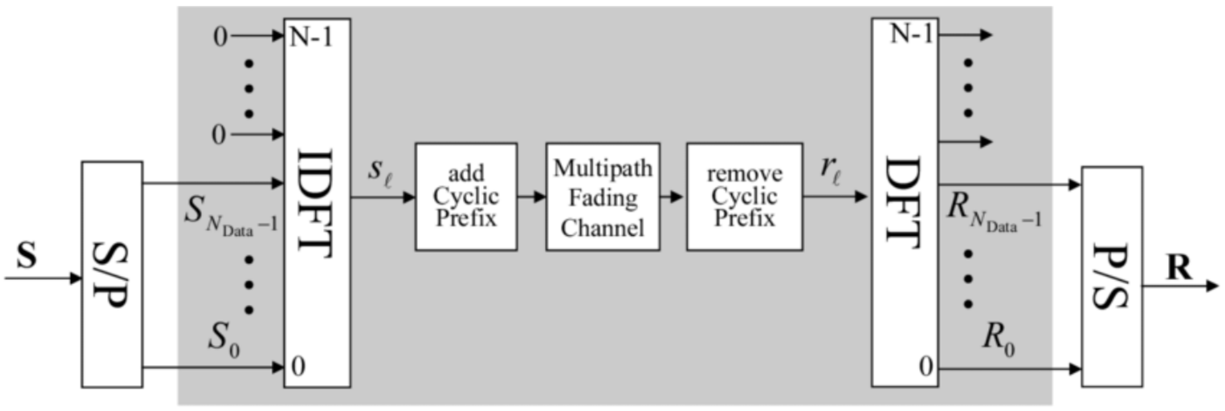
ADSP final - OFDM implementation using python

r12921a15 趙昱森

1. OFDM introduction:

正交頻分多工 (Orthogonal Frequency Division Multiplexing, OFDM) 是一種數位多重傳輸技術，廣泛應用於現代無線通訊系統中，如Wi-Fi、4G LTE 和 5G。

Workflow chart:



OFDM的基本流程如下：

Tx 端：

串行轉並行 (Serial to Parallel)

- 將串行數據流轉換為並行數據塊，準備進行調變。

逆離散傅立葉變換 (Inverse Discrete Fourier Transform, IDFT)

- 將並行數據流轉換到頻域，形成 Orthogonal 的 subcarriers。這裡的子載波在頻域上是正交的，從而避免相互干擾。

添加循環前綴 (Add Cyclic Prefix)

- 在每個OFDM symbol 前添加 Cyclic Prefix，以對抗多徑干擾和 symbol 間干擾，讓 channel 變成 circular convolution。循環前綴是一段複製的數據，來自 OFDM symbol 的尾部。

Channel 端：

多徑通道 (Multi-path Channel)

- Signal 經過 Multi-path channel 傳輸，可能受到 multi-path 效應、噪音和干擾的影響。

Rx 端：

去除循環前綴 (Remove Cyclic Prefix)

- 在進行任何 signal 處理之前，首先去除循 cyclic prefix，以恢復原始的OFDM symbol。

離散傅立葉變換 (Discrete Fourier Transform, DFT)

- 將經過 channel 傳輸的OFDM symbol 轉回 frequency domain QAM data。

並行轉串行 (Parallel to Serial, P/S)

- 將並行數據流轉換回串行數據流，恢復原始的數據訊息。

2. Code Implementation Using Python:

Step 1 to step 4 are Tx side, Step 5 is channel simulation, and step 6 to

Step 1. Subcarriers and pilot signals settings:

```
K = 64  # Number of OFDM subcarriers
CP = K // 4  # Length of the cyclic prefix: 25% of the block
P = 8  # Number of pilot carriers per OFDM block
pilot_value = 3 + 3j  # The known value each pilot transmits
```

- `K` 是 OFDM 的子載波數量，這裡設為 64。
- `CP` 是循環前綴的長度，這裡設為 25% 的 OFDM symbol長度。

- `P` 是每個 OFDM 塊中的導頻載波數量，這裡設為 8。
- `pilot_value` 是每個導頻載波傳輸的已知值，這裡設為複數 3 + 3j。

```
all_carriers = np.arange(K)  # Indices of all subcarriers ([0, 1
```

- `all_carriers` 是所有子載波的索引，從 0 到 K-1。

```
pilot_carriers = all_carriers[::K // P]
pilot_carriers = np.hstack([pilot_carriers, np.array([all_carrie
P += 1  # Update the number of pilot carriers
```

