HOMEWORK-1 REPORT

CSE 421

Efe Kayadelen
Student ID : 20220701088

Samet Alper Özdemir
Student ID : 20210702021

GITHUB REPO: https://github.com/blackwh1p/CSE421-Homeworks

Part-1:

In the first part of the homework, we used Mbed Studio and took 10 temperature readings from the STM32F746 Discovery board.

Our Code:

```
#include "mbed.h" // Include Mbed OS main library

#define WAIT_TIME_MS 200 // Define wait time between readings (200 ms)

AnalogIn intTemp(ADC_TEMP); // Create an AnalogIn object for the internal temperature sensor (ADC_TEMP)

int main()
{
    const float VREF = 3.3f; // Reference voltage used by ADC (usually 3.3V)
    const float V25 = 0.76f; // Sensor voltage at 25° C (from microcontroller datasheet)
    const float AVG_SLOPE = 0.0025f; // Average slope of sensor voltage change per ° C (2.5mV/° C)

    for (int i = 0; i = 10; i++) // Loop 10 times to take temperature readings
    {
        uint16_t raw = intTemp.read_u16(); // Read raw ADC value (0-65535)

        float vsense = (raw / 65535.0f) * VREF; // Convert raw ADC value to voltage

        float tempC = ((vsense - V25) / AVG_SLOPE) + 25.0f; // Convert voltage to temperature (° C) using formula

        printf("[%02d] raw = %5u  Temp = %.2f C¥r¥n", i + 1, (unsigned)raw, tempC); // Print reading number,
raw ADC, and temperature

        thread_sleep_for(WAIT_TIME_MS); // Wait for 200 milliseconds before next reading
    }

    printf("Done.¥r¥n"); // Print message when readings are finished

    while (true) // Infinite loop to keep the program running
    {
        thread_sleep_for(1000); // Sleep 1 second repeatedly to avoid using CPU fully
    }
}
```

Part-2:

In the second part, we chose the application "Feature Extraction from Audio Signals." Since we didn’t have a sensor or microphone to use with the STM32F746 Discovery board, we decided to use the dataset provided in the book. We downloaded the data file for this purpose. Our goal was to send the data in the “recordings” folder — which is the downloaded dataset — to the STM32F746 MCU using Python code. After that, we aimed to perform feature extraction through Mbed Studio on the STM32F746 Discovery board. Throughout the application, we didn’t get any errors in the code, but we couldn’t make it work properly. The reasons we think it didn’t work are as follows:

- We encountered a communication problem between the PC and the STM32F746 Discovery board. After running the Python code, we either couldn’t send the data or sent incorrect data.

