ラズベリーパイを用いた電子オルゴール

Music box using Raspberry Pi

坂下 達哉

Tatsuya Sakashita

玉川大学工学部情報通信工学科，194–8610 東京都町田市玉川学園 6–1–1
Department of Information & Communication Technology, College of Engineering, Tamagawa University,
6–1–1 Tamagawa-gakuen, Machida-shi, Tokyo 194–8610

Abstract

The Department of Information & Communication Technology has several courses of IoT. In these courses, students learn to control electronic devices by Raspberry Pi in Python. One of these courses is "Intelligent device experiment I" in third year. In the final lecture of this course, firstly, they make music box to play speakers by using hardware PWM. Secondly, they learn how to generate wav files. In both cases, the goal is to play "For Elise." Through this lecture, students can be motivated to learn a lot of programming techniques of Python. This lecture is an attempt of STREAM education.

Keywords: Raspberry Pi, IoT education, STREAM education, music box, wav file

1 はじめに

情報通信工学科では，以下のような IoT の講義と実験が開講されている．

- ２年次後期のインテリジェントデバイス入門では，Python 文法を学んだ後，LED，タクトスイッチ，フォトレジスタ，A/D 変換器をラズベリーパイから制御する基礎を学修する．
- ３年次前期のインテリジェントデバイス実験 I の後半 6 回が，ラズベリーパイの実習に当てられている [1,2].

ラズベリーパイ [3,4] とは，手のひらサイズのコンピュータである．Ubuntu Linux 等の OS をインストールして使用できるため，以下の長所を持つ．

- Linux で用いられているコマンド，プログラム言語およびライブラリが全て使用できる．
- 上記の言語のうち，Python が広く用いられる．Python では，豊富なパッケージが利用できるため，複雑なプログラムを短く書ける．
- IoT 用のパッケージも，標準の Python や C++ 等に準拠したライブラリとして公開されている．そのため，IoT 向けの特殊な言語を覚える必要がなく，汎用的かつ実用的なプログラミング技法が学べる．

今年度のインテリジェントデバイス実験 I の最終回では，「ラズベリーパイを用いた電子オルゴール」をテーマとした．これは，科学（波動の物理），技術（プログラミング・データ処理・可視化処理），ロボティクス（マイコン制御），工学（電子回路・信号処理），芸術（音楽），数学（物理・工業数学）の異分野融合を目指したテーマであり，玉川大学が推進する STREAM 教育の一環として位置付けられる．本稿では，このテーマについて報告を行う．

本稿の構成は，以下の通りである．2 節では，実験の概要（実験 1 および 2）を述べる．3 節および 4 節において，それぞれ実験 1 および実験 2 について詳述する．5 節でまとめを行う．

2 実験の概要

2.1 ラズベリーパイの環境

使用したラズベリーパイの環境は，以下の通りである．

- Raspberry Pi 3 Model B V1.2
- OS: Ubuntu 18.04.2 LTS
- Python 3.6.7

2.2 実験の構成と目的

実験は，以下にように分けられる．

実験 1　基板取付スピーカーを用いた電子オルゴール
実験 2　波形データの wavファイルへの録音

音は以下の３つの要素からなる．

1. 音程（ドレミファソラシド，周波数）
2. 音量（振幅）
3. 音色（波形）

このうち，実験１では音程のみを制御する．実験２では，全ての要素を制御する．

人の五感のうち，波の周波数や形を直に感じることができるのは聴覚のみである．音声処理を通して，信号処理を身近に体験することを実験全体の目的とする．

2.3 教材準備にあたっての方針

プログラムについては，以下の点に注意した．

- 基本的なプログラムは，ダウンロードできるようにした．
- 新たに導入するパッケージの使い方が複雑な場合は，このパッケージをラップするモジュールを別ファイルとしてまとめた．この工夫により，学生が目にするメインプログラムを簡単化できた．

上記の注意は，以下の学修効果を狙ったものである．

- まずはダウンロードしたプログラムの実行を体験し，面白さを味わえること．
- 不具合が起きた場合，電子回路，計算機環境（学生 PC およびラズベリーパイ），プログラム側（タイプミスも含む）の原因が考えられるが，学生にとって原因の切り分けは容易でない．上記のプログラムの準備により，プログラム側に起因する不具合は避けることができる．
- １段階ずつ動作を検証するプログラムを作成していくスタイルを紹介するため．これは，ソフトウエアの**テスト駆動開発**に通じる指針である．

使用した電子部品はスピーカーとコンデンサが２つずつのみであり，プログラムの工夫次第で可能性が広がる教材といえよう．本教材では，以下のプログラミング技法を含んでいる．

1. GPIO (General-Purpose Input/Output) の制御（3.3 節，3.4 節，3.6 節，3.7 節，3.8 節）
2. Python のリスト（C 言語の配列に相当）（3.5 節以降）
3. 正規表現を用いたパース処理（3.6 節，3.7 節，4.4 節）
4. プロセス単位での並列処理（3.7 節）
5. wavファイルの読み書き処理（4.1 節，4.2 節）
6. GUI 処理（波形の表示）（4.3 節）
7. コマンドライン引数の処理（4.3 節）
8. オブジェクト指向のクラスの定義（4.4 節）

