計算機科学実験及演習 4 音響信号処理 レポート

工学部 情報学科 計算機科学コース
学生番号: 1029358455 氏名: 登古紘平

2026 年 1 月 15 日

課題内容

音響信号ファイルを読み込み，音響信号を操作するグラフィカルユーザーインタフェースを作成せよ．少なくとも以下の操作を可能にすること．

1. ボイスチェンジ
2. トレモロ

加えて、このインタフェースがより便利なものになるように改良せよ。

1　作成したプログラムの説明

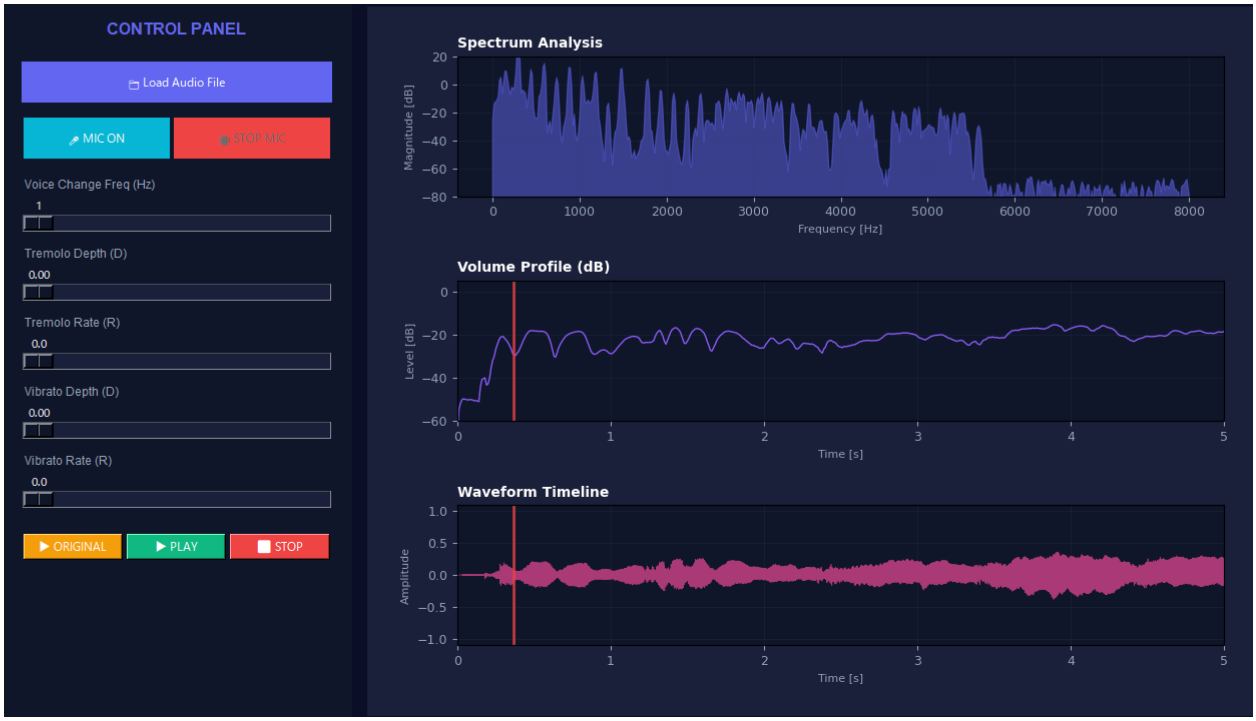

図 1　実際の GUI の画面

図 1 に、今回作成した GUI の画面を示す。これは、読み込んだ音声ファイルに対し、GUI 上で調節したパラメータを用いて、ボイスチェンジ、トレモロ、ビブラートの計算を行い、その音声を出力する。それに伴い、スペクトラム、音量、波形をプロットする。PLAY ボタンを押すと、音声が再生されるとともにアニメーションが動き、再生部分を可視化する。STOP ボタンを押すと停止し、ORIGINAL ボタンを押すと、元音声が再生される。プロットされるスペクトラムは再生部分のものである。画面左側の、パラメータを操作するスライドバーを動かすと、それに対応して波形が再プロットされる。図 2、3 に、再計算された波形がプロットされた GUI を示す。

