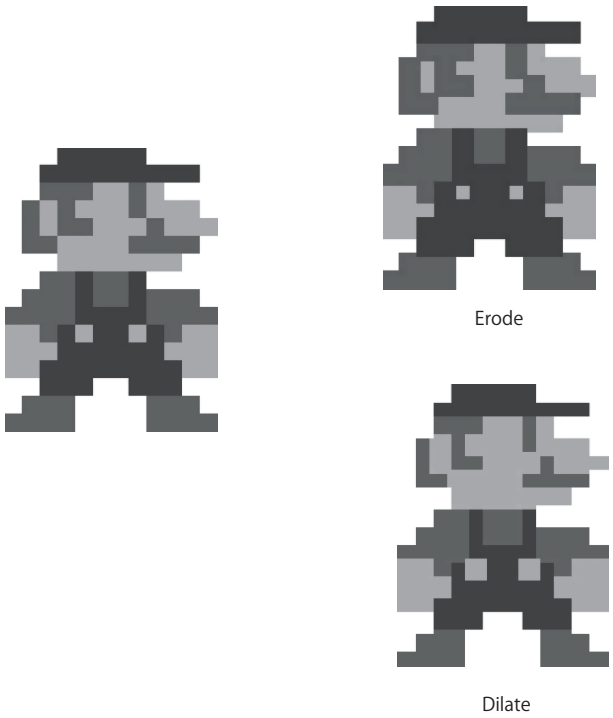

Figure 5.24: After three times dilating and eroding

Exercise 5.8 *(simplemorph) Compose a script to perform morphological transformation to a color image as shown in Fig.5.25.*

Figure 5.25: Morphological transformation of a color image

Figure 5.26: Example of Skeltonization

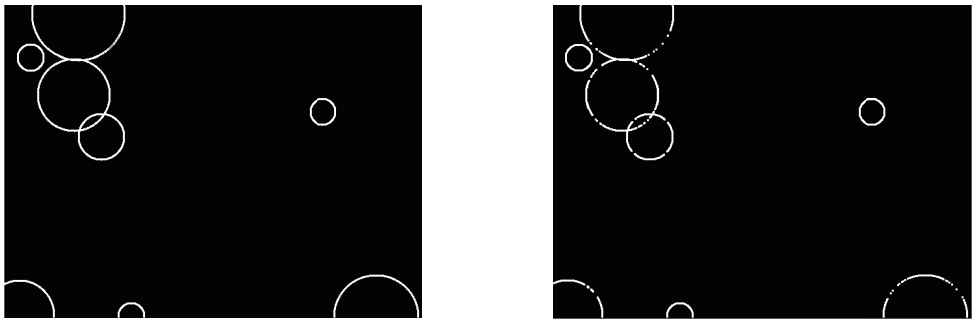

Figure 5.27: After opening of circles image

5.6 Skeltonization with morphological transformation

An interesting application of morphological transformation is skeltonization, which is a representation of an image with thin wires like in Fig. 5.26.

5.6.1 Opening and closing

There are two popular operations using Morphological transformation, opening and closing. Opening is an eroding followed by a dilation (Fig. 5.27). Closing is the opposite (Fig. 5.28).

By an opening, we remove a white pixel which surrounding by black pixels. Such white pixel is a noise, or an point on a white skelton wire because it would be disappeared by another erosion.

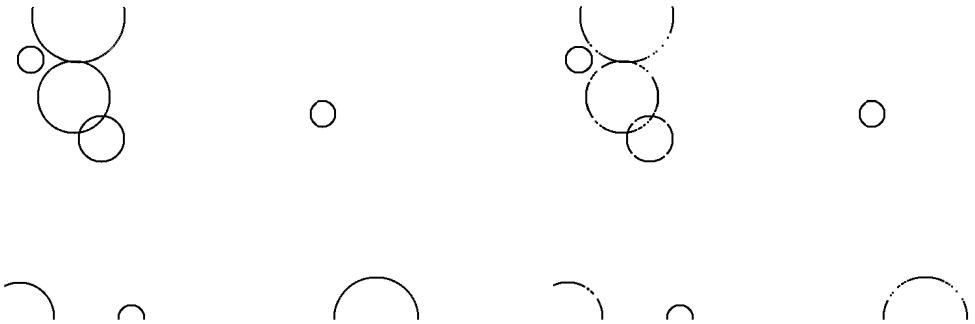

Figure 5.28: After closing of circles image

Figure 5.29: Curves before skeltonizing

5.6.2 Skeltonization

Suppose we have a binary image with thick white curves as in Fig. 5.29 before skeltonizing.

The following script produces a skeltonized image by repeatedly detecting the white pixels, which would be disappeared with another eroding. The `imgsubtract` function is the feature of skeltonization. `openimage` given to `imgsubtract` is the image where isolated white pixels are removed, and `img` is the image before the opening. In the `imgsubtract` function, the difference of the original image and the opened image are extracted by the exclusive or operation. By doing this, the isolated white pixels which were removed by the opening are given the true logic. By taking `or` of those pixels and the eroded image in the `imgor` function, we can extract a part of skeltonized image. The eroding is repeated until all the white pixels are removed from the image by using `countWhite` function. Fig. 5.30 illustrates the process for a 3×3 image.

```
function skel = skeltonize(Igorig, rev)
% suppose we have black background and white lines
Ig = uint8(imbinarize(Igorig));
[r, c] = size(Ig);
%colormap(gray(2));
skel = logical(ones(r,c));
if (rev)
    img = Ig;
else
    img = not(Ig);
end
while logical(true)
    openimg = morph(img, 'erode');
    openimg = morph(openimg, 'dilate');
    openimg = logical(openimg);
    temp = imgsubtract(img, openimg);
    eroded = logical(morph(img, 'erode'));
    skel = imgor(eroded, temp);
    img = skel;
    if (countWhite(eroded, r, c) == 0)
            break;
    end
end
end
function c = countWhite(Ig, r, col)
c = 0;
for i=1:r
    for j = 1:col
        if Ig(i,j)== 1
            c = c + 1;
        end
    end
end
end
function R = imgor(a, b)
```

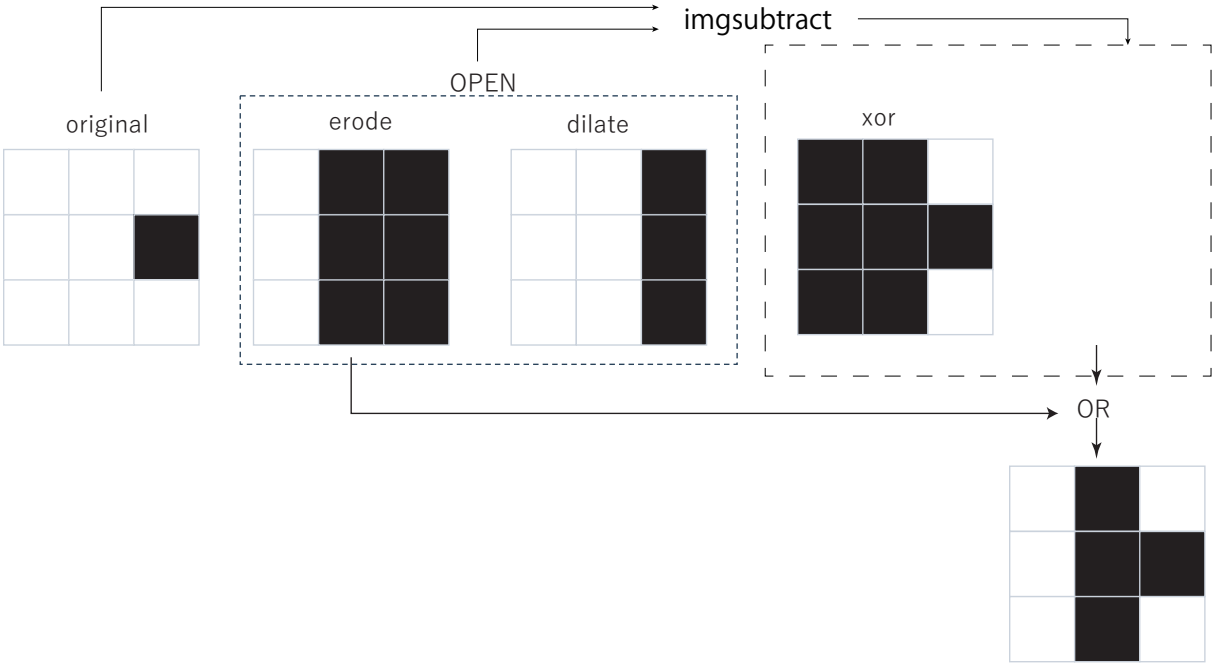

Figure 5.30: Illustration of image subtraction and skeltonization

Figure 5.31: Curves after skeltonizing

```
 [r,c] = size(a);
 R = zeros(r, c);
 for i=1:r
     for j=1:c
         R(i,j) = or(a(i,j), b(i,j));
     end
 end
end
function R = imgsubtract(a, b)
  [r, c] = size(a);
  [r1, c1] = size(b);
  if (r==r1 && c==c1)
      R = zeros(r, c);
      for i=1:r
          for j=1:c
              R(i,j) = xor(a(i,j), b(i,j));
          end
      end
  else
      fprintf("imgsubtract different dimensions %d %d %d %d\n", r,c,r1,c1);
  end
end
```

The output of this script is shown in Fig. 5.31.

Exercise 5.9 *(skeltest) In the previous example, we skeltonize white pixels in a black background. Compose a*

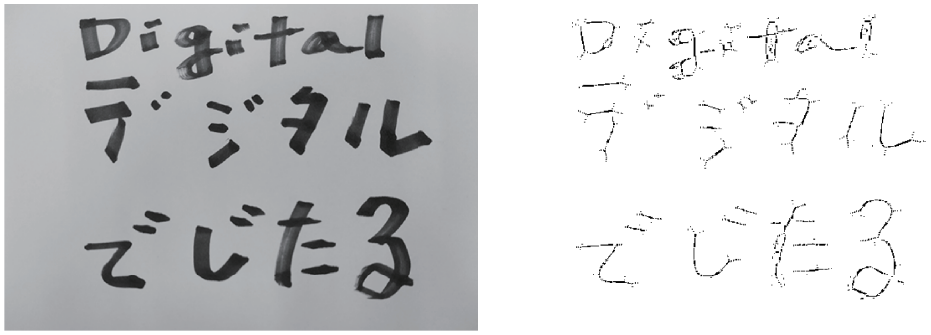

Figure 5.32: Skeltonize black curves in a white background example

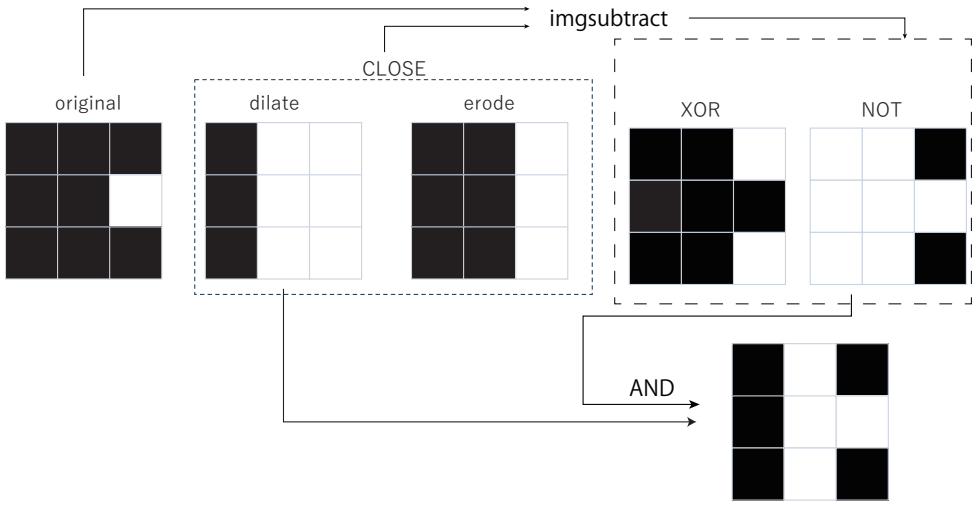

Figure 5.33: Reverse skeltonization bit operations

script which skeltonizes black pixels in a white background as in Fig. 5.32. For your reference, the reverse skeltonization demands pixel bit operation which is slightly different from the skeltonization of white curve on a black background as shown in Fig. 5.33. The point here is the merge of false logics.

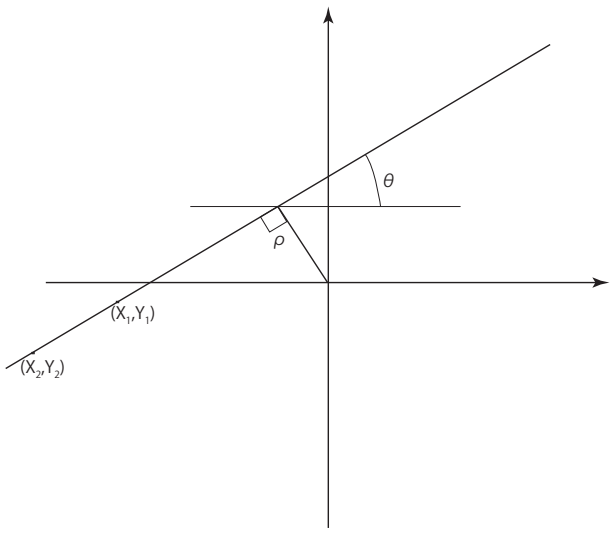

Figure 5.34: General description of a line

5.7 Line detection with Hough Transform

A line in x-y coordinate is usually represented by the following equation.

$$y = ax + b \tag{5.24}$$

But in this case, the lines $x = \text{constant}$ cannot be represented because $a = \infty$. The use of a and b is also inconvenient to detect lines by voting to a limited number of candidate lines by inspecting the image pixel by pixel. Instead of using the a, b representation, we use the inclination angle θ and the length from the origin ρ as the parameter to represent a line as in Fig.5.34. By doing this, both ρ and θ can be represented with a limited set of discrete values. This is the big advantage of using ρ and θ over the use of a and b in Eq. 5.24.