- Pilot carriers 每隔 PK 個 subcarrier 設置一次。

  `K/P`

- 最後一個載波也設為導頻載波，以方便 channel estimation。
- 導頻載波數量更新為 P + 1。

```
data_carriers = np.delete(all_carriers, pilot_carriers)
```

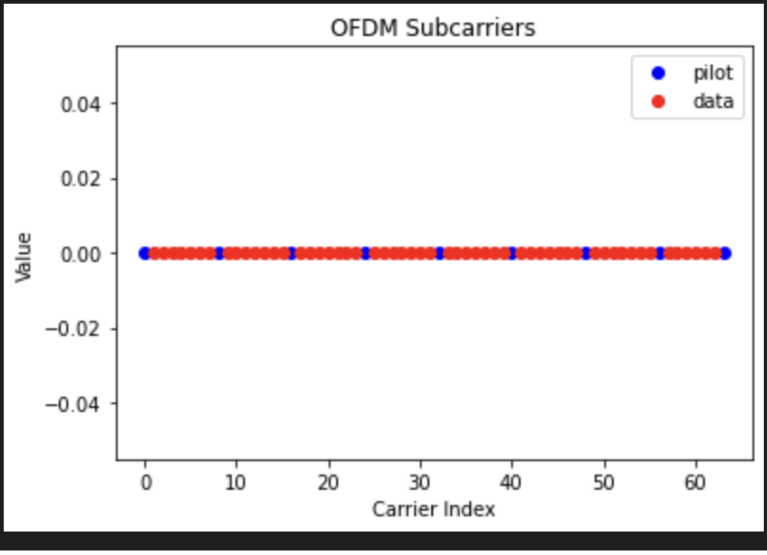
- `data_carriers` 是除去導頻載波後的所有數據載波。

顯示內容以及繪圖：

```
print(f"all_carriers:   {all_carriers}")
print(f"pilot_carriers: {pilot_carriers}")
print(f"data_carriers:  {data_carriers}")

plt.plot(pilot_carriers, np.zeros_like(pilot_carriers), 'bo', la
plt.plot(data_carriers, np.zeros_like(data_carriers), 'ro', labe
plt.xlabel('Carrier Index')
plt.ylabel('Value')
plt.title('OFDM Subcarriers')
plt.legend()
plt.show()
```

結果：

```
all_carriers:   [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63]
pilot_carriers: [ 0  8 16 24 32 40 48 56 63]
data_carriers:  [ 1  2  3  4  5  6  7  9 10 11 12 13 14 15 17 18 19 20 21 22 23 25 26 27
 28 29 30 31 33 34 35 36 37 38 39 41 42 43 44 45 46 47 49 50 51 52 53 54
 55 57 58 59 60 61 62]
```



Step 2. 16QAM and constellation settings:

```
mu = 4  # Bits per symbol (i.e., 16QAM)
payload_bits_per_OFDM = len(data_carriers) * mu  # Number of pay
```

- `mu` 是每個 symbol 的比特數，這裡設為 4，比特代表 16QAM 調製方案（2^4 = 16）。
- `payload_bits_per_OFDM` 是每個 OFDM symbol 中有效載荷比特的數量，計算方式是數據載波數量乘以每個 symbol 的比特數。

```
mapping_table = {
    (0, 0, 0, 0): -3 - 3j,
    (0, 0, 0, 1): -3 - 1j,
    (0, 0, 1, 0): -3 + 3j,
    (0, 0, 1, 1): -3 + 1j,
    (0, 1, 0, 0): -1 - 3j,
    (0, 1, 0, 1): -1 - 1j,
    (0, 1, 1, 0): -1 + 3j,
    (0, 1, 1, 1): -1 + 1j,
    (1, 0, 0, 0):  3 - 3j,
    (1, 0, 0, 1):  3 - 1j,
    (1, 0, 1, 0):  3 + 3j,
    (1, 0, 1, 1):  3 + 1j,
    (1, 1, 0, 0):  1 - 3j,
    (1, 1, 0, 1):  1 - 1j,
    (1, 1, 1, 0):  1 + 3j,
    (1, 1, 1, 1):  1 + 1j
}
```

