# Lab 3: Fourier Series Representation of Periodic Signals

| Author | Name & Student ID : 王卓扬（12112907）、冯彦捷（12010825） |
|---|---|

## Introduction

1. Use MATLAB to synthesize signals with the DTFS in two ways.
2. Use MATLAB to calculate DTFS in two ways.
3. Use MATLAB to generate periodic signals.
4. Use filter to calculate the response of a system described by differential equations.
5. Use freqz to calculate the frequency response of a system mentioned above.
6. Use fft and ifft to accelerate the calculating progress.
7. Compare the speed of brute method and fft algorithm.
8. Learn about time complexity and its calculation.
9. Use tic and toc to measure the time cost.

## Lab results & Analysis:

### ■ 3.5 Synthesizing Signals with the Discrete-Time Fourier Series

The discrete-time Fourier series (DTFS) is a frequency-domain representation for periodic discrete-time sequences. The synthesis and analysis equations for the DTFS are given by Eqs. (3.1) and (3.2). This exercise contains three sets of problems to give you practice working with these equations. The Basic Problems allow you to synthesize a very simple periodic discrete-time signal from its DTFS coefficients. In the Intermediate Problems, you will both analyze a set of periodic discrete-time signals to obtain their DTFS coefficients, and construct one of these signals by adding in a few coefficients at a time. For the Advanced Problem, you will write a function to find the DTFS coefficients of an arbitrary periodic discrete-time signal from one period of samples.

#### Basic Problems

In these problems, you will synthesize a periodic discrete-time signal with period $N = 5$ and the following DTFS coefficients

$$a_0 = 1,\ a_2 = a_{-2}^* = e^{j\pi/4},\ a_4 = a_{-4}^* = 2e^{j\pi/3}.$$

(a). Based on the DTFS coefficients, do you expect $x[n]$ to be complex-valued, purely real, or purely imaginary? Why?

**Solution (a):**

$x[n]$ is purely real.

**Analysis (a):**

We have

$$a_k = a_{-k}^*$$

According to the Conjugation Symmetry Principle, we know

$$x[n] = x^*[n]$$

That means $x[n]$ is purely real.

(b). Using the DTFS coefficients given above, determine the values of $a_0$ through $a_4$ and specify a vector **a** containing these values.

**Solution (b):**

$$\underline{a[k] = \left[1, 2e^{-j\frac{\pi}{3}}, e^{j\frac{\pi}{4}}, e^{-j\frac{\pi}{4}}, 2e^{j\frac{\pi}{3}}\right], k = 0, 1, 2, 3, 4}$$

**Analysis (b):**

We know $a_k = a_{k+N}$, and $N = 5$, so $a_1 = a_{-4+5} = 2e^{-j\frac{\pi}{3}}$, $a_3 = a_{-2+5} = e^{-j\frac{\pi}{4}}$.

(c). Using the vector **a** of DTFS coefficients and the synthesis equation, define a new vector **x** containing one period of the signal $x[n]$ for $0 \le n \le 4$. You can either write out the summation explicitly or you may find it helpful to use a **for** loop. Generate an appropriately labeled plot of $x[n]$ for $0 \le n \le 4$ using **stem**. Was your prediction in Part (a) correct? Note that if you predicted a purely imaginary or real signal, it may still have a very small ($< 10^{-10}$) nonzero real or imaginary part due to roundoff errors. If this is the case, set this part to be zero using **real** or **imag** as appropriate before making your plot.

**Solution (c):**

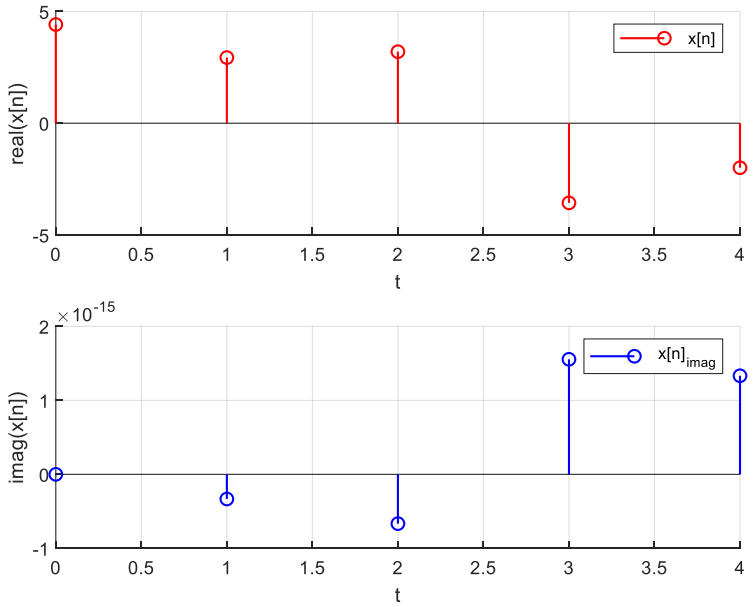



Fig 3.5.c

<u>The prediction is correct. And the values have tiny errors in imaginary part.</u>

**Analysis (c):**

According to the Synthesis Equation

$$x_n = \sum_{k=<N>} a_k e^{jk\frac{2\pi}{N}n}$$

Write ***x*** and ***a*** as column vectors. Let

$$E = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & e^{j\omega_0} & \dots & e^{j(N-1)\omega_0} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & e^{j(N-1)\omega_0} & \dots & e^{j(N-1)^2\omega_0} \end{bmatrix}, \ \omega_0 = \frac{2\pi}{N}$$

Then we have

$$\boldsymbol{x} = E\boldsymbol{a}$$

We represent the matrix E in MATLAB as

```
E=exp(2*pi*1j/N).^reshape(fix((0:N*N-1)/N).*mod((0:N*N-1),N),N,N);
```

## Intermediate Problems

For these problems, you will examine the DTFS representation of several different square waves. Specifically, you will look at signals

$$x_1[n] = \begin{cases} 1, & 0 \le n \le 7, \end{cases} \tag{3.4}$$

$$x_2[n] = \begin{cases} 1, & 0 \le n \le 7, \\ 0, & 8 \le n \le 15, \end{cases} \tag{3.5}$$

$$x_3[n] = \begin{cases} 1, & 0 \le n \le 7, \\ 0, & 8 \le n \le 31, \end{cases} \tag{3.6}$$

where $x_1[n]$, $x_2[n]$ and $x_3[n]$ have periods of $N_1 = 8$, $N_2 = 16$ and $N_3 = 32$, respectively.

(d). Define three vectors `x1`, `x2`, and `x3` representing one period of each of the signals $x_1[n]$, $x_2[n]$ and $x_3[n]$. Using these vectors, make appropriately labeled plots of each of the signals over the range $0 \le n \le 63$. Note: You will have to repeat the vectors you have defined to cover this range of samples.

**Solution (d):**

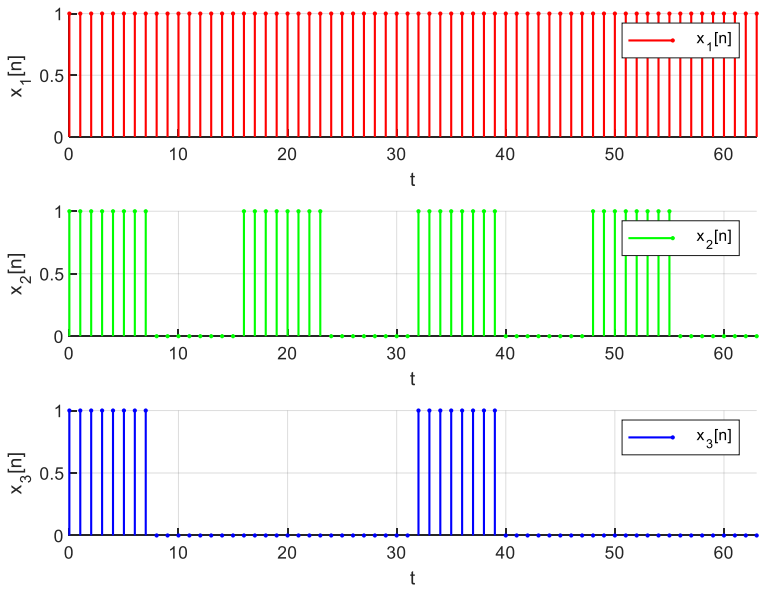



Fig 3.5.d

**Analysis (d):** We write a function *DTS.Periodic(D, V)* to generate DT periodic signals, which are shown in Fig 3.5.d.

(e). Exercise 3.1 explains how to use `fft` to compute the DTFS of a periodic discrete-time signal from one period of the signal. Using the `fft` function, define the vectors `a1`, `a2`, and `a3` to be the DTFS coefficients of $x_1[n]$ through $x_3[n]$, respectively. Generate appropriately labeled plots of the magnitude of each of the DTFS coefficient sequences

**Solution (e):**

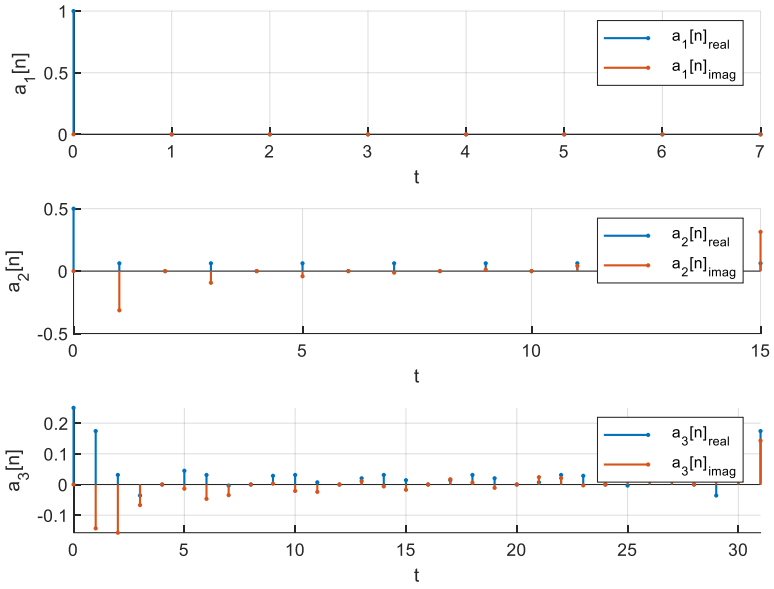



Fig 3.5.e (1)

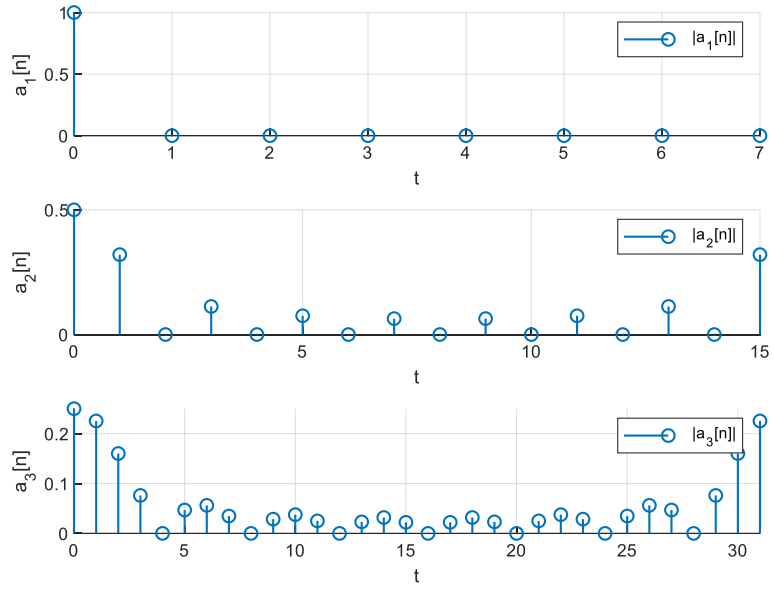



Fig 3.5.e (2)

**Analysis (e):** In Fig 3.5.e (1) are coefficients of DTFS of the periodic signals represented by the combination of the real part and the imaginary part. To see more clearly in Fig 3.5.e (2) are the length of the coefficients.

(f). In this part, you will examine how $x_3[n]$ appears when it is synthesized a few coefficients at a time. Using the vector `a3` you found in the previous part, define vectors `x3_2`, `x3_8`, `x3_12` and `x3_all` corresponding to the following four signals

$$x_{3\_2}[n] = \sum_{k=-2}^{2} a_k e^{jk(2\pi/32)n},$$

$$x_{3\_8}[n] = \sum_{k=-8}^{8} a_k e^{jk(2\pi/32)n},$$

$$x_{3\_12}[n] = \sum_{k=-12}^{12} a_k e^{jk(2\pi/32)n},$$

$$x_{3\_all}[n] = \sum_{k=-15}^{16} a_k e^{jk(2\pi/32)n},$$

on the interval $0 \le n \le 31$. Note that since $x_3[n]$ is real, the DTFS coefficients for this signal will be conjugate symmetric, i.e., $a_k = a^*_{-k}$. Because $a_k$ is conjugate symmetric and all of the sums except $x_{3\_all}[n]$ are symmetric about $k = 0$, the resulting time signals should be purely real. If you are unclear about why this is true, you may want to review the symmetry properties of the DTFS. Due to roundoff error in MATLAB, you may need to discard some very small imaginary parts of the signals you synthesize using `real`. The sums specified above are symmetric about $k = 0$ but the vector `a3` represents $a_k$ for $k = 0, \ldots, 31$ as `[a3(1),...,a3(32)]`, so you will need to determine which elements of `a` correspond to the negative values of $k$ when you implement the sums.

