sinsy-project

Programming Project 1

This is a collection of Python scripts for interacting with Sinsy.

Sinsy is a HMM-based Singing Voice Synthesis System that uses MusicXML files as input.

Please refer to the Documentation directory for more information.

Required software:

- Python 2.7 - http://python.org
- SoX - http://sox.sourceforge.net

Optional software:

- Sinsy - http://sinsy.sourceforge.net/

Misc software:

- MuseScore - http://musescore.org

Useful for viewing and playing both MusicXML and MIDI files. Can convert from MIDI to MusicXML.

Alternative Singing Voice Synthesis software

This project focuses on Sinsy, however there are other software packages that can be used for singing synthesis.

espeak / ecantorix

From the espeak website:

> eSpeak uses a "formant synthesis" method. This allows many languages to be provided in a small size. The speech is clear, and can be used at high speeds, but is not as natural or smooth as larger synthesizers which are based on human speech recordings.

- espeak - http://espeak.sourceforge.net/
- ecantorix - https://github.com/divVerent/ecantorix
- Examples: https://github.com/divVerent/ecantorix/wiki/Songs

festival

- festival - http://www.cstr.ed.ac.uk/projects/festival/
- flinger - No longer available

UTAU

- UTAU - http://utau2008.web.fc2.com
- Moresampler, Arpasing (Kanru Hua) - https://webhost.engr.illinois.edu/~khua5/index.php/2017/02/26/introducing-arpasing-for-english-utauloids/

Software Automatic Mouth (SAM)

- SAM - https://github.com/s-macke/SAM

1 Splitting Multiple Parts Of MusicXML File

The scripts `listPart.py` and `keepPart.py` will allow you to split a MusicXML file.

Generally you would run `listPart.py` to list all of the Part ID's that are found in the MusicXML file.

Then you would use that Part ID with `keepPart.py` to process the MusicXML file. It will output a new MusicXML file which only contains the specified Part ID and filtering out the other parts. The old MusicXML file is unaltered and remains intact.

The following explains how to use `listPart.py` and `keepPart.py`

listPart.py

This will list all the parts in an XML file.

```
usage: listParts.py [-h] [xmlfile]

positional arguments:
  xmlfile     name of the xml file or - for stdin

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python listParts.py input.xml

keepPart.py

Each XML file has one or more parts. Sinsy will only render only one part at a time, so if an XML file has more than one part, it is necessary to split it up so that all the parts get rendered separately. Conveniently, this script will help accomplish that task.

```
usage: keepPart.py [-h] xmlfile partid

positional arguments:
  xmlfile     name of the xml file or - for stdin
  partid      id of part to keep

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python keepPart.py input.xml P1

This will read the `input.xml` file and extract the part `P1`.

2 Audio Mixing With SoX

From the SoX home page:

> SoX is a cross-platform (Windows, Linux, MacOS X, etc.) command line utility that can convert various formats of computer audio files in to other formats. It can also apply various effects to these sound files, and, as an added bonus, SoX can play and record audio files on most platforms.

http://sox.sourceforge.net

Mixing Multiple WAV Files

Let's say you have three WAV files:

- vocals.wav
- piano.wav
- drums.wav

The simplest way to mix them is with this command:

```
sox -m vocals.wav piano.wav drums.wav output.wav
```

This will use those files as input and the result will be `output.wav`. Generally these files will be mono. If you want to mix stereo, you'll have to specify if you want to pan left or right.