- `mapping_table` 是 16QAM 的映射表，將 4 比特的二進制組合映射到對應的複數值（即 IQ 標誌），形成 16QAM 星座圖。

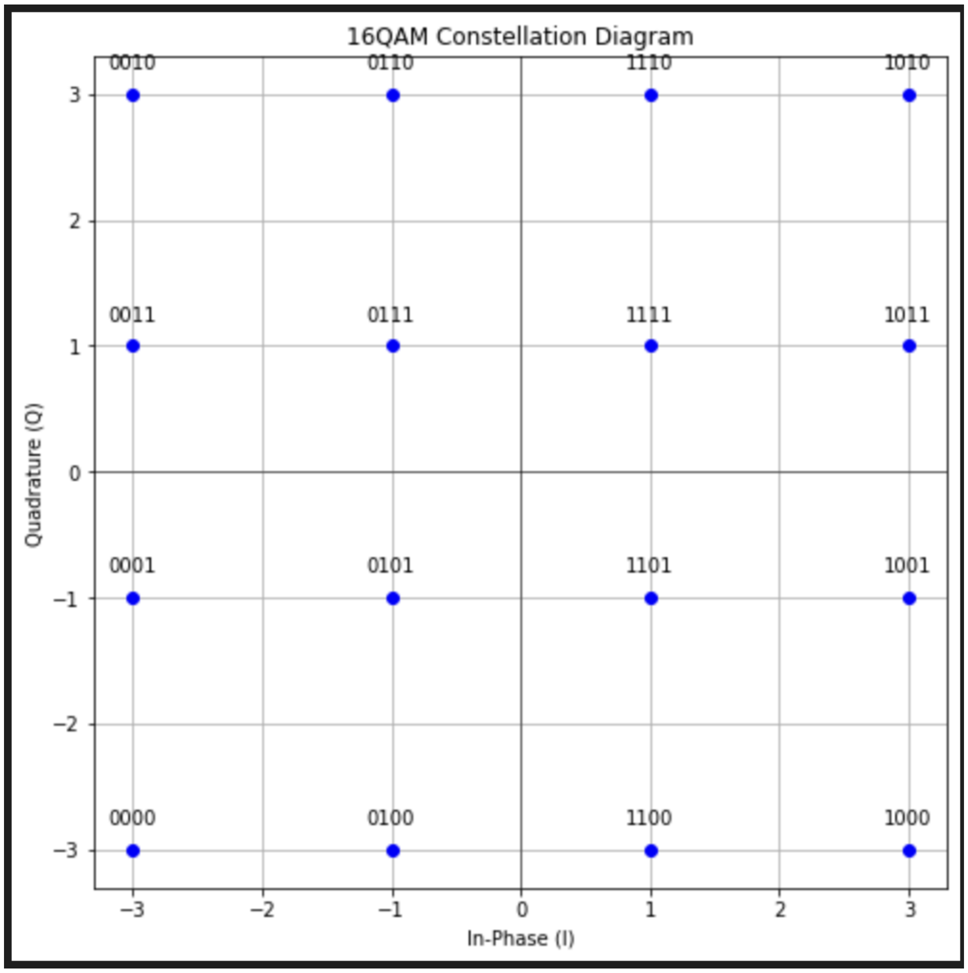
```
plt.figure(figsize=(8, 8))
for b in mapping_table:
    Q = mapping_table[b]
    plt.plot(Q.real, Q.imag, 'bo')
    plt.text(Q.real, Q.imag + 0.2, "".join(map(str, b)), ha='cen
```

- 創建一個 8x8 的圖形窗口。
- 遍歷映射表，將每個複數值的實部 (Q.real) 和虛部 (Q.imag) 作為點繪製在圖上，標記為藍色圓點 ('bo')。
- 使用 `plt.text` 在每個點的旁邊標註對應的二進制組合。

繪圖：

```
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(True)
plt.title('16QAM Constellation Diagram')
plt.xlabel('In-Phase (I)')
plt.ylabel('Quadrature (Q)')
plt.show()
```

結果：



Step 3. 16QAM mapping and channel response settings:

```
# Demapping table
demapping_table = {v: k for k, v in mapping_table.items()}
```

- `demapping_table` 是解調映射表，它將每個複數值（IQ symbol）mapping 回對應的 4 bit 二進制組合。這樣可以在 demodulation 時從接收到的 symbol 恢復原始 bit。

```python
# Channel response settings
channel_response = np.array([1, 0, 0.3 + 0.3j])  # The impulse r
H_exact = np.fft.fft(channel_response, K)
```

- `channel_response` 是無線通道的脈衝響應，這裡設定為一個具有 multipath effect 的 channel。第一個值為 1，代表直達路徑，第三個值為 0.3 + 0.3j，代表一個經過反射的路徑。
- `H_exact` 是該通道的頻率響應，通過對通道脈衝響應進行快速傅立葉變換 (FFT) 得到。

