Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. doi: 10.18637/jss.v000.i00

Metacommunicative Signal Engineering

Anneris Rodriguez | alr4559
Aerospace Department,
University of Texas at Austin

Akhil Sadam | as97822
Aerospace Department,
University of Texas at Austin

David Ventura Diaz | dv9486
Aerospace Department,
University of Texas at Austin

Abstract

A containerized package consisting of a Flask application, data persistence using a Redis database, and a Kubernetes cluster for performing asynchronous jobs. The purpose of this project is to study the science of communication, focusing on the resurging use of music as a prominent contemporary tool for communicating. This application supplies the user with a database that stores song data in multiple formats, as well as tools that allow the analysis of one or multiple song data objects.

Keywords: compose, telecommunications, music, theory, linear, algebra, Docker, Flask, Kubernetes, Python3, musicpy, R.

1. Motivation

Message interpretation has long been a difficulty. In stressful situations like those encountered onboard the ISS (International Space Station), or locations with poor connectivity, this problem becomes much more critical. In extreme weather conditions, such as when mountaineering, hyperventilation prevention and similar lifesaving techniques rely primarily on non-verbal communication, on the interpretation rather than the actual message.

Most communication protocols, however, ignore non-verbal and metacommunicative messaging, and strictly focus on the actual message. We seek to alleviate this need by engineering communication that provides interpretive guidelines for messages which may otherwise induce opposite or inadequate responses by the receiving party.

Considering phone calls are the most common method for long distance communication, we primarily restrict the metacommunicative aspect of communication to pitch, timbre, and rhythmic changes in speech. Some study has been conducted previously on these aspects: for example, persons onboard the ISS communicate with a single person on Earth, and this relatively constant supposedly induces stability.
We instead consider this problem from the perspective of music theory, a well-studied framework for pitch and rhythm with respect to emotion, that has not yet been applied here.

An initial step in the form of an application for chord-progression analysis will be taken in this paper. First, we will describe the basic functionality of this application.

2. Implementation & Functionality

This application makes use of a graphical interface in conjunction with programming interface routes that are typed into the address bar. To navigate to pages that show or return certain data, the user must type the proper route address into the address bar (or click on the link to that route). From there, the page can be interacted with graphically.

Each route contains a set of methods that perform unique functions within the application. In general, most routes will look like this:

https://isp-proxy.tacc.utexas.edu/as_tacc-2/<routeName>/<variable>

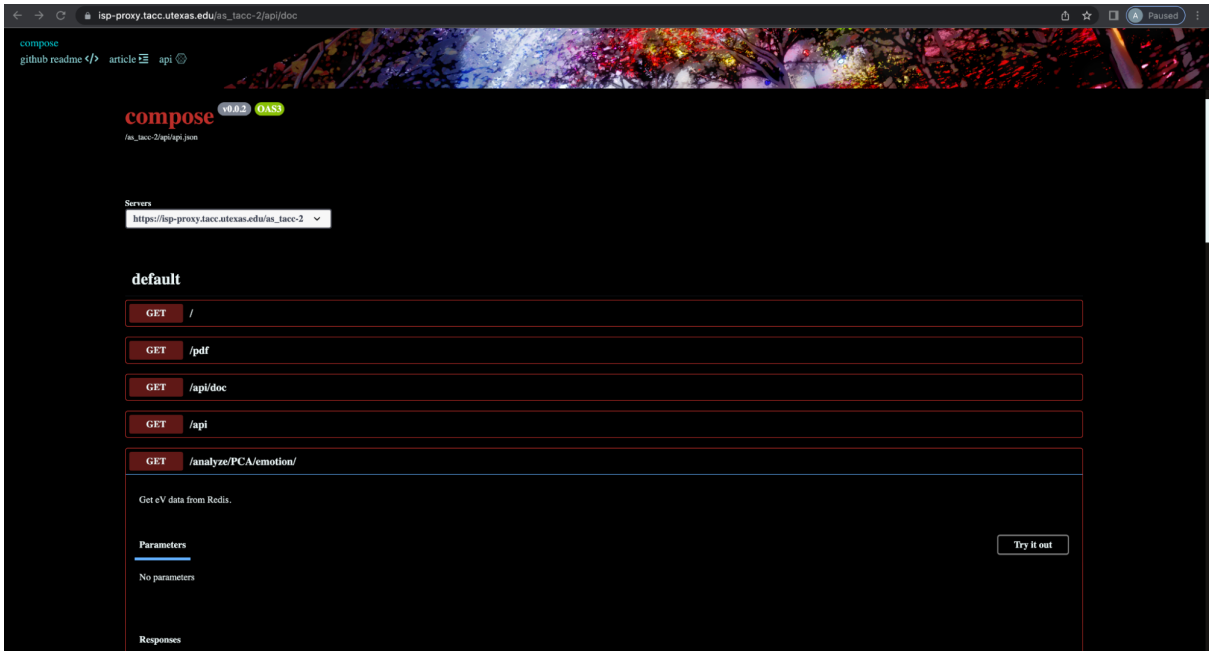
Where the *<routeName>* section contains one of the route methods written below, and the _<variable> _section, if applicable, contains a variable as needed by the route (usually, this variable is the song ID). It is important to note that a) no angle brackets should be written into the address, and b) only one slash character is necessary between each field.

2.1. Methods

Most routes and methods available in this application interact with the database to allow the user to create, read, update or delete data. This section provides a detailed description of each supported route.

Routes:

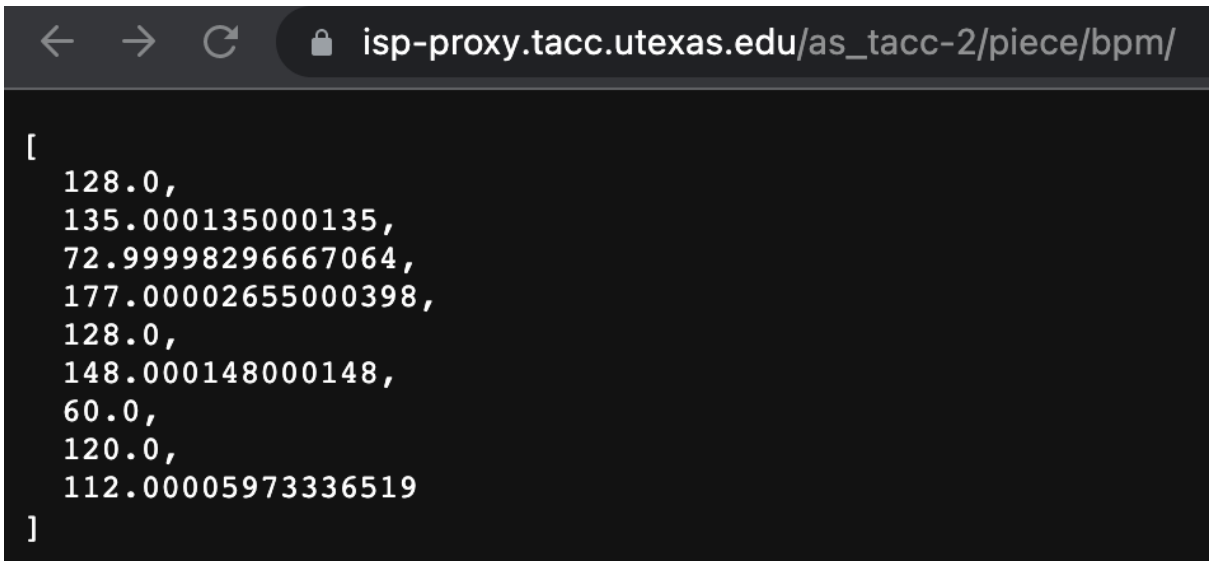
- /api/doc
  - A useful page to look at for a first-time user of this application. This page will show an interactable list of all callable routes within the application. When clicked on, each route will open a menu that displays details on the use of that route and a sample output.



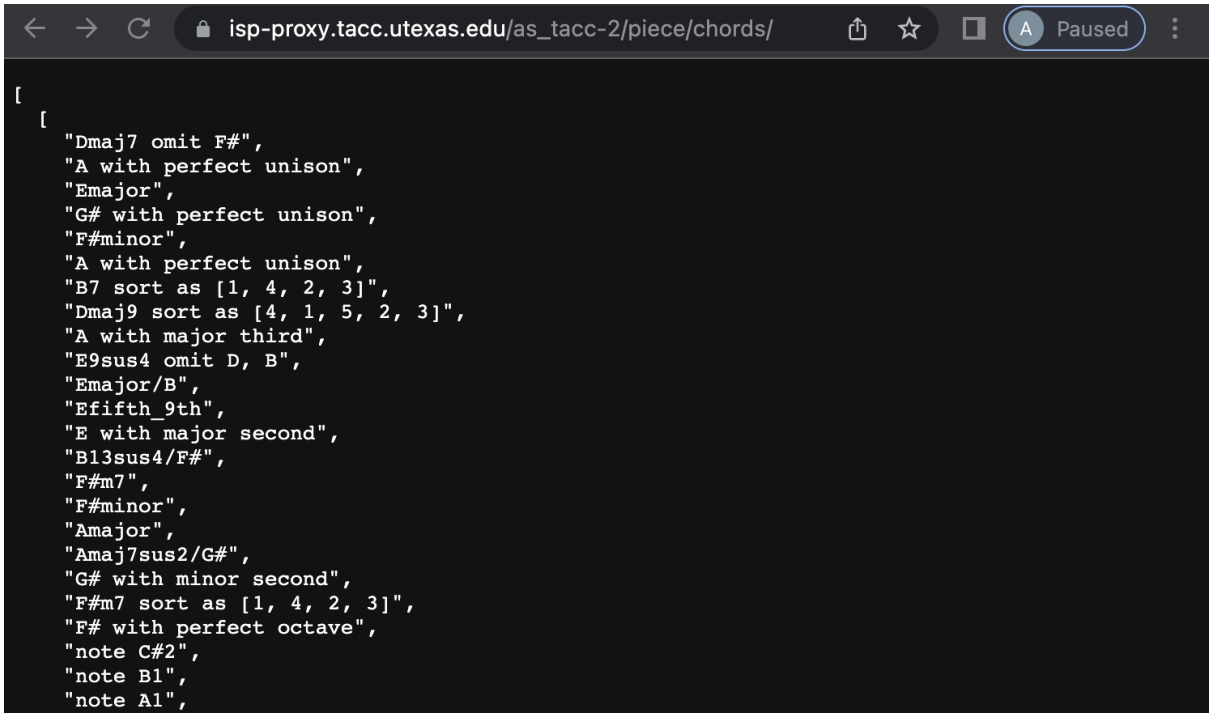
- /piece
  - The piece route returns a list of all songs that have been stored within the database in JSON format. Specific details on each song will also be listed, including the song's ID, name, and music data.

```
[davidvd@isp02 fp-trying-stuff]$ curl localhost:5035/piece/
[
  {
    "name": "Animenz_-_Crying_for_Rain",
    "type": "0",
    "chd": "-",
    "bars": "134.94791666666668",
    "bpm": "128.0",
    "time": "253.027s"
  },
  {
    "name": "Animenz_-_Unravel",
    "type": "0",
    "chd": "-",
    "bars": "131.19531250000063",
    "bpm": "135.000135000135",
    "time": "233.236s"
  },
  {
    "name": "Beethoven_-_Moonlight_Sonata_1stMvt",
    "type": "0",
    "chd": "-",
    "bars": "100.5",
    "bpm": "72.99998296667064",
    "time": "330.411s"
  },
```

- /piece/<songID>
  - This route returns individual song data. Given a song ID, it will return all data in the database that is stored with that particular song. Similar to the previously listed route.
- /piece/bpm
  - This route returns a list of the bpm (beats per minute) of every song within the database.

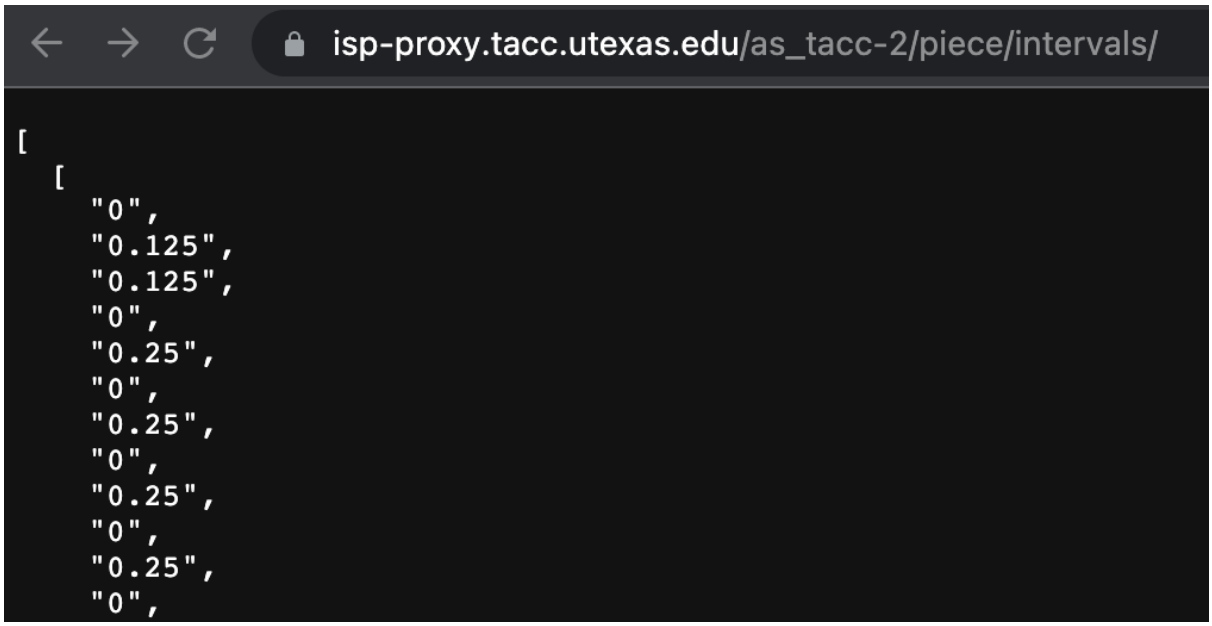


- /piece/bpm/<songID>
  - Given a song ID, this route will return the bpm of the requested song in JSON format. Similar to the previously listed route.
- /piece/chords
  - This method returns a JSON list of each chord and note used within the database. The chords and notes are separated by song in a list. Therefore, the returned object is a list of lists containing music data.