  Python Code:

```
# host_stream_wav.py - stream WAVs from ./recordings and save exactly-what-was-sent
import os, time, struct, csv, sys # stdlib imports for filesystem, timing, binary packing, CSV I/O,
and stdout control
import numpy as np # numerical operations and array handling
from math import gcd # greatest common divisor (for rational resampling)
from scipy.io import wavfile # read/write WAV files
from scipy.signal import resample_poly # high-quality rational resampling
import serial # PySerial for UART/COM communication

# ---- make prints show up immediately ----
try:
    sys.stdout.reconfigure(line_buffering=True) # on Python 3.7+, force line-buffered prints so output
appears immediately
except Exception:
    pass # if not available (older Python/IDE), ignore and continue

def p(*a, **kw):
    kw.setdefault("flush", True) # ensure print flushes by default so logs are visible right away
    print(*a, **kw) # wrapper around print

# ---- config (paths are relative to this .py file) ----
SER_PORT   = "COM6" # change if needed (e.g., "/dev/ttyACM0")  # serial port name to open
BAUD       = 115200 # UART baud rate
IN_FOLDER  = "./recordings"  # <-- your source folder with WAVs # input directory containing .wav files
FRAME_LEN  = 1024 # samples per frame to send per packet
FS_TARGET  = 8000 # target sample rate for streaming/MFCC
OUT_FRAMES = "./sent_frames # folder to save raw frames actually sent

# ---- helpers ----
def to_int16_any_dtype(x):
    x = np.asarray(x) # ensure numpy array
    if np.issubdtype(x.dtype, np.floating): # if float -> normalize and scale to int16
        m = float(np.max(np.abs(x))) if x.size else 1.0 # peak magnitude for normalization
        if m == 0.0: m = 1.0 # avoid division by zero for silent data
        x = (x / m).astype(np.float32) # normalize to [-1,1]
        return (np.clip(x, -1.0, 1.0) * 32767.0).astype(np.int16) # scale to int16 range
    if x.dtype == np.int32: # 32-bit PCM -> downshift to 16-bit
        x = np.right_shift(x, 16)  # drop lower 16 bits
        x = np.clip(x, -32768, 32767).astype(np.int16) # clamp to int16 range
        return x
    if x.dtype == np.int16: # already int16
        return x
    x = x.astype(np.float32) # other integer types -> convert via float path
    m = float(np.max(np.abs(x))) if x.size else 1.0
    if m == 0.0: m = 1.0
    x = x / m
    return (np.clip(x, -1.0, 1.0) * 32767.0).astype(np.int16) # final int16 conversion

def load_wav_8k_mono(path, fs_target=FS_TARGET):
    fs, d = wavfile.read(path) # read WAV file -> sample rate, data array
    if d.ndim > 1: # if stereo/multi-channel
        d = d[:, 0] # take left channel only (mono)
    if fs != fs_target: # resample if sample rate differs
        g = gcd(fs, fs_target) # simplify ratio with gcd
        d = resample_poly(d.astype(np.float32), fs_target // g, fs // g)   # polyphase resampling
(num/den)
    return to_int16_any_dtype(d)  # return as int16 mono 8 kHz

def frames(x, N, step=None):
    if step is None: step = N                               # default: non-overlapping frames
    for i in range(0, len(x) - N + 1, step):                # slide window in steps
        yield i // step, x[i:i+N]                           # yield frame index and slice

def rms(x):
    x = x.astype(np.float64)                                # use float64 for numeric stability
    return float(np.sqrt(np.mean(x * x))) if x.size else 0.0     # root-mean-square amplitude
```

```
def main():
    os.makedirs(OUT_FRAMES, exist_ok=True)                      # ensure output frames dir exists
    os.makedirs(IN_FOLDER, exist_ok=True)                        # ensure input folder exists (helps
users)

    # open serial
    p(f"[HOST] Opening serial {SER_PORT} @ {BAUD} ...")          # log serial open
    ser = serial.Serial(SER_PORT, BAUD, timeout=2, write_timeout=2)  # open COM port with read/write
timeouts
    time.sleep(1.0)                                               # small delay so MCU’ s USB CDC is
ready
    p(f"[HOST] Connected: {SER_PORT} @ {BAUD}")                  # confirmation log

    # list input wavs from ./recordings
    wavs = [os.path.join(IN_FOLDER, f) for f in os.listdir(IN_FOLDER) if f.lower().endswith(".wav")]
# gather .wav paths
    wavs.sort()                                                  # deterministic ordering
    if not wavs:
        p(f"[HOST] No WAVs in {IN_FOLDER}")                      # inform user if no input
        return                                                   # exit early

    # open logs
    mfcc_csv = open("mfcc_dump.csv", "w", newline="")             # CSV to store MFCC lines echoed by
MCU
    mfcc_writer = csv.writer(mfcc_csv)                           # CSV writer
    mfcc_writer.writerow(["file", "frame"] + [f"c{i}" for i in range(13)])  # header with 13 coeffs

    idx_csv = open("sent_frames_index.csv", "w", newline="")      # CSV to index/frame stats of what
was sent
    idx_writer = csv.writer(idx_csv)                             # CSV writer
    idx_writer.writerow(["file", "frame", "nsamples", "min", "max", "mean", "rms"])  # header for
diagnostics

    for w in wavs:                                               # iterate each WAV file
        base = os.path.basename(w)                               # filename
        stem, ext = os.path.splitext(base)                       # stem and extension
        p(f"[HOST] Streaming: {base}")                           # log which file is being streamed

        # prepare data we actually send
        d16 = load_wav_8k_mono(w, FS_TARGET)                     # get 8kHz mono int16 data

        # save the exact audio stream we’ ll send (whole file) into current py folder
        sent_wav_path = f"sent_{stem}__8k16.wav"                  # name for “exactly what was sent”
WAV
        wavfile.write(sent_wav_path, FS_TARGET, d16)             # write it (helps debugging/repro)
        p(f"       wrote: {sent_wav_path}")                      # log the saved file path

        # stream in non-overlapping frames
        for fidx, fr in frames(d16, FRAME_LEN, step=FRAME_LEN):  # iterate fixed-size frames
            # log stats & save per-frame raw bytes (exact payload after header)
            idx_writer.writerow([base, fidx, len(fr), int(fr.min()), int(fr.max()), float(fr.mean()),
rms(fr)])  # record stats

            # also save each frame as raw s16le in ./sent_frames
            raw_name = os.path.join(OUT_FRAMES, f"sent_{stem}__frame{fidx:05d}.s16le")  # per-frame raw
filename
            with open(raw_name, "wb") as fh:                        # open for binary write
                fh.write(fr.tobytes())                               # dump exact int16 little-endian
samples

            # packetize and send
            pkt = b'W' + struct.pack('<H', FRAME_LEN) + fr.tobytes()  # simple protocol: 'W' + 2-byte
len + payload
            ser.write(pkt)                                            # transmit packet over serial

            # read one MFCC line back (CSV)
            line = ser.readline().decode(errors='ignore').strip()     # read MCU response line (if any)
            if line:
                # keep the line and store to CSV (truncate/pad to 13 coeffs)
                try:
                    vals = [float(x) for x in line.split(",")]        # parse comma-separated floats
                except Exception:
                    vals = []                                         # if parse fails, store NaNs
                vals = (vals + [np.nan]*13)[:13]                      # ensure exactly 13 values
                mfcc_writer.writerow([base, fidx] + vals)             # write to MFCC CSV

    mfcc_csv.close()                                                  # close MFCC CSV file
    idx_csv.close()                                                   # close index/stats CSV file
    ser.close()                                                       # close serial port

    p("[HOST] Saved:")                                                # summarize outputs
    p("  - mfcc_dump.csv              (MCU MFCC lines)")              # MFCC lines collected from MCU
    p("  - sent_frames_index.csv      (stats per frame)")             # frame diagnostics
    p("  - sent_<orig>__8k16.wav      (exact stream audio per file)")# resampled audio actually sent
    p("  - sent_frames/sent_<orig>__frameNNNNN.s16le  (raw frames sent)") # raw frame blobs
    p("[HOST] Done.")                                                 # done indicator

if _name_ == "main":                                                    # NOTE: likely a typo; left
unchanged as requested
    main()                                                            # entry point call (will not run
unless the guard matches)
```

