Monophonic Instrument Playing Practice System Prototype

Mika Haarahiltunen

Bachelor's Thesis
December 2013

Degree Programme in Software Engineering
School of Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author Haarahiltunen, Mika	Type of publication Bachelor´s Thesis	Date 02.12.2013
	Pages 42	Language English
	Confidential ( ) Until	Permission for web publication (X)
Title Monophonic Instrument Playing Practice System Prototype		
Degree Programme Software Engineering		
Tutor Kotkansalo, Jouko		
Assigned by Huotari, Jouni		

Abstract

The goal of the project was to develop a prototype of a system that could be used for practicing the playing of breath instruments, such as flutes or saxophones. However, during the development process, the emphasis shifted from a breath instrument specific design towards a more generic design that could be used with any monophonic instrument. The development of the prototype included studying existing solutions, specifying the functional requirements and finally implementing the prototype. The idea was that the system could play and visualize the contents of Standard MIDI Files, the user would play along using some monophonic instrument and the system would tell in real time whether the user was playing correctly.

After the initial requirement analysis phase, roughly half of the time was spent on MIDI playback and visualization and the other half on pitch detection. A significant portion of the time was spent on studying these subjects, and as it turned out, implementing the planned features was far from trivial. The seemingly simple MIDI playback and visualization functionality required precise synchronization between multiple threads and introduced some unexpected problems, for example, the official Standard MIDI File specification was not freely available. The pitch detection functionality relied on somewhat complicated mathematics, suffered from latency and accuracy issues and some of the related algorithms were computationally very expensive. The real-time nature of the system required accurate audio and visualization synchronization and minimal pitch detection latency.

The thesis explains the theory and technical details of the implemented functionality and the problems that were encountered during the development. It also describes the shortcomings of the implementation and suggests some possible solutions for them.

Keywords
MIDI, signal processing, pitch detection

Miscellaneous

Contents

Figures

Tables

1 Overview

The goal of the project was to implement a prototype of a system that could be used as an aid for practicing monophonic breath instrument playing. Originally the system was intended especially for practicing the breath and mouth control techniques required for playing a saxophone. The original requirements were not fixed but included breath and mouth control practice feedback, recording the user input for further analysis and providing the user at least some form of rhythm practice functionality.

The first phase was finding out whether any existing systems implemented identical or similar functionality. No existing systems were found that were meant specifically for breath instruments or for practicing the breath and mouth control techniques. Some existing systems, e.g., SingStar game series for PlayStation consoles, did implement rhythm and pitch detection functionality but these systems were either not compatible with breath instruments or were too game oriented to be suitable for serious practicing.

The second phase was comparing the pros and cons of different input devices that could be used for translating user actions, such as breath, to a software friendly format. Using spirometers, MIDI breath controllers, MIDI saxophones, microphones and combinations of these were considered. During this process the emphasis shifted from hardware centric breath and mouth control analysis to more general and software centric rhythm and pitch detection. In the end, specialized hardware solutions were abandoned in favor of a microphone based solution. This was due to the wide availability and cheap price of microphones compared to the other options and due to the fact that mobile devices have built-in microphones.

The third phase was refining the application requirements. We decided on implementing two major features. The first was that the application should be able to play and visualize the contents of Standard MIDI Files. The second was that the microphone signal should be analyzed and compared in real-time to the MIDI data being played. Implementing some type of a scoring system as in SingStar or Guitar Hero game series was discussed but was not implemented in the prototype.

The fourth phase was developing the prototype. The prototype was implemented using the C++ programming language and Qt and Win32 software libraries. The Qt library was used for implementing the user interface and some of its functionality was also used in the back-end implementation. The Win32 library was used for MIDI playback and for recording microphone input. The Standard MIDI File parser, MIDI data visualization and pitch detection functionality was implemented from scratch.

The technical details of the Standard MIDI File format are described in detail in chapter 2. Signal processing algorithms for pitch detection and their practical limitations are described in chapter 3. The problems encountered in implementing the prototype and some implementation details are discussed in chapter 4. Chapter 5 discusses missing features and shortcomings of the prototype and possible solutions to some of these problems.

2 Standard MIDI File Format

2.1 Introduction

MIDI stands for Musical Instrument Digital Interface and was originally developed as a communication protocol for controlling various electronic instruments. The MIDI protocol was standardized in 1982 when the MIDI 1.0 specification was released. The MIDI specification is maintained by MIDI Manufacturers Association (MMA) which has made many enhancements and updates to the specification since its initial release. Even though the specification has changed over the years, it is still called "MIDI 1.0". (MIDI 2013.)

Despite common belief, MIDI is not an audio format. Even though the MIDI protocol was originally developed specifically for musical instruments, it can also be used for controlling other types of devices such as stage lighting and firework launchers, or a combination of these. A single MIDI stream can contain up to sixteen event channels that can be routed to the same or different devices. The MIDI protocol is a way for communicating timed events and is suitable for controlling and synchronizing the use of almost all types of digital devices, including computer software. (MIDI 2013.)

The Standard MIDI File (SMF) format is a binary file format that provides a standardized way of storing MIDI event sequences. The MIDI format is widely supported by musical arrangement software. On platforms that use file extensions, .mid is commonly used as the extension for these files. (MIDI 2013.)

The official Standard MIDI File specification was available only as a printed copy that could be ordered from the MIDI Manufacturers Association (MMA) website for a fee. Because the official specification was not freely available, the presented information is based on unofficial documentation. The presented information may be inaccurate or incomplete and use terms that differ from the terms used in the official specification. However, the presented information was used for implementing a working Standard MIDI File parser so it should be mostly correct and complete.

The Standard MIDI File format was originally designed to be used with very limited hardware. On early MIDI hardware, storage capacity and especially MIDI bandwidth was very limited. The file format uses various techniques such as variable-length values and running status for MIDI events and divided system exclusive events for reducing storage space and MIDI bandwidth requirements at the expense of added complexity. (MIDI 2013.)

It seems that some Standard MIDI File features, such as divided controller change MIDI events, were added as hacks into an existing specification that was not originally designed to support them. Most features that complicate the Standard MIDI File format, such as variable-length fields and MIDI event running status, are designed for reducing MIDI bandwidth usage. The divided controller change events actually increase the MIDI bandwidth usage instead of reducing it.

2.2 Data Types

Standard MIDI files store data in binary format as single-byte, multi-byte and variable-length values. A single-byte value contains 8 bits. Multi-byte values are stored using big-endian (most significant byte first) byte ordering. The length of multi-byte values is always either fixed (e.g., 16, 24 or 32 bits) or explicitly specified with a length field before the multi-byte value (e.g., ASCII string lengths are stored before the character buffer). Variable-length values are stored using a simple encoding scheme: If a byte in a variable-length value has its most significant bit set to 1, another byte follows. The last byte in a variable-length value has its most significant bit set to 0. In other words, each byte stores a flag bit and seven data bits. A variable-length value may use at most 4 bytes, i.e., can contain at most 28 data bits (4 bytes, 7 data bits per byte), and therefore the maximum representable value is 2^28-1 for an unsigned variable-length integer. The table below gives some examples: (MIDI File Format 2013.)

Table 1. Variable-length value examples

Value	**Variable-length**
00000000 00000000 00000000 00000000	[0]0000000
00000000 00000000 00000000 01000001	[0]1000001
00000000 00000000 00000000 11000001	[1]0000001 [0]1000001
00000000 00000000 10000000 11000001	[1]0000010 [1]0000001 [0]1000001
00001111 11111111 11111111 11111111	[1]1111111 [1]1111111 [1]1111111 [0]1111111

In practice, this type of encoding scheme allows the size of an average MIDI file to shrink to about 60% of the size of an otherwise identical file which uses 32-bit values instead of variable-length values. This is because a MIDI file consists mostly of MIDI events which consist of a variable-length delta time value and 3 data bytes. The delta time values are usually small and fit in a single-byte variable-length value, thus most MIDI events end up taking 4 bytes instead of 7 which would be the case if delta time values were stored as 32-bit values. In the late 1980s when the Standard MIDI File specification was written, this saving was significant given the limited amount of storage space on computers and floppy disks and especially the very limited MIDI bandwidth on older MIDI devices. (MIDI 2013.)

Some MIDI events use split 14-bit values which are formed from two bytes: The 7 least significant bits of one byte (LSB) define the 7 least significant bits and the 7 least significant bits of another byte (MSB) define the 7 most significant bits. The most significant bit of these two bytes is always set to 0. The following table visualizes how a split 14-bit value is formed from the bytes: (MIDI File Format 2013.)

Table 2. Split 14-bit value example

MSB	**LSB**	**14-bit Value**
0XXXXXXX	0YYYYYYY	XXXXXXXYYYYYYY

2.3 Data Chunks

Standard MIDI files always contain a header chunk and one or more track chunks. The

MIDI file format is extensible and can also contain custom data chunks. The MIDI specification states that MIDI file parsers must ignore any unknown data chunks without producing errors. Each chunk contains a 32-bit ID field (4 ASCII characters, e.g., "MTrk"), a 32-bit data block size field and a data block whose size is specified with the data block size field. Since all chunks follow this format, skipping any unknown chunks is trivial: Read the ID and data block size fields and skip the number of bytes specified by the data block size field if the chunk ID is not recognized. (MIDI File Format 2013.)

