# Part A Question 1

## (a)

The overlap-add method is an efficient way to evaluate the discrete convolution of a very long signal $x[n]$ of length $L_0$ with a FIR filter $h[n]$ of length $M$.

$$h[n] = \begin{cases} b_k & k = 0, 1, \cdots, M-1 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The covlution

$$y[n] = \sum_{k=0}^{M-1} h[k]x[n-k] = h[0]x[n] + h[1]x[n-1] + \cdots + h[M-1]x[n-M+1] \tag{2}$$

has length $L_0 + M - 1$, i.e. $y[n] = 0$ for $n < 0$ and $n \geq L_0 + M - 1$.

The concept is to divide the problem into multiple convolutions of $h[n]$ with short segments of $x[n]$:

$$x_k[n] := \begin{cases} x[n + (k-1)L] & n = 0, 1, \cdots, L-1 \\ 0 & n = L, L+1, \cdots, N-1 \end{cases} \tag{3}$$

where $L$ is an arbitrary segment length and $k = 1, 2, 3, \cdots$.

$$x[n] = \sum_k x_k[n - (k-1)L] \tag{4}$$

$y[n]$ can be written as a sum of short convolutions:

$$y[n] = \left( \sum_k x_k[n - (k-1)L] \right) * h[n] = \sum_k x_k[n - (k-1)L] * h[n] = \sum_k y_k[n - (k-1)L] \tag{5}$$

where $y_k[n] := x_k[n] * h[n]$ is zero for $n < 0$ and $n \geq L + M - 1$. And for any parameter $N \geq L + M - 1$, it is equivalent to the $N$-point circular convolution of $x_k[n]$, with $h[n]$, in the region $[0, N-1]$.

Reference: `https://en.wikipedia.org/wiki/Overlap%E2%80%93add_method`

# Part A Question 2

## (a)

$$X[k] = \text{DFT}\{x[m]\} = \sum_{m=0}^{N-1} x[m] W_N^{km} \tag{6}$$

$$\begin{aligned} y[n] &= \text{DFT}\{X[k]\} \\ &= \sum_{k=0}^{N-1} X[k] W_N^{kn} \\ &= \sum_{k=0}^{N-1} \sum_{m=0}^{N-1} x[m] W_N^{km} W_N^{kn} \\ &= \sum_{k=0}^{N-1} x[m] \sum_{m=0}^{N-1} W_N^{k(m+n)} \end{aligned}$$

Considering that $(m+n) \in [0, 2N-2]$,

$$\sum_{m=0}^{N-1} W_N^{k(m+n)} = \sum_{m=0}^{N-1} e^{\frac{-j2\pi}{N}k(m+n)} = \begin{cases} N & m = N-n \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Therefore,

$$y[n] = N \cdot x[N-n] \tag{8}$$

### Algorithm

$$X[k] \xrightarrow{\text{FFT}} y[n] \xrightarrow{\text{divided by N}} x[N-n] \xrightarrow{\text{flip}} x[n+1] \xrightarrow{\text{right shift by 1}} x[n] \tag{9}$$

| $y[n]$ | $y[0]$ | $y[1]$ | $y[2]$ | $\cdots$ | $y[N-3]$ | $y[N-2]$ | $y[N-1]$ |
|---|---|---|---|---|---|---|---|
| divided by N | $x[0]$ | $x[N-1]$ | x$[N-2]$ | $\cdots$ | $x[3]$ | $x[2]$ | $x[1]$ |
| flip | $x[1]$ | $x[2]$ | x[3] | $\cdots$ | $x[N-2]$ | $x[N-1]$ | $x[0]$ |
| right shift by 1 | $x[0]$ | $x[1]$ | x[2] | $\cdots$ | $x[N-3]$ | $x[N-2]$ | $x[N-1]$ |

Table 1: algorithm deduction

# Part A Question 3

## (a)

$$X[k] = X^*[\langle -k \rangle_{2N}] = X^*[2N-k] \tag{10}$$

$$\begin{aligned} X[k] &= \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi kn}{2N}} \\ X[2N-k] &= \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi(2N-k)n}{2N}} \\ X^*[2N-k] &= \sum_{n=0}^{N-1} x^*[n] e^{j\frac{2\pi(2N-k)n}{2N}} \\ &= \sum_{n=0}^{N-1} x^*[n] e^{j2\pi n} e^{-j\frac{2\pi kn}{2N}} \\ &= \sum_{n=0}^{N-1} x^*[n] e^{-j\frac{2\pi kn}{2N}} \end{aligned}$$

Note that: $(z+w)^* = z^* + w^*$, $(zw)^* = z^* w^*$

Substituting into $X[k] = X^*[2N-k]$ (Eq. 10) yields

$$\sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi kn}{2N}} = \sum_{n=0}^{N-1} x^*[n] e^{-j\frac{2\pi kn}{2N}} \tag{11}$$

Therefore, $x[n] = x^*[n]$, i.e. $x[n] = \text{IDFT}\{X[k]\}$ is real.

### (b)

$$
\begin{aligned}
X_0[k] &= X[k] + X[k+N] \\
&= X^*[2N-k] + X^*[2N-(k+N)] \\
&= X^*[2N-k] + X^*[N-k] \\
&= X^*[N-k] + X^*[(N-k)+N] \\
&= X_0^*[N-k]
\end{aligned}
$$

Hence, $X_0[k]$ is conjugate symmetric.

$$
\begin{aligned}
jX_1[k] &= W_{2N}^{-k}(X[k] - X[k+N]) \\
&= jW_{2N}^{-k}(X^*[2N-k] - X^*[2N-(k+N)]) \\
&= jW_{2N}^{-k}(X^*[2N-k] - X^*[N-k]) \\
&= jW_{2N}^{-k}(X^*[N-k] - X^*[(N-k)+N]) \\
&= jX_1^*[N-k] \\
&= -(j)^*X_1^*[N-k] \\
&= -(jX_1[N-k])^* \\
&= -(jX_1[\langle -k\rangle_N])^*
\end{aligned}
$$

Hence, $jX_1[k]$ is conjugate anti-symmetric.

