The Improvisor

Ajit Nath
July 23, 2014

1 INTRODUCTION

In this project we're trying to develop a tool for students to allow them to learn to play a new piece of music on guitar and perhaps add their own ideas to it. It is closely tied to automatic transcription and algorithmic music composition which fall under the wider field of machine auditioning. Machine auditioning as we know, includes transduction, grouping, use of musical knowledge and general sound semantics for the purpose of performing intelligent operations on audio and music signals[1].

1.1 SCOPE

With the advent of internet and computers, learning to play an instrument has become much easier. There are tonnes of websites and software which offer tabs and score allowing users to learn to play their favorite songs quickly.Guitar pro, tux guitar(GPL license) and websites such as `www.ultimate-guitar.com`, `www.fretplay.com` are very popular among students. Video streaming websites such as `www.youtube.com` and `www.vimeo.com` have also greatly helped spreading musical ideas among new students. However these methods being as good as they are do not make the learning process interactive. Moreover, transcribing is becoming a dying art among novice and intermediate guitar players . Gone are the days when people used to listen to a track on the tape over and over again and try to play. Ear training is very important aspect of being a better overall musician and tabs do not help with that. In fact many of famous contemporary guitar players do not even know how to read sheet music or tabs.But it is not to say ear workout is completely neglected by guitar learning software of today. There are many ear training software available online such as ear master, GNU solfege but they focus on training on intervals and chords rather than songs and the process can get

monotonous quickly. My aim for this project was to use an ear training approach to learning songs. In a way user's ear would be guided to identifying pitches and playing them properly.

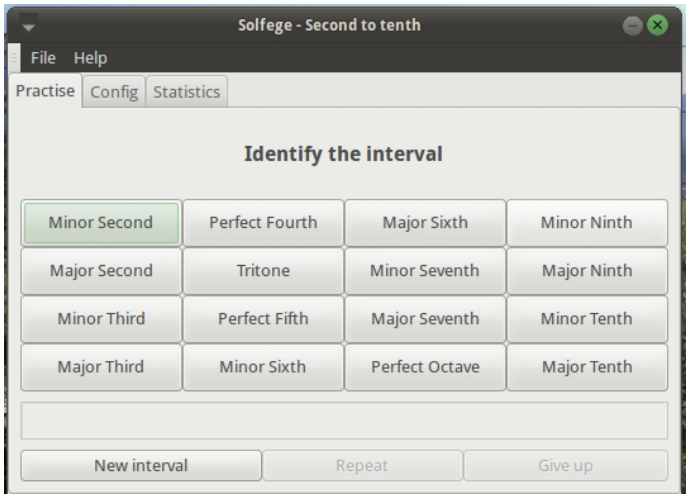

Figure 1.1: Solfege Ear Training Software for linux being used for interval training

1.2 Background

For readers unacquainted with music theory here's a quick background:

A piece of music has many elements which can be used to describe it, these include pitch, beat, melody, harmony, rhythm, timbre(or color) and structure. We'll be using these terminologies frequently in the following sections. Pitch can be defined as how 'low' or 'high' a musical note sounds. Most people perceive pitch as relative to a reference frequency. Perceived pitch is nearly always closely connected with the fundamental frequency of a note[2].In general, the higher the frequency of vibration, the higher the perceived pitch is, [3]. In western music there's a notion of 'tuning' or standard frequency relative to which, other notes are measured. Modern music uses a frequency of 440Hz as the standard.[4]

Notes or pitches can be arranged into different 'scales' and 'modes'. Western music generally divides the octave (An octave is an interval after which the notes of the scale repeat) into a series of 12 notes. This series of twelve notes is called a chromatic scale. In the chromatic scale, the interval between adjacent notes is called a half-step or semitone. Other scales commonly used in western music are Pentatonic(5 notes) and Major and Minor Diatonic(7 notes) The main objective of our project is to encourage instinctive musical ability. We do that by asking the user to play imitate what the computer plays by listening. If the user gets it wrong we use genetic 'crossover' operation to 'lead' him to the correct solution. In the next section we will discuss similar works which involve real-time human-computer music interaction.