2.3.1 Header Chunk

The following table illustrates the header chunk layout:

Table 3. Header chunk layout

Field	**Type**	**Size**	**Value**
ID	4 8-bit characters	4	”MThd”
Data block size	32-bit unsigned integer	4	6
Type	16-bit unsigned integer	2	0-2
Track count	16-bit unsigned integer	2	1-65535
Time division	16-bit unsigned integer	2	See following text

The header chunk contains information about the entire song. A valid MIDI file contains only one header chunk and it always comes first. The header chunk identified is "MThd" and the data block size is always 6. (MIDI File Format 2013.)

Valid values for the type field are 0, 1 and 2. Type 0 MIDI files contain one track chunk which contains all MIDI events for all channels, including time signature and tempo events. Type 1 MIDI files contain two or more synchronized track chunks where the first contains information common to all tracks, such as song title, time signature and tempo events. The other tracks contain only track specific data. Type 2 MIDI files contain multiple tracks that are not meant to be played simultaneously, e.g., drum patterns or other sequences. Type 0 and 1 MIDI files can also be used for the same purpose, making type 2 MIDI files somewhat less useful. Type 2 MIDI files are intended as a minor storage space optimization where the header chunk does not need to be duplicated for all tracks, as is the case if they were stored in separate type 0 or type 1 MIDI files. (MIDI File Format 2013.)

The track count field specifies the number of track chunks in the MIDI file. This should be 1 for type 0 MIDI files, between [2, 65535] for type 1 MIDI files and between [1, 65535] for type 2 MIDI files. (MIDI File Format 2013.)

The time division field specifies the time division for all tracks in the MIDI file. The time division affects how MIDI event delta time values are decoded into real time values. If the most significant bit is set to 0, the following 15 bits describe the time division in ticks per quarter note. If the most significant bit is set to 1, the following 7 bits specify a standard SMPTE (Society of Motion Picture & Television Engineers) frames per second value where valid values are 24, 25, 29 (interpreted as 29.97) and 30 frames per second. The remaining 8 bits specify the number of ticks per frame. If the time division value is 0x9978 (binary 10011001 01111000), the most significant bit is set to 1, the following 7 bits (binary 0011001, decimal 25) specify the number of frames per second and the remaining 8 bits (binary 01111000, decimal 120) specify the number of ticks per frame which results in 25 frames per second, 120 ticks per frame and 3000 (25 * 120) ticks per second. The time division is usually specified as ticks per quarter note but specifying the time division using frames per second and ticks per frame values might be useful if the MIDI events need to be synchronized with a display device update frequency. (MIDI File Format 2013.)

2.3.2 Track Chunk

The following table illustrates the track chunk layout:

Table 4. Track chunk layout

Field	**Type**	**Size**	**Value**
ID	4 8-bit characters	4	”MTrk”
Data block size	32-bit unsigned integer	4	See following text
Event buffer	See following text		

Track chunks contain event data for one or more event channels. The minimum and maximum number of track chunks in a valid MIDI file depend on the MIDI file type as specified in the header chunk. Track chunk identifier is "MTrk" and the data block size depends on the number and type of events in the event buffer. (MIDI File Format 2013.)

The event buffer contains all events for the track. There are no padding bytes before, between or after any event structures. Events are stored in the event buffer in the order they are intended to be executed. Different event types are described in detail in following sections. (MIDI File Format 2013.)

2.4 Events

The following table illustrates the event layout:

Table 5. Event layout

Field	**Type**	**Size**	**Value**
Delta time	Variable-length	1-4	See following text
Type	8-bit unsigned integer	1	See following text
Event data	See following text		

All event structures begin with a variable-length delta time value. The delta time values are specified in ticks since the previous event in the same track (MIDI File Format 2013).

The variable-length delta time value is followed by an 8-bit type field that defines the event type. There are three types of events: MIDI events, meta events and system exclusive events. If the most significant bit is set to 0, the event is a MIDI event and the following 7 bits define the actual MIDI event type and event channel. The type field is 0xFF for meta events and 0xF0 for system exclusive events. (MIDI File Format 2013.)

The type field is followed by event specific data whose length and contents depend on the event type (MIDI File Format 2013).

2.4.1 MIDI Events

The following table illustrates the MIDI event layout:

Table 6. MIDI event layout

Field	Type	Size	Value
Delta time	Variable-length	1-4	See following text
Type	8-bit unsigned integer	1	See following text
Parameter 1	8-bit unsigned integer	1	See following text
Parameter 2	8-bit unsigned integer	1	See following text

The variable-length delta time value is interpreted as ticks since the previous event in the same track for all event types (MIDI File Format 2013).

For MIDI events, the type field actually contains two 4-bit values: The 4 most significant bits define the MIDI event type and the 4 least significant bits define the event channel (0-15). (MIDI File Format 2013.)

If the most significant bit of the type field is 0, running status should be used. This means that the variable-length delta time value is followed by the parameter bytes and the type field is not present. In this case, the MIDI event type is taken from the previous MIDI event. Identifying the use of running status by checking the most significant bit is possible because all valid event type values have their most significant bit set to 1 and all valid MIDI event parameter values have their most significant bit set to 0. (MIDI File Format 2013.)

The parameter bytes are interpreted differently for different MIDI event types but valid parameter values are between [0, 127] for all MIDI event types. The following table lists the different MIDI event types and how the parameter bytes should be interpreted: (MIDI File Format 2013.)

Table 7. MIDI event types

Event Type	Value	Parameter 1	Parameter 2
Note Off	0x8	Note Number	Velocity
Note On	0x9	Note Number	Velocity
Note Pressure Change	0xA	Note Number	Pressure
Controller Change	0xB	Controller Number	Controller Value
Program Change	0xC	Program Number	-
Channel Pressure Change	0xD	Pressure	-
Pitch Change	0xE	Pitch Value (LSB)	Pitch Value (MSB)

Various sources used slightly different names for the event types and parameters (e.g., some sources used "pitch bend" instead of "pitch change" and "key" instead of "note number") and the official documentation was not freely available for referencing (one would have to buy a printed copy) so the event type names used here may not match with the ones used in the official documentation. Despite this, the event type names used here should still describe the functionality accurately.

The note number values used by Note Off, Note On and Note Pressure Change MIDI events can be converted to frequencies and back with the following equations where f is frequency and n is note number:

$$f = 440 \cdot 2^{(n-69)/12} \tag{1}$$

$$n = 69 + 12 \cdot \log_2\left(\frac{f}{440}\right) \tag{2}$$

Different sources use different octave number conventions for MIDI note numbers. The following table lists all MIDI note numbers and their corresponding note names, octaves and frequencies for standard tuning where MIDI note number 69 is A4 at 440Hz:

Table 8. MIDI note numbers, names and frequencies

#	Name	Frequency
0	C-1	8.176
1	C#/Db-1	8.662
2	D-1	9.177
3	D#/Eb-1	9.723
4	E-1	10.301
5	F-1	10.913
6	F#/Gb-1	11.562
7	G-1	12.250
8	G#/Ab-1	12.978
9	A-1	13.750
10	A#/Bb-1	14.568
11	B-1	15.434
12	C0	16.352
13	C#/Db0	17.324
14	D0	18.354
15	D#/Eb0	19.445
16	E0	20.602
17	F0	21.827
18	F#/Gb0	23.125
19	G0	24.500
20	G#/Ab0	25.957
21	A0	27.500
22	A#/Bb0	29.135
23	B0	30.868
24	C1	32.703
25	C#/Db1	34.648
26	D1	36.708
27	D#/Eb1	38.891
28	E1	41.203
29	F1	43.654
30	F#/Gb1	46.249
31	G1	48.999
32	G#/Ab1	51.913
33	A1	55.000
34	A#/Bb1	58.270
35	B1	61.735
36	C2	65.406
37	C#/Db2	69.296
38	D2	73.416
39	D#/Eb2	77.782
40	E2	82.407
41	F2	87.307
42	F#/Gb2	92.499
43	G2	97.999
44	G#/Ab2	103.826
45	A2	110.000
46	A#/Bb2	116.541
47	B2	123.471
48	C3	130.813
49	C#/Db3	138.591
50	D3	146.832
51	D#/Eb3	155.563
52	E3	164.814
53	F3	174.614
54	F#/Gb3	184.997
55	G3	195.998
56	G#/Ab3	207.652
57	A3	220.000
58	A#/Bb3	233.082
59	B3	246.942
60	C4	261.626
61	C#/Db4	277.183
62	D4	293.665
63	D#/Eb4	311.127
64	E4	329.628
65	F4	349.228
66	F#/Gb4	369.994
67	G4	391.995
68	G#/Ab4	415.305
69	A4	440.000
70	A#/Bb4	466.164
71	B4	493.883
72	C5	523.251
73	C#/Db5	554.365
74	D5	587.330
75	D#/Eb5	622.254
76	E5	659.255
77	F5	698.456
78	F#/Gb5	739.989
79	G5	783.991
80	G#/Ab5	830.609
81	A5	880.000
82	A#/Bb5	932.328
83	B5	987.767
84	C6	1046.502
85	C#/Db6	1108.731
86	D6	1174.659
87	D#/Eb6	1244.508
88	E6	1318.510
89	F6	1396.913
90	F#/Gb6	1479.978
91	G6	1567.982
92	G#/Ab6	1661.219
93	A6	1760.000
94	A#/Bb6	1864.655
95	B6	1975.533
96	C7	2093.005
97	C#/Db7	2217.461
98	D7	2349.318
99	D#/Eb7	2489.016
100	E7	2637.020
101	F7	2793.826
102	F#/Gb7	2959.955
103	G7	3135.963
104	G#/Ab7	3322.438
105	A7	3520.000
106	A#/Bb7	3729.310
107	B7	3951.066
108	C8	4186.009
109	C#/Db8	4434.922
110	D8	4698.636
111	D#/Eb8	4978.032
112	E8	5274.041
113	F8	5587.652
114	F#/Gb8	5919.911
115	G8	6271.927
116	G#/Ab8	6644.875
117	A8	7040.000
118	A#/Bb8	7458.620
119	B8	7902.133
120	C9	8372.018
121	C#/Db9	8869.844
122	D9	9397.273
123	D#/Eb9	9956.063
124	E9	10548.082
125	F9	11175.303
126	F#/Gb9	11839.822
127	G9	12543.854

