```cpp
//
// Created by daran on 1/12/2017 to be used in ECE420 Sp17 for the first time.
// Modified by dwang49 on 1/1/2018 to adapt to Android 7.0 and Shield Tablet updates.
//

#include "ece420_main.h"
#include "ece420_lib.h"
#include "kiss_fft/kiss_fft.h"
#include <cmath>
#include <vector>

// JNI Function
extern "C" {
JNIEXPORT float JNICALL
Java_com_ece420_lab4_MainActivity_getFreqUpdate(JNIEnv *env, jclass);
}

// Student Variables
#define F_S 48000 // sampling rate
#define FRAME_SIZE 1024
#define VOICED_THRESHOLD 2000000000  // Find your own threshold
float lastFreqDetected = -1;

// initalize kiss_fft parameters
#define  NUM_PEAKS 12
kiss_fft_cfg kcfg_fft; // configuration for conducting fft
kiss_fft_cfg kcfg_ifft; // configuration for conducting ifft
kiss_fft_cpx fin[FRAME_SIZE]; // input with samples for fft
kiss_fft_cpx fft_out[FRAME_SIZE]; // output of fft and updated to be an input for ifft
kiss_fft_cpx ifft_out[FRAME_SIZE]; // ifft output
bool kiss_initialized = false; // must initialize cfg variables
int peak_idx[NUM_PEAKS]; // number of peaks to be
```

```
returned by multiple_peak_detection
float peak_threshold = 0.5; // treshold for detecting peaks
float f_ifft[FRAME_SIZE]; // real valued float from output of ifft

/* function declarations */
bool isVoiced(const float* buffer, int len);
int multiple_peak_detection(const float* buffIn, int * buffOut, int buffIn_len, int num_peaks, float threshold);

void ece420ProcessFrame(sample_buf *dataBuf) {
    // Keep in mind, we only have 20ms to process each buffer!
    struct timeval start;
    struct timeval end;
    gettimeofday(&start, NULL);

    // Data is encoded in signed PCM-16, little-endian, mono
    float bufferIn[FRAME_SIZE];
    for (int i = 0; i < FRAME_SIZE; i++) {
        int16_t val = ((uint16_t) dataBuf->buf_[2 * i]) | (((uint16_t) dataBuf->buf_[2 * i + 1]) << 8);
        bufferIn[i] = (float) val;
    }

    // ********************** PITCH DETECTION ************************ //
    // In this section, you will be computing the autocorrelation of bufferIn
    // and picking the delay corresponding to the best match. Naively computing the
    // autocorrelation in the time domain is an O(N^2) operation and will not fit
    // in your timing window.
    //
    // First, you will have to detect whether or not a signal is voiced.
    // We will implement a simple voiced/unvoiced
```

```
detector by thresholding
    // the power of the signal.
    //
    // Next, you will have to compute autocorrelation in its O(N logN) form.
    // Autocorrelation using the frequency domain is given as:
    //
    //  autoc = ifft(fft(x) * conj(fft(x)))
    //
    // where the fft multiplication is element-wise.
    //
    // You will then have to find the index corresponding to the maximum
    // of the autocorrelation. Consider that the signal is a maximum at idx = 0,
    // where there is zero delay and the signal matches perfectly.
    //
    // Finally, write the variable "lastFreqDetected" on completion. If voiced,
    // write your determined frequency. If unvoiced, write -1.