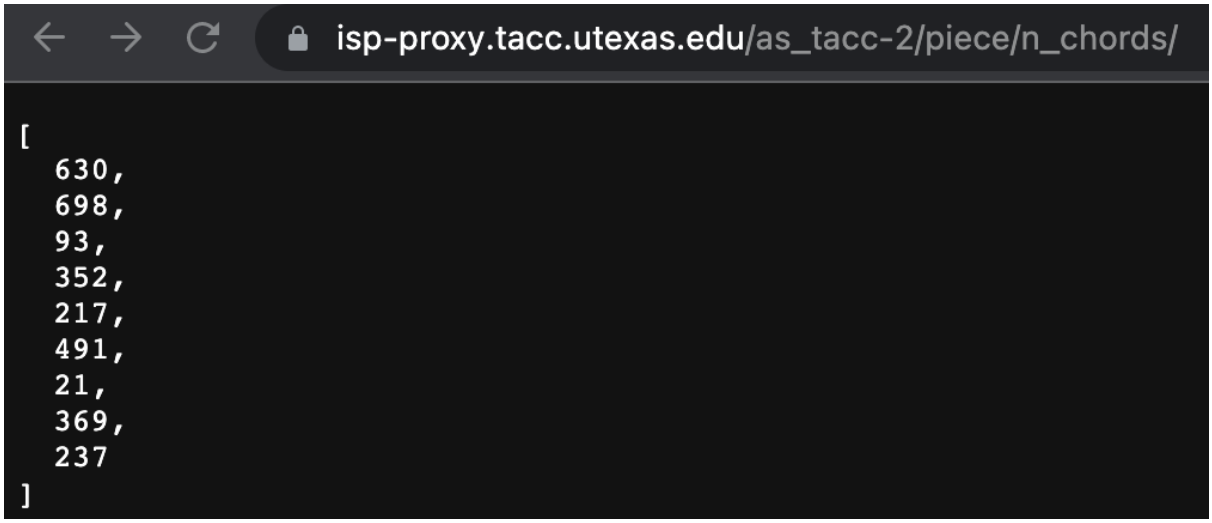


- /piece/chords/<songID>
  - This route returns a JSON list of every chord and note that is stored for the requested song in the database. The list will store all music data in the order that it appears in the song. Similar to the previously listed route.
- /piece/intervals

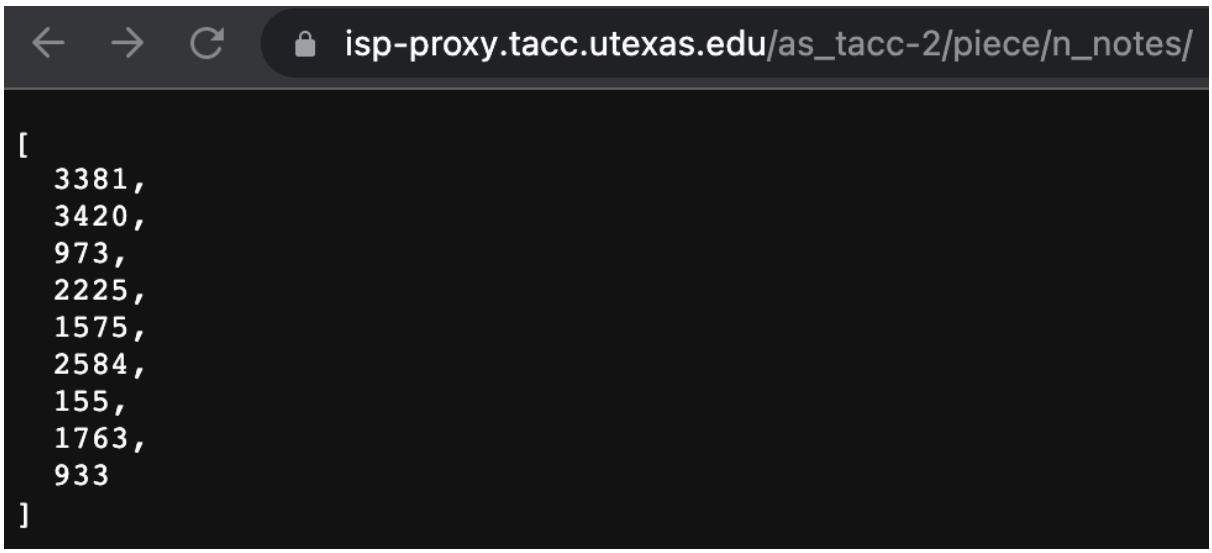
- When called, this route returns a JSON list containing the time interval data of each chord or note that is stored for every song in the database. The returned object is a list of lists holding the information for each song in order of song ID.



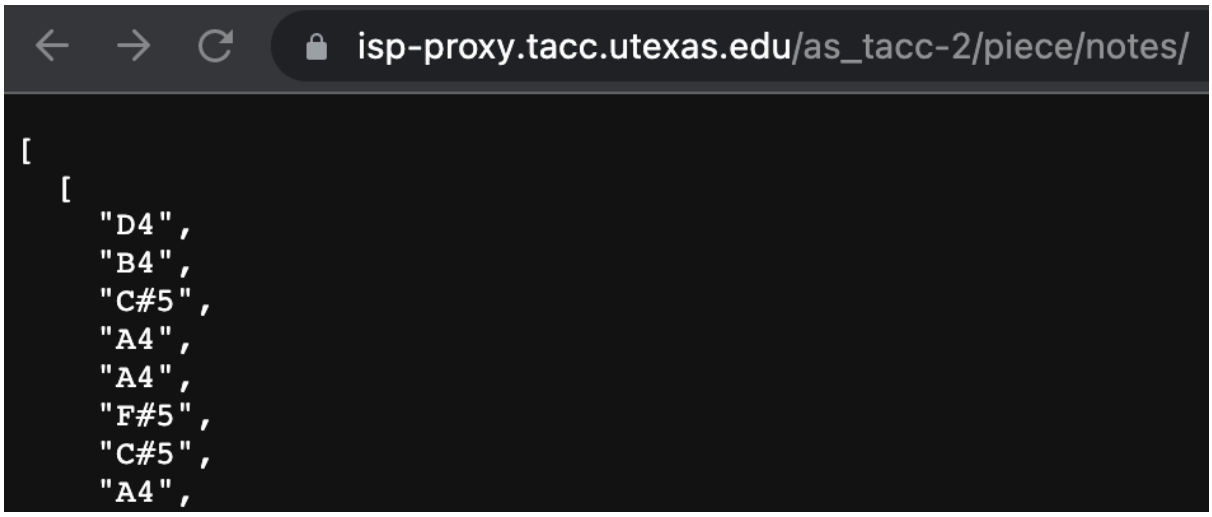
- /piece/intervals/<songID>
  - This route returns interval data of the requested song as a JSON list. The note intervals within the list are stored in the order they appear in the song. Similar to the previously listed route.
- /piece/n_chords
  - Returns a JSON list of the number of chords each song in the database contains.



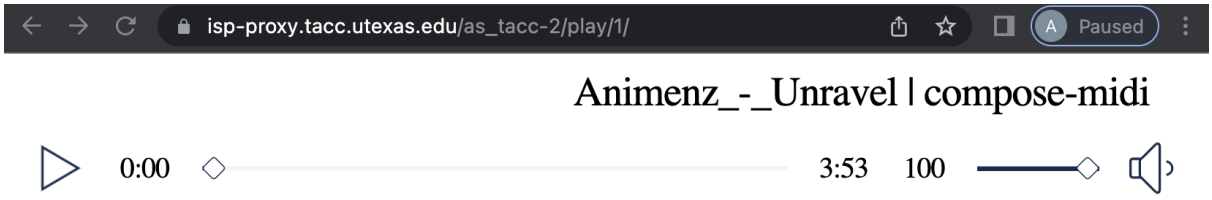
- /piece/n_chords/<songID>
  - Returns the number of chords the requested song contains. Similar to the previously listed route.
- /piece/n_notes
  - Returns a JSON list of the number of notes each song in the database contains.



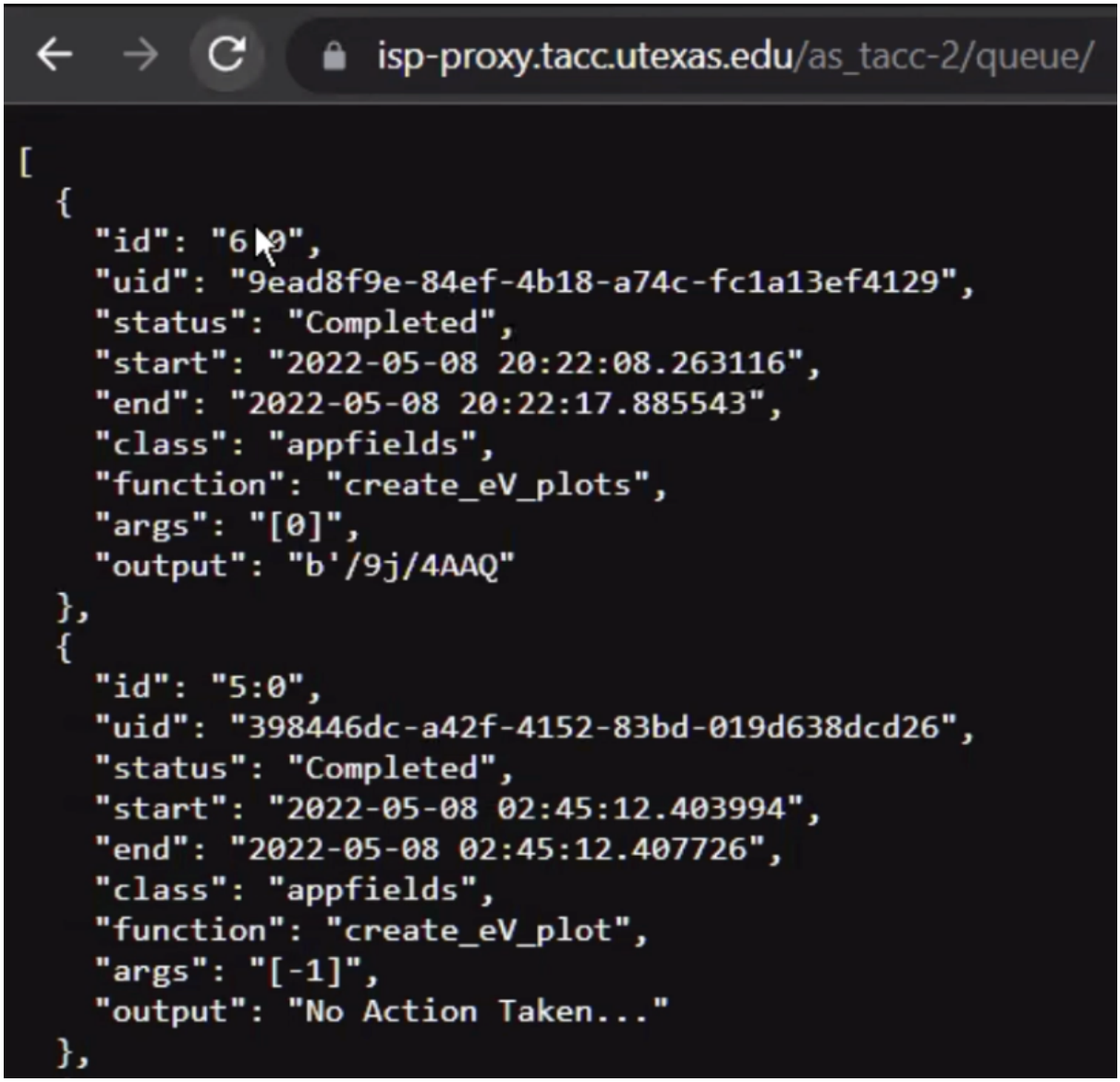
- /piece/n_notes/<songID>
  - Returns the number of notes that the requested song contains. Similar to the previously listed route.
- /piece/notes
  - This route returns a JSON list containing a list of the notes for each song in the database.