Note Off MIDI event stops playback of a specified note in the specified event channel, as if a keyboard key was released. Valid values for the note number are between [0, 127] and should match a currently playing note. The velocity, between [0, 127], defines how fast the note was released. If there has not been a matching Note On event for the specified note, this event is ignored. (MIDI File Format 2013.)

Note On MIDI event starts playback of a specified note in the specified channel, as if a keyboard key was pressed. Valid values for the note number are between [0, 127]. The velocity, between [0, 127], defines how fast the note was pressed. Note that a Note On MIDI event with zero velocity should be interpreted as a Note Off MIDI event, which can be used with running status to significantly reduce MIDI bandwidth requirements. (MIDI File Format 2013.)

Note Pressure Change MIDI event indicates a pressure change on a currently playing note in the specified channel. Valid values for the note number are between [0, 127] and should match a currently playing note. The pressure value, between [0, 127], specifies the pressure being applied (0 for no pressure, 127 for full pressure). This can be used to express changes in the note volume, like when a saxophonist changes the applied air pressure when playing a note. (MIDI File Format 2013.)

Controller Change MIDI event changes the specified controller value for the specified channel. There are 128 controller values which affect different attributes such as volume, pan and more. The controller number, between [0, 127], defines which controller value is changing and the controller value, between [0, 127], defines the new value. Note that some attributes are given as split 14-bit values whose most and least significant bytes come from two different controller values. The defined MIDI controllers are listed in the following table: (MIDI File Format 2013.)

Table 9. MIDI controller numbers and descriptions

#	Description
0	Bank Select
1	Modulation
2	Breath Controller
4	Foot Controller
5	Portamento Time
6	Data Entry (MSB)
7	Main Volume
8	Balance
10	Pan
11	Expression Controller
12	Effect Control 1
13	Effect Control 2
16-19	General-Purpose Controllers 1-4
32-63	LSB for controllers 0-31
64	Damper pedal (sustain)
65	Portamento
66	Sostenuto
67	Soft Pedal
68	Legato Footswitch
69	Hold 2
70	Sound Controller 1 (default: Timber Variation)
71	Sound Controller 2 (default: Timber/Harmonic Content)
72	Sound Controller 3 (default: Release Time)
73	Sound Controller 4 (default: Attack Time)
74-79	Sound Controller 6-10
80-83	General-Purpose Controllers 5-8
84	Portamento Control
91	Effects 1 Depth (formerly External Effects Depth)
92	Effects 2 Depth (formerly Tremolo Depth)
93	Effects 3 Depth (formerly Chorus Depth)
94	Effects 4 Depth (formerly Celeste Detune)
95	Effects 5 Depth (formerly Phaser Depth)
96	Data Increment
97	Data Decrement
98	Non-Registered Parameter Number (LSB)
99	Non-Registered Parameter Number (MSB)
100	Registered Parameter Number (LSB)
101	Registered Parameter Number (MSB)
121-127	Mode Messages

Program Change MIDI event changes the current program for the specified channel. The program number, between [0, 127], defines an instrument in the current program bank. If a device supports multiple program banks, the current program bank can be changed with a bank select Controller Change MIDI event. Note that the second parameter data byte exists in the MIDI file but is not used for anything and contains an undefined value. The program numbers and their corresponding instrument names available in the default program bank as described in Wikipedia (General MIDI 2013) are listed in the following table: (MIDI File Format 2013.)

Table 10. MIDI program numbers and instrument names

#	Instrument Name
0	Acoustic Grand Piano
1	Bright Acoustic Piano
2	Electric Grand Piano
3	Honky-tonk Piano
4	Electric Piano 1
5	Electric Piano 2
6	Harpsichord
7	Clavinet
8	Celesta
9	Glockenspiel
10	Music Box
11	Vibraphone
12	Marimba
13	Xylophone
14	Tubular Bells
15	Dulcimer
16	Drawbar Organ
17	Percussive Organ
18	Rock Organ
19	Church Organ
20	Reed Organ
21	Accordion
22	Harmonica
23	Tango Accordion
24	Acoustic Guitar (nylon)
25	Acoustic Guitar (steel)
26	Electric Guitar (jazz)
27	Electric Guitar (clean)
28	Electric Guitar (muted)
29	Overdriven Guitar
30	Distortion Guitar
31	Guitar Harmonics
32	Acoustic Bass
33	Electric Bass (finger)
34	Electric Bass (pick)
35	Fretless Bass
36	Slap Bass 1
37	Slap Bass 2
38	Synth Bass 1
39	Synth Bass 2
40	Violin
41	Viola
42	Cello
43	Contrabass
44	Tremolo Strings
45	Pizzicato Strings
46	Orchestral Harp
47	Timpani
48	String Ensemble 1
49	String Ensemble 2
50	Synth Strings 1
51	Synth Strings 2
52	Choir Aahs
53	Voice Oohs
54	Synth Choir
55	Orchestra Hit
56	Trumpet
57	Trombone
58	Tuba
59	Muted Trumpet
60	French Horn
61	Brass Section
62	Synth Brass 1
63	Synth Brass 2
64	Soprano Sax
65	Alto Sax
66	Tenor Sax
67	Baritone Sax
68	Oboe
69	English Horn
70	Bassoon
71	Clarinet
72	Piccolo
73	Flute
74	Recorder
75	Pan Flute
76	Blown bottle
77	Shakuhachi
78	Whistle
79	Ocarina
80	Lead 1 (square)
81	Lead 2 (sawtooth)
82	Lead 3 (calliope)
83	Lead 4 (chiff)
84	Lead 5 (charang)
85	Lead 6 (voice)
86	Lead 7 (fifths)
87	Lead 8 (bass + lead)
88	Pad 1 (new age)
89	Pad 2 (warm)
90	Pad 3 (polysynth)
91	Pad 4 (choir)
92	Pad 5 (bowed)
93	Pad 6 (metallic)
94	Pad 7 (halo)
95	Pad 8 (sweep)
96	FX 1 (rain)
97	FX 2 (soundtrack)
98	FX 3 (crystal)
99	FX 4 (atmosphere)
100	FX 5 (brightness)
101	FX 6 (goblins)
102	FX 7 (echoes)
103	FX 8 (sci-fi)
104	Sitar
105	Banjo
106	Shamisen
107	Koto
108	Kalimba
109	Bagpipe
110	Fiddle
111	Shanai
112	Tinkle Bell
113	Agogo
114	Steel Drums
115	Woodblock
116	Taiko Drum
117	Melodic Tom
118	Synth Drum
119	Reverse Cymbal
120	Guitar Fret Noise
121	Breath Noise
122	Seashore
123	Bird Tweet
124	Telephone Ring
125	Helicopter
126	Applause
127	Gunshot

Channel Pressure Change MIDI event indicates a pressure change on all currently playing notes in the specified channel. The pressure value, between [0, 127], specifies the pressure being applied (0 for no pressure, 127 for full pressure), just like in Note Pressure Change MIDI events. Note that the second parameter data byte exists in the MIDI file but is not used for anything and contains an undefined value. (MIDI File Format 2013.)

Pitch Change MIDI event changes the pitch of all currently playing notes in the specified channel. Legal values for the parameter bytes are between [0, 127]. The parameter bytes form a split 14-bit value that can be calculated as described in the data types section. The 14-bit value is then normalized between [-1, 1] so that 0 becomes -1, 2^13 becomes 0 and 2^14-1 becomes 1. Values below 0 decrease the pitch and values above 0 increase the pitch. The actual pitch change range is device dependent, making this event somewhat non-portable, but is usually +/-2 semi-tones. (MIDI File Format 2013.)

2.4.2 Meta Events

The following table illustrates the meta event layout:

Table 11. Meta event layout

Field	Type	Size	Value
Delta time	Variable-length	1-4	See following text
Type	8-bit unsigned integer	1	255
Meta event type	8-bit unsigned integer	1	0-255
Length	Variable-length	1-4	See following text
Data	See following text		

The variable-length delta time value is interpreted as ticks since the previous event in the same track for all event types (MIDI File Format 2013).