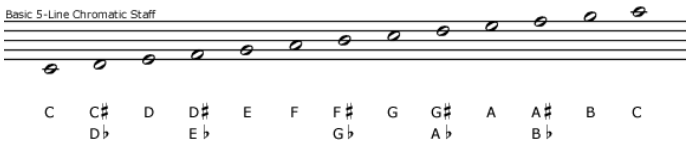

Figure 1.2: C chromatic scale

2 Related Work

2.0.1 Coach Guitar

Guitar Coach is an introduction to playing acoustic and classical guitar. The software details steps to follow to get started, but if you're already familiar with the basics, you don't have to go through it again. The methods are focused on developing good hand skills, musical awareness, and a thorough understanding of the guitar using video lessons and illustrations. It focuses on two-way interaction between user and software. [27]

2.1 Rock Prodigy

Rock prodigy is an IOS app similar to Coach Guitar in which you notes come by on the screen and you have to play those notes and you get points for that. It uses a propriety pitch detection algorithm. [28]

Figure 2.1: Rock prodigy, a guitar learning application(Image source : gizmag.com)

2.2 GenJam

John A. Biles created a genetic algorithm GenJam to improvise Jazz solos [7]. GenJam maintains hierarchy of related populations of melodic ideas that are mapped to specific notes suggested by the underlying chord progression being played. As GenJam plays its solos over the accompaniment of a standard rhythm section, a human mentor gives feedback, which is used to derive fitness values for the individual measures and phrases. GenJam then applies

various genetic operators to the populations to breed improved generations of ideas [7]. In other words, the solos generated are influenced by asthetic inclination of the user. Here's a schematic diagram of Genjam.

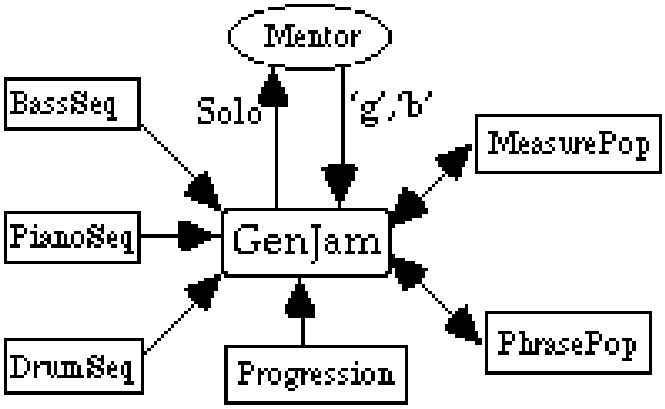

Figure 2.2: genjam schema

From the figure you can see that the additional information such as chord progression, rhythm section (bass section, piano sequence drum sequence) have to be provided to genjam. These are in form midi which could be produced using software like 'band in a box'.During the training phase, the genjam selects a random population of phrases and measures combines arranges them to produce licks. These are then heard by the mentor and who ranks them by pressing 'g' or 'b' on the keyboard ('g' stands for good and 'b' for bad likewise). The user can then 'trade fours' (A term used by jazz musicians which meaning two musicians play 4 measures each one after the other) with the software. However, training is a serious bottleneck because each phrase has to be heard individually one after the other. Consequently attempts were made to remove this bottle neck. John A Biles developed something called an autonomous genjam which doesn't have a training phase but instead uses a lick library to evolve new phrases. However since the licks are directly derived from what the user plays, the performance is very much based on how the skill level of the user[8]

3 METHODOLOGY

Before going into the details of implementation I'd like to talk a bit about some of the underlying concepts and ideas of this project.