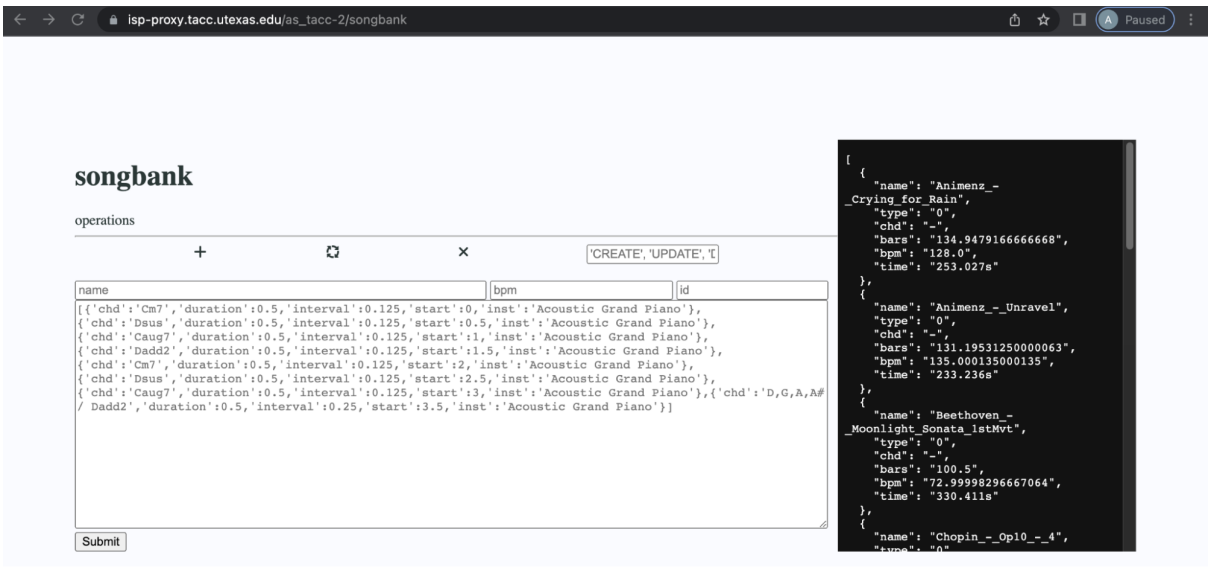
- /piece/notes/<songID>
  - Returns a list of all the notes that the requested song contains. Similar to the previously listed route.
- /play/<songID>
  - Reroutes the user to a page containing a media player that can play the requested song.



- /queue
  - The queue route returns a list of all the jobs that have been requested in JSON format. The list includes specific information regarding each individual job, including request ID, completion status, and job type.



- /songbank
  - This page allows the user to create, update and delete song data from the database. It contains fields for entering song data and a list of all the songs currently in the database.
    - CREATE: Fill out the song name, bpm, ID, and chord list fields. Make sure the mode is set to 'CREATE'. Then submit the form.
    - UPDATE: Fill out the song data fields as you wish to update them for a given song. Type the song ID of the song you wish to update. Then submit the form.
    - DELETE: Type the ID of the song you wish to delete. Submit the form.

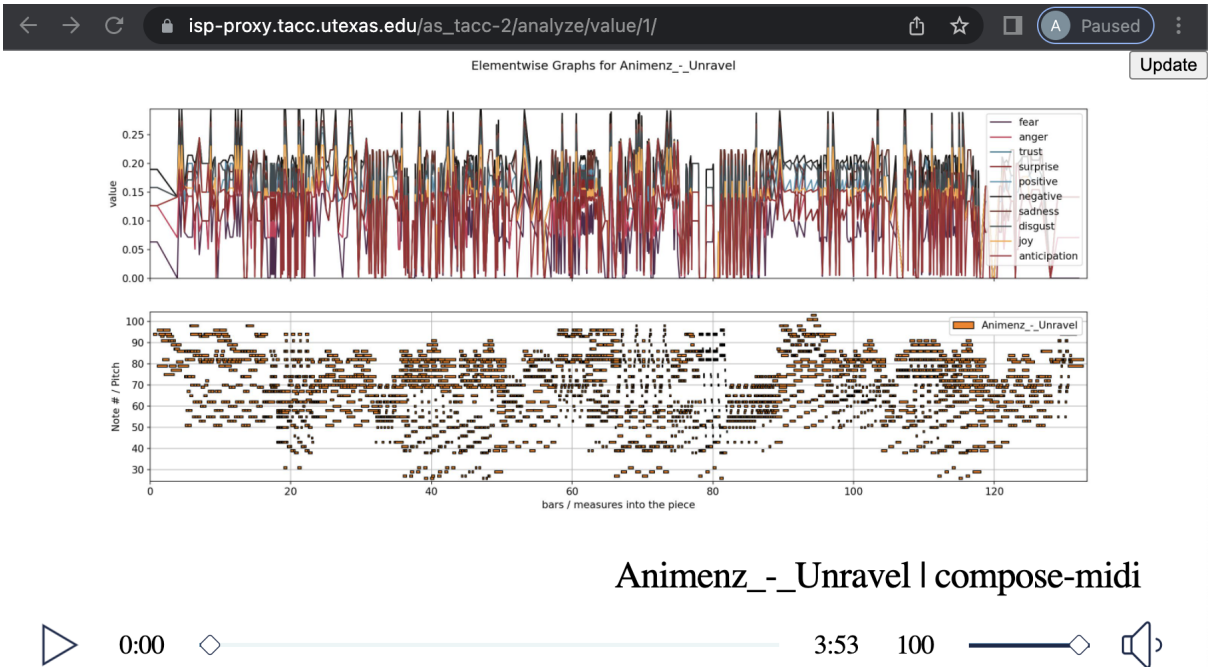


Analysis

Certain specialized routes and methods in this application allow for the analysis of the data within the database. Below are the analysis routes supported by this application, including routes that return graphical interpretations of emotional value in a song, or eV.

Routes:

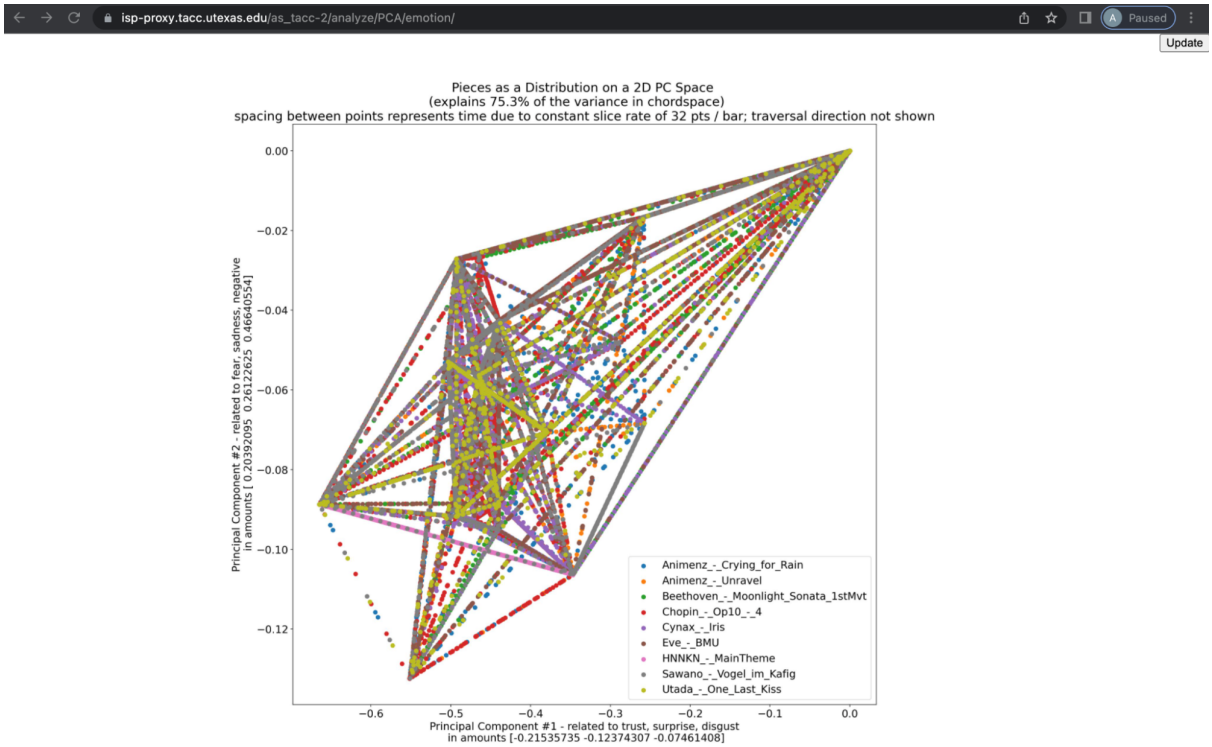
- /analyze/value/<songID>
    - Methods:
        - ‘GET’: returns an eV plot of the requested song.
        - ‘POST’: updates the eV plot of the requested song.



This route opens a page that shows the emotional values generated by the musical structure of the requested song over its course. Below this graph, sharing the same x-axis, is a depiction of the requested song as notes over time. In the top right corner of the page, there is an ‘update’ button

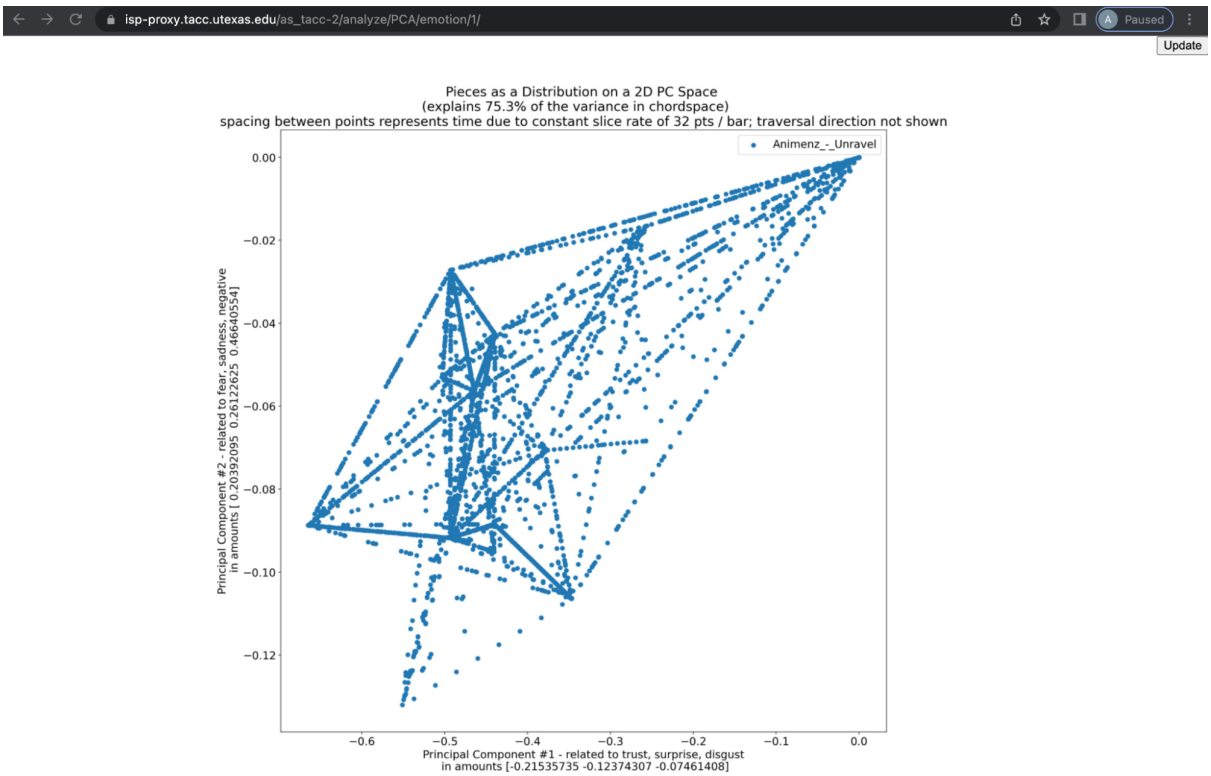
that updates the graph with the most recent version of the given song data when clicked. This page also contains a media player that can play the requested song.

- /analyze/PCA/emotion
  - Methods:
    - 'GET': returns eV data of all stored songs on a Principal Component Analysis graph.
    - 'POST': updates the eV plot.