For meta events, the type field value is always 255 which can be used to identify the event as a meta event. The actual type of the meta event is specified with the meta event type field value. Valid values for this field are between [0, 255]. (MIDI File

Format 2013.)

The variable-length length field describes the event data size in bytes. The event data contents depend on the meta event type. (MIDI File Format 2013.)

The Standard MIDI File specification defines fifteen meta event types. The standard meta event types and their IDs (meta event type field values) are listed in the following table: (MIDI File Format 2013.)

Table 12. Meta event types

ID	Meta Event Type
0	Pattern/Sequence Number
1	Text
2	Copyright Notice
3	Sequence/Track Name
4	Instrument Name
5	Lyrics
6	Marker
7	Cue Point
32	MIDI Channel Prefix
47	End Of Track
81	Set Tempo
84	SMPTE Offset
88	Time Signature
89	Key Signature
127	Sequencer Specific

Pattern/Sequence Number meta event defines the pattern number of a type 2 MIDI file or the sequence number of type 0 or 1 MIDI file. The length field value is always 2 and the event data contains a 16-bit big-endian unsigned integer. The delta time field value should always be 0 and this event should come before any MIDI events and non-zero delta time events. (MIDI File Format 2013.)

Text meta event defines a string that can be used for any reason such as comments. The length field value describes the string length in bytes and the event data contains the string. The string is usually ASCII encoded and may or may not contain a terminating NUL-character. (MIDI File Format 2013.)

Copyright Notice meta event defines a copyright information string. The length field value describes the string length in bytes and the event data contains the string. The string is usually ASCII encoded and may or may not contain a terminating NUL-character. The string format is usually "© year author", e.g., "© 1994 Nobuo Uematsu". The delta time field value should always be 0 and this event should be in the first track and come before any MIDI events and non-zero delta time events. (MIDI File Format 2013.)

Sequence/Track Name meta event defines the sequence name string when in a type 0 or type 2 MIDI file or in the first track of a type 1 MIDI file. When this meta event appears in any track after the first in a type 1 MIDI file, it defines the track name string. The length field value describes the string length in bytes and the event data contains the string. The string is usually ASCII encoded and may or may not contain a terminating NUL-character. The delta time field value should always be 0 and this event should come before any MIDI events and non-zero delta time events. (MIDI File Format 2013.)

Instrument name meta event defines the instrument name string for a channel specified with a previous MIDI Channel Prefix meta event. The length field value describes the string length in bytes and the event data contains the string. The string is usually ASCII encoded and may or may not contain a terminating NUL-character. (MIDI File Format 2013.)

Lyrics meta event defines a lyrics string. The length field value describes the string length in bytes and the event data contains the string. The string is usually ASCII encoded and may or may not contain a terminating NUL-character. These meta events can be used for implementing a karaoke-style system. (MIDI File Format 2013.)

Marker meta event defines a description string for a significant point in the sequence, e.g., the beginning of a new verse or chorus. The length field value describes the string length in bytes and the event data contains the string. The string is usually ASCII encoded and may or may not contain a terminating NUL-character. These events are usually in the first track but may appear in any track. (MIDI File Format

2013.)

Cue Point meta event defines a description string for a manually triggered action, e.g., the curtain call at the end of a performance. The length field value describes the string length in bytes and the event data contains the string. The string is usually ASCII encoded and may or may not contain a terminating NUL-character. These events are usually in the first track but may appear in any track. (MIDI File Format 2013.)

MIDI Channel Prefix meta event defines the MIDI channel for following meta events, such as the Instrument Name meta event. The effect of this event is terminated by another MIDI Channel Prefix meta event or any non-meta event. The length field is always 1 and the event data contains an 8-bit unsigned integer that specifies the MIDI channel index between [0, 15]. (MIDI File Format 2013.)

End Of Track meta event signals the end of a track chunk. This event must always appear as the last event of a track chunk. The length field value is always 0 and there is no event data. (MIDI File Format 2013.)

Set Tempo meta event defines the sequence tempo as microseconds per quarter note. If no Set Tempo meta event has been encountered, the default tempo is 500000 microseconds per quarter note (120 beats per minute). The length field value is always 3 and the event data contains a 24-bit big-endian unsigned integer that describes the tempo as microseconds per quarter note. Tempo can be converted from beats per minute (BPM) to microseconds per quarter note (MPQN) and back using the following equations where the constant 60,000,000 is the number of microseconds per minute: (MIDI File Format 2013.)

$$BPM = \frac{60{,}000{,}000}{MPQN} \tag{3}$$

$$MPQN = \frac{60{,}000{,}000}{BPM} \tag{4}$$

SMPTE Offset meta event defines the SMPTE starting point offset from the beginning of the track. It is defined in terms of hours, minutes, seconds, frames and sub-frames. There are always 100 sub-frames per frame regardless of what sub-division was specified in the header chunk. The length field value is always 5 and the event data contains bytes for frame rate/hour offset, minute offset, second offset, frame offset and sub-frame offset, in that order. (MIDI File Format 2013.)

The two most significant bits of the first data byte define the frame rate as frames per second. The following table lists the bit combinations and their corresponding frame rates: (MIDI File Format 2013.)

Table 13. SMPTE offset meta event frame rates

Bits	Frame Rate
00	24
01	25
10	29.97
11	30

The 6 least significant bits of the first data byte define the hour offset as an unsigned integer between [0, 23]. The second data byte defines the minute offset as an 8-bit unsigned integer between [0, 59]. The third data byte defines the second offset as an 8-bit unsigned integer between [0, 59]. The fourth data byte defines the frame offset as an 8-bit unsigned integer. The range of valid values for the frame offset depends on the frame rate, i.e., [0, 23] for 24 FPS, [0, 24] for 25 FPS and [0, 29] for 29.97 and 30 FPS. The fifth and final data byte defines the sub-frame offset as an unsigned integer between [0, 99]. (MIDI File Format 2013.)

Time Signature meta event defines the time signature for the sequence. The length field value is always 4 and the event data contains bytes for numerator, denominator, metronome pulse and the number of 1/32 notes per quarter note, in that order. The first data byte defines the numerator as an unsigned integer between [0, 255]. The denominator is defined as 2^n where n is the value of the second data byte as an unsigned integer between [0, 255]. The third data byte defines the number of 1/24 quarter notes (clock signals) between metronome clicks as an unsigned integer between [0, 255]. The fourth and final data byte defines the number of 32nd notes per quarter note (24 clock signals) as an unsigned integer between [1, 255]. The default time signature is 4/4, 24 1/24 quarter notes between metronome clicks and 8 1/32 notes per quarter note. (MIDI File Format 2013.)

Key Signature meta event defines the key and scale of a sequence. The length field value is always 2 and the event data contains bytes for key and scale, in that order. The first data byte defines the key as number of sharps or flats as an 8-bit two's complement signed integer between [-7, 7]. A positive value for the key specifies the number of sharps and a negative value specifies the number of flats. The second data byte defines the scale as an 8-bit unsigned integer whose value is 0 for a major scale and 1 for a minor scale. (MIDI File Format 2013.)

Sequencer Specific meta event defines sequencer specific information and is not portable. The length field value defines the number of bytes in the event data buffer. The first data byte is interpreted as an unsigned 8-bit integer and if the value is not 0, it specifies the sequencer specific event code. if the value is 0, the 2nd and 3rd data bytes should be interpreted as a 16-bit big-endian unsigned integer that specifies the event code. The sequencer specific event codes are documented in manufacturer specifications. (MIDI File Format 2013.)

2.4.3 System Exclusive Events

Like the Sequencer Specific meta event, system exclusive (SysEx) events are used for signalling MIDI hardware or software specific events. There are three types of system exclusive events: normal, divided and authorization system exclusive events. The following table illustrates the system exclusive event layout: (MIDI File Format 2013.)

Table 14. System exclusive event layout

Field	**Type**	**Size**	**Value**
Delta time	Variable-length	1-4	See following text
Type	8-bit unsigned integer	1	240 or 247
Length	Variable-length	1-4	See following text
Data	See following text		

The variable-length delta time value is interpreted as ticks since the previous event in the same track for all event types (MIDI File Format 2013).

For system exclusive events, the type field value is 240 for both normal and divided system exclusive events and 247 for authorization system exclusive events (MIDI File Format 2013).

The variable-length length field describes the event data size in bytes. The event data contents depend on the system exclusive event type. (MIDI File Format 2013.)

Normal and divided system exclusive events are identified by the value of the last byte of the event data buffer. If the byte value is 247, the event is a normal system exclusive event, otherwise it is a divided system exclusive event and following system exclusive events contain more data for the divided system exclusive event. Other than the first part of a divided system exclusive event are identified with event type field value 247. The last part of a divided system exclusive event has the last byte of its event data buffer set to 247. (MIDI File Format 2013.)

On older hardware with very limited MIDI bandwidth, a large amount of data in a normal system exclusive event could cause following MIDI events to be transmitted after the time they should be played. Divided system exclusive events allow splitting a large event data buffer to smaller blocks and transmitting them with other events between them, avoiding bandwidth issues. (MIDI File Format 2013.)