3.1 PITCH DETECTION

A pitch detection algorithm (PDA) is an algorithm designed to estimate the pitch or fundamental frequency of a quasi-periodic or virtually periodic signal, usually a digital recording of speech or a musical note or tone. This can be done in the time domain or the frequency domain or both the two domains.[9].

In case of time domain pitch detection, the period of quasi-periodic signal is estimated and inverted to give frequency. quasi-periodic signals are those, which behave somewhat like periodic signals but aren't strictly periodic which is how audio signals behave.

In the frequency domain, polyphonic detection is possible, usually utilizing the periodogram to convert the signal to an estimate of the frequency spectrum. This requires more processing power as the desired accuracy increases. The well-known efficiency of the FFT, a key part of the periodogram algorithm, makes it suitably efficient for many purposes.[11]. Popular example of frequency domain detection is cepstral analysis[12].Other algorithms such as YAPPT[13] makes use of both temporal and frequency domain techniques

In our implementation we're using YIN which is an auto-correlation pitch detection algorithm.[29]. Correlation can be defined as the measure of dependence between two quantities[**?**].

$$corr(X,Y) = \frac{cov(X,Y))}{\sigma_X, \sigma_Y} \tag{3.1}$$

The idea of auto-correlation is to provide a measure of similarity between a signal and itself at a given lag. In context of pitch detection we use it by stepping through the signal sample-by-sample and perform a correlation between a reference window and the lagged window. The correlation at "lag 0" will be the global maximum because as we're comparing the reference to a copy of itself. As we step forward, the correlation will necessarily decrease, but in the case of a periodic signal, at some point it will begin to increase again, then reach a local maximum. The distance between "lag 0" and that first peak gives you an estimate of your pitch/tempo

3.2 Musical Instrument Digital Interface(MIDI)

MIDI (/ËĹmÉłdi/; short for Musical Instrument Digital Interface) is a technical standard that describes a protocol, digital interface and connectors and allows a wide variety of electronic musical instruments, computers and other related devices to connect and communicate with one another[15]

Below is a description of structure of midi file :

A midi (.MID) file has two components, Header chunks and Track chunks. A midi file contains ONE header chunk describing the file format, etc., and any number of track chunks. A track may be thought of in the same way as a track on a multi-track tape deck. We may assign each to different component of the music.[16]

The header chunk essentially consists of meta data. It looks something like this.

$$header\_chunk = "MThd" + < header\_length > + < format > + < n > + < division >$$

"$MThd$" 4 bytes
: the literal string MThd, or in hexadecimal notation: 0x4d546864. These four characters at the start of the MIDI file indicate that this is a MIDI file.

$<header\_length>$ 4 bytes
: length of the header chunk (always 6 bytes long–the size of the next three fields which are considered the header chunk).

$<format>$ 2 bytes
: 0 = single track file format
1 = multiple track file format
2 = multiple song file format (i.e., a series of type 0 files)

$<n>$ 2 bytes
: number of track chunks that follow the header chunk

$<division>$ 2 bytes
: unit of time for delta timing. If the value is positive, then it represents the units per beat. For example, +96 would mean 96 ticks per beat.

[17]
A track chunk consists of a literal identifier string, a length indicator specifying the size of the track, and actual event data making up the track.

$$track\_chunk = "MTrk" + <length> + <track\_event> [+ <track\_event> ...]$$

"$MTrk$" 4 bytes
: the literal string MTrk. This marks the beginning of a track.

$<length>$ 4 bytes
: the number of bytes in the track chunk following this number.

$<track\_event>$
: a sequenced track event.

A track event consists of a delta time since the last event, and one of three types of events.

$$track\_event = <v\_time> + <midi\_event> | <meta\_event> | <sysex\_event>$$

$<v\_time>$ a variable length value specifying the elapsed time (delta time) from the previous event to this event.

$<midi\_event>$ any MIDI channel message such as note-on or note-off. Running status is used in the same manner as it is used between MIDI devices.

