SF2A: FROM SOLFEGE TO AUDIO (UNDER CONSTRUCTION)

JAMES CARLSON AND HAKON HALLGRIMUR

Abstract. We report on `sf2a`, a unix command-line program to create sound files from text such as `moderato: do re mi`. SF, a mini-language for music input is introduce. The program is based on a modular engine that may find other uses. Source code is available at http://github.com/jxxcarlson/sf2sound .

Contents

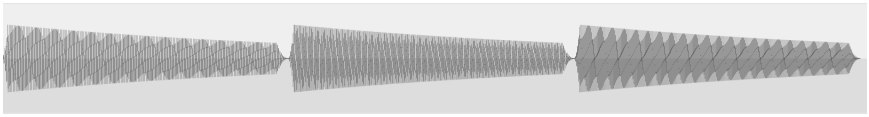
Date: March 20, 2011.
file = sf2a.tex.

1. Introduction

This report describes a unix command-line program, `sf2a`, that converts music, represented by conventional solfa syllables and other symbols, into a playable audio file. Here is an example:

```
% sf2a 'do re mi'
```

Executing this command produces the sound file `out.wav` which can be played on any computer and whose visual representation is as below.



out.wav

We can also set rhythm values, tempo, etc, as well as the name of the output file

```
% sf2a -o foo 'moderato: \
  q do . e re mi . q fa sol | h la sol'
```

The token `q` stands for a quarter note. All of the notes following it will be quarter notes until a new rhythm symbol is encountered. Thus the token `e` changes the current rhythm value to an eighth note, and `h` changes it to a half note. The command `moderato:` sets the tempo. We could also have said `tempo:144`. The symbols `.` and `|` are there only to make the text easier to read. They are ignored by `sf2a`. The symbol is used as with any unix command to break up a long line. The option `-o foo` sets the name of the output file, which will be `foo.wav`.

One can compose in more than one voice:

```
% sf2a -o foo 'decay:2.0 voice:1 do re do voice:2 sol fa mi'
```

In its current implementation, `sf2a` can handle up to ten voices may. For this many voices, command-line input is impractical. Here is a monophonic example of input taken from a file:

```
% sf2a -f ex3
```

where `ex3` is a file with contents

```
Example 2. By Anonymous
@attack:0.01 @release:0.02
```

```
@harmonics:1.0:0.4:0.2:0.1
moderato:
| f: q do re . e mi fa . q sol re |
| stacc: p: q do re . e mi fa .  q sol re |
| leg: do mi re do |
```

The output of this command is the file `out.wav`. Let us examine the input. We begin with the music and then discuss the commands like `attack:0.01`.

The symbol `f:` stands for *forte* and `p:` for *piano*. Likewise `leg:` and `stacc:` stand for *legato* and *staccato*. The symbols `|` and `.` have no meaning for `sf2a` and are ignored by it. They do, however, help us (humans) to parse, read, and write the text. The bar-line symbol `|` the period are used to separate whatever blocks of text the author would like to delineate. A system of ignorable text makes it possible to enter extensive notes if so desired. One can also insert explicit comments. e.g., `// This is a comment`.

Two of the symbols in the input are *accented* An accent is a trailing non-alphanumeric character. The accented symbol

`sol,`

signals the end of a phrase. At phrase endings, notes are slightly shortened and a compensating rest is inserted. The accented symbol `ti_` is the note `ti_` one octave below. Likewise `ti^` is `ti` one octave above.

Let's now look at the commands. The first two,

`@attack:0.01` and `@release:0.02`,

determine the attack and release of the note by shaping its waveform. The command

`@harmonics:1.0:0.4:0.2:0.1`

determines the timbre (tone color) of the note by controlling the relative proportions of the overtones of the fundamental tone. Finally, the command `fundamental:130` sets the frequency in Hertz of `do`.

None of the commands discussed are mandatory. If they are omitted, the variables which they control take their default values. See section 4 for a full discussion of the available commands.