上記のうち，前提とできる知識は項目１のみである．

演習課題の作成にあたっては，以下の点に留意した．

- 難問を少数出題するのではなく，簡単な課題を多数出題するという方針を立てた．
- 課題は，与えたプログラムを書き換える形態とした．
- 新たなパッケージや文法事項（リスト内包表記など）の調査は，選択の課題とした．また，明示されていなくても，そのような課題は自ら見つけることを推奨した．
- プログラムの作成ができなければ，次の節に進めないような設問の仕方は避けた．

3 [実験 1] 基板取付スピーカーを用いた電子オルゴール

実験１では，ラズベリーパイからスピーカーを制御する方法を学ぶ．

3.1 ラズベリーパイのハードウエア PWM

ラズベリーパイからスピーカーを操作をするには，以下の２通りの方法がある．

1. スピーカーに D/A 変換器を接続する．この D/A 変換器をラズベリーパイから制御する．
2. ディジタル信号により，スピーカーを制御する．ディジタル信号は，HIGHと LOWのみで構成されるため，出力できる波形は矩形波だけである．音色や音量を変えることはできない．

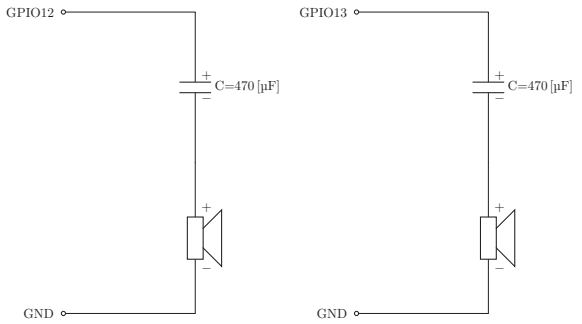
本稿では，手軽な後者の方法を用いる．この制御を簡単かつ精確に行うには，PWM (Pulse Width Modulation) という矩形波を発生させる機構を用いる．PWM には，ハードウエア PWM とソフトウエア PWM があるが，より高精度なハードウエア PWM を主に使用する．ハードウエア PWM を制御するため，pigpioという Python パッケージを用いた．

ラズベリーパイのハードウエア PWM には，PWM0 と PWM1 という２つのチャンネルがある．各チャンネルには，独立に周波数（およびデューティ比）を設定できる．これらのチャンネルに対応する GPIO は複数あるが，PWM0 に対しては GPIO12 を，PWM1 に対しては GPIO13 を用いることにする．

ハードウエア PWM は 2 チャンネルあるため，２つの旋律からなる楽曲の演奏が可能である．そこで，ベートーベンの「エリーゼのために」を題材とする．

3.2 回路図

スピーカーには，基板取付スピーカー [5] を用いた．図１に示す回路をブレッドボード上に作製した．図１では，**カップリングコンデンサ**を接続し，直流分を除去している．この除去により，スピーカーの振動板が片方に寄って損傷することを防止できる．



(a) スピーカー１　　(b) スピーカー２

図１　ラズベリーパイと２個のスピーカーの接続

3.3 スピーカーでの単音の演奏

本節では，スピーカー２個のうち「スピーカー 1」のみを用いる．スピーカー１から「ラ」を１秒間だけ発音するプログラムをソースコード１に示す．

ソースコード１　スピーカー１から「ラ」を鳴らす．`play_tone.py`

```
import pigpio
import time

PWM0_PIN = 12

p = pigpio.pi()
frequency = 440 #  オクターブ4の「ラ」の周波数
p.hardware_PWM(PWM0_PIN, frequency, 500000) #  周波数frequencyの
        PWMが出力される。500000はデューティ比50%を意味する。
time.sleep(1.0) #  1秒間待つ。(この間、PWMが出力され続ける。)
p.hardware_PWM(PWM0_PIN, 0, 0) #
        PWMの周波数を0、デューティ比を0とする。(PWMを止める。)
```

音程と周波数に関して，以下の取り決めを紹介した．

- 1 オクターブ上がると，どの音も周波数は 2 倍になる．
- 1 オクターブは，12 音からなる．各音は，1 オクターブを 12 等分した周波数となる．

3.4 スピーカーでの和音の演奏

和音を演奏するには，単に，基板取付用スピーカーを２個を使って出力すれば良い．スピーカー１で「ド」を，スピーカー２で「ミ」を同時に１秒間だけ鳴らすプログラムをソースコード２に示す．