Main Code:

```
#include "mbed.h" // Mbed OS main library
#include <cstdint> // fixed-width integer types like uint32_t
#include <cstdio> // printf, sprintf, etc.
#include <cstdlib> // malloc, free
#include "mfcc.h" // custom MFCC processing library header

// CONFIG
static const uint32_t FRAME_LEN = 1024; // frame size, must match host side exactly
static const uint32_t NUM_MELS  = 20; // number of Mel filter banks
static const uint32_t NUM_DCT   = 13; // number of MFCC output coefficients (c0-c12)
static const int      BAUD      = 115200; // UART baud rate for PC communication

// ST-LINK VCP
static UnbufferedSerial pc(USBTX, USBRX, BAUD); // create serial interface via USB (ST-Link Virtual
COM Port)

static bool read_exact(uint8_t* dst, size_t n) { // read exactly n bytes into dst
    size_t got = 0; // how many bytes have been read
    while (got < n) { // loop until all bytes are received
        ssize_t r = pc.read(dst + got, n - got); // try to read remaining bytes
        if (r <= 0) { ThisThread::sleep_for(1ms); continue; } // if none yet, wait 1ms and retry
        got += (size_t)r; // add number of bytes successfully read
    }
    return true; // return when full n bytes are received
}

int main() {
    pc.set_blocking(true); // make serial reads/writes blocking (wait until finished)
    printf("¥r¥n[STM32 MFCC] Ready. Send frames with host.¥r¥n"); // greet the host

    mfcc_instance_t S; // declare MFCC instance structure
    if (mfcc_init(&S, FRAME_LEN, NUM_MELS, NUM_DCT) != 0) { // initialize MFCC processor
        printf("MFCC init failed¥r¥n"); // print error if initialization failed
        while (true) { ThisThread::sleep_for(1s); } // halt program in 1-second sleep loop
    }
    int16_t* frame = (int16_t*)malloc(FRAME_LEN * sizeof(int16_t)); // allocate buffer for PCM input
frame
    float*   mfcc  = (float*)malloc(NUM_DCT * sizeof(float)); // allocate buffer for MFCC results
    if (!frame || !mfcc) { // check if memory allocation failed
        printf("malloc failed¥r¥n"); // error message
        while (true) { ThisThread::sleep_for(1s); } // halt program
    }

    while (true) { // infinite processing loop
        // 1) header 'W'
        uint8_t hdr = 0; // variable to store header byte
        if (!read_exact(&hdr, 1)) continue; // read 1 byte; if fail, restart loop
        if (hdr != 'W') continue; // ignore if header is not 'W'

        // 2) uint16 length (little-endian)
        uint8_t lenBytes[2]; // buffer for 2 length bytes
        if (!read_exact(lenBytes, 2)) continue; // read 2 bytes for frame length
        uint16_t n = (uint16_t)(lenBytes[0] | (lenBytes[1] << 8)); // combine bytes into 16-bit little-
endian integer

        if (n != FRAME_LEN) { // if host sent unexpected frame size
            size_t toDrain = (size_t)n * sizeof(int16_t); // number of bytes to dump/drain from serial
            uint8_t dump[64]; // temporary drain buffer
            while (toDrain) { // drain until all bytes are skipped
                size_t chunk = toDrain > sizeof(dump) ? sizeof(dump) : toDrain; // either 64 bytes or
remaining
                read_exact(dump, chunk); // read and discard
                toDrain -= chunk; // reduce remaining count
            }
            printf("Bad frame size %u (expected %u)¥r¥n", n, FRAME_LEN); // inform about wrong frame
size
            continue; // restart main loop
        }

        // 3) payload: int16 PCM
        if (!read_exact((uint8_t*)frame, FRAME_LEN * sizeof(int16_t))) { // read PCM sample frame into
buffer
            printf("Read error¥r¥n"); // error if cannot read
            continue; // skip this iteration
        }

        // 4) MFCC
        mfcc_compute(&S, frame, mfcc); // compute MFCC coefficients from PCM frame

        // 5) print CSV: c0..c12
        for (uint32_t i = 0; i < NUM_DCT; i++) { // loop through MFCC values
            printf("%s%.6f", (i ? "," : ""), mfcc[i]); // print comma-separated with 6 decimal places
        }
        printf("¥r¥n"); // end of CSV line
    }
}
```