2. Examples

The installation of `sf2a` brings with it several other programs: `mtalk` and `dict`. These are desibed below.

2.1. **Playing an audio file.** It is convenient to have a command line program to play audio files. If one is not available, one can do something like this (mac):

```
#!/bin/sh

open -a /Applications/QuickTime\ Player.app/ $1
```

Name the above file `play` and put in somewhere in the executable path. The you can do this:

```
% play foo.wav
```

We will assume the existence of such a command.

2.2. **Producing an audio file with `sf2a`.** Use `sf2a 'do re mi'` to produce a `.wav` file, `out.wav` representing this melody.

-h, –help show this help message and exit -f FILENAME, –file=FILENAME -o OUTPUT, –output=OUTPUT -s SCALE, –scale=SCALE

-h, –help: Display help.
-f, –filename: Take input from file: `sf2a -h mymelody/`
o, –output: As in `sf2a 'do re mi' -o foo`, write output to `foo.wav`.
-s, -scale: As in `sf2a 'do re mi' --scale diatonic` or `sf2a 'do re mi' --scale chromatic`, select scale. Default is XXXX.

2.3. **mtalk.** The `mtalk` program converts text into an audio file by converting it first to solfege, then running `sf2a` on it. Here is an example:

```
% mtalk 'I would like three apples.  Oh, and four pears too!' -p -t allego
```

Use `mtalk -h` for more information.

2.4. **Dictation.** Use the command `dict -m` to make a diction exercise — audio files and web page — form the file which specifies it: the *dictation file*.

Below is a dictation file with file name `dictation.txt`. It defines three dictation exercises in xml format. Note the header, defined by the `lesson` tag. The next section, delimited by the `setings` tag, determines parameters to be used in each dictation exercise. In this example there is only one voice ... XXXX. An exercise is delimited by the `ex` tag.

```
<dictation>

<lesson>1</lesson>
```

```
<title>Basic solfa patterns</title>
<source>Bradley, p. 2</source>
<date>5/6/2011</date>
</dictation>

<settings>

<voice>
octave:3
legato:
harmonics:1.0:0.5:0.25
</voice>

</settings>

<ex>
<index>1</index>
<content>
voice:1 do re ti_ do, fa mi do re do, sol la sol do, ti_ re do
</content>
</ex>

<ex>
<index>2</index>
<content>
voice:1 do mi re do, sol re ti_ do, fa mi sol do, ti_ sol re do
</content>
</ex>

</dictation>
```

2.4.1. *Command options.*

`-h, --help:` Show help.
`-r, --render:` Create the audio files specified by the dictation file.
`-w, --webpage:` Create the web page specified by the dictation file.
`-m, --make:` Render audio files and make web page.
`-i, --input:` Takes an arguemnt, the input file name. If the `-i` option is not used, `dictation.txt` is assumed.
`-o, --output:` Set the ouput file name. the default, used if `-o` is not specified, is `index.html`.
`-c, --catalogue:` diplay catalogue of exercises specified by dictation file.
`-v, --verbose:` Verbose output. Applies to `-c`.
`-p N, --play N:` Play exercise N.

3. Structure

3.1. **The SF Language.** It is apparent from the foregoing that the input to `sf2a`, be it on the command line or in a file, is expressed in a mini-language with its own vocabulary and grammar. We have dubbed this language "SF," for solfa. The language consists of (1) pitch symbols, (2) rhythm symbols, (3) commands. As noted above, the SF machine ignores inputs that are not part of the SF language.

1. Pitch symbols. The basic symbols are `do re mi fa sol la ti`, and their sharp and flat relatives: `di ri fi so, li` and `de ra me fe se le te`. These symbols may carry various accents. The symbol for a rest is `x`.

The symbols `do do1 do2` represent the notes C, C an octave higher, and C two octaves higher. The symbols `do_1 do_2` represent C an octave lower and two octaves lower. Since SF is designed for the convenience of humans, alternate notations abound. Thus `do_ do__` means the same thing as `do_1 do_2`, and `do^ do^^` means the same thing as does `do do1 do2`.