3 Pitch Detection

3.1 Introduction

Pitch detection has been researched for a long time but the research has been mostly focused on static processing as opposed to dynamic or real-time processing which was a requirement for the prototype. Most existing pitch detection algorithms are capable of detecting individual notes which is by itself a relatively complicated process. Detecting multiple simultaneously playing notes from a single audio signal is much more complicated and the algorithms that try to achieve this suffer from accuracy issues and are not suitable for real-time processing. Even with modern algorithms and static processing on powerful machines, translating a symphony performed by a full orchestra to musical notation from a single audio signal containing the sound from all instruments is practically impossible.

The prototype was targeted specifically for monophonic (only one note can be played at a time) instruments which simplified the pitch detection considerably. The pitch detection algorithms described here are targeted for detecting the pitch of individual notes from an audio signal that contain only the notes being detected. The two algorithm types discussed here are zero-crossing and autocorrelation algorithms.

3.2 Zero-crossing Algorithms

The simplest pitch detection algorithms are based on interpreting the shape of a plotted audio signal. The zero-crossing algorithm works by finding the points where the signal amplitude crosses the zero point from negative to positive or the other way around. The signal wavelength is the distance between two such points and the frequency is simply the inverse of the wavelength in seconds. (McLeod 2008, 11-12.)

In practice, the input signal will contain some signal noise which will cause false zero-crossings. The number of these false zero-crossings can be reduced by filtering the signal with a low-pass filter, i.e., each sample is calculated from the average or weighted average value of nearby samples in the source signal. The problem with this approach is that in practice, the source signal sample rate (the number of samples in the recorded sound wave) is typically at most 44100 samples per second. With lower frequencies, the average can be taken from a relatively large number of samples but with higher frequencies, taking the average from too many samples flattens the signal and information is lost.

Another way to reduce the number of false zero-crossings is to register a zero-crossing only when the amplitude crosses from a negative threshold to a positive threshold. This can be thought of as thickening the zero line. When used together, these two techniques significantly reduce the number of false zero-crossings caused by signal noise. The following figures 1 and 2 help visualizing the algorithm:

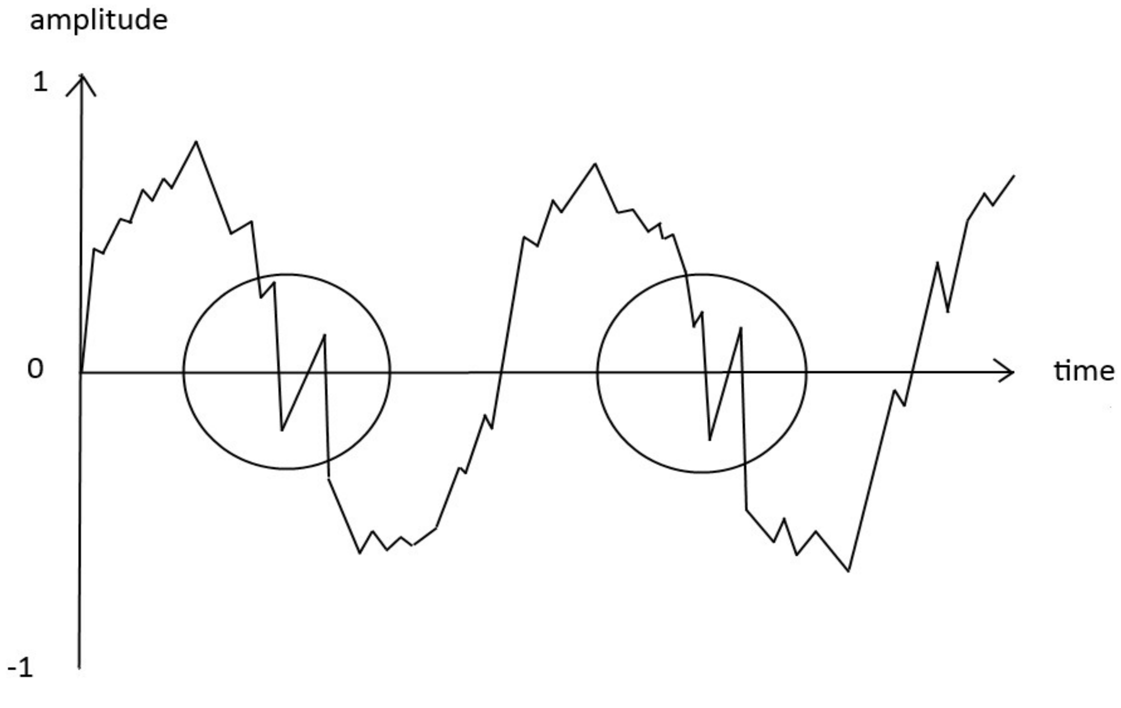


Figure 1. Signal with noise and marked false zero-crossings

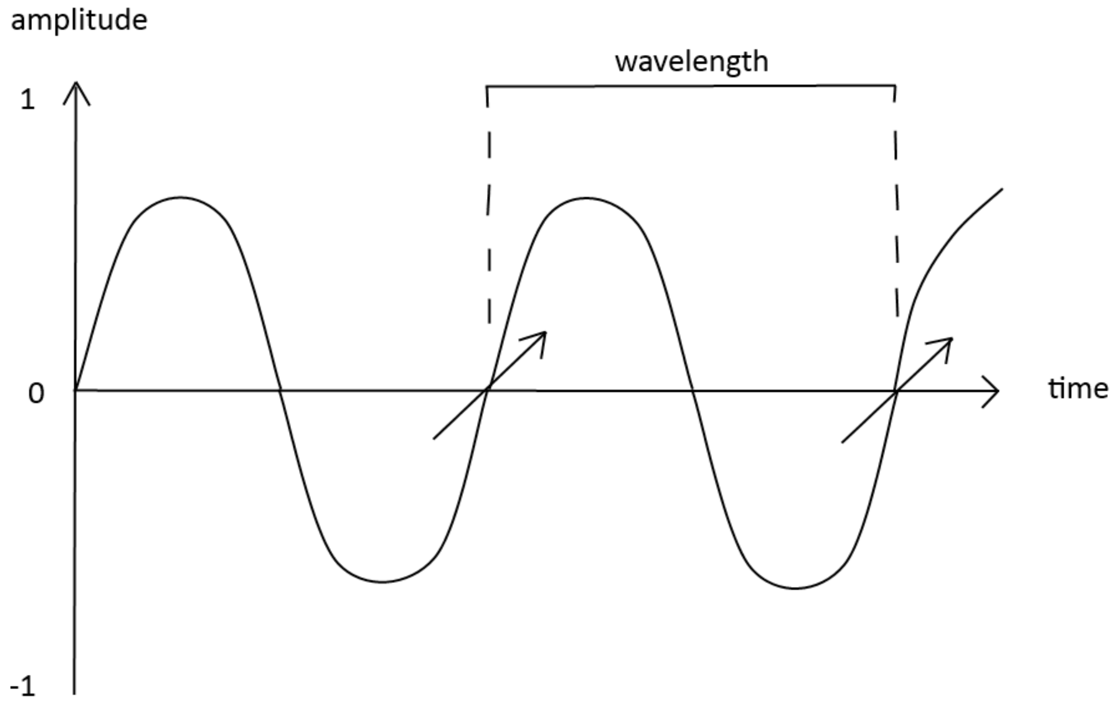


Figure 2. Filtered signal with marked zero-crossings

The zero-crossing algorithms are suitable for monophonic instruments that produce low to medium frequency audio signals that resemble sine waves. As an example, the shape of audio signals containing human singing are very close to sine waves. In addition, most people cannot produce very high frequency notes and therefore most songs do not contain them. On the other hand, some instruments, such as violins can produce high frequency audio signals that may contain multiple peaks and zero-crossings within a single wave, which is why the zero-crossing algorithm is suitable only for some instruments.

3.3 Autocorrelation Algorithms

Autocorrelation algorithms work by comparing a signal to a shifted version of itself. The idea is that when the amount of shift is close to the signal wavelength, the shifted signal will overlap the original signal. To test how closely the signals overlap each other, each sample in the original signal must be compared to the corresponding sample in the shifted signal and the results are summed to get a single autocorrelation value. These values are calculated for all possible shift amounts and the results form an autocorrelation function. The original signal must contain at least two whole wavelengths so that it can be shifted by a whole wavelength and can still contain another to compare to the original. (McLeod 2008, 12-19.)

One way to calculate the autocorrelation values is to calculate the squared difference of each value. The difference of the values is squared because the values are typically between [-1, 1] and the difference can therefore be negative but taking the square of the difference makes all autocorrelation values positive. This could also be done by taking the absolute value of the difference. The autocorrelation values do not matter, they make sense only when compared to other autocorrelation values of the same autocorrelation function. When the autocorrelation values are calculated using a difference function, they are closest to zero where the shifted signal overlaps the original most closely.

