Table of Contents

1. **Core DSP and Analysis Functions**
   - calculate_amplitudes_and_dc
   - fir_filter
   - remove_dc_offset_temp
   - normalize_to_weights
2. **File and Output Management**
   - save_amplitude_results
3. **Visualization and GUI Functions**
   - init_plot
   - update_live_plot
4. **Main Process Orchestration**
   - process_and_plot_live_data
5. **Networking and Threading Functions**
   - receive_data_loop
   - recvall
6. **Main Execution Block**
   - if __name__ == "__main__"

1. Core DSP and Analysis Functions

calculate_amplitudes_and_dc(raw_adc_values, sampling_rate)

- **Purpose:** To perform a one-time analysis on the *entire* received signal before any real-time plotting begins. It calculates the peak-to-peak amplitude (dynamic range) of both the raw and filtered signals, and it determines the signal's stable DC offset.
- **Args:**
  - raw_adc_values (list): The complete list of raw ADC integer values from a single file.
  - sampling_rate (float): The true sampling rate of the data (e.g., 50.0 Hz).
- **Returns:**
  - A tuple containing: (raw_amplitude, filtered_amplitude, stable_dc_offset).
- **Working Process:**
  1. **Calculate Stable DC Offset:** It first converts the entire list of raw ADC values into a NumPy array of floats. It then calculates the mean (average) of this entire array. This value is the **stable DC offset** and is critical for correct filtering.
  2. **Calculate Raw Amplitude:** It converts the raw ADC values to physical weights using normalize_to_weights and then finds the difference between

the maximum and minimum weight.
3. **Calculate Filtered Amplitude:**
   - It subtracts the stable DC offset (from step 1) from the entire signal.
   - It passes this entire DC-removed signal to the fir_filter function to get a fully filtered version.
   - It adds the stable DC offset back to this filtered result to restore its correct absolute scale.
   - It converts the reconstructed filtered signal to weights and calculates its peak-to-peak amplitude.
4. It returns the two calculated amplitudes and the stable DC offset.

fir_filter(values, cut_off_frequency, sampling_rate)

- **Purpose:** To apply a low-pass Finite Impulse Response (FIR) filter to a block of data using the cmsisdsp library. This is the core signal smoothing function.
- **Args:**
  - values (np.array): A NumPy array of signal data, which should already be DC-removed.
  - cut_off_frequency (float): The frequency (in Hz) above which signals should be attenuated (e.g., 10.0 Hz).
  - sampling_rate (float): The actual sampling rate of the data.
- **Returns:**
  - A tuple containing: (filtered_values, fir_coefficients).
- **Working Process:**
  1. **Design Coefficients:** It uses scipy.signal.firwin to design a set of ideal filter coefficients based on the desired cutoff and sampling rate. SciPy is used here because it is a powerful and standard tool for filter *design*.
  2. **Prepare CMSIS-DSP Instance:** It creates the necessary data structures for the cmsisdsp library:
     - fir_instance: An object that will hold the filter's configuration.
     - state_f32: A zero-filled array that acts as the filter's "memory" of previous samples. This is essential for continuous filtering.
  3. **Initialize Filter:** It calls dsp.arm_fir_init_f32(), passing it the instance, the coefficients, and the state buffer. This prepares the CMSIS-DSP filter for processing.
  4. **Execute Filter:** It calls dsp.arm_fir_f32(), the high-performance filtering function from the CMSIS-DSP library, to process the input values. This function performs the core mathematical operations.
  5. The final filtered data array is returned.

remove_dc_offset_temp(values)

- **Purpose:** To prepare a signal for filtering by centering it around zero.
- **Working Process:**
  1. It calculates the mean (average) value of the input data array.
  2. It subtracts this mean from every sample in the array.
  3. The resulting array, which now has an average value of zero, is returned.

normalize_to_weights(values)

- **Purpose:** To convert raw, abstract integer ADC values into meaningful physical units (in this case, grams).
- **Working Process:** It applies a linear conversion formula using two pre-defined calibration constants (ZERO_CAL and SCALE_CAL) to each ADC value.