2. Rhythm symbols. The basic symbols are `w h q e s t` for whole, half, quarter, eighth, sixteenth, and thirty-second note. These may carry accents. For example, the symbol `q.` is a dotted quarter note.

3. Commands. Examples are `f:` for forte and `tempo:144`. which sets the tempo to 144 beats per minute. Commands may take zero or several arguments. Thus `cresc:4:f` is a two- argument command. It means: crescendo from the current level to forte over 4 beats

As with pitch symbols, there are alternate notations. Thus `forte:` is the same as `f:`. Above, one could have written `cresc:4:forte`.

There is also a special clase of commands of the form `@anystring`. These commands are passed without change to the next stage of the pipeline. Examples already cited are `@attack:0.01` and `@harmonics:1.0:0.4:0.2:0.1`.

For a full description of SF, see section 4.

3.2. **The SF Pipeline.** The program `sf2a` produces audio from text by applying a sequence of transformations, as pictured below.

$$\text{solfa} \xrightarrow{\texttt{SF}} \text{tuples} \xrightarrow{\texttt{TU}} \text{waveform} \xrightarrow{\texttt{WA}} \text{audio}$$

This sequence of transformations is the *SF pipeline*. Each transformation is carried by an implementation of an abstract machine that processes an input stream to produce an output stream. These machines are SF, TU, and WA.

The first machine, SF, accepts SF text as input and produces *tuples* as output. A tuple T consists of numbers (f, d, a, τ), where f is the frequency in Hertz, d is the duration of the note in seconds, a is its amplitude, and τ is the time constant for the decay of the note.

The second machine, TU, accepts tuples as input and produces waveform data as output. Waveform data is a sequence of numbers $s_0, s_1, s_2, \ldots, s_N$. Suppose that $\phi(t)$ is the ultimate acoustic pressure wave produced by playing the sound file written by `sf2a ex1 'do re mi'`. Then $s_i = \phi(t_i)$. Thus the s_i are *samples* of the actual wave form ϕ. The *sample times* t_i are evenly spaced. One generally takes $t_{i+1} - t_i = 1/44{,}100$ seconds, about 20 microseconds. Thus a waveform file (or sample file) contains 44,100 numbers for each second of audio. This is the standard sample rate for CD's and MP3's.

The third machine, WA, converts the waveform file, which is a long column of ASCII text, into a binary file in some standard audio file format, in this case, `.wav`.

Let us take the solfa text `moderato: q do . e re mi . q fa sol | h la sol` and what happens to it as it moves down the pipeline.

1. The SF machine sends the stream of solfa tokens through a preprocessor to remove barlines and dots with space on either side. The result is notestream `moderato: q do e re mi q fa sol h la sol`. Next, it creates a stream of tuples $Q_1, Q_2, Q_3, ...$ as listed below.

```
261.62556530059868 0.52631578947368418 1.0 0.5
293.66476791740763 0.26315789473684209 1.0 0.5
329.62755691287003 0.26315789473684209 1.0 0.5
349.228231433004   0.52631578947368418 1.0 1.0
391.99543598174944 0.52631578947368418 1.0 1.0
440.00000000000017 1.0526315789473684  1.0 1.0
391.99543598174944 1.0526315789473684  0.3 1.0
```