### (c)

Calculate $q[n]$

$$
\begin{aligned}
q[n] = \mathrm{IDFT}\{Q[k]\} &= \frac{1}{N}\sum_{k=0}^{N-1}(X_0[k] + jX_1[k])e^{j\frac{2\pi n}{N}k} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\left(X[k] + X[k+N] + jW_{2N}^{-k}X[k] - jW_{2N}^{-k}X[k+N]\right)e^{j\frac{2\pi n}{N}k} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\left(X[k](1+jW_{2N}^{-k}) + X[k+N](1-jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\left(X[k](1+jW_{2N}^{-k}) + X[k+N](1-jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\left(X[k](1+jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k} + \frac{1}{N}\sum_{k=0}^{N-1}\left(X[k+N](1-jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\left(X[k](1+jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k} + \frac{1}{N}\sum_{k=N}^{2N-1}\left(X[k](1-jW_{2N}^{-(k-N)})\right)e^{j\frac{2\pi n}{N}(k-N)} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\left(X[k](1+jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k} + \frac{1}{N}\sum_{k=N}^{2N-1}\left(X[k](1+jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k}e^{-j2\pi n} \\
&= \frac{1}{N}\sum_{k=0}^{2N-1}\left(X[k](1+jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k}
\end{aligned}
$$

Note that: $-W_{2N}^{-(k-N)} = -(e^{-j\frac{2\pi}{2N}})^{-(k-N)} = -e^{j\frac{\pi(k-N)}{N}} = e^{j\frac{\pi(k-N)}{N}+j\pi} = e^{j\frac{\pi k}{N}} = W_{2N}^{-k}$

Calculate $q^*[n]$

$$\begin{aligned}
q^*[n] &= \frac{1}{N}\sum_{k=0}^{2N-1}\left(X^*[k](1-jW_{2N}^{k})\right)e^{-j\frac{2\pi n}{N}k}\\
&= \frac{1}{N}\sum_{k=0}^{2N-1}\left(X[2N-k](1-jW_{2N}^{k})\right)e^{-j\frac{2\pi n}{N}k}\\
&= \frac{1}{N}\sum_{k=1}^{2N}\left(X[k](1-jW_{2N}^{2N-k})\right)e^{-j\frac{2\pi n}{N}(2N-k)}\\
&= \frac{1}{N}\sum_{k=1}^{2N}\left(X[k](1-jW_{2N}^{-k})\right)e^{-j4\pi n}e^{j\frac{2\pi n}{N}k}\\
&= \frac{1}{N}\sum_{k=1}^{2N-1}\left(X[k](1-jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k} + X[2N](1-j)\\
&= \frac{1}{N}\sum_{k=1}^{2N-1}\left(X[k](1-jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k} + X[0](1-j)\\
&= \frac{1}{N}\sum_{k=0}^{2N-1}\left(X[k](1-jW_{2N}^{-k})\right)e^{j\frac{2\pi n}{N}k}
\end{aligned}$$

Note that: $W_{2N}^{2N-k} = (e^{-j\frac{2\pi}{2N}})^{2N-k} = e^{-j2\pi}(e^{-j\frac{2\pi}{2N}})^{-k} = W_{2N}^{-k}$

$x[2n]$ and $x[2n+1]$ can be expressed in terms of $q[n]$ and $q^*[n]$.

$$\frac{1}{2}\Re\{q[n]\} = \frac{1}{4}(q[n]+q^*[n]) = \frac{1}{2N}\sum_{k=0}^{2N-1}X[k]e^{j\frac{2\pi n}{N}k} = \frac{1}{2N}\sum_{k=0}^{2N-1}X[k]W_{2N}^{-2nk} = x[2n] \tag{12}$$

$$\frac{1}{2}\Im\{q[n]\} = \frac{1}{4j}(q[n]-q^*[n]) = \frac{1}{2N}\sum_{k=0}^{2N-1}X[k]W_{2N}^{-k}e^{j\frac{2\pi n}{N}k} = \frac{1}{2N}\sum_{k=0}^{2N-1}X[k]W_{2N}^{-(2n+1)k} = x[2n+1] \tag{13}$$

Q.E.D.

## (d)

1. Constitute $X_0[k] = X[k] + X[k+N]$ and $X_1[k] = W_{2n}-k(X[k]-X[k+N])$ $(k = 0, \cdots N-1)$
2. Constitute $Q[k] = X_0[k] + jX_1[k]$
3. Compute $N$-point IDFT of $Q[k]$
4. $x[2n] = \frac{1}{2}\Re\{q[n]\}$ and $x[2n+1] = \frac{1}{2}\Im\{q[n]\}$

## (e)

The advantage of overlap add method is that the circular convolution can be computed very efficiently as follows, according to the circular convolution theorem:

$$y[n] = \text{IDFT}\left(\text{DFT}(x[n])\cdot\text{DFT}(h[n])\right) \tag{14}$$

where DFT and IDFT refer to the discrete Fourier transform and inverse discrete Fourier transform, respectively, evaluated over $N$ discrete points.

### Radix-2 FFT

Each butterfly requires:
**one** complex multiplication
**two** complex additions

In total, there are: $\frac{N}{2}$ butterflies per stage $\times \log_2(N)$ stages.

### Convolution using FFT and IFFT

$$y[n] = \sum_{k=0}^{M-1} h[k]x[n-k], \quad Y[k] = H[k]X[k] \tag{15}$$

| | | |
|---|---|---|
| $H[k]$ | computed once and can be ignored | |
| DFT of $x[n]$ | $\frac{N}{2}\log_2(N)$ multiplications | $N\log_2(N)$ additions |
| Computation of $Y[k]$ | $N$ multiplications | |
| IDFT of $Y[k]$ | $\frac{N}{2}\log_2(N)$ multiplications | $N\log_2(N)$ additions |
| In total | $N\log_2(2N)$ multiplications | $2N\log_2(N)$ additions |

### Convolution using real FFT and conjugate symmetric IFFT

**FFT**

Use a $\frac{N}{2}$-point complex FFT to evaluate a $N$-point real FFT. This algorithm is shown in the appendix on page 19.

Complex multiplications:

$$\frac{N}{4}\log_2(\frac{N}{2}) + \frac{N}{2} \tag{16}$$

Complex additions:

$$\frac{N}{2}\log_2(\frac{N}{2}) + 2N \tag{17}$$

**Computation of $Y[k] = H[k]X[k]$**

Due to conjugate symmetry, only $\frac{N}{2}+1$ complex multiplications need to be calculated.

**IFFT**

Use a $\frac{N}{2}$-point general IFFT to evaluate a $N$-point conjugate symmetric IFFT. This algorithm is shown on page 7.