Adding reverb

The simplest way to add reverb is with this command:

```
sox vocals.wav output.wav reverb
```

Dynamic compression

soxCompand.py

This will run the `compand (dynamic compression)` effect using `sox` with the default parameters, on the specified .WAV file.

```
usage: soxCompand.py [-h] infile outfile

positional arguments:
  infile      name of input file
  outfile     name of output file

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python soxCompand.py input.wav output.wav

Normalizing

This command will normalize an audio file to 0 dB:

```
sox infile.wav outfile.wav gain -n
```

This command will apply 30 dB of gain with a limiter to prevent clipping:

```
sox infile.wav outfile.wav gain -l 30
```

3 Processing MusicXML Files With Sinsy Website

There are two options for generating sound files with Sinsy.

One option is to download the source, compile, and run it on your machine locally. The source is available at

http://sinsy.sourceforge.net

Only the Japanese voice model is available for generating sound files locally

The second option is to send the MusicXML file to the Sinsy website, and to download the generated WAV file. More voice models are available, but the Sinsy website is not always working. The Sinsy website is located at:

http://sinsy.jp

This can be done manually, or the `upload.py` script can be used to automate this process.

upload.py

This will send an XML file for processing to the Sinsy website, and download the resulting generated .WAV file.

```
usage: upload.py [-h] [--spkr SPKR] [--synalpha SYNALPHA]
                 [--vibpower VIBPOWER] [--f0shift F0SHIFT]
                 infile outfile

positional arguments:
  infile               name of the input xml file
  outfile              name of the output wave file

optional arguments:
  -h, --help           show this help message and exit
  --spkr SPKR          speaker, default is 4, Japanese voices: 0=Yoko 1=Xiang-
                       Ling 2=Namine Ritsu S 3=undefined 7=Yoko DNN, English
                       voices: 4=Xiang-Ling (Female) 5=Matsuo-P (Male),
                       Mandarin voices: 6=Xiang-Ling
  --synalpha SYNALPHA  synalpha, default is 0.55 (-0.8 to 0.8)
  --vibpower VIBPOWER  vibpower, default is 1 (0.0 to 2.0)
  --f0shift F0SHIFT    f0shift, default is 0 (-24 to 23)
```

Example usage:

> $ python upload.py --spkr 4 --synalpha 0.55 --vibpower 1 --f0shift 0 input.xml output.wav

4 Multiple Simultaneous Notes In A Vocal Part

If there are multiple simultaneous notes in a vocal part, these are represented in a MusicXML as separate 's and each voice is associated with a part.

To split these out, run `listVoiceParts.py` to list all of the voices and parts found in the MusicXML file. Then run `keepVoicePart.py` which will output a new MusicXML file with only the desired voice/part.

listVoiceParts.py

This will list all the voices and their respective parts in an XML file.

```
usage: listVoiceParts.py [-h] [xmlfile]

positional arguments:
  xmlfile     name of the xml file or - for stdin

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python listVoiceParts.py input.xml

keepVoicePart.py

Each part may have more than one voice. Generally, if there are multiple simultaneous notes (i.e. a chord), each note will be represented by a separate voice, so that each voice only contains a single note.

```
usage: keepVoicePart.py [-h] xmlfile voice partid

positional arguments:
  xmlfile     name of the xml file or - for stdin
  voice       voice to keep
  partid      id of part to keep

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python keepVoicePart.py input.xml 1 P1

This will read the `input.xml` file and extract voice `1` from part `P1`.

useChordNotes.py

This script will replace the first note of the chord with the second note of the chord. All single notes will be turned into rests.

If the desire is to sing multiple notes together, it would be preferable to use multiple voices and not chord notes.

The behaviour when there are more than 2 chord notes is undefined.

```
usage: useChordNotes.py [-h] [xmlfile]
```

```
positional arguments:
  xmlfile     name of the xml file or - for stdin

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

```
$ python useChordNotes.py Fragments.xml
```

useVoice.py

This script will replace the note for the first voice with the note for the specified voice, which may be better for rendering with Sinsy.

```
usage: useVoice.py [-h] xmlfile voice

positional arguments:
  xmlfile     name of the xml file or - for stdin
  voice       voice to use

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

```
$ python useVoice.py moments.xml 2
```

5 Dynamics

The Sinsy voice model doesn't have labelled data for dynamics so tags such as p, mp, mf, and f and crescendo/diminuendo tags are not handled.

The applyDynamicsToWAV.py will post-process a WAV file that was already generated by Sinsy, and will apply the dynamics tags and crescendo/diminuendo. Care should be exercised if dynamic compression is later applied which will reduce the dynamic range.

applyDynamicsToWAV.py