After applying an edge detection algorithm, we can extract feature points, including lines. Suppose pixels such as (X_1, Y_1) and (X_2, Y_2) are on a line, and we draw 180 lines centered at (X_i, Y_i) with angular resolution of 1 degree. The resolution 1 degree can be lower or higher but has to be a finite resolution. We can derive ρ corresponding to each θ. An arbitrary point on the line of θ inclination which passes (X_i, Y_i) can be represented with a parameter t such that

$$\left\{ \begin{array}{c} x \\ y \end{array} \right\} = \left\{ \begin{array}{c} X_i \\ Y_i \end{array} \right\} + t \left\{ \begin{array}{c} \cos\theta \\ \sin\theta \end{array} \right\}. \tag{5.25}$$

Eliminating t from the above equation such that

$$t = \frac{X_i + \sin\theta\rho}{\cos\theta} = \frac{Y_i - \cos\theta\rho}{\sin\theta} \tag{5.26}$$

it is apparent that (x, y) on a line with predefined set of ρ and θ shall satisfy the following relation.

$$Y_i \cos\theta = X_i \sin\theta + \rho \tag{5.27}$$

which relates to Eq.5.24 such that $a = \tan\theta$ and $b = \frac{1}{\cos\theta}\rho$.

The length L between the origin and a given set of X_i, Y_i, t and θ is given by the following equation.

$$L^2 = (X_i + t\cos\theta)^2 + (Y_i + t\sin\theta)^2 = X_i^2 + 2t(X_i\cos\theta + Y_i\sin\theta) + Y_i^2 + t^2 \tag{5.28}$$

ρ is the shortest of L, and therefore, can be given by taking the derivative of L in terms of t.

$$\frac{d(L^2)}{dt} = 2(X_i\cos\theta + Y_i\sin\theta) + 2t = 0 \tag{5.29}$$

Thus,

$$t = -(X_i \cos\theta + Y_i \sin\theta) \tag{5.30}$$

Substituting Eq.5.30 into Eq.5.28, ρ can be derived as follows.

$$\begin{aligned} \rho^2 &= X_i^2 - 2(X_i \cos\theta + Y_i \sin\theta)^2 + Y_i^2 + X_i^2 \cos^2\theta + 2X_iY_i \cos\theta \sin\theta + Y_i^2 \sin^2\theta \\ &= (Y_i \cos\theta - X_i \sin\theta)^2 \end{aligned} \tag{5.31}$$

Therefore ρ is given as follows.

$$\rho = |Y_i \cos\theta - X_i \sin\theta| \tag{5.32}$$

which matches ρ derived from Eq.5.27.

By changing θ with a predefined resolution, in our case 1 degree, we can derive a set of θ and ρ for a feature point. If a group of feature points form a line, the popularity of particular set of θ and ρ shall be relevant. This procedure is similar to a voting by feature points for line candidates. The length of ρ is also needed to be round down with a finite resolution.

5.7.1 Draw a line with given set of ρ and θ

Let us derive a function which draws white lines with a given set of ρ and θ on a black background. Because we need to apply an edge filter later on, we use grayscale colormap instead of the binary (white and black) colormap. Using Eq.5.27, the pixel on the given line can be discriminated. The following two functions, `createline` and `addline`, draws a line on new or given image. Note that the vertical and horizontal axes represent x and y axes, respectively.

Example 5.10 *(createline, addline)*

```
%
% create a line
function fig = createline(rho, theta, r, c)
row=r;
col=c;
fig = uint8(zeros(row,col));
for i=1:row
    for j=1:col
        if (ceil(j*cos(theta)) == ceil(i*sin(theta) + rho))
            fig(i,j) = 255;
        end
    end
end
end
%
% add line to an image
function fig = addline(f, rho, theta)
[row, col] = size(f);
fig = f;
for i=1:row
    for j=1:col
        if (ceil(j*cos(theta)) == ceil(i*sin(theta) + rho))
            fig(i,j) = 255;
        end
    end
end
end
```

Example 5.11 *(drawlines) The following m script produces an image with two lines defined by combination of ρ and θ. The produced image is shown in Fig.5.35*

```
f = createline(300, pi/8, 480, 640);
nf = f;
nf = addline(nf, 0, pi/3);
imshow(nf);
function fig = addline(f, rho, theta)
[row, col] = size(f);
fig = f;
```

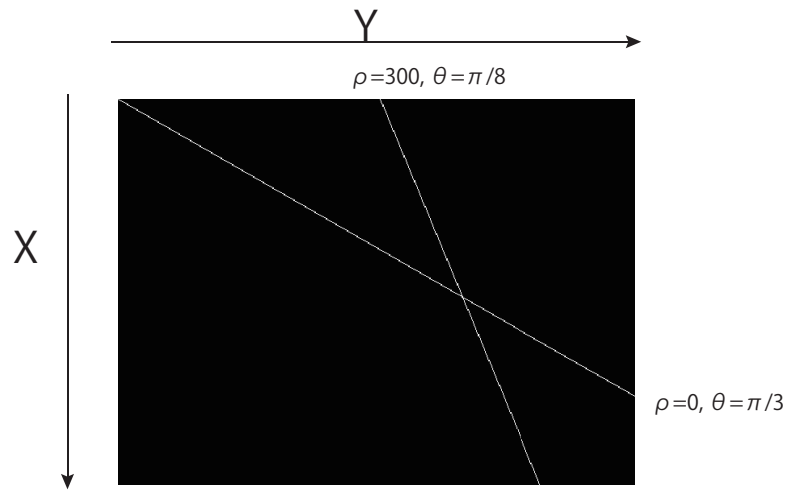

Figure 5.35: Draw two lines defined with combination of ρ and θ

```
for i=1:row
    for j=1:col
        if (ceil(j*cos(theta)) == ceil(i*sin(theta) + rho))
            fig(i,j) = 255;
        end
    end
end
end
function fig = createline(rho, theta, r, c)
row=r;
col=c;
fig = uint8(zeros(row,col));
for i=1:row
    for j=1:col
        if (ceil(j*cos(theta)) == ceil(i*sin(theta) + rho))
            fig(i,j) = 255;
        end
    end
end
end
```

5.7.2 Hough trasnformation to detect lines

Example 5.12 *(testcreatelinesimple) The following script produces an image with two lines, and detect them with Hough transformation. Because the features of original image are detected with an edge filter, one original line may produce more than one feature lines. Because of this, we draw top five entries detected. The original and detected lines are shown in Fig.5.36.*

```
f = createline(300, pi/8, 480, 640);
nf = f;
nf = addline(nf, 0, pi/3);
[row, col] = size(nf);
BW = edge(nf, 'canny');
subplot(1,2,1);
imshow(nf);
title('Original image');
[H,T,R] = houghline(BW);
maxh = max(H, [], 'all');
% scale the accumulator in 8bits range
Hr = uint8(ceil(H/maxh*255));
subplot(1,2,2);
% sort the accumulator in descending order
[rh, ch] = size(Hr);
```

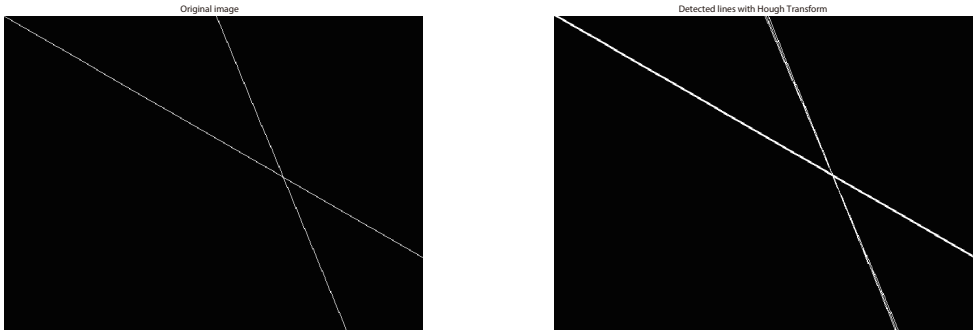

Figure 5.36: Line detection with Hough transform

```
oneHr = reshape(Hr, [1, rh*ch]);
sortOneHr = sort(oneHr, 'descend');
% the most frequent combination of rho and theta
[maxr, maxc] = find(Hr== sortOneHr(1));
subplot(1,2,2);
% maybe the same frequency is shared with multiple lines
nummax = size(maxr);
for kk=1:nummax
    fa = addline(f, maxr(kk), (maxc(kk))/180*pi);
end
%  second most frequent lines up to fifth.
for kk = 2:5
  [maxr, maxc] = find(Hr == sortOneHr(kk));
  nummax = size(maxr);
  for k=1:nummax
      fa = addline(fa, maxr(k), (maxc(k))/180*pi);
  end
end
imshow(fa);
title('Detected lines with Hough Transform');
function [h, ta, ra] = houghline(im)
[r,c] = size(im);
rho_m = ceil(sqrt(r^2+c^2));
t = 180;
ta = (1:1:t);
ra = (1:1:rho_m);
h = zeros(rho_m, t);
for i=1:r
    for j=1:c
          if (im(i,j) > 0)
            % accumulate the count for each lines from 1 to 179.
            for theta=1:t-1
               cs = cos(theta/180*pi);
               sn = sin(theta/180*pi);
               rho = floor(abs(j*cs-i*sn))+1;% avoid rho=0
               h(rho, theta) = h(rho, theta)+1;
            end
          end
    end
end
end
```

Exercise 5.10 *(testcreateline/lineHough) The above example fails to detect if we add a line with negative θ as shown in Fig.5.37. Revise the script to handle negative θ lines.*

5.8 Circle detection with Hough Transform

The principle of line detection can be directly extended to detect circles. Suppose we have a radius r circle whose center coordinate is X_c, Y_c as shown in Fig.5.38. For simplicity, it is assumed that the radius r of the circle is

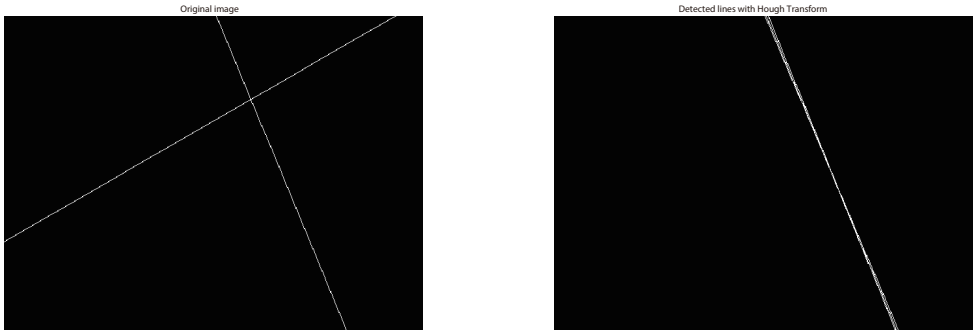

Figure 5.37: Line detection with Hough transform fails for negative θ line

known for now. In the figure, the candidate circles at an edge point x_e, y_e are also drawn.

The candidate circles can be represented mathematically as follows and we can accumulate the frequency of x, y by summing over all the edges.

$$(x - x_e)^2 + (y - y_e)^2 = r^2 \tag{5.33}$$

This way, the true center of the target circle should collect the maximum number of polls.

5.8.1 Draw a circle

The following two scripts create a black canvas with a specified size and add a circle by specifying its center and the radius, thickness of the circle and its color with RGB values.

Example 5.13 *(createbb, addcircle) The following script creates a black canvas with row, col size.*

```
function I = createbb(row, col)
I = uint8(zeros(row, col,3));
for i=1:row
    for j=1:col
        for k=1:3
           I(i,j,k) = 0;
        end
    end
end
```

The following script adds an circle whose center is x_c, y_c and radius dc with line width t with the colormap vector $kk = [RGB]$. The condition in the inner most loop enables to draw circles crossing the canvas boundary.

```
function iout = addcircle(iin, xc, yc, dc, t, kk)
[r, c, ~] = size(iin);
tt = ceil(t/2);
for j=dc-tt:dc+tt
      for angle=0:0.01:2*pi
          for k=1:3
            x = floor(xc+cos(angle)*j);
            y = floor(yc+sin(angle)*j);
            if ((x>0 && x<r) && (y>0 && y<c))
              iin(x,y,k) = kk(k);
            end
          end
      end
end
iout = iin;
```

For example, the following script first creates a canvas and draw a circle using `createbb` *and* `addcircle`.

```
img = createbb(480, 640);
rad = 80;
num = 4;
white = [255 255 255];
img = addcircle(img, 240, 22, rad, 1, white);
```

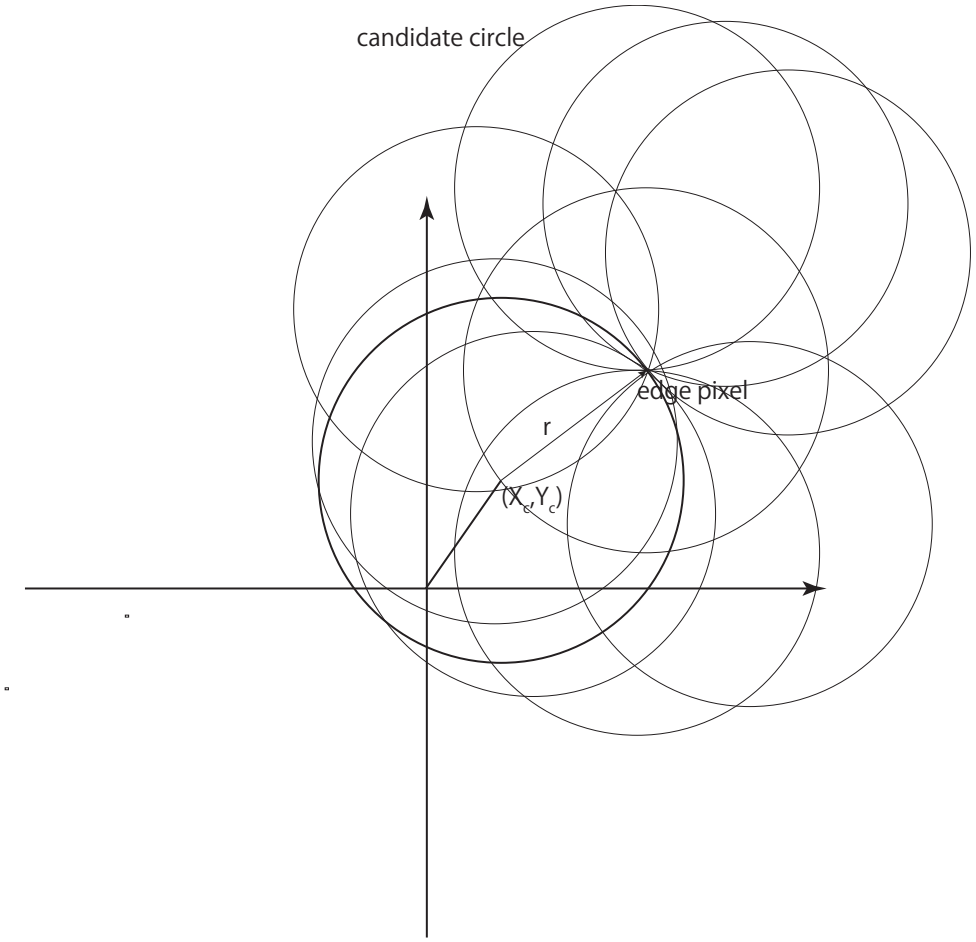

Figure 5.38: Detecting circle with Hough transform

5.8.2 Single circle detection with Hough transform

In order to accumulate the polls on candidate circles whose circumference is partly drawn in the canvas, the extended canvas is introduced as shown in Fig.5.39.