**Solution (f):**

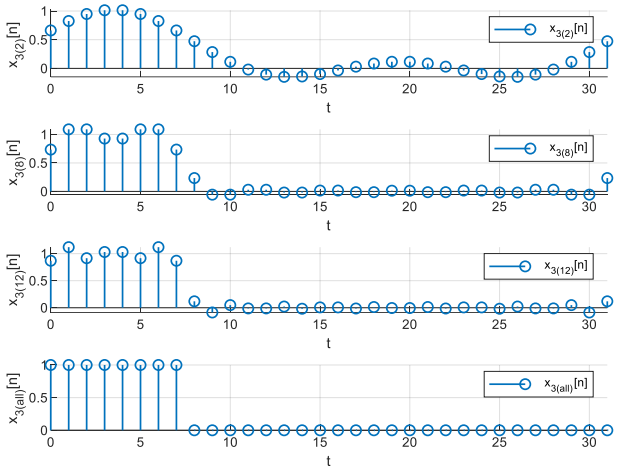



Fig 3.5.f

**Analysis (f):** We rearrange a[k] by index -15 ~ 16, and use *circshift* to roll the vector. The results are shown in Fig 3.5.f.

(g). Argue that $x_{3\_all}[n]$ must be a real signal.

**Solution (g):**

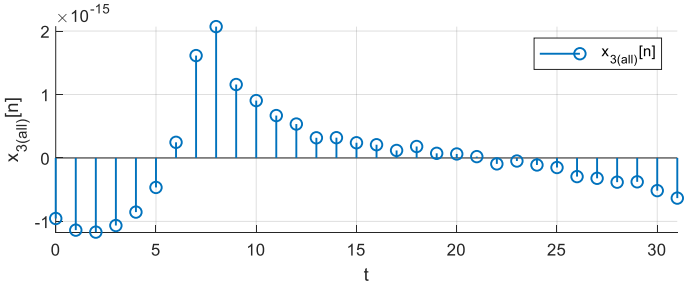



Fig 3.5.g

**Analysis (g):** From the point of view of experiment, the imaginary parts of $x_{3_{all}}[n]$ are all small enough to be ignored in Fig 3.5.g. To derive this:

First, for $\forall N \in \mathbb{N}^*$, we map $a_k$ to the domain $k \in [-\left\lfloor\frac{N-1}{2}\right\rfloor, \left\lfloor\frac{N}{2}\right\rfloor]$. And we have

$$a_k = a_{-k}^*$$

Let

$$x[n] = \sum_{k=<N>} a_k e^{jk\frac{2\pi}{N}n} := \sum_{k=<N>} f(k)$$

So for $k > 0$ we have

$$\begin{aligned} f(k) + f(-k) &= a_k e^{jk\frac{2\pi}{N}n} + a_{-k} e^{-jk\frac{2\pi}{N}n} \\ &= a_k e^{jk\frac{2\pi}{N}n} + a_k^* e^{-jk\frac{2\pi}{N}n} \\ &= a_k e^{jk\frac{2\pi}{N}n} + \left(a_k e^{jk\frac{2\pi}{N}n}\right)^* \\ &= f(k) + f^*(-k) \end{aligned}$$

which is real. Moreover, we have $f(0) = a_0 = a_{-0}^*$, which is real. When *N* is even, we have $f\left(\frac{N}{2}\right) = a_{N-1}$, which is real in this case. So the summation $x[n]$ of $f(k)$ is real.

(h). Generate a sequence of appropriately labeled plots using `stem` showing how the signals you created converge to $x_3[n]$ as more of the DTFS coefficients are included in the sum. Specifically, `x3_all` should be equal to the original vector `x3` within the bounds of MATLAB's roundoff error. Does the synthesis of this discrete-time square wave display the Gibb's phenomenon?

**Solution (h):**

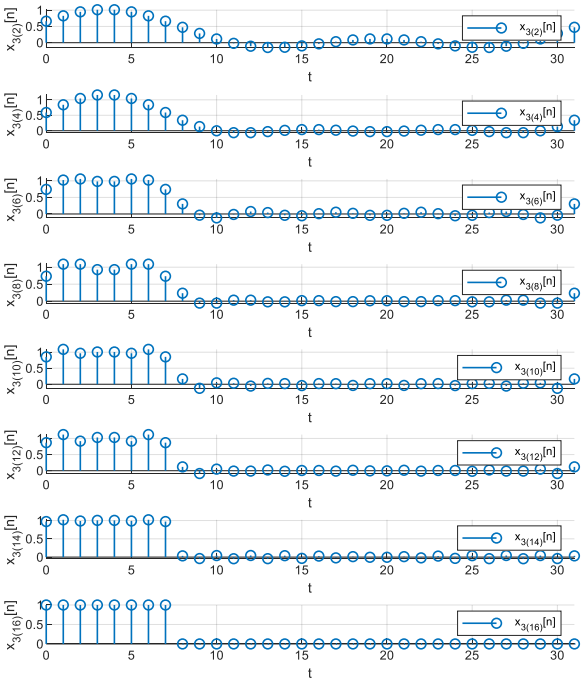



Fig 3.5.h

And there is no Gibb's phenomenon.

**Analysis (h):** We plot the situation from $x_{3\_2}$ to $x_{3\_all}$ in Fig 3.5.h.

## Advanced Problem

For this problem, you will write a function which computes the DTFS coefficients of a periodic signal. Your function should take as arguments the vector `x`, which specifies the values of the signal $x[n]$ over one period, and `n_init` which specifies the time index $n$ of the first sample of `x`. Your function should return the vector `a` containing the DTFS coefficients $a_0$ through $a_{N-1}$, where $N$ is the number of samples in `x`, or equivalently, the period of $x[n]$. The first line of your M-file should read

```
function a = dtfs(x,n_init)
```

Verify your function is working correctly by computing the DTFS coefficients for the signals given below and demonstrate that the output of your function matches the outputs below.

```
>> dtfs([1 2 3 4],0)
ans =
   2.5000          -0.5000 + 0.5000i  -0.5000            -0.5000 - 0.5000i
>> dtfs([1 2 3 4],1)
ans =
   2.5000           0.5000 + 0.5000i   0.5000 + 0.0000i   0.5000 - 0.5000i
>> dtfs([1 2 3 4],-1)
ans =
   2.5000          -0.5000 - 0.5000i   0.5000 - 0.0000i  -0.5000 + 0.5000i
>> dtfs([2 3 4 1],0)
ans =
   2.5000          -0.5000 - 0.5000i   0.5000            -0.5000 + 0.5000i
>> dtfs([ones(1,4) zeros(1,4)],0)
ans =
  Columns 1 through 4

   0.5000           0.1250 - 0.3018i        0             0.1250 - 0.0518i
  Columns 5 through 8
        0           0.1250 + 0.0518i        0             0.1250 + 0.3018i
>> dtfs([ones(1,4) zeros(1,4)],2)
ans =
  Columns 1 through 4
   0.5000          -0.3018 - 0.1250i        0             0.0518 + 0.1250i
  Columns 5 through 8
        0           0.0518 - 0.1250i        0            -0.3018 + 0.1250i
```

**Solution (i):**

The results are as below:

```
+=========================================================================+
>> dtfs([1 2 3 4], 0)
ans = 2.5000 + 0.0000i  -0.5000 + 0.5000i  -0.5000 - 0.0000i  -0.5000 - 0.5000i
>> dtfs([1 2 3 4], 1)
ans = 2.5000 + 0.0000i   0.5000 + 0.5000i   0.5000 - 0.0000i   0.5000 - 0.5000i
>> dtfs([1 2 3 4], -1)
```

```
ans = 2.5000 + 0.0000i  -0.5000 - 0.5000i   0.5000 + 0.0000i  -0.5000 + 0.5000i
>> dtfs([2 3 4 1], 0)
ans = 2.5000 + 0.0000i  -0.5000 - 0.5000i   0.5000 + 0.0000i  -0.5000 + 0.5000i
>> dtfs([ones(1, 4) zeros(1, 4)], 0)
ans = 0.5000 + 0.0000i   0.1250 - 0.3018i  -0.0000 - 0.0000i   0.1250 - 0.0518i
0.0000 - 0.0000i   0.1250 + 0.0518i   0.0000 - 0.0000i   0.1250 + 0.3018i
>> dtfs([ones(1, 4) zeros(1, 4)], 2)
ans = 0.5000 + 0.0000i  -0.3018 - 0.1250i   0.0000 + 0.0000i   0.0518 + 0.1250i
0.0000 - 0.0000i   0.0518 - 0.1250i  -0.0000 + 0.0000i  -0.3018 + 0.1250i
+=======================================================================+
```

Which match the answers well.

**Analysis (i):** The code **(class *DTS* is shown in Appendix)**:

```
function a = dtfs(x, n_init)
    a = DTS({n_init, n_init + length(x) - 1}, x).dtfs;
end
```

## ■ 3.8 First-Order Recursive Discrete-Time Filters

This exercise demonstrates the effect of first-order recursive discrete-time filters on periodic signals. You will examine the frequency responses of two different systems and also construct a periodic signal to use as input for these systems. This exercise assumes you are familiar with using `fft` and `ifft` to compute the DTFS of a periodic signal as described in Tutorial 3.1. In addition, it is also assumed you are proficient with the `filter` and `freqz` commands described in Tutorials 2.2 and 3.2. Several parts of this exercise require you to generate vectors which should be purely real, but have very small imaginary parts due to roundoff errors. Use `real` to remove these residual imaginary parts from these vectors.

This exercise focuses on two causal LTI systems described by first-order recursive difference equations:

$$\begin{array}{lrcl} \text{System 1:} & y[n] - 0.8y[n-1] & = & x[n], \\ \text{System 2:} & y[n] + 0.8y[n-1] & = & x[n]. \end{array}$$

The input signal $x[n]$ will be the periodic signal with period $N = 20$ described by the DTFS coefficients

$$a_k = \begin{cases} 3/4\,, & k = \pm 1\,, \\ -1/2\,, & k = \pm 9\,, \\ 0\,, & \text{otherwise}\,. \end{cases} \tag{3.10}$$

### Intermediate Problems

(a). Define vectors `a1` and `b1` for the difference equation describing System 1 in the format specified by `filter` and `freqz`. Similarly, define `a2` and `b2` to describe System 2.

**Solution (a):** The code:

```
a1 = [1, -0.8];
b1 = [1];
a2 = [1, 0.8];
b2 = [1];
```

(b). Use `freqz` to evaluate the frequency responses of Systems 1 and 2 at 1024 points between 0 and $2\pi$. Note that you will have to use the `'whole'` option with `freqz` to do this. Use `plot` and `abs` to generate appropriately labeled graphs of the magnitude of the frequency response for both systems. Based on the frequency response plots, specify whether each system is a highpass, lowpass, or bandpass filter.

**Solution (b):**

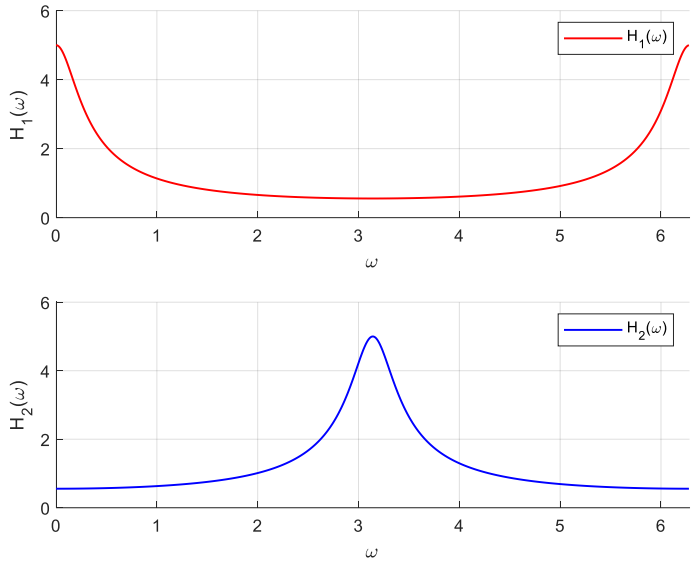



Fig 3.8.b

System 1 is a low-pass filter;

System 2 is a high-pass filter.

**Analysis (b):** As shown in Fig 3.8.b, we can see from the frequency response that System 1 is a low-pass filter for its high gain near $\pi$, while the System 2 is a high-pass filter for its high gain near $0$ and $2\pi$.

(c). Use Eq. (3.10) to define the vector `a_x` to be the DTFS coefficients of $x[n]$ for $0 \leq k \leq 19$. Generate a plot of the DTFS coefficients using `stem` where the x-axis is labeled with frequency $\omega_k = (2\pi/20)k$. Compare this plot with the frequency responses you generated in Part (b), and for each system state which frequency components will be amplified and which will be attenuated when $x[n]$ is the input to the system.

**Solution (c):**

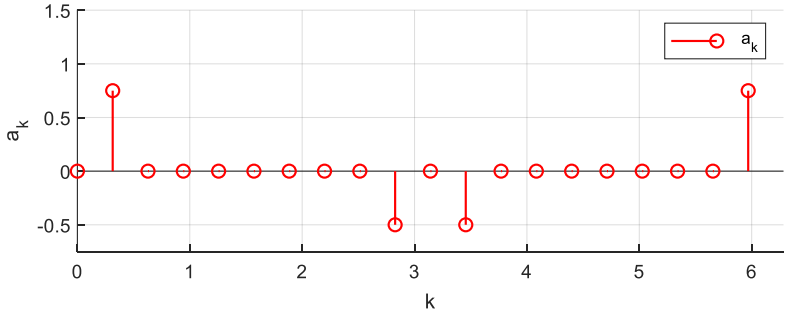



Fig 3.8.c

Statement seen below.