mfcc.c code:

```
#include "mfcc.h" // MFCC public header (types/macros/CMSIS includes)
#include <string.h> // memset for state init
#include <math.h> // math: cosf, sqrtf, logf, expf, floorf

#ifndef M_PI
#define M_PI 3.14159265358979323846 // define PI if not provided by math.h
#endif

// ===== Static buffers (single instance) ===== // static, fixed-size working memory
static float    g_window[FFT_MAX]; // Hamming window coefficients
static float    g_dct[NDCT_MAX * NMEL_MAX]; // DCT matrix (nDct x nMels)
static uint32_t g_fPos[NMEL_MAX]; // per-mel start bin index
static uint32_t g_fLen[NMEL_MAX]; // per-mel span length (bins)
static uint32_t g_fOffs[NMEL_MAX]; // per-mel offset into packed weights
static float    g_pack[PACK_MAX]; // packed triangular filter weights

static float    g_melC[NMEL_MAX + 2u]; // mel centers (nMels+2) including edges

// scratch for compute (reuse each call) // temporary per-frame buffers
static float    g_frame[FFT_MAX]; // time-domain windowed frame (float)
static float    g_fftbuf[FFT_MAX]; // RFFT output buffer (CMSIS layout)
static float    g_pows[(FFT_MAX/2u) + 1u]; // power spectrum (0..N/2)
static float    g_melE[NMEL_MAX]; // log mel energies

// ===== helpers ===== // small utility functions with internal linkage
static inline float freq2mel(float f) { return 1127.0f * logf(1.0f + f / 700.0f); } // Hz→mel conversion
static inline float mel2freq(float m) { return 700.0f  * (expf(m / 1127.0f) - 1.0f); } // mel→Hz
conversion

static void make_hamming(float *w, uint32_t N) { // fill Hamming window of length N
    if (N < 2) { if (N == 1) w[0] = 1.0f; return; } // degenerate cases
    const float two_pi = 2.0f * (float)M_PI; // constant 2π
    for (uint32_t n = 0; n < N; n++) {
        w[n] = 0.54f - 0.46f * cosf(two_pi * (float)n / (float)(N - 1)); // Hamming formula
    }
}

static void make_dct(float *M, uint32_t nDct, uint32_t nMels) { // build DCT-II matrix
    const float norm = sqrtf(2.0f / (float)nMels); // orthonormal scaling (except k=0 not special-
cased)
    for (uint32_t k = 0; k < nDct; k++) { // output coefficient index
        for (uint32_t n = 0; n < nMels; n++) { // input mel index
            float s = ((float)n + 0.5f) / (float)nMels; // sample location
            M[k * nMels + n] = cosf((float)k * (float)M_PI * s) * norm; // fill matrix entry
        }
    }
}

static int build_mel_filterbank(uint32_t fftLen, uint32_t nMels,
                                uint32_t *pos, uint32_t *len, uint32_t *offs,
                                float *packed, uint32_t *packed_used)
{ // compute triangular mel filters and pack weights
    const uint32_t half   = fftLen / 2u; // N/2 bins up to Nyquist
    const float    fmin_m = freq2mel(20.0f); // lower mel at 20 Hz (skip DC)
    const float    fmax_m = freq2mel((float)SAMP_FREQ * 0.5f); // Nyquist in mel
    const float    bin_hz = (float)SAMP_FREQ / (float)fftLen; // Hz per FFT bin

    // centers on mel scale (nMels+2) // equally spaced mel points incl. edges
    const float step = (fmax_m - fmin_m) / (float)(nMels + 1u); // mel spacing
    for (uint32_t i = 0; i < nMels + 2u; i++) g_melC[i] = fmin_m + step * (float)i; // fill centers

    // first pass: spans & total // compute bin spans for each filter
    uint32_t total = 0; // total packed weights counter
    for (uint32_t m = 0; m < nMels; m++) {
        float fL = mel2freq(g_melC[m]); // left freq
        float fC = mel2freq(g_melC[m+1]); // center freq
        float fR = mel2freq(g_melC[m+2]); // right freq

        uint32_t bL = (uint32_t)floorf(fL / bin_hz); // left bin
        uint32_t bC = (uint32_t)floorf(fC / bin_hz); // center bin (floor)
        uint32_t bR = (uint32_t)floorf(fR / bin_hz); // right bin

        if (bL < 1u)  bL = 1u;          // skip DC bin // avoid k=0
        if (bR > half) bR = half;       // clamp to Nyquist // avoid out-of-range
        if (bC <= bL)  bC = bL + 1u; // enforce ordering
        if (bR <= bC)  bR = bC + 1u; // enforce ordering
        pos[m] = bL; // store start bin
        len[m] = (bR - bL + 1u); // store span length (inclusive)
        total += len[m]; // accumulate packed size
    }
    if (total > PACK_MAX) return -1; // insufficient packed buffer

    // second pass: fill packed weights // write triangular ramps into packed[]
    uint32_t cur = 0; // current write index in packed
    for (uint32_t m = 0; m < nMels; m++) {
        float fL = mel2freq(g_melC[m]); // left Hz
        float fC = mel2freq(g_melC[m+1]); // center Hz
        float fR = mel2freq(g_melC[m+2]); // right Hz

        uint32_t bL = pos[m]; // start bin
        uint32_t bR = bL + len[m] - 1u; // end bin
        uint32_t bC = (uint32_t)floorf(fC / bin_hz); // center bin again
        if (bC <= bL) bC = bL + 1u; // keep proper order
        if (bC >= bR) bC = bR - 1u; // keep proper order