```
usage: applyDynamicsToWAV.py [-h] [--inxml INXML] --inwav INWAV
                             [--outdynamics OUTDYNAMICS]
                             [--outenvelope OUTENVELOPE] [--outwav OUTWAV]
                             [--pppp PPPP] [--ppp PPP] [--pp PP] [--p P]
                             [--mp MP] [--mf MF] [--f F] [--ff FF] [--fff FFF]
                             [--ffff FFFF] [--sp SP] [--sf SF]
                             [--smoothing SMOOTHING] [--crescendo CRESCENDO]
                             [--diminuendo DIMINUENDO]

optional arguments:
  -h, --help            show this help message and exit
  --inxml INXML         name of the input xml file or stdin if not specified
  --inwav INWAV         name of the input wav file
  --outdynamics OUTDYNAMICS
                        name of output file for dynamics information for
                        debugging (optional)
  --outenvelope OUTENVELOPE
                        name of output file for envelope information for
                        debugging, the resulting file can be plotted using a
                        program like gnuplot (optional)
  --outwav OUTWAV       name of output file for wave or stdout if not
                        specified
  --pppp PPPP           value for dynamic pppp, between 0.0 and 1.0, default
                        is 0.1
  --ppp PPP             value for dynamic ppp, between 0.0 and 1.0, default is
                        0.1
  --pp PP               value for dynamic pp, between 0.0 and 1.0, default is
                        0.2
  --p P                 value for dynamic p, between 0.0 and 1.0, default is
                        0.3
  --mp MP               value for dynamic mp, between 0.0 and 1.0, default is
                        0.4
  --mf MF               value for dynamic mf, between 0.0 and 1.0, default is
                        0.6
  --f F                 value for dynamic f, between 0.0 and 1.0, default is
                        0.8
  --ff FF               value for dynamic ff, between 0.0 and 1.0, default is
                        0.9
  --fff FFF             value for dynamic fff, between 0.0 and 1.0, default is
                        1.0
  --ffff FFFF           value for dynamic ffff, between 0.0 and 1.0, default
                        is 1.0
  --sp SP               value for dynamic sp, between 0.0 and 1.0, default is
                        0.3
  --sf SF               value for dynamic sf, between 0.0 and 1.0, default is
                        0.8
  --smoothing SMOOTHING
                        value for smoothing dynamic changes, represents time
                        percentage of a measure, default is 0.05
  --crescendo CRESCENDO
                        change value for applying crescendo, default is 0.15
```