ソースコード２　スピーカー１から「ド」，スピーカー２から「ミ」を鳴らす．`play_waon.py`

```
import pigpio
import time

PWM0_PIN = 12
PWM1_PIN = 13

p = pigpio.pi()
frequency0 = 261 #  オクターブ4の「ド」の周波数
frequency1 = 330 #  オクターブ4の「ミ」の周波数
p.hardware_PWM(PWM0_PIN, frequency0, 500000) #
        PWM0_PINから周波数frequency0のPWMが出力される。
p.hardware_PWM(PWM1_PIN, frequency1, 500000) #
        PWM1_PINから周波数frequency1のPWMが出力される。
time.sleep(1.0) #  1秒間待つ。(この間、PWMが出力され続ける。)
p.hardware_PWM(PWM0_PIN, 0, 0) #
        PWM0の周波数を0、デューティ比を0とする。(PWMを止める。)
p.hardware_PWM(PWM1_PIN, 0, 0) #
        PWM1の周波数を0、デューティ比を0とする。(PWMを止める。)
```

3.5 楽譜データの形式

実験１と実験２では「エリーゼのために」の楽譜を使用する．楽譜は，Python の２重リストを用いて，以下のように表現する．（楽譜を２重リスト形式で定義するというアイデアは，製品 [6] のサンプルプログラムを参考にした．）

```
UPPER_REPEAT1 = [
    ["e5",16],["e-5",16],
    ["e5",16],["e-5",16],["e5",16],["b4",16],["d5",16],["c5",16],
    ["a4",8],["r",16],["c4",16],["e4",16],["a4",16],
    ["b4",8],["r",16],["e4",16],["a-4",16],["b4",16],
    ["c5",8],["r",16],["e4",16],["e5",16],["e-5",16],
    ["e5",16],["e-5",16],["e5",16],["b4",16],["d5",16],["c5",16], #
            第5小節終わり
    ["a4",8],["r",16],["c4",16],["e4",16],["a4",16],
    ["b4",8],["r",16],["e4",16],["c5",16],["b4",16]
]
```

上記の２重リストは，上パート（主旋律）の楽譜の最初の部分である．

２重リストのうち，内側の各リストは１つの音符または休符を表す．内側のリストの各要素の意味は，以下の通りである．

0 番目の要素　（文字列．音名とオクターブ）

音名は，プログラムで扱いやすいようにアルファベットで表記する．カタカナ音名とアルファベット名との対応を表 1 に示す．語尾に以下を付けることもできる．

- シャープ（半音上げる）を + で表す．
- フラット（半音下げる）を–で表す．

1 **番目の要素**　（整数．音符または休符の分割数）

例えば，16は，16 分音符を意味する．

2 **番目の要素**　（文字列．付点がある場合）

付点音符・休符の場合，半角ピリオドからなる文字列"."を持つ．付点でない場合は，この要素は存在しない．

表 1　カタカナ音名とアルファベット名の対応

カタカナ音名	ド	レ	ミ	ファ	ソ	ラ	シ	休符
アルファベット名	c	d	e	f	g	a	b	r

例を以下に挙げる．

- 最初の音符を表すリスト ["e5",16]は，オクターブ 5 の「ミ」の 16 分音符を意味する．
- オクターブ 4 の「ラ」のフラットは"a-4" と表す．
- 付点 8 分音符は ["e5",8,"."]と表す．
- ["r",16]は 16 分休符を表す．

リストの結合には，以下のように +演算子を用いる．これにより，楽譜データを結合することができる．

```
UPPER_PART = UPPER_REPEAT1 + UPPER_sono1_after_REPEAT1 +
        UPPER_REPEAT1 + UPPER_sono2_after_REPEAT1 +
        UPPER_REPEAT2 + UPPER_sono1_after_REPEAT2
```

上・下パートとも楽譜を与えたのは，全体の 4 分の 1 程度である．残りの部分をリスト形式で追記する事も選択の課題とした．

3.6　「エリーゼのために」の上パートのみの演奏

「エリーゼのために」の上パート（主旋律）を「スピーカー 1」のみを用いて演奏してみよう．そのためのプログラムをソースコード 3 に示す．

ソースコード 3　「エリーゼのために」の上パートのみを演奏する．`play_for_Elise_upper_part.py`

```
import pigpio
import re
import time


PWM0_PIN = 12

TEMPO = 60 # テンポ(1分あたりの四分音符の数)
DURATION_PER_NOTE = 60 / TEMPO # 4分音符1つあたりの発音時間

# 楽譜(2重リスト)UPPER_PARTとLOWER_PARTの定義は省略

NOTE_DIFF = {"c": -9, "c+": -8, "d-": -8, "d": -7, "d+": -6,
             "e-": -6, "e": -5, "f": -4, "f+": -3, "g-": -3,
             "g": -2, "g+": -1, "a-": -1, "a": 0, "a+": 1,
             "b-": 1, "b": 2, "r": None}

note_pattern = re.compile(r"^([a-gr])(\+|\-)?(\d+)?", re.IGNORECASE)
TUNING = 440 # 基準となるオクターブ4の「ラ」の周波数

def play_tone(p, pin, tone):
    note_match = note_pattern.search(tone[0])
    (note_code, accidential, octave) = note_match.groups()
    note_acc = note_code + (accidential if accidential else "") #
            シャープやフラットの文字を付加
    # 音程の取得と周波数の計算
    if note_code == "r": # 休符(rest)の場合
        frequency = 0
    else: # 音符の場合
        note_diff = NOTE_DIFF[note_acc]
        octave = int(octave) # 文字列を整数に変換
        frequency = TUNING * \
                    (2 ** (octave -4)) * \
                    ((2 ** note_diff) ** (1 / 12.0))

    duration = 4 * DURATION_PER_NOTE / tone[1] #
            発音(または非発音)時間を計算
    if len(tone) > 2:
        if tone[2] == ".": # 付点音符(または休符)の場合
            duration *= 1.5
    p.hardware_PWM(pin, int(frequency), 500000) # デューティ比50%
    time.sleep(duration)

def play(p, pin, song):
    p.set_mode(pin, pigpio.OUTPUT)
    try:
        for t in song:
            play_tone(p, pin, t)
    except KeyboardInterrupt:
        pass

    p.hardware_PWM(pin, 0, 0) #
            PWMの周波数を0、デューティ比を0とする。(PWMを止める。)
    p.set_mode(pin, pigpio.INPUT)


# 以降はメインプログラム
p = pigpio.pi()
play(p, PWM0_PIN, UPPER_PART)
```