$<meta\_event>$ an SMF meta event.

$<sysex\_event>$ an SMF system exclusive event.

I've used python's midi library [18] which provides high level representation of midi file structure. I found it easy to use. Here's a brief review of python-midi : A MIDI file is represented as a hierarchical set of objects. At the top is a Pattern, which contains a list of Tracks, and a Track is is a list of MIDI Events.

The MIDI Pattern class inherits from the standard python list, so it supports all list features such as append(), extend(), slicing, and iteration. Patterns also contain global MIDI meta-data: the resolution and MIDI Format.

The MIDI Track class also inherits from the standard python list. It does not have any special meta-data like Pattern, but it does provide a few helper functions to manipulate all events within a track.

There are 27 different MIDI Events supported. Three of the most common ones you'll see are :

1. The NoteOnEvent captures the start of note, like a piano player pushing down on a piano key. The tick is when this event occurred, the pitch is the note value of the key pressed, and the velocity represents how hard the key was pressed.

2. The NoteOffEvent captures the end of note, just like a piano player removing her finger from a depressed piano key. Once again, the tick is when this event occurred, the pitch is the note that is released, and the velocity has no real world analogy and is usually ignored. NoteOnEvents with a velocity of zero are equivalent to NoteOffEvents.

3. The EndOfTrackEvent is a special event, and is used to indicate to MIDI sequencing software when the song ends. With creating Patterns with multiple Tracks, you only need one EndOfTrack event for the entire song. Most MIDI software will refuse to load a MIDI file if it does not contain an EndOfTrack event.

The NoteOn and NoteOff events are inherited from the NoteEvent Class.

A tick represents the lowest level resolution of a MIDI track. Tempo is always analogous with Beats per Minute (BPM) which is the same thing as Quarter notes per Minute (QPM). The Resolution is also known as the Pulses per Quarter note (PPQ). It analogous to Ticks per Beat (TPM).

Tempo is set by two things. First, a saved MIDI file encodes an initial Resolution and Tempo. You use these values to initialize the sequencer timer(A sequencer is a device or a software which can play, record and edit midi files [19]). The Resolution should be considered static to a track, as well as the sequencer. During MIDI playback, the MIDI file may have encoded sequenced (that is, timed) Tempo change events. These events will modulate the Tempo at the time they specify. The Resolution, however, can not change from its initial value during playback.

Under the hood, MIDI represents Tempo in microseconds. In other words, you convert Tempo to Microseconds per Beat. If the Tempo was 120 BPM, the python code to convert to microseconds looks like this:

```
>>> 60 * 1000000 / 120
500000
```

This says the Tempo is 500,000 microseconds per beat. This, in combination with the Resolution, will allow you to convert ticks to time. If there are 500,000 microseconds per beat, and if the Resolution is 1,000 than one tick is how much time?

```
>>> 500000 / 1000
500
>>> 500 / 1000000.0
0.00050000000000000001
```

In other words, one tick represents .0005 seconds of time or half a millisecond. Increase the Resolution and this number gets smaller, the inverse as the Resolution gets smaller. Same for Tempo.

We know a Pattern is an array of Tracks and Tracks are arrays of Events. Each Track may represent different instruments.We can extract pitch information from NoteEvents using the method

```
pitch = noteevent.get_pitch()
```

I used the following code to extract pitches from midi tracks.

```
def pitches(track):
    pitches = []
    for i in track:
        if(isinstance(i, midi.NoteEvent)):
            pitches.append(i.get_pitch())
    return pitches
```

The above method takes a track as the input parameter and returns an array of pitches.You can then perform various mathematical operations on the array and write them back to events using the method

```
noteevent.set_pitch(pitch_value)
```