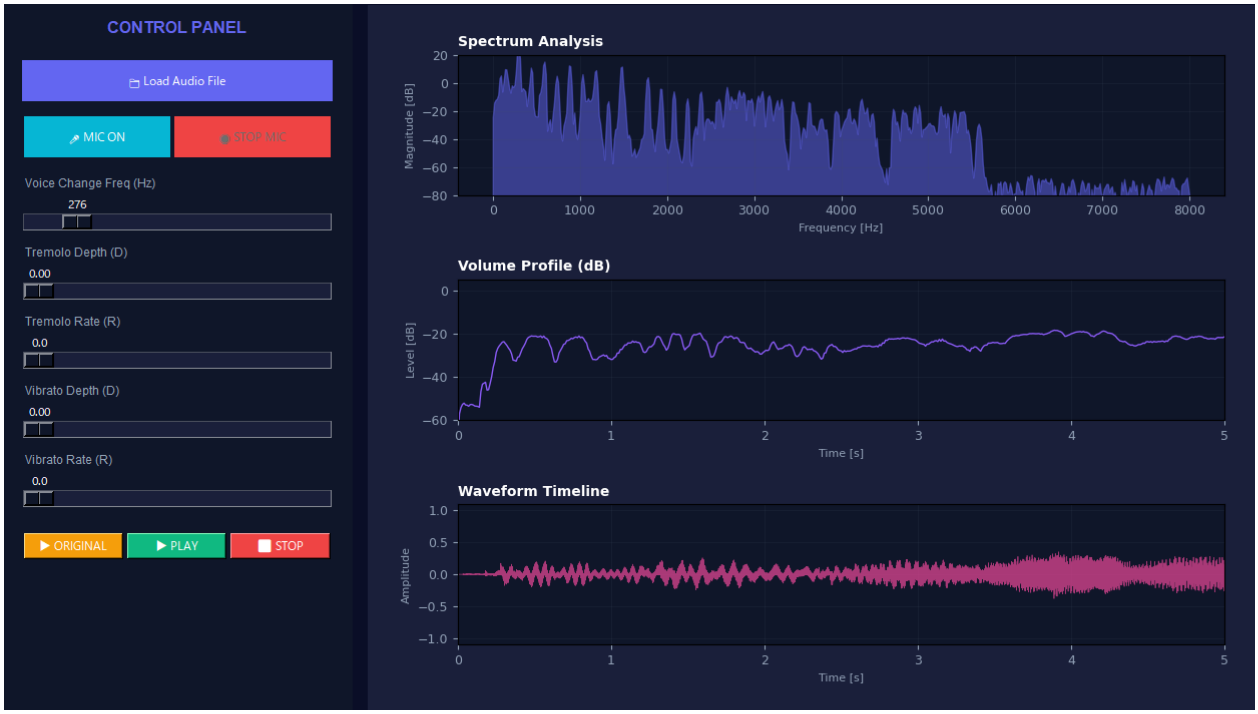

図 2　ボイスチェンジを作用させた GUI

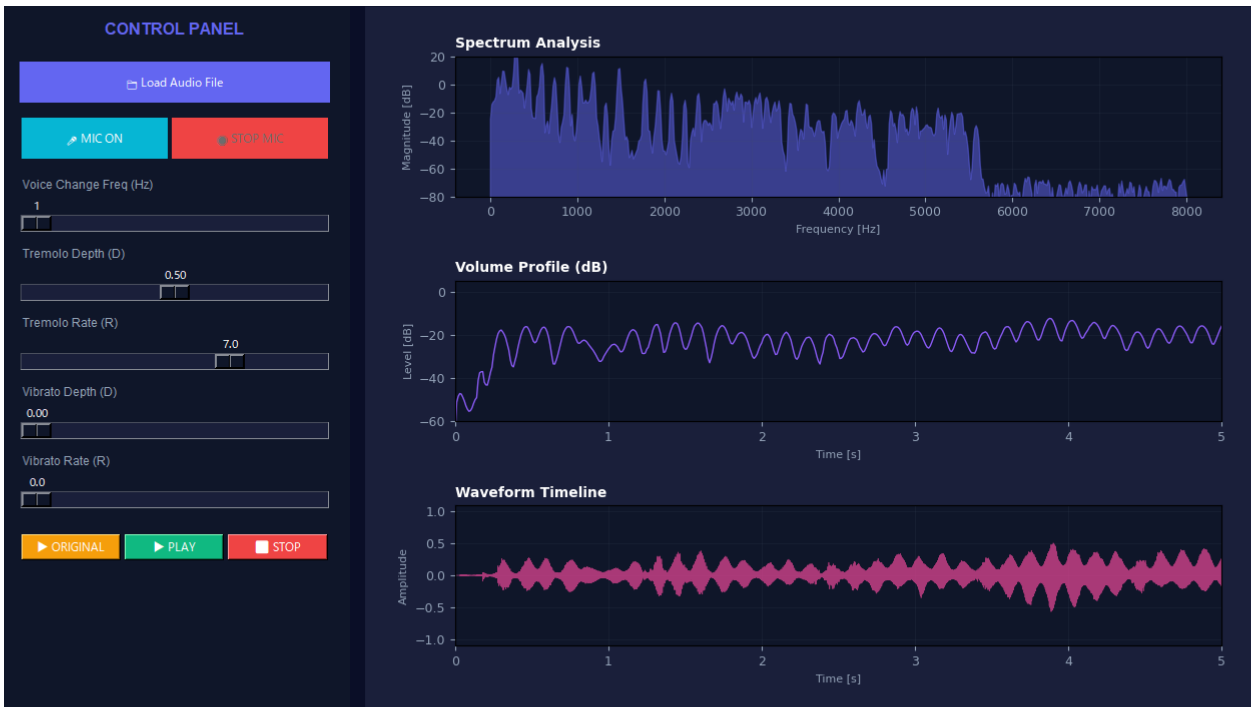

図 3　トレモロを作用させた GUI

また、これらの基本的な音響信号操作の機能に加え、私の GUI の特徴的な機能として、入力する音声をリ

アルタイムで変換し出力する機能を実装した。GUI の画面左上に、 ”MIC ON” 、 ”STOP MIC” ボタンがある。 ”MIC ON” ボタンを押すと、入力受付状態になり、マイクに向かって発声すると、スライドバーのパラメータにより調節された音声が出力として返る。作用できる変更は、ボイスチェンジ、トレモロである。これにより、自身の声を変換して楽しむことができる。以下に、ソースコードの抜粋とその説明を示す。

- 正弦波生成関数 (ボイスチェンジ用)

```
def generate_sinusoid(sampling_rate, frequency, duration):
    t = np.arange(int(sampling_rate * duration)) / sampling_rate
    return np.sin(2.0 * np.pi * frequency * t)
```

sampling_rate はサンプリングレート、frequency は生成する正弦波の周波数、duration は生成する正弦波の時間的長さである。

- トレモロ

```
def tremolo(input_signal, fs, D, R):
    if D <= 0: return input_signal
    t = np.arange(len(input_signal))
    tremolo_envelope = 1.0 + D * np.sin(2.0 * np.pi * R * t / fs)
    return input_signal * tremolo_envelope
```

スライドバーで D=0 としたときに、確実にトレモロがかからないように、深度が 0 なら何もしないように設定している。

- ビブラート

```
def vibrato(input_signal, fs, D, R):
    if D <= 0 or R <= 0: return input_signal
    n = len(input_signal)
    t = np.arange(n)
    delay_samples = D * 100
    tau = delay_samples * np.sin(2.0 * np.pi * R * t / fs)
    indices = t - tau
    indices = np.clip(indices, 0, n - 1)
    return np.interp(indices, t, input_signal)
```

遅延の深さをサンプル数に変換 (係数 100 は調整値) し、配列の範囲外に出ないように制限している。また、delay_samples が連続的に変化するため、参照すべきインデックスが整数値にならない。デジタル信号の離散的な値を線形補間することで、ピッチの滑らかな揺れを再現している。

- dB

```
def calculate_db_profile(signal, sr, size_frame=512, size_shift=160):
```

```
    db_list = []
    time_list = []
    for i in np.arange(0, len(signal) - size_frame, size_shift):
        idx = int(i)
        x_frame = signal[idx:idx + size_frame]
        current_rms = np.sqrt(np.mean(x_frame ** 2))
        current_db = 20 * np.log10(current_rms + 1e-12)
        db_list.append(current_db)
        time_list.append(idx / sr)
    return np.array(time_list), np.array(db_list)
```