```
  --diminuendo DIMINUENDO
                        change value for applying diminuendo, should be
                        negative, default is -0.15
```

The workflow for using this script is to generate the initial WAV file either using `upload.py`, directly using the http://sinsy.jp website, or by running the `sinsy` binary locally if it has been installed.

So now you have an XML file that we'll call `song.xml`. You also have a WAV file that you generated from this XML file, we'll call it `song.wav`. Now you use this script to process the WAV file using the dynamics information specified by the `song.xml` file.

This will generate a new WAV file with using the dynamics and crescendo/diminuendo information from the XML file. In this case, the `--inxml` argument would be `song.xml`. The `--inwav` argument would be `song.wav`. The new WAV file will be written to `stdout`.

Example usage:

> $ python applyDynamicsToWAV.py --inxml song.xml --inwav song.wav >output.wav

This will read the XML file from `song.xml`, the WAV file from `song.wav`, and write the WAV file `output.wav`

The options `--p`, `--f`, and related options allow you to specify the actual volume (or amplitude) to use when generating the new WAV file.

- *p* means soft (piano)
- *pp* means very soft
- *ppp* means very very soft
- *pppp* means very very very soft
- *f* means loud or strong (forte)
- *ff* means very loud
- *fff* means very very loud
- *ffff* means very very very loud
- *mp* means half soft (mezzo piano)
- *mf* means half loud (mezzo forte)

These notations when used in sheet music are somewhat ambiguous, it is up to the interpretation of the performer exactly how soft or how loud the music should be played. Specifying a value of `--f 1.0` means that if the dynamics in the xml file specify *f*, then the value of 1.0 means that the sound samples for that portion of the WAV file will be unaltered. A value such as 1.5 will increase the volume (amplitude) by 50% by multiplying the sound sample by 1.5, but clipping is a possibility. A value such as 0.5 will decrease the volume by 50% by multiplying the sound sample by 0.5.

In order to avoid sudden amplitude changes the `--smoothing` option can be used to specify how long it should take for the volume to change. A value of 1.0 means that it will take an entire measure to change the volume (the change in volume will be linear). This would be basically the same thing as crescendo or diminuendo since the change will be very gradual. A smaller value like 0.05 means that the transition will take place in 5% of the measure time, so it would be rather quick in this case.

The `--crescendo` and `--diminendo` options specify how much of an increase or decrease in volume should be applied. If the current volume is 0.5 and the `--crescendo` is 0.15, then the volume will start at 0.5 and gradually change to 0.65 over the course of the measure. One issue here is that not all XML files specify exactly how loud or soft the crescendo should be at the end of the crescendo, so it is mostly just guesswork.

There are a couple of options that can be used to see exactly what is being done to the WAV file.

The first is the `--outdynamics` option. This will output a file that contains the dynamics information from the XML file.

Example usage:

> $ python applyDynamicsToWAV.py --inxml song.xml --inwav song.wav --outdynamics dynamics.txt >/dev/null

This is an example of what the `dynamics.txt` file might look like:

```
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 1}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 2}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 3}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 4}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 5}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 6}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 7}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 8}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 9}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 10}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 11}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 12}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 13}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 14}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 15}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 16}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 17}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 18}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 19}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 20}
{'crescendo': 0, 'dynamics': 'p', 'diminuendo': 0, 'number': 21}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 22}
{'crescendo': 0, 'dynamics': 'mp', 'diminuendo': 0, 'number': 23}
...
```

The **number** field indicates the measure, and the **dynamics** field indicates the dynamics that are applied to that measure. The fields **crescendo** and **diminuendo** are either true (1) or false (0).

The other option for debugging is the `--outenvelope` option, which will output data that can be plotted using a program like **gnuplot**. The dynamics information that gets applied to the WAV file can be thought of as an envelope, similar to an Attack-Delay-Sustain-Release (ADSR) envelope.

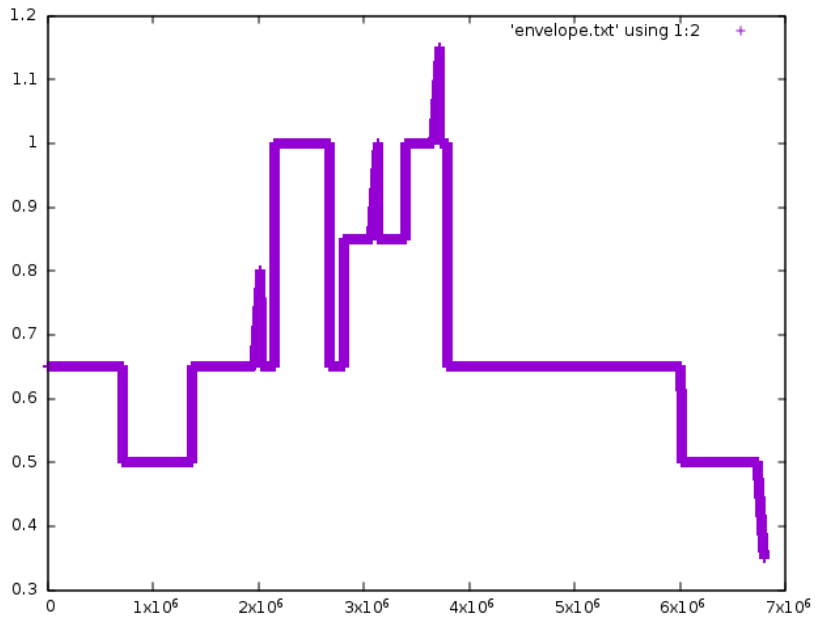
Example usage:

> $ python applyDynamicsToWAV.py --inxml song.xml --inwav song.wav --outenvelope envelope.txt >/dev/null

This write the envelope data to the file `envelope.txt` which can be plotted using **gnuplot**. First, run **gnuplot** and at the prompt type:

> gnuplot> plot 'envelope.txt' using 1:2

You should see a graph that looks something like this (depending on the dynamics):



The x-axis is the sample number (i.e. time) and the y-axis is the amplitude (i.e. volume).

printDynamics.py

This will extract and print the dynamics tags (like p, mp, f) in an XML file.