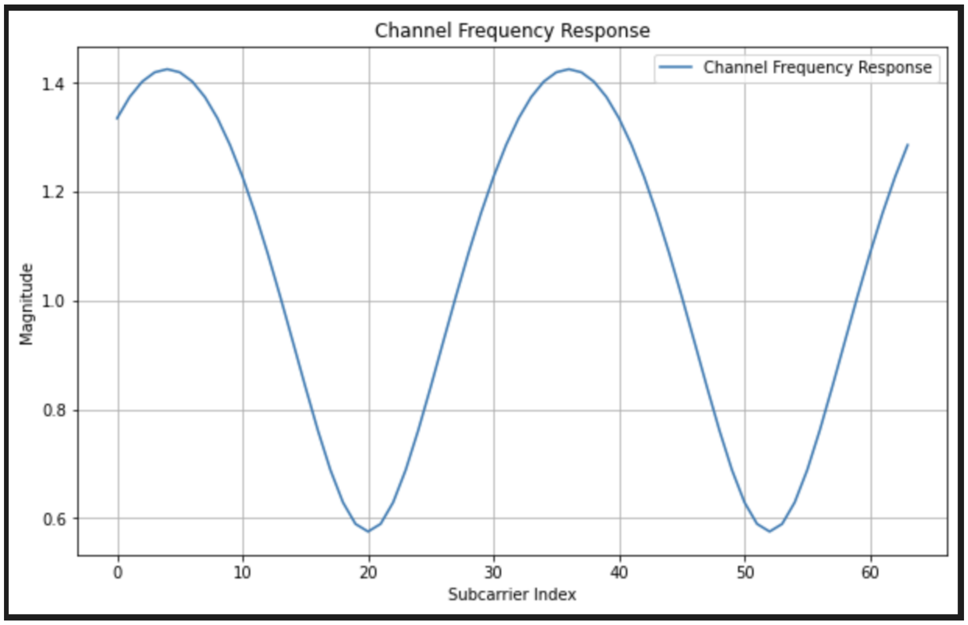
```python
# Signal-to-noise ratio (SNR) in dB at the receiver
SNR_db = 25
```

- SNR 在這裡設置為 25 db。

繪製：

```python
# Plot the frequency response of the channel
plt.figure(figsize=(10, 6))
plt.plot(all_carriers, abs(H_exact), label='Channel Frequency Re
plt.title('Channel Frequency Response')
plt.xlabel('Subcarrier Index')
plt.ylabel('Magnitude')
plt.grid(True)
plt.legend()
plt.show()
```

結果：



Step 4. Generate random bits and form OFDM signals:

Generate random bit data:

```
bits = np.random.binomial(n=1, p=0.5, size=payload_bits_per_OFDI
```

- 生成 `payload_bits_per_OFDM` 個隨機比特，這些比特服從二項分佈 (binomial distribution)，即每個比特為 0 或 1，概率均為 0.5。

Serial to parallel function:

```
def serial_to_parallel(bits):
    return bits.reshape((len(data_carriers), mu))

# Convert bits to parallel
bits_SP = serial_to_parallel(bits)
```

- 定義 `serial_to_parallel` 函數，將串行比特流轉換為並行的比特塊，每個塊的大小為 `len(data_carriers) x mu`。

Define bit-to-16QAM mapping function:

```
def mapping(bits):
    return np.array([mapping_table[tuple(b)] for b in bits])

# Convert bits to QAM symbols
QAM = mapping(bits_SP)
```

- 定義 `mapping` 函數，將比特塊映射為對應的 16QAM symbol。

Create OFDM symbols using QAM payload:

```
def ofdm_symbol(QAM_payload):
    symbol = np.zeros(K, dtype=complex)  # Initialize the overal
    symbol[pilot_carriers] = pilot_value  # Allocate the pilot s
    symbol[data_carriers] = QAM_payload  # Allocate the data sub
    return symbol

# Create an OFDM symbol with the QAM payload
OFDM_data = ofdm_symbol(QAM)
```

- 定義 `ofdm_symbol` 函數，初始化一個大小為 `K` 的複數數組來存放 OFDM symbol。
- 在導頻載波位置分配導頻值，在數據載波位置分配 QAM 載荷。

Define IDFT function:

```
def idft(ofdm_data):
    return np.fft.ifft(ofdm_data)

# Perform IDFT to get time-domain OFDM samples
OFDM_time = idft(OFDM_data)
```