**Analysis (c):** $a_x$ is plotted in Fig 3.8.c. For System 1, $a_1$ and $a_{-1} = a_{19}$ will be amplified while $a_9$ and $a_{-9} = a_{11}$ will be attenuated. For System 2, $a_1$ and $a_{-1} = a_{19}$ will

be attenuated while $a_9$ and $a_{-9} = a_{11}$ will be amplified.

(d). Define `x_20` to be one period of $x[n]$ for $0 \leq n \leq 19$ using `ifft` with `a_x` as specified in Tutorial 3.1. Use `x_20` to create `x`, consisting of 6 periods of $x[n]$ for $-20 \leq n \leq 99$. Also define `n` to be this range of sample indices, and generate a plot of $x[n]$ on this interval using `stem`.

**Solution (d):**

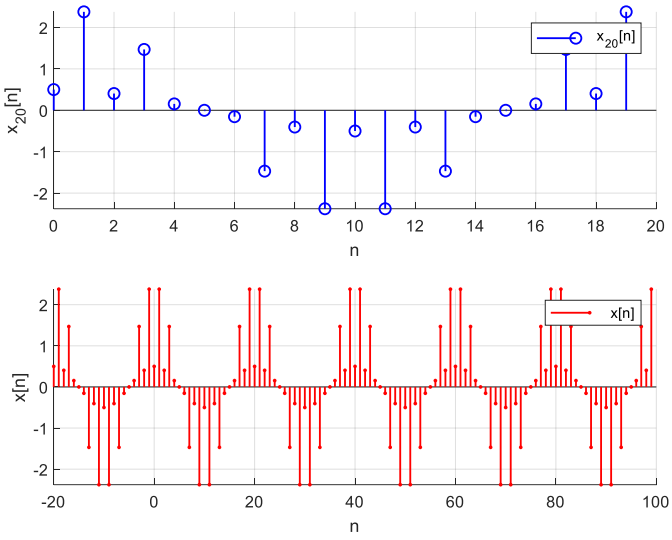



Fig 3.8.d

**Analysis (d):** $x_{20}[n]$ and $x[n]$ are shown in Fig 3.8.d. They look weird.

(e). Use `filter` to compute `y1` and `y2`, the outputs of Systems 1 and 2 when $x[n]$ is the input. Plot both outputs for $0 \leq n \leq 99$ using `stem`. Based on these plots, state which output contains more high frequency energy and which has more low frequency energy. Do the plots confirm your answers in Part (c)?

**Solution (e):**

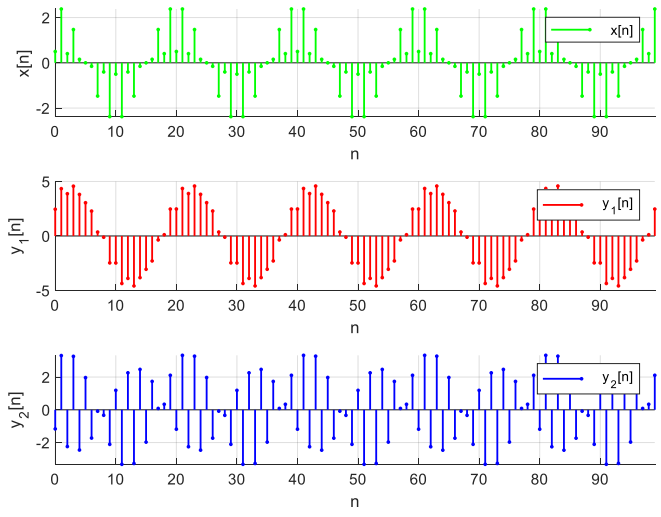



Fig 3.8.e

**Analysis (e):** The results are shown in Fig 3.8.e. We can see that $y_1[n]$ has more low frequency energy, and $y_2[n]$ has more high frequency energy. That confirmed our answers in Part (c).

(f). Define `y1_20` and `y2_20` to be the segments of `y1` and `y2` corresponding to $y_1[n]$ and $y_2[n]$ for $0 \leq n \leq 19$. Use these vectors and `fft` to compute `a_y1` and `a_y2`, the DTFS coefficients of `y1` and `y2`. Use `stem` and `abs` to generate plots of the magnitudes of the DTFS coefficients for both sequences. Do these plots agree with your answers in Part (e)?

**Solution (f):**

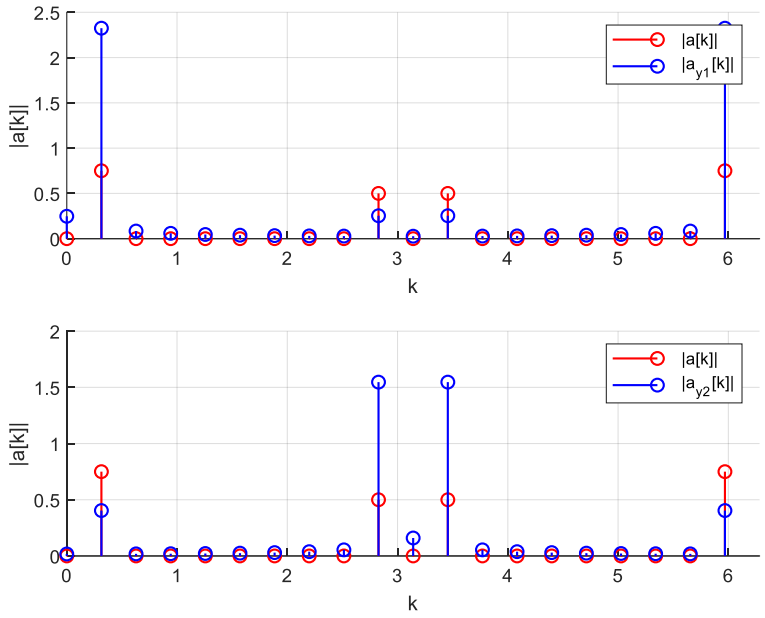



Fig 3.8.f

**Analysis (f):** The results are shown in Fig 3.8.f. We plotted the absolute value of $a_{y1}[k]$ and $a_{y2}[k]$ together with $a[k]$. And we can obviously see that the gain agrees our answers in parts before.

## ■ 3.9 Frequency Response of a Continuous-Time System

This exercise demonstrates the effect of the frequency response of a continuous-time system on periodic signals. You will examine the response of a simple linear system to each of the harmonics that compose a periodic signal as well as to the periodic signal itself. In this exercise you will need to use the function `lsim` as discussed in the Tutorial 3.3.

Consider a simple RC circuit that has a system function given by

$$H(s) = \frac{1}{1 + RCs},$$

whose input is given by

$$x(t) = \cos(t),$$

and whose output is $y(t)$. For the problems that follow, use `t=linspace(0,20,1000)` for all simulations, and assume that the time constant $RC$ is 1.

## Intermediate Problems

(a). Use the function `lsim` to simulate the response of the system $H(s)$ to $x(t)$ over $0 \leq t \leq 20$, storing the response in the signal $y(t)$. Plot the output $y(t)$ and the input $x(t)$ for $10 \leq t \leq 20$ on the same graph, and note the amplitude and phase change from input to output. Can you explain each of these effects in terms of the frequency response? Hint: Use the system function $H(s)$ to determine the exact form of the output $y(t)$ when the input is $x(t) = \cos(t)$.

**Solution (a):**

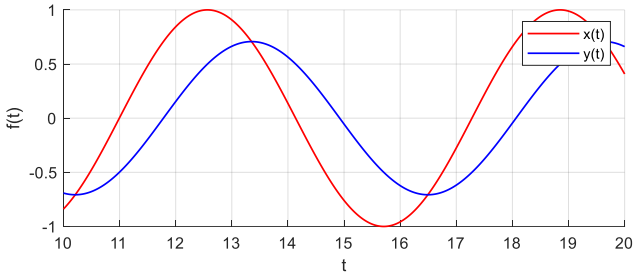



Fig 3.9.a

**Analysis (a):** The input and output are shown in Fig 3.9.a. The amplitude is decreased and the phase changes. That happens because:

1. The low frequency energy are deserted which leads to a lower amplitude.
2. The delay in RC circuit leads to the phase change.

The exact form derivation:

$$x(t) = \cos(t) = \frac{1}{2}e^{jt} + \frac{1}{2}e^{-jt}$$

Turns to

$$y(t) = \frac{1}{2}\frac{1}{1+j}e^{jt} + \frac{1}{2}\frac{1}{1-j}e^{-jt}$$

After sent into $H(s)$.

(b). Now look at the response of the system to the square wave that results from first passing $x(t)$ through a hard-limiter, $x_2(t) = \text{sign}(\cos(t))$. A simple way to use selective indexing to generate the square wave is

```
>> x2=cos(t);
>> x2(x2>0)=ones(size(x2(x2>0)));
>> x2(x2<0)=-ones(size(x2(x2<0)));
```

Create the signal `x2` and use `lsim` to simulate the response of the circuit to the square wave. Plot the resulting output $y_2(t)$ over the interval, $10 \leq t \leq 20$.

**Solution (b):**

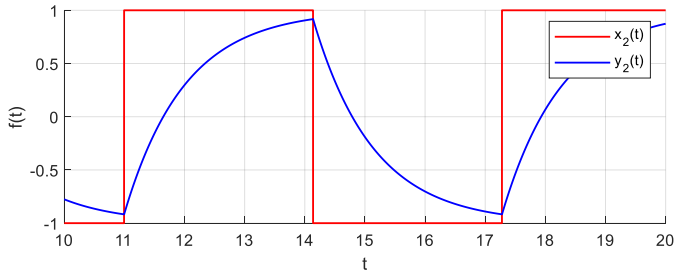



Fig 3.9.b (1)

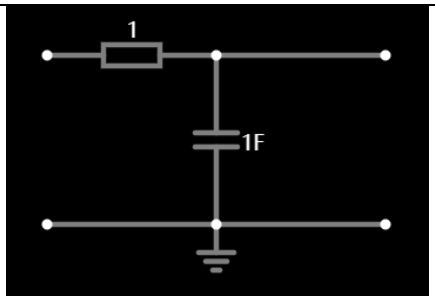



Fig 3.9.b (2)

**Analysis (b):** The results are shown in Fig 3.9.b (1). In addition, we can see that the $H(s)$ in fact represents a low-pass filter in shown in Fig 3.9.b (2).

## Advanced Problems

(c). Analytically calculate the CTFS for the square wave **x2**. You may find it helpful to first find a relationship between the signal $x_2(t)$ and the signal $s(t)$ defined in Eq. (3.9). Use the ten lowest frequency nonzero CTFS coefficients of **x2** to create the first 5 harmonic components individually. For example if you have the positive and negative CTFS coefficients stored in the vectors **apos_k** and **aneg_k**, respectively, you could construct the first harmonic of the input as follows

```
>> s1=apos_k(1)*exp(j*t)+aneg_k(1)*exp(-j*t);
```

Construct the signals **s1**, **s2**, **s3**, **s4**, and **s5**. Plot the sum of these signals on the same graph as the square wave **x2**.

**Solution (c):**

Analytically calculation of CTSF for $x_2(t)$:

$$a_k = \frac{1}{T}\int_T x(t)e^{-jk\omega_0 t}dt$$

$$= \frac{1}{T}\int_{-\frac{\pi}{2}}^{\frac{3\pi}{2}} x(t)e^{-jk\omega_0 t}dt$$

$$= \frac{1}{T}\left(\int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} e^{-jk\omega_0 t}dt - \int_{\frac{\pi}{2}}^{\frac{3\pi}{2}} e^{-jk\omega_0 t}dt\right)$$

$$= \frac{j}{2\pi k}\left(e^{-jk\omega_0\frac{\pi}{2}} - e^{jk\omega_0\frac{\pi}{2}} - e^{-jk\omega_0\frac{3\pi}{2}} + e^{-jk\omega_0\frac{\pi}{2}}\right)$$

$$= \frac{2\sin\left(\frac{k\pi}{2}\right)}{k\pi}$$

The relationship between $x_2(t)$ and $s(t)$:

$$a_{k(x_2)} = a_{k(s)} - e^{jk\omega_0\frac{T}{2}}a_{k(s)} = \frac{2\sin\left(\frac{k\pi}{2}\right)}{k\pi}$$

Generate the components:

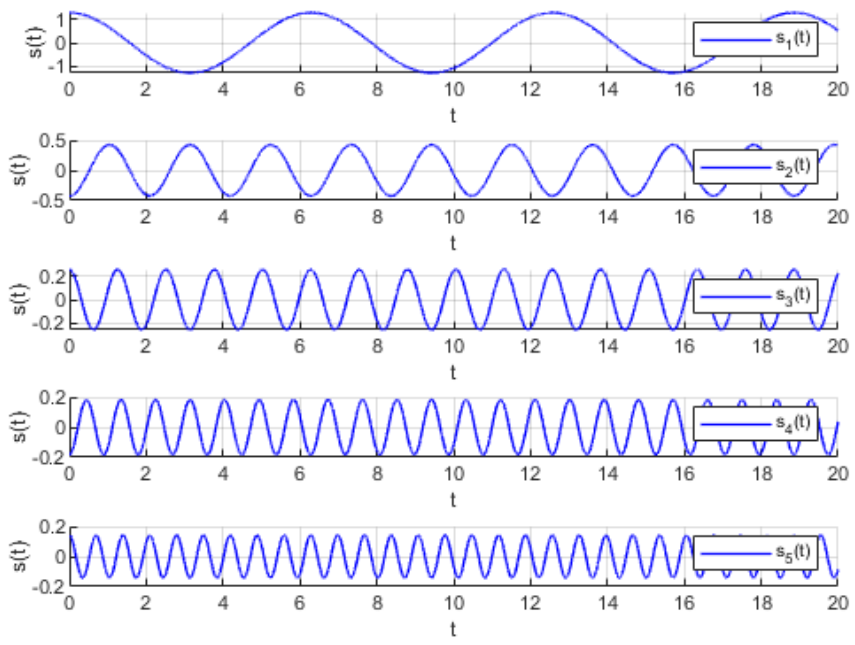



Fig 3.9.c (1)