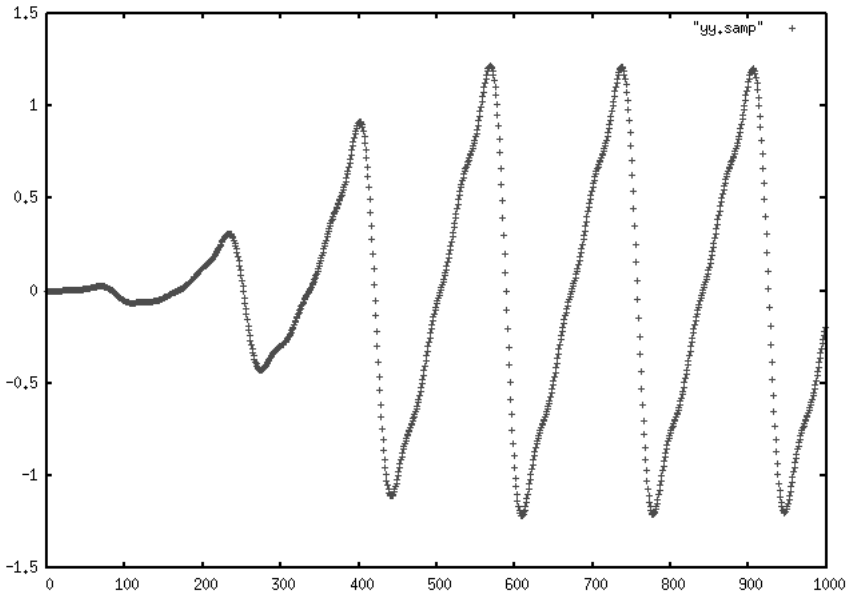
The tuples determine the "music" to be played: the columns, reading from left right, give the pitch (frequency), duration, loudness (amplitude), and decay (how fast the note dies away). The decay constant has the dimension of times. A smaller decay constant, corresponds to more percussive sounds; a larger one corresponds to more sustained sounds.

The tuple stream can also contain special commands like `@attack:0.01`, which are passed through from the solfa text unchanged. In a moment, we will see what the special commands do.

2. The next transformation maps a the stream of tuples $Q_1, Q_2, \ldots, Q_N$ to stream of numbers representing a sampled waveform, as described above. Here are the first ten lines in the waveform stream:

```
0.00000000
0.00000014
0.00000084
0.00000256
0.00000578
0.00001103
0.00001891
0.00003005
0.00004516
0.00006501
```

The waveform stream contains 185,682 numbers. Since 44,1000 numbers stream by each second, we can compute the playing time from the file length: 185,682/44,100 = 4.21 seconds. This is, of course, the same as the sum of the numbers in the second column of the tuple stream. Below is a graphical representation of the first 1000 numbers in the waveform stream:



Waveform, 1000 samples

3. The final transformation creates an audio file in .wav format from the sampled waveform file. It is a binary file, which we may view in hexdecimal form. Below are the the first 20,480 bits of the file, organized in lines of eight 256 bit words:

```
4952    4646    aae8    0005    4157    4556    6d66    2074
0010    0000    0001    0001    ac44    0000    5888    0001
0002    0010    4550    4b41    0018    0000    0001    0000
2631    4d84    47ad    3f1e    0000    0000    b730    0000
0000    0000    6164    6174    aaa4    0005    0000    0000
0000    0000    0000    0000    0000    0000    0001    0001
0001    0002    0003    0003    0004    0005    0007    0008
000a    000c    000e    0010    0012    0015    0018    001b
001e    0021    0025    0029    002d    0031    0035    003a
003e    0043    0048    004d    0053    0059    005f    0066
```

The Python program `solfa2sound` plumbs the output of one stage of the pipeline to the input of the next, effecting the transformation of solfa text into a sound file.

3.3. **The SF Machine.** The SF machine accepts one token of the SF language at time. It operates as follows:

(a) If the token is a note, the machine writes a quadruple to output. The value of the quadruple depends on the input token and the state of the machine, e.g., the value of the `duration` register.
(b) If the token is a rhythm symbol, it changes the value of the duration register of the machine.
(c) If the token is a command of the form `@anystring:` or `@anystring:arg1` etc., that command is executed. By `anystring` we mean a nonempty string consisting of the character `a..z`.
(d) If the token is a command of the form `@anystring:` or `@anystring:arg1` etc., it is passed on unchanged to the TU machine for processing.
(e) If none of the above hold, the token is ignored.

The registers of the SF machine are as follows. // NEEDS UPDATE

`frequency:` Set by the input token if it is a note.