    // ********************* START YOUR CODE HERE *********************** //

    /* initialize kiss cfg variables */
    if (!kiss_initialized) {
        kcfg_fft = kiss_fft_alloc(FRAME_SIZE,0, nullptr, nullptr);
        kcfg_ifft = kiss_fft_alloc(FRAME_SIZE,1, nullptr, nullptr);
        kiss_initialized = true;
    }

    /* find if voice was detected */
    bool voiced = isVoiced(bufferIn, FRAME_SIZE);

    /* only want to do additional computations if the frame is voiced */
    if (voiced) {

        /* fill in fft in */
        for (int i = 0; i < FRAME_SIZE; i++) {
            fin[i].r = bufferIn[i];
            fin[i].i = 0.0;
        }

        /* compute fft */
        kiss_fft(kcfg_fft, fin, fft_out);

        /* perform conjugate multiplication */
        for (int i = 0; i < FRAME_SIZE; i++) {
            fft_out[i].r = (fft_out[i].r * fft_out[i].r) + (fft_out[i].i * -fft_out[i].i);
            fft_out[i].i = 0.0;
        }

        /* perform ifft */
        kiss_fft(kcfg_ifft, fft_out, ifft_out);

        /* store only the real parts of the ifft_out */
        for (int i = 0; i < FRAME_SIZE; i++)
            f_ifft[i] = ifft_out[i].r;

        /* find peaks */
        int num_peaks_detected = multiple_peak_detection(f_ifft, peak_idx, FRAME_SIZE, NUM_PEAKS, peak_threshold);

        /* find maximum peak and use that to find l */
        int l = 0;
        float max_peak_val = -1;
        float curr_val;
        /* skip the first peak since it leads to bad estimations */
        for (int i = 1; i < num_peaks_detected; i++) {
            curr_val = f_ifft[peak_idx[i]];
            /* check if current peak is the maximum
```

```
from the list of peaks */
            if (curr_val > max_peak_val) {
                l = peak_idx[i];
                max_peak_val = curr_val;
            }
        }

        /* update fundamental frequency */
        if (l != 0)
            lastFreqDetected = F_S / l;
        else
            lastFreqDetected = -1;
    }
    else
        lastFreqDetected = -1;

    // ********************* END YOUR CODE HERE ************************* //
    gettimeofday(&end, NULL);
    LOGD("Time delay: %ld us",  ((end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 + start.tv_usec)));
}

/*
 * Description: Determines if a frame of samples contains a voice
 * Inputs:
 *      const float* buffer -- buffer of samples
 *      int len -- length of the buffer
 * Outputs:
 *      None
 * Returns:
 *      bool voiced -- if the frame contains a voice
 * Effects:
 *      None
 */
bool isVoiced(const float* buffer, int len) {
    bool voiced = false;

    int sum = 0;
    for (int i = 0; i < len; i++) {
        sum += std::fabs(buffer[i]) * std::fabs(buffer[i]);
    }

    if (sum > VOICED_THRESHOLD)
        voiced = true;

    return voiced;
}

/*
 * Description: Returns a list of indices in buffIn that correspond to local maximums based on a
 * threshold
 * Inputs:
 *      const float* buffIn -- array of sample inputs
 *      int buffInlen -- length of buffIn sample inputs
 *      int num_peaks -- number of peaks to return in buffOut
 *      float threshold -- threshold that peaks must cross
 * Outputs:
 *      int* buffOut -- array to write indices of local maximums into
 * Returns:
 *      int buffOut_size -- number of peaks returned, capped at num_peaks
 * Effects:
 *      uses std::vector which is inefficient and may be too slow for the thread
 */
int multiple_peak_detection(const float* buffIn, int* buffOut, int buffIn_len, int num_peaks, float threshold) {
    std::vector<int> thresh_indices; // indices whose values meet the threshold
    /* find indices of buff in that meet the threshold */
    for (int i = 0; i < buffIn_len; i++) {
        if (buffIn[i] > threshold)
            thresh_indices.push_back(i);
    }

    /* boundaries to look for local maximums */
    int curr_start = thresh_indices[0];
    int curr_end = -1;

    /* number of peaks found thus far */
    int buffOut_size = 0;

    int idx; // current index from thresh_indices
    for (int i = 1; i < thresh_indices.size(); i++) {
        /* break if already found desired number of peaks */
        if (buffOut_size == num_peaks)
            break;

        /* get the current index from threshold indices */
        idx = thresh_indices[i];

        /* update right most index to look for in a slice */
        if ((curr_end == -1) || (idx - 1 == curr_end)) {
            curr_end = idx;
            continue;
        }
        /* find max value from previous slice if discontinuity is found,
         * and initialize start and end idx for next slice
         */
        else if ((curr_end > -1) && (idx - 1 != curr_end)) {
            /* get idx of peak and place it into user buffer */
            buffOut[buffOut_size] = findMaxArrayIdx(
buffIn, curr_start, curr_end);
            ++buffOut_size;
            /* update indexes for next slice */
            curr_start = idx;
            curr_end = -1;
        }
    }

    return buffOut_size;
}

JNIEXPORT float JNICALL
Java_com_ece420_lab4_MainActivity_getFreqUpdate(
JNIEnv *env, jclass) {
    return lastFreqDetected;
}
```