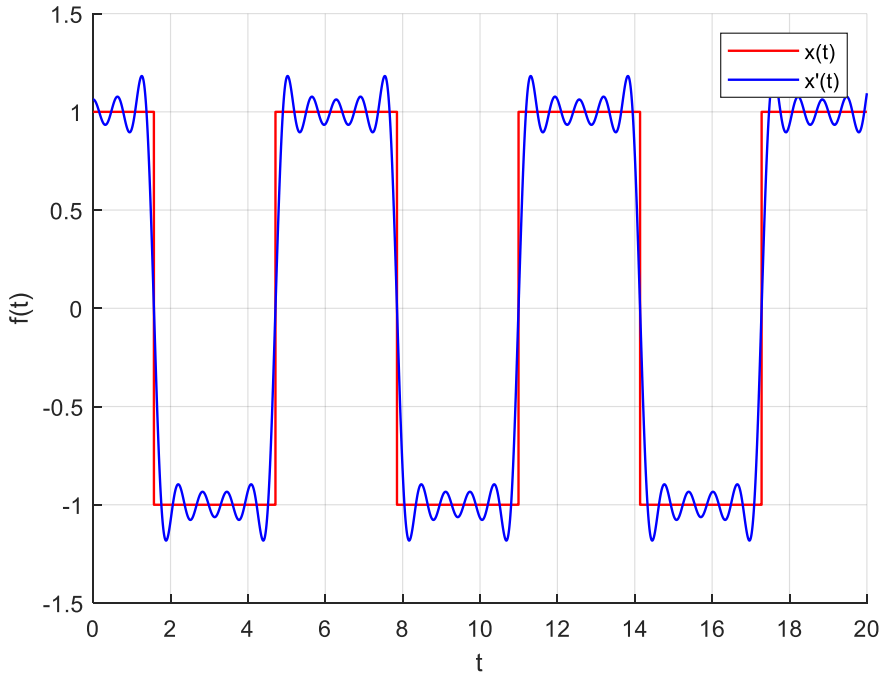



Fig 3.9.c (2)

**Analysis (c):** The components are shown in Fig 3.9.c (1). And the synthesis of $s_i(t)$ is shown in Fig 3.9.c (2).

(d). Since the circuit is linear, the response of the system to the square wave input can be calculated by finding the response of the system to each harmonic component separately, and then summing the results. Verify this by using `lsim` to find the

responses `y1`, ..., `y5` to the signals `s1`, ..., `s5` as well as the response of the system to the signal `ssum` which is the sum of the first five harmonic components, `s1` through `s5`.

**Solution (d):**

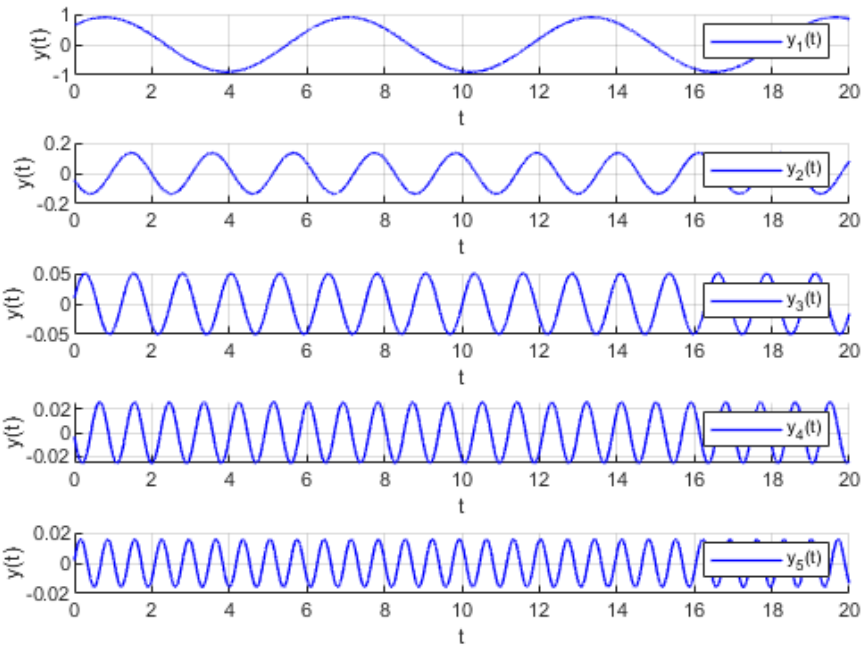



Fig 3.9.d (1)

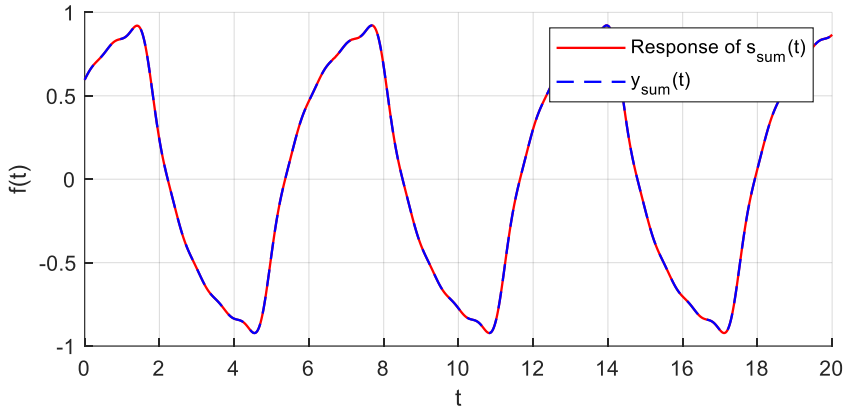



Fig 3.9.d (2)

**Analysis (d):** The $y_i(t)$ are shown in Fig 3.9.d (1). We calculate the response to $s_{sum}(t)$ and the summation of $y_i(t)$ and plot them in Fig 3.9.d (2). And we can see that they are equal.

(e). Compare the response of the system to the sum of the first 5 harmonics to the response of the system to the original square wave. Can you explain why the two responses are so similar? Hint: Consider the energy in the CTFS of `x2` as a function of the number of coefficients used in the approximation to `x2`.

**Solution (e):**

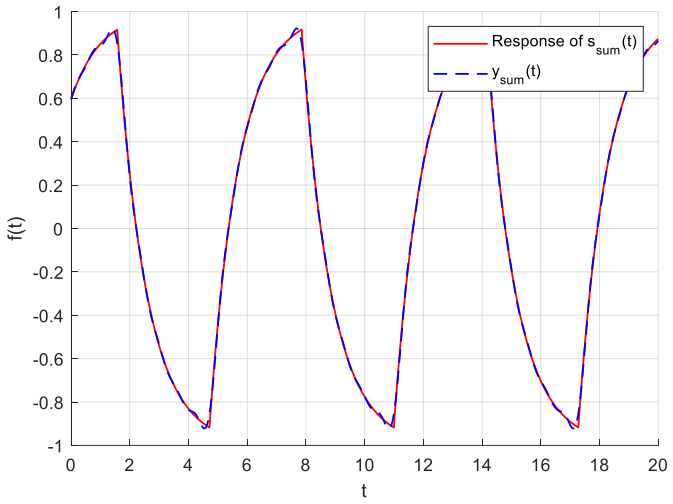



Fig 3.9.e

**Analysis (e):** The comparison is shown in Fig 3.9.e. The reason:

$$x(t) = \sum_{k=-\infty}^{+\infty} a_k e^{jk\omega_0 t}$$

$$\hat{x}_p(t) = \sum_{k=-p}^{+p} a_k e^{jk\omega_0 t}$$

And we know that when $p \to \infty$, $\hat{x}_p(t) \to x(t)$. So is the energy function. So the response of the input behaves similarly, that is when $p \to \infty$, $\hat{y}_p(t) \to y(t)$.

(f). Verify that your signals `y1`, ..., `y5` are correct by constructing each signal from the system function $H(s)$ and the CTFS for `x2`. For each, plot both the analytically determined and the simulated signals over $10 \leq t \leq 20$.

**Solution (f):**

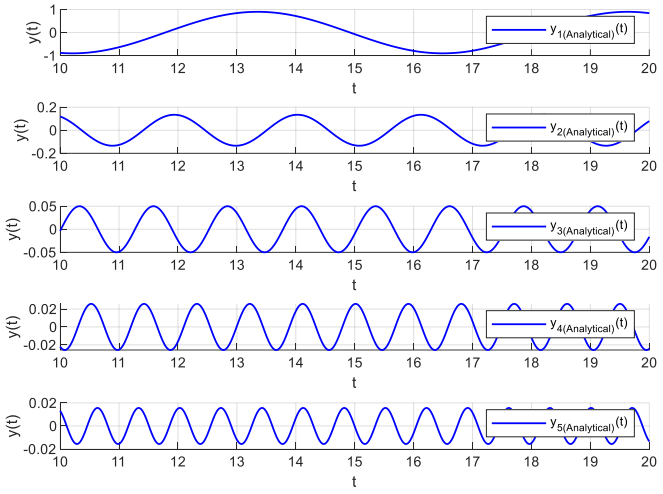



Fig 3.9.f (1)

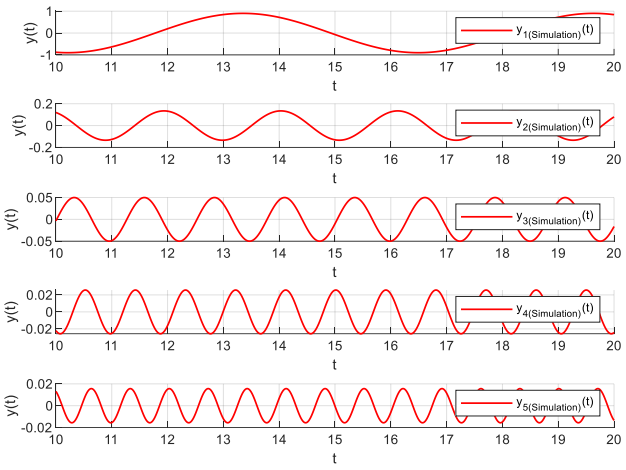



Fig 3.9.f (2)

**Analysis (f):** The results are shown in Fig 3.9.f (1) (2).

The derivation for Method CTFS:

$$H(j\omega) = \frac{1}{1 + RCj\omega} = \frac{1}{1 + j\omega}$$

So

$$H(k) = \frac{1}{1 + jk\omega_0} = \frac{1}{1 + jk}$$

Then

$$b_k = \frac{1}{1 + jk} a_k$$

## ■ 3.10 Computing the Discrete-Time Fourier Series

### The FFT Algorithm for Computing the DTFS

The DTFS for a periodic discrete-time signal with fundamental period $N$ is given by Eq. (3.2).

### Advanced Problems

(a). Argue that computing each coefficient $a_k$ requires $N + 1$ complex multiplications and $N - 1$ complex additions. Assume that the functions $e^{-jk(2\pi/N)n}$ do not require any computation. Using this result, how many operations are required to compute the DTFS for a signal with fundamental period $N$? Note that the number of operations required is independent of the particular signal $x[n]$.

**Solution (a):**

(1) Derivation seen below.

(2) $2N^2$ times.

**Analysis (a):**

(1) First, the $\Sigma$ adds up $N$ numbers which needs $N - 1$ additions; Second, in each addend we do 1 multiplication between $x[n]$ and $exp$, and there is a $\frac{1}{N}$ to be multiplied in the end. So the total number of multiplications is $N + 1$.

(2) From (1) we know calculating each $a_k$ costs $N+1+N-1=2N$ operations. And we have $N$ $a_k$ to be calculated. So the answer is $\underline{2N^2}$.

While the DTFS is useful for analyzing many signals and systems, its popularity is due in part to the existence of a fast algorithm known as the Fast Fourier Transform (FFT). Before the landmark paper by Cooley and Tukey[1] was published in 1965, the fastest algorithms used at the time for computing the DTFS coefficients required $\mathcal{O}(N^2)$ operations. ($\mathcal{O}(N^2)$ operations means that the number of complex additions and multiplications required is a polynomial in $N$ of order 2. When characterizing the growth in computational complexity of an algorithm, the lower order terms can be ignored, since they become negligible for large $N$.) The FFT algorithm proposed by Cooley and Tukey, however, requires only $\mathcal{O}(N \log N)$ operations. For large $N$, the computational savings can be tremendous; e.g., for $N = 800$, compare $N^2 = 1 \times 10^6$ with $N \log N = 3 \times 10^3$. The difference is three orders of magnitude. The FFT essentially allows Fourier analysis to be applied to signals with very large fundamental periods. In the following problem, you will plot the number of operations required by the FFT and compare them with an $\mathcal{O}(N^2)$ algorithm.

For the next two parts, you should assume $x[n]$ has fundamental period $N$ and takes on the values $x[n] = (0.9)^n$ on the interval $0 \le n \le N-1$.

(b). If you have not already done the Advanced Problem in Exercise 3.5 writing **dtfs**, do so now. You will compare the amount of computation this algorithm requires with the amount required by **fft**. You can measure the number of operations used by **dtfs** to implement Eq. (3.2) by using the internal **flops** (floating point operations, i.e., additions and multiplications) counter of MATLAB as follows:

```
>> x = (0.9).^[0:N-1]; % create one period of x[n]
>> flops(0);  % set MATLAB's internal computation counter to 0
>> X = dtfs(x,0); % Store the DTFS of x[n] in X
>> c = flops;  % Store the number of operations in c
```

Find c for computing X using **dtfs** for $N = 8, 32, 64, 128$, and 256. Save these values in the vector **dtfscomps**.