The target image is firstly converted to a grayscale image and its edge is detected in this case with canny filter. The candidate circles are generated by rotating the given radius circle for 360 degree with 1 degree resolution. A finer or coarser resolution may work though.

Example 5.14 *(findcirclesingle)*
The following script produce one circle and detect it using a Hough transform function `houghcirclesingle`. *The detected circle is overwritten to the original canvas with a green line.*

```
img = createbb(480, 640);
rad = 80;
white = [255 255 255];
img = addcircle(img, 240, 22, rad, 1, white);
[r,c] = houghcirclesingle(img, rad);
green = [0 255 0];
img = addcircle(img, r, c, rad, 1, green);
imshow(img);
function [row, col] = houghcirclesingle(im, rad)
[r,c,d] = size(im);
img = rgb2gray(im);
img = edge(img, 'canny');  % detect edges
```

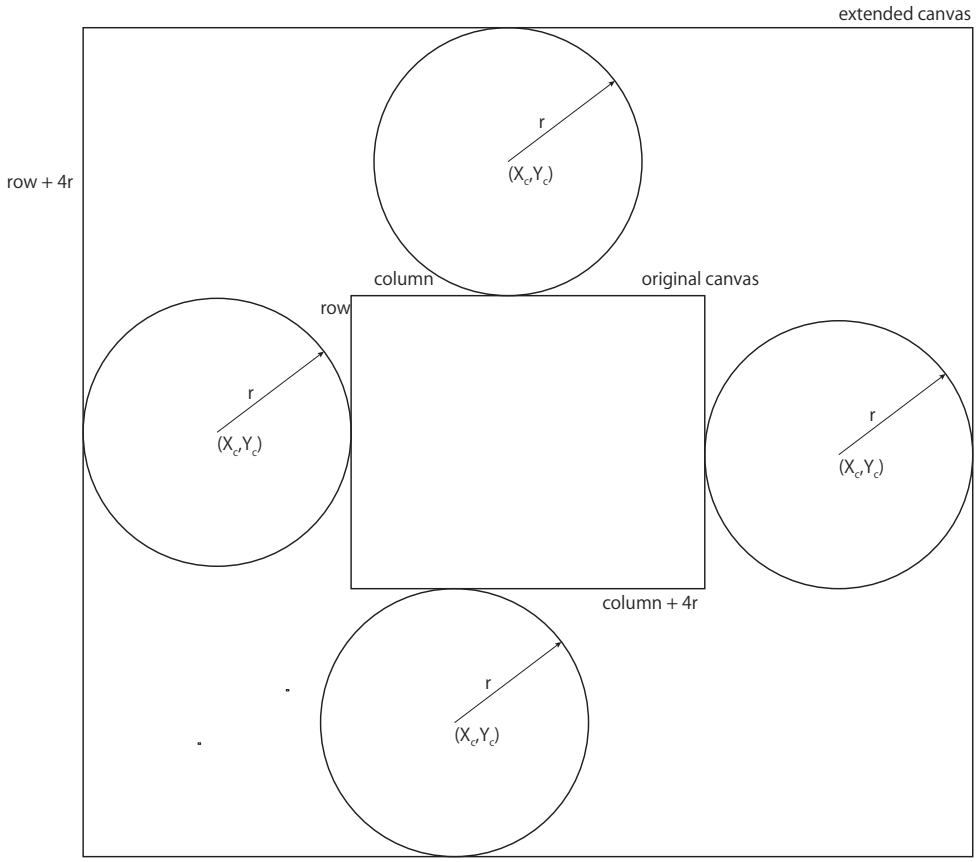

Figure 5.39: Extended canvas to poll for circle candidates crossing boundary

```
centers = uint16(zeros(r+4*rad, c+4*rad)); % possible center position
for i=1:r
    for j=1:c
          if (img(i,j) > 0)   % if current pixel is an edge
            for theta=0:359  % produce candidate circle and poll for the center.
               cs = rad*cos(theta/180*pi)+i;
               sn = rad*sin(theta/180*pi)+j;
               centers(floor(cs)+2*rad, floor(sn)+2*rad) = ...
                    centers(floor(cs)+2*rad, floor(sn)+2*rad)+1;% accumulator
            end
          end
    end
end
rc = reshape(centers, [1, (r+4*rad)*(c+4*rad)]);  % layout in one dimension
[v, i] = max(rc);  % extract first pixel value and its index
row = mod(i,r+4*rad) - 2*rad;  % convert the one dim to two dim index
col= floor(i/(r+4*rad)) - 2*rad;
end
```

The result is shown as Fig.5.40. It is shown that the circle, which crosses the canvas boundary can be successfully detected.

Exercise 5.11 *(findcircle) Revise the script to detect more than one circles in an image with Hough transform. Radius of all the circles can be assumed to be known.*

Figure 5.40: Result of Hough transform to find a single circle part of which is drawn in the canvas.

5.8.3 Variable radius circle detection with Hough transform

By extending the voting to varying radius, we can detect circles with unknown radius using Hough transform. There are several points to be considered.

- Don't try to start with too small radius such as 1 because radius 1 circles corresponding to one white pixels.
- Note that the `find` function is convenient to retrieve the original rows and columns of an array from its one dimensional representation to sort its entries. But if we have more than two dimensions in the array, like in the case of variable radius circle detection where the `H` matrix includes, radius, row and column, the second and third index are aggregated into a linear list. We need to extract the each entries.

```
% create a canvas
img = createbb(480, 640);
white = [255 255 255];
% draw three circles
for i=1:3
  rad = floor(100*rand);
  if rad < 20
      rad = 20;
  end
  img = addcircle(img, floor(480*rand), floor(640*rand), rad, 1, white);
end
%
%  find these circles with variable radius Hough transform
[rr, r,c, numr] = houghcirclev(img, 100, 20);
green = [0 255 0];
for i = 1:numr
  img = addcircle(img, r(i), c(i), rr(i), 1, green);
end
imshow(img);
% parameters are
% returns radius, center x and center y and the total number of detected circles
function [rr, row, col, toln] = houghcirclev(im, rad, num)
```

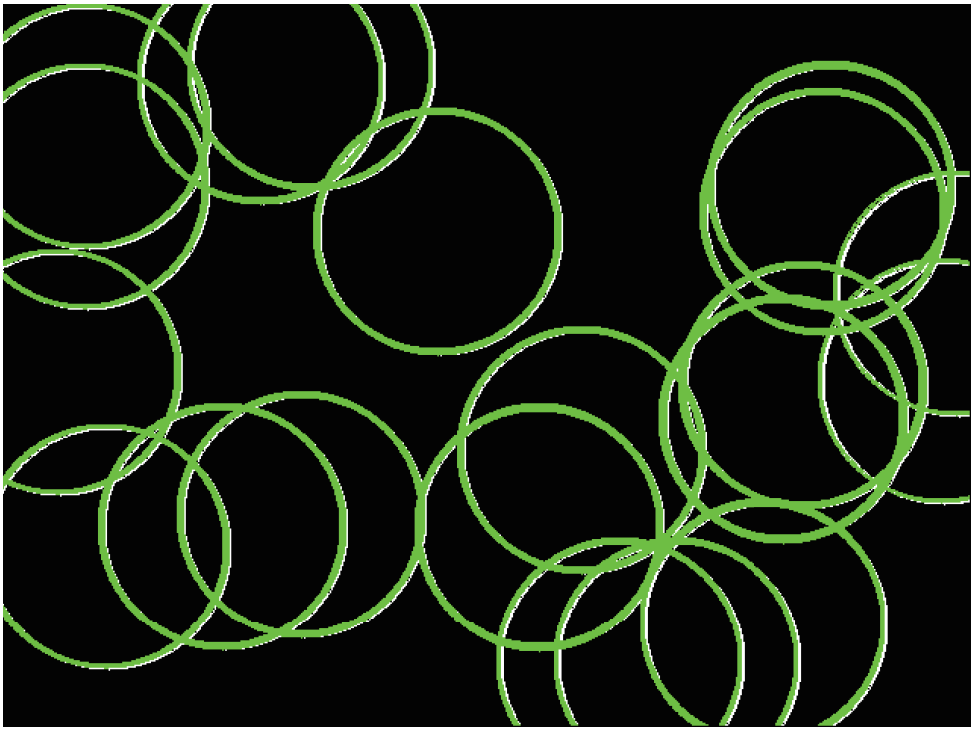

Figure 5.41: Detection of more than one circles in an image example

```
[r,c,~] = size(im);
centers = uint16(zeros(rad, r+2*rad, c+2*rad)); % possible center position
for i=1:r
    for j=1:c
          if (img(i,j) > 0)
            for theta=0:359
               for k=10:rad  % start from 10 to avoid tiny
                                % circles which could be one pixel.
                 cs = k*cos(theta/180*pi)+i;
                 sn = k*sin(theta/180*pi)+j;
                 centers(k, floor(cs)+rad, floor(sn)+rad) = ...
                      centers(k, floor(cs)+rad, floor(sn)+rad)+1;% accumulator
               end
            end
          end
    end
end
% convert to one dimensional array to sort the elements
rc = reshape(centers, [1, rad*(r+2*rad)*(c+2*rad)]);
sortOneHr = sort(rc, 'descend');
%maxv = max(Hr,[], 'all');
toln = 0;
rr = [];
row = [];
col = [];
for k=1:num % top num votes
    % the second return value of find is a linear list of
    % the rows and cols.
    [rrc, rc] = find(centers == sortOneHr(k));
    [numr, ~] = size(rrc);
    rr = [rr rrc'];
    toln = toln+numr;
    for i=1:numr
        rem= mod(rc(i)-1, 2*rad+r);
        q = (rc(i)-1 - rem)/(2*rad+r);
        col = [col q+1-rad];
        row = [row rem+1-rad];
```

```
    end
end
end
```

Exercise 5.12 *(findcoinv) Compose a m script which detect circle shapes in a natural figure. We can accept redundant detection as shown in Fig. 5.42. Layout the original natural image and the detected circles horizontally as in the example.*

Original natural image Detected circles

Figure 5.42: Detection of circles in a natural image example

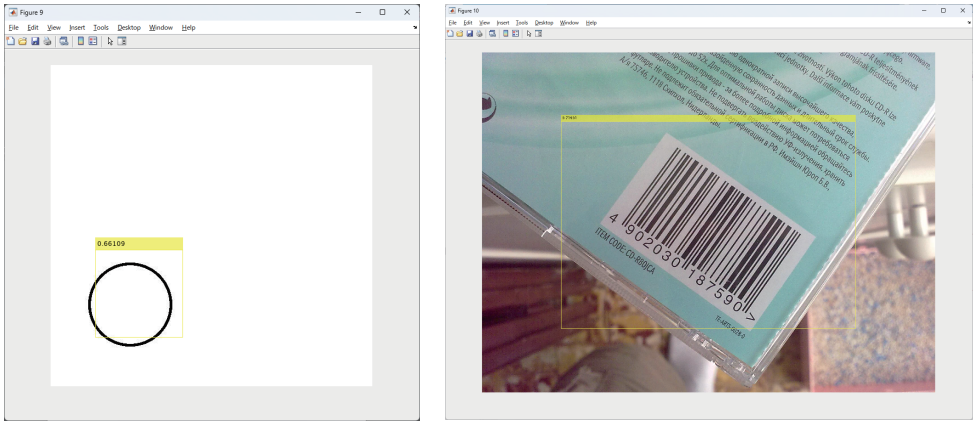

Figure 5.43: Object localization using bounding box. A circle detected from an image (left). Barcode detected from an image (right).

5.9 Object detection using Deep Learning

There is an emerging signal processing technique to detect objects from an image without using predefined object features as in Hough transform but by automatically extracting the feature of specific object with training data and ground truth. This is referred to as Object Detection in the machine learning field.

In this subsection, we experience this interesting field by running an Object Detection with existing Deep Learning Network with a modification to detect circles in a given image. `MATLAB` furnishes many machine learning functionalities. About Object Detection, `https://jp.mathworks.com/help/vision/ug/getting-started-with-object-detection-using-deep-learning.html?lang=en` is a good starting point.

There are fundamentally two tasks in Object Detection. One is to localize objects in an image and the other is the classification of the localized object. The localization is to detect portions of a given image, which contains a particular object. The localization is typically presented as a rectangular area referred to as a bounding box as shown in Fig. 5.43.

Among many Object Detection methods available in `MATLAB`, we examine `YOLO v2` to detect circles in an image because it can be swiftly trained with normal PC without GPU. `YOLO v2` features fast simultaneous object detection and classification exploiting Convolutional Neural Network (CNN). Higher version of `YOLO v4` is available in `MATLAB R2023b`.

In CNN, a given image is usually resized into a fixed size and fixed depth image, $128 \times 128 \times 3$ in MATLAB YOLO v2, and then iteratively convoluted by a given set of kernels to produce convoluted images. In `YOLO v2`, the number of kernels are 16 in the first stage as shown in Fig. 5.44.

Convoluted images are selectively activated by using an nonlinear activation function such as ReLU and sigmoid function and the size of images are reduced with Pooling function. This process is repeated for a predefined times to produce a set of feature maps, in the case of YOLO v2, $16 \times 16 \times 128$. The localization and classification are done by convoluted feature maps as shown in Fig. 5.45. Such overview of a neural network can be produced by `analyzeNetwork` command.

Example feature maps produced by CNN are shown in Fig. 5.46 with its original image. As shown in the figure, 25 feature maps roughly retain the original image because any convoluted image is produced by neighbor pixels.

Let us generate `YOLO v2` network to detect circles in an image. `MATLAB` provides an example YOLO v2 Object Detector for vehicles `https://www.mathworks.com/help/vision/ug/train-an-object-detector-using-you-only-look-once.html?s_tid=srchtitle_site_search_3_yolo%20object%20detection`.

The following toolboxes and a model network need to be installed from `Add-Ons` menu.

