To properly connect the output of a crystal radio to the analog-to-digital converter (ADC) of an STM32 microcontroller, you need to consider factors such as voltage levels, impedance matching, and signal conditioning. Here's how you can calculate and design the connection:

1. \*\*Voltage Levels:\*\*

- Measure the output voltage range of the crystal radio. This typically varies based on the strength of the received signal and the sensitivity of the radio components.

- Ensure that the output voltage range of the crystal radio is compatible with the input range of the STM32 ADC. The STM32 ADC typically has a maximum input voltage of VDDA (analog supply voltage) and a minimum input voltage of 0V.

2. \*\*Impedance Matching:\*\*

- Determine the output impedance of the crystal radio. This can vary depending on the antenna, tuning circuit, and output stage of the radio.

- Match the output impedance of the crystal radio to the input impedance of the ADC. The input impedance of the STM32 ADC is typically in the range of tens of kilohms to several megohms.

3. \*\*Signal Conditioning:\*\*

- If the output voltage range of the crystal radio does not match the input range of the ADC, use voltage dividers or amplifiers to scale or amplify the signal accordingly.

- If the output impedance of the crystal radio is significantly different from the input impedance of the ADC, use impedance matching circuits such as voltage followers or buffer amplifiers to minimize signal distortion and maximize signal transfer.

4. \*\*Calculations:\*\*

- Use Ohm's law to calculate the required resistor values for voltage dividers or voltage follower circuits to scale or buffer the signal.

- Calculate the impedance matching network using the formula for impedance transformation, considering the desired impedance ratio and the impedance values of the crystal radio output and ADC input.

5. \*\*Example:\*\*

- Let's say the output voltage range of the crystal radio is 0.1V to 1.0V, and the output impedance is approximately 1 kilohm.

- The STM32 ADC has a reference voltage (VREF) of 3.3V and an input impedance of 50 kilohms.

- To scale the output voltage range to fit within the ADC's input range, you can use a voltage divider with appropriate resistor values.

- For example, to scale the output voltage by a factor of 3, you can use a resistor divider with a 2:1 ratio. Using a 10 kilohm resistor for R1 and a 5 kilohm resistor for R2 would achieve this scaling.

Once you have determined the proper connection scheme based on these calculations, you can implement the circuit using appropriate resistors, amplifiers, and other components as needed. Be sure to verify the circuit's performance through testing and calibration, adjusting component values as necessary to achieve optimal results.