**Solution (b):**

*flops* is not available in new versions. Use *tic* and *toc*.

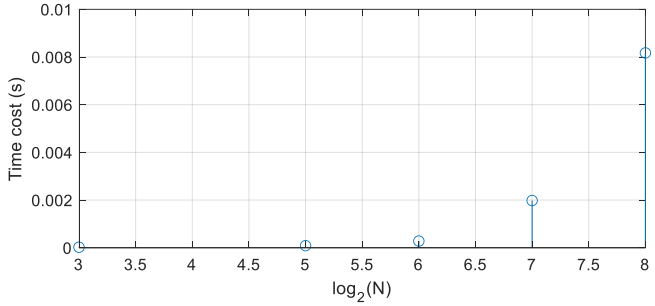



Fig 3.10.b

**Analysis (b):** In Fig 3.10.b, we can see the time cost increases as N increases. In order to guarantee the accuracy we ran the algorithm for 100 times and calculated the average time.

(c). Now, compute the DTFS coefficients of $x[n]$ for $N = 8, 32, 64, 128$, and 256 using **fft** as shown in Tutorial 3.1. Use **flops** to find the number of operations used for each value of $N$ and store these values in the vector **fftcomps**. Plot **dtfscomps** and **fftcomps** versus $N$ using **loglog**. How does the number of operations required by **fft** compare to that required by **dtfs**, particularly for large values of N?

**Solution (c):**

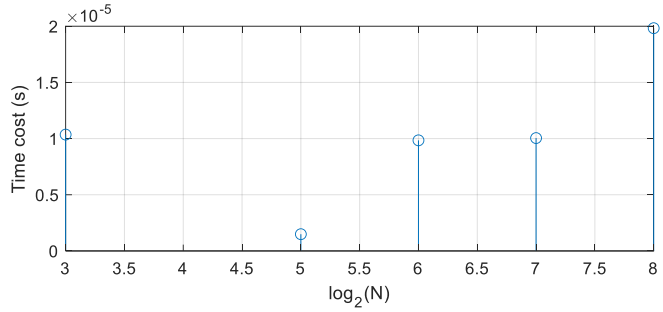



Fig 3.10.c (1)

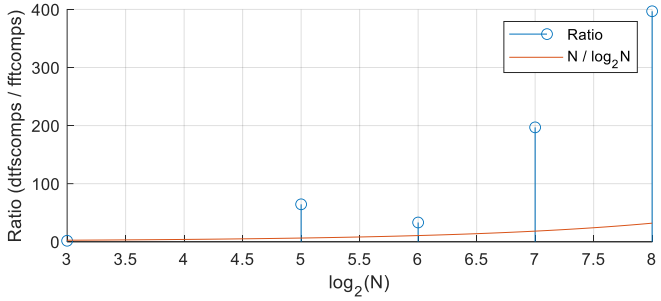



Fig 3.10.c (2)

**Analysis (c):** In Fig 3.10.c (1), we can see all the time cost is very low. In order to guarantee the accuracy we ran the algorithm for 100000 times and calculated the average time. But for some reason the ratio in Fig 3.10.c (2) deviated, maybe the time constant.

If computational savings is defined as the ratio of the operations required by the slow algorithm to those of the fast algorithm, note that the savings provided by the FFT increase as $N$ increases (at the rate of roughly $N/\log N$).

## Periodic Convolution with the FFT

In many applications, LTI systems are implemented using periodic convolution. In the following problems, you will implement the periodic convolution of two discrete-time signals using two different methods, one of which takes advantage of the computational savings provided by the FFT. The periodic convolution of two periodic discrete-time signals $x[n]$ and $h[n]$, both with fundamental period $N$, is

$$y[n] = \sum_{r=0}^{N-1} x[r]h[n-r]\,. \tag{3.11}$$

(d). What is the fundamental period, $N_y$, of $y[n]$? Argue that directly implementing the periodic convolution according to Eq. (3.11) requires $\mathcal{O}(N^2)$ operations (additions and multiplications). Remember that computing one period of $y[n]$ is sufficient to characterize the entire signal.

**Solution (d):**

(1) $N_y = N$.

(2) Derivation seen below.

**Analysis (d):**

(1) We have $y[n] = \sum_{r=0}^{N-1} x[r]h[n-r] = \sum_{r=0}^{N-1} x[r]h[n+N-r] = y[n+N]$. So

$N$ is a period of $y[n]$. And $N_y$ cannot be smaller than the fundamental period of $x[n]$ and $y[n]$, $N$. So $\underline{N_y = N}$.

(2) Calculating a certain $y[n_0]$ requires $N$ multiplications and $N-1$ additions. And to calculate one period of $y[n]$ we need $N(N+N-1) = 2N^2 - N$ operations, which indicates the time complexity $\underline{O(N^2)}$.

(e). Assume both $x[n]$ and $h[n]$ have fundamental period $N = 40$, and are given by $x[n] = (0.9)^n$ and $h[n] = (0.5)^n$ over the interval $0 \le n \le N-1$. Compute the periodic convolution of $x[n]$ with $h[n]$ and plot $y[n]$ for $0 \le n \le N_y - 1$. Store the number of operations, given by `flops`, required to implement the convolution in `f40c`. Remember to set `flops(0)` after creating `x` and `h`, the vectors representing $x[n]$ and $h[n]$. Hint:

To implement the periodic convolution, first store $x[n]$ and $h[n]$ over the interval $0 \le n \le N-1$ in the row vectors `x` and `h`, respectively. Set `flops(0)` and then use `conv([x x],h)`. The periodic convolution can be extracted from a portion of this signal.

**Solution (e):**

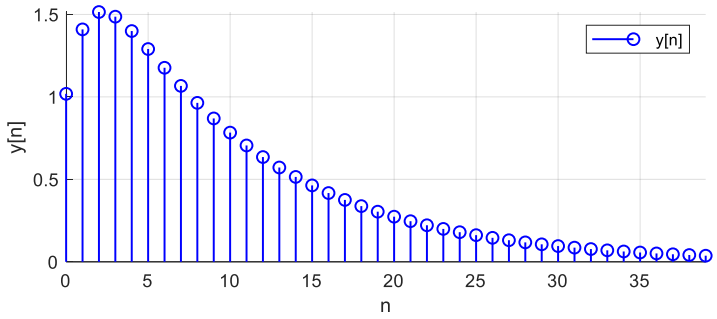



Fig 3.10.e

$\underline{f40c = 0.13766ms}$

**Analysis (e):** In Fig 3.10.e is the result when $N = 40$ using *conv*, and $f40c$ is shown above.

(f). Repeat Part (e) for $N = 80$, again plotting a period of $y[n]$ and storing the number of operations in `f80c`.

**Solution (f):**

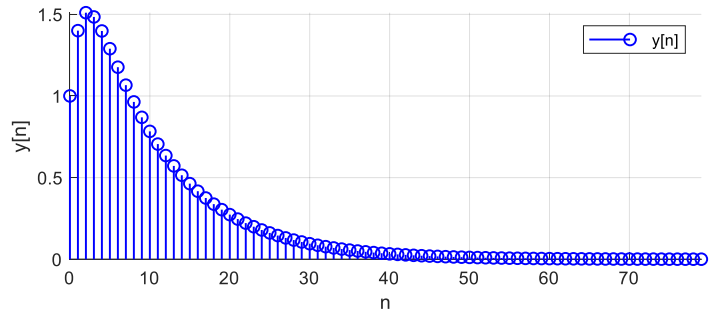



Fig 3.10.f

$\underline{f80c = 0.14617ms}$

**Analysis (f):** In Fig 3.10.f is the result when $N = 80$ using *conv*, and $f80c$ is shown above.

(g). Assume $x[n]$ and $h[n]$ are defined as in Part (e). Use the periodic convolution property of the DTFS to implement the periodic convolution[2]. Namely, compute the DTFS coefficients of both $x[n]$ and $h[n]$ using `fft` as described in Tutorial 3.1. Then, use the periodic convolution property to form the DTFS coefficients of $y[n]$. Finally, synthesize $y[n]$ from the DTFS coefficients using `ifft`. The `ifft` algorithm is nearly identical to the FFT, and also requires $\mathcal{O}(N \log N)$ operations for a signal with fundamental period $N$. To check the validity of your implementation, plot $y[n]$ for $0 \leq n \leq N_y - 1$ and compare this signal to that computed in Part (e). Remember, as discussed in Tutorial 3.1, the signal $y[n]$ might have a small imaginary component due to numerical round-off errors. Store the total number of operations required to compute $y[n]$ in `f40f`, and remember to set `flops(0)` after creating the representations of $x[n]$ and $h[n]$ in MATLAB.

**Solution (g):**

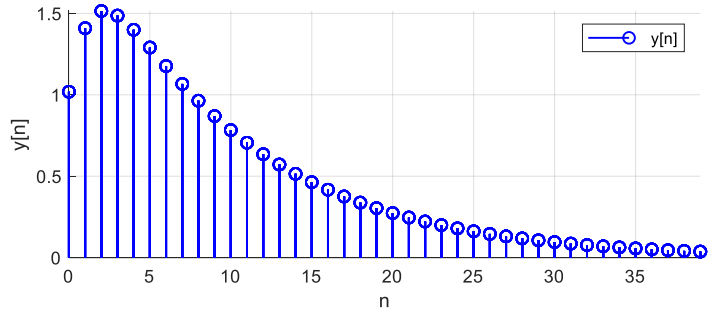



Fig 3.10.g

$\underline{f40f = 0.12565ms}$

**Analysis (g):** In Fig 3.10.g is the result when $N = 40$ using *fft* and *ifft*, and $f40f$ is shown above.

(h). Repeat Part (g) for $N = 80$, again plotting a period of $y[n]$ and storing the number of operations in `f80f`. Again check the validity of your implementation by comparing $y[n]$ with that computed in Part (f).

**Solution (h):**

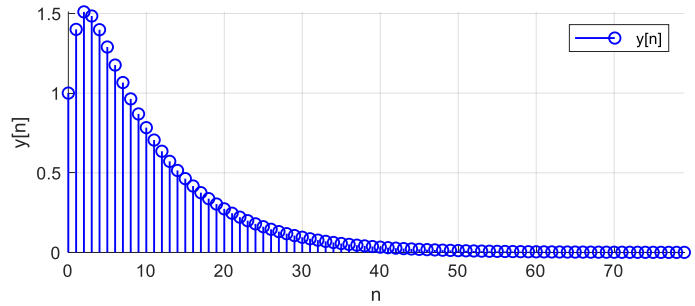



Fig 3.10.h

$\underline{f80f = 0.19749ms}$

**Analysis (h):** In Fig 3.10.h is the result when $N = 80$ using *fft* and *ifft*, and $f80f$ is shown above. **By the way, some reasons which we guess that may be associated to the optimization measures on matrix operations in MATLAB made $f80c < f80f$. This situation will not happen when $N$ grows large enough, which can be seen in (i).**

(i). Compute the ratios of `f40c` to `f40f` and `f80c` to `f80f`. How do these ratios compare for $N = 40$ and $N = 80$? Which method of convolution is more efficient for each value of $N$? Which method would you choose for $N > 80$? Justify your answer.

**Solution (i):**

$$\frac{f40c}{f40f} = 1.096,\ \frac{f80c}{f80f} = 0.740^{*}$$

* *The reason for* $0.740 < 1$ *has been stated in* ***Analysis (g).***

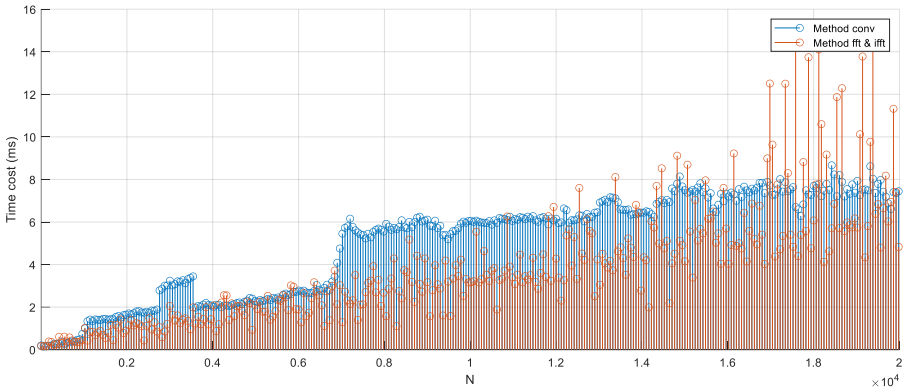



Fig 3.10.i (1) Time cost comparison for different Ns (Trial 1, up to 20000)

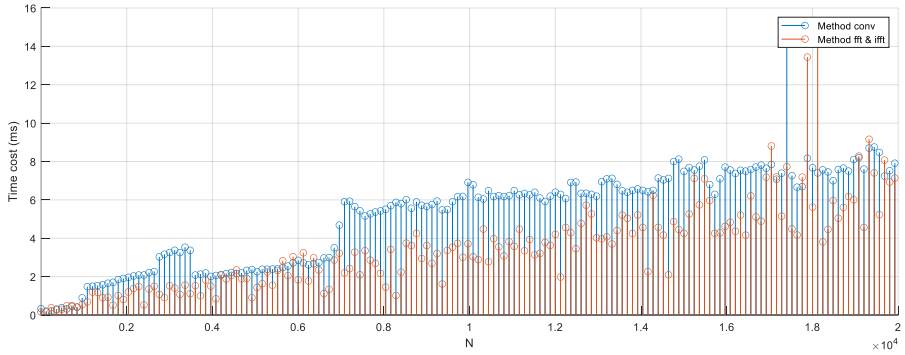



Fig 3.10.i (2) Time cost comparison for different Ns (Trial 2, up to 20000)