Since the sample window should contain at least two whole wavelengths, one way to calculate the autocorrelation function is to compare the first half of the sample window contents to a part of the sample window contents that contains just as many samples but the offset or shift *s* of the first sample is between [0, *W* / 2], where *W* is the number of samples in the sample window. For squared difference autocorrelation, this can be described mathematically as:

$$d(s)=\sum_{j=0}^{W/2-1}(x_j-x_{j+s}),\quad 0\leqslant s\leqslant W/2 \tag{5}$$

The following figures 3, 4 and 5 help visualizing the situation:

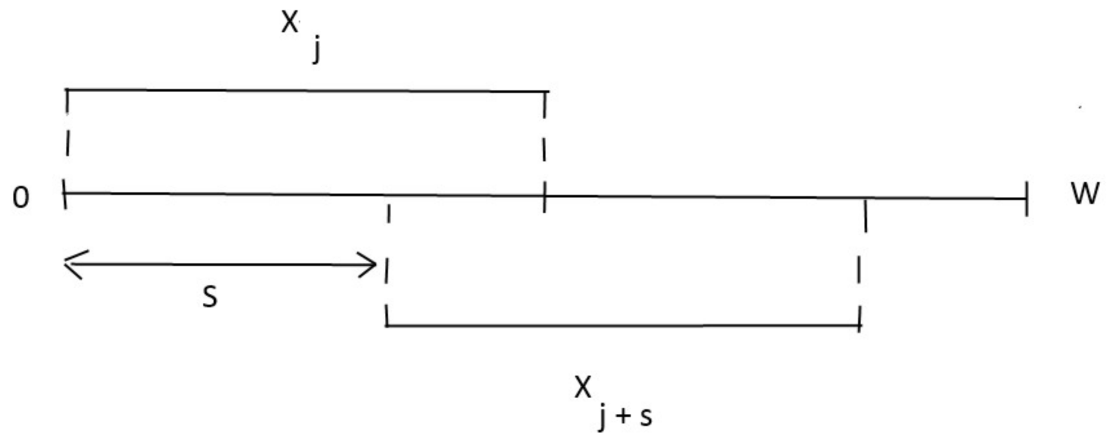


Figure 3. One way to get the original and shifted signals in autocorrelation

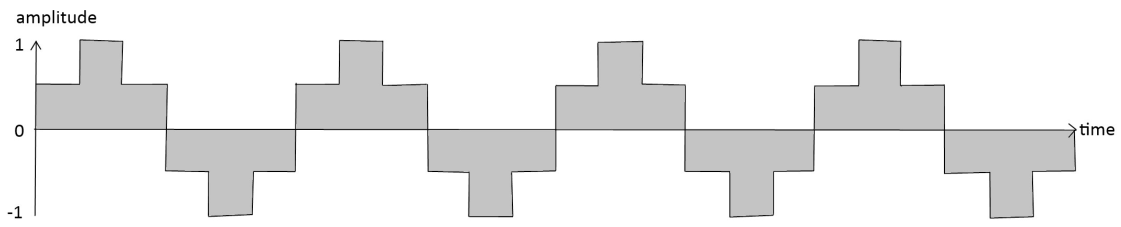


Figure 4. Example sample window

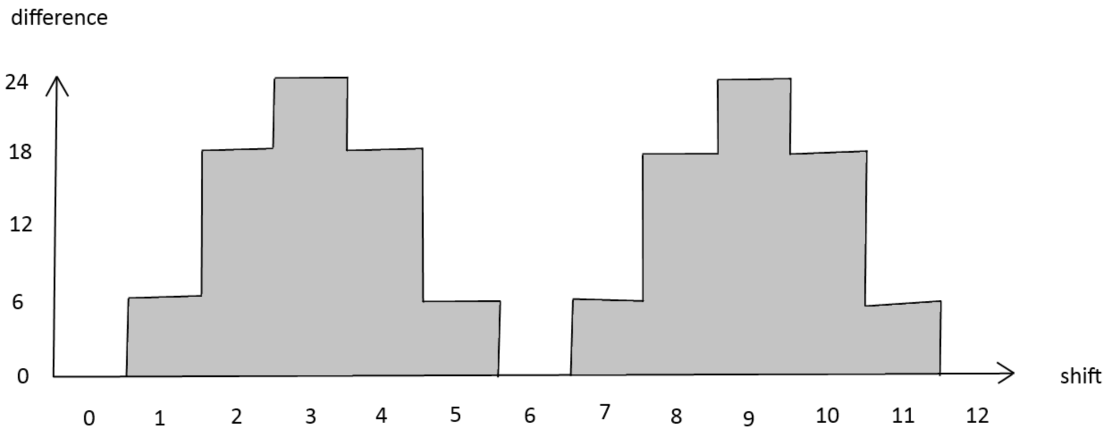


Figure 5. Example squared difference autocorrelation function

The first value of the squared difference autocorrelation function will always be zero. This is because the shift for the first autocorrelation value is zero and therefore the original signal is being compared to itself. The other throughs in the autocorrelation function are possible frequencies. In the example, the second through occurs when shift is 6 samples which is obviously the wavelength of the source signal. The third through occurs when shift is 12 samples which is twice the wavelength of the source signal, i.e., a lower octave. In practice, the autocorrelation function may also contain throughs at half-octaves and the lower octave autocorrelation values may actually be closer to zero than the autocorrelation value for the correct octave. This is why an octave picking algorithm must be used.

The prototype implements the octave picking algorithm as follows: Find the second through in the autocorrelation function and mark it as the match. Then find all remaining throughs and at each through, mark it as the match if the through is *k* units closer to zero than the previously marked through, where *k* is an arbitrary threshold constant. It was found by experimentation that 6% of the maximum autocorrelation value worked very well as the through picking threshold constant *k*.

Unlike zero-crossing algorithms, autocorrelation algorithms are suitable for all monophonic instruments, including instruments that produce signals that contain multiple zero-crossings within a single wavelength. The biggest source of errors in autocorrelation algorithms is octave picking. The previously described octave picking algorithm worked very well with squared difference autocorrelation after a suitable through picking threshold value had been found by experimentation. This was also the frequency calculation method used in the final version of the prototype. The following figure illustrates a signal that is problematic for zero-crossing algorithms but not for autocorrelation algorithms:

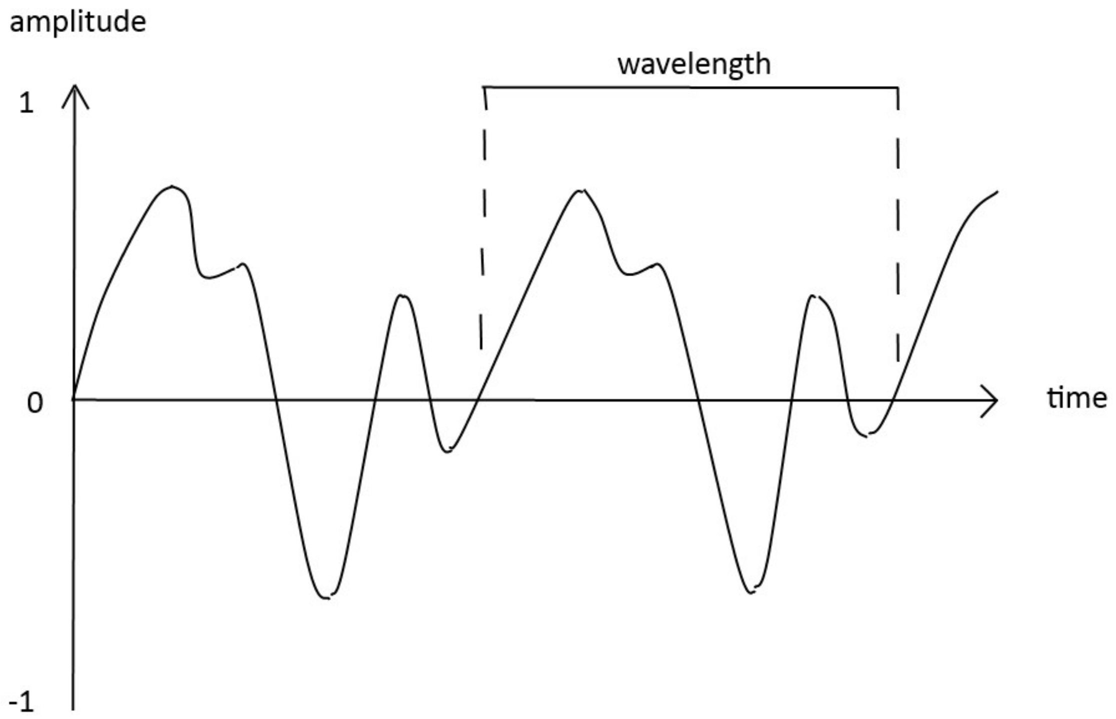


Figure 6. Signal with multiple zero-crossing within a single wavelength

3.4 Practical Issues

An important concept that is relevant for both types of algorithms is the sample window size. The sample window size is the length of the recorded audio signal that the algorithm requires as input before it can detect a pitch from it. If the sample window size is too small, these algorithms cannot detect the pitch from the given samples. If the sample window size is bigger than required, the algorithms may perform unneeded processing depending on their implementation.

