SUSTech DSP Lab5 Report

Cao Zhengyang
Department of Electronics and Electrical Engineering
12110623@mail.sustech.edu.cn

I. Introduction

The presented document details a comprehensive exploration of digital signal processing concepts and algorithms, primarily focusing on the Discrete Fourier Transform (DFT) and the Fast Fourier Transform (FFT).

II. Continuation of DFT Analysis

A. Shifting the Frequency Range

1) DFTsum: The DFTsum() function is from Lab 4, and the result is as follows.

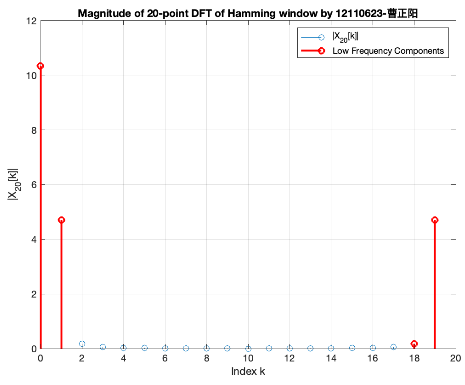

Fig. 1. DFT Magnitude Plot of $|X_{20}[k]|$

Here I'm using a red line to identify the low-frequency components, since the DFT of a real-valued signal is symmetric, the positive frequencies are mirrored in the negative frequencies around $\frac{N}{2}$, I choose four most significant four value as low-frequency components.

2) DTFTsample fftshift(): We can get the function following the steps below:

- Calculate the DTFT samples using the DTFTsum() function.
- Compute the frequency vector w using the relationship: w = (2 * pi / N) * k.
- Adjust w to ensure it ranges from -pi to pi.
- Shift the DTFT samples X and the frequency vector w using fftshift().

The function is as follows:

```
function [X, w] = DTFTsamples(x)
    % Calculate the length
    N = length(x);

    % Compute the DFT samples using the
       DFTsum() function
    [X_DFT, k] = DFTsum(x);

    % Compute the frequency vector w
    w = (2 * pi / N) * k;

    % Adjust w to ensure it ranges from -
       pi to pi
    w(w >= pi) = w(w >= pi) - 2 * pi;

    % Shift the DTFT samples X and the
       frequency vector w
    X = fftshift(X_DFT);
    w = fftshift(w);
end
```

The result is as follows:

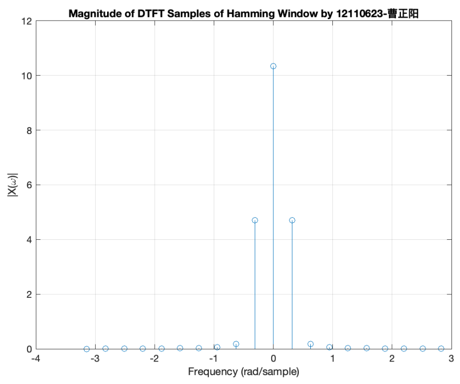

Fig. 2. DTFT Magnitude Plot of $|X_{20}[k]|$

Basically, it's just some editing about Fig. 1

B. Zero Padding

Since the DTFT sample is a discrete signal, we may stem it intuitively. However, the use of stem() might make it challenging to find differences. Therefore, I use plot() in my final code.

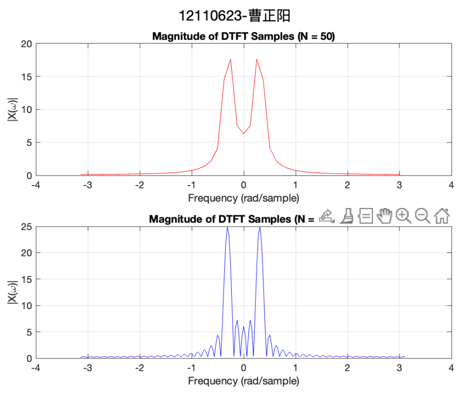

Fig. 3. DTFT Magnitude Plot of $|X_{20}[k]|$

Comparing these two with the real DTFT of x compute with DTFT.m provided in lab4 :