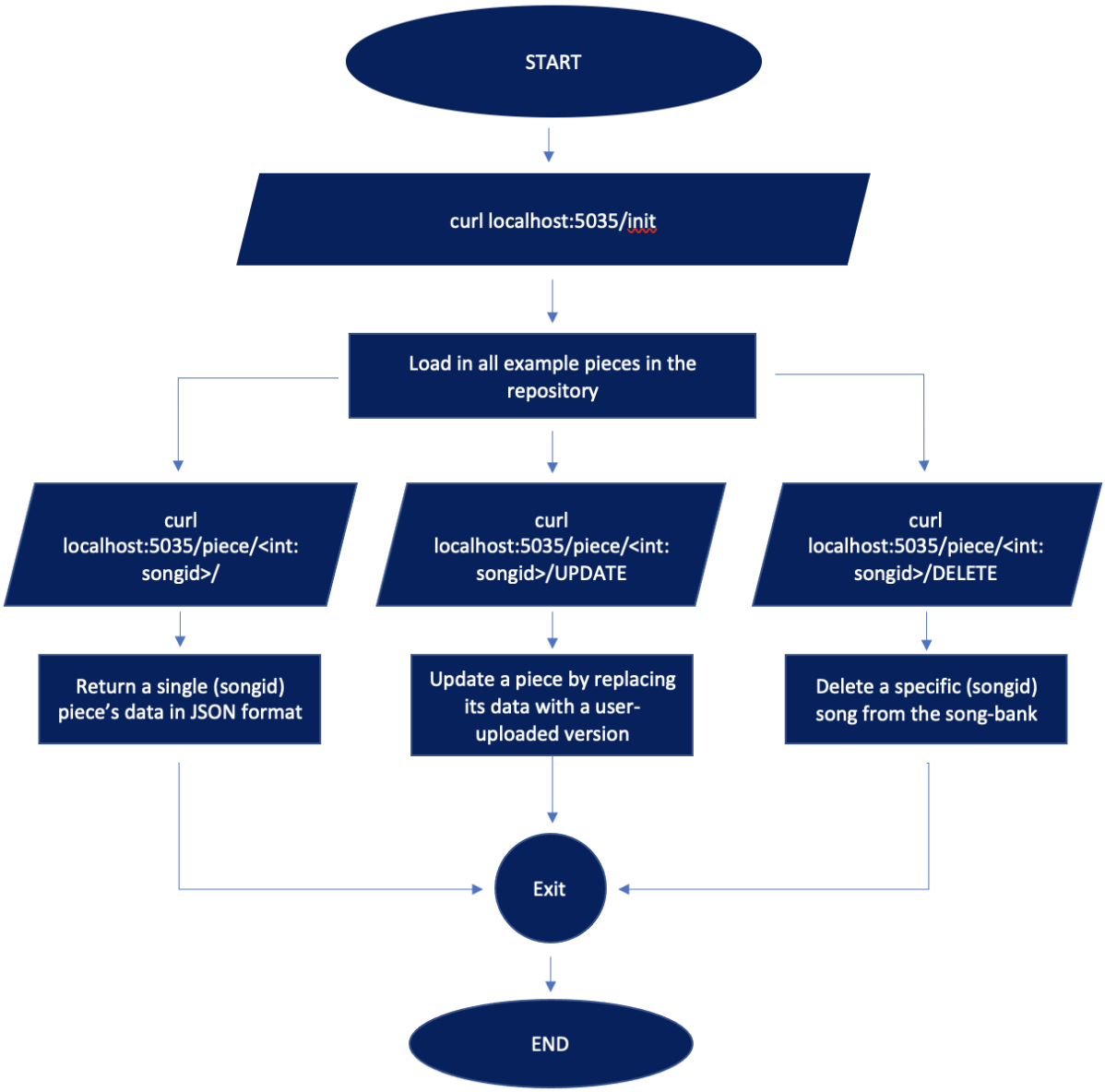
Returns a Principal Component Analysis graph of the calculated emotional values of every song stored in the database. The principal component axes are instead based upon the available chords, and so represent the entire manipulable emotional space. The two most relevant such axes have been selected for plotting, and since each axis is a linear combination of the emotional value parameters, the relationship is also given in the axes labels.

- /analyze/PCA/emotion/<songID>
  - Methods:
    - 'GET': returns eV data of the requested song on a PCA graph.
    - 'POST': updates the eV plot.



Similar to the previous route, this route also returns an emotional value PCA graph. This graph will show the data of a single, user-requested song rather than all of the stored songs within the database.

2.2. CRUD Operations

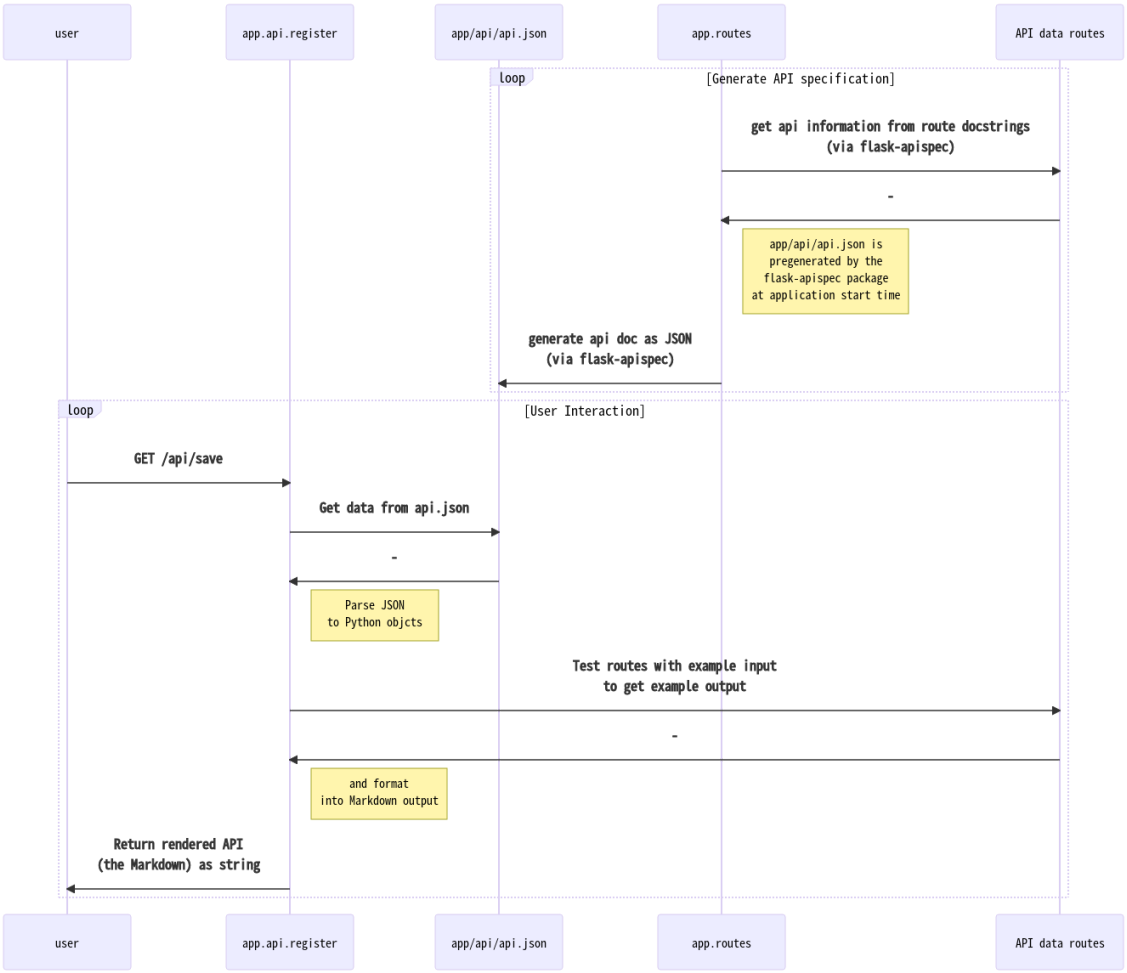


The behavioral diagram above illustrates the CRUD operations performance for the piece API portion[1]. To initialize the application, the user must curl the init route, which will create the example pieces in the repository. Once song data has been loaded, the user can read the data of a single piece by curling the /piece/<int:songid>/ route. Another route is the /piece/<int:songid>/UPDATE, which will update a piece by replacing its data with the user-uploaded data. Finally, the user is able to delete a song from the song-bank by curling /piece/<int:songid>/DELETE. Note that there are more read routes not shown in the diagram for conciseness, as well as another create route.

2.3. Documentation System

Below is a sequence diagram illustrating the process of documentation generation.

[1] https://github.com/akhilsadam/compose-dev/blob/main/app/api/piece.py



First, before the user (left-most object) performs any action, the app.routes module initializes the documentation with the help of the flask-apispec package. The app.routes module, via the flask-apispec package, parses the api information from documentation strings in the route code files (here noted as API data routes), and saves it to a file called app/api/api.json, which can be navigated to and downloaded.

Now consider the user. When the user uses the curl utility or a browser to navigate to /api/save, a HTTP GET request is sent to the app.api.register module, which requests the pre-generated api.json file (also a HTTP request), and converts that to Python objects. Particularly, the module parses out the example input commands and creates a curl command to test the route.

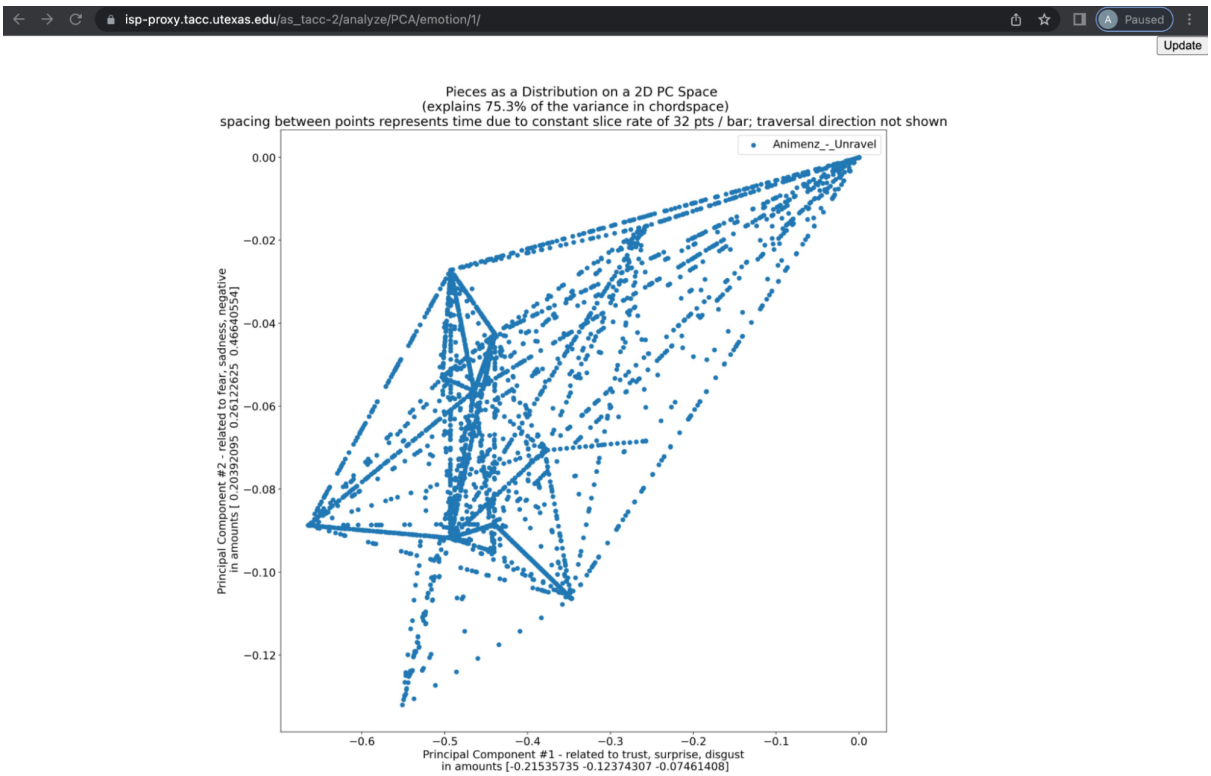
Then it calls the appropriate route and collects the example output. (In this diagram all API data routes are clustered together as API data routes for simplicity.) With the example data, the API specification is now complete. The app.api.register module now formats everything into a simple Markdown file, which is then returned to the user as a string.

2.4. Data Format

The application has a couple of songs preloaded into the system. Data is primarily presented in two distinct forms to accommodate the required functionality:

The first format in which the data was stored was as a musicpy[2] object. The musicpy object stores the beats per minute (BPM), track number and channel, instrument, tempo changes, notes, and intervals, as shown below:

[2] https://github.com/Rainbow-Dreamer/musicpy




```
127.99947916666666, tempo change to 120.0 starts at 128.0, B4, C#5, A4, C#5, F#5, A4, B4, G#4, B4,
G#5, G#4, B5, G#5, B5, G#6, F#6, F#6, E6, F#6, F#6, F#5, A5, A5, F#6, E6, F#5, A5, F#6, tempo
change to 109.99990833340972 starts at 132.5, C#6, C#7, tempo change to 100.0 starts at 133.0, A5,
F#6, E6, tempo change to 89.9999550000225 starts at 133.5, G#5, E6, tempo change to
45.000011250002814 starts at 133.75, F#6, tempo change to 89.9999550000225 starts at
133.99947916666667, F#5] with interval [0, 0.125, 0.125, 0.25, 0, 0.25, 0.25, 0.25, 0.25, 0, 0.25,
0.25, 0.25, 0.25, 0, 0.25, 0.25, 0.03802083333333333, 0.0390625, 0.42291666666666666, 0, 0, 0,
0.25, 0, 0.25, 0, 0, 0.5, 0, 0.5, 0, 0, 0, 0.25, 0, 0.25, 0, 0, 0, 0.25, 0, 0, 0.75, 0, 0,
0.041666666666666664, 0.041666666666666664, 0.041666666666666664, 0.041666666666666664,
0.041666666666666664, 0.041666666666666664, 0.0359375, 0.035416666666666666, 0.0359375,
0.035416666666666666, 0.0359375, 0.035416666666666666, 0.0359375, 0.03125, 0.03125, 0.03125,
0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125,
0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125,
0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03125, 0.03854166666666667,
0.14895833333333333, 0, 0, 0.125, 0.125, 0.125, 0, 0.125, 0.0625, 0.125, 0.125, 0.1875, 0, 0,
0.125, 0.125, 0.125, 0, 0.125, 0.0625, 0.125, 0, 0.125, 0, 0, 0, 0.1875, 0, 0.125, 0, 0.125, 0,
0.125, 0, 0.125, 0, 0.0625, 0, 0.125, 0, 0.125, 0, 0, 0, 0.25, 0, 0.1875, 0, 0.1875, 0, 0.25, 0,
0.125, 0, 0.1875, 0, 0.1875, 0, 0.25, 0.25, 0.125, 0.1875, 0.1875, 0.25, 0.25, 0.125, 0.125, 0.125,
0.125, 0.0625, 0.0625, 0.0625, 0.125, 0.125, 0.0625, 0.0625, 0.0625, 0.0625, 0.125, 0.0625, 0.0625,
0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.25, 0.125, 0.125, 0.125, 0.125, 0.034375, 0.215625, 0.25,
0.25, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 1.25, 0.0625, 0.0625, 0.0625,
0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0, 0, 0.1875, 0, 0, 0.1875, 0.034375, 0.215625, 0.25,
0.125, 0, 0, 0.125, 0.125, 0.125, 0.0625, 0.1875, 0.034375, 0.090625, 0.125, 0.125, 0, 0, 0.125,
0.0625, 0.0625, 0.125, 0.0625, 0.0625, 0.0625, 0.125, 0.125, 0.0625, 0.0625, 0.0625, 0, 0, 0.0625,
0, 0, 0.125, 0.0625, 0.125, 0.0625, 0.0625, 0.0625, 0.0625, 0.25, 0.125, 0, 0, 0.125, 0.125, 0.125,
```

The example data above is the information for 'Crying for Rain' by Animenz in musicpy object format. The data of the songs stored in such format are located in the Redis database 4. From the example above, we observe that the piece has its title, followed by the BPM. Then, it states the

track number and the channel it will be playing on. The data also specifies the instrument played for the data, and the start time of the song. Subsequently, the notes of the song are recorded with the tempo change value and the time it will start at. Finally, the song's intervals are defined since the song is a time data-series. Note that multiple tracks can be defined for a song, and they will have the same format as the example shown above.