`duration:` Holds the current duration in seconds. Set by the input token if it is a rhythm symbol.

`decay:` Default is 0.5. Set by `decay:0.4` and commands like `staccato:`, `legato:`

`amplitude:` Holds the current amplitude. The default value is 1.0. Set by commands such as `amplitude:0.123`, `forte:` and `piano:` .

`beat:` This can be `e`, `q` or `h`, for eighth, quarter, of half note. A quarter note is the default.

`tempo:` The tempo in beats per minute.

`beatDuration:` This has the value `60.0/tempo`.

`transpose:` If its contents are n, the pitch is shifted by n semitones. Default value is zero.

3.4. **The TU Machine.** The TU machine transforms a stream of tuples to a waveform represented as a sequence of samples. It also accepts certain commands which affect the synthesis of the waveform.

`@harmonic:a:b:c:...:` – sets the amplitudes of the fundamental tone to `a`, of the first overtone to `b`, etc.

`@attack:p:` – sets the length of the attack phase of a note to $t1 = p \times t$ seconds, where the duration of the note is t seconds.

`@release:p:` – sets the length of the release phase of a note to $t2 = p \times t$ seconds, where the duration of the note is t seconds.

4. SF Language Reference

4.1. **Pitch symbols.** The primary pitch symbols are `do re mi fa sol la ti` and their flatted and sharped versions, `de ra me se le te` and `di ri fi si li`. The pitch accents are

`+:` Raises pitch by a semitone, as in `fa+`. Multiple values, e.g., `fa++` are permitted.

`-:` Lowers pitch by a semitone, as in `ti-`. Multiple values, e.g., `ti--` are permitted.

`<n> :` A positive integer suffix of n means: raise the pitch by $(n-1)$ octaves.

`^:` Raise the pitch by an octave, e.g. `do_` is an octave above middle C

`_ :` Lower the pitch by an octaves, e.g., `do_`

`,:` The comma is used to mark phrase endings, as in

```
e mi fa q re h do, q ti_ ti_ re fa ...
```

The note accented by the comma is shortened and a compensating rest is added in order to keep the beat. The relative size of the compensating breath is by default 0.3 of the time value of the note. This value can be set by the user, as in `breath:0.3`.

NOTE: The method used to shape phrase endings is still rather crude. Another possibility to do this: (1)save current value of `release`, (2) emit `release:shorter value` (3) emit note, (4) emit `release: stored value`.

4.2. **SF commands.** A list of commands:

Articulation

`legato:`, `marcato: staccato:`, `decay:arg`

Pitch

`o:arg` where (i) `arg` is (i) `+` repeated n times. This shifts the melody up by n octaves; `-` repeated n times. This shifts the melody down by n octaves; `+n` shifts up by n octaves; `-n` shifts down by n octaves. This command writes $\pm 12n$ the `transpose` to register.

`t:arg` where `+n` shifts up by n semitones; `-n` shifts down by n semitones. This command writes $\pm n$ to the `transpose` register.

Tempo

`tempo:144`

`prestissimo. presto:`, `allegro:`, `moderato:`, `larghetto`, `largo`, `grave:`, `lento:`

`allegro:+`, `allegro:++`, `allegro:-`, `allegro:--` increases or decreases the tempo by a fixed factor.

`tempo:+`, etc. As above, but adjusts current tempo. The sequence `allegro: tempo:+` has the same effect as `allegro:+`.

`accelerando:8:160`

`ralentando:8:120`

Volume

Dynamics, long: `fortissimo:`, `forte:`, `mezzoforte:`, `mezzopiano:`, `piano:`, `piannissimo:`.

Dynamics, short: `ff`, `f`, `mf`, `mp`, `p`, `pp`.

`cresc:beats:level`, `decresc:beats:level`.

4.3. **TU commands.** `@attack:0.01` Sets the attack phase of a note to 0.01 of the total duration of the note.

`@release:0.02` Sets the release phase of a note to 0.02 of the total duration of the note.