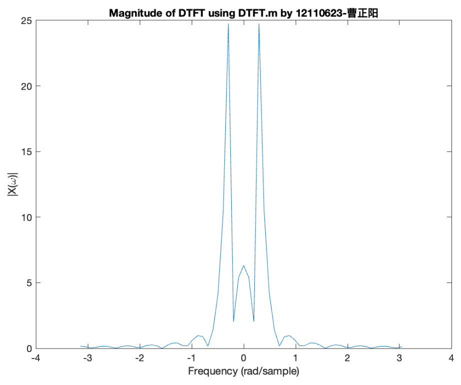

Fig. 4. DTFT Magnitude Plot of $|X_{20}[k]|$ using DTFT.m

a) Which plot looks more like the true DTFT: Since the zero-padding result closely resembles the DTFT outcome. It is easy to find that zero padding does produce a finer sampling of the DTFT.

b) Why the plots look so different: Someone from UZH [1] explains why zero padding makes DFT work better:" The addition of zeros to the end of the time-domain waveform does not improve the underlying frequency resolution associated with the time-domain signal. The only way to improve the frequency resolution of the time-domain signal is to increase the acquisition time and acquire longer time records."

So, after using zero padding, the effect on DFT is adding the sample points on the frequency domain, the reason why these two plots look different is that the one without zero-padding ends up with poor sampling of DTFT. They are the result of sampling the same signal with varying sampling points.

III. The Fast Fourier Transform Algorithm

A. Implementation of Divide-and-Conquer DFT

a) dcDFT: Follow the steps:

- Check if the length of x is even
- Divide the input signal into even and odd parts
- Compute DFT of even and odd parts
- Calculate twiddle factors
- Multiply the DFT result of the odd part by twiddle factors
- Combine the two DFT's to form X.

The function code is as follows

```
function X = dcDFT(x)

% Check if the length of x is even
if mod(length(x), 2) ~= 0
    error('Input vector length must be
        even dude');
end

% Divide the input signal into even and
    odd parts
x0 = x(1:2:end);  % Even indices
x1 = x(2:2:end);  % Odd indices

% Compute DFT of even and odd parts
X0 = DFTsum(x0);
X1 = DFTsum(x1);

% Calculate twiddle factors
N = length(x);
W_N = exp(-1i * 2 * pi / N * (0:N/2-1));

% Multiply the DFT result of the odd part
    by twiddle factors
X1 = X1 .* W_N;

% Combine the DFT results
X = zeros(1, N);
for k = 1:N/2
    X(k) = X0(k) + X1(k);
    X(k + N/2) = X0(k) - X1(k);
end
end
```

The result is as follows:

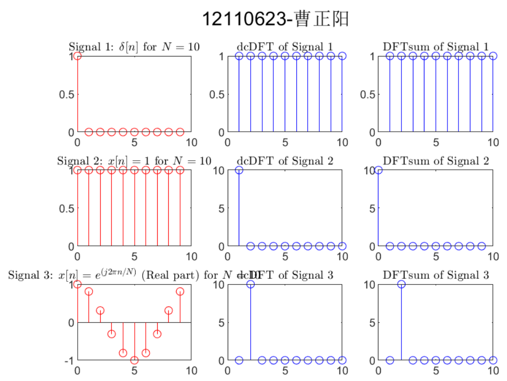

Fig. 5. Comparing Magnitude Plot of $|X_{20}[k]|$ between dcDFT and DFTsum

Signal 2 and Signal 3 show some differences when compared to DFTsum using dcDFT, suggesting a frequency shift in the results. This might be due to issues in how dcDFT is implemented.

b) the number of multiplies: The number of multiplications required in this approach for computing an N-point DFT is given by the formula

$$N + N^2/2$$

. This can be proven as follows:

Proof: It is evident that each butterfly operation requires one multiplication, resulting in a stage of N points that necessitates N multiplications. Combining this with the fact that the $N/2$-points DFTsum operation requires $(N/2)^2$ multiplications since there's two length N for loop inside the function. So the total number of multiplications becomes $N + 2 \times (N/2)^2 = N + N^2/2$.