Complex multiplications:

$$\frac{N}{4}\log_2(\frac{N}{2}) + \frac{N}{2} \tag{18}$$

Complex additions:

$$\frac{N}{2}\log_2(\frac{N}{2}) + \frac{3N}{2} \tag{19}$$

**In total**

Complex multiplications:

$$\left(\frac{N}{4}\log_2(\frac{N}{2}) + \frac{N}{2}\right) \times 2 + \frac{N}{2} + 1 = \frac{N}{2}\log_2(4N) + 1 \approx \frac{N}{2}\log_2(4N) \tag{20}$$

Complex additions:

$$N\log_2(N) + 2.5N \tag{21}$$

### Complex multiplications per output data point

$$c_{MLT}(\nu) = \frac{N\log_2(2N)}{L} = \frac{\frac{N}{2}\log_2(N) + N + 1}{N - M + 1} = \frac{2^\nu(0.5\nu + 1) + 1}{2^\nu - M + 1} \tag{22}$$

### Complex additions per output data point

$$c_{ADD}(\nu) = \frac{2N\log_2(N)}{L} = \frac{N\log_2(N) + 2.5N}{N - M + 1} = \frac{2^\nu(\nu + 2.5)}{2^\nu - M + 1} \tag{23}$$

# Part B Task 1

## (a)

### `iffta(X)`

```
function x = iffta(X)
    X_conj = conj(X);
    x_conj = fft(X_conj);
    x = conj(x_conj);
    x = x / length(X);
end
```

### `ifftb(X)`

```
function x = ifftb(X)
    tmp = fft(X);
    tmp = tmp / length(X);
    tmp = fliplr(tmp);              % flip array left to right
    x = circshift(tmp, 1, 2);       % shift by 1 in the 2nd dimension (right shift by 1)
end
```

## (b)

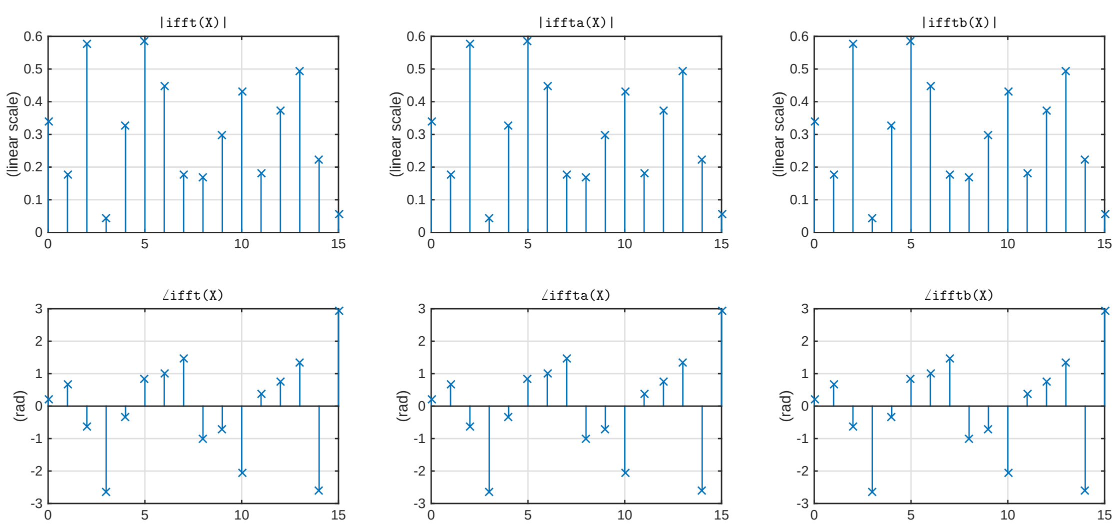


Figure 1: `ifft(X)`, `iffta(X)` and `ifftb(X)`

MATLAB code can be found in Appendix on page 20.

# Part B Task 2

## (a) isConjugateSymmetric(X)

```
function bool = isConjugateSymmetric(X)
    N = length(X);
    X = X(2:N);     % discard the first element

    RePart = real(X);
    RePartReverse = fliplr(RePart);

    ImPart = imag(X);
    ImPartReverse = fliplr(ImPart);

    tolerance = eps('single');

    bool1 = any(abs(RePart-RePartReverse) > tolerance);
    bool2 = any(abs(ImPart+ImPartReverse) > tolerance);

    bool = ~(bool1 || bool2);
end
```

## (b) ifftcs(X)

```
function x = ifftcs(X)
    if ~isConjugateSymmetric(X)
        error('input is not conjugate symmetric');
    end

    N = length(X) / 2;

    if N ~= round(N)
        error('input is not a length-2N sequence');
    end

    index = 1:N;

    X0(index) = X(index) + X(index + N);
    W = exp(1j * pi / N) .^ (index-1);
    X1(index) = W .* (X(index) - X(index + N));

    Q = X0 + 1j * X1;

    q = ifft(Q);

    x(index*2-1) = 0.5 * real(q(index));
    x(index*2) = 0.5 * imag(q(index));
end
```

### (c)

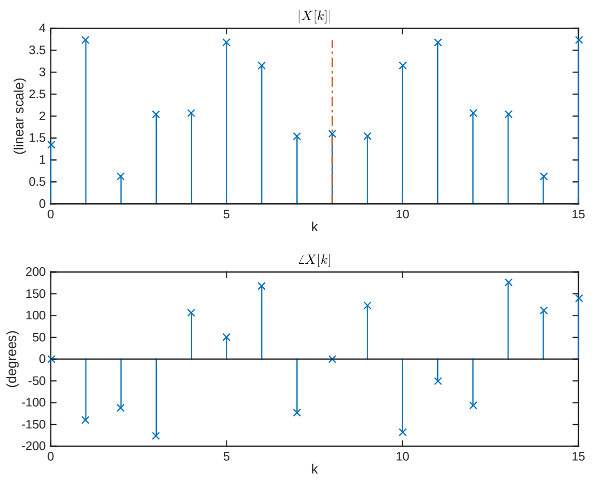


Figure 2: $|X[k]|$ and $\angle X[k]$

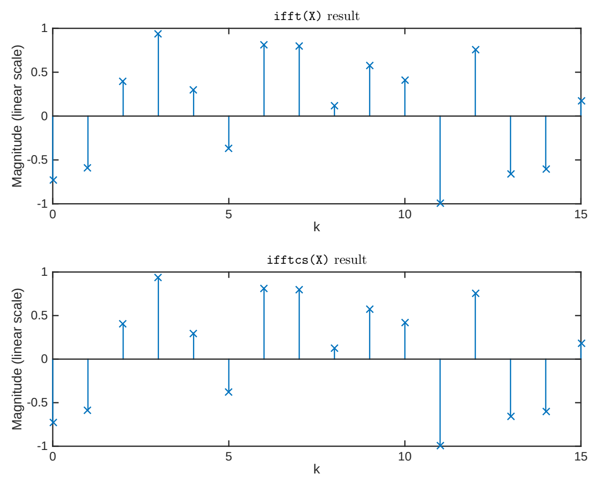


Figure 3: `ifft(X)` and `ifftcs(X)`

MATLAB code can be found in Appendix on page 21.

## Part B Task 3

### (a)

MATLAB code can be found in Appendix on page 13.

## Part B Task 4

**MATLAB code can be found in Appendix on page 17.**

### (a)

The preliminary $N_{order} = 39$ can be calculated by `firpmord(f,a,dev,fs)` function. However, the performance of the FIR filter cannot meet the design specifications, especially the stopband ripple term. We increment the filter order until all specifications are satisfied. Eventually, when $N_{order} = 46$, all specifications are satisfied (shown in Fig. 5).