```
usage: printDynamics.py [-h] [xmlfile]

positional arguments:
  xmlfile     name of the xml file or - for stdin

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python printDynamics.py input.xml

6 Staccato

For staccato notes, the `shortenStaccatoNotes.py` will halve the length of notes labelled as staccato.

shortenStaccatoNotes.py

This will take all notes that are labelled as staccato, make it so that their length is halved, and followed by a rest of the same length.

```
usage: shortenStaccatoNotes.py [-h] [xmlfile]

positional arguments:
  xmlfile     name of the xml file or - for stdin

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python shortenStaccatoNotes.py input.xml

printStaccatoNotes.py

This will extract and print the stacatto note tags in an XML file.

```
usage: printStaccatoNotes.py [-h] [xmlfile]

positional arguments:
  xmlfile     name of the xml file or - for stdin

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python printStaccatoNotes.py input.xml

7 Replacing Lyrics

printLyrics.py

This will extract and print the lyric tags in an XML file.

```
usage: printLyrics.py [-h] [xmlfile]

positional arguments:
  xmlfile     name of the xml file or - for stdin

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python printLyrics.py input.xml

replaceLyrics.py

This will replace all the lyrics text with the specified string.

```
usage: replaceLyrics.py [-h] xmlfile lyrics

positional arguments:
  xmlfile     name of the xml file or - for stdin
  lyrics      replace lyrics with new lyrics, syllables separated by
              whitespace, in a single argument enclosed in quotes

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python replaceLyrics.py input.xml "these are the lyrics and they will just repeat"

Changing All The Lyrics

If you wanted to change all the lyrics in the text, first you could extract the lyrics using `printLyrics.py`:

> $ python printLyrics.py Fragments.xml >lyrics.txt

The `lyrics.txt` file should look like this:

>

> a a o o a a o o a a o o a a o o oh oh no no ah oh oh no oh oh oh oh oh oh oh oh or or or or all all or or or or oh, oh, oh, oh, e e e e e e e e e e e e e e e e e e e e e e e e e e e e e e e e e a, ow, a, ow, a, ow, a, ow, oh, oh, oh, oh, or or or or or or raw, a do, a do, a do, a do, or or or or or or, or or or, or or or, end, end, end, end, here, he, he, he, wall, wall, wall, wall, hell, hell, hell, hell, no, no, no, no, do no, do no, do no, do no, where, where, or or or, own n, war, wal low, oh, oh, hand le, hand le, head on, hand le, hand le, head on, dare he, dare he, won her, won her, done her, done her, hell done, hell done n raw howl, we warned, we'll drone n n n we'll drone n n n we'll drone n n n we'll drone n n n we'll drone n n n we'll drone n n n o o a a o o o o a a o o oh oh no no ah oh oh no no no no no no no no no no no no no or or or or or or all or all or or or or oh oh, oh, oh, e e e e e e e e e e e e e e e e e e e e e e e e e e e e e e e e e a, ow, a, ow, a, ow, a, ow, o o, o o, o o, o o, or, or, or or, or, or or, or, or or, or, or oo, oo, oo, oo, oo, oo, oo, oo, oo, oo, oo, oo, or or or or or or, or or or, or or or, e e e e he, here, he, here, or all, or all, or all, or all, e ell, e ell, e ell, e ell, no, no, no, no, no, no, no, no, air, air, air, air, or, a lone n, or, d low, oh, oh, and, and, don, and, and, don, e air, ee ee, e air, ee ee, were no, were no, no where, no where, now here, now here, we ee ee war or or we ee ee war or or or we warned, we warned, we warned, we'll drone n wan der hole, no where lad, no where lad, no where lad, we'll drone n n n we'll drone n n n d d d d d d d d he he he he he he he he or or or or war, war, war, war, woe, woe, woe, woe, e e e e e e e e end, end, end, end, a, ow a, ow a, ow a, ow low, low, low, low, raw, raw, raw, raw, do, do, do, do, law, law, law, law, rend, rend, rend, rend, he, he, here, here, or, all, or, all, or, all, or, all, e ell, e ell, e ell, e ell, oh oh oh oh do, do, do, do, air, air, air, air, or, own n, or oh, oh, oh, oh, oh, oh, and, and, o, o, o, o, o, on, and, and, o, o, o, o, o, on, air, ee, air, ee, er, er, er, oh, oh, oh, er, er, er, oh, oh, oh, no, e air, no, e air, ow, ere, ow, ere, we'll warn, we'll warn, or d we warned, we warned, we'll drone n n n we'll drone n n n we'll drone n n n we'll drone n n n we'll drone n n n n n n how, how, how, how, n, n, do, law, n, n, wall, wall, wall, wall, hell, oh oh oh oh nn, where, where, whore, lone n, warn d, n, n, n, dare he, dare he, her, her, er, er, her, her, er, er, e e ell, u u un, e e ell, u u un n or or or or or a a a a a owl, we warned, we'll drone n n n we'll drone n n n we'll drone n n n we'll drone n n n we'll drone n n n we'll drone n n n

If you wanted to change all of the `a's` to `uh's`, then you could use this command:

> $ cat lyrics.txt | sed 's/a/uh/g' >newLyrics.txt

Finally, these new lyrics can be fed back into the XML file:

> $ python replaceLyrics.py Fragments.xml "`cat newLyrics.txt`" >FragmentsWithNewLyrics.xml

addLyrics.py

If the XML file does not have tags this script will add them.

```
usage: addLyrics.py [-h] xmlfile lyrics

positional arguments:
  xmlfile     name of the xml file or - for stdin
  lyrics      add lyrics, syllables separated by whitespace, in a single
              argument enclosed in quotes

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python addLyrics.py input.xml "these are the lyrics and they will just repeat"

8 Render All Parts Voices Chords of an XML File

renderAllPartsVoicesChords.py

This will generate a shell script to render all parts, voices, and chords in an XML file.

```
usage: renderAllPartsVoicesChords.py [-h] xmlfile outputdir

positional arguments:
  xmlfile     name of the xml file
  outputdir   name of the directory to use for the temporary files and final
              WAV file

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python renderAllPartsVoicesChords.py Fragments.xml tempdir >fragments.sh