As for considering complex number as the form: "$xi + y$", each complex multiplies need 2 multiplies instead, one for the real part, aother for the Imaginary part. So the final result is

$$2N + 2 \times N^2$$

B. Divide and Conquer

a) function: For FFT2(), it just simply write a function following the equation. As for the FFT4() and FFT8(), I follow the following steps to finish the function:

- Divide the input into even and odd parts
- Compute 2-point or 4-point FFTs for even and odd parts
- Multiply by twiddle factors
- Combine the results

The final function is as follows:

```
function X = FFT2(x)
X = zeros(1, 2);
X(1) = x(1) + x(2);
X(2) = x(1) - x(2);
end

function X = FFT4(x)
X = zeros(1, 4);

% Divide the input into even and odd
    parts
x_even = x(1:2:end);
x_odd = x(2:2:end);

% Compute 2-point FFTs for even and odd
    parts
X_even = FFT2(x_even);
X_odd = FFT2(x_odd);

% Multiply by twiddle factors
twiddle = exp(-1i * 2 * pi / 4 * (0:1));
X_odd = X_odd .* twiddle;

% Combine the results
X(1:2) = X_even + X_odd;
X(3:4) = X_even - X_odd;
end

function X = FFT8(x)
X = zeros(1, 8);

% Divide the input into even and odd
    parts
x_even = x(1:2:end);
x_odd = x(2:2:end);

% Compute 4-point FFTs for even and odd
    parts
X_even = FFT4(x_even);
X_odd = FFT4(x_odd);

% Multiply by twiddle factors
twiddle = exp(-1i * 2 * pi / 8 * (0:3));
X_odd = X_odd .* twiddle;

% Combine the results
X(1:4) = X_even + X_odd;
X(5:8) = X_even - X_odd;
end
```

b) list the output of FFT8 for the case 1: Here is the plot:

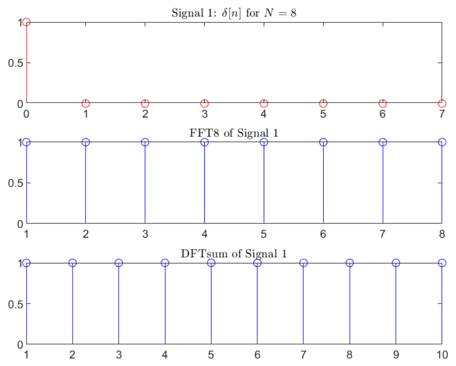

Fig. 6. Magnitude Plot of signal 1 using FFT8()

And other signal also been calculated:

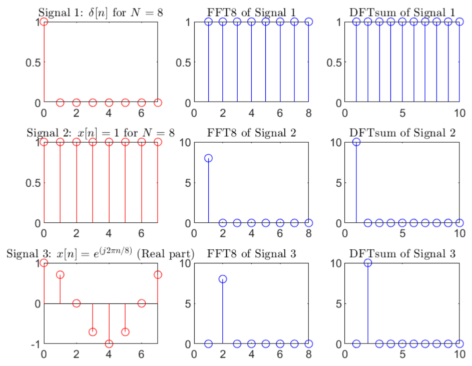

Fig. 7. Magnitude Plot of all signal using FFT8()

Again, there exists some differences when compared to DFTsum using FFT8(), suggesting a frequency shift in the results. This might be due to issues in how FFT8() is implemented.

c) calculate the total number of multiplies: The total number of multiplications by twiddle factors required for an 8-point FFT (a multiply is a multiplication by a real or complex number) is given by $(2+2+4)\times 2 = 16$.

d) determine a formula: For $N = 2^p$ point FFT, the overall multiplications are $\frac{N}{2}\log_2 N = p2^{p-1}$. Compared to the number of multiplications required for direct implementation ($N^2 = 4^p$), when $p = 10$, FFT only needs $10 \times 2^9 = 5120$ multiplications, while the direct method needs $4^{10} = 1048576$ multiplications.