3.3 節で実習したように，スピーカーの駆動には，周波数と発音時間の情報が必要である．その情報を，ソースコード 3 では以下のように求めている．

1. 関数 playでは，2 重リストから順に 1 重リスト（1 つの音符についての情報）を取り出し，関数 play_tone に渡す．
2. 関数 play_toneでは，1 重リストから，正規表現の処理を行うパッケージ reを用いて，音符データ（オクターブ，シャープとフラットも含めた階名，付点の有無を含めた音符の長さ）を取り出す．この音符データを用いて，以下の計算を行う．
   (a) オクターブと階名から，周波数 frequencyを計算する．
   (b) 音符の長さから，発音時間（休符の場合は非発音時間）durationを計算する．

3.7　「エリーゼのために」の上・下パートの演奏

複数パートをもつ楽曲において，（2 つ目の音符以降の）発音のタイミングはバラバラである．このような制御をプログラムで行うには工夫が要る．ここで，楽譜を読み，音を鳴らす処理は，パートごとに独立していることに気づくだろう．そこで，上・下パートに対応する**子**

プロセスを1つずつ立ち上げ，これらの処理を並行して行わせることにする．プロセスの処理には，Python のパッケージである multiprocessingを用いる．

以上の方針で，「エリーゼのために」の上・下パートを演奏するプログラムをソースコード4に示す．

ソースコード4　「エリーゼのために」の上・下パートを和音で演奏する．play_for_Elise_waon.py

```
import pigpio
import re
import multiprocessing
import time


PWM0_PIN = 12
PWM1_PIN = 13

TEMPO = 60 # テンポ(1分あたりの四分音符の数)
DURATION_PER_NOTE = 60 / TEMPO # 4分音符1つあたりの発音時間

# 楽譜(2重リスト)UPPER_PARTとLOWER_PARTの定義は省略

NOTE_DIFF = {"c": -9, "c+": -8, "d-": -8, "d": -7, "d+": -6,
             "e-": -6, "e": -5, "f": -4, "f+": -3, "g-": -3,
             "g": -2, "g+": -1, "a-": -1, "a": 0, "a+": 1,
             "b-": 1, "b": 2, "r": None}

note_pattern = re.compile(r"^([a-gr])(\+|\-)?(\d+)?", re.IGNORECASE)
TUNING = 440 # 基準となるオクターブ4の「ラ」の周波数

def play_tone(p, pin, tone):
    note_match = note_pattern.search(tone[0])
    (note_code, accidential, octave) = note_match.groups()
    note_acc = note_code + (accidential if accidential else "") #
            シャープやフラットの文字を付加
    # 音程の取得と周波数の計算
    if note_code == "r": # 休符(rest)の場合
        frequency = 0
    else: # 音符の場合
        note_diff = NOTE_DIFF[note_acc]
        octave = int(octave) # 文字列を整数に変換
        frequency = TUNING * \
                    (2 ** (octave -4)) * \
                    ((2 ** note_diff) ** (1 / 12.0))

    duration = 4 * DURATION_PER_NOTE / tone[1] #
            発音(または非発音)時間を計算
    if len(tone) > 2:
        if tone[2] == ".": # 付点音符(または休符)の場合
            duration *= 1.5
    p.hardware_PWM(pin, int(frequency), 500000) # デューティ比50%
    time.sleep(duration)

def play(p, pin, song):
    p.set_mode(pin, pigpio.OUTPUT)
    try:
        for t in song:
            play_tone(p, pin, t)
    except KeyboardInterrupt:
        pass

    p.hardware_PWM(pin, 0, 0) #
            PWMの周波数を0、デューティ比を0とする。(PWMを止める。)
    p.set_mode(pin, pigpio.INPUT)


# 以降はメインプログラム
p = pigpio.pi()
p1 = multiprocessing.Process(target=play, args=(p, PWM0_PIN,
        UPPER_PART))
p2 = multiprocessing.Process(target=play, args=(p, PWM1_PIN,
        LOWER_PART))

p1.start()
p2.start()

p.stop()
```

ソースコード4のプログラムを理解するために，multiprocessingパッケージの理解が不可欠である．そのため，multiprocessingの使用例（ソースコード5）を与え，この動作を理解することも選択の演習課題とした．

ソースコード5　multiprocessingの簡単な使用例 multiprocessing_simple.py

```
import multiprocessing
import time

def func(str):
    while True:
        print(str)
        time.sleep(1)


p1 = multiprocessing.Process(target=func, args=("1"))
p2 = multiprocessing.Process(target=func, args=("2"))

p1.start()
p2.start()
```

3.8　精度が悪いソフトウエア PWM を使用した演奏

ソフトウエア PWM を用いた場合の音質を体験するため，プログラム（本稿への掲載は省略）を用意した．ソフトウエア PWM を扱うため，RPi.GPIOというパッケージを用いた．

4　[実験 2] 波形データの wavファイルへの録音

コンピュータにおいて，音声データはディジタルデータとして扱われる．その仕組みを wavファイル形式を用いて学修する．

実験2のプログラムの実行は，ラズベリーパイだけでなく，Python がインストールされたパソコンでも可能である．ラズベリーパイやパソコンには，基板取付用スピーカーよりも高度なオーディオを扱える仕組みが備わっている．ラズベリーパイ本体には，スピーカーは内臓されていないが，3.5mm ステレオミニプラグ用の端子が備わっている．この端子にヘッドホンやイヤホンを接続できる．

生成された wavファイルは，以下の2通りの方法で再生できる．

1. ラズベリーパイにヘッドホンやイアホンを接続して，aplayコマンドで再生する．
2. ラズベリーパイから Windows 等のパソコンにファイルをコピーする．パソコンの内臓スピーカーやイヤホンを用いて再生する．

4.1　wavファイルの読み書きを行うパッケージ

wavファイルの読み書きには，Python3 に標準ライ

ブラリとして含まれる waveパッケージを用いる．wave パッケージを簡単に扱えるように，ソースコード 6 に示す wave_toolというモジュールを用意した．wave_tool は，簡単のため，クラス化は施していない．代わりに，各関数の引数として，作成したファイルオブジェクトを渡す方式にしている．

ソースコード 6　wav ファイルに録音するためのモジュール wave_tool.py

```
import wave
import struct

channels = 2
sample_width = 2
sample_rate = 44100
sample_bits = sample_width * 8
max_gain = 2 ** sample_bits / 2 -1

def open(output_file_name, sampling_rate=44100):
    global sample_rate
    sample_rate = sampling_rate
    output_wave_file = wave.open(output_file_name, "wb")
    output_wave_file.setnchannels(channels)
    output_wave_file.setsampwidth(sample_width)
    output_wave_file.setframerate(sample_rate)
    return output_wave_file

def close(output_wave_file):
    output_wave_file.close()

def write(file, data, gain=max_gain):
    file.writeframesraw(b"".join([struct.pack("h", int(x*gain) ) for x in
        data]))
```

このモジュールについては，次に示すメインプログラムのソースコード 7 を見ながら使い方を理解するだけでよく，中身を理解する必要はない事を補足した．（もちろん，中身を理解することを課題としても良い．）

4.2　単音の wavファイルの生成

wave_toolを用いたもっとも簡単なサンプルとして，単音を生成してみる．ソースコード 7 は，1 秒間の正弦波のデータを作成し wavファイルに出力するプログラムである．周波数は 440 [Hz]（オクターブ 4 の「ラ」）である．

ソースコード 7　正弦波のデータを作成し，wav ファイルに出力する．generate_sin_wave.py

```
import wave_tool
import math

def sin_wave(frequency, tick):
    tick_mod = tick % wave_tool.sample_rate
    t = tick_mod / wave_tool.sample_rate
    omega = 2.0 * math.pi * frequency
    value = math.sin(omega * t)
    return value


# 以降はメインプログラム
import numpy as np
file = wave_tool.open("sin.wav")

frequency = 440.0
data = [sin_wave(frequency, i) for i in range(wave_tool.sample_rate)] #
        リスト内包表記を用いる。1秒間の正弦波をリストとして得る。
print("data[:10]=", data[:10]) # 最初の10個を表示してみる。
wave_tool.write(file, data) # ファイルに書き込む。
wave_tool.close(file)
```

ソースコード 7 を実行すると，確認のため，ターミナルに音データの最初の 10 個が表示される．（音は出ない．）

```
data[:10]= [0.0, 0.06264832417874369, 0.1250505236945281,
     0.18696144082725336, 0.2481378479437379,
     0.30833940305910035, 0.36732959406137883,
     0.42487666788983847, 0.4807545410165317,
     0.5347436876541296]
```

ここで表示された数値は，$-1.0 \sim 1.0$ に正規化された値である．また，ファイル sin.wavがカレントディレクトリに生成される．

音色については，正弦波，三角波，矩形波で作った波形を実際に聞いてみるという課題を出題した．このような簡単な波形でも，異なる音色に聞こえることがわかる．ノコギリ波の作成は選択の演習課題とした．

デフォルトの音質は，音楽 CD と同等とした：

- サンプリング周波数：44.1 [kHz]
- 量子化ビット数：16 ビット
- チャンネル数：2（ステレオ）

サンプリング周波数は，**open**関数のキーワード引数 sampling_rateにより変更できる．サンプリングレートを変えた時の音質の変化を調べることも課題とした．また，write関数のキーワード引数 gainにより増幅度を指定できる．

4.3　wavファイルの波形の描画

4.2 節では，波形のリスト要素の値を見たが，波形が正しく作られたかを確かめるには可視化することが望ましい．wav ファイルを可視化するためのプログラムをソースコード 8 に示す．ソースコード 8 は道具として使えれば良く，中身を理解する必要はない事を補足した．

ソースコード 8　wav ファイルを可視化するためのプログラム plot_wav.py

```
import wave
import numpy as np
import matplotlib.pyplot as plt
import sys

args = sys.argv

if len(args) < 2:
    print("args=", args)
    print("エラー：wavファイルの名前を指定してください。")
    exit()

if len(args) >= 3:
    end_tick = int(args[2]) #
        コマンドライン引数の2つ目は、描画の終了ティック
    print("end_tick=", end_tick) # ユーザの確認のため、画面に表示する。

file_name = args[1] # コマンドライン引数の１つ目はファイル名
```

```
wf = wave.open(file_name, "r")
buf = wf.readframes(wf.getnframes())

# バイナリデータを整数型(16bit)に変換
data = np.frombuffer(buf, dtype="int16")
if len(args) >= 3:
    data = data[:end_tick]

# グラフ化
plt.plot(data, linewidth=0.3, marker='o', markersize=1) #
        線は細く、マーカーは小さくする。
plt.grid()
plt.title(file_name)
plt.xlabel("時間(整数値)")
plt.ylabel("値(整数値)")
plt.savefig(file_name + ".png")
plt.show()
```

上記のプログラム plot_wav.pyの実行に先立って，ラズベリーパイに sshコマンドによりログインする際に，オプション-YCを付けなければならない．これにより，X Window をデータ圧縮モードで転送できる．

実行コマンドは，以下の通りである．コマンドライン引数として，描画したい wavファイル名と描画を終了する（整数単位の）時間を渡す．

```
python plot_wav.py sin.wav 1000
```

実行すると，別ウインドウで画像が表示されるとともに，図 2 に示す画像ファイル sin.wav.pngがカレントディレクトリに作成される．なお，matplotlibでは pdf ファイルに日本語フォントの埋め込みができないため，pngファイルとして出力している．

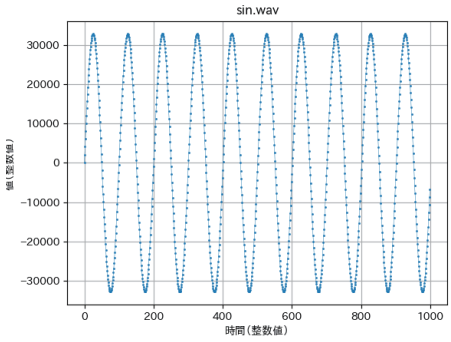


図 2　plot_wav.pyで作成された画像ファイル sin.wav.png

4.4　「エリーゼのために」の上・下パートの録音

実験 1 で定義した楽譜を解釈し，wavファイルを生成するプログラムを用意した．このプログラムは**ミュージックシーケンサー**の簡易版である．

シーケンサーのモジュールを，ソースコード 9 に示す．波形の生成はソースコード 10 で定義されるクラスで行う．(4.2 節では，波形を関数として定義したが，角周波数などの定数の事前計算による効率化のため，クラスとした．）このモジュールの作成にあたっては，プログラム [7] とその講習会のスライド [8] を参考にした．