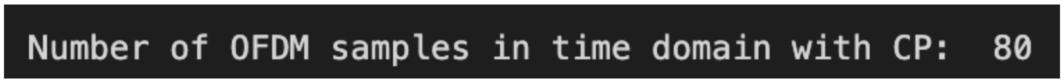
- 定義 `idft` 函數，對 OFDM symbol 進行 IDFT，將頻域數據轉換為時域數據。

Add cyclic prefix:

```
def add_cp(OFDM_time):
    cp = OFDM_time[-CP:]               # take the last CP sampl
    return np.hstack([cp, OFDM_time])  # ... and add them to the
OFDM_withCP = add_cp(OFDM_time)
print ("Number of OFDM samples in time domain with CP: ", len(OI
```

- 定義 `add_cp` 函數，取時域 OFDM 樣本的最後 `CP` 個樣本，並將它們添加到樣本的開頭，形成帶有 CP 的 OFDM symbol。
- Print 帶有 CP 的 time domain OFDM 樣本數量。

結果：



```
Number of OFDM samples in time domain with CP:  80
```

Step 5. Channel settings, transmitter side, and receiver side:

Channel settings:

```
def channel(signal):
    convolved = np.convolve(signal, channel_response)
    signal_power = np.mean(abs(convolved**2))
    sigma2 = signal_power * 10**(-SNR_db/10)  # Calculate noise

    # Generate complex noise with given variance
    noise = np.sqrt(sigma2/2) * (np.random.randn(*convolved.sha
    return convolved + noise
```

- `channel` 函數模擬 signal 通過無線通道時的影響，包括 signal 的卷積和噪聲的添加。
- `np.convolve(signal, channel_response)` 對輸入 signal 進行 convolution，模擬 multi-path effect。
- `signal_power` 計算功率。

- `sigma2` 計算噪聲功率，根據 signal power 和設定的信噪比 (SNR) 來確定。
- 生成符合 SNR 的 noise。
- 返回添加噪聲後的 signal。

Set OFDM_TX = OFDM_withCP:

```
OFDM_TX = OFDM_withCP
```

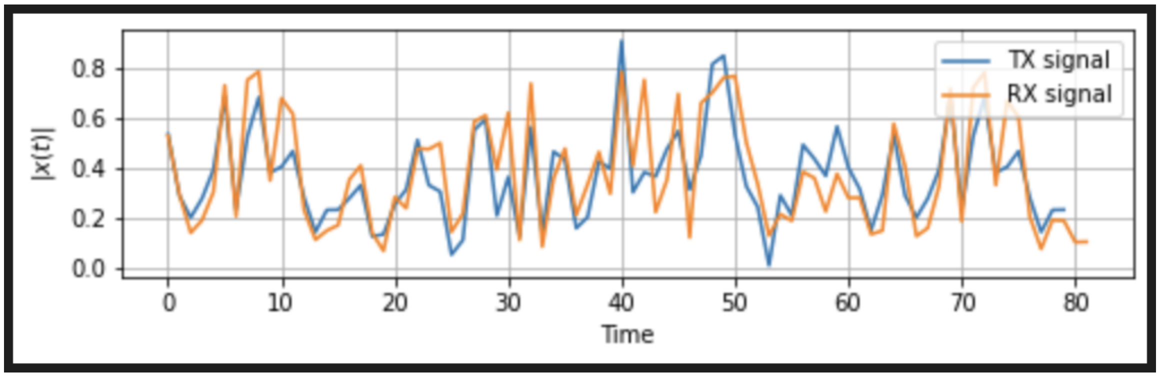
Put transmit data into channel function:

```
OFDM_RX = channel(OFDM_TX)
```

繪製：

```
# Plot the transmitted and received signals
plt.figure(figsize=(8, 2))
plt.plot(abs(OFDM_TX), label='TX signal')
plt.plot(abs(OFDM_RX), label='RX signal')
plt.legend(fontsize=10)
plt.xlabel('Time')
plt.ylabel('$|x(t)|$')
plt.grid(True)
plt.show()
```

結果：



Step 6. Rx signal processing:

Remove CP:

```
# Function to remove the Cyclic Prefix (CP) from the received si
def remove_cp(signal):
    return signal[CP:(CP + K)]

# Remove the CP from the received OFDM signal
OFDM_RX_noCP = remove_cp(OFDM_RX)
```

- `remove_cp` 函數接收一個 signal，並移除前 `CP` 個樣本。返回的 signal 長度為 `K`，從第 `CP` 個樣本開始，到第 `CP + K` 個樣本結束。