Here's a small example from the documentation on how to create a midi file from scratch.

```
import midi
# Instantiate a MIDI Pattern (contains a list of tracks)
```

```
pattern = midi.Pattern()
# Instantiate a MIDI Track (contains a list of MIDI events)
track = midi.Track()
# Append the track to the pattern
pattern.append(track)
# Instantiate a MIDI note on event, append it to the track
on = midi.NoteOnEvent(tick=0, velocity=20, pitch=midi.G_3)
track.append(on)
# Instantiate a MIDI note off event, append it to the track
off = midi.NoteOffEvent(tick=100, pitch=midi.G_3)
track.append(off)
# Add the end of track event, append it to the track
eot = midi.EndOfTrackEvent(tick=1)
track.append(eot)
# Print out the pattern
print pattern
# Save the pattern to disk
midi.write_midifile("example.mid", pattern)
```

Similar to the write method shown above, we could read a midi file from disk using

```
pattern = midi.read_midifile("filename")
```

It is important to know that pitches in a midi file are stored as midi numbers instead of actual frequencies.

3.3 Genetic Operations

Two main genetic operations are :

1. Crossover
2. Mutation

Crossover
: In genetic algorithms, crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. Cross over is a process of taking more than one parent solutions and producing a child solution from them. We'll be dealing entirely with mutations in this project[22]

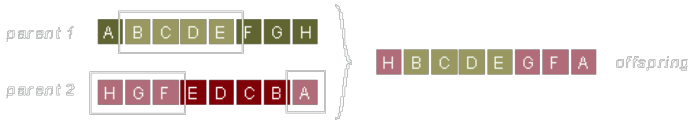

Figure 3.1: crossover

Mutation

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution.

The purpose of mutation in GAs is preserving and introducing diversity. Mutation should allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution. This reasoning also explains the fact that most GA systems avoid only taking the fittest of the population in generating the next but rather a random (or semi-random) selection with a weighting toward those that are fitter.[23]

Figure 3.2: mutation

For this project I am using python programming language. I've used PyGTK for the graphical User Interface. This application is designed to run on linux systems. I am using aubio python library with alsa audio for real time pitch detection using YIN. The input sample rate for recording is set at 44100Hz.

There are 4 important scripts in the program :

- teacher.py
- extract_bar.py
- pitch_detection_realtime.py
- genetic.py

teacher.py

This is the main script which is used to access the program. It contains the GUI code to interact with various functions in the application. It is also where we compare the user input to the midi file.

The following section of the code is used for the comparison :

```
x = [i for i, j in zip(comp_notes, user_notes) if i == j]
                boolean_value = set(x) == set(comp_notes)
```

With these notes we're effectively calculating a set difference between the notes in the midi and results after pitch detection. In other words we just make sure that notes in $midi$ are all present in the $user_note$. We don't care about any extra notes user played. Although we do care that the order in which the notes in midi is the same as order in user notes.

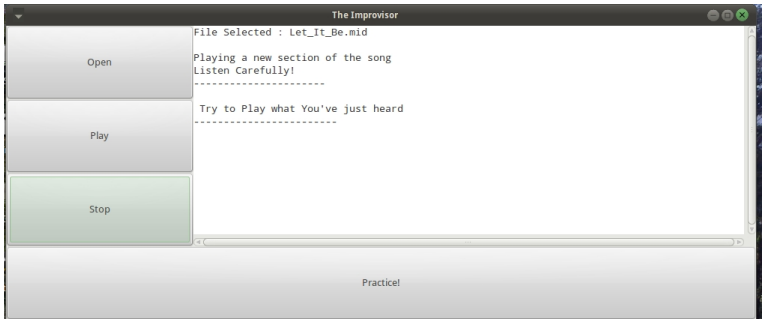

Figure 3.3: Improvisor Window

extract_bar

extracts a section of the midi file and stores it in another file

temp.mid

. We assume the time signature to be $\frac{4}{4}$. Thus formula for ticks per bar can be expressed as