e-12 は log(0) のエラー回避用の微小値である。音圧レベルとして表現するため、RMS（実効値）の常用対数に 20 を乗じている。

- 現在のスライダー値を取得してエフェクトを適用する

```
    def apply_effects(self):
        if self.orig_signal is None: return
        freq = self.slider_voice_change_freq.get()
        t_d = self.slider_tremolo_depth.get()
        t_r = self.slider_tremolo_rate.get()
        v_d = self.slider_vibrato_depth.get()
        v_r = self.slider_vibrato_rate.get()

        # ボイスチェンジ
        if freq > 1:
            sin_wave = generate_sinusoid(self.sr, freq, len(self.orig_signal)/self.sr)
            x_vc = self.orig_signal * sin_wave
        else:
            x_vc = self.orig_signal.copy()

        x_trem = tremolo(x_vc, self.sr, t_d, t_r)
        self.processed_signal = vibrato(x_trem, self.sr, v_d, v_r)
        self.times_db, self.dbs = calculate_db_profile(self.processed_signal, self.sr)
```

この関数により、スライドバーから取得した値を用いて実際に計算を行う。計算順序は、ボイスチェンジ、トレモロ、ビブラートの順である。同時に、dB も計算する。

2 問題点と考察

- リアルタイムでの解析
  今回、選択した音声ファイルを解析するにあたって、再生前にパラメータを調整し、それをもとに編集して波形と音声を出力している。理想として、再生中にパラメータをサイドバー上で調節した瞬間に、それが計算に反映されて、波形のプロットと音声が変更されるよう実装しようと考えたが、動作が重くなりすぎてしまい、とても GUI として成り立つものではなかったためこのような実装となった。このことから、同時に行うことはできないが、音声の再生と波形のプロットを切り替えるようさらに GUI 上で操作を行えるようにし、それぞれ別に計算を行うことによりこの問題は改善できるのではないかと考えられる。

- マイク入力におけるビブラート作用
  マイクより入力においては、波形のプロットは行わず、編集された音声のみを出力しているため、リアルタイムで音声を加工することに成功した。ボイスチェンジ、トレモロにおいては問題なかったが、ビブラートの実装は断念した。ビブラートは、遅延によって実装されるため、コールバック関数を用いて実装を行う場合、前のブロックの音を保持して用いなければならない。今回のプログラムにおけるコールバック処理は、固定されたブロックサイズごとに独立して計算を行う仕様となっており、ブロックの境界において位相の連続性を維持することが極めて困難であった。また、以下に示すのは、LLM を用いたリファクタリング結果である。

  目的: リアルタイムのビブラート処理に対する知見をより深めるため。
  使用したモデル: Gemini3
  使用したプロンプト:
  リアルタイムでマイク入力からビブラート処理を行うにはどのような処理が必要？

  LLM からの出力:

  1. リングバッファ（循環バッファ）の構築
     リアルタイム処理では、マイクから入ってきた最新の音を常に保存し続け、古い音を捨てていく専用のメモリ領域（リングバッファ）が必要です。ビブラートは「音を遅らせて再生する」ことで実現するため、最大遅延時間（例：20ms 分）以上のバッファを常に確保しておきます。

  2. 書き込みポインタと読み出しポインタの分離
     バッファに対して、以下の 2 つのポインタを動かします。
     書き込みポインタ (Write Pointer): マイクから入ってきた最新の音を、バッファの現在の位置に書き込み、1 つ進めます。
     読み出しポインタ (Read Pointer): 書き込みポインタよりも「少し後ろ（過去）」の位置から音を読み出します。

3. LFO による読み出し位置の動的変化
   ここがビブラートの核心です。読み出しポインタの位置を固定せず、**LFO（低周波発振器）**を使って前後に揺らします。読み出し位置を書き込み位置に近づける（遅延を短くする）と、ピッチが上がります。読み出し位置を書き込み位置から遠ざける（遅延を長くする）と、ピッチが下がります。この「遅延時間の伸縮」がドップラー効果と同じ原理で音程の揺れ（ビブラート）を生みます。

4. 分数インデックスと線形補間
   LFO で読み出し位置を計算すると、「10.5 サンプル前」のように整数ではない位置を指すことがほとんどです。デジタルの波形は整数位置にしかデータがないため、前後のサンプルの値から計算する**「線形補間（Linear Interpolation）」**が必要です。