Reducing the number of samples that need to be recorded before they can be analyzed reduces the latency between the signal generation (playing an instrument) and pitch detection which is critical in real-time processing. If a pitch detection algorithm is implemented so that a whole sample window is recorded before it is analyzed, the sample window size directly affects the latency. As an example, the frequency of MIDI note number 7 (G-1) is 12.250 Hz and its wavelength is therefore about 0.082 seconds (the inverse of frequency). The minimum required sample window size is about 0.082 seconds (one wavelength) for zero-crossing algorithms and about 0.164 seconds (two wavelengths) for autocorrelation algorithms. This magnitude of latency, especially the latency for autocorrelation algorithms, is unacceptable in real-time applications.

The algorithmic complexity of zero-crossing algorithms is O(n) and their sample window size must fit only one wavelength. The algorithmic complexity of the brute-force autocorrelation algorithms is $O(n^2)$ and their sample window size must fit two wavelengths. As an example, the frequency of MIDI note number 0 (C-1) is about 8.176 Hz and its wavelength is therefore about 0.122 seconds (the inverse of frequency). If the source signal sample rate is the typical 44100 samples per second, the sample window must contain about 5400 samples for zero-crossing algorithms and about 10800 samples for autocorrelation algorithms. To calculate a frequency, the zero-crossing algorithms would therefore require about 5400 operations while the brute-force autocorrelation algorithms would require about 117 million operations. In other words, the brute-force autocorrelation algorithms are not suitable for real-time processing with large sample window sizes.

4 Implementation

4.1 Overview

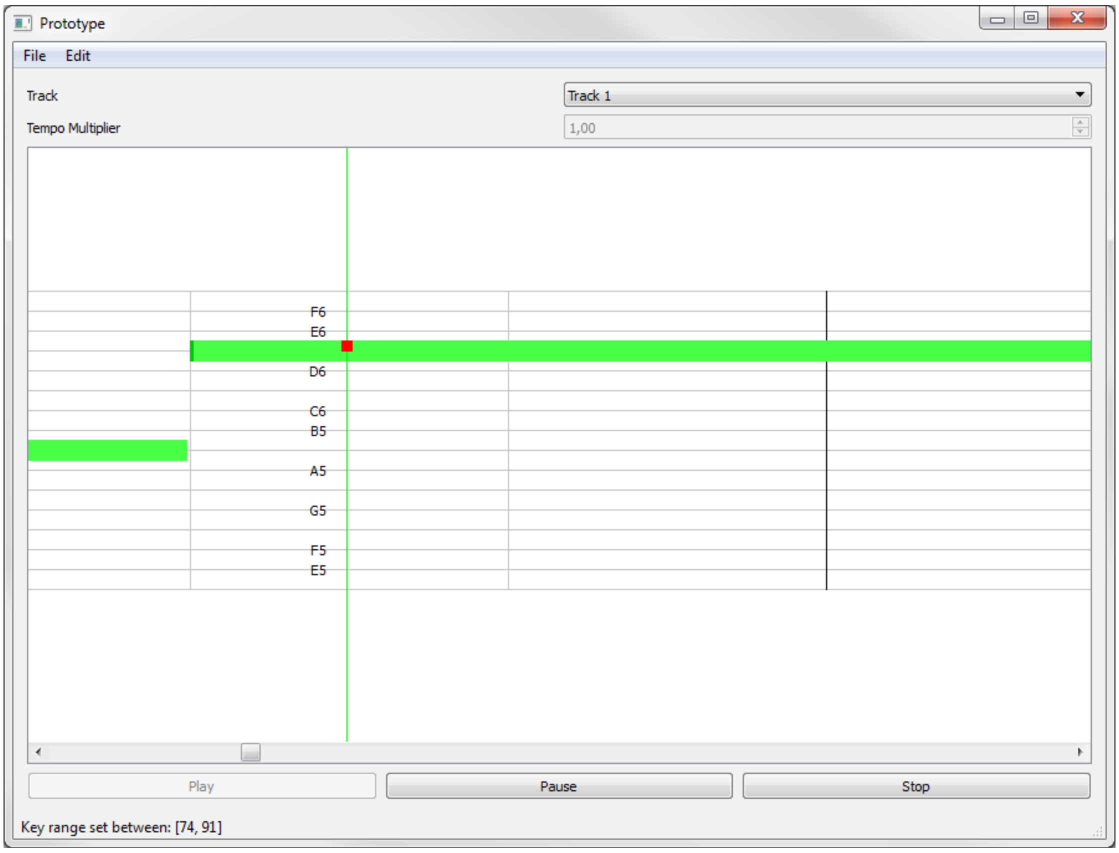
The prototype itself is not very complicated and its source code size is only about five thousand lines of code. Most of the development time was spent on studying the related subjects and implementing small test programs. As an example, before the MIDI playback functionality was integrated to the prototype, two smaller MIDI playback test programs were implemented. The first was a program that would send individual MIDI events to the MIDI driver when numeric keys were pressed on the keyboard. The second was a program that would send a hard-coded MIDI event buffer to the MIDI driver when the space key was pressed. Implementing the first program revealed that sending individual MIDI event was not an option due to large latencies. Implementing the second program revealed that buffered playback could be used in the prototype implementation. For the final MIDI playback implementation, the hard-coded MIDI event buffer was removed and double buffering and multi-threading with related thread synchonization were added to the buffered playback implementation.

The first pitch detection implementation used a zero-crossing algorithm. With some experimentation and tweaking, the zero-crossing algorithm accuracy was improved considerably but it was still only usable for very few instruments. The zero-crossing based pitch detection algorithm was abandoned and an autocorrelation based pitch detection algorithm was implemented. The autocorrelation based algorithm also required a lot of experimentation and tweaking but in the end, it was a lot more reliable than the earlier zero-crossing based algorithm. During the development, some pitch detection related test programs were written and the pitch detection implementation was rewritten almost completely multiple times.

The implementation consists of three major parts: MIDI playback, pitch detection and their visualization. The MIDI playback and audio recording functionality are running in worker threads while the pitch detection and visualization is running in the UI (user interface) thread. The UI thread handles all synchronization between the threads such as synchronization of MIDI playback and visualization.

On each frame, the main loop in the UI thread processes any UI, MIDI playback and wave input events, in that order and then updates the visualization. The UI events consist of button clicks, short key presses etc. The MIDI playback events consist of synchronization and buffer complete events. The wave input thread generates only buffer complete events.

The following figure shows a screen capture of the prototype:



Key range set between: [74, 91]

Figure 7. The prototype running

4.2 MIDI Playback

The MIDI playback thread plays queued MIDI event buffers. These event buffers are updated in the UI thread and sent to the MIDI playback thread. When a buffer is completed, the UI thread is notified and the buffer can be updated and queued again. The implementation uses double buffering so that while one buffer is being used by the MIDI playback thread, the other can be updated in UI thread. The Win32 MIDI event buffers use a non-standard format which is why the Standard MIDI File contents must be preprocessed before they can be sent to the Win32 MIDI playback device.

The MIDI playback thread sends synchronization events when the first MIDI event is played in the MIDI playback thread after playback has been started. In practice, there is a slight delay before playback actually starts and and the use of these synchronization events improves the synchronization of MIDI playback and visualization by a few milliseconds. However, this was an insignificant improvement since the Win32 MIDI playback implementation suffers from random latency issues. The playback position can be queried from the device but the reported position may be off by as much as 200 milliseconds. This differs greatly between different songs and slightly between different playbacks of the same song. The prototype allows the user to specify this latency but due to its seemingly random nature, this does not solve the problem.

MIDI stream playback was inconvenient in a few ways. Seeking and pausing within a MIDI stream would ignore notes that should already be playing at the position where playback is started because their Note On MIDI events reside before the position. To implement seeking and pausing almost correctly, some events, e.g., the Note On events of playing notes, would need to be added at the position where playback is started. This would still not be perfectly correct because a note may sound very different in the beginning than in the middle or in the end. The tempo scaling feature in the prototype is implemented by adding extra Set Tempo meta events within the event stream but has the limitation that the tempo scaling can be changed only by restarting the whole song.

4.3 Wave Input

The wave input thread handles recording audio buffers. The UI thread sends these buffers to the wave input thread and when a buffer is full, the UI thread is notified. The recorded audio buffer can then be processed in the UI thread and when it is no longer needed, it can be reused. As in the MIDI playback thread the wave input implementation uses double buffering so that while one buffer is being used by the wave input thread, the other can be processed in UI thread.

The size of the audio buffers is equal to the sample window size and their input frequency is 44100 samples per second. The sample window size can be adjusted by the user but by default it is sized so that it fits two whole wavelengths of a note that is one half-tone lower than the lowest note in the active track. The lowest note has the longest wavelength and the one half-tone margin is left to account for the user playing incorrect notes.

The amount of time the UI thread spends processing a recorded audio buffer must not exceed the audio buffer length in seconds. This is because the implementation uses two buffers of equal size and if one is being processed for too long, it will not get sent to the wave input thread in time and information is lost.

4.4 Pitch Detection

The pitch detection implementation uses a squared difference autocorrelation algorithm. When an audio buffer has been recorded in the wave input thread, it is given to the pitch detection pipeline. The audio buffer sample values are originally 16-bit unsigned integers where 0 represents the most negative amplitude and the maximum value represents the most positive amplitude. The audio buffer sample amplitudes are converted to floating point values between [-1, 1] and the converted sample values are then given to the pitch detection algorithm.