ソースコード 9　楽譜データから wavファイルを生成するためのモジュール score.py

```
import re
import wave_tool

TUNING = 440 # 基準となるオクターブ4の「ラ」の周波数

NOTE_DIFF = {"c": -9, "c+": -8, "d-": -8, "d": -7, "d+": -6,
             "e-": -6, "e": -5, "f": -4, "f+": -3, "g-": -3,
             "g": -2, "g+": -1, "a-": -1, "a": 0, "a+": 1,
             "b-": 1, "b": 2, "r": None}

def score_to_sequence(song, tempo=120):
    tick_position = 0

    note_pattern = re.compile(r"^([a-gr])(\+|\-)?(\d+)?", re.
        IGNORECASE)

    sequence = list()
    for tone in song:
        note_match = note_pattern.search(tone[0])
        (note_code, accidential, octave) = note_match.groups()
        note_acc = note_code + (accidential if accidential else "") #
            シャープやフラットの文字を付加
        # 音程の取得と周波数の計算
        if note_code == "r": # 休符(rest)の場合
            note_frequency = 0
        else: # 音符の場合
            note_diff = NOTE_DIFF[note_acc]
            octave = int(octave) # 文字列を整数に変換
            note_frequency = TUNING * \
              (2 ** (octave -4)) * \
              ((2 ** note_diff) ** (1 / 12.0))

        # 音符(または休符)の長さ
        note_length = tone[1]
        note_on_tick = wave_tool.sample_rate * (60.0 / tempo) * (4.0 /
            note_length)
        if len(tone) > 2:
            if tone[2] == ".": # 付点音符(または休符)の場合
                note_on_tick *= 1.5

        # 発音の終了ティックを更新する
        tick_position += note_on_tick

        # シーケンス(音楽データ)に追記
        sequence.append((note_frequency, int(tick_position)))

    return sequence


class sequencer(object):
    def __init__(self):
        self.tracks = dict()
        self.volumes = dict()
        self.sequences = dict()

    def add_track(self, track_id, sequence_data, form, volume=1.0):
        self.tracks[track_id] = form
        self.volumes[track_id] = volume
        self.sequences[track_id] = sequence_data

    def get_value(self, tick, stereo=True):
        for (track_id, sequence_data) in self.sequences.items():
            if tick > sequence_data[0][1]:
                del sequence_data[0]

            if len(sequence_data) == 0:
                return None

            (frequency, off_tick) = sequence_data[0]
            form = self.tracks[track_id]
            form.set_frequency(frequency)

        if stereo:
            return [ self.tracks[0].get(tick) * self.volumes[0] , self.tracks
                [1].get(tick) * self.volumes[1] ] #
                [ 左チャンネル, 右チャンネル ]
        else:
            result = 0
```

```
            for (track_id, form) in self.tracks.items():
                result += form.get(tick) * self.volumes[track_id]
            result = result / len(self.tracks) #  正規化して、足し込み
            return [result, result]


def render(file_name, sequencer, stereo=True, gain=wave_tool.max_gain):
    file = wave_tool.open(file_name)

    tick = 0
    while True:
        data = sequencer.get_value(tick, stereo)
        if data is None:
            break
        wave_tool.write(file, data, gain)

        tick += 1

        if (tick % 50000) == 0:
            print("{:8d} ticks ... ".format(tick))

    wave_tool.close(file)
```

ソースコード 10　正弦波，矩形波，三角波を生成するクラス `wave_form.py`

```
import wave_tool
import math

class sin_wave():
    def __init__(self, frequency=440.0):
        self.frequency = frequency

    def set_frequency(self, frequency):
        self.frequency = frequency
        self.omega = 2.0 * math.pi * self.frequency

    def get(self, tick):
        value = math.sin(self.omega * tick / wave_tool.sample_rate)
        return value


class square_wave():
    def __init__(self, frequency=440.0, duty=0.5):
        self.frequency = frequency
        self.duty = duty

    def set_frequency(self, frequency):
        self.frequency = frequency

    def get(self, tick):
        if self.frequency == 0:
            return 0
        else:
            l = wave_tool.sample_rate / self.frequency
            change_point = l * self.duty

            if (tick % l) <= change_point:
                return 1
            else:
                return -1


class triangular_wave():
    def __init__(self, frequency=440.0):
        self.frequency = frequency

    def set_frequency(self, frequency):
        self.frequency = frequency

    def get(self, tick):
        if self.frequency == 0:
            return 0
        else:
            l = wave_tool.sample_rate / self.frequency
            slope = 1 / (l/4)
            tick_mod = tick % l
            if (tick_mod < l/4):
                value = slope * tick_mod
            elif (l/4 <= tick_mod < 3*l/4):
                value = 1 -slope * (tick_mod-l/4)
            else: #  3*l/4 <= tick_mod < lの場合
                value = -1 + slope * (tick_mod-3*l/4)
            return value
```

ソースコード 9 とソースコード 10 を用いて「エリーゼのために」の上・下パートを wavファイルに出力するプログラムをソースコード 11 に示す．

ソースコード 11　「エリーゼのために」の上・下パートを wavファイルに出力する．`generate_wav_for_Elise.py`

```
import score
import wave_form

#  楽譜（２重リスト）UPPER_PARTとLOWER_PARTの定義は省略
music1 = score.score_to_sequence(UPPER_PART, tempo=60)
music2 = score.score_to_sequence(LOWER_PART, tempo=60)

sequencer = score.sequencer()
sequencer.add_track(0, music1, form=wave_form.square_wave()) #
        トラック0：上パート、矩形波を使用
sequencer.add_track(1, music2, form=wave_form.sin_wave()) #
        トラック1：下パート、正弦波を使用

score.render("For_Elise.wav", sequencer)
```

ソースコード 11 の実行により，ファイル `For_Elise.wav`が生成される．（このファイルに録音された時間は 46 秒，ファイル容量は 8M バイト，生成に要した時間は 90 秒ほどである．）このファイルには，左チャンネルに上パート，右チャンネルに下パートが録音されている．このステレオの効果は，イヤホンで聞くとよく分かる．