Conduct DFT to received signal, transform them back to TF domain:

```
def DFT(OFDM_RX):
    return np.fft.fft(OFDM_RX)
OFDM_demod = DFT(OFDM_RX_noCP)
```

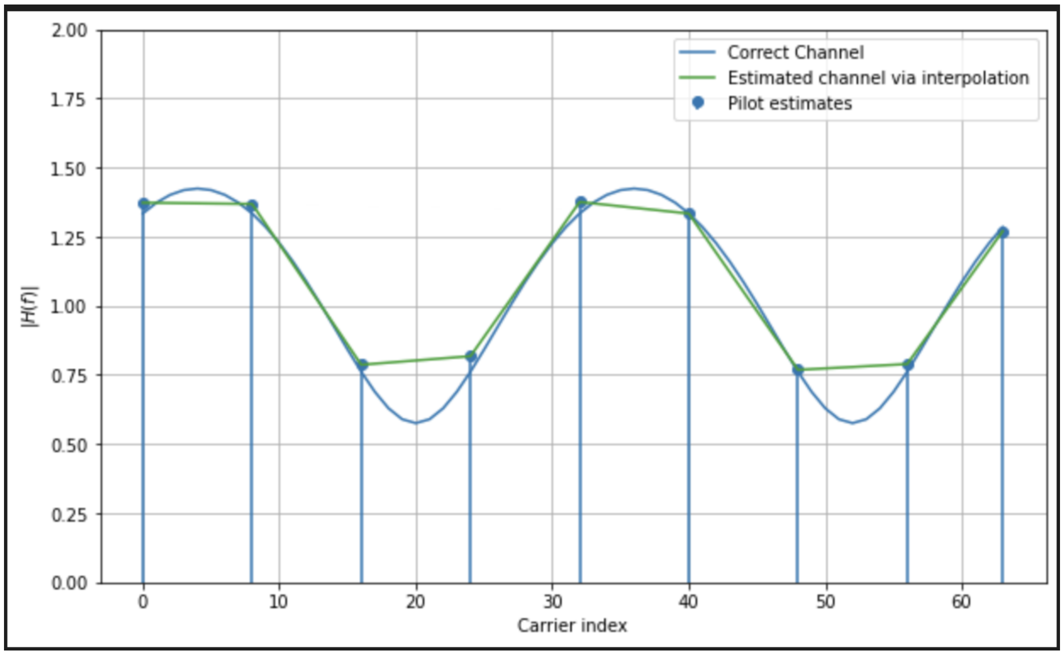
Step 7. Rx side channel estimation:

Channel estimation:

```
def channel_estimate(ofdm_demod):
    pilots = ofdm_demod[pilot_carriers]  # Extract the pilot va
    Hest_at_pilots = pilots / pilot_value  # Divide by the trans

    # Perform interpolation between the pilot carriers to get ar
    # of the channel in the data carriers. Here, we interpolate 
    Hest_abs = scipy.interpolate.interp1d(pilot_carriers, abs(He
    Hest_phase = scipy.interpolate.interp1d(pilot_carriers, np.a
    Hest = Hest_abs * np.exp(1j * Hest_phase)
    # Plot the correct and estimated channel responses
    plt.figure(figsize=(10, 6))
    plt.plot(all_carriers, abs(H_exact), label='Correct Channel
    plt.stem(pilot_carriers, abs(Hest_at_pilots), label='Pilot e
    plt.plot(all_carriers, abs(Hest), label='Estimated channel v
    plt.grid(True)
    plt.xlabel('Carrier index')
    plt.ylabel('$|H(f)|$')
    plt.legend(fontsize=10)
    plt.ylim(0, 2)
    plt.show()

    return Hest
```

- `pilots`：從接收的 OFDM symbol `ofdm_demod` 中提取導頻載波位置的值。
- `Hest_at_pilots`：將接收的導頻值除以已知的導頻值，得到導頻載波位置的通道估計。
- `Hest_abs`：使用線性插值方法對導頻載波位置的通道估計的幅度進行插值，得到所有載波位置的幅度估計。
- `Hest_phase`：使用線性插值方法對導頻載波位置的通道估計的相位進行插值，得到所有載波位置的相位估計。
- `Hest`：將插值後的幅度和相位組合，得到所有載波位置的複數通道估計。

結果：



透過 Pilot signals 大致描繪出了 Channel state information。

Step 8. channel equalization and estimated received QAM data:

Define equalize function:

```
# Function to equalize the received OFDM symbol using the estima
def equalize(ofdm_demod, hest):
    return ofdm_demod / hest
# Perform channel equalization
equalized_Hest = equalize(OFDM_demod, Hest)
```

- `equalize` 函數接收兩個參數：
    - `ofdm_demod`：經過 DFT 處理的接收 OFDM symbol。
    - `hest`：估計的通道響應。
- 通過將接收的 OFDM symbols 除以估計的通道響應，來補償通道對 signal 的影響，實現 channel equalization。

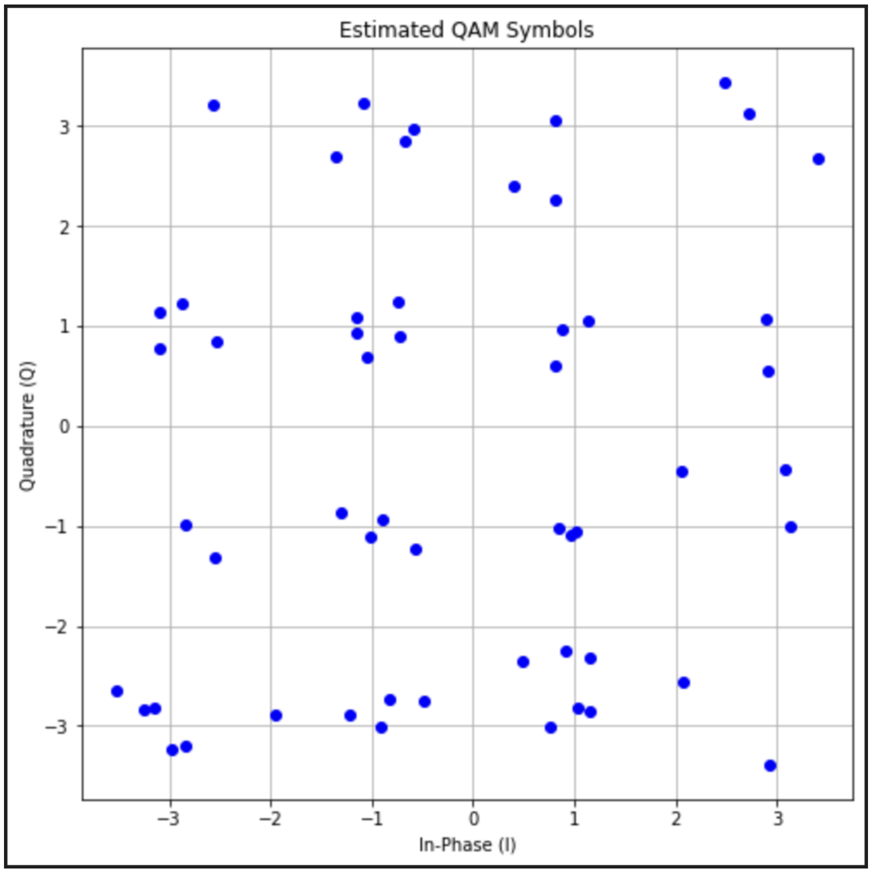
Get payload from previous results:

```
# Function to extract the payload from the equalized OFDM symbol
def get_payload(equalized):
    return equalized[data_carriers]
QAM_est = get_payload(equalized_Hest)
```

繪製：

```
# Plot the estimated QAM symbols
plt.figure(figsize=(8, 8))
plt.plot(QAM_est.real, QAM_est.imag, 'bo')
plt.title('Estimated QAM Symbols')
plt.xlabel('In-Phase (I)')
plt.ylabel('Quadrature (Q)')
plt.grid(True)
plt.show()
```

結果：



Step 9. Make QAM data into bits:

Demapping function:

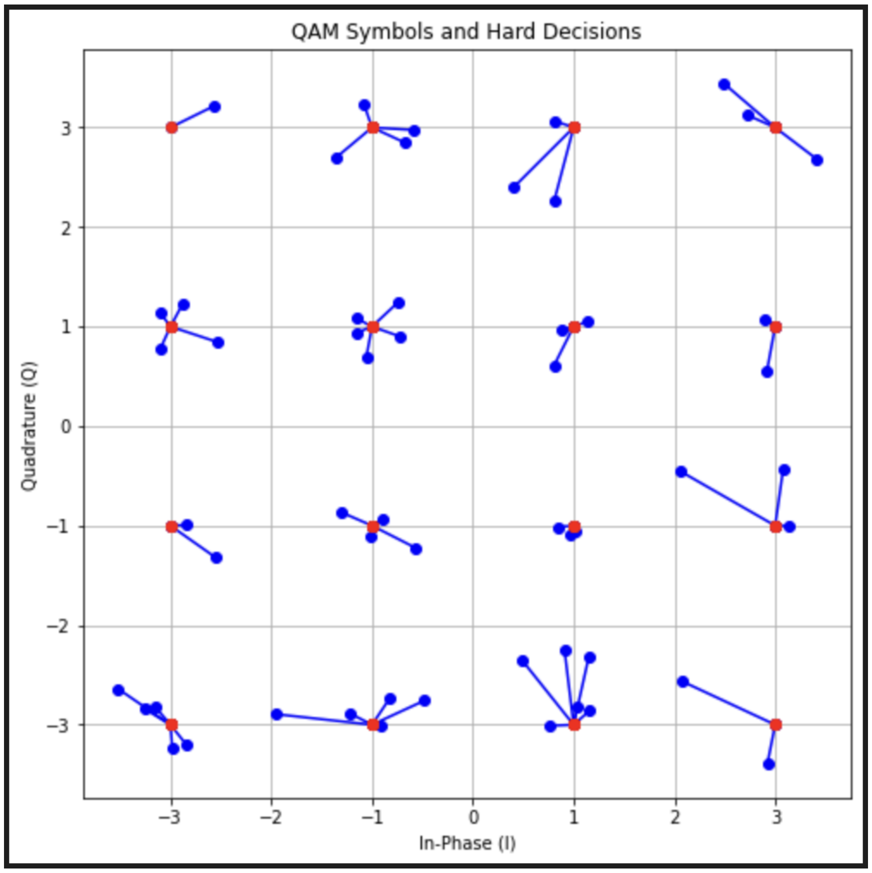
```
# Function to demap the received QAM symbols to bits
def demapping(QAM):
    # Array of possible constellation points
    constellation = np.array([x for x in demapping_table.keys()

    # Calculate distance of each RX point to each possible poin
    dists = abs(QAM.reshape((-1, 1)) - constellation.reshape((1,

    # For each element in QAM, choose the index in constellatio
    # that belongs to the nearest constellation point
    const_index = dists.argmin(axis=1)
    # Get back the real constellation point
    hard_decision = constellation[const_index]

    # Transform the constellation point into the bit groups
    return np.vstack([demapping_table[C] for C in hard_decision

# Demap the estimated QAM symbols to bits
PS_est, hard_decision = demapping(QAM_est)
```

- `constellation`：將所有可能的點存儲在一個數組中。
- `dists`：計算每個接收的 QAM symbol 與每個點之間的距離，得到一個距離矩陣。
- `const_index`：對於每個接收的 QAM symbol，選擇距離最近的點。
- `hard_decision`：根據最近的點索引獲得實際的點。
- 返回解調後的比特組和硬判決（即最近的星座點）。

繪製：

```
# Plot the QAM symbols and their nearest constellation points
plt.figure(figsize=(8, 8))
for qam, hard in zip(QAM_est, hard_decision):
    plt.plot([qam.real, hard.real], [qam.imag, hard.imag], 'b-o
plt.plot(hard_decision.real, hard_decision.imag, 'ro')
plt.title('QAM Symbols and Hard Decisions')
plt.xlabel('In-Phase (I)')
plt.ylabel('Quadrature (Q)')
plt.grid(True)
plt.show()
```

結果：



Step 10. Turn parallel data into serial data and see the results:

P/S function:

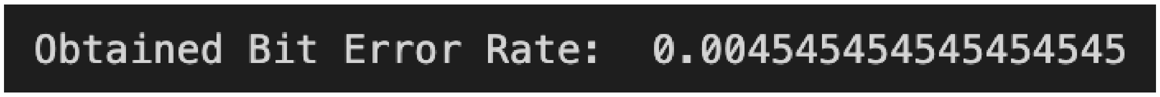
```
# Function to convert parallel bits back to serial
def parallel_to_serial(bits):
    return bits.reshape((-1,))

# Convert the estimated bits from parallel to serial
bits_est = parallel_to_serial(PS_est)
```

Calculate bit error rate:

```
# Calculate and display the bit error rate (BER)
bit_error_rate = np.sum(abs(bits - bits_est)) / len(bits)
```

```
print("Obtained Bit Error Rate: ", bit_error_rate)
```

結果：



```
Obtained Bit Error Rate:  0.004545454545454545
```

3. References:

Dammann, Armin. (2006). On Antenna Diversity Techniques for OFDM Systems.

Github resources: https://github.com/ad-astra-et-ultra/OTFS-and-OFDM-Transceivers