`@harmonics:1.0:0.4:0.2:0.1` Sets the strength of the overtones in a note. Here the fundamental has amplitude 1.0, the first overtone 0.4, etc.

5. Installation

5.1. **Download and install.** Download the complete installation from

```
http://github.com/jxxcarlson/sf2sound
```

Unzip/untar the download, cd to the folder that has been extracted, and run the command

```
% sudo sh setup.sh -install YOUR_USER_NAME
```

You might have to configure the variables `$INSTALL_DIR` and `$BIN_DIR`. The first is where all the files go. The second is where you put a symbolic link to `ui.py`, `dict.py`, and `mtalk.py`.

5.2. **Parts list.** The `sf2a` program consists of a goodly number of parts.

`dict.py` : Calls `sf2a` to create a web page of dictation exercise.

`ui.py` : Defines the user interface and is the file executed to drive the entire package. Calls `run(input, output, SCALE)` in `driver.py`

`driver.py` : Coordinates the parts of the SF pipeline. Imports `SFM:SFM`, `parse:getChunk`, `comment:stripComments`, `scales:scale`. The main function is `run(input, output, SCALE)`, which calls `quad2samp`, `mix`, and `text2sf`.s

`quad2samp.c` : Source file for `quad2samp`.

`quad2samp` : Takes a file of quadruples as input, produces a waveform sample file as output.

`mix.c` : Source file for `mix`.

`mix` : Takes $N \leq 10$ waveform sample files as input, produces waveform sample file as output which represents the average of the input files.

`text2sf`: Converts waveform sample file to `.wav` file. XXX

`SFM.py` : The SFM machine. Imports `parse:splitToken`, `note:Note`, `rhythm:Rhythm`, `dynamics:Dynamics`, `stringUtil:*`, `listUtil:interval`, `mapInterval`.

`note.py` : foo, bar

`scales.py` : foo, bar

`rhythm.py` : foo, bar

`dynamics.py` : foo, bar

`parse.py` : foo, bar

`comment.py` : foo, bar

`stringUtil.py` : foo, bar

`stack.py` : foo, bar

`ring.py` : foo, bar

`melody.py` : foo, bar

`listUtil.py` : foo, bar

6. To do

Big things

(1) A command line ear training program for prototyping purposes.
(2) Multi-voice input. Idea: label voices as voice:1 ... endVoice:1, voice:2 ... endVoice:2, ... Process each voice independently ... write files _voice1.samp, _voice2.samp, ... The interleave these to produce a single file foo.samp. Finally, convert this to a wave file. Question: does text2sf support more than two channels?
(3) MIDI keyboard input.
(4) Build a player and integrate it into program

Smaller things

(1) `cresc:8:forte`. etc.
(2) `marcato`
(3) ralentando, accelerando
(4) `temp+`, `temp-`
(5)

Business

(1) Graphical user interface. This would require a C or C++ SF engine to replace the current Python code ... a big job!

(2) An iPad ear training program.
(3) An open source library / website of melodies and other materials.
(4) Sell product to music schools.

References

[1] The Audio Programming Book, Richard Boulanger and Victor Lazzarini, eds.
[2] Audio Programming on iOS – http://timbolstad.com/about/
[3] http://www.portaudio.com/trac/wiki/TutorialDir/Compile/MacintoshCoreAudio
[4] http://www.portaudio.com/docs/v19-doxydocs/
[5] http://www.portaudio.com/download.html
[6] Portaudio tutorial: http://www.portaudio.com/trac/wiki/TutorialDir/TutorialStart
[7] Building portaudio for mac: http://www.portaudio.com/trac/wiki/TutorialDir/Compile/MacintoshCoreAudio
[8] http://people.csail.mit.edu/hubert/pyaudio/
[9] ortaudio-bug: http://pujansrt.blogspot.com/2010/03/compiling-portaudio-on-snow-leopard-mac.html

James Carlson: jxxcarlson@mac.com, www.math.utah.edu/˜carlson
Hakon Hallgrimur: hhallgrimur.wordpress.com