- Computer Vision Toolbox
- Deep Learning Toolbox

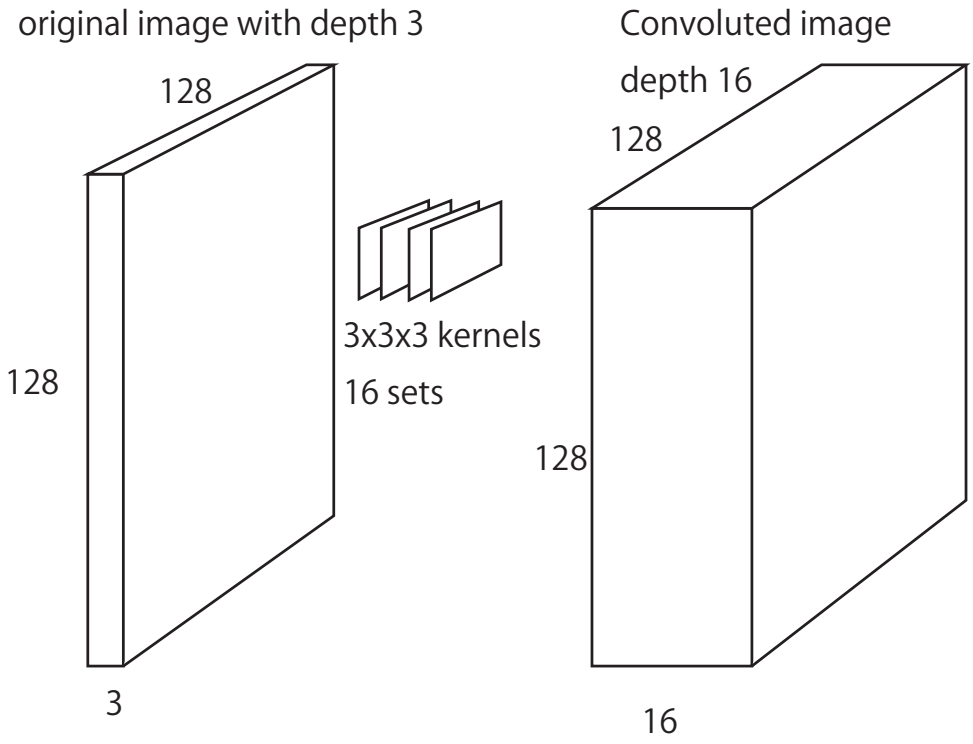

Figure 5.44: Image convolution with multiple kernels

- Deep Learning Toolbox Model for ResNet-50 Network
- Image Processing Toolbox

The example YOLO v2 Object Detector is trained to detect vehicles. We exploit the example to detect circles with new dataset. The new dataset is produced using `imageLabeler` application, which can be found under `Add Ons` tabs as Fig. 5.47.

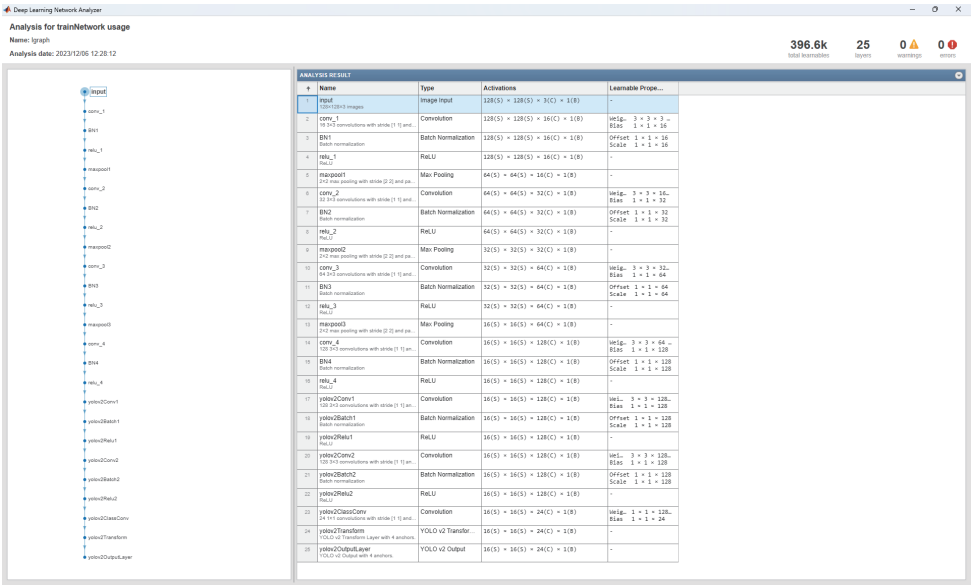

Figure 5.45: YOLO v2 network structure

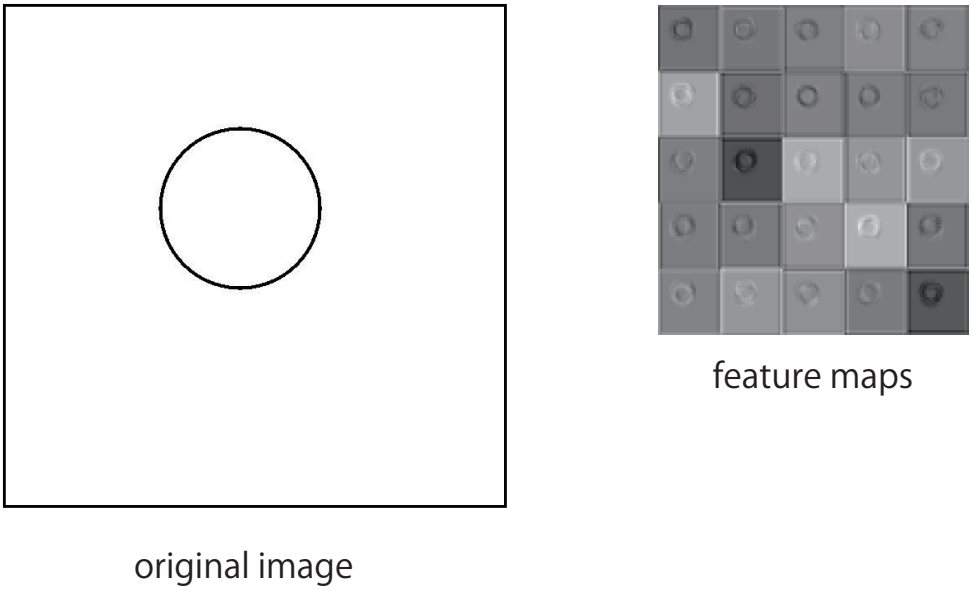

Figure 5.46: Feature map example produced from an original image

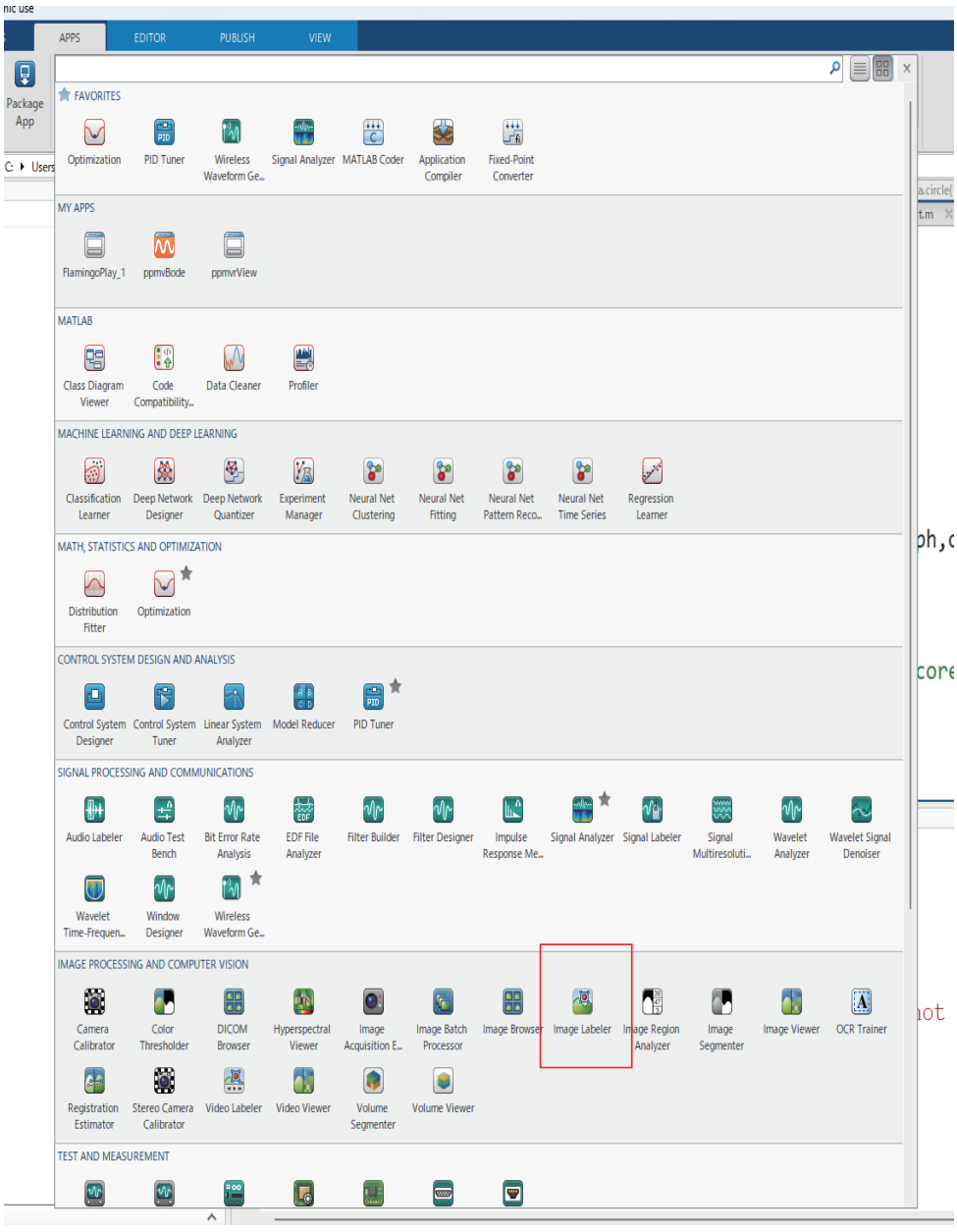

Figure 5.47: Training dataset can be conveniently produced with ImageLabeler application

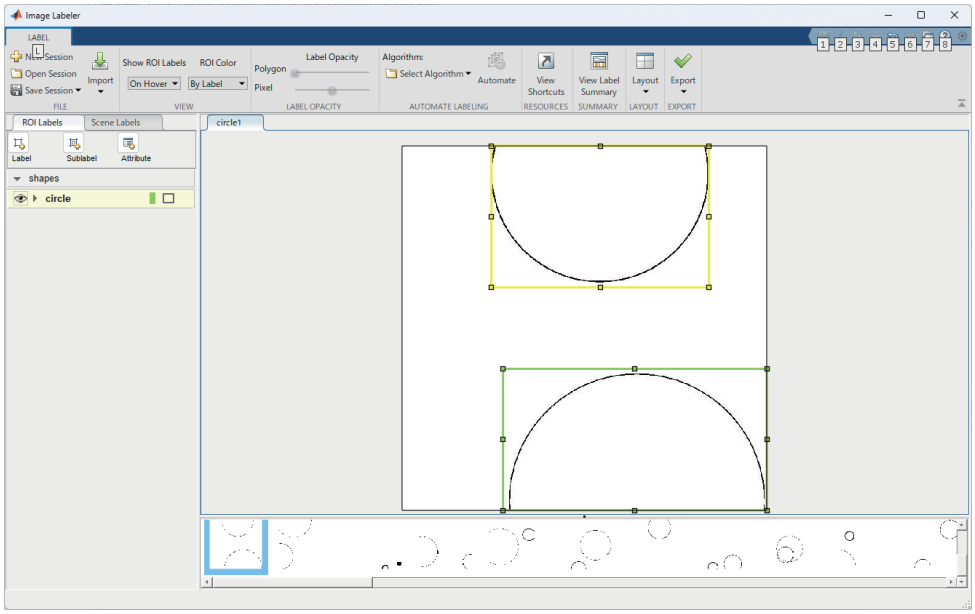

Figure 5.48: Annotating training images with ImageLabeler

5.9.1 Preparation of ground truth data with ImageLabeler

Starting `ImageLabeler` with a folder name, which contains training images such that

```
>> imageLabeler('circles')
```

produces an window as shown in Fig. 5.48 to define the ground truth locations of circles. The images of circles need to be produced before the usage of `ImageLabeler'`. In this example, every image contains randomly generated two circles. Ground truth data can be added by manually drawing a rectangular around each circle. The default name of the ground truth data is `gTruth`, which is changed to `circleTruth` in this example.

The ground truth data can be copied into the workspace with the `Export` button.

The ground truth data comprises `DataSource`, `LabelDefinition`, `LabelData`. `DataSource` points to the training data set, `LabelDefinition` defines the candidate classes, in this example, we only have `circle` label. `LabelData` defines the ground truth bounding box, which is given as a `table`. A `table` in `MATLAB` can store heterogeneous (mixed) data types. `cell` is another data structure in `MATLAB` to store heterogeneous data types, but we can only access an element in `cell` by its row-column index. On the other hand, we can use metadata to refer elements in a `table`.

As an image may include more than one circle, an entry of `LabelData` could be multiple rows cell, which can be produced by the following m script.

```
labels =  table('Size', [0, 1], 'VariableNames',{'circle'}, 'VariableType', {'table'});
%labels.Properties.VariableNames={'circle'};
for kk= 1: numData
    clf;
    I = initI(I, row,col);
    cnum = randi(numCircle);
for k=1:cnum
  d  = 200;
  fil=0; % fit in the region
  dc = randi(d)+10;% minimum 2 pixels
  xc = randi(row);
  yc = randi(col);
  fprintf("(%d,%d) %d\n",xc, yc, dc);
  for i=1:row
      for j=1:col
```

```
          if abs(sqrt((i-xc)*(i-xc) + (j-yc)*(j-yc) - dc*dc))<20
              I(i,j,:) = 0;
          end
      end
  end
  xxmin = max(xc-dc-10, 1);
  xxmax = min(xc+dc+10, row);
  yymin = max(yc -dc-10, 1);
  yymax = min(yc+dc+10, col);
  tt  =[yymin, xxmin, (yymax-yymin),  (xxmax-xxmin)];
  if (k==1)
      celldata = tt;
  else
      celldata = [celldata; tt]; % row appended to an array.
  end
end
celldata = {celldata};  % convert an array to cell.
ttl =table(celldata, 'VariableNames',{'circle'});
if (kk == 1)
  labels = ttl ;
else
  labels = [labels;ttl]; % table entry
end
imwrite(I, strcat(strcat('circles/circle',num2str(kk,"%03d")),'.jpg'));
end
```

5.9.2 Training YOLO v2 network with ground truth data

The following m script produces a CNN which detect circles in a given image. As the original network detects cars (vehicles) rather than circles. We exploit the original network as the initial value to train the network to detect circles. This is referred to as transfer learning. Transfer learning is in general faster than CNN training from random variables.