各関数にはキーワード引数を定義し，振る舞いを変更できるようにした．キーワード引数を表 2 に示す．さらに，以下に補足する：

- キーワード引数 formにより，トラック（パート）ごとに波形（音色）を指定できる．波形はソースコード 10 においてクラスとして定義されており，そのクラスのオブジェクトを渡す．
- キーワード引数 stereo=Falseの場合，２つのトラックの算術平均が両チャンネルに出力される．つまり，実質的にモノラルとなる．

表 2　関数に渡すキーワード引数

関数名	キーワード引数名	デフォルト値
score_to_sequence	tempo	60
add_track	form	
add_track	volume	1.0
render	stereo	True

5　まとめ

インテリジェントデバイス実験 I で実施した「ラズベリーパイを用いた電子オルゴール」について述べた．実験は，以下に分けて行った．

実験 1　基板取付スピーカーを用いた電子オルゴール

実験 2　波形データの wavファイルへの録音

どちらの実験でも，「エリーゼのために」の上・下パートを演奏することをゴールとして，プログラムを段階的に改良していく方式をとった．本稿では，メインプログラムを簡単化するために作成したパッケージや波形の描画ツールとなるプログラムも合わせて紹介した．

5.1　学修効果

本テーマは，1 回分の授業（3 時間）として実施した．実験テキストは，物理 (波動，音楽理論，電気回路)，プログラミングにまたがる内容を盛り込んだため，30 ページを超えた．そのため，必須として出された課題をこなすだけで精一杯だったと考えられる．特に，3.5 節以降のプログラムを理解するのは，難しかったようである．大部分のプログラムを用意して与えたため，時間が掛かっても，プログラムをゼロから自力で組む体験をできないという問題もあった．

来年度以降は，本テーマを 2 回以上の授業に分けて実施することも検討している．また，プログラムの実行の体験は低学年の別の授業で行うという方法もあるだろう．そのようにすれば，本授業では電子回路とプログラムの内容の理解に重点をおくことができる．

5.2　今後の課題

実験 1 に関する課題として，以下があげられる．

- スピーカー制御のリアルタイム性を高めるために，楽譜データの前処理を行う．
- DMA (Direct Memory Access) を用いた 3 個以上のスピーカーの制御
- 音質向上のため，単電源オペアンプを用いた増幅回路を作成し，ラズベリーパイと基板取付スピーカーの間に接続する．
- ラズベリーパイではジッターの影響がある．ジッターの影響を排除するため，ラズベリーパイと PIC マイコンを組み合わせる．

実験 2 に関する課題として，以下が挙げられる．

- wavファイルの生成時間の短縮する．例えば，C++ を用いて実装し，Python から利用できるようにバインディングを作成する．
- 各種のエフェクタのプログラム実装
- wavファイルの再生の形で，任意の波形の生成装置（ファンクションジェネレータ）を作製できる．これを電子回路実験の入力電圧として用いる．

実験 1 と実験 2 のプログラムは，どちらも数百行程度で実現できた．これは，20 年前のマイコンでは考えられなかったことであり，その点においては画期的といえる．一方，現在はハイレゾレベルの音声処理が当たり前に行われる時代である．そのような最新の技術も取り入れ，学生の興味を惹く教材の開発に取り組みたい．

参考文献

[1] 坂下達哉. インテリジェントデバイス実験 I 後半　実験テキスト. 工学部 情報通信工学科, 2019.

[2] 坂下達哉. インテリジェントデバイス実験 I 後半　スライド. 工学部 情報通信工学科, 2019.

[3] 金丸隆志. カラー図解 最新 Raspberry Pi で学ぶ電子工作作って動かしてしくみがわかる. ブルーバックス. 講談社, 2016.

[4] 福田和宏.　これ 1 冊でできる！ラズベリー・パイ 超入門 改訂第 5 版 Raspberry Pi 1+/2/3 (B/B+)/Zero/Zero W 対応. ソーテック社, 2018.

[5] 秋月電子通商.　基板取付用スピーカー（磁気サウンダー）直径 12mm QMB-111GPN. http://akizukidenshi.com/catalog/g/gP-11240/.

[6] Kuman. 44 個キット Raspberry Pi 用センサー センサーモジュール 38 センサーモジュール +ADC0832 チップセット +GPIO 拡張ボード + ジャンパーワイヤー電子部品 電作キット 実験用 Raspberry Pi 3 / 2 Model B B+ A A+ に適用 K47.

[7] 磯蘭水. 「ソフトシンセを作りながら学ぶ Python プログラミング (初級者)」で使ったシンセモジュールとサンプルプログラム. https://github.com/oldgeese/toysynth.

[8] 磯蘭水. ソフトシンセを作りながら学ぶ Python プログラミング. https://www.slideshare.net/RansuiIso/python-14315727.

2020 年 2 月 29 日原稿受付，2020 年 3 月 6 日採録決定
Received, February 29, 2020; accepted, March 6, 2020