The song's data is stored as a musicpy object to obtain information for analysis, such as the notes, tempo changes, and chords. This data format is more succinct and precise about the time data-series of the song.

The second format in which the data was stored was a JSON list of dictionaries. Each dictionary in the list stores the bars, BPM, chords, name of the song, and time, as shown below:

```
[piece] Animenz_-_Crying_for_Rain
BPM: 140.0
track 1 channel 0 Piano | instrument: Acoustic Grand Piano | start time: 0 | [tempo change to
140.00014000014 starts at 0.0, B4, C#5, A4, C#5, F#5, A4, B4, G#4, B4, G#5, G#4, B4, A4, C#5, A5,
A4, B4, D#5, B5, tempo change to 130.00013000013 starts at 3.5, F#5, A5, F#6, E5, E6, F#5, A5, F#6,
C#6, C#7, tempo change to 120.0 starts at 5.0, F#5, A5, F#6, E5, E6, tempo change to 80.0 starts at
5.5, E5, G#5, E6, tempo change to 40.0 starts at 5.75, F#5, F#6, tempo change to 80.0 starts at
5.999479166666666, tempo change to 140.00014000014 starts at 6.0, B4, C#5, E5, F#5, B5, E5, F#5,
B5, C#6, E6, F#6, B6, C#7, F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6,
F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6, F#7, F#6, F#6, F#7,
C#6, E6, C#7, B6, A6, E6, G#6, A6, G#6, E6, F#6, C#6, E6, C#7, B6, A6, E6, G#6, A6, G#6, E5, E6,
F#5, A5, C#6, F#6, C#6, C#7, B5, B6, A5, A6, G#5, G#6, A5, A6, G#5, G#6, E5, E6, F#5, A5, C#6, F#6,
F#5, F#6, C#6, C#7, F#6, F#7, tempo change to 128.0 starts at 11.875, A3, C#4, A4, C#4, A4, A3,
C#4, C#4, C#4, B3, A3, G#3, A3, A3, G#3, G#3, G#3, G#3, F#3, G#3, A3, A3, A3, B3, A3, B3, B3, A3,
B3, A3, B3, A3, B3, A3, C#4, C#4, B3, A3, A3, D#4, E4, C#4, B3, A3, G#3, A3, G#3, A3, G#3, A3, G#3,
F#3, F#3, E3, G#3, B3, E4, G#4, B4, E5, E4, A4, C#5, A5, A4, C#5, A5, B4, C#5, C#5, C#5, E4, G#4,
B4, A4, B4, A4, B4, B4, C#5, B4, A4, E4, F#4, B4, B4, A4, B4, B4, A4, B4, C#5, B4, A4, B4, A4, D#4,
F#4, B4, D#4, F#4, B4, A4, B4, B4, A4, B4, A4, C#5, C#5, E4, F#4, B4, A4, A4, D#5, E5, C#5, B4, E4,
G#4, A4, G#4, A4, G#4, A4, G#4, A4, E4, G#4, F#4, A3, C#4, E4, D4, C#4, B3, E4, E5, F#4, A4, D5,
F#5, A3, C#4, D4, F#4, A4, C#5, F#5, C#5, A5, F#5, D5, C#5, A4, F#4, C#4, A3, A4, A5, B4, B5, A4,
A5, B4, C#5, B5, A4, A5, B4, B5, A4, C#5, A5, B4, B5, A4, A5, B4, B5, C#5, F5, G#5, C#6, C#5, G#5,
C#6, G#6, C#7, F#4, F#5, A4, A5, C#5, F#5, A5, C#6, E5, F#5, B5, E5, F#5, B5, E5, F#5, B5, E5, A5,
E5, G#5, A5, G#5, E5, E5, E5, G#5, C#6, E5, B5, E5, B5, E5, B5, E5, A5, E5, G#5, A5, G#5, E5, E5,
G#5, E5, F#5, G#5, F#5, G#5, F#5, G#5, F#5, G#5, A5, C#5, E5, A5, D#5, G#5, B4, F#5, G#5, D#5, F#5,
G#5, D#5, F#5, G#5, A4, A5, A4, A5, A4, A5, E4, E5, A4, A5, C#5, C#6, B4, B5, A4, A5, A4, A5, G#4,
```

The example data above is a single dictionary entry in the list. The data of the songs stored in such format are located in the Redis database 3. We observe that the dictionary contains the number of bars the song has, the song's bpm, and chords. Note that the chords ("chd") has a value "-", this means that the chords are in musicpy object format, therefore, they are not displayed in the JSON list of dictionaries for conciseness. It also includes the name of the song and time duration. All of the songs stored in a dictionary in the list will have the same format as the example above.

The song's data is stored as a JSON list of dictionaries to obtain basic information about the song easily. From each dictionary, the bars, bpm, name, and time data can quickly be accessed without concern for details such as notes and chords.

2.5. Input Data

- The application uses midi data from several composers, listed below. In addition, self-generated datasets are also used upon user request.

```
Kawaki wo Ameku (Crying for Rain) - Minami (arr. Animenz)
Unravel - TK from Ling Tosite Sugire (arr. Animenz)
Iris - Cynax
Bokura mada Underground (We're Still Underground) - E ve
Higurashi no Naku Koro ni - Shimamiya Eiko (Main Theme)
Vogel im Kafig (Bird in a Cage) - Sawano Hiroyuki
One Last Kiss - Utada Hikaru

Moonlight Sonata (1st Movement) - Ludwig van Beethoven
Opus 10, Number 4 (Torrent) - Fryderyk Franciszek Chopin
```

- DO NOT download / distribute these midi reductions in any form or fashion (they are only available here under fair use).

3. Research & Preliminary Results

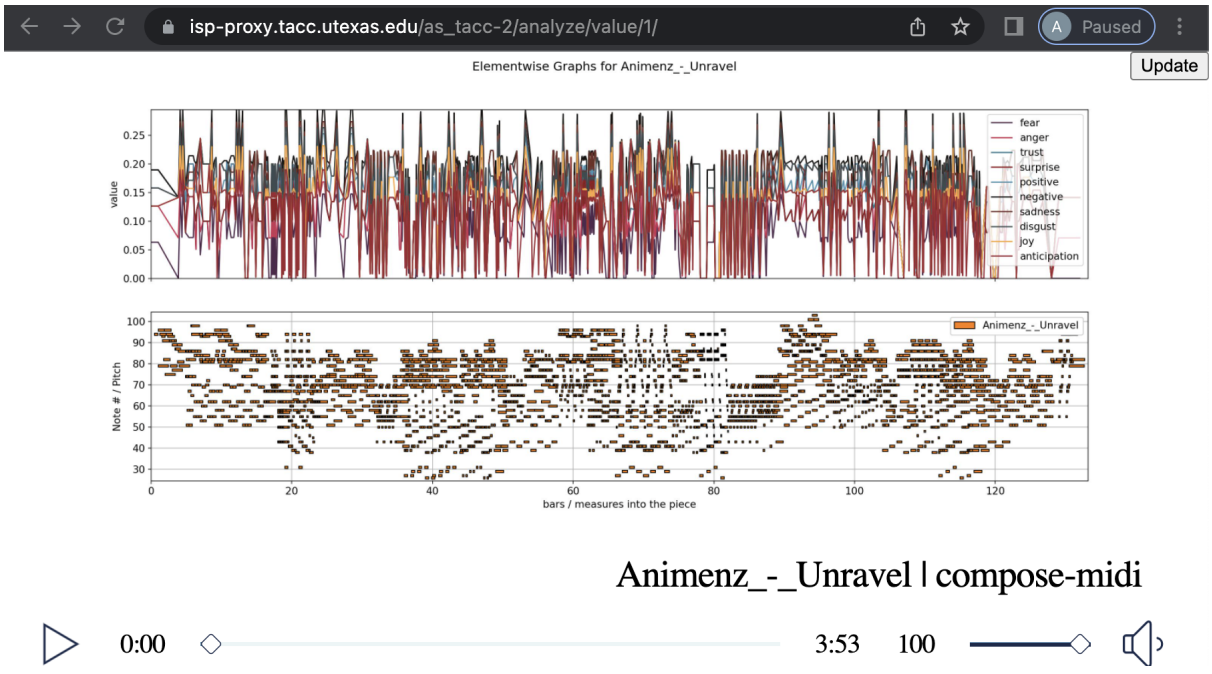
A brief summary of relavant theory and the preliminary results we derive are noted here. Note that the code for sections not depicted above is still experimental and thus should not be considered validated.

3.6. Elementwise State Representation

We start by using a vertical model for music - a track is sectioned into individual chords, which are treated separately. This is in line with the ISS rationale - that the individual character of a sound defines its emotion.

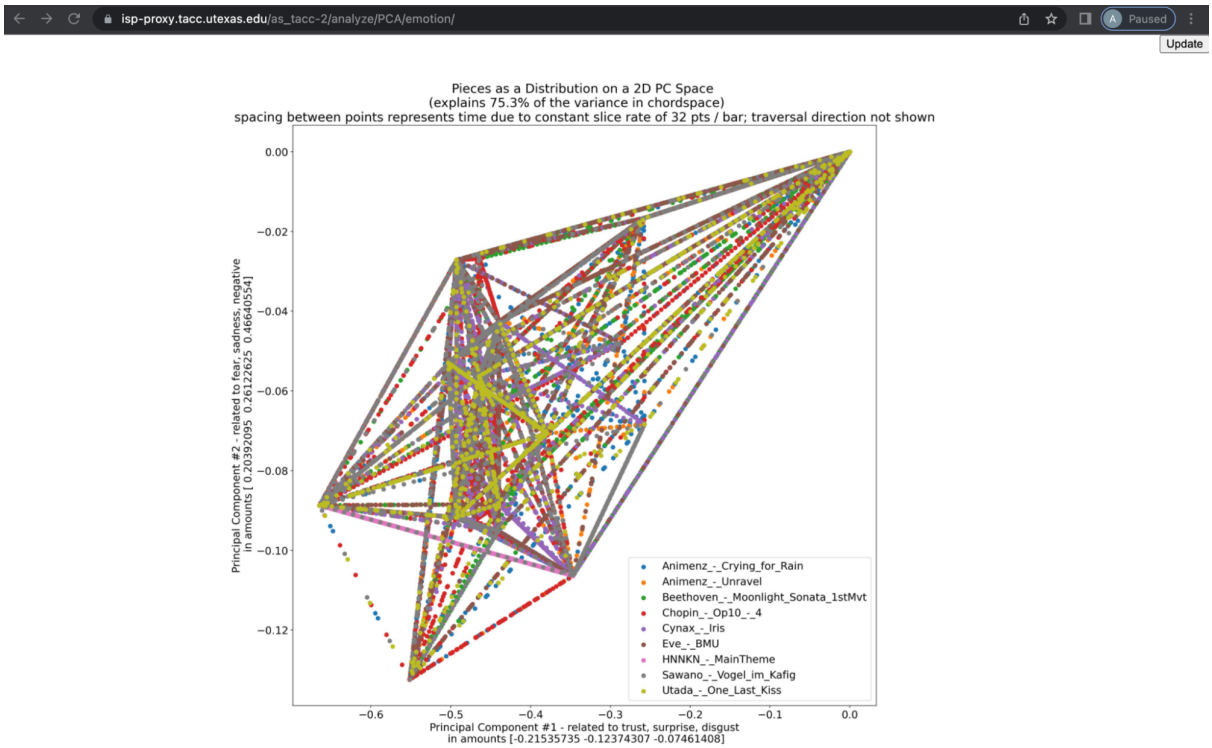
Now using Western musical theory's commonly accepted emotions for particular chords, we create a database mapping a chord to an 'emotional vector'. This is done by parsing a text description for a chord with a natural language processing algorithm called nrclex[3].

These emotional vectors can then be collated into a matrix and plotted as seen below:



Doing a PCA (Principal Component Analysis) transformation on the database to see which emotions are best represented by musical chords, and then plotting these emotional vectors on those axis, we get the following, which presumably explains most of the relations in the selected songs.

[3] https://pypi.org/project/NRCLex/



3.7. Derivative State Representation

But this is not what music theory teaches. Most songs are relatively emotionally invariant upon transposition, where the whole song is shifted by a constant pitch. So we should investigate a representation based upon the change between two or more chords.

In *The Geometry of Musical Chords*[4], by Dimitri Tymoczko, we can see that the mapping of all intervals (the distance between any two *notes*, which are discretized pitches along a twelve-tone scale) can be mapped onto a Mobius strip, as seen in the below figure.