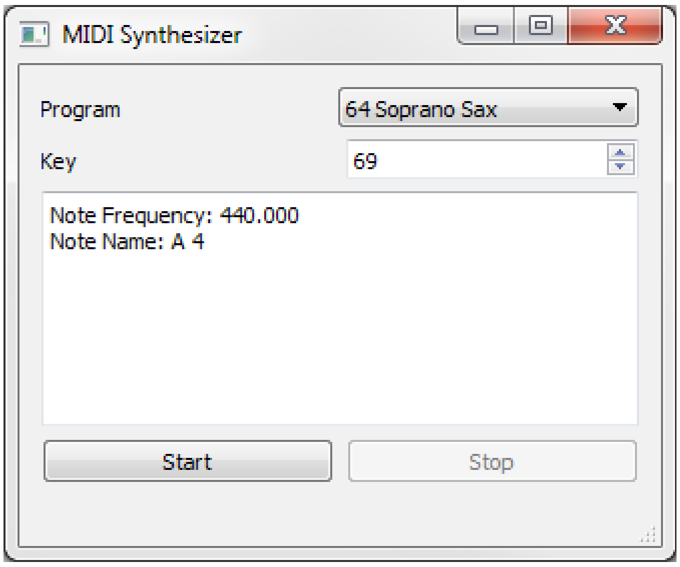
The first phase in pitch detection is determining whether the user has provided any input. This is done by finding the highest absolute amplitude value and if it is below a specified input threshold constant, the sample window contents are interpreted to contain noise. The threshold value can be adjusted by the user but the default value was working very well in tests.

The second phase in pitch detection is running the squared difference autocorrelation algorithm for the expected note range. By default, the expected note range includes notes between one half-tone below and one half-tone above the notes in the active track. The expected note range can also be adjusted by the user. By limiting the note range, the pitch detection algorithm has less room for error and latency is minimized.

The last phase in pitch detection is octave picking. The prototype uses the octave picking algorithm described in section 3.3. After a pitch has been found, it is then converted to a MIDI note number using equation 2 and the result is visualized by drawing a pitch marker on the screen.

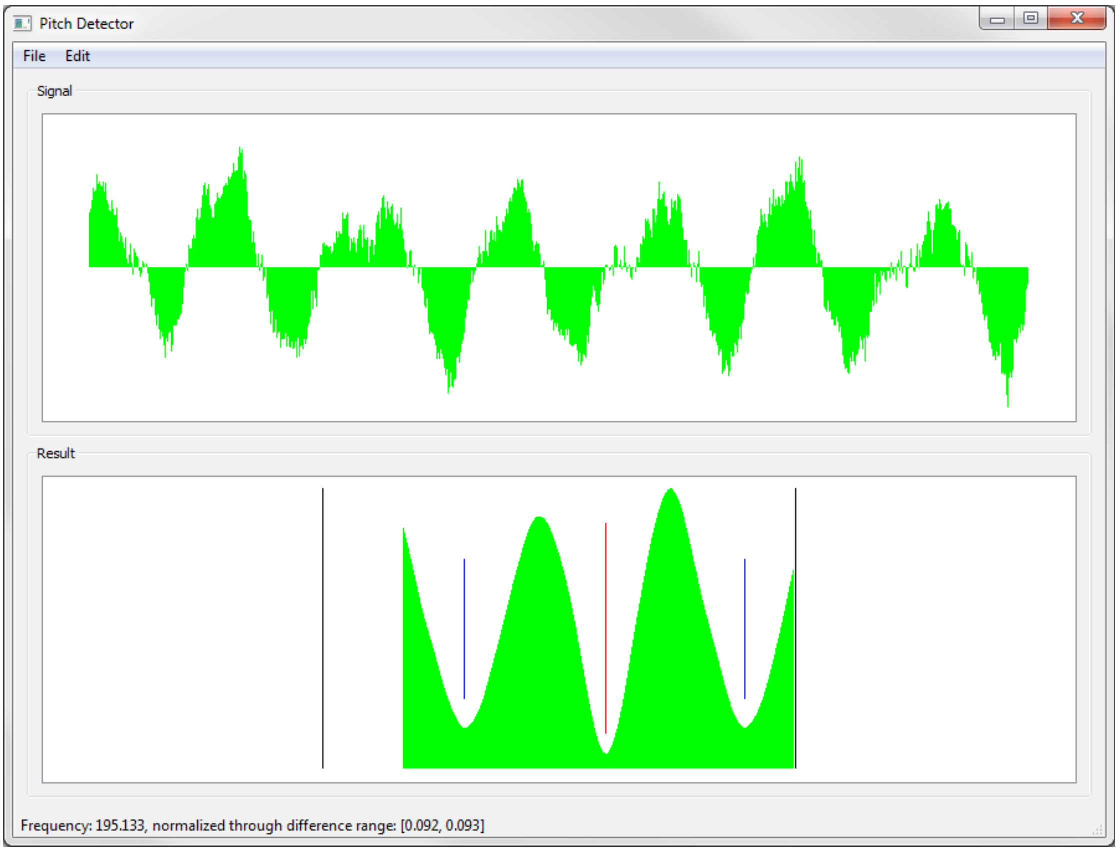
4.5 Testing

During the development of the prototype, a major concern was the accuracy of the pitch detection algorithm. Since the algorithm is meant for processing a large amount of data in real-time, using a C++ debugger for verifying the results was very inconvenient. For this reason, two utility programs were developed for testing the pitch detection algorithm: one for providing input data and another for visual debugging. The program that provided input data was used for playing individual MIDI notes through the speakers using a user specified instrument and note number. The program that provided visual debugging information was reading microphone input, running a pitch detection algorithm on the recorded data and visualizing the input data and results in real-time. The microphone was placed right next to the speakers to minimize signal noise. The following figures 8 and 9 show screen captures of the described programs:



Note Frequency: 440.000
Note Name: A 4

Figure 8. Program for providing input data



Frequency: 195.133, normalized through difference range: [0.092, 0.093]

Figure 9. Program for visual debugging

5 Further Work

The prototype implements most of the originally planned functionality but has some unresolved issues and incomplete or missing features. One major missing feature is recording the user input to an audio file for further analysis. This would be relatively easy to implement since the prototype already implements audio buffer recording for pitch detection. The recorded audio buffers could be stored to a large in-memory buffer or written directly to an audio file. However, any runtime audio data preprocessing or file output should be handled in a separate thread since the UI thread must not get blocked because it is also responsible for MIDI playback and pitch detection. The best option would probably be to store the audio data to a large in-memory buffer which would get written to an audio file when the buffer is full or when the user stops playing.

The biggest problems with the prototype pitch detection algorithm are latency and the algorithmic complexity. The latency issues could be solved by using a sliding sample window. This means that the whole sample window is not recorded at once but only small parts of it and the sample window would be formed from these parts. The oldest part gets discarded when a new part is recorded. The sample window size could also be changed dynamically based on the expected input. The algorithmic complexity could be reduced from $O(n^2)$ to $O(\log(n))$ by using a Fast Fourier Transform (FFT) algorithm for calculating the autocorrelation function. This was not implemented in the prototype due to a lack of understanding of the mathematical basis behind the algorithm. If background noise becomes an issue, e.g., due to low quality microphones on some mobile devices, the noise can be reduced with the same kind of signal filtering schemes that were used for zero-crossing algorithms.

The MIDI stream playback implementation is platform dependent and works only on Windows XP and later Windows operating systems. As discussed in section 4.1, the implementation also has some other issues, such as the random latency problem. All of these problems could be resolved by converting the MIDI data to a waveform data and playing that instead of streaming MIDI events. This approach would also be platform independent. We found one existing open source C++ library, Timidity++ that could be used for implementing this type of functionality.

If a production quality system is to be implemented, it should include some sort of a scoring system. The prototype shows the detected pitch in real time but does not compare it to the song being played. Implementing the scoring system would not be trivial. Examples of features that might be difficult to implement are distinguishing between playing a long note and playing the same note multiple times in fast succession and separating vibrato from fast note changes. The first iteration in developing a scoring system could be based on comparing the detected input to the expected input whenever the pitch detection algorithm is executed. The comparison could allow some error, maybe based on a difficulty level. The score could be given based on the number successful comparisons, perhaps as a percentage of all comparisons.

A production quality system would need to be rewritten almost completely. The Standard MIDI File parser might be the only module that could be reused as is. Everything else is either works only on Windows or uses the Qt software library, which might be too heavyweight for mobile applications. For example, a 64-bit Windows release version of the prototype includes 40 megabytes of Qt specific dynamic link libraries while the executable itself is only 134 kilobytes. The installation package size might reduce considerable if the Qt library was linked statically but this was not tested during the development of the prototype. However, the Qt software library was used mostly for implementing the user interface, which would need to be reimplemented anyway.

References

MIDI. Retrieved 01.12.2013. http://en.wikipedia.org/wiki/MIDI

MIDI File Format. Retrieved 01.12.2013.
http://www.sonicspot.com/guide/midifiles.html

General MIDI. Retrieved 01.12.2013. http://en.wikipedia.org/wiki/General_MIDI

McLeod, P. 2008. Fast, Accurate Pitch Detection Tools for Music Analysis. PhD thesis. University of Otago.