Thus, the length of filter response is

$$M = 47 \tag{24}$$

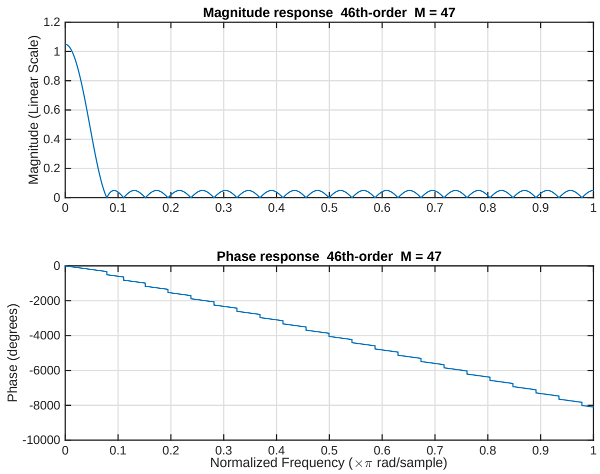


Figure 4: FIR filter frequency response

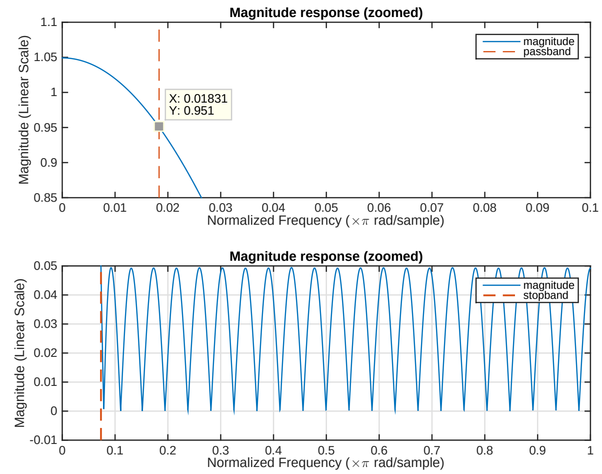


Figure 5: FIR filter frequency response (zoomed)

## (b)

### Test signal

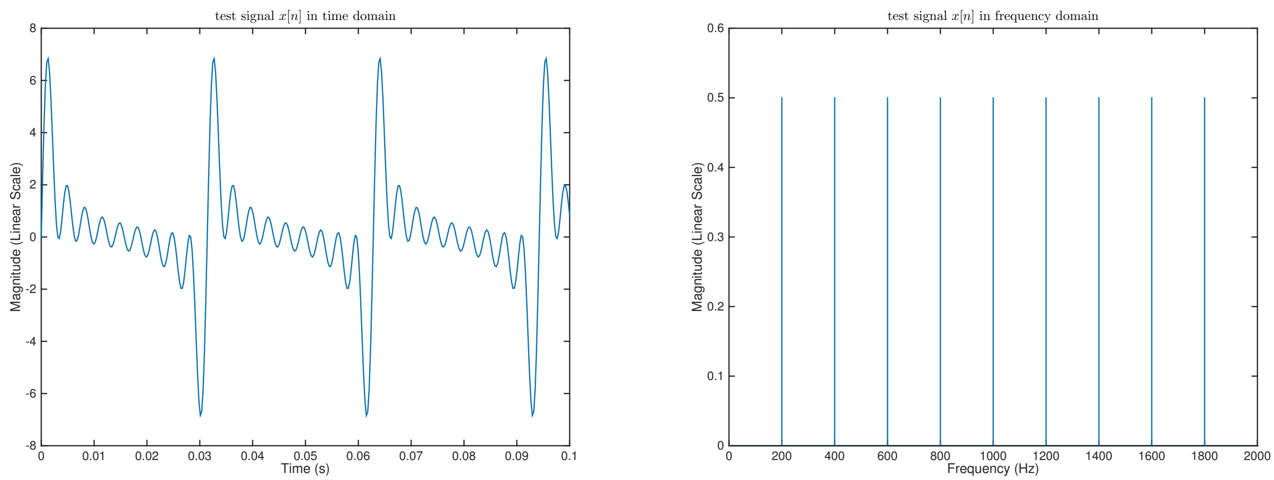


Figure 6: Test signal

As is shown in Fig. 6, a nine-tone sinusoid signal is designed as the test signal.

$$x[n] = \sum_{k=1}^{9} \sin(2\pi k f_0 n / F_s) \tag{25}$$

where $f_0 = 200$ Hz.

## Optimal block length

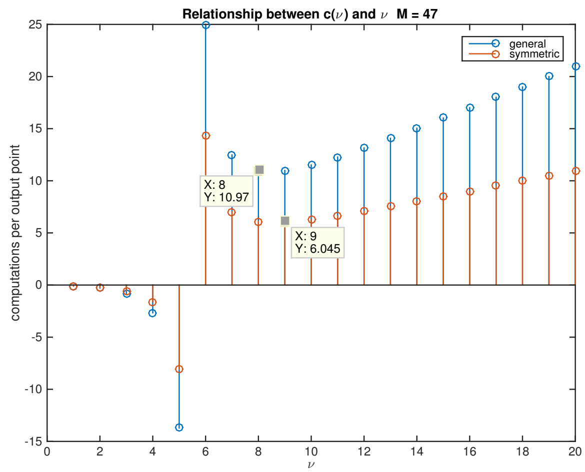


Figure 7: Relationship between $c(\nu)$ and $\nu$ for $M = 47$

The relationship between $c(\nu)$ and $\nu$ for $M = 47$ (Fig. 7) can be plotted based on Eq.22 on page 6. It can be clear seen that, when $N = 2^9 = 512$, $c(\nu)$ reaches its minimum $c_{MLT}(9) = 6.045$.

$$N_{\text{optimal}} = 2^9 = 512 \tag{26}$$

Taking advantages of symmetry reduces *complex multiplications per output data point* by