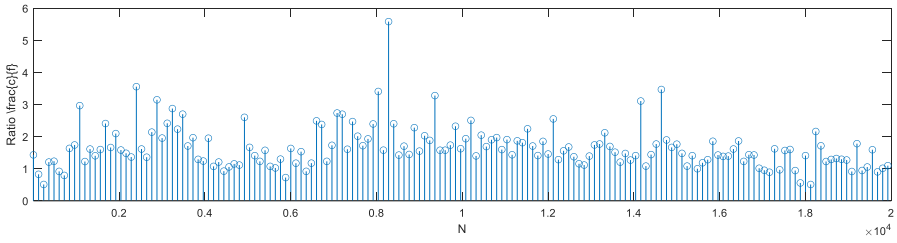



Fig 3.10.i (3) Ratio $\frac{c}{f}$ for different Ns (Trial 3, up to 20000)

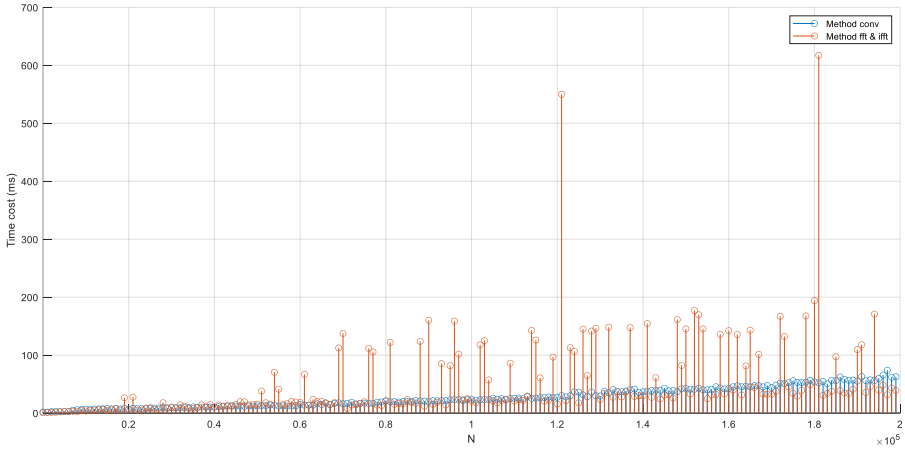


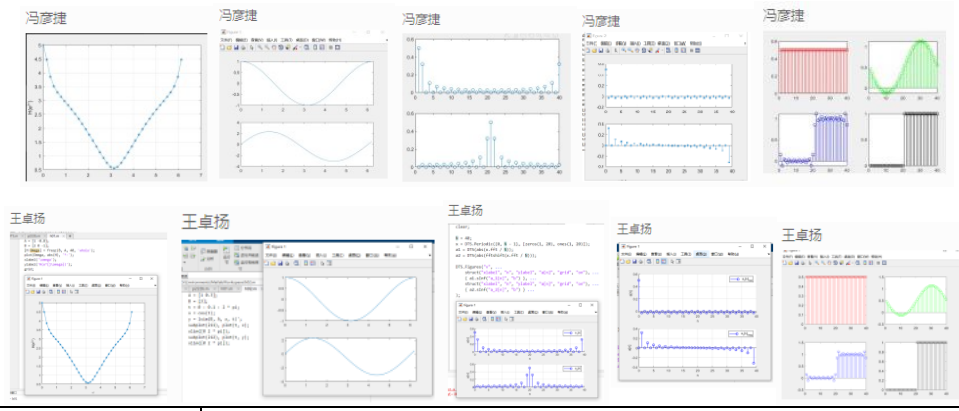

Fig 3.10.i (4) Time cost comparison for different Ns (Trial 4, up to 200000)

**Analysis (i):** All the questions can be answered by Fig 3.10.i, from which we see that as the $N$ increases, we prefer to choose the faster algorithm using *fft* and *ifft*. But from Fig 3.10.i (4) we think *fft* and *ifft* are not so stable in MATLAB.

# Experience

1. We practiced more on how to calculate FS, FT and synthesis signals.
2. We upgraded the class *DTS* to process more complex situations.
3. In the future the type security may be implemented.
4. We program more efficiently than before.
5. We learnt about time complexity and now we know how to calculate the time cost.
6. We obtained more than all these things……

*** The screenshots in lab class:**




| **Score** | 99 |
|---|---|

# Appendix

**Util Class for DT Signal Processing – DTS.m**

*For the sake of simplicity, we developed a util class to help us with common tasks, including DT signal generating and storing, plotting, and doing basic operations like plus, minus, power, convolution, etc. And this time we updated it to meet the new demands.*

*Here's the code in DTS.m:*

```
classdef DTS
    %{
        [ 一维离散时间信号类 ]
        Latest: 20220331 by Gralerfics
        属性
            domain - 定义域，行向量
            value - 值，与 domain 等长行向量
            sample - 采样率，单位长度内样本数量
                * Warning: 采样率不为 1 时，构造函数信号注意将自变量换元为原来的 1 / sample;
        成员方法
            DTS(domainArg, valueArg, [sampleArg]) - 构造方法
                * 仅一个参数（valueArg）时默认 domainArg 为 0 : length(valueArg) - 1; 若该参数为 DTS 类型则表复制功能.
                * 若直接给出 domainArg 向量，单位时间长度为 1 / sample，即两向量长度要相同.
                * 若采用 {l, r} 作为 domainArg，则单位时间长度为 1，自动生成采样总数长度的向量.
                  而 length(valueArg) 需为 (r - l) * sampleArg + 1.
                * 若 valueArg 长度不合要求，但为一个标量，则用该量填充到合法长度.
            filter(xArgs, yArgs) - 单目 Filter
            lsim(xArgs, yArgs) - 单目 Lsim
            dtfs() - 单目 Dtfs
            idtfs() - 单目 IDtfs
            fft() - 单目 FFT
            ifft() - 单目 IFFT
            stretch(a) - 将 a * n 代入 n, a 为整且不为 0
            shift(b) - 将 n + b 代入 n, b 为整
                * 以上两个操作的单位时间长度为 1.
                * 例如: x[n] --shift(2)-> x[n + 2] --stretch(2)-> x[2n + 2].
            func(s) - 将 value 传入指定函数并将返回值封装为新信号
                * 形如 func(x[n]). 若要 func(n) 请借助 Identity 生成 x[n] = n.
            cut(s) - 截取信号（截定义域）
                * 单位时间长度为 1.
            toPeriod(x) - 单目 Toperiod
            sInf([legend], [color], [style], [type]) - 生成信号绘制信息结构体
            以及一系列算符重载
                plus(+) - 叠加
                minus(-) - 削减
                uminus(-) - 取反
```

```
            times(.*) - 按元素乘
            mtimes(*) - 卷积
            rdivide(./) - 右按元素除
            power(.^) - 按元素求幂
            mpower(^) - 连续卷积
        静态方法
            Step(domainArg, [sample]) - 单位阶跃函数
            Impulse(domainArg, [sample]) - 单位脉冲函数
            Identity(domainArg, [sample]) - 单位函数, y = x
            Periodic(domainArg, valueArg) - 生成离散周期函数
            Toperiod(x) - 将传入的信号视为一个周期, 将其归一化至 [0, N - 1]
            Conv(a, b) - 卷积信号 a 与 b
            Pconv(a, b) - 周期卷积 a 与 b, 返回结果的 [0, N - 1] 周期
            Pconvfft(a, b) - FFT 加速的周期卷积
            Filter(B, A, x) - 差分方程系统响应（B, A 升序; 即 y[n - 0] 优先）
            Lsim(B, A, x) - 微分方程系统响应（B, A 降序; 即 y(k)(n) 优先）
            GetE(N) - 计算 E 矩阵
            Dtfs(x) - Discrete Fourier Series, 传入一个周期长度信号, 返回信号, 会
按定义域调节
            Idtfs(x) - Synthesis, 传入 N 长度信号, 返回向量, 会按定义域调节
            Fft(x) - fft(x), Fast Fourier Transform, 传入一个周期的信号, 从 0 开
始
            Ifft(a) - ifft(a), 返回一个周期的信号, 从 0 开始
            Freqz(xArgs, yArgs, S) - 差分方程系统频率响应
            Stems(Dims, Sigs) - 绘图
                Dims - 结构体, 可含 xlim, ylim, xlabel, ylabel, title, grid 域
                Sigs - Cell 数组, 每个元素为结构体, 必含 signal 域, 可含 legend,
style, color 域
            Figures(type, Dims1, Sigs1, ...) - 多子图排列绘图
    %}
    properties
        domain
        value
        sample
    end
    methods (Access = private)
        function [ra, rb] = checkTwo(a, b)
            ra = a;
            rb = b;
            if class(ra) == "double"; ra = DTS(rb.domain, ra * ones(1,
length(rb.domain)), rb.sample); end
            if class(rb) == "double"; rb = DTS(ra.domain, rb * ones(1,
length(ra.domain)), ra.sample); end
```

```
            if ra.sample ~= rb.sample; error("Non-identical sampling rates!");
end
        end
        function y = mergeTwo(a, b)
            yDomain = min(a.domain(1), b.domain(1)) : max(a.domain(end),
b.domain(end));
            yValue = zeros(1, length(yDomain));
            y = DTS(yDomain, yValue, a.sample);
        end
    end
    methods
        function obj = DTS(varargin)
            if nargin == 1
                if class(varargin{1}) == "DTS"
                    obj = varargin{1};
                else
                    obj = DTS(0 : length(varargin{1}) - 1, varargin{1});
                end
            else
                if nargin > 2
                    obj.sample = varargin{3};
                else
                    obj.sample = 1;
                end
                if class(varargin{1}) == "cell"
                    obj.domain = varargin{1}{1} * obj.sample : varargin{1}{2} *
obj.sample;
                else
                    obj.domain = varargin{1};
                end
                if nargin > 1
                    if length(varargin{2}) == length(obj.domain)
                        obj.value = varargin{2};
                    elseif length(varargin{2}) == 1
                        obj.value = varargin{2}(1) .* ones(1,
length(obj.domain));
                    else
                        error('Wrong interval of value vector!');
                    end
                else
                    obj.value = zeros(1, length(obj.domain));
                end
            end
        end
```

```
        function y = filter(obj, xArgs, yArgs)
            y = DTS.Filter(xArgs, yArgs, obj);
        end
        function y = lsim(obj, xArgs, yArgs)
            y = DTS.Lsim(xArgs, yArgs, obj);
        end
        function a = dtfs(obj)
            a = DTS.Dtfs(obj);
        end
        function x = idtfs(obj)
            x = DTS.Idtfs(obj);
        end
        function a = fft(obj)
            a = DTS.Fft(obj);
        end
        function x = ifft(obj)
            x = DTS.Ifft(obj);
        end
        function y = stretch(obj, a)
            yDomain = obj.domain(mod(obj.domain, a) == 0) ./ a;
            yValue = obj.value(mod(obj.domain, a) == 0);
            if a < 0
                yDomain = fliplr(yDomain);
                yValue = fliplr(yValue);
            end
            y = DTS(yDomain, yValue, obj.sample);
        end
        function y = shift(obj, b)
            y = DTS((obj.domain(1) - b * obj.sample) : (obj.domain(end) - b *
obj.sample), obj.value, obj.sample);
        end
        function y = func(obj, s)
            yDomain = obj.domain;
            yValue = eval(s + "(obj.value);");
            y = DTS(yDomain, yValue, obj.sample);
        end
        function y = cut(obj, s)
            if class(s) == "cell"
                s = s{1} * obj.sample : s{2} * obj.sample;
            else
                s = s(1) * obj.sample : s(end) * obj.sample;
            end
            yDomain = s;
            l = max(obj.domain(1) - s(1), 0);
```

```
            r = max(s(end) - obj.domain(end), 0);
            yValue = [zeros(1, l) obj.value(ismember(obj.domain, s)) zeros(1,
r)];
            y = DTS(yDomain, yValue, obj.sample);
        end
        function rst = sInf(obj, varargin)
            rst.signal = obj;
            if nargin > 1; if varargin{1} ~= ""; rst.legend = varargin{1}; end;
end
            if nargin > 2; if varargin{2} ~= ""; rst.color = varargin{2}; end;
end
            if nargin > 3; if varargin{3} ~= ""; rst.style = varargin{3}; end;
end
            if nargin > 4; if varargin{4} ~= ""; rst.type = varargin{4}; end;
end
        end
        function y = toPeriod(obj)
            y = DTS.Toperiod(obj);
        end
        function y = plus(a, b)
            [a, b] = checkTwo(a, b);
            y = mergeTwo(a, b);
            s = find(y.domain == a.domain(1)); i = s : (s + length(a.value) -
1); y.value(i) = y.value(i) + a.value;
            s = find(y.domain == b.domain(1)); i = s : (s + length(b.value) -
1); y.value(i) = y.value(i) + b.value;
        end
        function y = minus(a, b)
            y = plus(a, uminus(b));
        end
        function y = uminus(a)
            yDomain = a.domain;
            yValue = -a.value;
            y = DTS(yDomain, yValue, a.sample);
        end
        function y = times(a, b)
            [a, b] = checkTwo(a, b);
            y = mergeTwo(a, b);
            s = find(y.domain == a.domain(1)); i = s : (s + length(a.value) -
1); y.value(i) = y.value(i) + a.value;
            s = find(y.domain == b.domain(1)); i = s : (s + length(b.value) -
1); y.value(i) = y.value(i) .* b.value;
        end
        function y = mtimes(a, b)
```

```
            if class(a) == "double" || class(b) == "double"
                y = a .* b;
            else
                y = DTS.Conv(a, b);
            end
        end
        function y = rdivide(a, b)
            [a, b] = checkTwo(a, b);
            y = mergeTwo(a, b);
            s = find(y.domain == a.domain(1)); i = s : (s + length(a.value) -
1); y.value(i) = y.value(i) + a.value;
            s = find(y.domain == b.domain(1)); i = s : (s + length(b.value) -
1); y.value(i) = y.value(i) ./ b.value;
        end
        function y = power(a, b)
            [a, b] = checkTwo(a, b);
            y = mergeTwo(a, b);
            s = find(y.domain == a.domain(1)); i = s : (s + length(a.value) -
1); y.value(i) = y.value(i) + a.value;
            s = find(y.domain == b.domain(1)); i = s : (s + length(b.value) -
1); y.value(i) = y.value(i) .^ b.value;
        end
        function y = mpower(a, b)
            y = DTS(a.domain, a.value, a.sample);
            for i = 2 : b
                y = y * a;
            end
        end
    end
    methods (Static)
        function u = Step(varargin)
            s = 1;
            if nargin > 1; s = varargin{2}; end
            u = DTS(varargin{1}, 0, s);
            u.value = u.domain >= 0;
        end
        function d = Impulse(varargin)
            s = 1;
            if nargin > 1; s = varargin{2}; end
            d = DTS(varargin{1}, 0, s);
            d.value(d.domain == 0) = s;
        end
        function y = Identity(varargin)
            s = 1;
```

```
            if nargin > 1; s = varargin{2}; end
            y = DTS(varargin{1}, 0, s);
            y.value = y.domain ./ s;
        end
        function y = Periodic(domainArg, valueArg)
            y = DTS(domainArg, valueArg(mod(domainArg{1} : domainArg{2},
length(valueArg)) + 1));
        end
        function y = Toperiod(x)
            y = DTS(circshift(x.value, x.domain(1)));
        end
        function z = Conv(x, y)
            if x.sample ~= y.sample; error("Non-identical sampling rates!"); end
            zDomain = (x.domain(1) + y.domain(1)) : (x.domain(end) +
y.domain(end));
            zValue = conv(x.value, y.value) ./ x.sample;
            z = DTS(zDomain, zValue, x.sample);
        end
        function z = Pconv(x, y)
            if length(x.domain) ~= length(y.domain); error("Period should be
equivalent!"); end
            N = length(x.domain);
            z = DTS.Toperiod(x) * (DTS.Toperiod(y) + DTS.Toperiod(y).shift(N));
            z = z.cut(0 : N - 1);
        end
        function z = Pconvfft(x, y)
            if length(x.domain) ~= length(y.domain); error("Period should be
equivalent!"); end
            N = length(x.domain);
            az = DTS.Toperiod(x).fft .* DTS.Toperiod(y).fft ./ N;
            z = DTS(az).ifft * N;
        end
        function y = Filter(xArgs, yArgs, x)
            y = DTS(x.domain, filter(xArgs, yArgs, x.value), x.sample);
        end
        function y = Lsim(xArgs, yArgs, x)
            y = DTS(x.domain, lsim(xArgs, yArgs, x.value, x.domain / x.sample)',
x.sample);
        end
        function E = GetE(N)
            E = exp(2 * pi * 1j / N) .^ reshape(fix((0 : N * N - 1) / N) .* mod((0 :
N * N - 1), N), N, N);
        end
        function a = Dtfs(x)
```

```
            if x.sample ~= 1; error('This can be applied to DT Signals (sample
= 1)!'); end
            N = length(x.domain);
            E = DTS.GetE(N);
            a = transpose(E' * transpose(circshift(x.value, x.domain(1))) / N);
        end
        function x = Idtfs(a)
            N = length(a.domain);
            A = transpose(a.value) * ones(1, N);
            NK = (a.domain' * ones(1, N)) .* (ones(1, N)' * (0 : N - 1));
            x = DTS({0, N - 1}, sum(A .* exp(1j .* NK .* (2 * pi / N))));
        end
        function a = Fft(x)
            a = fft(x.value);
        end
        function x = Ifft(a)
            x = DTS(ifft(a.value));
        end
        function [h, w] = Freqz(xArgs, yArgs, S)
            [h, w] = freqz(xArgs, yArgs, S, "whole");
        end
        function Stems(Dims, Sigs)
            hold on;
            for f = ["xlim", "ylim", "xlabel", "ylabel", "title", "grid"]
                if isfield(Dims, f)
                    eval(f + "(Dims." + f + ");");
                end
            end
            handles = [];
            legends = [];
            for i = 1 : length(Sigs)
                sig = Sigs{i};
                s = sig.signal;
                args = {s.domain ./ s.sample, s.value};
                if isfield(sig, 'style')
                    args = [args, {sig.style}];
                end
                if isfield(sig, 'color')
                    args = [args, 'color', {sig.color}];
                end
                args = [args, {'lineWidth', 1}];
                h = nan;
                if isfield(sig, 'type')
                    switch sig.type
```

```
                    case "plot"
                        h = plot(args{:});
                    otherwise
                        h = stem(args{:});
                end
            else
                h = stem(args{:});
            end
            if isfield(sig, 'legend')
                handles = [handles, h];
                legends = [legends, sig.legend];
            end
        end
        if ~isempty(legends)
            legend(handles, legends');
        end
    end
    function Figures(varargin)
        n = (nargin - 1) / 2;
        switch varargin{1}
            case "v"
                p = 100 * n + 10;
            case "h"
                p = 100 + n * 10;
        end
        for i = 1 : n
            subplot(p + i);
            DTS.Stems(varargin{i * 2}, varargin{i * 2 + 1});
        end
    end
end
end
```