2. File and Output Management

save_amplitude_results(...)

- **Purpose:** To save the results of the one-time analysis to a permanent text file for record-keeping.
- **Working Process:**
  1. It creates an analysis_results directory if one does not already exist.
  2. It creates a new, unique filename based on the original input filename (e.g., analysis_datafile.txt).
  3. It opens this new file in write mode ('w') and writes the calculated raw and filtered amplitudes, along with a timestamp, into the file.

3. Visualization and GUI Functions

init_plot(label)

- **Purpose:** To create and configure the matplotlib plot window at the start of each new file simulation.
- **Working Process:**
  1. It creates a figure and two vertically stacked subplots.
  2. It adjusts the plot layout to make space on the left for the speed control widgets.
  3. It sets the titles, labels, and grids for both the "Raw Data" and "Filtered Data" plots.
  4. **Interactive Widget Creation:**
     - It creates a RadioButtons widget panel on the left side of the window.
     - It populates the widget with the available speed options (e.g., "1ms", "20ms").
     - It defines and attaches a callback function (on_speed_change) to the

widget. This function is automatically executed whenever the user clicks a new radio button.

5. It connects another callback function to the window's "close" event, which safely signals the entire application to shut down.

update_live_plot()

- **Purpose:** To refresh the plot window with new data during the real-time simulation loop.
- **Working Process:**
  1. It reads the most recent data points from the global current_raw_buffer and current_filtered_buffer.
  2. It updates the x and y data for the two lines on the plots.
  3. It automatically re-scales the Y-axis limits based on the visible data to ensure the signal is always framed nicely.
  4. It forces the matplotlib canvas to redraw itself, creating the animation effect.

4. Main Process Orchestration

process_and_plot_live_data(raw_adc_values, file_name)

- **Purpose:** To act as the main controller for processing a single file.
- **Working Process:**
  1. **One-Time Analysis:** It first calls calculate_amplitudes_and_dc and save_amplitude_results to perform the upfront analysis for the entire file.
  2. **Initialize Plot:** It calls init_plot to create the plot window.
  3. **Real-Time Loop:** It iterates through the raw_adc_values one by one. In each iteration, it:
     - Pauses the program for the USER_SELECTED_INTERVAL_MS (which is controlled by the radio buttons). This creates the real-time playback effect.
     - Adds the new raw ADC value to a buffer.
     - Once the buffer is large enough, it calls the fir_filter function on the most recent window of data.
     - It adds the new raw and filtered points to the plotting buffers.
     - It calls update_live_plot() to refresh the screen.

5. Networking and Threading Functions

receive_data_loop(data_queue)

- **Purpose:** To run continuously in a separate, background thread and handle all network communication, preventing the main application from freezing while waiting for data.

- **Working Process:**
    1. It establishes a TCP connection to the server.
    2. It reads and discards the initial configuration message from the server (since the user now controls the speed).
    3. It enters a loop, using the recvall helper function to reliably receive data based on the length-prefixing protocol.
    4. After successfully receiving a complete file's content, it parses the text to extract a list of ADC values.
    5. It places the list of values and the filename into the thread-safe data_queue.
    6. When the server sends the END_OF_TRANSMISSION signal or disconnects, the loop terminates, and a None value is placed in the queue to signal completion to the main thread.

recvall(sock, n)

- **Purpose:** A small utility function to ensure that exactly n bytes of data are received from a TCP socket, as TCP can sometimes send data in smaller chunks.

6. Main Execution Block

if __name__ == "__main__"

- **Purpose:** This is the entry point of the application.
- **Working Process:**
    1. It creates the thread-safe data_queue.
    2. It creates and starts the receive_data_loop on a new background thread.
    3. It enters the main application loop, where it continuously tries to get a new data file from the queue.
    4. When a file is retrieved, it calls process_and_plot_live_data to start the analysis and visualization for that file.
    5. This loop continues until it retrieves the None signal from the network thread, indicating that all files have been processed.
    6. The finally block ensures a graceful shutdown, waiting for the plot window to be closed and exiting the application cleanly.