Yes, you're correct. We need to include the code for ADC (Analog-to-Digital Converter) initialization and data acquisition. Here's the updated code including ADC initialization and data acquisition:

```python

import time

from pymodbus.client.sync import ModbusSerialClient as ModbusClient

import numpy as np

from scipy.signal import butter, lfilter

import pyb

# Modbus Configuration

MODBUS\_PORT = '/dev/ttyUSB0' # Specify the UART port

MODBUS\_BAUDRATE = 9600 # Specify the baud rate

MODBUS\_SLAVE\_ADDRESS = 1 # Specify the slave address of the Modbus LCD screen

# Initialize Modbus client

client = ModbusClient(method='rtu', port=MODBUS\_PORT, baudrate=MODBUS\_BAUDRATE)

client.connect()

# Function to send data to Modbus LCD screen

def send\_data\_to\_lcd(data: int) -> None:

"""

Sends processed data to Modbus LCD screen.

Parameters:

- data: Processed data to be displayed on the LCD screen (integer).

Returns:

- None.

"""

# Write data to Modbus register

client.write\_register(register\_address=0, value=data, unit=MODBUS\_SLAVE\_ADDRESS)

# Signal Processing Functions

def apply\_low\_pass\_filter(data: np.ndarray, cutoff\_freq: float, sampling\_freq: float) -> np.ndarray:

"""

Applies a low-pass filter to the input data.

Parameters:

- data: Input data array.

- cutoff\_freq: Cut-off frequency of the low-pass filter (in Hz).

- sampling\_freq: Sampling frequency of the input data (in Hz).

Returns:

- Filtered data array.

"""

nyquist\_freq = 0.5 \* sampling\_freq

normalized\_cutoff\_freq = cutoff\_freq / nyquist\_freq

b, a = butter(N=4, Wn=normalized\_cutoff\_freq, btype='low')

filtered\_data = lfilter(b, a, data)

return filtered\_data

def calculate\_mean(data: np.ndarray) -> float:

"""

Calculates the mean of the input data array.

Parameters:

- data: Input data array.

Returns:

- Mean value of the data.

"""

mean\_value = np.mean(data)

return mean\_value

def calculate\_rms(data: np.ndarray) -> float:

"""

Calculates the root mean square (RMS) of the input data array.

Parameters:

- data: Input data array.

Returns:

- RMS value of the data.

"""

rms\_value = np.sqrt(np.mean(np.square(data)))

return rms\_value

def detect\_frequency(data: np.ndarray, sampling\_freq: float) -> float:

"""

Detects the dominant frequency in the input data using FFT.

Parameters:

- data: Input data array.

- sampling\_freq: Sampling frequency of the input data (in Hz).

Returns:

- Dominant frequency in the data (in Hz).

"""

fft\_result = np.fft.fft(data)

freqs = np.fft.fftfreq(len(data), 1 / sampling\_freq)

dominant\_freq\_index = np.argmax(np.abs(fft\_result))

dominant\_freq = freqs[dominant\_freq\_index]

return abs(dominant\_freq)

def detect\_envelope(data: np.ndarray) -> np.ndarray:

"""

Detects the envelope of the input data.

Parameters:

- data: Input data array.

Returns:

- Envelope of the data.

"""

envelope = np.abs(data)

return envelope

# ADC Configuration

adc = pyb.ADC(pyb.Pin.board.X1) # Initialize ADC on pin X1

sampling\_freq = 10000 # Sampling frequency in Hz

num\_samples = 1024 # Number of samples to acquire

# Main loop for real-time monitoring

while True:

try:

# Data acquisition

for i in range(num\_samples):

buffer[i] = adc.read()

# Convert ADC readings to voltage (assuming 12-bit resolution and 3.3V reference)

voltage\_data = (np.array(buffer) / 4095) \* 3.3

# Signal processing

filtered\_data = apply\_low\_pass\_filter(voltage\_data, cutoff\_freq=10, sampling\_freq=sampling\_freq)

processed\_data\_mean = calculate\_mean(filtered\_data)

processed\_data\_rms = calculate\_rms(filtered\_data)

detected\_freq = detect\_frequency(voltage\_data, sampling\_freq=sampling\_freq)

envelope\_data = detect\_envelope(filtered\_data)

# Convert processed data to integer for Modbus transmission

processed\_data\_int\_mean = int(processed\_data\_mean)

processed\_data\_int\_rms = int(processed\_data\_rms)

detected\_freq\_int = int(detected\_freq)

max\_envelope\_value = int(np.max(envelope\_data))

# Send processed data to Modbus LCD screen

send\_data\_to\_lcd(processed\_data\_int\_mean)

send\_data\_to\_lcd(processed\_data\_int\_rms)

send\_data\_to\_lcd(detected\_freq\_int)

send\_data\_to\_lcd(max\_envelope\_value)

# Print debug messages

print("Mean sent to Modbus LCD:", processed\_data\_int\_mean)

print("RMS sent to Modbus LCD:", processed\_data\_int\_rms)

print("Detected frequency sent to Modbus LCD:", detected\_freq\_int)

print("Max envelope value sent to Modbus LCD:", max\_envelope\_value)

# Wait for a short period before sending the next data

time.sleep(1)

except Exception as e:

# Handle exceptions

print("Error:", e)

# Close Modbus connection

client.close()

```

In this updated version of the code:

1. \*\*ADC Configuration:\*\*

- Initializes the ADC on pin `X1` of the STM32 microcontroller.

- Sets the sampling frequency and number of samples to acquire.

2. \*\*Data Acquisition:\*\*

- Acquires analog data from the ADC and converts it to voltage readings.

3. \*\*Main Loop:\*\*

- Reads analog data from the ADC.

- Applies signal processing to the acquired data.

- Detects mean, RMS, dominant frequency, and envelope of the data.

- Converts processed data to integers for Modbus transmission.

- Sends processed data to the Modbus LCD screen.

- Prints debug messages for monitoring and troubleshooting.

With the inclusion of ADC configuration and data acquisition, the code is now capable of acquiring analog data from the ADC and processing it before displaying on the Modbus LCD screen.