   $y(t) = (1 - f) \cdot x[n] + f \cdot x[n+1]$（ここで f はインデックスの小数部分）

   これを行わないと、読み出し位置が切り替わるたびに「プチプチ」という段差ノイズが発生します。
   実装のイメージ（疑似コード）
   もし Python で実装を拡張するなら、以下のようなロジックをクラス化して保持することになります。

```
class RealTimeVibrato:

    def __init__(self, buffer_size=4000):
        self.buffer = np.zeros(buffer_size)
        self.write_ptr = 0
        self.phase = 0.0

    def process(self, chunk, rate, depth, sr):
        out = np.zeros_like(chunk)
        for i in range(len(chunk)):
            # 現在の音をバッファに保存
            self.buffer[self.write_ptr] = chunk[i]

            # LFO で遅延量を計算（例: 10ms ± 5ms）
            delay = (depth * 100) * (1 + np.sin(2 * np.pi * self.phase))
            read_ptr = (self.write_ptr - delay) % len(self.buffer)

            # 線形補間して読み出し
            idx_low = int(np.floor(read_ptr))
            idx_high = (idx_low + 1) % len(self.buffer)
            frac = read_ptr - idx_low
```

```
            out[i] = (1 - frac) * self.buffer[idx_low] + frac * self.buffer[idx_high]

            # ポインタと位相の更新
            self.write_ptr = (self.write_ptr + 1) % len(self.buffer)
            self.phase = (self.phase + rate / sr) % 1.0
        return out
```

まとめ：なぜ難易度が高いのか

ブロックの跨ぎ: 前のブロックの最後の方のデータを、次のブロックの冒頭で「過去の音」として参照し続ける必要がある。
補間処理: 単純な配列インデックス参照ではなく、常に計算（補間）が必要。
レイテンシ: 遅延を作り出す処理なので、設定しすぎると自分の声が遅れて聞こえてしまう。
このように、ビブラートは「時間軸をゴムのように伸ばし縮みさせる」処理であるため、振幅を変えるだけのトレモロよりも一段階上の実装技術が必要になります。

- ボイスチェンジ、トレモロのリアルタイム処理
  音声ファイルに対するトレモロ、ボイスチェンジ処理は、これまでの演習で実装していたプログラムを流用した。これは、処理を行う音声を最初に全部受け取っているが、マイク入力における処理はリアルタイムで行われるため、新たに処理を記述する必要があった。以下に、プログラムの抜粋を示す。

```
    # マイク入力時のリアルタイム処理コールバック
    def mic_audio_callback(self, indata, outdata, frames, time, status):
        x = indata[:, 0].copy() # 入力データの取得
        t_array = np.arange(frames) / self.sr
        freq = self.slider_voice_change_freq.get()
        # ボイスチェンジ処理
        if freq > 1.0:
            # 連続したサイン波を作るために、前回の位相 (phase_r) を引き継ぐ
            carrier = np.sin(2.0 * np.pi * (self.phase_r + freq * t_array))
            x *= carrier
            self.phase_r = (self.phase_r + freq * frames / self.sr) % 1.0 # 位 相
更新

        t_depth = self.slider_tremolo_depth.get()
        t_rate = self.slider_tremolo_rate.get()
        if t_depth > 0:
            # 位相 (phase_t) を使って連続性を保つ
            trem_env = 1.0 + t_depth * np.sin(2.0 * np.pi *
            (self.phase_t + t_rate * t_array))
```

```
        x *= trem_env
        self.phase_t = (self.phase_t + t_rate * frames / self.sr) % 1.0

    # 出力バッファに書き込む＝スピーカーから音が出る
    outdata[:, 0] = x
    if outdata.shape[1] > 1: outdata[:, 1] = x
```

上記のソースコードにおいて、リアルタイム処理を実現するための鍵となるのは、self.phase_r 、self.phase_t を用いた位相の保持と更新である。マイク入力によるリアルタイム処理では、音声をブロック単位（本プログラムでは 1024 サンプル）で分割して処理を行う。各ブロックの処理ごとに時間配列 t_array を 0 から生成し直すと、ブロックの継ぎ目で変調用のサイン波が 0 にリセットされてしまい、これが波形の不連続によるノイズの原因となる。この問題を解決するため、本プログラムでは各ブロックの処理終了時の位相状態を保持し、次回のコールバック呼び出し時にその値を初期位相として引き継ぐ設計とした。ボイスチェンジ処理、トレモロ処理のいずれにおいても、この手法を用いることで、滑らかなエフェクト適用を可能にした。