        offs[m] = cur; // record offset into packed array for this mel

        for (uint32_t b = bL; b <= bC; b++) { // rising slope
            float hz = (float)b * bin_hz; // bin frequency
            float w  = (hz - fL) / (fC - fL); // 0→1 ramp between L and C
            if (w < 0.0f) w = 0.0f; if (w > 1.0f) w = 1.0f; // clamp numeric error
            packed[cur++] = w; // store weight
        }
        for (uint32_t b = bC + 1u; b <= bR; b++) { // falling slope
            float hz = (float)b * bin_hz; // bin frequency
            float w  = (fR - hz) / (fR - fC); // 1→0 ramp between C and R
            if (w < 0.0f) w = 0.0f; if (w > 1.0f) w = 1.0f; // clamp
            packed[cur++] = w; // store weight
        }
    }
    *packed_used = cur; // return how many weights were written
    return 0; // success
}

// ===== API ===== // externally visible functions
int mfcc_init(mfcc_instance_t *S, uint32_t fftLen, uint32_t nMels, uint32_t nDct) {
    if (!S) return -1; // null pointer guard
    if ((fftLen & (fftLen - 1u)) != 0u) return -2; // must be power of 2
    if (fftLen > FFT_MAX || nMels > NMEL_MAX || nDct > NDCT_MAX) return -3; // bounds check

    memset(S, 0, sizeof(*S)); // clear struct to known state
    S->fftLen       = fftLen; // store FFT length
    S->nbMelFilters = nMels; // store number of mel filters
    S->nbDctOutputs = nDct; // store number of DCT outputs

    // map pointers to static buffers // avoid dynamic allocation on MCU
    S->windowCoefs  = g_window; // use global window
    S->dctMatrix    = g_dct; // use global DCT matrix
    S->filterPos    = g_fPos; // mel start bins
    S->filterLen    = g_fLen; // mel lengths
    S->filterOffset = g_fOffs; // mel offsets into packed weights
    S->packedFilters= g_pack; // packed weights buffer

    // fill window and DCT // precompute invariants
    make_hamming(S->windowCoefs, fftLen); // Hamming window
    make_dct(S->dctMatrix, nDct, nMels); // DCT matrix

    // build mel filterbank // compute bin ranges and weights
    uint32_t used = 0; // how many packed weights
    if (build_mel_filterbank(fftLen, nMels, S->filterPos, S->filterLen,
                             S->filterOffset, S->packedFilters, &used) != 0)
        return -4; // filter bank failed (insufficient buffer)
    (void)used; // not needed later // suppress unused warning
    // CMSIS RFFT init // configure real FFT instance
    if (arm_rfft_fast_init_f32(&S->rfft, fftLen) != ARM_MATH_SUCCESS) return -5; // init error

    return 0; // success
}

void mfcc_free(mfcc_instance_t *S) {
    (void)S; // nothing to free (static buffers) // API provided for symmetry
}

void mfcc_compute(const mfcc_instance_t *S, const int16_t *audio, float *mfcc_out) {
    const uint32_t N  = S->fftLen; // frame length
    const uint32_t NH = N / 2u; // half-length (Nyquist bin index)

    // int16 → float + window // convert PCM to float and apply window
    for (uint32_t i = 0; i < N; i++) {
        g_frame[i] = ((float)audio[i] / 32768.0f) * S->windowCoefs[i]; // scale to [-1,1], window
    }

    // RFFT // compute FFT of real input frame
    arm_rfft_fast_f32((arm_rfft_fast_instance_f32*)&S->rfft,  g_frame,  g_fftbuf,  0);  //  forward
transform

    // power spectrum (rfft layout: re0, reN/2, re1, im1, ...) // CMSIS packed format
    g_pows[0]  = g_fftbuf[0] * g_fftbuf[0]; // DC bin power (pure real)
    g_pows[NH] = g_fftbuf[1] * g_fftbuf[1]; // Nyquist bin power (pure real)
    for (uint32_t k = 1; k < NH; k++) {
        float re = g_fftbuf[2u*k + 0u]; // real part of bin k
        float im = g_fftbuf[2u*k + 1u]; // imag part of bin k
        g_pows[k] = re*re + im*im; // magnitude^2 (power)
    }

    // mel energies (log of weighted magnitude) // triangular filters over |X[k]|
    for (uint32_t m = 0; m < S->nbMelFilters; m++) {
        uint32_t start = S->filterPos[m]; // first bin for this mel
        uint32_t L     = S->filterLen[m]; // number of bins
        uint32_t off   = S->filterOffset[m]; // offset in packed weights

        float acc = 0.0f; // accumulator
        for (uint32_t j = 0; j < L; j++) {
            float mag = sqrtf(g_pows[start + j]); // magnitude from power
            acc += mag * S->packedFilters[off + j]; // apply triangular weight
        }
        if (acc < 1e-12f) acc = 1e-12f; // floor to avoid log(0)
        g_melE[m] = logf(acc); // natural log energy
    }

    // DCT → MFCC // project log-mel vector via DCT to cepstral domain
    for (uint32_t i = 0; i < S->nbDctOutputs; i++) {
        const float *row = &S->dctMatrix[i * S->nbMelFilters]; // DCT row i
        float sum = 0.0f; // dot product accumulator
        for (uint32_t m = 0; m < S->nbMelFilters; m++) sum += row[m] * g_melE[m]; // dot(row, melE)
        mfcc_out[i] = sum; // store coefficient i
    }
}
```