$$ticksperbar = resolution \times 4 \tag{3.2}$$

The function takes two arguments a pattern which represents the midi file and start which is the position in the file from where we start extracting Our function looks like this :-

```
def extract_bar(start, pattern):
    pattern.make_ticks_abs()
    bar = pattern.resolution * 4
    track = pattern[0]

    new = midi.Pattern()
    track_new = midi.Track()

    for event in track:
```

```
        if(event.tick >= start and event.tick <= (start + bar) and isinstance(event, midi.
            tick_val = event.tick - start
            new_event = midi.Event.copy(event)
            new_event.tick = tick_val
            track_new.append(new_event)
        if(event.tick > (start + bar)):
            break

    if track_new:
        result = midi.Track()
        result.append(track_new[0])
        for x in xrange(len(track_new) - 1):
            ev = midi.Event.copy(track_new[x + 1])
            ev.tick = track_new[x + 1].tick - track_new[x].tick
            result.append(ev)

        result.append(midi.EndOfTrackEvent())
        new.append(result)

        midi.write_midifile("temp.mid", new)

    return event.tick
```

pitch_extract_realtime.py

As the name suggests this is used for pitch detection. We use single channel audio input with a sampling rate of 44100HZ. The frame size is 1024 bytes. Alsa Audio linux framework has been used to record the input. It's worth noting that **we don't concern ourselves with the octave notes. The formula to convert frequencies to notes is :**

$$midi = log_2\left(\frac{frequency}{440}\right) \times 12 + 69 \tag{3.3}$$

$$Note = midi \mod 12 \tag{3.4}$$

The energy of the note indicates its 'loudness' in order to reduce the effect of noise, I use a threshold energy value for note to be registered.

```
    import alsaaudio
import struct
from aubio.task import *
from math import *
```

```
import time

# constants
CHANNELS = 1
INFORMAT = alsaaudio.PCM_FORMAT_FLOAT_LE
RATE = 44100
FRAMESIZE = 1024
PITCHALG = aubio_pitch_yin
PITCHOUT = aubio_pitchm_freq

# set up audio input
recorder = alsaaudio.PCM(type=alsaaudio.PCM_CAPTURE)
recorder.setchannels(CHANNELS)
recorder.setrate(RATE)
recorder.setformat(INFORMAT)
recorder.setperiodsize(FRAMESIZE)

# set up pitch detect
detect = new_aubio_pitchdetection(FRAMESIZE, FRAMESIZE / 2, CHANNELS,
                                  RATE, PITCHALG, PITCHOUT)
buf = new_fvec(FRAMESIZE, CHANNELS)


def pitches():

    # main loop
    runflag = 1
    notes = []
    start = time.time()
    # maximum idle time allowed
    TIMEOUT = 4

    while runflag:

     # read data from audio input
        [length, data] = recorder.read()

        # convert to an array of floats
        floats = struct.unpack('f' * FRAMESIZE,data)

        # copy floats into structure
        for i in range(len(floats)):
            fvec_write_sample(buf, floats[i], 0, i)
```

```
        # find pitch of audio frame
        freq = aubio_pitchdetection(detect, buf)
        midi = int(round(log((freq / 440.0), 2) * 12 + 69))

        note = midi % 12
        # Find enery of the audio frame

        energy = vec_local_energy(buf)

        # Only print values if energy is greater than 1

        if(energy > 1):
            print "{:10.4f} {:10.4f}".format(note, energy)
            notes.append(note)
          start = time.time()

      else :
          if((time.time() - start) > TIMEOUT):

              break

    return notes

if __name__ == '__main__' :
print pitches()
```

I've used a timer in my code so that the recording stops if the user doesn't play anything for 4 seconds

`genetic.py`

This script applies the crossover operation on two lists.We only cross the among genes which are not correct. The probability of crossing over a gene depends on the number of times it occurs in the array in continuity. continuity indicates that a note is prominent . The note that occurs just once has zero probability for crossover. We ask the users to play the note slowly and carefully thus we assume that if the note occurs abruptly in one frame it is most likely an error or a passing note.