C. Recursive FFT Stage

a) function: The step of writing the function is similar to FFT8() or FFT4():

- Decide use FFT2 or not
- Divide the input into even and odd parts
- Compute FFTs for even and odd parts
- Multiply by twiddle factors recursively
- Combine the results

The final function is as follows:

```
function X = fft_stage(x)
N = length(x);

% Base case: if N=2, compute the 2-pt DFT
if N == 2
    X = zeros(1, 2);
    X(1) = x(1) + x(2);
    X(2) = x(1) - x(2);
    return;
end

% Recursive case: perform FFT steps
x_even = x(1:2:end);
x_odd = x(2:2:end);

X_even = fft_stage(x_even);
X_odd = fft_stage(x_odd);

twiddle = exp(-1i * 2 * pi / N * (0:N
    /2-1));
X_odd = X_odd .* twiddle;

X = zeros(1, N);
X(1:N/2) = X_even + X_odd;
X(N/2+1:N) = X_even - X_odd;
end
```

b) result plot: The plot shows that results of fft_stage() is same as FFT8(), which price that fft_stage() is right.

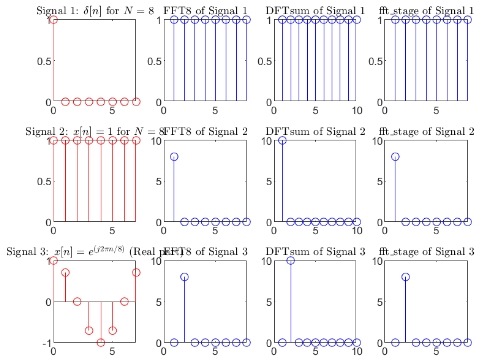

Fig. 8. Comparing Magnitude Plot of $|X_{20}[k]|$ between FFT8, DFTsum and fft_stage

IV. Conclusion

This document provides a detailed exploration of digital signal processing with a focus on the Discrete Fourier Transform (DFT) and the Fast Fourier Transform (FFT). The analysis includes a discussion on frequency range shifting and zero padding effects on signals, followed by an in-depth examination of FFT algorithms, such as Divide-and-Conquer DFT and recursive FFT stages. MATLAB code snippets and graphical representations accompany the explanations. The report concludes with a comprehensive summary of key findings, emphasizing the computational efficiency of various algorithms and highlighting the advantages of FFT over direct implementation in terms of required multiplications. Overall, the document serves as a valuable resource for understanding and implementing signal processing techniques.

References

[1] Zero Padding, Zero Padding. Available at: https://www.physik.uzh.ch/local/teaching/SPI301/LV-2015-Help/lvanlsconcepts.chm/lvac_Zero_Padding.html (Accessed: 05 December 2023).

Appendix

Here are listings of the source code provided.

A. Shifting the Frequency Range

```
% 5.2.1 Shifting the Frequency Range
N = 20;
x = hamming(N);
[X, k] = DFTsum(x);
figure;
stem(k, abs(X));
xlabel('Index k');
ylabel('|X_{20}[k]|');
title('Magnitude of 20-point DFT of
    Hamming window by 12110623-
    caozhengyang');
grid on;

N = 20;
x = hamming(N);

[X_DTFT, w_DTFT] = DTFTsamples(x);

figure;
stem(w_DTFT, abs(X_DTFT));
xlabel('Frequency (rad/sample)');
ylabel('|X(\omega)|');
title('Magnitude of DTFT Samples of
    Hamming Window by 12110623-
    caozhengyang');
grid on;
```