[4] https://dmitri.mycpanel.princeton.edu/files/publications/science.pdf

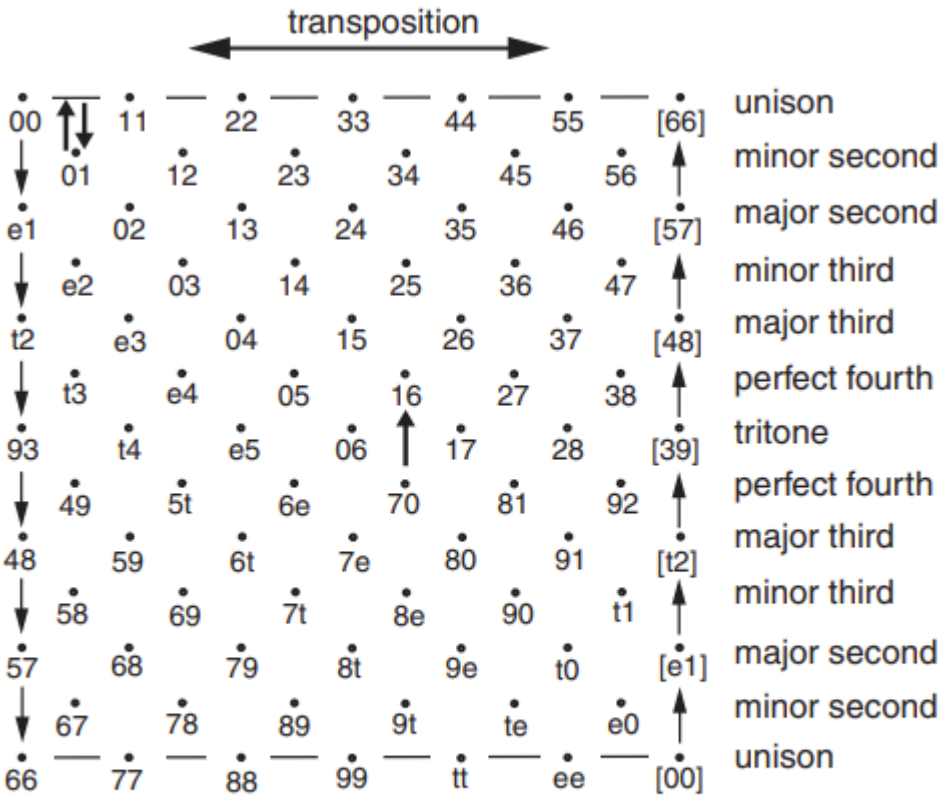


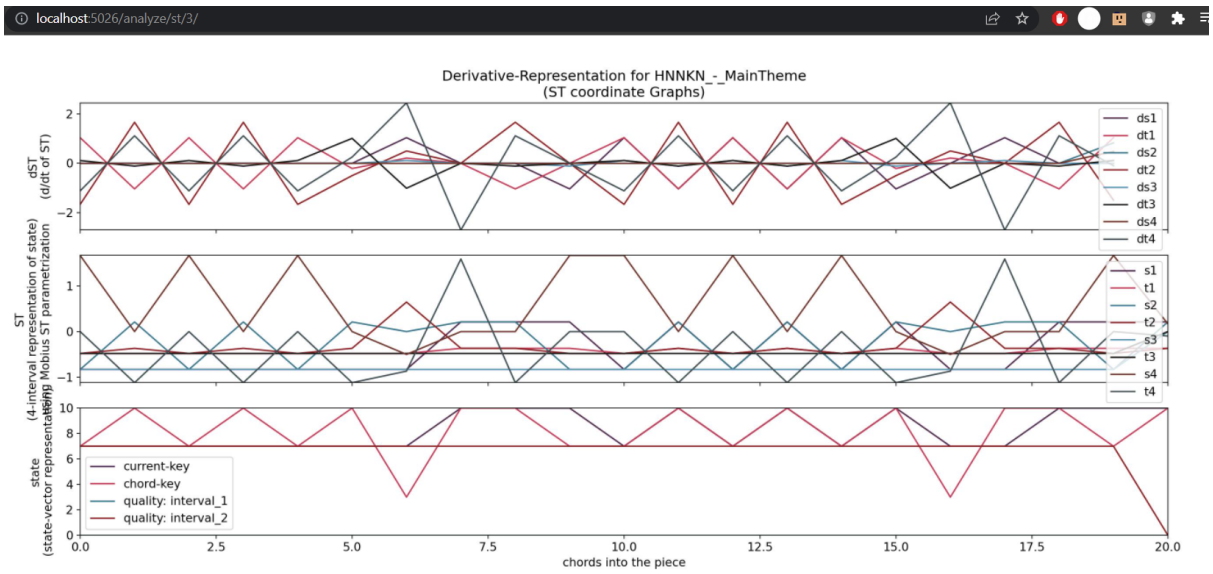
Fig. 2. The orbifold $\mathbb{T}^2/\mathcal{S}_2$. C = 0, C♯ = 1, etc., with B♭ = t, and B = e. The left edge is given a half twist and identified with the right. The voice leadings (C, D♭) → (D♭, C) and (C, G) → (C♯, F♯) are shown; the first reflects off the singular boundary.

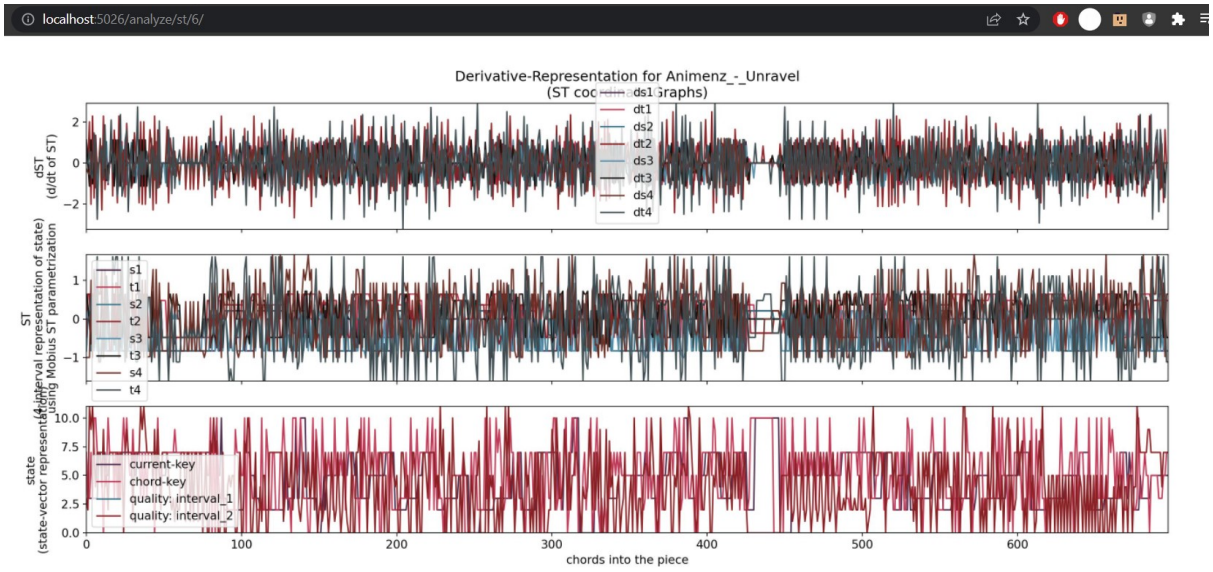
Note the top-down symmetry, and that the left boundary is identical to the inverted right boundary. So this figure can be either twisted into a Mobius strip, or converted to a toroidal surface mapping (as in the case of the Tonnetz[5]). Note further that the upper and lower boundaries are singular, so any motion between chords must reflect across that boundary.

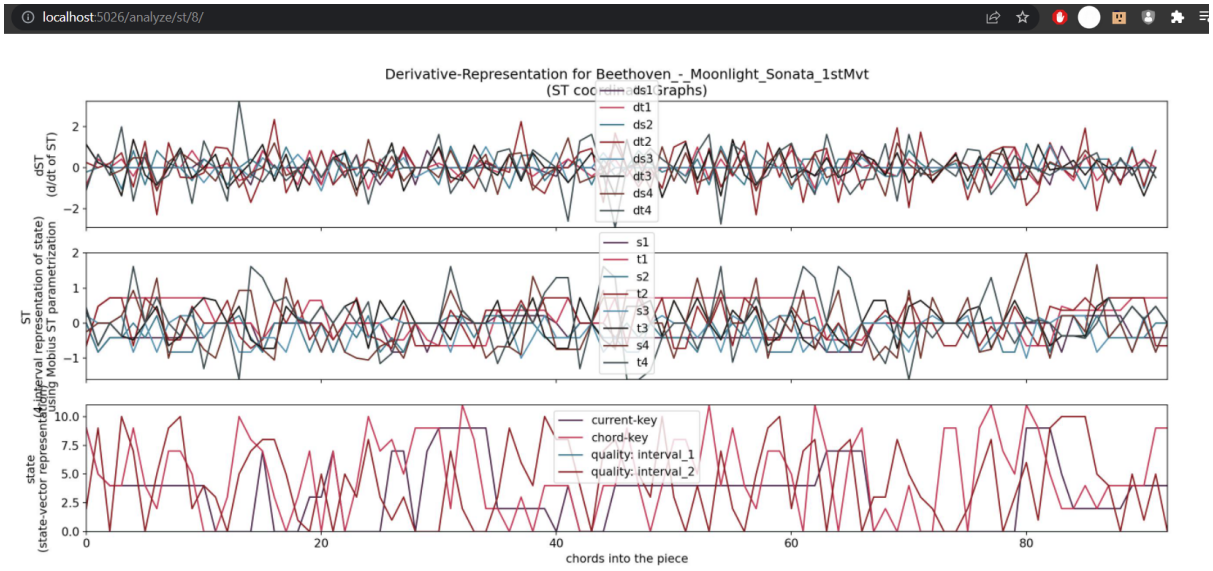
So we now have a mapping between note pitches and a vector position, so vector displacements can be derived, albeit on a non-Euclidean space. Defining the singularities with judicious use of the modulus operator, a model mapping is derived.

Using this mapping, we can construct a system of chord changes; a vector derivative denoting emotional change, if you will. We will call this dST, where ST are xy-style coordinates on the above plot, and d is used to denote that this is a time derivative. The full model will not be described here, but the following plot will illustrate the expected model result for two particular pieces.

[5] https://en.wikipedia.org/wiki/Tonnetz


localhost:5026/analyze/st/3/
Derivative-Representation for HNNKN_-_MainTheme
(ST coordinate Graphs)
dST (d/dt of ST)
ds1
dt1
ds2
dt2
ds3
dt3
ds4
dt4
ST (4-interval representation of state) using Mobius ST parametrization
s1
t1
s2
t2
s3
t3
s4
t4
state (state-vector representation of state)
current-key
chord-key
quality: interval_1
quality: interval_2
chords into the piece


localhost:5026/analyze/st/6/
Derivative-Representation for Animenz_-_Unravel
(ST coordinate Graphs)
dST (d/dt of ST)
ds1
dt1
ds2
dt2
ds3
dt3
ds4
dt4
ST (4-interval representation of state) using Mobius ST parametrization
s1
t1
s2
t2
s3
t3
s4
t4
state (state-vector representation of state)
current-key
chord-key
quality: interval_1
quality: interval_2
chords into the piece


localhost:5026/analyze/st/8/
Derivative-Representation for Beethoven_-_Moonlight_Sonata_1stMvt
(ST coordinate Graphs)
dST (d/dt of ST)
ds1
dt1
ds2
dt2
ds3
dt3
ds4
dt4
ST (4-interval representation of state) using Mobius ST parametrization
s1
t1
s2
t2
s3
t3
s4
t4
state (state-vector representation of state)
current-key
chord-key
quality: interval_1
quality: interval_2
chords into the piece

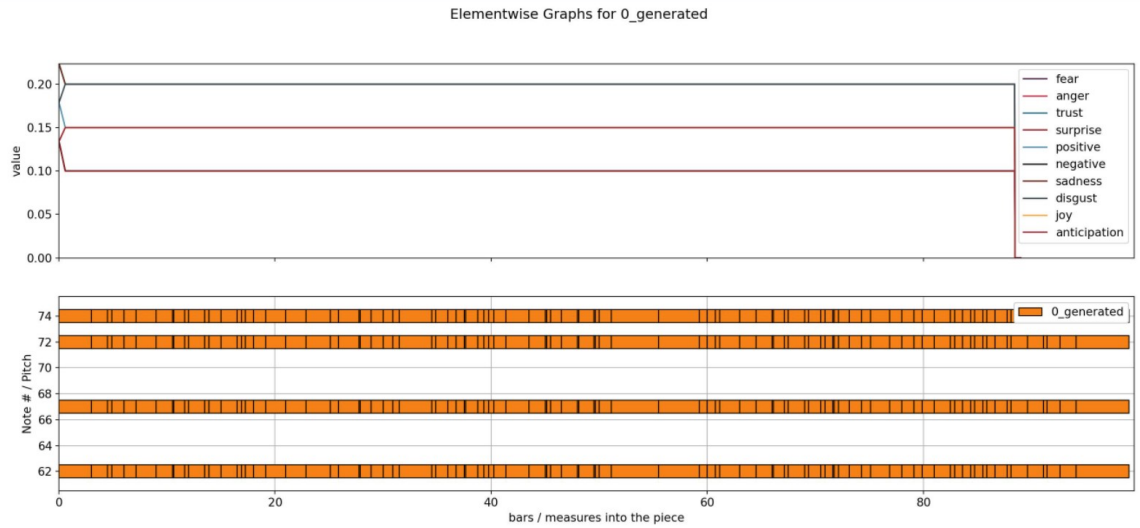
So it does seem to capture the repeating nature of these pieces, even if we lose the timescale. Assuming that chords are chosen both for timbre and musical function, and that those generally work in concert, we can use the following linear model to convert dST to the emotional value matrix λ by a simple matrix transformation.