**Programs for 3.5**

**(c)**

```
clear;

N = 5;
a = DTS([-2 -1 0 1 2], [exp(-1j * pi / 4), 2 * exp(1j * pi / 3), 1, 2 * exp(-1j
* pi / 3), exp(1j * pi / 4)]);
x = a.idtfs;

DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "real(x[n])", "grid", "on"), ...
```

```
    { x.func('real').sInf("x[n]", "r") }, ...
    struct("xlabel", "t", "ylabel", "imag(x[n])", "grid", "on"), ...
    { x.func('imag').sInf("x[n]_{imag}", "b") } ...
);
```

**(d)**

```
clear;

x1 = DTS.Periodic({-10, 70}, 1);
x2 = DTS.Periodic({-10, 70}, [ones(1, 8), zeros(1, 8)]);
x3 = DTS.Periodic({-10, 70}, [ones(1, 8), zeros(1, 24)]);

v = [0 63];
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "x_1[n]", "xlim", v, "grid", "on"), ...
    { x1.sInf("x_1[n]", "r", ".") }, ...
    struct("xlabel", "t", "ylabel", "x_2[n]", "xlim", v, "grid", "on"), ...
    { x2.sInf("x_2[n]", "g", ".") }, ...
    struct("xlabel", "t", "ylabel", "x_3[n]", "xlim", v, "grid", "on"), ...
    { x3.sInf("x_3[n]", "b", ".") } ...
);
```

**(e)**

```
clear;

a1 = DTS(DTS(ones(1, 8)).dtfs);
a2 = DTS(DTS([ones(1, 8), zeros(1, 8)]).dtfs);
a3 = DTS(DTS([ones(1, 8), zeros(1, 24)]).dtfs);

figure(1);
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "a_1[n]", "xlim", [0 7], "grid", "on"), ...
    { a1.func('real').sInf("a_1[n]_{real}", "", "."),
a1.func('imag').sInf("a_1[n]_{imag}", "", ".") }, ...
    struct("xlabel", "t", "ylabel", "a_2[n]", "xlim", [0 15], "grid", "on"), ...
    { a2.func('real').sInf("a_2[n]_{real}", "", "."),
a2.func('imag').sInf("a_2[n]_{imag}", "", ".") }, ...
    struct("xlabel", "t", "ylabel", "a_3[n]", "xlim", [0 31], "grid", "on"), ...
    { a3.func('real').sInf("a_3[n]_{real}", "", "."),
a3.func('imag').sInf("a_3[n]_{imag}", "", ".") } ...
);

figure(2);
DTS.Figures("v", ...
```

```
    struct("xlabel", "t", "ylabel", "a_1[n]", "xlim", [0 7], "grid", "on"), ...
    { a1.func('abs').sInf("|a_1[n]|") }, ...
    struct("xlabel", "t", "ylabel", "a_2[n]", "xlim", [0 15], "grid", "on"), ...
    { a2.func('abs').sInf("|a_2[n]|") }, ...
    struct("xlabel", "t", "ylabel", "a_3[n]", "xlim", [0 31], "grid", "on"), ...
    { a3.func('abs').sInf("|a_3[n]|") } ...
);
```

**(f)**

```
clear;

a3 = DTS(circshift(DTS([ones(1, 8), zeros(1, 24)]).dtfs, 15)).shift(15);
x3_2 = a3.cut(-2 : 2).cut(-15 : 16).idtfs;
x3_8 = a3.cut(-8 : 8).cut(-15 : 16).idtfs;
x3_12 = a3.cut(-12 : 12).cut(-15 : 16).idtfs;
x3_all = a3.idtfs;

v = [0 31];
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "x_{3(2)}[n]", "xlim", v, "grid", "on"), ...
    { x3_2.func('real').sInf("x_{3(2)}[n]") }, ...
    struct("xlabel", "t", "ylabel", "x_{3(8)}[n]", "xlim", v, "grid", "on"), ...
    { x3_8.func('real').sInf("x_{3(8)}[n]") }, ...
    struct("xlabel", "t", "ylabel", "x_{3(12)}[n]", "xlim", v, "grid",
"on"), ...
    { x3_12.func('real').sInf("x_{3(12)}[n]") }, ...
    struct("xlabel", "t", "ylabel", "x_{3(all)}[n]", "xlim", v, "grid",
"on"), ...
    { x3_all.func('real').sInf("x_{3(all)}[n]") } ...
);
```

**(g)**

```
clear;

a3 = DTS(circshift(DTS([ones(1, 8), zeros(1, 24)]).dtfs, 15)).shift(15);
x3_all = a3.idtfs;

v = [0 31];
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "x_{3(all)}[n]", "xlim", v, "grid",
"on"), ...
    { x3_all.func('imag').sInf("x_{3(all)}[n]") } ...
);
```

**(h)**

```
clear;

a3 = DTS(circshift(DTS([ones(1, 8), zeros(1, 24)]).dtfs, 15)).shift(15);

v = [0 31];
Arg = {"v"};
for I = 2 : 2 : 16
    Arg = [Arg, {
        struct("xlabel", "t", "ylabel", "x_{3(" + num2str(I) + ")}[n]", "xlim",
v, "grid", "on"), ...
        { a3.cut(-I : I).cut(-15 : 16).idtfs.func('real').sInf("x_{3(" +
num2str(I) + ")}[n]") }
    }];
end
DTS.Figures(Arg{:});
```

**(dtfs.m)**

```
function a = dtfs(x, n_init)
    a = DTS({n_init, n_init + length(x) - 1}, x).dtfs;
end
```

**Programs for 3.8**

**(b)**

```
clear;

a1 = [1, -0.8]; b1 = [1];
a2 = [1, 0.8]; b2 = [1];
S = 1024;
[h1, w1] = freqz(b1, a1, S, 'whole');
[h2, w2] = freqz(b2, a2, S, 'whole');

DTS.Figures("v", ...
    struct("xlabel", "\omega", "ylabel", "H_1(\omega)", "xlim", [0, 2 * pi],
"grid", "on"), ...
    { DTS(w1, abs(h1)).sInf("H_1(\omega)", "r", "", "plot") }, ...
    struct("xlabel", "\omega", "ylabel", "H_2(\omega)", "xlim", [0, 2 * pi],
"grid", "on"), ...
    { DTS(w2, abs(h2)).sInf("H_2(\omega)", "b", "", "plot") } ...
);
```

**(c)**

```
clear;
```

```
w_0 = 2 * pi / 20;
a_x = DTS(w_0 * (0 : 19), [0, 3 / 4, zeros(1, 7), -1 / 2, 0, - 1 / 2, zeros(1,
7), 3 / 4]);

DTS.Figures("v", ...
    struct("xlabel", "k", "ylabel", "a_k", "xlim", [0, 2 * pi], "ylim", [-0.75,
1.5], "grid", "on"), ...
    { a_x.sInf("a_k", "r") } ...
);
```

**(d)**

```
clear;

a_x = DTS([0, 3 / 4, zeros(1, 7), -1 / 2, 0, - 1 / 2, zeros(1, 7), 3 / 4]);
x_20 = a_x.idtfs.func('real');
x = DTS.Periodic({-20, 99}, x_20.value);

DTS.Figures("v", ...
    struct("xlabel", "n", "ylabel", "x_{20}[n]", "grid", "on"), ...
    { x_20.sInf("x_{20}[n]", "b") }, ...
    struct("xlabel", "n", "ylabel", "x[n]", "grid", "on"), ...
    { x.sInf("x[n]", "r", ".") } ...
);
```

**(e)**

```
clear;

a1 = [1, -0.8]; b1 = [1];
a2 = [1, 0.8]; b2 = [1];
a_x = DTS([0, 3 / 4, zeros(1, 7), -1 / 2, 0, - 1 / 2, zeros(1, 7), 3 / 4]);
x_20 = a_x.idtfs.func('real');
x = DTS.Periodic({-20, 99}, x_20.value);
y1 = x.filter(b1, a1);
y2 = x.filter(b2, a2);

DTS.Figures("v", ...
    struct("xlabel", "n", "ylabel", "x[n]", "xlim", [0 99], "grid", "on"), ...
    { x.sInf("x[n]", "g", ".") }, ...
    struct("xlabel", "n", "ylabel", "y_1[n]", "xlim", [0 99], "grid", "on"), ...
    { y1.sInf("y_1[n]", "r", ".") }, ...
    struct("xlabel", "n", "ylabel", "y_2[n]", "xlim", [0 99], "grid", "on"), ...
    { y2.sInf("y_2[n]", "b", ".") } ...
);
```