B. Zero Padding

```
% 5.2.2 Zero Padding
clear;
% Generate the finite-duration signal x[n
    ] for N1 = 50
n1 = 0:49;
x_n1 = sin(0.1 * pi * n1);

% Compute DTFT samples for N1 = 50
[X_DTFT_50, w_DTFT_50] = DTFTsamples(x_n1
    );

% Generate the finite-duration signal x[n
    ] for N2 = 200
n2 = 0:199;
n2(n2>=49) = 0; % explain that sin(0) = 0
x_n2 = sin(0.1 * pi * n2);

% Compute DTFT samples for N2 = 200
[X_DTFT_200, w_DTFT_200] = DTFTsamples(
    x_n2);

% Plot the magnitude of DTFT samples
    versus frequency for N1 = 50
figure;
subplot(2, 1, 1);
plot(w_DTFT_50, abs(X_DTFT_50), 'r');
xlabel('Frequency (rad/sample)');
ylabel('|X(\omega)|');
title('Magnitude of DTFT Samples (N = 50)
    ');
grid on;

% Plot the magnitude of DTFT samples
    versus frequency for N2 = 200
subplot(2, 1, 2);
plot(w_DTFT_200, abs(X_DTFT_200), 'b');
xlabel('Frequency (rad/sample)');
ylabel('|X(\omega)|');
title('Magnitude of DTFT Samples (N =
    200)');
grid on;

sgtitle('12110623-caozhengyang');
```

C. Implementation of Divide-and-Conquer DFT

```
% 5.3.1 Implementation of Divide-and-
    Conquer DFT
N = 10;

% Signal 1: x[n] = delta[n] for N = 10
x1 = zeros(1, N);
x1(1) = 1;  % Impulse at n=0

% Signal 2: x[n] = 1 for N = 10
x2 = ones(1, N);

% Signal 3: x[n] = exp(j2*pi*n/N) for N =
     10
n = 0:N-1;
x3 = exp(1i * 2 * pi * n / N);


% Test dcDFT function
X1 = dcDFT(x1);
X2 = dcDFT(x2);
X3 = dcDFT(x3);
X1_DFTsum = DFTsum(x1);
X2_DFTsum = DFTsum(x2);
X3_DFTsum = DFTsum(x3);

% Plot the results
figure;

subplot(3, 3, 1);
stem(n, x1, 'r');
title('Signal 1: $\delta[n]$ for $N = 10$
    ', 'Interpreter', 'latex');

subplot(3, 3, 2);
stem(fftshift(abs(X1)), 'b');
```

```
title('dcDFT of Signal 1', 'Interpreter',
     'latex');

subplot(3, 3, 3);
stem(fftshift(abs(X1_DFTsum)), 'b');
title('DFTsum of Signal 1', 'Interpreter'
    , 'latex');

subplot(3, 3, 4);
stem(n, x2, 'r');
title('Signal 2: $x[n] = 1$ for $N = 10$'
    , 'Interpreter', 'latex');

subplot(3, 3, 5);
stem(abs(X2), 'b');
title('dcDFT of Signal 2', 'Interpreter',
     'latex');

subplot(3, 3, 6);
stem(n, real(X2_DFTsum), 'b');
title('DFTsum of Signal 2', 'Interpreter'
    , 'latex');

subplot(3, 3, 7);
stem(n, real(x3), 'r');
title('Signal 3: $x[n] = e^{(j2\pi n / N)
    }$ (Real part) for $N = 10$', '
    Interpreter', 'latex');

subplot(3, 3, 8);
stem(abs(X3), 'b');
title('dcDFT of Signal 3', 'Interpreter',
     'latex');

subplot(3, 3, 9);
stem(abs(X3_DFTsum), 'b');
title('DFTsum of Signal 3', 'Interpreter'
    , 'latex');

sgtitle('12110623-caozhengyang');
```

D. Recursive Divide and Conquer fft2, fft4 and fft8

```
%% 5.3.2 Recursive Divide and Conquer
    fft2, fft4 and fft8
% Test FFT8 function
N = 8;

% Signal 1: x[n] = delta[n] for N = 10
x1 = zeros(1, N);
x1(1) = 1;  % Impulse at n=0

% Signal 2: x[n] = 1 for N = 10
x2 = ones(1, N);

% Signal 3: x[n] = exp(j2*pi*n/N) for N =
     10
n = 0:N-1;
x3 = exp(1i * 2 * pi * n / N);


% Test FFT8 function
X1_FFT8 = FFT8(x1);
X2_FFT8 = FFT8(x2);
X3_FFT8 = FFT8(x3);

% Plot the results
figure;

subplot(3, 3, 1);
stem(n, x1, 'r');
title('Signal 1: $\delta[n]$ for $N = 8$'
    , 'Interpreter', 'latex');

subplot(3, 3, 2);
stem(abs(X1_FFT8), 'b');
title('FFT8 of Signal 1', 'Interpreter',
    'latex');

subplot(3, 3, 3);
stem(abs(X1_DFTsum), 'b'); % need run
    5.3.1 first to get X1_DFsum
title('DFTsum of Signal 1', 'Interpreter'
    , 'latex');

subplot(3, 3, 4);
stem(n, x2, 'r');
title('Signal 2: $x[n] = 1$ for $N = 8$',
     'Interpreter', 'latex');

subplot(3, 3, 5);
stem(abs(X2_FFT8), 'b');
title('FFT8 of Signal 2', 'Interpreter',
    'latex');

subplot(3, 3, 6);
stem(real(X2_DFTsum), 'b');
title('DFTsum of Signal 2', 'Interpreter'
    , 'latex');

subplot(3, 3, 7);
stem(n, real(x3), 'r');
title('Signal 3: $x[n] = e^{(j2\pi n / 8)
    }$ (Real part)', 'Interpreter', 'latex
    ');

subplot(3, 3, 8);
stem(abs(X3_FFT8), 'b');
title('FFT8 of Signal 3', 'Interpreter',
    'latex');
```

```
subplot(3, 3, 9);
stem(abs(X3_DFTsum), 'b');
title('DFTsum of Signal 3', 'Interpreter'
    , 'latex');

% Display the results
disp('FFT8 result for x[n] = 1 for N = 8:
    ');
disp(X2_FFT8);
```

E. Recursive Divide and Conquer fft_stage

```
%% 5.3.2 Recursive Divide and Conquer
    fft_stage
% Test fft_stage on the three 8-pt
    signals
X1_fft_stage = fft_stage(x1);
X2_fft_stage = fft_stage(x2);
X3_fft_stage = fft_stage(x3);

% Plot the results
figure;

% Signal 1
subplot(3, 4, 1);
stem(n, x1, 'r');
title('Signal 1: $\delta[n]$ for $N = 8$'
    , 'Interpreter', 'latex');

subplot(3, 4, 2);
stem(abs(X1_FFT8), 'b');
title('FFT8 of Signal 1', 'Interpreter',
    'latex');
ylim([0, 1]); % Set ylim

subplot(3, 4, 3);
stem(abs(X1_DFTsum), 'b');
title('DFTsum of Signal 1', 'Interpreter'
    , 'latex');
ylim([0, 1]); % Set ylim

subplot(3, 4, 4);
stem(abs(X1_fft_stage), 'b');
title('fft\_stage of Signal 1', '
    Interpreter', 'latex');
ylim([0, 1]); % Set ylim

% Signal 2
subplot(3, 4, 5);
stem(n, x2, 'r');
title('Signal 2: $x[n] = 1$ for $N = 8$',
     'Interpreter', 'latex');

subplot(3, 4, 6);
stem(abs(X2_FFT8), 'b');
title('FFT8 of Signal 2', 'Interpreter',
    'latex');
ylim([0, 10]); % Set ylim

subplot(3, 4, 7);
stem(real(X2_DFTsum), 'b');
title('DFTsum of Signal 2', 'Interpreter'
    , 'latex');
ylim([0, 10]); % Set ylim

subplot(3, 4, 8);
stem(abs(X2_fft_stage), 'b');
title('fft\_stage of Signal 2', '
    Interpreter', 'latex');
ylim([0, 10]); % Set ylim

% Signal 3
subplot(3, 4, 9);
stem(n, real(x3), 'r');
title('Signal 3: $x[n] = e^{(j2\pi n / 8)
    }$ (Real part)', 'Interpreter', 'latex
    ');

subplot(3, 4, 10);
stem(abs(X3_FFT8), 'b');
title('FFT8 of Signal 3', 'Interpreter',
    'latex');
ylim([0, 10]); % Set ylim

subplot(3, 4, 11);
stem(abs(X3_DFTsum), 'b');
title('DFTsum of Signal 3', 'Interpreter'
    , 'latex');
ylim([0, 10]); % Set ylim

subplot(3, 4, 12);
stem(abs(X3_fft_stage), 'b');
title('fft\_stage of Signal 3', '
    Interpreter', 'latex');
ylim([0, 10]); % Set ylim

% Display the results
disp('FFT8 result for x[n] = 1 for N = 8:
    ');
disp(X2_FFT8);
```

F. FUNCTIONs

```
%% FUNCTIONs

function [X, w] = DTFTsamples(x)
N = length(x);
[X_DFT, k] = DFTsum(x);
w = (2 * pi / N) * k;
w(w >= pi) = w(w >= pi) - 2 * pi;
X = fftshift(X_DFT);
```

```
w = fftshift(w);
end

function X = dcDFT(x)
% Check if the length of x is even
if mod(length(x), 2) ~= 0
    error('Input vector length must be
       even dude');
end

%j = sqrt(-1);
%i == j %turns out that we can just use i
x0 = x(1:2:end);  % Even because the
   Matlab index starts from fucking 1
x1 = x(2:2:end);  % Odd

X0 = DFTsum(x0);
X1 = DFTsum(x1);

N = length(x);
W_N = exp(-1i * 2 * pi / N * (0:N/2-1));

X1 = X1 .* W_N;
X = zeros(1, N);
for k = 1:N/2
    X(k) = X0(k) + X1(k);
    X(k + N/2) = X0(k) - X1(k);
end
end

% Function to compute the 2-point FFT
function X = FFT2(x)
X = zeros(1, 2);
X(1) = x(1) + x(2);
X(2) = x(1) - x(2);
end

% Function to compute the 4-point FFT
function X = FFT4(x)
X = zeros(1, 4);

% Divide the input into even and odd
   parts
x_even = x(1:2:end);
x_odd = x(2:2:end);

% Compute 2-point FFTs for even and odd
   parts
X_even = FFT2(x_even);
X_odd = FFT2(x_odd);

% Multiply by twiddle factors
twiddle = exp(-1i * 2 * pi / 4 * (0:1));
X_odd = X_odd .* twiddle;

% Combine the results
X(1:2) = X_even + X_odd;
X(3:4) = X_even - X_odd;
end

% Function to compute the 8-point FFT
function X = FFT8(x)
X = zeros(1, 8);

% Divide the input into even and odd
   parts
x_even = x(1:2:end);
x_odd = x(2:2:end);

% Compute 4-point FFTs for even and odd
   parts
X_even = FFT4(x_even);
X_odd = FFT4(x_odd);

% Multiply by twiddle factors
twiddle = exp(-1i * 2 * pi / 8 * (0:3));
X_odd = X_odd .* twiddle;

% Combine the results
X(1:4) = X_even + X_odd;
X(5:8) = X_even - X_odd;
end


function X = fft_stage(x)
N = length(x);

% Base case: if N=2, compute the 2-pt DFT
if N == 2
    X = zeros(1, 2);
    X(1) = x(1) + x(2);
    X(2) = x(1) - x(2);
    return;
end

% Recursive case: perform FFT steps
x_even = x(1:2:end);
x_odd = x(2:2:end);

X_even = fft_stage(x_even);
X_odd = fft_stage(x_odd);

twiddle = exp(-1i * 2 * pi / N * (0:N
   /2-1));
X_odd = X_odd .* twiddle;

X = zeros(1, N);
X(1:N/2) = X_even + X_odd;
X(N/2+1:N) = X_even - X_odd;
end
```