Example 5.15 *(yolotest)*

```
%
%  retrieve the ground truth data
[imds,bxds] = objectDetectorTrainingData(circleTruth);
%
%  load YOLO v2 network
vehicleDetector = load('yolov2VehicleDetector.mat');
lgraph = vehicleDetector.lgraph;
%
%  image data and lable data combined
cds = combine(imds, bxds);
options = trainingOptions('sgdm', ...
       'InitialLearnRate', 0.001, ...
       'Verbose',true, ...
       'MiniBatchSize',16, ...
       'MaxEpochs',30, ...
       'Shuffle','every-epoch', ...
       'VerboseFrequency',10);
% train YOLOv2 network with the trainData
[detector,info] = trainYOLOv2ObjectDetector(cds,lgraph,options);
```

We can test the object detector by giving an image including circles.

```
[imfile, pathname] = uigetfile('*.jpg');
imfile = strcat(pathname,imfile);
Iraw = imread(imfile);
I = imread(imfile);
%
% detect circles and produce bounding boxes
[bboxes,scores] = detect(detector,I);
if(~isempty(bboxes))
  Iraw = insertObjectAnnotation(Iraw,'rectangle',bboxes,scores);
  figure
  imshow(Iraw)
end
```

Circles detected with the trained `YOLO v2` network are as in Fig 5.49.

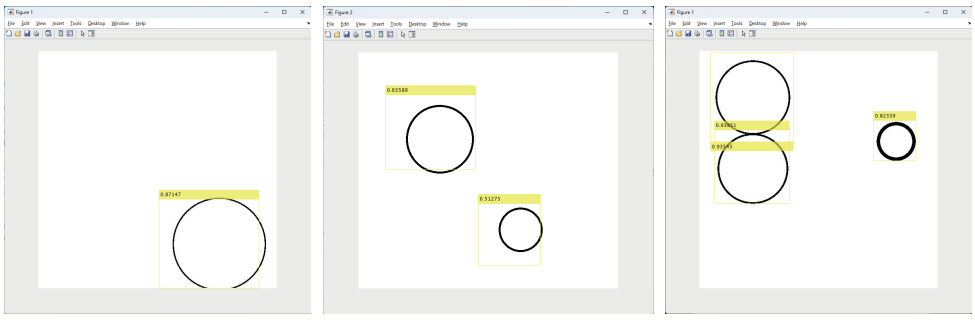

Figure 5.49: Circle detection using transfer learning of YOLO v2

On the other hand, there are still failed inference as in Fig. 5.50 as we provided only single circle training data.

As the training uses stochastic steepest gradient method (SGDM) with small updates, the training demands many iterations. When we use 500 training data with 30 generation of model updates, it took about about 5.5 minutes. The convergence of SGDM is shown in Fig. 5.51.

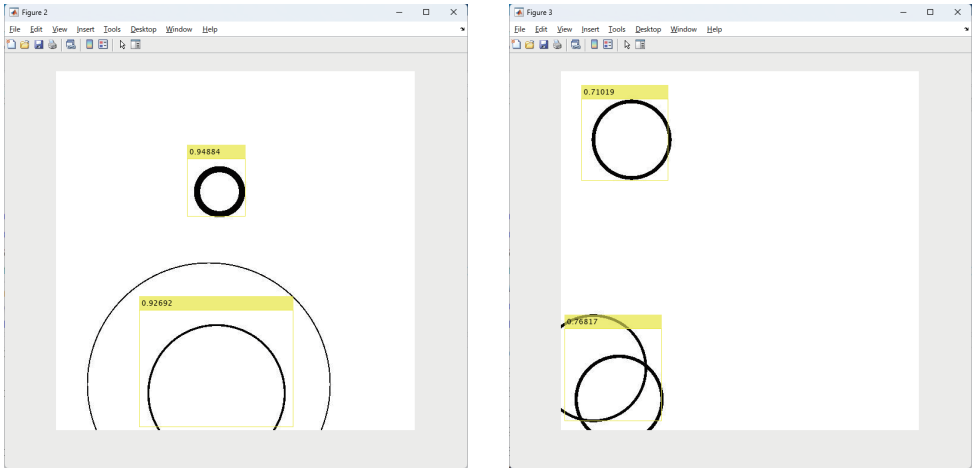

Figure 5.50: Failed circle detection with YOLO v2

However, once we establish a network, the inference is very fast, the neural net established with 500 training data can produce an inference with 0.29 sec (Fig. 5.52). This significant inference speed is another fascinating advantage of deep learning.

Exercise 5.13 *(circle generation and detection) Train YOLO v2 network with a set of circle images and produce an Object Detector of circles from an image file of size* $(600 \times 600 \times 3)$*. Upload the detector as a* `*.mat` *file with variable name* `detector`*. An example set of 50 circle images is available on* `SOL`*.*

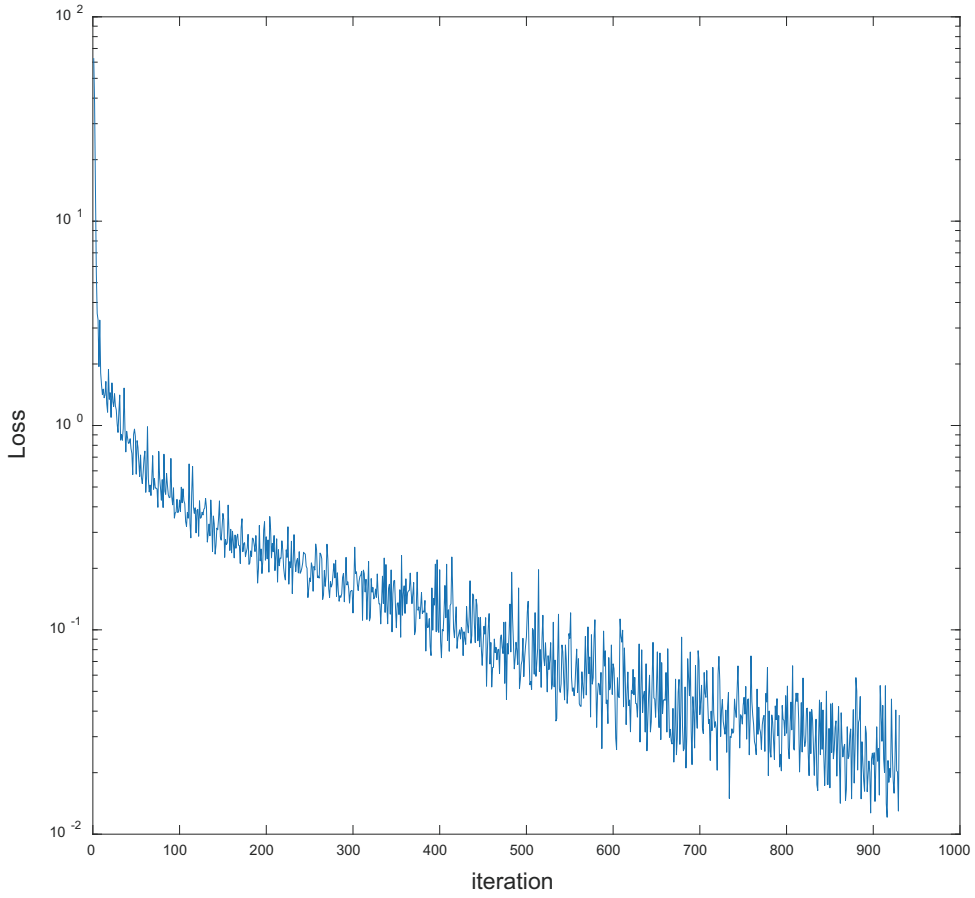

Figure 5.51: Convergence of SGDM with Yolo v2

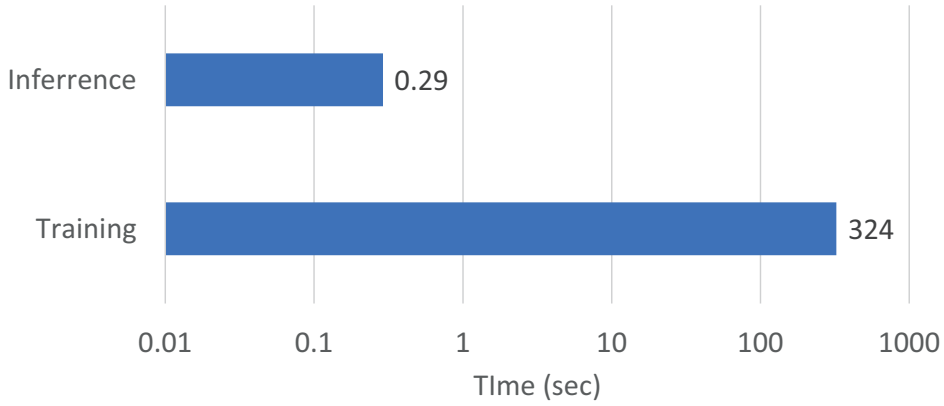

Figure 5.52: A comparison of training and inference with Yolo v2 for circle detection

6 Analog Communications

Digital signal processing enables us to send and receive data wirelessly without deep knowledge of electrical circuit. Wireless media may be audio, supersonic or radio. Audio is the air pressure vibration within human perception, usually less than 20 kHz. Air vibration above 20 kHz are referred to as ultrasonic. Radio propagates with electric and magnetic fields which are closely coupled. Radio wave is defined as less than 3000 GHz frequency in the international radio regulation `https://www.itu.int/pub/R-REG-RR`. Above the frequency, the electrical field still propagate but they are usually referred to as light, instead. In this course, we use either sonic(audio) or ultrasonic communication channels for both transmission and receiving. We examine the FM radio, a radio wave, for receiving in this section.

In wireless communication, either analog or digital data (baseband signal) is superposed on a dedicated frequency channel, which is referred to as a carrier, and is transferred to the corresponding receiver. This principle is the same and the base of sonic, ultrasonic, radio and light communications. Superposing baseband signal onto a carrier is referred to as modulation, and the extraction of baseband signal from the received superposed carrier is demodulation (Fig.6.1). By using modulation and demodulation we can simultaneously send multiple baseband signals by using different carriers. In earlier classes, we learn how to transmit 440 Hz and 880 Hz tones simultaneously. We can superpose two sets of baseband signal independently onto the two carriers, 440Hz and 880Hz. In the recipient's side, we tune either 440 Hz or 880 Hz to selectively obtain the target baseband signal. Different carriers provide independent communication channels. This is the mechanism of radio and TV broadcasting.

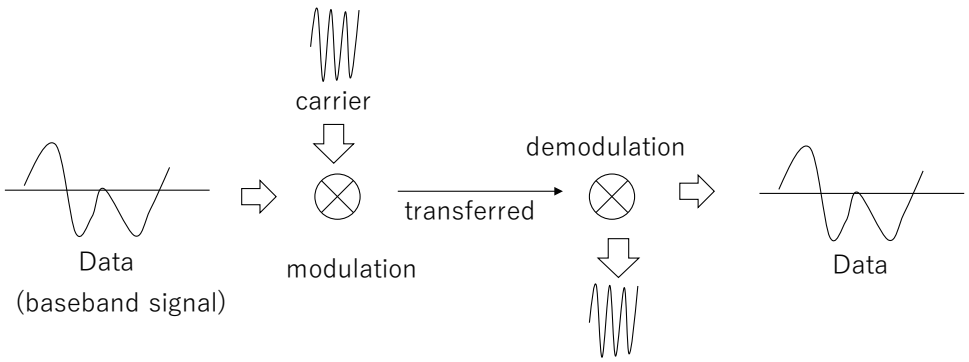

Figure 6.1: Modulation and demodulation of data

6.1 Types of modulation

There are three basic types of analog modulation, Amplitude modulation (AM), Frequency modulation (FM) and Phase modulation (PM). Suppose our carrier frequency is f_c and its amplitude is 1. The time domain carrier signal $y(t)$ with no phase offset can be written as follows.

$$y_c(t) = \sin(2\pi f_c t) \tag{6.1}$$

Let us express a carrier by a sine or cosine wave without any phase offset for simplicity as follows.

$$y_c(t) = \sin(2\pi f_c t + \phi) \tag{6.2}$$

The baseband signal is denoted by $x(t)$.

In AM, $x(t)$ may be added an offset x_m to enforce its positiveness such that

$$x(t)' = x(t) + x_m \geq 0. \tag{6.3}$$

Unless the bias x_m is added, the AM signal wave shape looks like in Fig.6.2. Since the carrier vibrates in between positive and negative values, two signals drawn in the blue and red lines (envelops) result in the same modulated

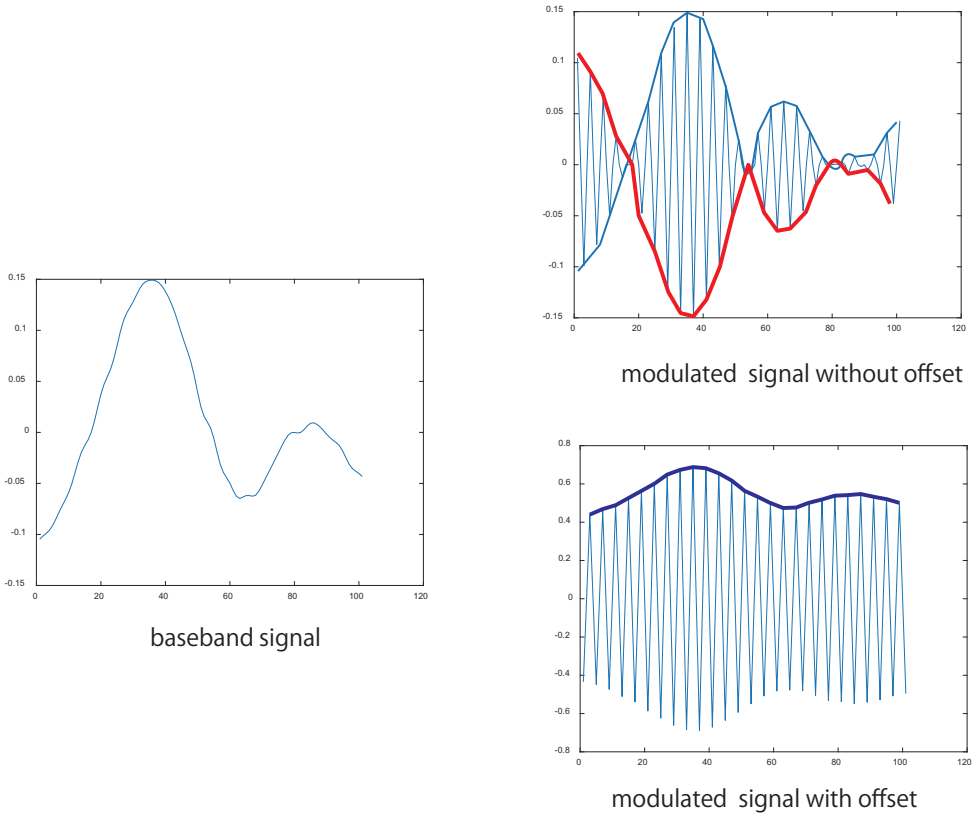

Figure 6.2: AM modulated signal without offset

signal. However, if we confine the signal to be only positive by adding the offset x_m, we can avoid this ambiguity as shown in the figure.

AM changes the amplitude of carrier according to the baseband signal with an offset $x'(t)$.

$$y(t) = x'(t) \sin 2\pi f_c t \tag{6.4}$$

PM changes the carrier phase according to the baseband signal $x(t)$. The baseband signal $x(t)$ is converted to phase deviation with a modulation index m_p such that

$$y(t) = \sin(2\pi f_c t + \phi) = \sin(2\pi f_c t + m_p x(t)) \tag{6.5}$$

FM changes the carrier frequency according to the baseband signal $x(t)$. The baseband signal $x(t)$ is converted to frequency deviation through a modulation index m'_f such that

$$y(t) = \sin(2\pi(f_c + m'_f x(t))t) = \sin(2\pi f_c t + 2\pi m'_f x(t)t) \tag{6.6}$$

By rewriting $x(t)t$ as $z(t)$ and $m_f = 2\pi m'_f$, the FM signal can be represented by

$$y(t) = \sin(2\pi f_c t + m_f z(t)) \tag{6.7}$$

Since the time derivative of phase is the angular velocity, the instantaneous angular velocity of the PM signal is given by the following equation.

$$\frac{d}{dt}(2\pi f_c t + m_p x(t)) = 2\pi f_c + m_p \frac{dx(t)}{dt} \tag{6.8}$$

The second term can be recognized as the baseband signal of FM, therefore,

$$m_f z(t) = m_p \frac{dx}{dt}. \tag{6.9}$$

This reveals that FM signal is proportional to the time derivative of PM signal.

These are the case where we have analog signal as the baseband signal. In the case of digital modulation, there is not much difference except we have only binary baseband signal (zero or one). Thus, we use terms Amplitude Shift Keying (ASK), Phase Shift Keying (PSK) and Frequency Shift Keying (FSK) in digital communications. Keying is a synonym of coding and encoding.

6.2 Amplitude Modulation and Demodulation

6.2.1 Modulation

Example 6.1 *(modulate5k440) Suppose our baseband signal is 440 Hz tone. Let us produce a modulated signal whose carrier frequency is 5000 Hz with AM. Since 20000 Hz sampling frequency is higher than the Nyquist frequency of the 5000 Hz AM signal, it is referred to as oversampling. 4 times or 8 times oversampling are common to stabilize the signal processing*[4]*. The following scripts produces a modulated signal and writes to an audio file* `test.wav`.

```
%%  modulate5k440.m
%    a 440 Hz tone is modulated
%    onto 5kHz carrier
Fs = 20000; % sampling rate
fx = 440; % baseband signal freq.
Fc = 5000; %carrier frequency
Len = 2; % 2 seconds duration
t = (0:1/Fs:Len-1/Fs); % from 0 to 2 second
x = (1+sin(2*pi*fx*t)); %baseband signal (positive)
f = (1/Len:1/Len:Fs); % frquency components
y = x .* cos(2*pi*Fc*t); %superpose (modulation)
sound(y, Fs);
plot(f, abs(fft(y)));
figure;
plot(t(1:100), y(1:100));
audiowrite('test.wav', y, Fs); %save the sound file
```

[4]Oversampling spreads the noise into the wider frequency region,which results in improved SNR

You can easily see that AM is fundamentally a dot multiplication of the baseband and carrier signals. You can use `ammod(y, Fc, Fs)` *facility of* MATLAB *to modulate baseband signal with carrier frequency F_c and the sampling rate F_s. Note that if you use* `ammod` *for modulation, you'd better use corresponding demodulator* `amdemod`.

The obtained time and frequency domain signals are shown in Fig.6.3. Since we simply multiply the baseband signal to the carrier, we produce 5440 Hz and 4560 (5000 - 440) Hz components. This frequency layout can be derived from the additive law of trigonometric functions.

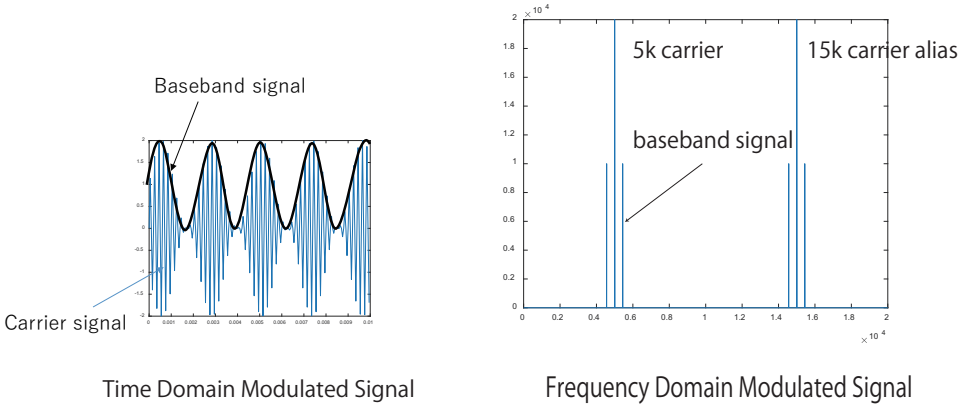

Figure 6.3: AM modulated signals

In the above example, we produce a baseband signal and a carrier simultaneously. But usually the baseband signal is produced with its own Nyquist frequency regardless the choice of carrier frequency. As the sampling rate of the baseband signal usually differs from that of carrier frequency, much lower, we need to adjust the sampling rate to multiply these two signals. The change of sampling rate is referred to as resampling.

For example, in the above example, the Nyquist frequency of 440 Hz baseband signal is 880 Hz. If we oversample by 4, the sampling frequency is 3520 Hz. Since the carrier frequency is 5000 Hz, the sampling rate of the carrier frequency is at least 10,000 Hz. We need to resample the baseband signal to produce a modulated signal.

Resampling is applied by specifying a fraction describing the ratio before and after the resampling. The principle is least common multiple. For example, if we have a baseband signal $x(t)$ with sampling frequency F_b while we have a difference sampling frequency F_s for the carrier, the baseband signals should be resampled with $newx = resample(x, n_c, n_b)$ where n_b and n_c are the minimum integer to describe the fraction $n_c/n_b = F_c/F_b$. For example, if $F_s = 20000$ and $F_b = 4000$, the baseband signal can be resampled with

$$newx(t) = resample(x, n_c, n_b); \tag{6.10}$$

The n_b, n_c can be derived with `rat` function by giving the fraction. For example,

```
 [n,d] = rat(4000/20000);
fprintf('n=%d d=%d\n', n, d);
>n=1 d=5
```

Resampling produces new sampling points by interpolating the original sampling points as shown in Fig.6.4.

Exercise 6.1 *(modulatevoice) We produce a voice message file in Example 2.4 by recording with embedded microphone. Compose a* `m` *script to modulate the voice as the baseband signal to 5 kHz carrier and confirm the envelop of the modulated carrier looks similar to the baseband signal as shown in Fig.6.5.*

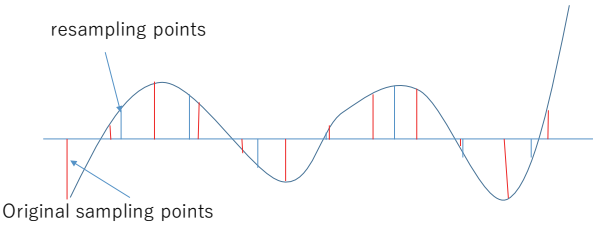

Figure 6.4: Adjust the sampling rate with resample

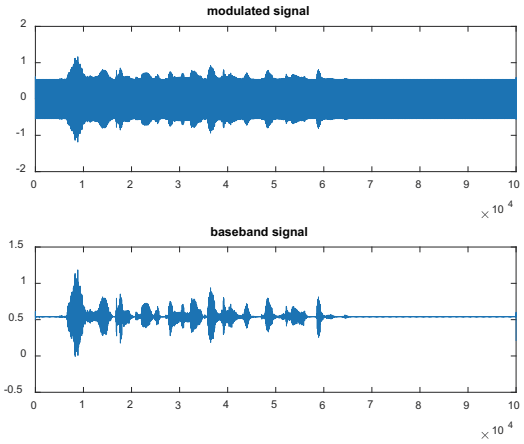

Figure 6.5: Modulated voice message

6.2.2 Demodulation

Let us retrieve the audio data and demodulate to recover the original sound. In order to extract x from amplitude modulated signal $(1+x)\sin 2\pi f_c t$, we multiply the carrier frequency to obtain

$$\sin 2\pi f_c t(1+x)\sin 2\pi f_c = -(1+x)\frac{1}{2}(cos(4\pi f_c t)-1) = \frac{1}{2}(1+x) - \frac{1}{2}\cos 4\pi f_c t \tag{6.11}$$

by the additive law of trigonometric function. As it is shown in the above equation, the high frequency components $cos(4\pi f_c t)$ should be suppressed with proper low pass filtering. We use a 20 tap FIR filter to suppress the high frequency component. The cut off frequency is set to be the carrier frequency in the following script.

Example 6.2 *(voicedemod)*

The following script first retrieves an audio data and demodulate by multiplying the carrier.

```
%%  amdemodulation
%
Fs = 20000; %audio out sampling
Fc = 5000;   %carrier freqency
[yy, Fss] = audioread('voice5k20k.wav');
% sound(yy, Fss);  % playback the modulated signal
[n,d] = rat(Fss/Fs);
[r,c] = size(yy);
t = (0:1/Fss:r/Fss-1/Fss);
f = (Fss/r:Fss/r:Fss);
yc = cos(2*pi*Fc*t);
z2c = yy' .* yc; %demodulation
Rp  = 0.00057565; % Corresponds to 0.01 dB peak-to-peak ripple
Rst = 1e-6;       % Corresponds to 80 dB stopband attenuation
eqnum = firceqrip(20,Fc/(Fss/2),[Rp Rst],'passedge');
fvtool(eqnum,'Fs',Fss,'Color','White') % Visualize filter
```

```
lowpassFIR = dsp.FIRFilter('Numerator', eqnum); %apply LPF
z = lowpassFIR(z2c');
fz = resample(z, n, d); %change the sampling frequency
sound(fz, Fs);
```

The spectrum before and after the LPF is compared in Fig.6.6.

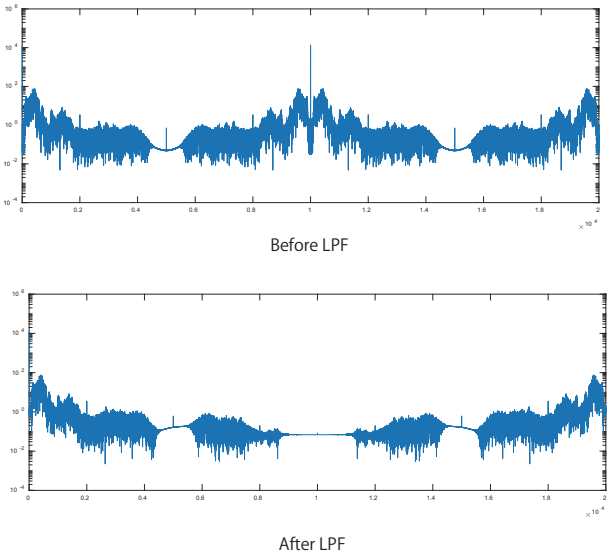

Figure 6.6: LPF after multiplication is essential in demodulation

MATLAB tips: Suppress audio overlap

In the script above, `sound(yy, Fss)` is commented out. If we uncomment the line, the playback of the modulated signal and the playback the demodulated signal by the last line `sound(fz, Fs)` overlaps. This is because with `sound` function, `MATLAB` simply send the signal to audio device and proceeds to the next instruction without checking if the audio device is playing or not. To suppress this overlap, the following works.

```
p = audioplayer(yy, Fss); % use audioplayer instead of sound
while p.isplaying      % wait the audioplayer to complete its current task.
end
```

Example 6.3 *(voiceiq) If you change* `yc = cos(2*pi*Fc*t);` *to* `yc = sin(2*pi*Fc*t);`*, there is no sound produced. How do you demodulate signal without knowing if it is modulated with a sine or cosine carrier or even intermediate of sine and cosine. This is practically important because the carrier phase at the receiver depends on the original carrier and also the distance between the transmitter and the receiver.*

The example below solves this phase ambiguity problem using both cosine and sine at the receiver. The following example assumes there is $\frac{\pi}{6}$ *phase offset between the transmitter and the receiver. After applying a LPF of the downconverted signal, the baseband component is modulated only with the baseband signal but signal comprises the real and the imaginary signals as in Fig. 6.7.*

To take both the real and imaginary component of the baseband signal, the absolute value $fz = resample(abs(z), d, n);$ *is usually used as in the following example.*

```
%%  amdemodulation
%
Fss = 68000;
Fs = 20000;
Fc = 20000;
```

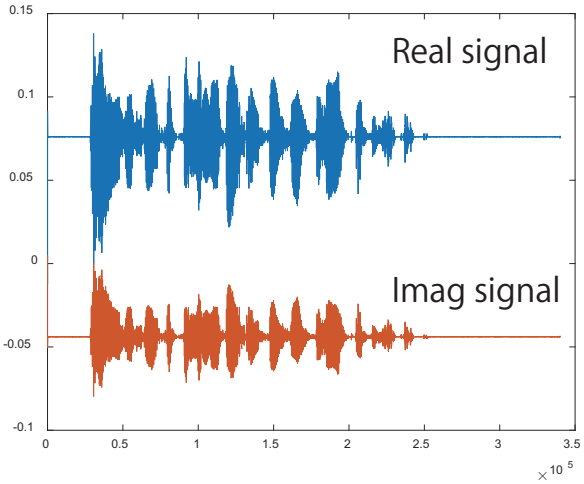

Figure 6.7: LPF after downconversion of a AM signal

```
filename = sprintf('voice%dk%dk.wav',Fc/1000,Fss/1000);
[yy, Fss] = audioread(filename);
p = audioplayer(yy, Fss);
while p.isplaying
end
[n,d] = rat(Fss/Fs);
[r,c] = size(yy);
t = (0:1/Fss:r/Fss-1/Fss);
f = (0:Fss/r:Fss-Fss/r);
semilogy(f/1000, abs(fft(yy)/r));
xlabel('Frequency (kHz)');
zz = complex(yy, zeros(r,1));
yc = complex(cos(2*pi*Fc*t+pi/6)', -sin(2*pi*Fc*t+pi/6)');
z2c = yy .* yc;   % down conversion
Rp  = 0.00057565; % Corresponds to 0.01 dB peak-to-peak ripple
Rst = 1e-6;       % Corresponds to 80 dB stopband attenuation
eqnum = firceqrip(100,Fc/(Fss/2),[Rp Rst],'passedge');
lowpassFIR = dsp.FIRFilter('Numerator', eqnum);
z = lowpassFIR(z2c);
fz = resample(abs(z), d, n);
sound(fz, Fs);
```

A convenient feature of AM is its robustness to the deviation of the transmit and receiving carrier frequency. It is usually difficult for a receiver to identify exactly the carrier frequency of the transmitter because of the imperfect and independent clocks in each of the endpoints. Fig. 6.8 shows the received AM signals subjected to four cases — no carrier error to 1 % carrier frequency error. The demodulated signals, in all of the case, do not degraded even in the cases of frequency error.

In the above example and exercise, the carrier frequency 5000 Hz is used but it can be any frequency. Even non-audible sound (ultrasonic) can be used as a carrier. The perception of high frequency tone degraded as we get older. This is the famous mosquito tone testing.

Exercise 6.2 *(mosquito) Compose a* `m` *script to check how high frequency you can pick up by changing frequency. In order to prompt user for input, you can use* `input` *function. The following captures a character from the keyboard after urging the user to type in.*

```
prompt = 'Input your answer [y/n]?';
xinp = input(prompt, 's');
```

By using frequency above 18 kHz, the carrier sound cannot be picked up by most of us. If you create a sound file with a voice message modulated onto 20 kHz or above frequency with `audiowrite` named `test.wav` and

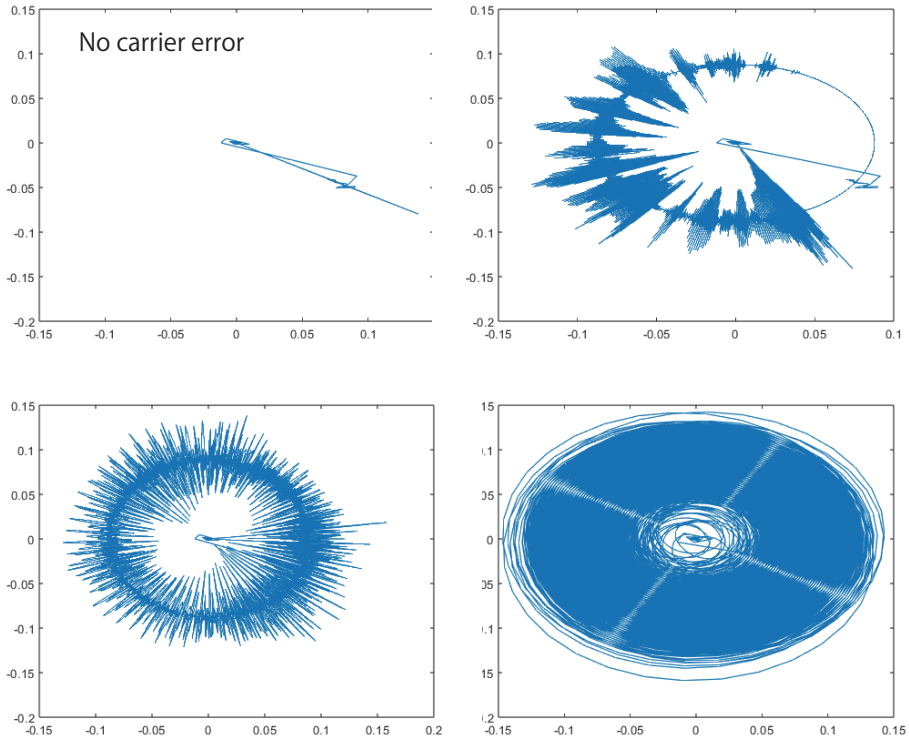

Figure 6.8: IQ constellation of AM received signal with various carrier frequency errors.

play with MATLAB or other music player, we can transmit a message without being recognized. But we need to have a good microphone to pickup the carrier as most of the microphone suppress the frequency component above 20 kHz.

Exercise 6.3 *(voicedemod20k) Compose a pair of modulator and demodulator to handle ultrasonic modulated signals (the carrier frequency 20 kHz) by modifying the previous example.*

6.3 Frequency Modulation and Demodulation

In actual implementation of frequency modulation (and phase modulation too), a single trigonometric function carrier is inconvenient because it periodically changes the amplitude. Particularly when the carrier amplitude is close to zero, the frequency modulated signal should suffers from the environment radio noise. To avoid this problem, we combine cosine and sine to compensate each other such that

$$y_c(t) = \cos(2\pi F_c t) + \sin(2\pi F_c t) \tag{6.12}$$

where F_c denotes the carrier frequency. This way, the amplitude of carrier is constant because

$$|y_c| = \sqrt{\cos^2(2\pi F_c t) + \sin^2(2\pi F_c t)} = 1. \tag{6.13}$$

In order to handle cosine and sine signals separately, we usually use a complex representation by putting the imaginary unit j to the sine component. We will follow this convention in the subsequent discussion.

$$y_c(t) = \cos(2\pi F_c t) + j\sin(2\pi F_c t) \tag{6.14}$$

The other way around is to receive with both cosine and sine carriers. This way, we can capture and demodulate signals even the transmitter only uses either cosine or sine carrier.

By introducing a scale factor m_p to adjust baseband signal x to appropriate frequency deviation, the modulated signal can be represented as follows using Eq.6.9.

$$y(t) = \cos(2\pi F_c t + m_p \frac{dx}{dt}) + j\sin(2\pi F_c t + m_p \frac{dx}{dt}) \tag{6.15}$$

Example 6.4 *(modulatevoicefm) In the complex plane, $y(t)$ rotates $\frac{2\pi F_c}{F_s}$ radian where F_s is the sampling rate. We have to adjust $m_p \frac{dx}{dt}$ within the range of $-\frac{\pi F_c}{F_s} < m_p \frac{dx}{dt} < \frac{\pi F_c}{F_s}$ by choosing appropriate m_p as shown in Fig.6.9. In an audio file we can store the real and imaginary parts of modulated signal in the first and the second channels, respectively, in a form of array. The first and the second channels are for the left and right channels of stereo sound. Following example produces FM signal of an audio file. The* `dev` *is derived by observing the signal* `vrr` *to find the maximum and minimum values are 0.6 and -0.6, respectively. Suppose the maximum/minimum values is reached from zero at one sample duration $1/20000$ second, the derivative becomes $12000 = 0.6 \times 20000$ and this value shall be adjusted by m_p to be less than the carrier progress within one sample such that*

$$5000 > \frac{m_p x}{2\pi} = m_p \frac{12000}{2\pi} \tag{6.16}$$

m_p less than 1.0 is sufficiently small in this case. In the following example, $m_p = 0.2$ is used. In other words, $m_p > \frac{5000 \times 2\pi}{12000} = 2.61$ does not work because it exceeds the rotation angle of the carrier wave per one sample.

```
%%  voice data fm modulate
%
Fs = 20000; %sampling frequency
Fc = 5000; %carrier frequency
dev = 0.2;  %adjust sigma
[v, Fvs] = audioread('voice.wav');
[n, d] = rat(Fvs/Fs);
vrr = resample(v, n, d);
[r, c] = size(vrr);
t = (0:1/Fs:r/Fs-1/Fs);
```

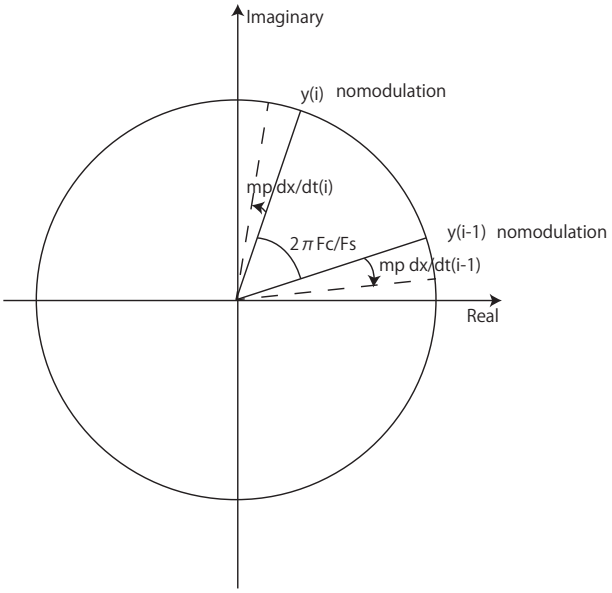

Figure 6.9: Frequency modulated signal in the complex plane

```
y = zeros(r,2); % IQ signal storage
for i=1:r
    y(i,1) = cos(2*pi()*(Fc+vrr(i)*dev)*t(i)); % carrier amp is one
    y(i,2) = sin(2*pi()*(Fc+vrr(i)*dev)*t(i));
end
subplot(2,1,1)
plot(y(:,1));
title('modulated signal')
subplot(2,1,2)
plot(vrr)
title('baseband signal')
audiowrite('voice5k20kfm.wav', y, Fs);
```

Demodulation of the FM signal can be done by measuring consecutive two samples. Suppose the angle between the consecutive samples between $y(i)$ and $y(i-1)$ is $\Omega(i)$, the modulated signal can be derived such that

$$\frac{dx}{dt}(i) = \frac{dx}{dt}(i-1) + \frac{\Omega(i) - 2\pi F_c/F_s}{m_p} \tag{6.17}$$

as shown in Fig.6.10. $\Omega(i)$ can be calculated by taking inner product of the two samples y_{i-1} and y_i such that

$$\cos\Omega = \frac{y_i^T y_{i-1}}{|y_i||y_{i-1}|} \tag{6.18}$$

Note that the inner product exploits the arc-cosine function to yield Ω and the subtraction with the carrier phase $2\pi F_c/F_s$ produces either a positive or a negative baseband signal. Taking a difference of two consecutive samples is equivalent to differentiation.

We can also apply a downconversion to separate the baseband component ϕ of FM signal. IQ signal components of an fm modulated signal can be written as follows, in general.

$$I = \cos(2\pi(F_c + \phi)t) \tag{6.19}$$
$$Q = \sin(2\pi(F_c + \phi)t) \tag{6.20}$$

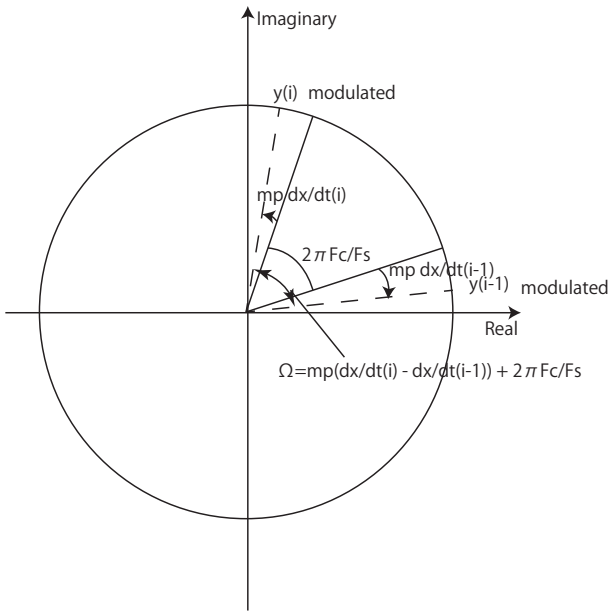

Figure 6.10: Demodulation of FM signal

These IQ signal basically rotates in the complex plane with average angular speed of $2\pi F_c$. A complex down conversion can be performed such that

$$\begin{bmatrix} \cos(2\pi F_c t + \Psi) & \sin(2\pi F_c t + \Psi) \\ -\sin(2\pi F_c t + \Psi) & \cos(2\pi F_c t + \Psi) \end{bmatrix} \begin{Bmatrix} \cos(2\pi(F_c + \phi)t) \\ \sin(2\pi(F_c + \phi)t) \end{Bmatrix} = \begin{Bmatrix} \cos(2\pi(\cancel{F_c} - \cancel{F_c} - \phi)t + \Psi) \\ \sin(2\pi(\cancel{F_c} - \cancel{F_c} - \phi)t + \Psi) \end{Bmatrix}, \tag{6.21}$$

where Ψ is the phase offset between the transmitter and the receiver. An example of down converted FM signal on a complex plane is given in Fig. 6.11. Ψ can be removed either by subtracting the mean angle, or by subtracting

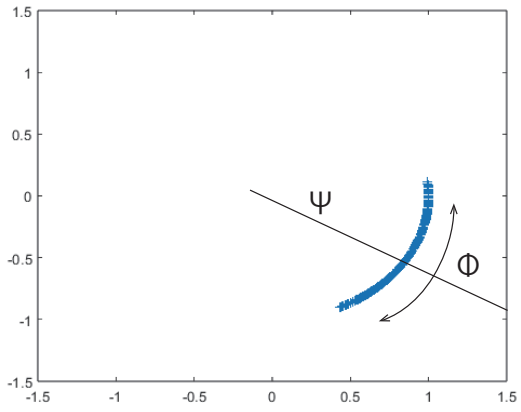

Figure 6.11: Downconverted FM signal constellation

consecutive two samples. Note that as the subtracting consecutive two samples is practically a differentiation of the original baseband signal, we need to integrate the result to recover original baseband signal.

Example 6.5 *(voicefmdc) The following script segment demodulate a downconverted fm modulated signal.*

```
my = mean(angle(yf)); % phase offset
z = zeros(r,1); % for integratoin
for i=1:r-1
    %% y(i) = (angle(yf(i))-my)/2/pi/dev; % use of the average
    z(i+1) = z(i) + (angle(yf(i+1)*conj(yf(i))))/2/pi/dev; % derivative and integration
end
```

Example 6.6 *(voicedemodfm) Compose a script to demodulate a FM signal generated by Example 6.4.*

```
%%  fmdemodulation
%
Fs = 20000;
Fc = 5000;
dev = 0.2;
[yy, Fss] = audioread('voice5k20kfm.wav');
[n,d] = rat(Fss/Fs);
[r,c] = size(yy);
t = (0:1/Fss:r/Fss-1/Fss);
f = (Fss/r:Fss/r:Fss);
y = zeros(1,r);
prev = 0;
for i=1:r-1
    y(i) = acos(...
           (yy(i+1,1)*yy(i,1)+yy(i+1,2)*yy(i,2))...
           /(sqrt(yy(i+1,1)^2+yy(i+1,2)^2)*sqrt(yy(i,1)^2+yy(i,2)^2))...
           ) -2*pi*Fc/Fss;
    %prev = y(i);
    y(i) = y(i)/2/pi/dev + prev;
    prev = y(i);
end
fz = resample(y, d, n);
sound(fz, Fs);
```

Exercise 6.4 *(amcontinous) The following script produces AM modulated 440 Hz tone for 10 seconds.*

```
%%  tone modulated long signal
%   sample rate tone
Fs = 20000; % sampling
fx = 440;
Fc = 5000; %carrier frequency
t = (0:1/Fs:10-1/Fs); % from 0 to 10 seconds
x = (1+0.5*cos(2*pi*fx*t));
y = x .* cos(2*pi*Fc*t);
audiowrite('longtone.wav', y, Fs);  % save the audio data as a wave file.
```

In the previous exercise, we demodulate the AM signal by first retrieving all the audio data and multiplying the carrier frequency and applying LPF. Rewrite the script such that we retrieve the audio data by frame and piecewise demodulate by using `dsp.AudioFileReader` *and* `audioDeviceWriter` *such that*

```
fileReader = dsp.AudioFileReader(...
    'test.wav',...
    'SamplesPerFrame',framesize);
deviceWriter = audioDeviceWriter(...
    'SampleRate',Fsa);
```

and loop the signal processing with the following `while` *loop*

```
while ~isDone(fileReader)
    signal = fileReader();
     %% compose here
    deviceWriter(s);
end
```

By doing this we can handle long audio data such as music stored in files.

7 FM stereo receiver

Let us build a FM radio receiver using digital signal processing techniques. FM radio uses the carrier frequency range from 76 to 95 MHz in Japan. To capture radio wave, we need to have antenna and a frequency converter which allows us to process the captured data in computer. This can be done with a low cost USB receiver. We use a commercial USB receiver RTL-SDR for exercises.

Figure 7.1: Realtek USB receiver

7.1 Preparation

In order to use a RTL-SDR in your MATLAB, you first need to download the driver from `https://jp.mathworks.com/hardware-support/rtl-sdr.html`. From `home` tab go to `Add-ons/get hardware support hpa` Search for RTL-SDR radio and install. After the installation, go `Add-ons/Manage Add-Ons` and click `Setup` of RTL-SDR column as shown in Fig.7.2. You just need to follow the instructions shown. During the installation

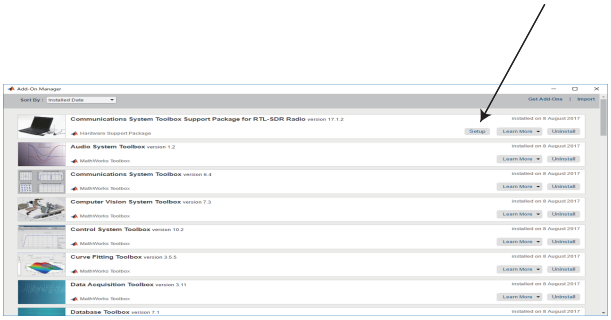

Figure 7.2: RTL-SDR setup window in MATLAB

please carefully choose `RTL 2838UHIDIR` to replacement of driver (Fig.7.3). If you see `Generic RTL283U OEM` driver in the Radio connectivity window as shown in Fig.7.4, the driver is properly installed.

After the preparation, you can check the configuration by typing in

```
>>sdrinfo
```

which should return the following.

```
>> sdrinfo
ans =
                RadioName: 'Generic RTL2832U OEM'
```

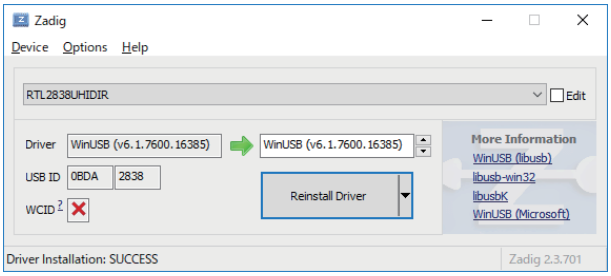

Figure 7.3: Selection of driver

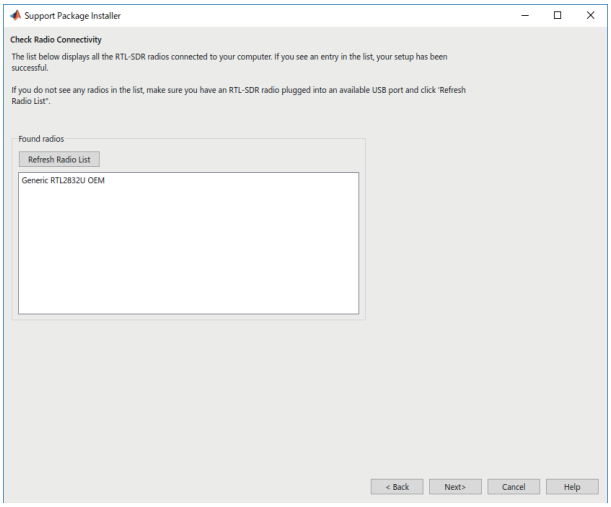

Figure 7.4: Window showing the driver is properly installed

```
            RadioAddress: '0'
             RadioIsOpen: 0
                TunerName: 'R820T'
            Manufacturer: 'Realtek'
                 Product: 'RTL2838UHIDIR'
              GainValues: [29 x 1 double]
    RTLCrystalFrequency: 28800000
  TunerCrystalFrequency: 28800000
            SamplingMode: 'Quadrature'
            OffsetTuning: 'Disabled'
```

After confirming the above, we can further test drive an example FM receiver by typing `FMReceiverExample`. In the following, we listen to NHK FM at 81.9 MHz (`81.9e+6`) for 20 seconds.

```
>FMReceiverExample
> Specify run time in seconds [10.000000]: 20
> Enter signal source.
>1) File
>2) RTL-SDR radio
>
> Signal source [1]: 1
Searching for radios connected to your host computer...
> Enter the number corresponding to the radio you would like to use.
>1) RTL-SDR [Radio Address: 0]
>> Radio [1]:
> Enter FM channel frequency (Hz) [1.025000e+08]: 81.9e+6
```

Windows 11 may not be able to recognized the dongle automatically. If that is the case, please go `https://zadig.akeo.ie/` and download the driver and install. This workaround worked for me.

7.2 Driver download failure

If you fail to download the third party softwares from Hardware support package installer, check the `MATLAB` log, which is located under your home directory such as

```
/Users/mitsugi/AppData/Local/Temp/mathworks_mitsugi.log
```

Your `AppData` directory may be invisible in windows.

The log can be read with a text editor. In my case, the following error was in the log.

```
java.net.SocketException: Address family not supported by protocol family
```

This can be interpreted as a java library tried to use a wrong connection method to download the files. A workaround was to change the environmental variables `_JAVA_OPTIONS` by adding

```
Add -Djava.net.preferIPv4Stack=true
```

7.3 FM radio anatomy

FM stereo broadcasting transfers left (L) and right (R) channels data by separately transfer the combined (L+R) and difference (L-R) signal. in the transmitter side. The receiver demodulates either only the combined signal to produce only monaural sound or both the combined and difference signal to produce stereo sound. The reconstruction of stereo signal demands good quality signal reception.

The combined signal centers the carrier frequency with 15 kHz bandwidth. The difference signal is centered 38 kHz occupying 15 kHz in both side as shown in Fig.7.5. The sampling rate of the audio signal should cover the 15 kHz bandwidth, thus needs to be larger than 30 kHz. Let us use 48 kHz. The value is borrowed from the example in the exercise below. In addition to the stereo audio, we can send digital data using Data Radio Channel

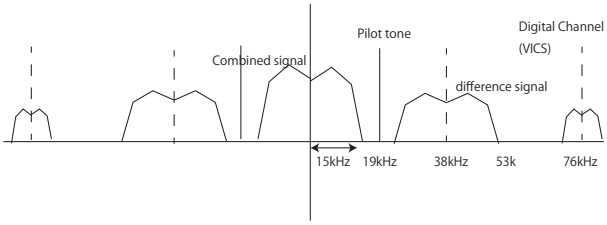

Figure 7.5: FM stereo signal

(DARC) protocol, which is used in VICS . The whole signal (including combined and difference signal and pilot) is frequency modulated with frequency deviation of 75 kHz.

Usually audio signal has less intensity in the high frequency region while a FM transmission is vulnerable to noises when the signal intensity is low. Because of this reason, high frequency signal of FM transmission easily degrades. To mitigate this degradation, the frequency characteristics of the baseband audio signal is modified to enhance high frequency components to increase SNR before transmission and is recovered in the receiver. These signal processing are referred to as pre-emphasis and de-emphasis.

The premphasis is done with the following high pass filter.

$$H_p(f) = 1 + j2\pi f\tau \tag{7.1}$$

where τ is a constant (50 usec in Japan and US). The deemphasis filter is the reciprocal of the HPF

$$H_d(f) = \frac{1}{1 + j2\pi f\tau} \tag{7.2}$$

The frequency response of the preemphasis and deemphasis filters are shown in Fig.7.6.

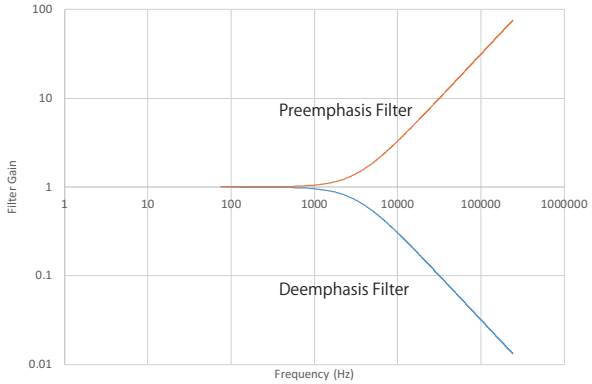

Figure 7.6: Preemphasis and deemphasis filters frequency responses

In order to demodulate a FM broadcasting, we first need to connect to RTL-SDR, this can be done with the following by specifying the center (carrier) frequency (in case of 81.9MHz, it is 81900000), sampling rate (240kHz to cover 75 kHz bandwidth) and the frame size (3200 samples/sec borrowed from the example).

```
h = comm.SDRRTLReceiver('0','CenterFrequency', centerfreq, ...
   'SampleRate',samplerate, ...
```

```
    'SamplesPerFrame', framesize, ...
    'EnableTunerAGC', true, ...
    'OutputDataType', 'double');
```

To repeat the continuous acquisition of frames from RTLSDR and process them in MATLAB, the following program segment can be used.

```
%create audio player with the audio sampling rate (audiosr)
player = audioDeviceWriter( 'SampleRate',audiosr);
if ~isempty(sdrinfo(h.RadioAddress)) % acquire frame until empty
    while(1)
        [audiodata, ~] = step(h);  % fetch data and save in audio variable.
       demodData = fmd(audiodata); % demodulation
      % apply lowpass filter
      % obtain 19k pilot tone to extract stereo signals
      % peak fequency required to decode stereo sounds
      % apply diemphasis filter
      % resample (->48kHz)
      % aggregates two channel
      aggch = horzcat(leftch,rightch); %left and right channels are merged
       player(aggch); %playback the audio stream as stereo
    end
end
```

If a stereo receiver is required, we need to demodulate the difference signal populated at 38kHz, which should be extracted with a band pass filter from 23 kHz to 52 kHz (38 kHz $\pm$ 15 kHz). When we use `MATLAB` it is easy to create a 38 kHz carrier to down convert the difference signal to baseband. But in general radio set hardware, producing a carrier requires additional clock circuitry. The pilot signal of 19 kHz is , in fact, provided to produce a 38 kHz carrier without enforcing a radio set to implement additional clock circuitry.

We can extract the 19 kHz carrier as follows. But in general, artificially produced 38 kHz carrier works as well.

```
d = fdesign.peak('N,F0,BW,Ast',20,2*19000/samplerate,.02,80);
peakf = design(d,'cheby2','SystemObject',true);
fvtool(peakf)
```

Applying this filter to the received signal produces a 19 kHz pilot tone. In order to create the doubled frequency tone, we square the 19 kHz tone, normalize and apply a high pass filter.

```
        a19k = peakf(demodData); % apply the peak filter to received signal
        a38k = a19k .* a19k;  % square to produce double frequency tone
        a38k = a38k/(max(a38k2)); %nomarize
        a38k = hpf38kFIR(a38k2); % apply a high pass filter to extract 38 kHz tone
```

After all the filtering and processing, 240kHz sampling frequency is too much for the baseband signal, which only has 15 kHz bandwidth. Resampling before streaming the demodulated signal to the audio player is recommended.

We can store IQ data of FM broadcasting captured by RTL-SDR and playback later. We can use the mechanism of audio stream writing/reading mechanism for this purpose. Since a frame captured by RTL-SDR stores IQ data in a complex array while an audio data is an array of double or integer, we need to covert it to a double array as two channels, one for In-phase and the other for Quadrature stream. An example script to write a frame data continuously to a file with `AudioFileWriter` is as follows.

```
iqw = dsp.AudioFileWriter('radioiq.wav', ...
    'SampleRate', samplerate, 'DataType', 'double');
if ~isempty(sdrinfo(h.RadioAddress))
    while(1)
        [data, ~] = step(h);  % no 'len' output needed for blocking operation
        ddata = [real(data) imag(data)];
        iqw(ddata);
```

To replay the stored IQ stream, we use `AudioFileReader`. It should be noted that after acquiring from the two channel data from the file, we need to convert the two channel data to an array of complex number for demodulation.

```
iqr = dsp.AudioFileReader('radioiq.wav', ...
      'SamplesPerFrame', framesize);
while ~isDone(iqr)
        audiodata = iqr(); % capture two channel data from the file
           % the following is needed to recover IQ stream format
        complexaudio = complex(audiodata(:,1), audiodata(:,2));
```

Exercise 7.1 *(rtl2file/rtlfromfile) Compose either monaural or stereo FM receiver script. You can use* `MATLAB` *function such as* `comm.FMDemodulator` *and* `FilterDesigners` *but copying entire* `FMDemodulatorExample` *is not allowed.*

References

[1] C. Bishop. *Pattern Recoginition and Machine Learning*. Springer, 2006.

[2] A. Kaehler G. Bradski. *Learning Open CV3*. O'Reilly, 2017.

[3] J.G. Proakis D.G. Manolakis. *Digital Signal Processing, 4th edition*. Prentice Hall, 2007.

[4] J.G. Proakis D.G. Manolakis. *Digital Communications, 5th edition*. MacGraw-Hill, 2008.

[5] R.G.Lyons. *Understanding Digital Signal Processing, 3rd edition*. Prentice Hall, 2010.

[6] J.Lilly S.Olhed. Generalized morse wavelets as a superfamily of analytic wavelets. *IEEE Transactions on Signal Processing*, 60(11):6036–6041, 2012.