mfcc.h code:

```
#include "mbed.h" // Include Mbed OS main library
#include <cstdint> // Fixed-size integer types (uint32_t, etc.)
#include <cstdio> // Standard I/O for printf()
#include <cstdlib> // For malloc(), free()
#include "mfcc.h" // MFCC function and structure definitions
```

```
// CONFIG
static const uint32_t FRAME_LEN = 1024;   // Must match host frame size
static const uint32_t NUM_MELS  = 20; // Number of Mel filter banks
static const uint32_t NUM_DCT   = 13; // Number of MFCC coefficients to output
static const int      BAUD      = 115200; // UART baud rate

// ST-LINK VCP
static UnbufferedSerial pc(USBTX, USBRX, BAUD); // Serial object using ST-LINK’s virtual COM port

static bool read_exact(uint8_t* dst, size_t n) { // Read exactly n bytes from serial
    size_t got = 0; // Track how many bytes received
    while (got < n) { // Loop until all n bytes are read
        ssize_t r = pc.read(dst + got, n - got); // Read remaining bytes
        if (r <= 0) { ThisThread::sleep_for(1ms); continue; } // If no data yet, wait and retry
        got += (size_t)r; // Add received byte count
    }
    return true; // Always returns true once n bytes are read
}

int main() {
    pc.set_blocking(true); // Make serial reads/writes blocking
    printf("¥r¥n[STM32 MFCC] Ready. Send frames with host.¥r¥n"); // Startup message

    mfcc_instance_t S; // Declare MFCC configuration object
    if (mfcc_init(&S, FRAME_LEN, NUM_MELS, NUM_DCT) != 0) { // Initialize MFCC engine
        printf("MFCC init failed¥r¥n"); // Error if initialization fails
        while (true) { ThisThread::sleep_for(1s); } // Stop program forever
    }

    int16_t* frame = (int16_t*)malloc(FRAME_LEN * sizeof(int16_t)); // Buffer to hold audio frame
(signed 16-bit)
    float*   mfcc  = (float*)malloc(NUM_DCT * sizeof(float)); // Buffer to store MFCC output
    if (!frame || !mfcc) { // Check if memory allocation failed
        printf("malloc failed¥r¥n");
        while (true) { ThisThread::sleep_for(1s); } // Halt program if failed
    }

    while (true) { // Infinite loop to process incoming audio frames
        // 1) header 'W'
        uint8_t hdr = 0; // Variable to store header byte
        if (!read_exact(&hdr, 1)) continue; // Read 1 byte from serial
        if (hdr != 'W') continue; // If header isn't 'W', ignore and restart loop

        // 2) uint16 length (little-endian)
        uint8_t lenBytes[2]; // Buffer to hold length bytes
        if (!read_exact(lenBytes, 2)) continue; // Read 2 bytes from serial
        uint16_t n = (uint16_t)(lenBytes[0] | (lenBytes[1] << 8)); // Combine into 16-bit number

        if (n != FRAME_LEN) { // If received frame size doesn't match expected size
            // drain unexpected payload
            size_t toDrain = (size_t)n * sizeof(int16_t); // Number of bytes to skip
            uint8_t dump[64]; // Temporary buffer for dumping data
            while (toDrain) { // Loop until all extra bytes are discarded
                size_t chunk = toDrain > sizeof(dump) ? sizeof(dump) : toDrain; // Read in chunks of
64 bytes max
                read_exact(dump, chunk); // Read and discard
                toDrain -= chunk; // Decrease remaining bytes
            }
            printf("Bad frame size %u (expected %u)¥r¥n", n, FRAME_LEN); // Report mismatch
            continue; // Skip to next frame
        }

        // 3) payload: int16 PCM
        if (!read_exact((uint8_t*)frame, FRAME_LEN * sizeof(int16_t))) { // Read actual PCM audio data
            printf("Read error¥r¥n"); // Error if failed to read
            continue; // Skip to next
        }
        // 4) MFCC
        mfcc_compute(&S, frame, mfcc); // Compute MFCC on received frame

        // 5) print CSV: c0..c12
        for (uint32_t i = 0; i < NUM_DCT; i++) { // Loop through MFCC coefficients
            printf("%s%.6f", (i ? "," : ""), mfcc[i]); // Print as CSV format
        }
        printf("¥r¥n"); // New line after each MFCC frame
    }
}
```