$$\frac{10.97 - 6.045}{10.97} \times 100\% = 45\% \tag{27}$$

## Output verification

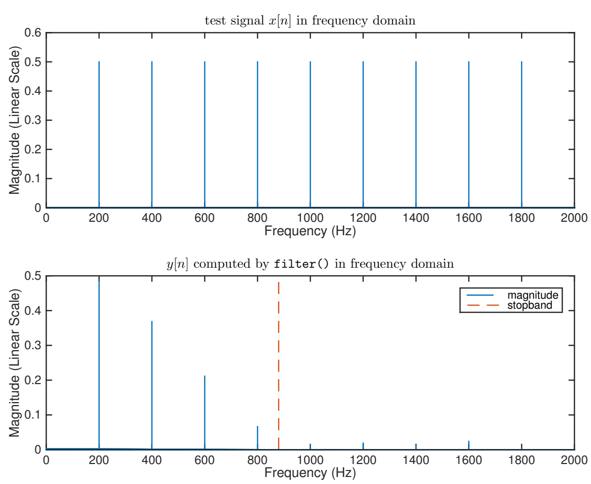


Figure 8: Test signal and `filter` output

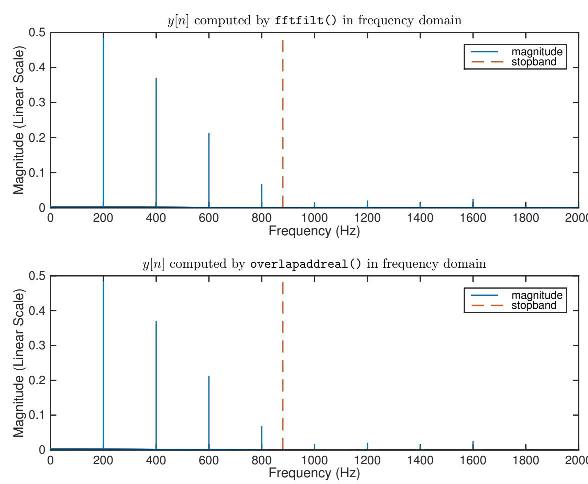


Figure 9: `fftfilt` output and `overlapaddreal` output

As is shown in Fig. 8 and Fig. 9, `filter` output, `fftfilt` output and `overlapaddreal` output are consistent. In case of any discrepancy, several lines of code are programmed to check whether outputs are accordant. If not, **warning** messages will be displayed.

```
% Display warning message if results are inconsistent
if any(abs(y_filter - y_fftfilt) > eps('single'))
    warning('filter() result and fftfilt() result are inconsistent.');
end

if any(abs(y_filter - y_overlapaddreal) > eps('single'))
    warning('filter() result and overlapaddreal() result are inconsistent.');
end
```

### Functions comparison

`filter` filters data with recursive (IIR) or nonrecursive (FIR) filter. `fftfilt` filters data using the efficient FFT-based method of overlap-add, a frequency domain filtering technique that works only for **FIR** filters.

When the input signal is relatively large, it is advantageous to use `fftfilt` instead of `filter`, which performs $M$ multiplications for each sample in $x$, where $M$ is the filter length. `fftfilt` performs 2 FFT operations - the FFT of the signal block of length $L$ plus the inverse FT of the product of the FFTs - at the cost of $\frac{1}{2}L\log_2(L)$ where $L$ is the block length. It then performs $L$ point-wise multiplications for a total cost of $L + L\log_2(L)$ multiplications.

The cost ratio is therefore

$$\frac{\texttt{fftfilt()}}{\texttt{filter()}} = \frac{L + L\log_2(L)}{ML} = \frac{L(1+\log_2 L)}{ML} = \frac{\log_2(2L)}{M} \tag{28}$$

As a result, `fftfilt` becomes advantageous when $\log_2(2L)$ is less than $M$.

Reference: `https://www.mathworks.com/help/signal/ref/fftfilt.html`

## (c)

### `filter()`

Complex multiplications per second

$$M \cdot F_s = 47 \times 24000 = 1128000 \tag{29}$$

Complex additions per second

$$(M-1) \cdot F_s = 46 \times 24000 = 1104000 \tag{30}$$

### `fftfilt()`

Complex multiplications per second

$$F_s \cdot (1 + \log_2 F_s) = 373217.9 \tag{31}$$

Complex additions per second

$$2F_s \cdot (1 + \log_2 F_s) = 746435.8 \tag{32}$$

### Taking advantages of symmetry

Complex multiplications per second

$$c(9)_{MLT} \cdot F_s = 145082 \tag{33}$$

$c(9)_{MLT} = 6.045$ is calculated by Eq.22 on page 6.

Complex additions per second

$$c(9)_{ADD} \cdot F_s = 303245 \tag{34}$$

$c(9)_{ADD} = 12.63$ can be calculated by Eq.23 on page 6.

# Part C

`_LabTasks.c` and `Params.h` can be found on page 14.

During the workshop session, we fed a two-tone (200Hz and 1kHz) sinusoidal signal into the DSP board. The DSP board swapped output source every 5 seconds (input, `process_time()` output and `process_block()` output in turn). We consider `process_time()` and `process_block()` functioned normally, because 1kHz-component was effectively attenuated.

In order to conduct a more rigorous test, we used MATLAB to generate a test signal ($\frac{24000}{200} \times 3 = 360$ points, span of three periods) and obtained the output via built-in function `filter()`. (`test.m` can be found on page 21.)

At the next stage, we imported the test signal into a `.c` file and simulated the filtering process. `test.c` and `SPWS3.h` can be found on page 22. After compiling these two files with aforementioned `_LabTasks.c` and `Params.h`, output variables `output_time[]` and `output_block[]` can be inspected in "debug perspective".

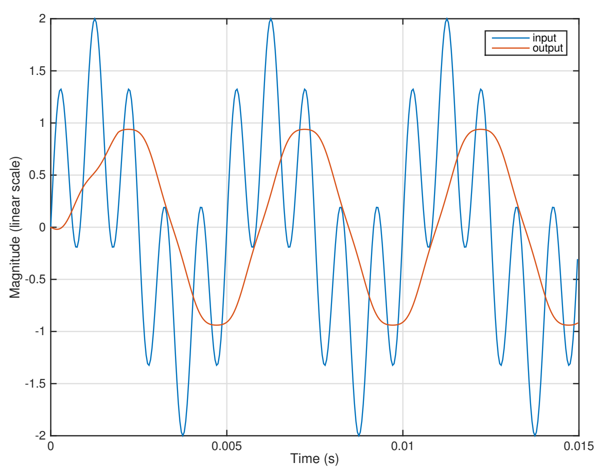


Figure 10: MATLAB `filter()` output

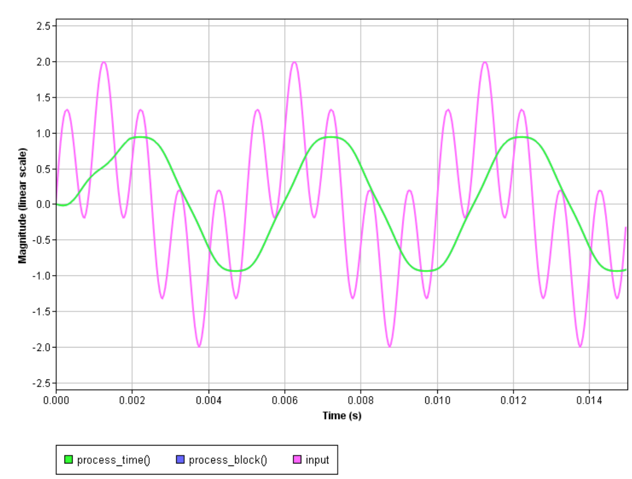


Figure 11: `process_time()` and `process_block()`

As is shown in Figure 10 and Figure 11, the outputs of `process_time()` and `process_block()` cannot be differentiated, and these two waveforms are consistent with the MATLAB simulation.