The output `fragments.sh` will be a shell script:

```
# Output directory: tempdir
mkdir "tempdir"
#
# Rendering for the following parts:
#    {'chord': False, 'voice': '1', 'part': 'P1'}
python keepPart.py "Fragments.xml" P1 >"tempdir/P1-1.xml"
python upload.py "tempdir/P1-1.xml" "tempdir/P1-1.wav"
#    {'chord': True, 'voice': '1', 'part': 'P1'}
python keepPart.py "Fragments.xml" P1 | python useChordNotes.py >"tempdir/P1-1-chord.xml"
python upload.py "tempdir/P1-1-chord.xml" "tempdir/P1-1-chord.wav"
#    {'chord': False, 'voice': '1', 'part': 'P2'}
python keepPart.py "Fragments.xml" P2 >"tempdir/P2-1.xml"
python upload.py "tempdir/P2-1.xml" "tempdir/P2-1.wav"
#    {'chord': True, 'voice': '1', 'part': 'P2'}
python keepPart.py "Fragments.xml" P2 | python useChordNotes.py >"tempdir/P2-1-chord.xml"
python upload.py "tempdir/P2-1-chord.xml" "tempdir/P2-1-chord.wav"
#    {'chord': False, 'voice': '1', 'part': 'P3'}
python keepPart.py "Fragments.xml" P3 >"tempdir/P3-1.xml"
python upload.py "tempdir/P3-1.xml" "tempdir/P3-1.wav"
#    {'chord': True, 'voice': '1', 'part': 'P3'}
python keepPart.py "Fragments.xml" P3 | python useChordNotes.py >"tempdir/P3-1-chord.xml"
python upload.py "tempdir/P3-1-chord.xml" "tempdir/P3-1-chord.wav"
#    {'chord': False, 'voice': '1', 'part': 'P4'}
python keepPart.py "Fragments.xml" P4 >"tempdir/P4-1.xml"
python upload.py "tempdir/P4-1.xml" "tempdir/P4-1.wav"
#    {'chord': True, 'voice': '1', 'part': 'P4'}
python keepPart.py "Fragments.xml" P4 | python useChordNotes.py >"tempdir/P4-1-chord.xml"
python upload.py "tempdir/P4-1-chord.xml" "tempdir/P4-1-chord.wav"
# Mix
sox -m "tempdir/P1-1.wav" "tempdir/P1-1-chord.wav" "tempdir/P2-1.wav" "tempdir/P2-1-chord.wav"
"tempdir/P3-1.wav" "tempdir/P3-1-chord.wav" "tempdir/P4-1.wav" "tempdir/P4-1-chord.wav" "tempdir/mix.wav"
sox "tempdir/mix.wav" "tempdir/norm.wav" gain -n
sox "tempdir/norm.wav" "tempdir/reverb.wav" reverb
python soxCompand.py "tempdir/reverb.wav" "tempdir/output.wav"
```

You can tweak this script to your liking. The final output will be `tempdir/output.wav`.

A Misc

These are miscellaneous scripts.

printNotes.py

This will extract and print the note/pitch tags in an XML file.

```
usage: printNotes.py [-h] [xmlfile]

positional arguments:
  xmlfile     name of the xml file or - for stdin

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

> $ python printNotes.py input.xml

replaceWithHarmony.py

This will shift the pitch of all the notes up or down N semitones or half steps (which ever you want to call it). There are 12 semitones in an octave.

```
usage: replaceWithHarmony.py [-h] xmlfile halfsteps

positional arguments:
  xmlfile     name of the xml file or - for stdin
  halfsteps   number of half steps +/-

optional arguments:
  -h, --help  show this help message and exit
```

Example usage:

This will shift it up 4 half steps.

> $ python replaceWithHarmony input.xml 4

This will shift it down 4 half steps.

> $ python replaceWithHarmony input.xml -4

generateStochasticSong.py

This will generate an stochastic song based on the name and lyrics given in the arguments. Make sure you have enough lyrics for a song (about 60 syllables should be enough). If there are unrecognized words you will have to add them to the syllable dictionary in the script.

```
usage: generateStochasticSong.py [-h] [--tempo TEMPO]
                                 songname lyricsfile [scale]

positional arguments:
  songname       name of the song
```

```
  lyricsfile     name of the lyrics file
  scale          'major', 'minor', or 'blues', default is 'major'

optional arguments:
  -h, --help     show this help message and exit
  --tempo TEMPO  tempo, default is 700
```

Example usage:

First generate some lyrics using the weather script:

> $ python retrieveWeatherData.py London >lyrics.txt

Then use those lyrics to generate the song:

> $ python generateStochasticSong.py London lyrics.txt major >london.xml

It sounds better if you turn the volume all the way down to -11.