Below I'll illustrate the algorithm on which Improvisor operates.

1. **User Selects a Midi File an listens to it**
2. **Computer Extract a small section of the file and plays it to the user. It's 'fitness' being maximum. We push this value onto a stack**

3. **User tries to play the section corresponding to the value on the stack.If he gets it wrong**
    - **System performs crossover between user Input and true value of section. It modifies half the wrong notes user played. Push the fitness of the 'child' onto the stack**
    - **Repeat step 3**
4. **If user gets it right. Pop the stack, play the section corresponding to the stack value and go to step 3. If the stack is empty go to the next section of the song.**

4 PERFORMANCE

The performance bottle neck for Improvisor is the pitch detection algorithm. I experimented with fft based pitch detection and correlation based pitch detection algorithms. For fft I used code from an open source guitar tuner by Bjorn Roche[26]. The convergence was really slow about 2 secs for every note. Where as yin converges in a fraction of a second. Considering we're dealing with real time input convergence rate is a big factor in choice of algorithm.

REFERENCES

[1] **Lambert, M. Surhone (2010). "Overview". Computer Audition. ISBNhttps://88df6ea0630aed8027ff-0caf779119a6537399728d4d80523795.ssl.cf5.rackcdn.com/mvvtrwg 9786131357299.**

[2] **Benade, Arthur H. (1960). Horns, Strings, and Harmony.Science Study Series.**

[3] **Boretz, Benjamin (1995). Meta-Variations: Studies in the Foundations of Musical Thought.**

[4] **Cavanagh, Lynn ([1999]). "A Brief History of the Establishment of International Standard Pitch A=440 Hertz"**

[5] **Jackendoff, Ray and Fred Lerdahl (1981). "Generative Music Theory and Its Relation to Psychology."**

[6] **Gorow 2002, 212.**

[7] **Al Biles, Genjam 1993,**

[8] **Al Biles , Autonomous Eliminating the fitness bottleneck by eliminating the fitness**

[9] **D. Gerhard. Pitch Extraction and Fundamental Frequency: History and Current Techniques**

[10] **Paul Brossier, Automatic annotation of musical audio for interactive systems**

[11] Hayes, Monson (1996). Statistical Digital Signal Processing and Modeling

[12] A. Michael Noll, âĂIJCepstrum Pitch Determination

[13] Stephen A. Zahorian and Hongbing Hu. A Spectral/temporal method for Robust Fundamental Frequency Tracking

[14] FranÃğois-Michel De Rainville, FÃľlix-Antoine Fortin, Marc-AndrÃľ Gardner, Marc Parizeau and Christian GagnÃľ, "DEAP – Enabling Nimbler Evolutions"

[15] Swift, Andrew. (May-Jun 1997.), "A brief Introduction to MIDI", SURPRISE (Imperial College of Science Technology and Medicine), retrieved 22 August 2012

[16] http://faydoc.tripod.com/formats/mid.htm

[17] http://www.ccarh.org/courses/253/handout/smf/

[18] Giles Hall, "python-midi (An open source python library for managing midi files"

[19] www.wikipedia.com

[20] Mitchell 1996

[21] Whitley 1994

[22] `http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html`

[23] `CrossoverandMutation".http://www.obitko.com/:MarekObitko`

[24] A "Hello World!" Genetic Algorithm Example, James Matthews

[25] Kennedy, J.; Eberhart, R. (1995)

[26] http://blog.bjornroche.com/2012/07/frequency-detection-using-fft-aka-pitch.html

[27] `www.amazon.com`

[28] `www.rockprodigy.com`

[29] A. de CheveignÃľ and H. Kawahara. YIN, a fundamental frequency estimator for speech and music

[30] Aitken, Alexander Craig (1957) Statistical Mathematics