# Appendix

## overlapaddreal(B, x, N)

```
function y = overlapaddreal(B, x, N)
    M = length(B);          % length of filter response

    if N < M
        error('N is less than the length of filter response M.');
    end

    L = N - M + 1;          % x[n] segment length

    N_x = length(x);        % length of x[n]
    kmax = ceil(N_x/L);     % number of data blocks

    B = [B zeros(1, N-M)];
    H = fft(B);

    x = [x zeros(1, kmax*L - N_x)];
    % append zeros to make up (kmax * L) elements

    y = zeros(1, kmax*L);
    y_k_buffer = zeros(1, M-1);

    overlap_index = 1:M-1;

    for k = 1:kmax
        index_start = (k-1)*L + 1;
        index_end = k * L;

        x_k = [x(index_start:index_end) zeros(1, M-1)];
        % append M-1 zeros

        X_k = fft(x_k);

        Y_k(1:N/2+1) = X_k(1:N/2+1) .* H(1:N/2+1);
        Y_k(N/2+2:N) = conj(Y_k(N/2:-1:2));
        % Y_k = X_k .* H;

        y_k = ifft(Y_k);
        % y_k = filter(B, 1, x_k);

        y_k(overlap_index) = y_k(overlap_index) + y_k_buffer(overlap_index);
        % add overlapped M-1 points together

        y(index_start:index_end) = y_k(1:L);
        % output first L points to y

        y_k_buffer(overlap_index) = y_k(overlap_index+L);
        % store last M-1 points for next round
    end

    y = y(1:N_x);
    % keep the lengths of x and y equal
end
```

## Params.h

```
// TODO: 0. Modify these constants to match the filter you have designed

// length of filter
#define M 47

// buffer size
#define N 512

// input data processing block size
#define L (N-M+1)

#define BUFFER_SIZE     (M-1)
#define REM(INDEX)      ((INDEX) + BUFFER_SIZE) % BUFFER_SIZE
// if an index is negative, a specified position from the end of the array will be returned.
// e.g. given an array x[8], x[REM(-1)] and x[REM(7)] both refer to x[7].
```

## _LabTasks.c

```
#include "SPWS3.h"
#include "Params.h"

complex_fract32 twiddle[N/2] = { 0 };
complex_fract32 filter_fft[N] = { 0 };

complex_fract32 input_fft[N] = { 0 };
complex_fract32 output_fft[N] = { 0 };
fract32 output_save[M-1] = { 0 };

// array b
float b[] = { -0.023442, 0.002569, 0.003110, 0.004042, 0.005350,
        0.006991, 0.008953, 0.011183, 0.013656, 0.016321,
        0.019156, 0.022085, 0.025091, 0.028018, 0.030937,
        0.033753, 0.036369, 0.038771, 0.040886, 0.042684,
        0.044126, 0.045181, 0.045806, 0.046024, 0.045806,
        0.045181, 0.044126, 0.042684, 0.040886, 0.038771,
        0.036369, 0.033753, 0.030937, 0.028018, 0.025091,
        0.022085, 0.019156, 0.016321, 0.013656, 0.011183,
        0.008953, 0.006991, 0.005350, 0.004042, 0.003110,
        0.002569, -0.023442 };

float process_time(float x0)
{
    // TODO: 1. Implement the filter using time domain methods

    static float xBuffer[BUFFER_SIZE] = {0.0};          // BUFFER_SIZE = (M-1) is defined in 'Params.h'
    static int current = 0;

/*
    float y = b[0] * x0;
    for (int i = 1; i <= BUFFER_SIZE; i++) {
        y += b[i] * xBuffer[REM(current-i)];
    }
*/

    // M = 47 odd
    // y[n] = h[0]x[n] + h[1]x[n - 1] + ... + h[M - 1]x[n - M + 1]
    //      = (h[0]x[n] + h[M - 1]x[n - M + 1]) + (h[1]x[n-1] + h[M - 2]x[n - M + 2]) + ... + h[(M-1)/2]x
        [(M-1)/2]

    float y = b[0] * (x0 + xBuffer[REM(current)]);      // Macro 'REM(current)' is defined in 'Params.h'
```

```c
    // h[0]x[n] + h[M - 1]x[n - M + 1]

    xBuffer[current] = x0;
    // save current x0 into xBuffer after 'y' is calculated, thus the size of 'xBuffer' can be reduced by
        1 (from M to M-1).

    for (int i = 1; i <= BUFFER_SIZE/2-1; i++) {
        y += b[i] * (xBuffer[REM(current-i)] + xBuffer[REM(current+i)]);
    }
    // (h[1]x[n-1] + h[M - 2]x[n - M + 2]) + ... + (h[(M-1)/2-1]x[(M-1)/2+1] + h[(M-1)/2+1]x[(M-1)/2-1])

    y += b[BUFFER_SIZE/2] * xBuffer[REM(current-BUFFER_SIZE/2)];
    // h[(M-1)/2]x[(M-1)/2]

    current++;
    current %= BUFFER_SIZE;

    return y;
}

void init_process()
{
    int i;

    // calculate twiddle factors
    twidfftrad2_fr32(twiddle, N);

    // copy filter coefficients to input array to do fft
    for (i = 0; i < M; i++)
        input_data[i] = (1 << 30) * b[i];
        // [ note ]
        // Here we should scale by (1 << 31)-1 for full scale, however
        // doing so can cause overflows in fixed point, so we halve it
        // here and put back the factor 2 on output.

    // do fft
    int filter_blk_exp;
    rfft_fr32(input_data, filter_fft, twiddle, 1, N, &filter_blk_exp, 1);

    // rescale data points
    for (i = 0; i < N; i++)
    {
        filter_fft[i].re = filter_fft[i].re << (filter_blk_exp);
        filter_fft[i].im = filter_fft[i].im << (filter_blk_exp);
    }

    // clear input array
    for (i = 0; i < M; i++)
        input_data[i] = 0;
}

void process_block(fract32 output[])
{
    // TODO: 2. Implement the filter using the overlap-add method

    int index = 0;

    // do fft
    int block_exponent;
    rfft_fr32(input_data, input_fft, twiddle, 1, N, &block_exponent, 1);

    // Y[k] = H[k] X[k]
```

```
/*
    for (index = 0; index < N; index++) {
        output_fft[index] = cmlt_fr32(filter_fft[index], input_fft[index]);
    }
*/
    // use conjugate symmetry to reduce complex computations
    output_fft[0] = cmlt_fr32(filter_fft[0], input_fft[0]);
    for (index = 1; index < N/2; index++) {
        output_fft[index] = cmlt_fr32(filter_fft[index], input_fft[index]);
        output_fft[N-index] = conj_fr32(output_fft[index]);
    }
    output_fft[N/2] = cmlt_fr32(filter_fft[N/2], input_fft[N/2]);

    complex_fract32 output_complex[N]= { 0 };

    // do ifft
    ifft_fr32(output_fft, output_complex, twiddle, 1, N, &block_exponent, 1);

    for (index = 0; index < N; index++) {
        // output_complex[index].re = output_complex[index].re << (block_exponent);
        // output_complex[index].im = output_complex[index].im << (block_exponent);
        // rescale data points

        // output[index] = output_complex[index].re;
        // the output will be real so copy just the real part

        output[index] = output_complex[index].re << (block_exponent);
        // combine the previous lines of code into a single line
    }

    // overlap add
/*
    MATLAB style code
    index = 1:M-1;
    output[index] = output[index] + output_save[index];
    output_save[index] = output[L+index];
*/
    for (index = 0; index < M-1; index++) {
        output[index] += output_save[index];
        output_save[index] = output[L+index];
    }
}
```

## Part B Task 4

```matlab
clear;
close all;

fs = 24E3;              % sampling frequency
f_pass = 220;           % passband frequency in Hz
f_stop = 880;           % stopband frequency in Hz

N = 2^9;                % block length

a = [1 0];
dev = [0.05 0.05];

%% FIR filter design
[n_order, fo, ao, w] = firpmord([f_pass f_stop], a, dev, fs);
n_order = n_order + 7;  % increment the filter order until all specifications are satisfied

numerator = firpm(n_order, fo, ao, w);
clear a dev fo ao w;
M = num2str(length(numerator));

[h_FIR, w_FIR] = freqz(numerator, 1, 2^12);

figure;
subplot(2, 1, 1);
plot(w_FIR/pi, abs(h_FIR));
title(['Magnitude response  ' num2str(n_order) 'th-order  M = ' M]);
ylabel('Magnitude (Linear Scale)');
grid on;

subplot(2, 1, 2);
plot(w_FIR/pi, rad2deg(phase(h_FIR)));
title(['Phase response  ' num2str(n_order) 'th-order  M = ' M]);
xlabel('Normalized Frequency (\times\pi rad/sample)');
ylabel('Phase (degrees)');
grid on;

figure;
subplot(2, 1, 1);
plot(w_FIR/pi, abs(h_FIR));
title('Magnitude response (zoomed)');
xlabel('Normalized Frequency (\times\pi rad/sample)');
ylabel('Magnitude (Linear Scale)');
axis([0 0.1 0.85 1.1]);

hold on;
plot([f_pass*2/fs f_pass*2/fs], [0.5 1.5], '--');
legend('magnitude', 'passband');

subplot(2, 1, 2);
plot(w_FIR/pi, abs(h_FIR));
title('Magnitude response (zoomed)');
xlabel('Normalized Frequency (\times\pi rad/sample)');
ylabel('Magnitude (Linear Scale)');
axis([0 1 -0.01 0.05]);
grid on;
clear h_FIR w_FIR;

hold on;
plot([f_stop*2/fs f_stop*2/fs], [-0.5 0.5], '--', 'linewidth', 1.5);
legend('magnitude', 'stopband');
```

```
%% generate test signal

f0 = (1:9)*200;      % test signal frequencies
N_x = fs;            % test signal length

vector = 2 * pi * (0:N_x-1) / fs;

x_martrix = zeros(length(f0), N_x);

for index=1:length(f0)
    x_martrix(index,:) = sin(vector * f0(index));
end
clear index;

x = sum(x_martrix);

%% FIR filter implementation and test

y_filter  = filter(numerator, 1, x);
y_fftfilt  = fftfilt(numerator, x);
y_overlapaddreal = overlapaddreal(numerator, x, N);

[~, X] = single_side_FFT(x, fs);
[~, Y_filter] = single_side_FFT(y_filter, fs);
[~, Y_ffftfilt] = single_side_FFT(y_fftfilt, fs);
[frequency_range, Y_overlapaddreal] = single_side_FFT(y_overlapaddreal, fs);

figure;
plot(vector, x);
title('test signal $x[n]$ in time domain', 'Interpreter', 'latex');
xlabel('Time (s)');
ylabel('Magnitude (Linear Scale)');
xlim([0 0.1]);

figure;
stem(frequency_range, X, 'marker', 'none');
title('test signal $x[n]$ in frequency domain', 'Interpreter', 'latex');
xlabel('Frequency (Hz)');
ylabel('Magnitude (Linear Scale)');
xlim([0 2000]);

figure;
subplot(2, 1, 1);
stem(frequency_range, X, 'marker', 'none');
title('test signal $x[n]$ in frequency domain', 'Interpreter', 'latex');
xlabel('Frequency (Hz)');
ylabel('Magnitude (Linear Scale)');
xlim([0 2000]);

subplot(2, 1, 2);
stem(frequency_range, Y_filter, 'marker', 'none');
title('$y[n]$ computed by \texttt{filter()} in frequency domain', 'Interpreter', 'latex');
xlabel('Frequency (Hz)');
ylabel('Magnitude (Linear Scale)');
hold on;
plot([f_stop f_stop], [0 0.5], '--');
legend('magnitude', 'stopband');
xlim([0 2000]);

figure;
subplot(2, 1, 1);
```

```
stem(frequency_range, Y_ffftfilt, 'marker', 'none');
title('$y[n]$ computed by \texttt{fftfilt()} in frequency domain', 'Interpreter', 'latex');
xlabel('Frequency (Hz)');
ylabel('Magnitude (Linear Scale)');
hold on;
plot([f_stop f_stop], [0 0.5], '--');
legend('magnitude', 'stopband');
xlim([0 2000]);

subplot(2, 1, 2);
stem(frequency_range, Y_overlapaddreal, 'marker', 'none');
title('$y[n]$ computed by \texttt{overlapaddreal()} in frequency domain', 'Interpreter', 'latex');
xlabel('Frequency (Hz)');
ylabel('Magnitude (Linear Scale)');
hold on;
plot([f_stop f_stop], [0 0.5], '--');
legend('magnitude', 'stopband');
xlim([0 2000]);

% Display warning message if results are inconsistent
if any(abs(y_filter - y_fftfilt) > eps('single'))
    warning('filter() result and fftfilt() result are inconsistent.');
end

if any(abs(y_filter - y_overlapaddreal) > eps('single'))
    warning('filter() result and overlapaddreal() result are inconsistent.');
end
```

## fft_real(x)

```
clear;
close all;

x = randn(1, 16);

N = length(x) / 2;
index = 1:N;

x_o = x(index*2-1);     % odd index
x_e = x(index*2);       % even index

z = x_o + 1j * x_e;

Z = fft(z);

% Z1 = conj(Z[N-k])
Z1 = fliplr(Z);
Z1 = circshift(Z1, 1, 2);
Z1 = conj(Z1);

X_o = 0.5 * (Z + Z1);
X_e = -0.5j * (Z - Z1);

W = exp(-1j * pi / N) .^ (index-1);

X(index) = X_o + X_e .* W;
X(index+N) = X_o - X_e .* W;

if any(abs(fft(x) - X) > eps('single'))
    warning('fft() result and fft_real() result are inconsistent.');
end
```

## Part B Task 1 (b)

```
clear;
close all;

N = 16;
n = 0:N-1;

input = complex(randn(1, N), randn(1, N));

output = ifft(input);
outputA = iffta(input);
outputB = ifftb(input);

magnitude0 = abs(output);
magnitudeA = abs(outputA);
magnitudeB = abs(outputB);

phase0 = angle(output);
phaseA = angle(outputA);
phaseB = angle(outputB);

subplot(2, 3, 1);
stem(n, magnitude0, 'marker', 'x');
title('\texttt{|ifft(X)|}', 'Interpreter', 'latex');
ylabel('(linear scale)');
grid on;

subplot(2, 3, 2);
stem(n, magnitudeA, 'marker', 'x');
title('\texttt{|iffta(X)|}', 'Interpreter', 'latex');
ylabel('(linear scale)');
grid on;

subplot(2, 3, 3);
stem(n, magnitudeB, 'marker', 'x');
title('\texttt{|ifftb(X)|}', 'Interpreter', 'latex');
ylabel('(linear scale)');
grid on;

subplot(2, 3, 4);
stem(n, phase0, 'marker', 'x');
title('\angle \texttt{ifft(X)}', 'Interpreter', 'latex');
ylabel('(rad)');
grid on;

subplot(2, 3, 5);
stem(n, phaseA, 'marker', 'x');
title('\angle \texttt{iffta(X)}', 'Interpreter', 'latex');
ylabel('(rad)');
grid on;

subplot(2, 3, 6);
stem(n, phaseB, 'marker', 'x');
title('\angle \texttt{ifftb(X)}', 'Interpreter', 'latex');
ylabel('(rad)');
grid on;

set (gcf, 'Position', [200 200 1000 420]);
```

## Part B Task 2 (c)

```
clear;
close all;

N = 16;
symmetry_axis = N/2;
n = 0:N-1;

x = randn(1, N);
X = fft(x);

output1 = ifft(X);
output2 = ifftcs(X);

figure;
subplot(2,1,1);
stem(n, abs(X), 'marker', 'x');
title('$|X[k]|$', 'Interpreter', 'latex');
xlabel('k');
ylabel('(linear scale)');
hold on;
plot([symmetry_axis symmetry_axis], [0 max(abs(X))], '-.');
subplot(2,1,2);
stem(n, rad2deg(angle(X)), 'marker', 'x');
title('$\angle X[k]$', 'Interpreter', 'latex');
xlabel('k');
ylabel('(degrees)');

figure;
subplot(2,1,1);
stem(n, output1, 'marker', 'x');
title('\texttt{ifft(X)} result', 'Interpreter', 'latex');
xlabel('k');
ylabel('Magnitude (linear scale)');
subplot(2,1,2);
stem(n, output2, 'marker', 'x');
title('\texttt{ifftcs(X)} result', 'Interpreter', 'latex');
xlabel('k');
ylabel('Magnitude (linear scale)');
```

## test.m

```
clear;
close all;
load('lowpass_filter_numerator');

%% test signal

fs = 24E3;          % sampling frequency
f0 = [200 1000];    % test signal frequencies
N_x = 360;          % test signal length

time_vector = (0:N_x-1) / fs;

x_martrix = zeros(length(f0), N_x);

for index=1:length(f0)
    x_martrix(index,:) = sin(2 * pi * f0(index) * time_vector);
end
clear index;

```

```
x = sum(x_martrix, 1);

%% filter

output = filter(numerator, 1, x);

plot(time_vector, x);
hold on;
plot(time_vector, output);
xlabel('Time (s)');
ylabel('Magnitude (linear scale)');
legend('input', 'output');
grid on;

len = length(output);

for i = 1:len
    fprintf('%f\t', output(i));
end
fprintf('\n');
```

## SPWS3.h

```
#include <filter.h>

float process_time(float);
void init_process(void);
void process_block(fract32[]);

extern float b[];
extern fract32 input_data[];
```

## test.c

```
#include <stdio.h>
#include <math.h>
#include "SPWS3.h"
#include "Params.h"

#define LENGTH 360
#define BLOCK_NUMBER (LENGTH/L + 1)
#define LENGTH_NEW (BLOCK_NUMBER*L)

// input data buffer
fract32 input_data[N] = { 0 };

// output data buffers
fract32 output_buffer0[N] = { 0 };
fract32 output_buffer1[N] = { 0 };
fract32* output_current  = output_buffer0;       // pointer to buffer for processing
fract32* output_playback = output_buffer1;       // pointer to buffer to be played out

float t_vector[LENGTH] = {0.0};
float output_time[LENGTH] = {0.0};
float output_block[LENGTH] = {0.0};

float input[] = { ... };

int main(void) {
    int i;

    printf("L=%d\n", L);
```

```c
    printf("%d blocks\n", BLOCK_NUMBER);
    printf("%d elements\n", LENGTH_NEW);

    for (i = 0; i < LENGTH; ++i) {
        t_vector[i] = i / 24E3;
    }
    printf("t_vector finished\n");

    // padding zeros
    float input_padding_with_zeros[LENGTH_NEW] = {0.0};
    for (i = 0; i < LENGTH; ++i) {
        input_padding_with_zeros[i] = input[i];
    }
    printf("padding zeros finished\n");

    // process_block
    init_process();
    float output_block_temp[LENGTH_NEW] = {0.0};
    for (int j = 0; j < BLOCK_NUMBER; j++) {
        fract32* temp;
        temp = output_playback;
        output_playback = output_current;
        output_current = temp;

        for (i = 0; i < L; i++) {
            input_data[i] = input_padding_with_zeros[i+j*L] * (1 << 30);
        }
        process_block(output_current);

        temp = output_playback;
        output_playback = output_current;
        output_current = temp;

        for (i = 0; i < L; i++) {
            output_block_temp[i+j*L] = output_playback[i];
        }
    }
    for (i = 0; i < LENGTH; i++) {
        output_block[i] = output_block_temp[i] / (1 << 29);
        // 30 - 1 = 29
        // Note: scaling to avoid overflows in fixed point.
        // See init_process() for more information.
    }
    printf("process_block finished\n");

    // process_time
    for (i = 0; i < LENGTH; ++i) {
        output_time[i] = process_time(input[i]);
    }
    printf("process_time finished\n");

    return 0;
}
```