**(f)**

```
clear;

a1 = [1, -0.8]; b1 = [1];
a2 = [1, 0.8]; b2 = [1];
w_0 = 2 * pi / 20;
a_x = DTS([0, 3 / 4, zeros(1, 7), -1 / 2, 0, - 1 / 2, zeros(1, 7), 3 / 4]);
x_20 = a_x.idtfs.func('real');
x = DTS.Periodic({-5, 20}, x_20.value);
A_y1 = DTS(w_0 * (0 : 19), x.filter(b1, a1).cut({0, 19}).dtfs).func('abs');
A_y2 = DTS(w_0 * (0 : 19), x.filter(b2, a2).cut({0, 19}).dtfs).func('abs');
A_x = DTS(w_0 * (0 : 19), a_x.value).func('abs');

DTS.Figures("v", ...
    struct("xlabel", "k", "ylabel", "|a[k]|", "xlim", [0, 2 * pi], "ylim", [0,
2.5], "grid", "on"), ...
    { A_x.sInf("|a[k]|", "r"), A_y1.sInf("|a_{y1}[k]|", "b") }, ...
    struct("xlabel", "k", "ylabel", "|a[k]|", "xlim", [0, 2 * pi], "ylim", [0,
2], "grid", "on"), ...
    { A_x.sInf("|a[k]|", "r"), A_y2.sInf("|a_{y2}[k]|", "b") } ...
);
```

## Programs for 3.9

**(a)**

```
clear;

S = 1000;
RC = 1;
x = DTS.Identity({0, 20}, S).func('cos');
y = x.lsim([1], [RC, 1]);

DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "f(t)", "grid", "on"), ...
    { x.cut({10, 20}).sInf("x(t)", "r", "", "plot"), y.cut({10,
20}).sInf("y(t)", "b", "", "plot") } ...
);
```

**(b)**

```
clear;

S = 1000;
RC = 1;
x2 = DTS.Identity({0, 20}, S).func('cos').func('sign');
y2 = x2.lsim([1], [RC, 1]);
```

```
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "f(t)", "grid", "on"), ...
    { x2.cut({10, 20}).sInf("x_2(t)", "r", "", "plot"), y2.cut({10,
20}).sInf("y_2(t)", "b", "", "plot") } ...
);
```

**(c)**

```
clear;

S = 1000;
x2 = DTS.Identity({0, 20}, S).func('cos').func('sign');
apos_k = DTS(2 * DTS(DTS.Identity({1, 10}) .* pi ./ 2).func('sin') ./
(DTS.Identity({1, 10}) .* pi)).shift(-1).stretch(2).value;
aneg_k = DTS(2 * DTS(DTS.Identity({-10, -1}) .* pi ./ 2).func('sin') ./
(DTS.Identity({-10, -1}) .* pi)).shift(1).stretch(-2).value;
ssum = DTS({0, 0}, 0, S);
Idn = DTS.Identity({-5, 20}, S);
for I = 1 : 5
    k = 2 * I - 1;
    s(I) = apos_k(I) * DTS(1j * k * Idn).func('exp') + aneg_k(I) * DTS(-1j *
k * Idn).func('exp');
    ssum = ssum + s(I);
end

figure(1);
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "s(t)", "grid", "on"), { s(1).cut({0,
20}).sInf("s_1(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "s(t)", "grid", "on"), { s(2).cut({0,
20}).sInf("s_2(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "s(t)", "grid", "on"), { s(3).cut({0,
20}).sInf("s_3(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "s(t)", "grid", "on"), { s(4).cut({0,
20}).sInf("s_4(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "s(t)", "grid", "on"), { s(5).cut({0,
20}).sInf("s_5(t)", "b", "", "plot") } ...
);

figure(2);
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "f(t)", "grid", "on"), ...
    { x2.cut({0, 20}).sInf("x(t)", "r", "", "plot"), ssum.cut({0,
20}).sInf("x'(t)", "b", "", "plot") } ...
```

```
);
```

**(d)**

```
clear;

S = 1000;
RC = 1;
x2 = DTS.Identity({-5, 20}, S).func('cos').func('sign');
apos_k = DTS(2 * DTS(DTS.Identity({1, 10}) .* pi ./ 2).func('sin') ./
(DTS.Identity({1, 10}) .* pi)).shift(-1).stretch(2).value;
aneg_k = DTS(2 * DTS(DTS.Identity({-10, -1}) .* pi ./ 2).func('sin') ./
(DTS.Identity({-10, -1}) .* pi)).shift(1).stretch(-2).value;
ssum = DTS({0, 0}, 0, S);
ysum = DTS({0, 0}, 0, S);
Idn = DTS.Identity({-5, 20}, S);
for I = 1 : 5
    k = 2 * I - 1;
    s(I) = apos_k(I) * DTS(1j * k * Idn).func('exp') + aneg_k(I) * DTS(-1j *
k * Idn).func('exp');
    y(I) = s(I).lsim([1], [RC, 1]);
    ssum = ssum + s(I);
    ysum = ysum + y(I);
end

figure(1);
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(1).cut({0,
20}).sInf("y_1(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(2).cut({0,
20}).sInf("y_2(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(3).cut({0,
20}).sInf("y_3(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(4).cut({0,
20}).sInf("y_4(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(5).cut({0,
20}).sInf("y_5(t)", "b", "", "plot") } ...
);

figure(2);
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "f(t)", "grid", "on"), ...
    { ssum.lsim([1], [RC, 1]).cut({0, 20}).sInf("Response of s_{sum}(t)", "r",
"", "plot"), ysum.cut({0, 20}).sInf("y_{sum}(t)", "b", "--", "plot") } ...
);
```

**(e)**

```
clear;

S = 1000;
RC = 1;
x2 = DTS.Identity({-5, 20}, S).func('cos').func('sign');
apos_k = DTS(2 * DTS(DTS.Identity({1, 10}) .* pi ./ 2).func('sin') ./
(DTS.Identity({1, 10}) .* pi)).shift(-1).stretch(2).value;
aneg_k = DTS(2 * DTS(DTS.Identity({-10, -1}) .* pi ./ 2).func('sin') ./
(DTS.Identity({-10, -1}) .* pi)).shift(1).stretch(-2).value;
ysum = DTS({0, 0}, 0, S);
Idn = DTS.Identity({-5, 20}, S);
for I = 1 : 5
    k = 2 * I - 1;
    s(I) = apos_k(I) * DTS(1j * k * Idn).func('exp') + aneg_k(I) * DTS(-1j *
k * Idn).func('exp');
    y(I) = s(I).lsim([1], [RC, 1]);
    ysum = ysum + y(I);
end

DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "f(t)", "grid", "on"), ...
    { x2.lsim([1], [RC, 1]).cut({0, 20}).sInf("Response of s_{sum}(t)", "r",
"", "plot"), ysum.cut({0, 20}).sInf("y_{sum}(t)", "b", "--", "plot") } ...
);
```

**(f)**

```
clear;

S = 1000;
RC = 1;
x2 = DTS.Identity({-5, 20}, S).func('cos').func('sign');
apos_k = DTS(2 * DTS(DTS.Identity({1, 10}) .* pi ./ 2).func('sin') ./
(DTS.Identity({1, 10}) .* pi)).shift(-1).stretch(2).value;
aneg_k = DTS(2 * DTS(DTS.Identity({-10, -1}) .* pi ./ 2).func('sin') ./
(DTS.Identity({-10, -1}) .* pi)).shift(1).stretch(-2).value;
msum = DTS({0, 0}, 0, S);
ysum = DTS({0, 0}, 0, S);
Idn = DTS.Identity({-5, 20}, S);
for I = 1 : 5
    k = 2 * I - 1;
    s(I) = apos_k(I) * DTS(1j * k * Idn).func('exp') + aneg_k(I) * DTS(1j * k
* Idn).func('exp');
```

```
    m(I) = s(I).lsim([1], [RC, 1]);
    y(I) = apos_k(I) * 1 / (1 + 1j * k) * DTS(1j * k * Idn).func('exp') + aneg_k(I)
* 1 / (1 - 1j * k) * DTS(-1j * k * Idn).func('exp');
    ysum = ysum + y(I);
    msum = msum + m(I);
end

figure(1);
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(1).cut({10,
20}).sInf("y_{1(Analytical)}(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(2).cut({10,
20}).sInf("y_{2(Analytical)}(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(3).cut({10,
20}).sInf("y_{3(Analytical)}(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(4).cut({10,
20}).sInf("y_{4(Analytical)}(t)", "b", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { y(5).cut({10,
20}).sInf("y_{5(Analytical)}(t)", "b", "", "plot") } ...
);

figure(2);
DTS.Figures("v", ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { m(1).cut({10,
20}).sInf("y_{1(Simulation)}(t)", "r", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { m(2).cut({10,
20}).sInf("y_{2(Simulation)}(t)", "r", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { m(3).cut({10,
20}).sInf("y_{3(Simulation)}(t)", "r", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { m(4).cut({10,
20}).sInf("y_{4(Simulation)}(t)", "r", "", "plot") }, ...
    struct("xlabel", "t", "ylabel", "y(t)", "grid", "on"), { m(5).cut({10,
20}).sInf("y_{5(Simulation)}(t)", "r", "", "plot") } ...
);
```

**Programs for 3.10**

**(b)**

```
clear;

m = [3, 5, 6, 7, 8];
dtfscomps = [];
T = 100;
for N = 2 .^ m
    x = 0.9 .^ DTS.Identity({0, N - 1});
```

```
    tic;
    for I = 1 : T; x.dtfs; end
    dtfscomps = [dtfscomps, toc / T];
end

stem(m, dtfscomps), xlabel('log_2(N)'), ylabel('Time cost (s)');
grid on;
```

**(c)**

```
clear;

m = [3, 5, 6, 7, 8];
dtfscomps = [];
T = 100;
for N = 2 .^ m
    x = 0.9 .^ DTS.Identity({0, N - 1});
    tic;
    for I = 1 : T; x.dtfs; end
    dtfscomps = [dtfscomps, toc / T];
end

fftcomps = [];
T = 100;
for N = 2 .^ m
    x = 0.9 .^ DTS.Identity({0, N - 1});
    tic;
    for I = 1 : T; x.fft; end
    fftcomps = [fftcomps, toc / T];
end

figure(1);
stem(m, fftcomps), xlabel('log_2(N)'), ylabel('Time cost (s)');
grid on;

figure(2);
hold on, grid on;
stem(m, dtfscomps ./ fftcomps);
plot(3 : 0.1 : 8, 2 .^ (3 : 0.1 : 8) ./ (3 : 0.1 : 8));
xlabel('log_2(N)'), ylabel('Ratio (dtfscomps / fftcomps)');
legend("Ratio", "N / log_2N")
```

**(e)**

```
clear;
```

```
N = 40;
T = 10000;
x = 0.9 .^ DTS.Identity({0, N - 1});
h = 0.5 .^ DTS.Identity({0, N - 1});
tic;
for I = 1 : T; y = DTS.Pconv(x, h); end
f40c = toc / T;

DTS.Figures("v", ...
    struct("xlabel", "n", "ylabel", "y[n]", "xlim", [0, N - 1], "grid",
"on"), ...
    { y.sInf("y[n]", "b") } ...
);
```

**(f)**

```
clear;

N = 80;
T = 10000;
x = 0.9 .^ DTS.Identity({0, N - 1});
h = 0.5 .^ DTS.Identity({0, N - 1});
tic;
for I = 1 : T; y = DTS.Pconv(x, h); end
f80c = toc / T;

DTS.Figures("v", ...
    struct("xlabel", "n", "ylabel", "y[n]", "xlim", [0, N - 1], "grid",
"on"), ...
    { y.sInf("y[n]", "b") } ...
);
```

**(g)**

```
clear;

N = 40;
T = 10000;
x = 0.9 .^ DTS.Identity({0, N - 1});
h = 0.5 .^ DTS.Identity({0, N - 1});
tic;
for I = 1 : T; y = DTS.Pconvfft(x, h); end
f40f = toc / T;

DTS.Figures("v", ...
```

```
    struct("xlabel", "n", "ylabel", "y[n]", "xlim", [0, N - 1], "grid",
"on"), ...
    { y.sInf("y[n]", "b") } ...
);
```

**(h)**

```
clear;

N = 80;
T = 10000;
x = 0.9 .^ DTS.Identity({0, N - 1});
h = 0.5 .^ DTS.Identity({0, N - 1});
tic;
for I = 1 : T; y = DTS.Pconvfft(x, h); end
f80f = toc / T;

DTS.Figures("v", ...
    struct("xlabel", "n", "ylabel", "y[n]", "xlim", [0, N - 1], "grid",
"on"), ...
    { y.sInf("y[n]", "b") } ...
);
```

**(i)**

```
clear;

L = 1; S = 1000; R = 200000;
C = []; F = [];
for N = L : S : R
    disp(N);
    x = 0.9 .^ DTS.Identity({0, N - 1});
    h = 0.5 .^ DTS.Identity({0, N - 1});
    T1 = 2; T2 = 2;
    tic; for I = 1 : T1; DTS.Pconv(x, h); end
    C = [C, toc / T1];
    tic; for I = 1 : T2; DTS.Pconvfft(x, h); end
    F = [F, toc / T2];
end

figure(1);
hold on, grid on;
stem(L : S : R, C * 1000);
stem(L : S : R, F * 1000);
xlim([L, R]);
xlabel("N"), ylabel("Time cost (ms)");
```

```
legend(["Method conv", "Method fft & ifft"])

figure(2);
stem(L : S : R, C ./ F);
xlim([L, R]);
xlabel("N"), ylabel("Ratio \frac{c}{f}");
```