$$\lambda * A = \partial ST \partial t$$

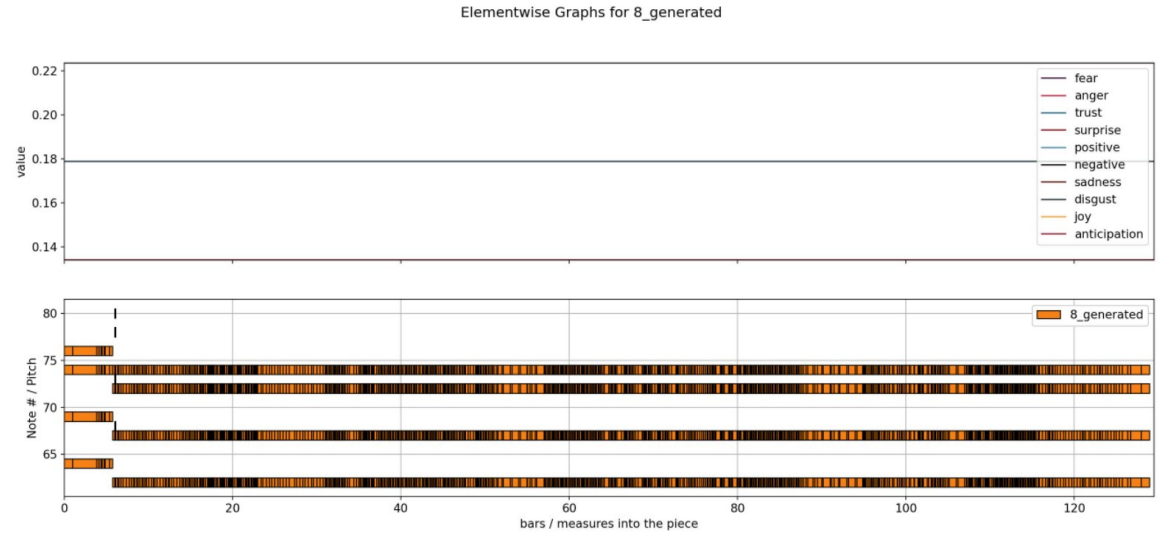
Then this can be applied backward to generate chord progressions that produce/reduce/shap a particular emotion over time.

For now, we make a sanity test: set λ to the unit vector and check that the resulting progression does not sound particularly emotional.

Doing this for Beethoven's Moonlight Sonata and Unravel, we see the following:



Beethoven



Unravel

So the flatline clearly shows that there is some merit to this investigation. We see that the derivative representation does a very good job of representing the variance (at least in these two examples).

3.8. Conclusion

So returning to the ISS example, the proper identification of voice changes seems more important than the overall timbre, which is what had primarily been studied. We hope that more study in this area can lead to appropriate usage of this metacommunication, rather than simply relying on the recipient's deduction skills.

Thank you for reading!

4. Ethical and Professional Responsibilities in Engineering Situations

When engineers intend to create new technologies, they must keep in mind the ethical and professional responsibilities that are required for any engineering process. In the development of the current application, we applied two critical engineering ethics principles as well as the proper attribution of credit. The two engineering ethics principles we employed in the design of this project were beneficence and non-maleficence, and we also properly and explicitly gave credit to the authors of work utilized in our application.

When developing any engineering system, developers must keep in mind engineering ethics principles. In the current project, the two principles used were beneficence and nonmaleficence. Such principles are defined as follows:

- **Beneficence:** The principle of beneficence refers to the obligation that an engineer has to do good, meaning to act in the benefit of the user of the product or system developed.
- **Non-maleficence:** The principle of nonmaleficence refers to an engineer's obligation to do no harm, meaning that no harm must originate from the product or system developed.

In our project, such principles were prioritized as an essential part of the design phase. The beneficence principle presents itself in the motivation for our application. The intent for developing this project was to provide an application that could positively affect the user by being able to analyze a song's emotion. This could be extremely useful when trying to determine songs that are suitable for people that require music that evokes a specific emotion, such as happiness. Furthermore, music is also a stress-reliever and mindfulness promoter. For the second principle, non-maleficence, we considered the possible uses that this application could have, and no harm could originate from this system. The application can be used to analyze songs, so it does not cause harm to the user and it cannot be employed in such a way that could produce injury.

Furthermore, one professional responsibility present in our project is proper accreditation. In various engineering projects, the use of someone else's work to develop new systems is not unusual. However, it is crucial that proper attribution of credit is given to the authors of software or data used not developed by the contributors of the current project. For the development of our project, we utilized a music programming language in python called *'musicpy'* developed by the Github user *Rainbow-Dreamer*. The Github repository can be found by following the link in the Data section of the report. Also, since we utilized songs to analyze them, we credited the artist that created each song. Each of the songs pre-loaded into the application have the name of the artist in the title of the song. By providing proper accreditation, we ensure that the users of our project can navigate the data used correctly and determine its validity.

5. Appendix

5.9. Installation Detail

The following commands are all terminal commands, and are expected to run on a Ubuntu 20.04 machine with Python3, and are written in that fashion. Mileage may vary for other systems.

From Source:

Clone the repository and then use the makefile to start the containers (make sure Docker has been installed prior).
`git clone https://github.com/akhilsadam/compose-dev`
`cd compose-dev`
`make iterate`

Now either use a browser or a curl utility to interact with the application at `https://localhost:5026/`. Further instructions are provided in the API documentation below, and in the writeup, so please refer there as necessary.

To perform integration tests, once the `make iterate` command has been run, in another terminal, type the following:
`pytest`
If no errors can be seen in terminal output, the application has passed the integration tests.

Kubernetes Deployment:

Prerequisites: Create your public urls and ports, and place them in the `home` directory in a file called `portinfo`. The style should be as follows:

```
docker port: 5026
kube port 1: 30026
kube port 2: 30126
public url 1: "https://isp-proxy.tacc.utexas.edu/as_tacc-1/"
public url 2: "https://isp-proxy.tacc.utexas.edu/as_tacc-2/"
```

Here we have given the ports as expected for the primary deployment cluster.

Navigate to your preconfigured Kubernetes cluster via terminal, and then run the following commands in the terminal.
`git clone https://github.com/akhilsadam/compose-dev`
`cd compose-dev`
`make cubeiterateT`

If you are deploying to production, do `make cubeiterate` instead of `make cubeiterateT`.

The Kubernetes services, PVCs, deployments, and pods should now be up and running.

Note we do not perform integration tests on the Kubernetes cluster, however; if that is necessary, please tweak the `/test/test_api.py` file with the following replacement.
`'http://localhost:5026/api/save'` -> `<your public url with proxy>/api/save`
Now running `pytest` in the terminal should test the application.

From Docker

As before, in a terminal, do the following:
`git clone https://github.com/akhilsadam/compose-dev`
`cd compose-dev`
`make run`

To perform integration tests, once the `make run` command has been successfully run, in another terminal, type the following:
`pytest`

5.10. REST API:

ENDPOINT: /

- Description: Get homepage HTML
- Parameters:
  - N/A
- Responses:
  - A 200 response will : Return homepage HTML
- Example: `curl -X GET http://localhost:5026/ -H "accept: application/json"`

ENDPOINT: /pdf

- Description: Get writeup HTML
- Parameters:
  - N/A
- Responses:
  - A 200 response will : Return writeup HTML
- Example: `curl -X GET http://localhost:5026/pdf -H "accept: application/json"`

ENDPOINT: /api/doc

- Description: Get API HTML
- Parameters:
  - N/A
- Responses:
  - A 200 response will : Return API HTML
- Example: `curl -X GET http://localhost:5026/api/doc -H "accept: application/json"`

ENDPOINT: /api

- Description: Get API as HTML page
- Parameters:
  - N/A
- Responses:
  - A 200 response will : Return API HTML
- Example: `curl -X GET http://localhost:5026/api -H "accept: application/json"` yields:

```
Proxy-Modified Swagger UI
```

```
deepLinking: true,
 presets: [
 SwaggerUIBundle.presets.apis,
 SwaggerUIStandalonePreset
 ],
 plugins: [
 SwaggerUIBundle.plugins.DownloadUrl
 ],
 layout: "BaseLayout"
 })
 window.ui = ui
```

ENDPOINT: `/piece/bpm/`

- Description: Get BPM data for all pieces from Redis.
- Parameters:
  - `start` : Index (int) to start iterating from in the data list. An example would be : `0`
- Responses:
  - A 200 response will : Return BPM data for a piece as JSON
- Example:
  `curl -X GET http://localhost:5026/piece/bpm/ -H "accept: application/json"`
  yields:

```
[
  72.99998296667064,
  112.00005973336519,
  177.00002655000398,
  120.0,
  128.0,
  60.0,
  128.0,
  148.000148000148,
  135.000135000135
]
```

ENDPOINT: `/piece/bpm/{songid}/`

- Description: Get BPM data for a piece from Redis.
- Parameters:
  - `songid` : Index (int) select from for the data list. An example would be : `0`
- Responses:
  - A 200 response will : Return BPM data for a piece as JSON
- Example:
  `curl -X GET http://localhost:5026/piece/bpm/0/ -H "accept: application/json"`
  yields:

```
"72.99998296667064"
```

ENDPOINT: `/piece/chords/`

- Description: Get chord data for all pieces from Redis.
- Parameters:
  - `start` : Index (int) to start iterating from in the data list. An example would be : `0`

- Responses:
  - A 200 response will : Return chord data for all pieces as a JSON list
- Example:
  `curl -X GET http://localhost:5026/piece/chords/ -H "accept: application/json"`
  yields:

```
[
  [
    "G with major second",
    "F with major sixth",
    "E with perfect octave",
    "E with minor seventh",
    "A with perfect fifth / A5 (A power chord)",
    "note E2",
    "Dmadd2/A",
    "G with major sixth",
    "G with minor seventh",
    "F with major second",
    "C with major third",
    "C with minor third",
    "note D#3",
....
    "C with major sixth",
    "Fmajor sort as [1, 3, 2]",
    "A#fifth_9th/F",
    "Gm9 sort as [1, 3, 4, 5, 2]",
    "Gm9 sort as [1, 3, 4, 5, 2]",
    "G13sus2 omit A, E",
    "D#major sort as [1, 3, 2]",
    "D#major sort as [1, 3, 2]",
    "D#maj13 omit G#",
    "F13sus4 omit D# /G",
    "Gm7 omit A#",
    "Gmadd2 sort as [1, 4, 2, 3]",
    "Gmadd2 sort as [1, 4, 2, 3]"
  ]
]
```

ENDPOINT: /piece/chords/{songid}/

- Description: Get chord data for a piece from Redis.
- Parameters:
  - `songid` : Index (int) select from for the data list. An example would be : `0`
- Responses:
  - A 200 response will : Return chord data for a piece as a JSON list
- Example:
  `curl -X GET http://localhost:5026/piece/chords/0/ -H "accept: application/json"`
  yields:

```
[
  "G with major second",
  "F with major sixth",
  "E with perfect octave",
  "E with minor seventh",
  "A with perfect fifth / A5 (A power chord)",
```

```
"G with major sixth",
  "G with minor seventh",
  "F with major second",
  "C with major third",
  "C with minor third",
  "note D#3",
  "A# with perfect fourth",
....
  "Fmajor/A",
  "B with diminished fifth",
  "Amaj7 omit C# /G#",
  "F with major sixth",
  "E with minor seventh",
  "E with minor seventh",
  "E with minor seventh",
  "Asus",
  "E with major third",
  "E with perfect octave",
  "E with perfect fourth",
  "E with minor second",
  "Aminor/E",
  "A with perfect octave"
]
```

ENDPOINT: `/piece/intervals/`

- Description: Get intervals for a piece from Redis.
- Parameters:
  - `start` : Index (int) to start iterating from in the data list. An example would be : `0`
- Responses:
  - A 200 response will : Return intervals for a piece as a JSON list
- Example:
  `curl -X GET http://localhost:5026/piece/intervals/ -H "accept: application/json"`
  yields:

```
[
  [
    "0",
    "0.125",
    "0.125",
    "0.125",
    "0.125",
    "0.125",
    "0.125",
    "0.125",
    "0.125",
    "0.125",
    "0.125",
    "0.125",
    "0.125",
....
    "0.0625",
    "0",
    "0.0625",
    "0.0625",
```

```
"0.0625",
    "0.0625",
    "0.0625",
    "0.0625",
    "0.0625",
    "1.0078125",
    "0"
  ]
]
```

ENDPOINT: `/piece/intervals/{songid}/`

- Description: Get intervals for a piece from Redis.
- Parameters:
  - `songid` : Index (int) select from for the data list. An example would be : `0`
- Responses:
  - A `200` response will : Return intervals for a piece as a JSON list
- Example:
  `curl -X GET http://localhost:5026/piece/intervals/0/ -H "accept: application/json"`
  yields:

```
[
  "0",
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0",
....
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0.125",
  "0.75",
  "0",
  "0",
  "0",
  "0",
  "0.75",
  "0",
  "1.5",
  "0"
]
```

ENDPOINT: `/piece/n_chords/`

- Description: Get the number of chords for all pieces from Redis.
- Parameters:

    - `start` : Index (int) to start iterating from in the data list. An example would be : `0`
- Responses:
    - A `200` response will : Return the number of chords for all pieces as a JSON list
- Example:
  `curl -X GET http://localhost:5026/piece/n_chords/ -H "accept: application/json"`
  yields:

```
[
  93,
  237,
  352,
  369,
  217,
  21,
  630,
  491,
  698
]
```

ENDPOINT: `/piece/n_chords/{songid}/`

- Description: Get number of chords for a piece from Redis.
- Parameters:
    - `songid` : Index (int) select from for the data list. An example would be : `0`
- Responses:
    - A `200` response will : Return number of chords for a piece as a int
- Example:
  `curl -X GET http://localhost:5026/piece/n_chords/0/ -H "accept: application/json"`
  yields:

```
93
```

ENDPOINT: `/piece/n_notes/`

- Description: Get the number of notes for all pieces from Redis.
- Parameters:
    - `start` : Index (int) to start iterating from in the data list. An example would be : `0`
- Responses:
    - A `200` response will : Return the number of notes for all pieces as a JSON list
- Example:
  `curl -X GET http://localhost:5026/piece/n_notes/ -H "accept: application/json"`
  yields:

```
[
  973,
  933,
  2225,
  1763,
  1575,
  155,
  3381,
  2584,
  3420
]
```

ENDPOINT: `/piece/n_notes/{songid}/`

- Description: Get number of notes for a piece from Redis.
- Parameters:
  - `songid` : Index (int) select from for the data list. An example would be : `0`
- Responses:
  - A `200` response will : Return number of notes for a piece as a int
- Example:
  `curl -X GET http://localhost:5026/piece/n_notes/0/ -H "accept: application/json"`
  yields:

```
973
```

ENDPOINT: `/piece/notes/`

- Description: Get the notes for all pieces from Redis.
- Parameters:
  - `start` : Index (int) to start iterating from in the data list. An example would be : `0`
- Responses:
  - A `200` response will : Return the notes for all pieces as a JSON list
- Example:
  `curl -X GET http://localhost:5026/piece/notes/ -H "accept: application/json"`
  yields:

```
[
  [
    "A2",
    "E3",
    "A3",
    "C4",
    "E3",
    "A3",
    "C4",
    "E3",
    "A3",
    "C4",
    "E3",
    "A3",
    "C4",
....
    "D6",
    "A#4",
    "A#6",
    "D5",
    "G5",
    "A#5",
    "A5",
    "A#5",
    "C6",
    "A#5",
    "A5",
    "A#5",
    "tempo change to 135.000135000135 starts at 0.0"
  ]
]
```

ENDPOINT: `/piece/notes/{songid}/`

- Description: Get notes for a piece from Redis.
- Parameters:
  - `songid` : Index (int) select from for the data list. An example would be : `0`
- Responses:
  - A 200 response will : Return notes for a piece as a JSON list
- Example:
  `curl -X GET http://localhost:5026/piece/notes/0/ -H "accept: application/json"`
  yields:


```
[
  "A2",
  "E3",
  "A3",
  "C4",
  "E3",
  "A3",
  "C4",
  "E3",
  "A3",
  "C4",
  "E3",
  "A3",
  "C4",
  "G2",
....
  "A3",
  "E3",
  "C3",
  "E3",
  "C3",
  "A2",
  "A2",
  "A3",
  "C4",
  "E4",
  "A4",
  "A2",
  "A3",
  "tempo change to 72.99998296667064 starts at 0.0"
]
```


ENDPOINT: `/flask-apispec/static/{item}`

- Description: Get SwaggerUIBundle item as necessary
- Parameters:
  - `item` : missing swagger item. An example would be : `swagger-ui.css`
- Responses:
  - A 200 response will : Return SwaggerUIBundle url redirect
- Example:
  `curl -X GET http://localhost:5026/flask-apispec/static/swagger-ui.css -H "accept: applicatio`
  yields:


```
.swagger-ui{color:#3b4151;
/\*! normalize.css v7.0.0 | MIT License | github.com/necolas/normalize.css \*/font-family:sans-ser
```

```
lack-30:hover{color:rgba(0,0,0,.3)}.swagger-ui .hover-black-20:focus,.swagger-ui .hover-black-20
```

```
/\*# sourceMappingURL=swagger-ui.css.map\*/
```

ENDPOINT: `/swagger-ui-bundle.js`

- Description: Get SwaggerUIBundle item as necessary
- Parameters:
  - N/A
- Responses:
  - A 200 response will : Return SwaggerUIBundle url redirect
- Example:
  `curl -X GET http://localhost:5026/swagger-ui-bundle.js -H "accept: application/json"`
  yields:

```
/\*! For license information please see swagger-ui-bundle.js.LICENSE.txt \*/
!function(e,t){"object"==typeof exports&&"object"==typeof module?module.exports=t():"function"==
```

ENDPOINT: `/api/save`

- Description: Get API as rendered string
- Parameters:
  - N/A
- Responses:
  - A 200 response will : Return rendered API as string
- Example:
  `curl -X GET http://localhost:5026/api/save -H "accept: application/json"`

ENDPOINT: `/analyze/PCA/emotion/`

- Description: Get eV data from Redis.
- Parameters:
  - N/A
- Responses:
  - A 200 response will : Return a 2D chordspace plot of all pieces (the eV) as HTML
- Example:
  `curl -X GET http://localhost:5026/analyze/PCA/emotion/ -H "accept: application/json"`
  yields:

```
No plot available yet; a job was submitted. Please wait a few moments and try again ... If you have done s
```

ENDPOINT: `/analyze/PCA/emotion/{songid}/`

- Description: Get eV data from Redis.
- Parameters:
  - `songid` : Index (int) of song to plot. An example would be : `0`
- Responses:
  - A 200 response will : Return a 2D chordspace plot of a single piece eV as HTML
- Example:
  `curl -X GET http://localhost:5026/analyze/PCA/emotion/0/ -H "accept: application/json"`
  yields:

```
No plot available yet; a job was submitted. Please wait a few moments and try again ... If you have done s
```

ENDPOINT: /analyze/st/{songid}/

- Description: Get st data and return plot.
- Parameters:
  - `songid` : Index (int) select from for the data list. An example would be : `0`
- Responses:
  - A `200` response will : Return a single piece (st) plot as HTML
- Example:
  `curl -X GET http://localhost:5026/analyze/st/0/ -H "accept: application/json"`
  yields:

```
No plot available yet; a job was submitted. Please wait a few moments and try again ... If you have done s
```

ENDPOINT: /analyze/value/{songid}/

- Description: Get eV data and return plot.
- Parameters:
  - `songid` : Index (int) select from for the data list. An example would be : `0`
- Responses:
  - A `200` response will : Return a single piece (eV) plot as HTML
- Example:
  `curl -X GET http://localhost:5026/analyze/value/0/ -H "accept: application/json"`
  yields:

```
No plot available yet; a job was submitted. Please wait a few moments and try again ... If you have done s
```

ENDPOINT: /init

- Description: Create examples.
- Parameters:
  - N/A
- Responses:
  - A `201` response will : Update Redis database with examples
- Example: `curl -X POST http://localhost:5026/init -H "accept: application/json"`
  yields:

```
Submitted Initialization Job.
```

ENDPOINT: /null/{songid}/

- Description: Generate a nullspace piece.
- Parameters:
  - `songid` : Index of song to generate for. An example would be : `0`
- Responses:
  - A `200` response will : Index of generated song.
- Example: `curl -X GET http://localhost:5026/null/0/ -H "accept: application/json"`
  yields:

```
{
  "args": "[0]",
  "class": "appfields",
  "end": "-",
```

```
"output": "-",
  "start": "-",
  "status": "Queued",
  "uid": "79edd6c2-29e5-4a0f-b6f6-0335cbc572ec"
}
```


ENDPOINT: `/piece/`

- Description: Get piece data from Redis.
- Parameters:
  - `start` : Index (int) to start from for the data list. An example would be : `0`
- Responses:
  - A 200 response will : Return all piece data as JSON
- Example: `curl -X GET http://localhost:5026/piece/ -H "accept: application/json"` yields:


```
[
  {
    "bars": "100.5",
    "bpm": "72.99998296667064",
    "chd": "-",
    "name": "Beethoven_-_Moonlight_Sonata_1stMvt",
    "time": "330.411s",
    "type": "0"
  },
  {
    "bars": "40.6171875",
    "bpm": "112.00005973336519",
    "chd": "-",
    "name": "Utada_-_One_Last_Kiss",
    "time": "87.037s",
....
    "bpm": "148.000148000148",
    "chd": "-",
    "name": "Eve_-_BMU",
    "time": "274.05s",
    "type": "0"
  },
  {
    "bars": "131.19531250000063",
    "bpm": "135.000135000135",
    "chd": "-",
    "name": "Animenz_-_Unravel",
    "time": "233.236s",
    "type": "0"
  }
]
```


ENDPOINT: `/piece/CREATE`

- Description: Update a song in the songbank. Complete example not available here.
- Parameters:
  - N/A
- Responses:
  - A 201 response will : Return a confirmation message stating that the update was a success.

- Example:
  `curl -X POST http://localhost:5026/piece/CREATE -H "accept: application/json"`
  yields:

```
Invalid JSON or other error. Exception: 400 Bad Request: The browser (or proxy) sent a request that this
```

ENDPOINT: /piece/{songid}/DELETE

- Description: Delete a song from the songbank.
- Parameters:
  - `songid` : Index of song to delete from songbank. An example would be : `0`
- Responses:
  - A `201` response will : Return a deletion confirmation message.
- Example:
  `curl -X POST http://localhost:5026/piece/0/DELETE -H "accept: application/json"`
  yields:

```
Successfully deleted.
```

ENDPOINT: /piece/{songid}/

- Description: Get piece data from Redis.
- Parameters:
  - `songid` : Index (int) select from for the data list. An example would be : `0`
- Responses:
  - A `200` response will : Return a single piece data as JSON
- Example:
  `curl -X GET http://localhost:5026/piece/0/ -H "accept: application/json"` yields:


```
{
  "bars": "88.77864583333334",
  "bpm": "177.00002655000398",
  "chd": "-",
  "name": "Chopin_-_Op10_-_4",
  "time": "120.378s",
  "type": "0"
}
```


ENDPOINT: /piece/{songid}/UPDATE

- Description: Update a song in the songbank. Complete example not available here.
- Parameters:
  - `songid` : Index of song to update in songbank. An example would be : `1`
- Responses:
  - A `201` response will : Return a confirmation message stating that the update was a success.
- Example:
  `curl -X POST http://localhost:5026/piece/1/UPDATE -H "accept: application/json"`
  yields:

```
Song not in database.
```

ENDPOINT: /play/{songid}/

- Description: Play a piece from Redis.

- Parameters:
  - `songid` : Index of song to play from songbank. An example would be : `0`
- Responses:
  - A 200 response will : Play a piece.
- Example: `curl -X GET http://localhost:5026/play/0/ -H "accept: application/json"` yields:

....

```
window.onscroll = () => {
 let scrollTop = window.scrollY;
 let docHeight = document.body.offsetHeight;
 let winHeight = window.innerHeight;
 let scrollPercent = scrollTop / (docHeight - winHeight);
 let scrollPercentRounded = Math.round(scrollPercent \* 100);
 document.querySelector(".avance-reculer").style.background = `linear-gradient(to right, rgb(188,
}
```

ENDPOINT: `/songbank`

- Description: Get songbank GUI.
- Parameters:
  - N/A
- Responses:
  - A 200 response will : Return songbank GUI as HTML.
- Example:
  `curl -X GET http://localhost:5026/songbank -H "accept: application/json"` yields:

```
....
 // unescape("%3Cscript type="text/javascript"%3E" +
 // " window.onscroll = () => {" +
 // "let scrollTop = window.scrollY;"+
 // "let docHeight = document.body.offsetHeight;"+
 // "let winHeight = window.innerHeight;"+
 // "let scrollPercent = scrollTop / (docHeight - winHeight);"+
 // "let scrollPercentRounded = Math.round(scrollPercent \* 100);"+
 // "document.querySelector(".avance-reculer").style.background ="+
 // "`linear-gradient(to right, rgb(188, 45, 41) ${scrollPercentRounded}%, rgb(23, 20, 18) ${scrollP
 // " %3C/script%3E");
 // doc.body.innerHTML = "<div class="avance-reculer"></div>" + doc.body.innerHTML;
}
```

ENDPOINT: /queue/

- Description: Get job data from Redis.
- Parameters:
  - `start` : Job Index (int) in time to start from for the data list. An example would be : `0`
- Responses:
  - A 200 response will : Return all queued job data as JSON
- Example: `curl -X GET http://localhost:5026/queue/ -H "accept: application/json"` yields:


```
[
  {
    "args": "[]",
    "class": "initialize",
    "end": "-",
    "function": "init",
    "id": "9:0",
    "output": "-",
    "start": "-",
    "status": "Queued",
    "uid": "9cd3e1f0-dea0-458f-a159-33083151c44f"
  },
  {
    "args": "[0]",
    "class": "appfields",
....
    "status": "Completed",
    "uid": "8c6949c8-9774-48f9-82da-5f30aed082b0"
  },
  {
    "args": "[]",
    "class": "initialize",
    "end": "2022-05-12 07:16:40.562422",
    "function": "init",
    "id": "0:0",
    "output": "None",
    "start": "2022-05-12 07:16:40.548847",
    "status": "Completed",
```

```
"uid": "84442c17-b48b-41b7-ad05-4f83b5e16505"
  }
]
```

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi: 10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd