Data Visualization Final Paper

July 2020

Abstract

The study of automatic music composition has a rich history of adapting algorithms developed in originally varied contexts. Among the most promising of these methods is the so-called "Markov Chain Composition Method", which allows powerful and robust automatic composition by recording the path of random walkers on networks encoding note and transition information. Here we present a proposal for developing a unique tool to visualize a variant of this method, where the network is generated from an existing piece of music and can be modified to a user's specifications. Our visualization will give the ability see how a base musical piece is transformed into different network structures depending on the encodings scheme, as well as how a random walker traverses the network to produce a new music piece. It will also give users the option to alter the network's edge weights and community partition method, thus creating more customized and better sounding compositions.

Contents

1 Authors

Daniel Kaiser and Kaitlin Pet

2 Introduction

2.1 Background

Ever since the 1950s, researchers and composers have created automatic music composition systems using a variety of methods. These methods involve defining or training a set of "composition rules", then using those rules to generate music(Fernandez and Vico 2013). A composer can then publish the algorithmically composed piece as a complete work, or more commonly, use it as 'raw material' for further processing(Fernandez and Vico 2013). The earliest examples of this practice stem from the Serialist movement where composers wrote pieces based on a strict set of note-generation rules (Fernandez and Vico 2013). More modern examples involve generating music using constructs from computer science and information theory, a notable example being Issac Schankler's works composing with cellular automaton (Schankler 2019).

One issue preventing more widespread adoption of algorithmic composition methods by composers is the technical barrier needed to 1) understand more complex algorithms and 2) program synthesizers that can create music from those algorithms. Many state-of-the-art artificial composition systems, such as Open AI's Deep Learning Jukebox (Dhariwal et al. 2020) and Donna Quick's Haskell-based composition systems (Quick 2010), require a strong computer science and math basis to use and can be extremely daunting for the average composer.

In response to these potential challenges, one of the authors of this proposal (Kaitlin Pet) developed an intuitive network-based music composition system for the class I606: Network Science in Spring 2020. The composition system developed by Pet is based on the Markov Chain Composition Method, an algorithmic composition method where the path of a random walker traversing a network is converted to a sequence of notes. A similar composition method is described in a 2009 paper entitled "Complex network structure of musical compositions: Algorithmic generation of appealing music" (Liu, Tse, and Small 2010). Therein Liu et al. describe a simple process of encoding a musical piece into a network. First, a piece of written music is converted to a weighted directed graph by a user-determined encoding method. This encoding determines which features of music become notes and which becomes links. For example, each distinct pitch could be classified as a different node (Liu, Tse, and Small 2010) or each harmony can be classified as a different node (Chen, Lin, and Jeng 2010). In these two cases, a 'link' would be defined as the transition from pitch to pitch or the transition from harmony to harmony. The weight of each link is equal to the number of times a node transition occurs within the piece (e.g. given a pitch-based encoding, if "C4" is followed by "G4" five times, the weight of the "C4, G4" link would be five).

The graph can then be further processed by taking communities into account. Communities are first determined by a partitioning algorithm of the user's choice, e.g. InfoMap (Rosvall and Bergstrom 2008). The links within and between communities can then be strengthened or weakened to affect the random walker's traversal through the network. Generally, increasing the weight of links joining different communities will increase the probability of a random walker leaving its current community, thus creating a composition with more shifts between communities. However, the extent of this phenomenon and the effect of increasing within-community edge weights depends on the structure of the community subgraph and how it is situated in the overall graph. We especially care about community structure because it can be "sonically represented" in the generated composition by varying note length based on community membership or the type of community transition. For example, if the graph is clustered using a hierarchical partitioning algorithm, the duration of a note about to leave a community could be lengthened to create a natural delimination in the music. Furthermore, transitioning to a more 'similar' community could result in a different duration change than transitioning to a more 'different' community, thus creating a sense of hierarchical structure in the resulting piece.

Written on paper or viewed in static form, this naturally intuitive composition system can look daunting or confusing. In order to make this system more accessible, an interactive visualization is necessary so composers without a math or computer science background can easily see both how and why a composition is generated. Visualization allows a composer to both understand the 'original' data network and decide the best ways to change it. Composers would thus have the power to create algorithmic music with a high level of understanding and control of the composition process, without having to possess a high amount of technical know-how.

2.2 Existing Visualizations

There is ample precedence in both scientific literature and elsewhere for the use of networks as a visualization tool for music composition. Here we shall discuss both these instances as well as more generalized visualization tools we have drawn inspiration from. Let us begin with the network visualizations of music composition.

Liu's Markov chain-based network figure is reproduced below. It clearly shows nodes and links but unlike what we are looking for, is a static image.

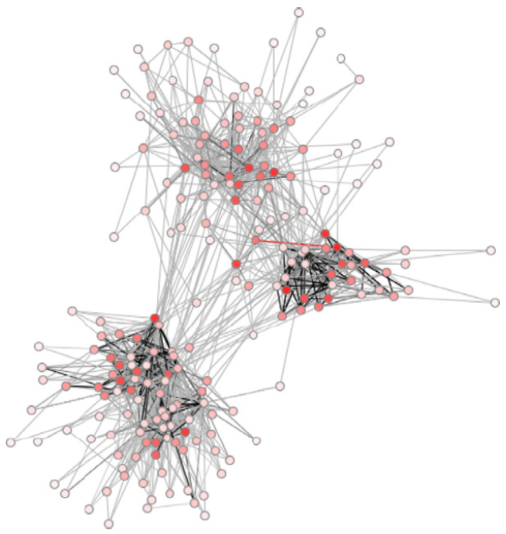
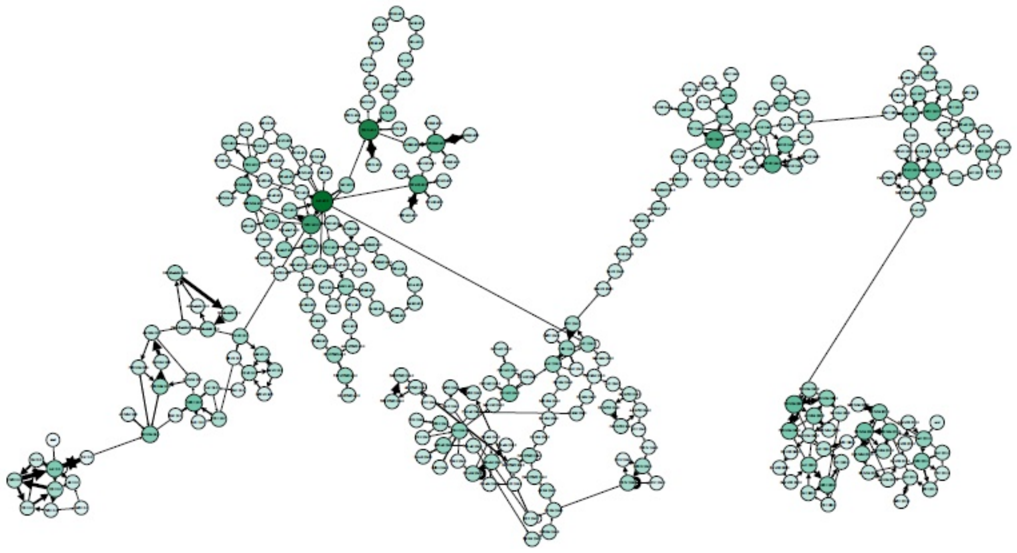
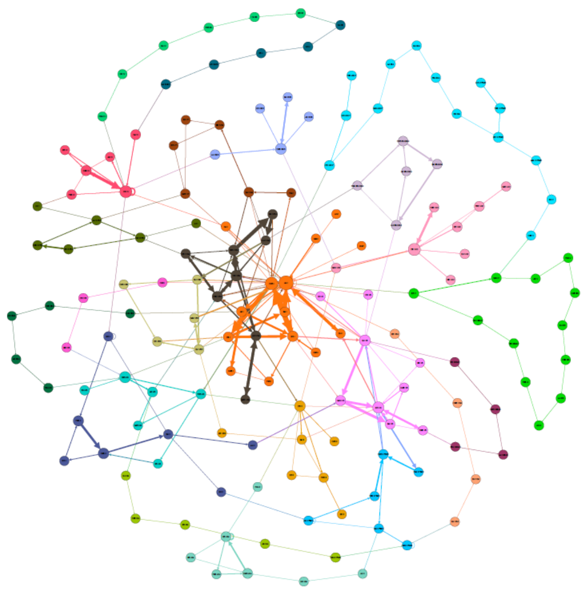
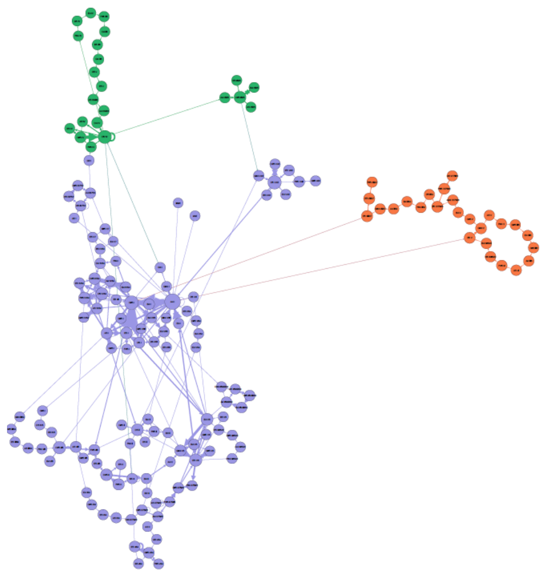


Fig. 1. A sample network from Bach's violin solos. Darkness of coloring of nodes indicates relative degrees, and darkness of coloring of edges indicates relative weights.

Here, the color scheme makes it easy to see which notes show up often in the piece, as well as which pairs of notes are often more commonplace than others. In this vein, presented below are similar images from prior work by author Kaitlin Pet (unpublished). These figures prove to show the extent of data we can convey within a network framework of music composition, namely, note encodings, edge weights, and community structure.







These networks form the core of our proposed project. They convey the information necessary to accomplish the project goals in a visually pleasing, clear, and concise format, as well as providing an interface for easy manipulation. Given the desire to alter a music network to produce an original piece, manipulation of network properties (such as edge weight between communities) directly affect the music composed by a random walker on the network. Notably, a graphic of a network allows for an easy check on the effect of manipulations being implemented.

Moving past particular example visualizations to more broad examples similar to the data we are handling, we now turn our attention inspirations in the general practice of visualizing network data in a beautiful and effective way. Of particular note, given our intention to utilize random walkers, is the mapequation demo (Rosvall and Bergstrom 2008), available at:
https://www.mapequation.org/apps/MapDemo.html

Included below is a screenshot, but truly a static image does not do it justice. Nevertheless, some valuable portions are still visible. Most importantly, this visualization is able to animate a random walker on a network with color coded communities embedded in the graph. Retrieving and displaying information gathered from a random walk is essential to our project, and the mapequation demo forms a beautiful, simple interface with which to accomplish these goals. It gives an example layout for where to place buttons, what order to embed animations, how to use color scheme effectively, and other variable parameters to consider (e.g. the speed of the random walker). Given we intend to develop a web application for this project, this successful and elegant web application for visualizing certain processes on networks contains exactly the visual elements necessary for a valuable contribution.

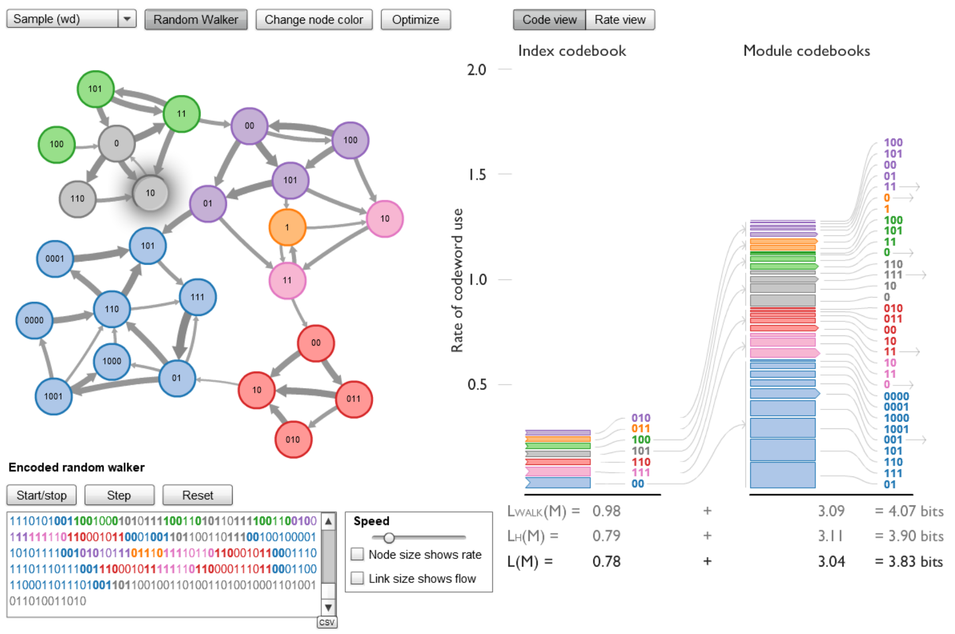
These form the core of the visualizations that inspired this project. While we shan't discuss them here, it may be noted that relevant visualizations can also be found at (Ognyanova 2019, Nodus 2020, Morris 2010).

3 Data and Methods

3.1 Data

Since we are mainly concerned with creating a data visualization framework, our aim is for the framework to be able to visualize any music 'data'. For demonstration purposes, the authors have prepared two scores in MusicXML, a specification language which used to store music score data electronically. These scores are created from two works , *Fantasie For Flute Without Bass* by G. P. Telemann and *Fur Elise* by Ludwig van Beethoven. The scores were prepared thus:

1. Midi versions of the scores were downloaded from IMSLP (https://imslp.org



```
1110101001100100010101111001101011011110011001001
111111101100010110001001101100110111001001000001
10101111001010101110111011110110110001011000111 0
1110111011100111000101111111011000011101100011 00
110001101110100110110010011010011010010001101001
011010011010
```

```
LWALK(M) =  0.98    +    3.09   = 4.07 bits
LH(M) =     0.79    +    3.11   = 3.90 bits
L(M) =      0.78    +    3.04   = 3.83 bits
```

) , a free online sheet music database for music no longer under copyright.

2. Scores were viewed in a commercial music editing software and inspected for errors by comparing them with IMSLP's published score PDFs.

3. *Fantasie for Solo Flute No. 1* was reharmonized in order to demonstrate the Roman Numeral encoding and the Grouped Roman numeral encoding, which use harmony to determine what constitutes nodes and links. This altered score is located in Appendix I.

4. Since *Fur Elise* already possessed harmony, less processing was needed. However, there were still many sections where only one or two notes sounded at a time and harmony was implied from context. In order to optimize the performance of the visualization, the authors consolidated broken chords into one and two-beat units so Roman Numeral Analysis using the music21 python package (Cuthbert and Ariza 2010) would recognize implied chords as the same unit. This altered score is located in Appendix II.

5. Both pieces were exported as .xml files. In this format, the data can now be uploaded to the visualization application. By default, the visualization shows *Fantasie for Solo Flute No. 1*, but we have included the *Fur Elise* .xml file in our Github repository (linked at end of paper).

6. Further processing is done within the python side of the application to change the .xml specified score information to different types of networks.

This process is described in detail in the Methods section.

3.2 Methods Overview

3.2.1 Visualization Requirements

There are several requirements for an effective visualization tool in this area. First and foremost is the need to effectively visualize a network. While there are extensive resources suggesting certain techniques (Wills 2009), we elected for the common 'force-directed' approach which shall be expanded upon later in the exposition.

Second, the metadata on our network also has to be considered. There are three items of importance to examine here: community assignment for nodes/edges, the note associated to each node, and edge weights. All three of these items have standard visualizations to be considered: coloring, labels, and thickness, respectively. Nodes and links can be colored to represent community assignment, the names of notes can be processed as labels on their associated nodes, and the larger an edge weight between two nodes, the thicker the corresponding link. These standard techniques will prove to be adequate for our purposes, primarily due to the generally small network sizes in this area. A midling to small number of nodes often yield an also middling to small number of communities, of which a dozen or so colors can adequately cover. There is often enough "room" to fit all of the nodes inside of a viewing frame without overlap, and hence note labels are clear as to where they are assigned. Lastly, differing opacities allow for a host of edges of differing thickness to be easily decipherable.

A priori, these are usual and theoretically-grounded methods of visualizing network data. Our tool, however, is designed not only as a visualization tool but as a general aid for music composition, and hence the network data must be fundamentally mutable. As a result, our visualization techniques must be inherently dynamic. For this reason, we have elected to use a hybrid Python/D3 approach to the creation of our tool. The Python (*Python* 2020) backend allows for robust calculations on our data, most notably a host of community detection algorithms - additionally, Flask (*Flask* 2020), a package for Python, allows a simple Python web server with very little 'setup.' The D3 (*D3* 2020) (via Observable notebooks) front end allows for interactive visualizations in a well-known and aesthetically pleasing framework. Furthermore, processes on networks are a well-explored and generally aesthetic topic in D3. This allows us to do very natural looking force-directed layouts, and vital to music networks, random walks. Namely, as part of the composition portion of the tool, we wish to generate a random walk on our directed weighted networks and show that walk in real time while simultaneously playing the appropriate note. This way, the user can hear the piece being composed and understand via their animated visualization what it is in the network data that is producing those patterns. The framework will be expanded upon in a latter section.

3.2.2 Candidate Visualization Frameworks

It is worthwhile to note the frameworks we neglected to use after brief experimentation. Notably there is Dash, a powerful Python framework for creating interactive web applications, built on top of Plotly (*Dash* 2020), the well-known Python package for creating informative and interactive data visualizations. This was a pure Python approach (as opposed to D3/Observable's unique flavor or JavaScript) and hence easier to implement given the authors' backgrounds and the general ease with which one can learn Python. However we ultimately deemed it less effectual than D3's tools in the areas described above. There was also work done to integrate Cytoscape.JS (*Cytoscape.JS* 2020), a JavaScript framework for dealing specifically with network data, into Dash because of its ease of manipulation and attractive visuals. It did indeed provide a quick, intuitive approach to display our network data, but unfortunately, we had troubles random walks that could render sonically as well as visually.

3.3 Data Analysis

3.3.1 Graph Creation

This visualization system has four possible methods of encoding score data into network form. These different methods offer a range of different views of the data, from an extremely global view where each distinct pitch is a separate node, to more local views where the node designation of a pitch depends on its position in the score. Some methods also offer the opportunity to parse nodes based on harmonic as well as melodic information.

1. Basic Encoded Graph: This graph encoding maps each distinct pitch as a separate node (i.e. the total number of nodes is the set of distinct pitches in the original dataset). To create this encoding, the XML data was parsed using music21 (a Python music analysis package) and converted into a sequence of notes ordered as they appeared in the piece (Cuthbert and Ariza 2010). A multi-edge directed network was created using networkx (Hagberg, Schult, and Swart 2008), where each note became a node in the graph and each note transition (sequence of two adjacent notes) was converted to a directed edge. If a note transition was repeated, another edge between the original two nodes was added to create a multi-edge directed graph. A "start" and "end" node were added to facilitate beginning and terminating the random walk.

2. Forced Grouping Graph: To build this graph, the piece is divided into discreet sections, where each section is encoded as a subgraph with only one link exiting and entering the subgraph. This method allows differences between note-to-note transition probabilities in different sections of the piece to be reflected.

3. Roman Numeral Graph: Roman numeral analysis is a chord classification scheme based on how different harmonies function within a piece (Staneković 2020) By using Roman numeral analysis to classify nodes, node transitions will now reflect both a piece's harmonic and melodic transition patterns. To encode the harmonized version into nodes, the music21 chordify method was first used to convert each simultaneously occurring instance of melody and accompaniment into a separate chord. Then, each chord was classified using Roman Numeral Analysis and converted to a node. Each chord transition became a directed edge.

4. Roman Numeral Graph with Forced Grouping: This is a combination of the Roman Numeral Graph and Forced Grouping Graph described above. The advantage of this method is a rich node encoding scheme that takes into account local transition probability differences and harmonic structure.

3.3.2 Rendering Random Melody

To create an algorithmic composition from these graphs, a random walk is run from the 'start' node to the 'end' node. The probability node A will step to node B depends on the percent of node A's outgoing edges that lead to node B. This list of nodes can then be translated back into a melody in two main ways.

The first method is the most obvious route: for every node in the walk, play each pitch sequentially for the same duration (Current algorithm defaults to a sixteenth note). Because every note is the same length, music generated with this method shows very little rhythmic structure.

One can also consider the overall graph structure when making note length assignments. The state of a node relative to overall graph structure in the random walk can be taken into account by increasing the duration of notes about to transition into different sections. In the Forced Grouping and Roman Numeral Forced Grouping graph encodings, this can translate to the source node of the section outgoing link in a random walk receiving extra length. This method is computationally trivial but successfully delimits different sections of the random walk that draw from different sets of note transition probabilities.

A less trivial use of this method involves using communities naturally occurring in the graph instead of subsections pre-programmed by the user. In this case, the note length corresponding to a given node A depends on if the next node B stays within the same community or transitions to a new one. Further delineation through note duration can be drawn in hierarchical communities, where travel between high-level communities can result in a different length assignments than travel between sub-communities. This way, a complex rhythmic fabric can be created that, by definition, has hierarchical rhythmic structure.

3.3.3 Community Detection

Community structure is valuable for both analytic and generative purposes. In the *analytic* sphere, visualizing how different nodes and links are associated with one another gives a proxy for random walk behavior. The intuitive notion of a strong community usually includes more connections amongst members of the community than entering/exiting the community (Fortunato and Hric 2016). Hence, random walks entering a strong community are likely to spend a considerable amount of time within a community - this process is used to discover communities within Infomap (Rosvall, Axelsson, and Bergstrom 2009). In the *generative* sphere, different community mappings provide different opportunities for mapping the parameters of generated compositions. As stated above, certain random walk generation methods are based on community *hierarchy* data; namely, note duration in the random walk is classified on the basis of hierarchy assignment. For this reason, a portion of the community detection algorithm suite decided upon are hierarchical methods, specifically, Infomap, Louvain, and Hierarchical Link Communities.

The "natural" choice for community detection in music networks appears to be Infomap. Infomap is a community detection method with deep ties to random walks on networks, and given the random walk process we wish to investigate, yield communities that give direct, immediate, at-a-glance views into the expected behavior of a random walker. The full details of the method are explained here (Rosvall and Bergstrom 2008; Rosvall, Axelsson, and Bergstrom 2009).

The Louvain method refers to a popular, fast, greedy modularity optimization method of community detection developed in France (Blondel et al. 2008). The method has become incredibly popular and in many ways a "go-to" for node-based community detection given its incredible speed and reasonable accuracy on many classic benchmarks. It is also hierarchical. Given these strengths and the ease of implementation, it seems incorrect to not include the method as one of our partition options.

The label propagation method is an intriguing method wherein every node simply adopts the most popular community label of its neighbors, repeating this until convergence. It is notably very quick and also entirely topologically driven, requiring no parameters nor inputs aside from the network and a choice to update community labels synchronously or asynchronously (Raghavan, Albert, and Kumara 2007). While not hierarchical, it's implementation is trivial in igraph and still produces meaningful results in our graphs, and hence we have elected to include it as well.

The last of our selected methods[1] is a link-based method created by Ahn et

[1] Note this is the list of possible methods at the time of writing, there are plans to continue the project after the course completes.

al. rred to as "Hierarchical Link Communities." The method detects hierarchical communities *of links*. Consequently, it is possible for nodes to be placed in several communities if node community assignments are based off of their edge community assignments. The ability to detect nodes rightfully belonging to several groups makes this a powerful method within the scope of our networks, and warranted its inclusion. Note that, at the time of writing, we have not included any hierarchical returned data from the method, i.e., the tool currently only shows the partition of links the method deems is "optimal" via the measure *partition density*. See (Ahn, Bagrow, and Lehmann 2010) for details.

4 Results and insights

4.1 Visualization Framework

As stated earlier, we decided to use the D3 Observable framework for front-end visualization and a python-based server for back-end calculations and real time data analysis. D3 Observable was initially attractive because of its highly interactive and customize graphics, fast processing speed, and ability to integrate with sound production. It successfully delivered on those initial promises. We found that an addition advantage of D3 Observable was its highly active visualization community - since users are encouraged to share the source code of their notebooks, it was easy to quickly adapt desired features in from other visualizations into our own. Consulted visualizations are cited in our Observable Notebook:
https://observablehq.com/@kpet123/music-composition-graph-link-view

The python backend framework was also fairly successful. We were able to work within the Flask app framework to do necessary graph generation and community detection operations. While there exists seemingly an infinitude of algorithms for community detection in Python, two collections stand out as especially common, namely, the built-in methods to the networkx and igraph network packages. It is the opinion of the authors that igraph forms a more appropriate choice for many network data given its C core and impressive suite of included methods, yielding fast performance across many algorithms in a handful of lines of code. Namely, there exists built-in methods for Infomap, Louvain, and Label Propagation. Consequently, the generation of these communities is nearing trivial with the igraph framework, and hence we have elected for this reason, in addition to potential speed improvements[2], to convert network calculations to igraph for the community detection calculations.

[2]At the size of network we've initially analyzed, speed differences between pure Pythonic networkx and C-based igraph are negligible to all but the most eerily impatient of users.

4.2 Visualization as an Analytic Tool

The uploaded data can be viewed in four different conditions: Basic Encoded Graph, Forced Grouping Graph, Roman Numeral Graph, and Roman Numeral Forced Grouping Graph. Each of these views give a different window into the note's transition properties in the original piece. For example, the Basic Encoded graph provides the most straightforward view by mapping of nodes to pitches in a one-to-one relationship. As a result, node properties in the graph correspond directly to note properties in the piece. For example, in the Telemann graph one can see that the most connected note (highest degree) is A4.

Other graph encodings, like the Roman Numeral Encoding Graph allows one to get a more holistic view of the piece as a combination of melody and harmony. Each node is labeled as its pitch, but mousing over will show its Roman Numeral harmonic designation. An interesting discovery from the Telemann dataset was that the graph contained long 'strings' of nodes that looped back and forth the main graph body, indicating distinct harmony sequences. This type of structure is uniquely suited to being viewed in an animated, interactable format. At a 'normal' node size where labels are clearly visible, these string-like section tend to 'tangle' with the rest of the graph. However, when node size is shifted to be very small, these structures can be easily seen. The Grouped Roman Numeral Telemann graph was even more sequential, with a majority of the nodes with only one incoming an outgoing link. Our dynamic view allows one to pull apart the graph and change its scale, revealing the underlying structure.

Community Visualization

Each of the four graph encodings in term allow for the four types of community partitions described above. The basic graph encoding returns trivial structure for the Telemann data because global structure was nearing fully connected. Unsurprisingly, the Forced Grouping and Roman Numeral Grouping returned communities that conformed to the initial subgraph assignment. The Roman Numeral graph yielded interesting and nontrival community assignments, loosely dividing high connectivity nodes from more string-like sections.

This visualization also allows for a temporal view of graph community and other graph structures by allowing the user to run the original melody as a walk on the network. This way, users can see in real time how a walker stays within communities or exits from them. It also gives a sense of how the the original data 'fills out' the network, as well as a temporal sense of when certain groups of nodes are played in the piece. This process can be repeated with random walks to see what extent the 'true' walk aligns with stochastic scrambling. Unsurprisingly, walks created from the basic graph preserves very little of the melody's original features, but walks done on the Roman Numeral and Grouped Roman numeral graphs have sections exactly matching the original melody because of the graph's 'stringy' characteristics.

4.3 Visualization as a Generative Tool

Each graph encoding can generate two kinds of random walks - those that ignore community structure and those that consider community structure. Generating a random walk using the 'Ignore community structure' setting returns a melody consisting of equal-length notes that can then be "played" on the graph (In the Forced Grouping and Random Walk Forced Grouping graphs, "trivial" transition notes denoting changes between subgraphs are also lengthened for listening clarity). Generating a random walk using the "Consider community structure" setting returns a melody that consists of different note lengths depending on whether a note is the last one in a certain community. Currently, notes that are not on 'community edges' are assigned as sixteenth notes (1/4 of a beat), notes changing between top-level communities are assigned to a quarter note (one beat), notes changing between second-level communities are assigned to an eight note (half beat), notes changing between third level communities are assigned to a triplet (1/3 of a beat) and notes changing between fourth level communities are assigned to a thirty-second note (1/8 of a beat). This mapping parameter would ideally be a composer's choice and way for them to express their unique voice in creating the composition, something they can tinker with to produce their ideal sound. Even with this static mapping, the authors have observed in graphs created from both the Telemann and Fur Elise data that generating walks with different community partitions produce very different sounding melodies. This is especially apparent when comparing non-hierarchical partitions like Label Propagation with the hierarchical ones. Non-hierarchical communities can only produce pieces with two note lengths (sixteenth note if no change, quarter note if community changes), resulting in a much simpler rhythmic fabric.

Another method for users to further process the graph is through changing edge weights. For example, in the Grouped Roman Numeral graph for the Telemann data, random walks tend to stay in the first subgraph for a very long time because nodes within the subgraph are highly connected but the transition out of the subgraph only has a weight of one. This could be considered undesirable by a composer. To combat this potential problem, the weight of the outgoing edge can be increased to shorten the time spent in that community. Currently there is no part of the application quantifying this effect, and desired weights need to be tweaked manually by the composer. (See Future Directions for better potential ways to integrating edge weight changes in the visualization and quantitatively show their effect).

4.4 Design Considerations

4.4.1 Application Body

The layout and functionality of the application body are not negligible and shall be discussed here. The main body of the application is a rendering of the Observable notebook embedded within the application displaying the network data. One can drag both the view and position of individual nodes easily, as well as select certain edges for weight manipulation (a text entry box just below the chart). Items such as node radius and force-directed attraction strength are sliders below the chart so as not to clog up the display itself. In addition to the Observable-based manipulation methods, additional python-based functionalities such as uploading a musical score, switching between encoding methods, switching between community visualization methods, generating different types of random walks, changing edge weight and starting the playback of the random walk or original melody are included under the chart. This display is still fairly rudimentary and we plan on making improvements (see Future Directions).

Color

Since color serves several functions in this visualization, choosing colors was vitally important. There were three categories of objects that needed to be distinguished: nodes, links, 'change indicators'. Nodes and links are colored to reflect community structure, while 'change indicators' are used to indicate that an edge has been selected with the intention of alteration or has been altered.

- Nodes

  To color nodes we initially tried D3's default color palette - d3.scale10 - but found that though color groupings were easily distinguishable, red and green were both categories, thus making the visualization inaccessible to color-blind individuals. We then tried sampling from the Viridis color space, but more than four categories were hard to distinguish. In addition, the bright yellow at the end of the Viridis spectrum made node labels (white) hard to discern. This phenomena indicates that the best practice of varying colormaps by lightness and saturation would need to be altered in our visualization because all colors needed to be dark enough to contrast well with white labels. A sensible discretization of Viridis or other lightness-spanning colormaps would seemingly require a nonlinear interpolation Instead, we've opted to select from D3's built-in categorical color schemes, namely, Dark2 and Pastel1. Note the selection of two default color schemes - for the hierarchical methods, two levels of hierarchy were returned for the data observed. Hence, we need to display two node clusterings simultaneously. While there are a host of methods to do this, we assigned the node center color as the highest hierarchical community assignment and the node outline color as the second highest hierarchical community assignment. We chose two separate color schemes with different levels of saturation to allow for easy differentiation, even in a black-and-white print. It should be noted that though distinguishability is very high between inner and outer community color, colors within both the Dark2 and Pastel1 palettes are very similar. See Future directions for potential solutions to this problem.

- Links

Coloring links was a bit more tricky because links have two simultaneous color considerations : community (link communities) and 'change indicators' to facilitate graph interaction. One method we used to distinguish between these two classes was opacity, or lightness. Normal links were shown in low opacity, while selected links had much higher opacity values. This scheme two additional benefits: 1) the darker and more vibrant selected links 'pop' to the viewer (a result of the preattentive processing phenomena discussed earlier in the Data Visualization course) and 2) more transparent normal links that overlap with each other can still be distinguished. The second benefit is especially noticeable in the Basic Graph Encoding because there are relatively few nodes, each with a high number of transitions. At full saturation, the links looked like a impenetrable blob, but at low saturation individual links can be distinguished. Given the similar restrictions to coloring as alluded to above (and throughout the course), albeit of only one assignment per link, we have also chosen a categorical coloring scheme, instead of interpolating a spectrum.

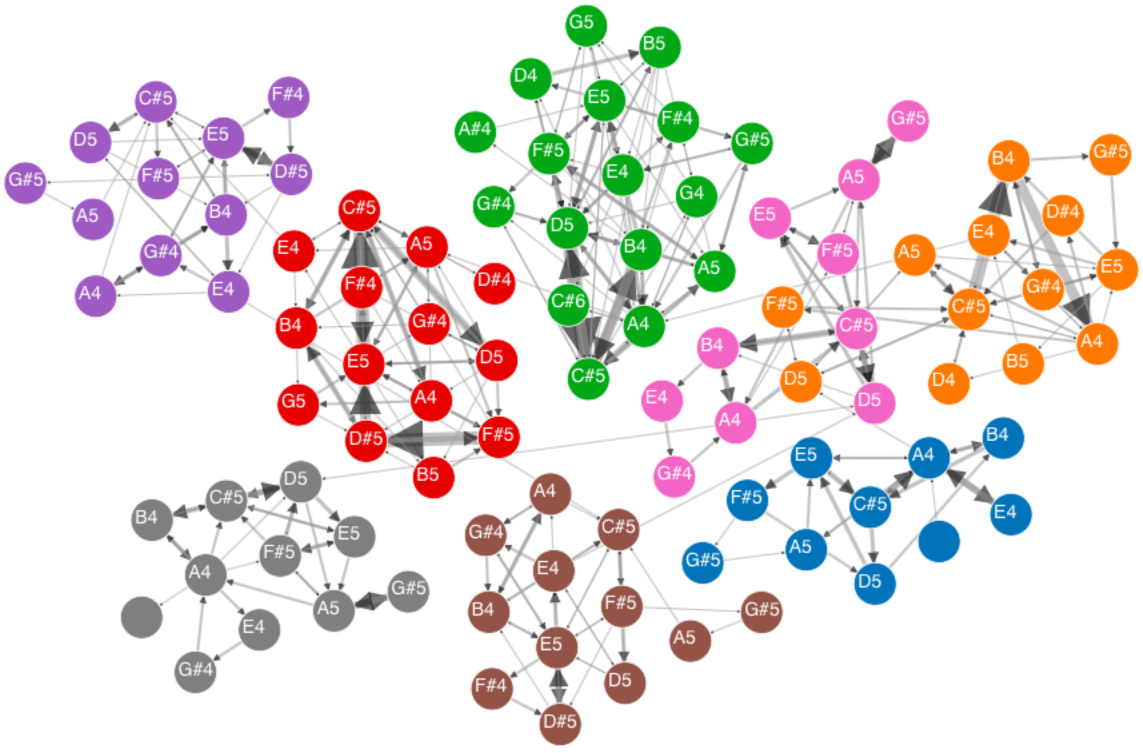
Shapes

- Nodes

  We decided to use the traditional circular node in our graph representation. All nodes are the same size for ease of implementation, but there are potentially other options to explore (see Future Directions).

- Links

  Link shape was a very important consideration in this visualization because links are a proxy for the probability transition table showing how likely a walker on one node will move to another. Thus, it is important for users to 1) distinguish between incoming and outgoing links to compare real neighbors (connected by outgoing link) to 'fake' ones (connected by incoming link) and 2) compare outgoing link thickness to get an intuitive sense of which true neighboring node is the most likely next step in the walk. Initially, we tried straight links between nodes, which resulted in a very clean visualization where adjacent nodes were distinguishable through arrowhead size (see figure below for straight links in the Forced Grouping encoding of Telemann). However, we ran into issues in how the user would select links for weight changes - in the case of a bidirectional arrow, only one arrow would be selectable. Therefore, we shifted to a framework were links were curved. To minimize clutter (the total number of links connected to many nodes were now doubled), we scaled the link thickness to its weight via a logarithmic relationship. Though this had the benefit of making links easier to distinguish and the overall graph cleaner, there was now no longer a direct proportional relationship between link thickness and transition probability, only the heuristic that slightly thicker lines are more likely to be traversed. Overall, we felt curved, log-scaled links were the best solution to balance these considerations.



Optimizing the Force-Directed Framework

Since this visualization aims to show a variety of different graphs with vastly different node counts, an important issue we faced was consistently generating a visualization that filled the given space and created sufficient space between nodes for them to coalesce into groups. Due to the nature of the D3 force-direct layout, an arrangement that allowed the smaller Basic encoding graph to render nicely caused the far denser Grouped graph to be squished on the side of the SVG frame the network was rendered in. Though we added sliders for users to change simulation attributes such as node size and node repulsive force, we wanted the visualization to automatically generate a decent-looking layout with minimal user fiddling. To achieve this goal, we used two main strategies:

- Automatically scaling node size. The default node radius is calculated for each graph so the node-to-whitespace ratio is 10:1. This way, smaller graphs are rendered with large nodes and larger graphs are rendered with small nodes.

- Automatically setting inter-node distance. Setting the 'force-collide parameter' to 1.5 * the node radius prevented nodes from squeezing themselves on the edge of the chart viewbox. This did cause a few nodes to migrate out of the viewbox, but this could be fixed by manually decreasing the node size by a little.

5 Discussion

5.1 Core Goals

We were successful in our initial goal of creating a music visualization and generation tool that is accessible to composers with limited technical know-how. Obtaining data to feed into this visualization is straightforward and possible for anyone with a basic composition software package (e.g. Musescore or Sibelius). Once a user acquires the data, they only need to upload it to the application to see a graph encoding.

As our main goal was creating a visualization tool and we are not the intended users of this tool, we do not have 'insights into the data' in a traditional sense. However, we succeeded in merging the two ways musicians experience music - pure audio and score-based 'instructions' - into a cohesive whole. Through this visualization, one can see and hear in real time how a score is progressing, as well as getting a feel of the piece's past and future through seeing connected nodes and link thicknesses. The entire presentation is also pleasant to look at and listen to.

Our visualization also provides insights as a transparent, easy-to-use generative tool. As stated in the introduction, the biggest advantage of a Markov-chain based algorithmic composition framework is its intuitive nature. Though the concept of visualizing a Markov-chain composition space has been around for a while, few have capitalized on its potential as an interactive, manipulable system. By labeling each node with its note name and playing the corresponding note on click, we have created a visualization where a composer understands the note-to-node connection on an intuitive level. The default playback mode - showing the original melody of the submitted piece as a walk on the graph - further cements the idea of a movement through the graph being constrained by its shape and link properties. With this understanding of how the visualization works, a composer can now manipulate the system (via changing edge weights, graph encoding, and community assignment) to customize the algorithmically generated random walk.

5.2 Future Directions

While much has been accomplished, the nature of this project as a tool necessitates continual improvement. As such, future directions have been considered throughout the semester. Broadly speaking, there are two categories for future change: calculations and visualizations.

5.2.1 Calculations

Two vital components to music composition via network science, as seen in the above exposition, are random walks and community detection. These aspects have significant room for expansion within our project. We shall iterate them here.

- Community Detection: Support for more community detection methods will be added as requested. One avenue the authors particularly want to explore is the inclusion of several new link-based methods. Secondly, related to general network calculations, functionality for exploring network centrality measures via node size will be added.
- Random Walk: The ability to write the random walk to a score file is vital and will be added shortly. Presently, the random walk is represented as a JSON dictionary, which can be read by the graph "instrument" but not by any other software or human. Writing a to a machine/human readable format such as MusicXML or MIDI is essential for the portability of this visualization.
- Computation Speed: Several computations, especially those involving music21's Roman Numeral Analysis function, are very slow, taking an excess of 20 seconds to parse a few pages of music. Ideally, we could find a way to optimize this process so users do not need to wait as long for graphs to generate.

5.2.2 Visualization

The second category of improvement is the visual aspects themselves. Most of the actual improvements can be boiled down to added customizability in visualization options, such as different methods of spatially embedding the data, different color scheme options, altering node and link shapes, etc. We may also work on designing a new D3 colormap that fits our needs of having high contrast to white while still varying enough in lightness/hue to be easily distinguishable. Additionally, we plan on designing a cleaner, more intuitive dashboard where encoding option buttons are grouped in columns as well as rows to create a cleaner display. Lastly, we plan on visualizing more network property distributions, such as degree and other centrality measures, as well as adding other methods of visualizing communities such as community network projections and adjacency block visualization.

We would especially like to note potential advances in how edge weights are chosen and manipulated. Right now only one edge weight can be manipulated at a time, which can be cumbersome if a composer want to make many changes on a dense graph. A solution would be introducing selectors that filter edges by certain criteria. For example, if a composer wants to increase flow between node-based communities, they could potentially select all such edges at once by applying a filter. This approach could create even more exciting possibilities for customizing our algorithmic composition tool.

The Github repository containing this project will be actively maintained and is available to the interested reader at: https://github.com/kpet123/Network-Composition.

References

[20a] *Cytoscape.JS*. 2020. URL: https://js.cytoscape.org/.

[20b] *D3*. 2020. URL: https://github.com/d3/d3/wiki.

[20c] *Dash*. 2020. URL: https://dash.plotly.com/.

[20d] *Flask*. 2020. URL: https://flask.palletsprojects.com/en/1.1.x/.

[20e] *Python*. 2020. URL: https://docs.python.org/3/reference/.

[ABL10] Yong-Yeol Ahn, James P. Bagrow, and Sune Lehmann. "Link communities reveal multiscale complexity in networks". In: *Nature* 466.7307 (Aug. 2010). Number: 7307 Publisher: Nature Publishing Group, pp. 761–764. ISSN: 1476-4687. DOI: 10.1038/nature09182. URL: https://www.nature.com/articles/nature09182 (visited on 07/30/2020).

[Blo+08] Vincent D. Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 9, 2008), P10008. ISSN: 1742-5468. DOI: 10.1088/1742-5468/2008/10/P10008. arXiv: 0803.0476. URL: http://arxiv.org/abs/0803.0476 (visited on 07/30/2020).

[CA10] Michael Scott Cuthbert and Christopher Ariza. "music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data". en. In: (2010), p. 6.

[CLJ10] Wei-An Chen, Jihg-Hong Lin, and Shyh-Kang Jeng. "Harmony Graph, a Social-Network-Like Structure, and Its Applications to Music Corpus Visualization, Distinguishing and Music Generation". en. In: (2010), p. 18.

[Dha+20] Prafulla Dhariwal et al. "Jukebox: A Generative Model for Music". In: *arXiv:2005.00341 [cs, eess, stat]* (Apr. 2020). arXiv: 2005.00341. URL: http://arxiv.org/abs/2005.00341 (visited on 07/30/2020).

[FH16] Santo Fortunato and Darko Hric. "Community detection in networks: A user guide". In: *Physics Reports* 659 (Nov. 2016), pp. 1–44. ISSN: 03701573. DOI: 10.1016/j.physrep.2016.09.002. arXiv: 1608.00163. URL: http://arxiv.org/abs/1608.00163 (visited on 07/30/2020).

[FV13] Jose David Fernandez and Francisco Vico. "AI Methods in Algorithmic Composition: A Comprehensive Survey". In: *Journal of Artificial Intelligence Research* 48 (Nov. 2013). arXiv: 1402.0585, pp. 513–582. ISSN: 1076-9757. DOI: 10.1613/jair.3908. URL: http://arxiv.org/abs/1402.0585 (visited on 03/14/2020).

[HSS08] Aric A Hagberg, Daniel A Schult, and Pieter J Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". en. In: (2008), p. 5.

[LTS10] Xiao Fan Liu, Chi K. Tse, and Michael Small. "Complex network structure of musical compositions: Algorithmic generation of appealing music". en. In: *Physica A: Statistical Mechanics and its Applications* 389.1 (Jan. 2010), pp. 126–132. ISSN: 03784371. DOI: `10.1016/j.physa.2009.08.035`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0378437109006827` (visited on 02/27/2020).

[Mor10] Robert Morris. "Some Musical Applications of Minimal Graph Cycles". In: *Theory and Practice* 35 (2010), pp. 95–117. ISSN: 07416156. URL: `http://www.jstor.org/stable/41784452`.

[Nod20] Nodus. In: (2020). URL: `https://noduslabs.com/cases/network-graph-musical-composition-instrument/`.

[Ogn19] Katya Ognyanova. In: (2019). URL: `www.kateto.net/network-visualization`.

[Qui10] Donya Quick. "Generating Music Using Concepts from Schenkerian Analysis and Chord Spaces". en. PhD thesis. Yale, May 2010.

[RAB09] M. Rosvall, D. Axelsson, and C. T. Bergstrom. "The map equation". In: *The European Physical Journal Special Topics* 178.1 (Nov. 2009), pp. 13–23. ISSN: 1951-6355, 1951-6401. DOI: `10.1140/epjst/e2010-01179-1`. URL: `http://link.springer.com/10.1140/epjst/e2010-01179-1` (visited on 07/30/2020).

[RAK07] Usha Nandini Raghavan, Reka Albert, and Soundar Kumara. "Near linear time algorithm to detect community structures in large-scale networks". In: *Physical Review E* 76.3 (Sept. 11, 2007), p. 036106. ISSN: 1539-3755, 1550-2376. DOI: `10.1103/PhysRevE.76.036106`. arXiv: `0709.2938`. URL: `http://arxiv.org/abs/0709.2938` (visited on 07/30/2020).

[RB08] M. Rosvall and C. T. Bergstrom. "Maps of random walks on complex networks reveal community structure". en. In: *Proceedings of the National Academy of Sciences* 105.4 (Jan. 2008), pp. 1118–1123. ISSN: 0027-8424, 1091-6490. DOI: `10.1073/pnas.0706851105`. URL: `http://www.pnas.org/cgi/doi/10.1073/pnas.0706851105` (visited on 03/31/2020).

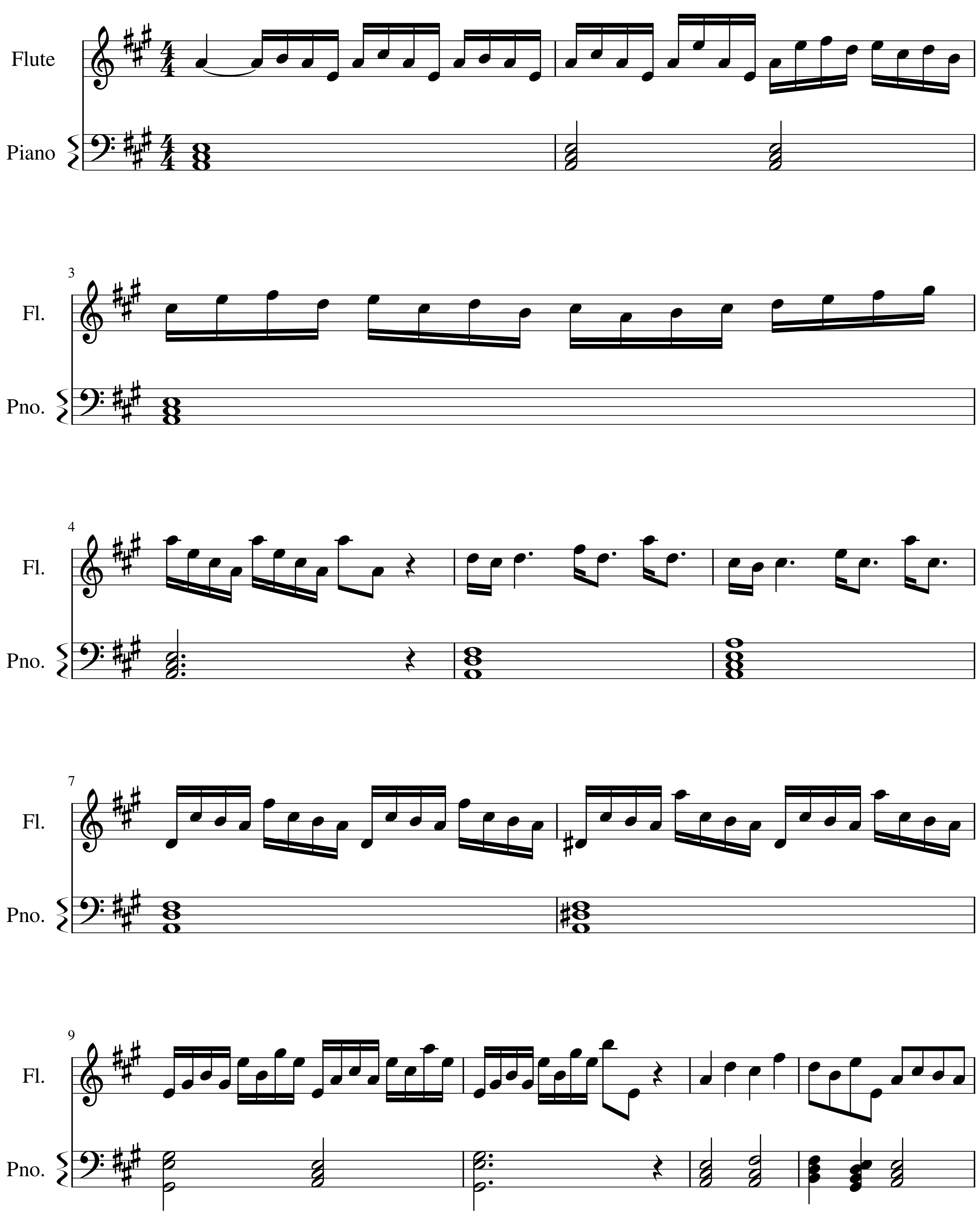
[Sch19] Issac Schankler. *Because Patterns by Isaac Schankler*. en-US. Library Catalog: aerocademusic.com. May 2019. URL: `https://aerocademusic.com/because-patterns` (visited on 07/30/2020).

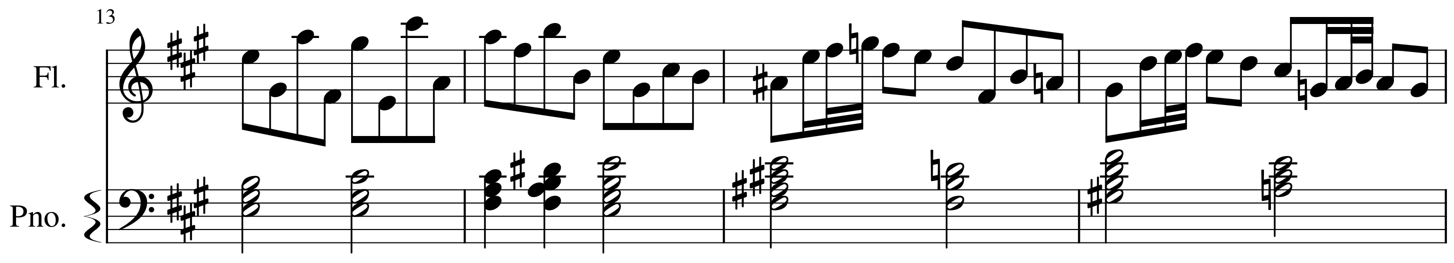
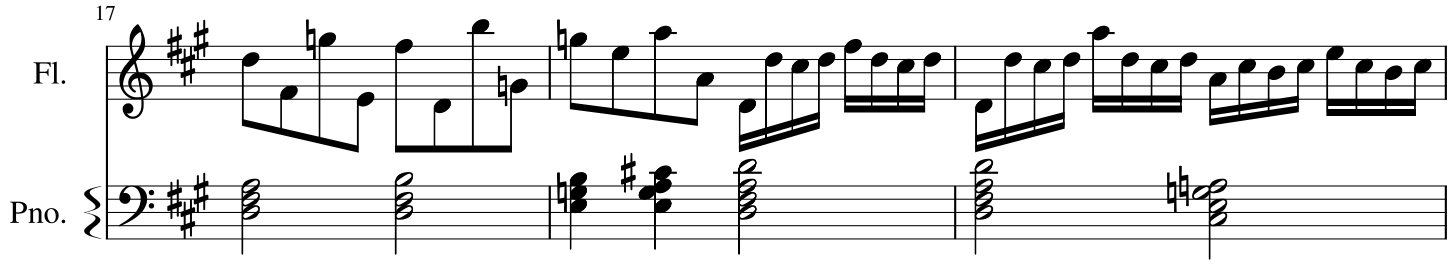
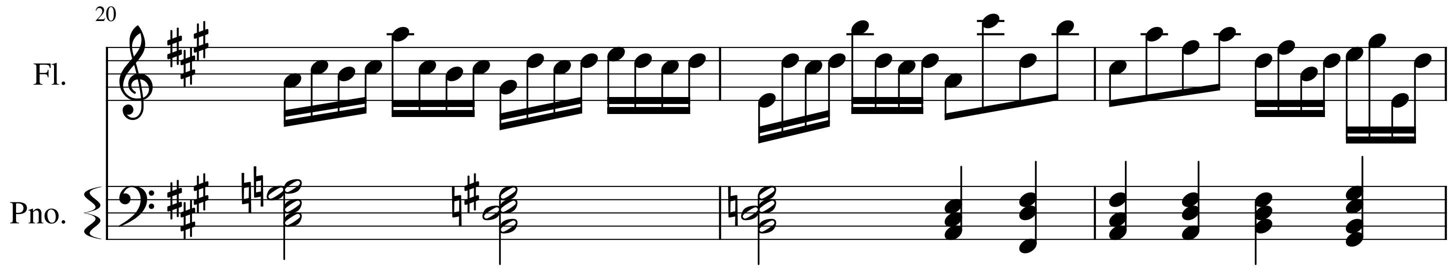
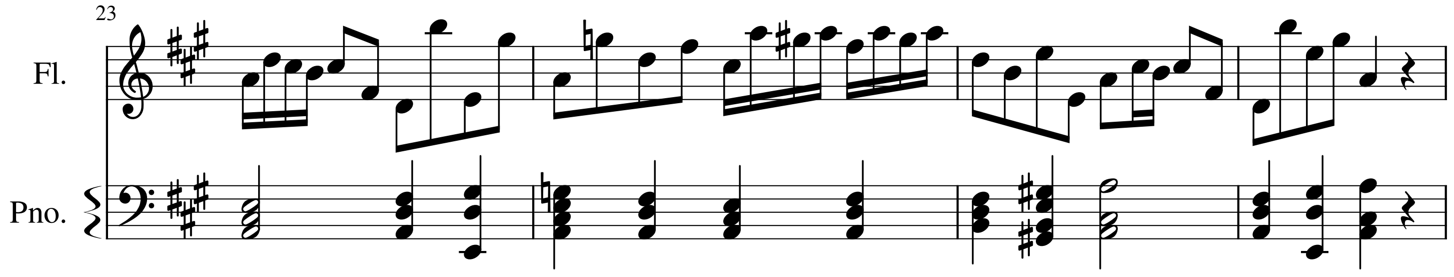
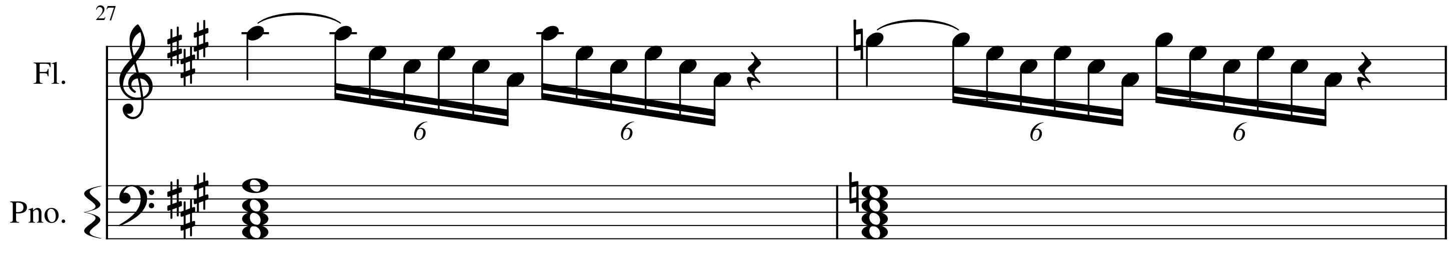
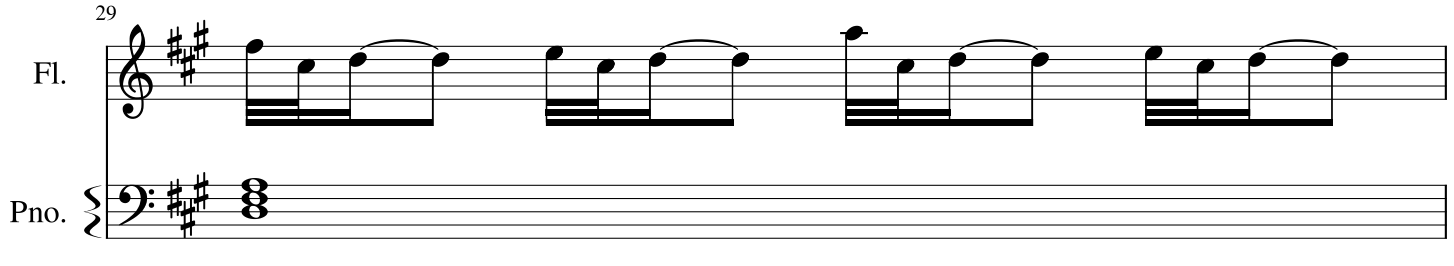
[Sta] Hrvoje Staneković. *Musical harmony, theory with examples from musical compositions by J.S.Bach and F.Chopin.* en. Self-Published Bookd. Library Catalog: sites.google.com. URL: `https://sites.google.com/view/musicalharmonysite/home-page` (visited on 03/13/2020).

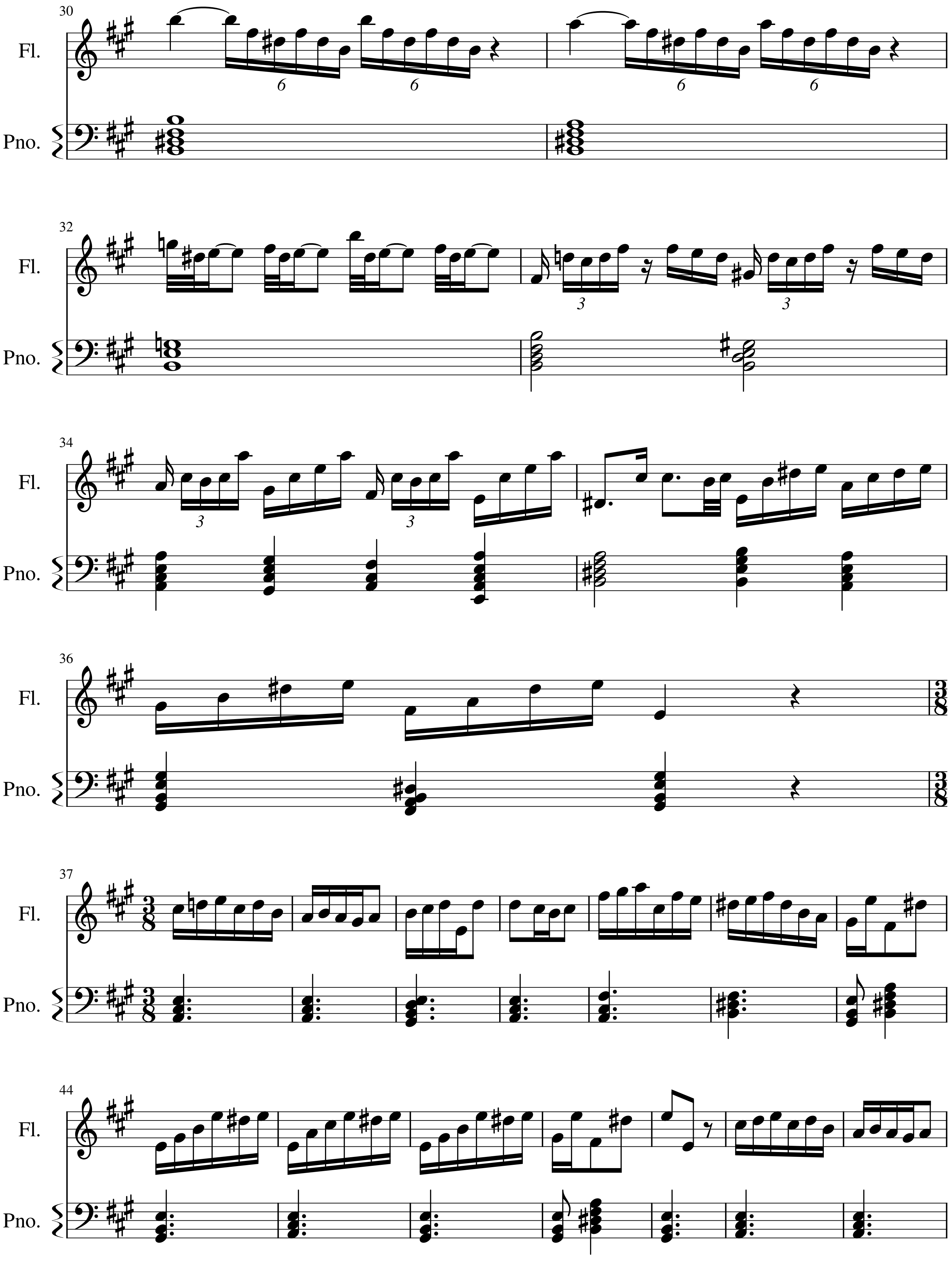
[Wil09] Graham Wills. “Visualizing Network Data”. In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 3432–3437. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_1381. URL: https://doi.org/10.1007/978-0-387-39940-9_1381 (visited on 07/30/2020).

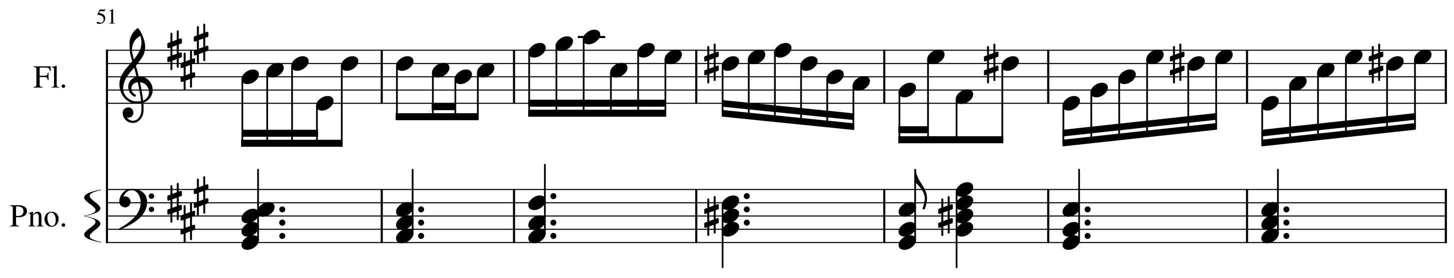
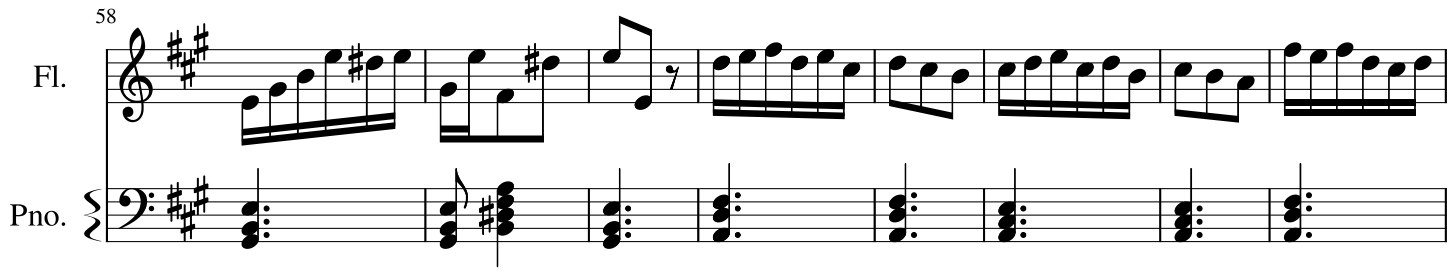
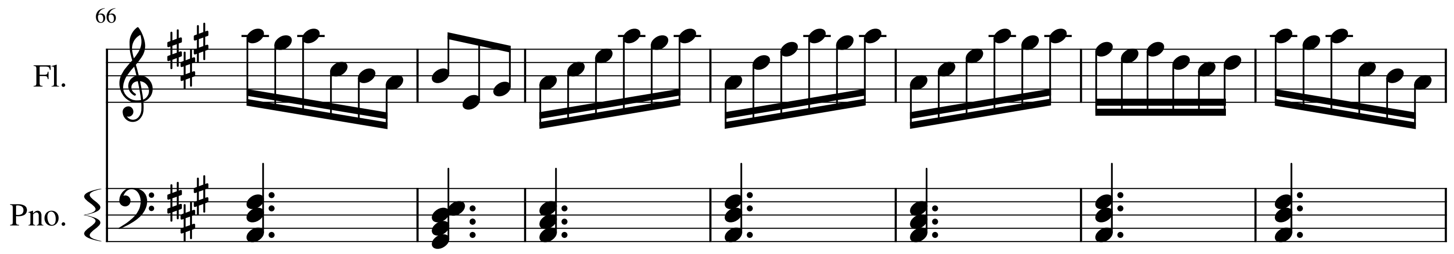
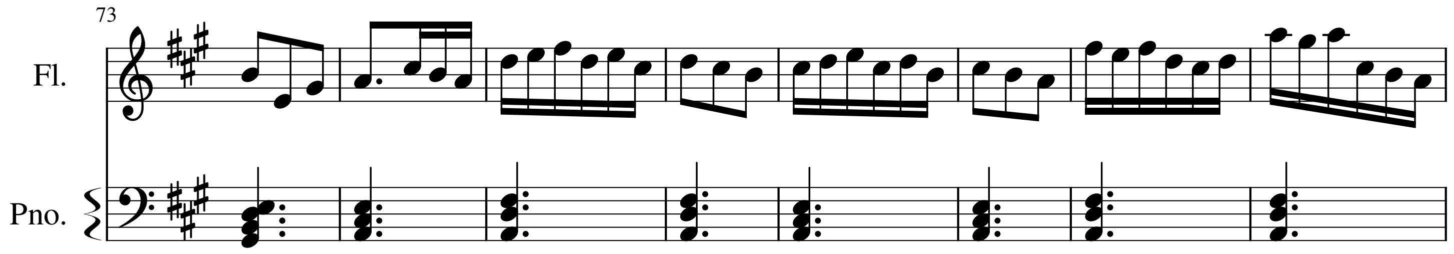
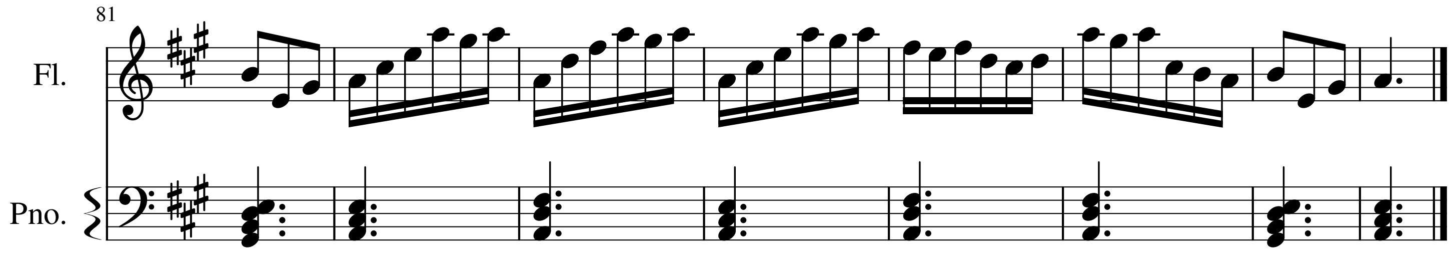
Appendix I: Harmonized Version of G. P. Telemann’s Fantasie for Solo Flute No. 1

IMSLP236801-WIMA.a3ab-TelemannWV40.02_Fantasia01-A




13
Fl.
Pno.

17
Fl.
Pno.

20
Fl.
Pno.

23
Fl.
Pno.

27
Fl.
Pno.
6
6
6
6

29
Fl.
Pno.

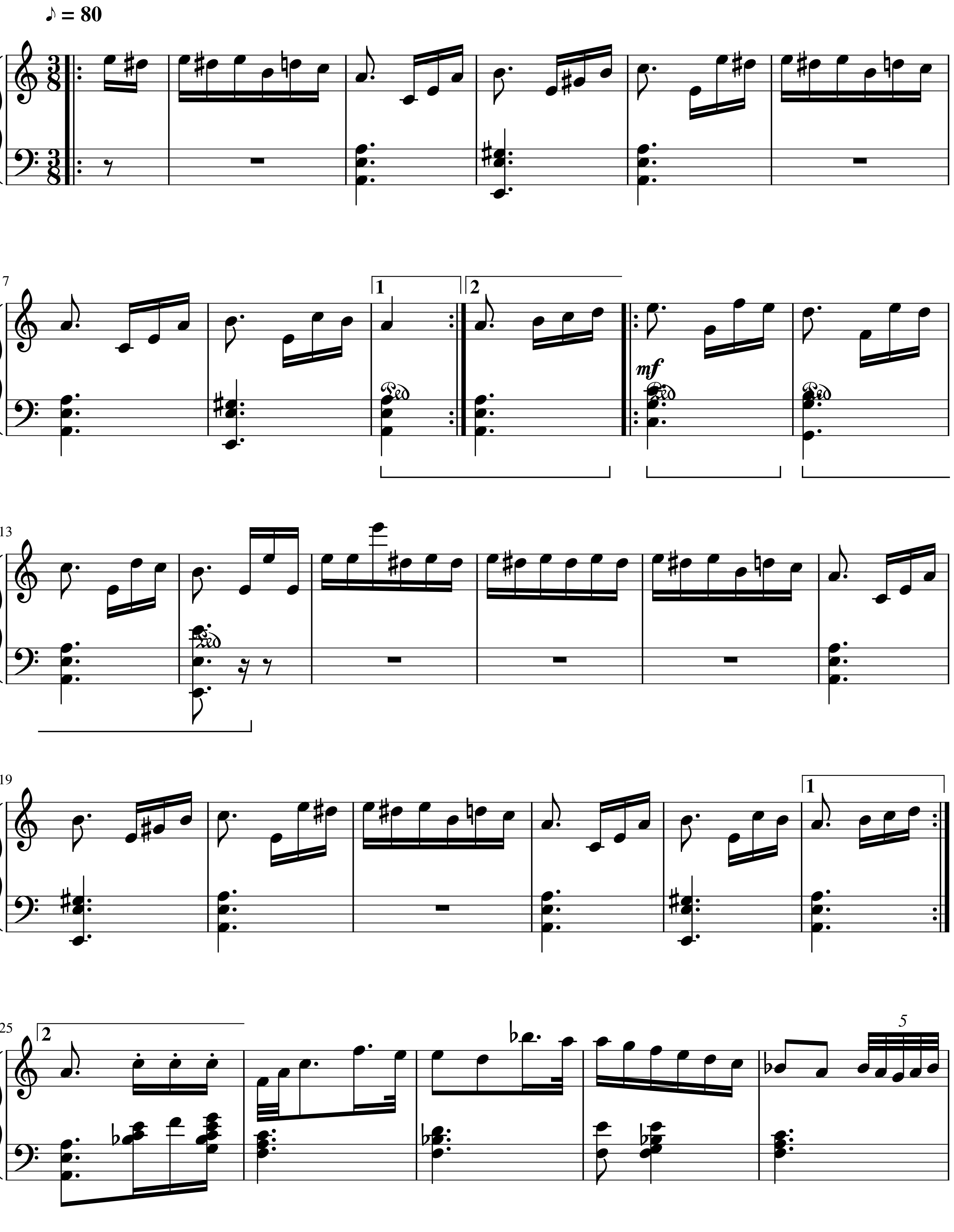

30
Fl.
Pno.
32
Fl.
Pno.
34
Fl.
Pno.
36
Fl.
Pno.
37
Fl.
Pno.
44
Fl.
Pno.

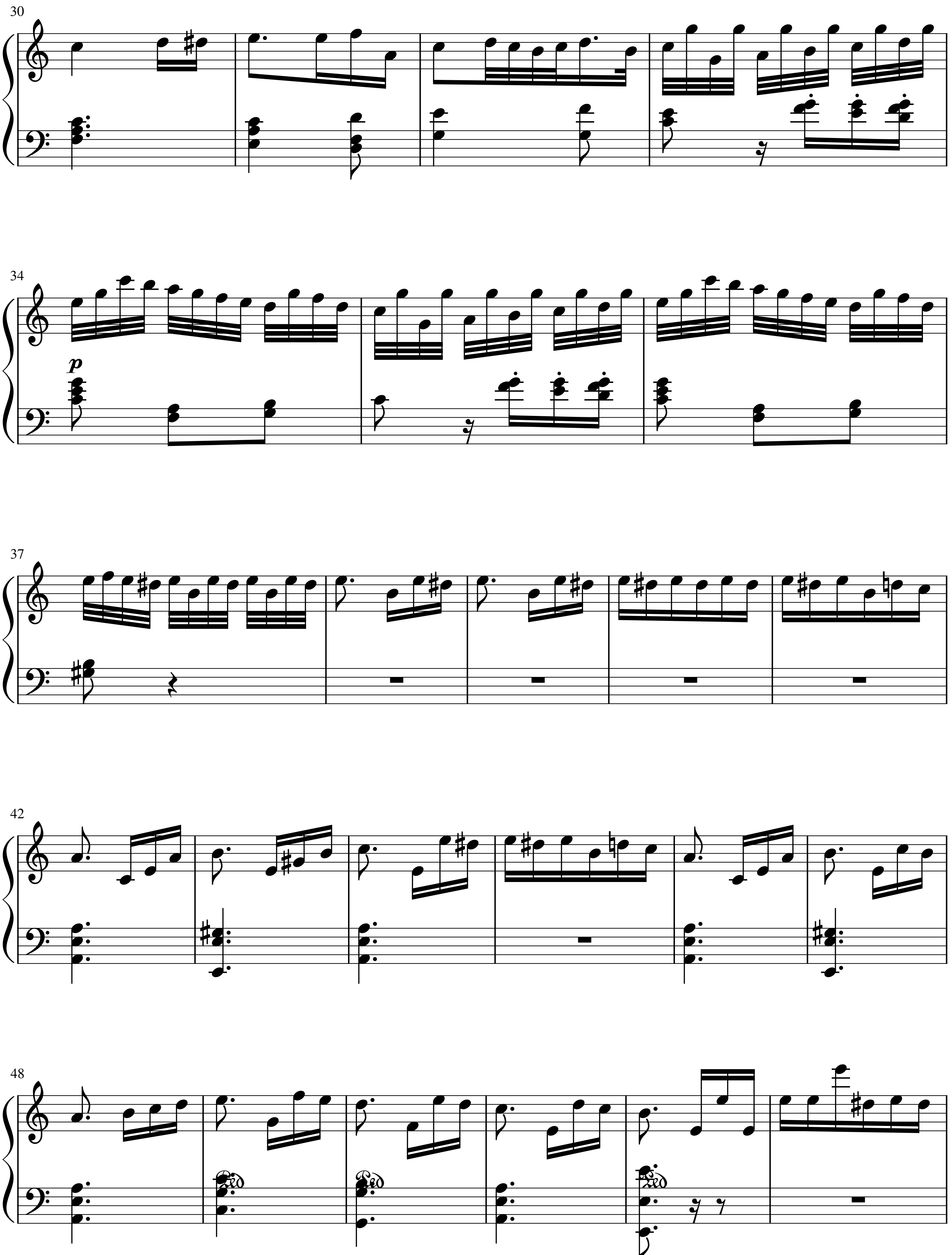

51
Fl.
Pno.

58
Fl.
Pno.

66
Fl.
Pno.

73
Fl.
Pno.

81
Fl.
Pno.

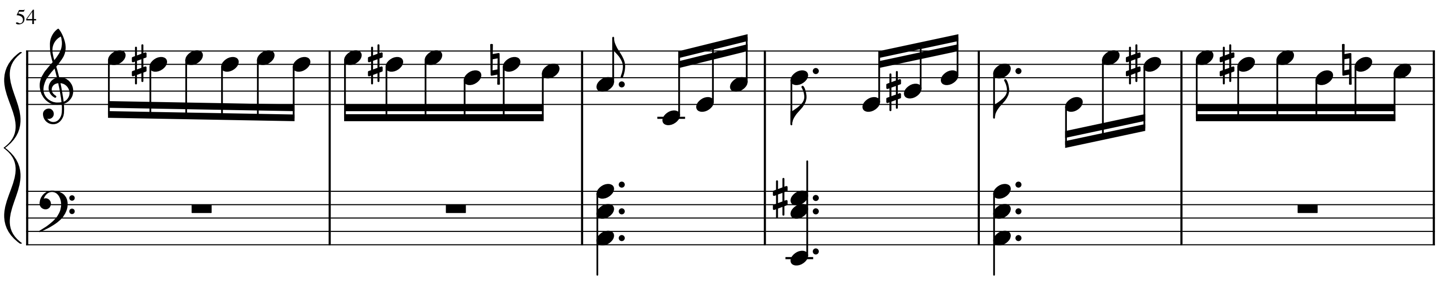
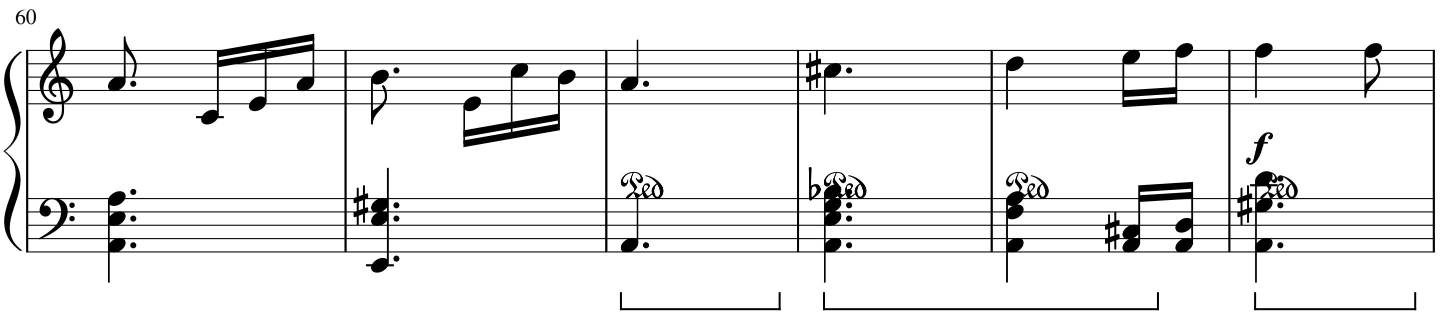
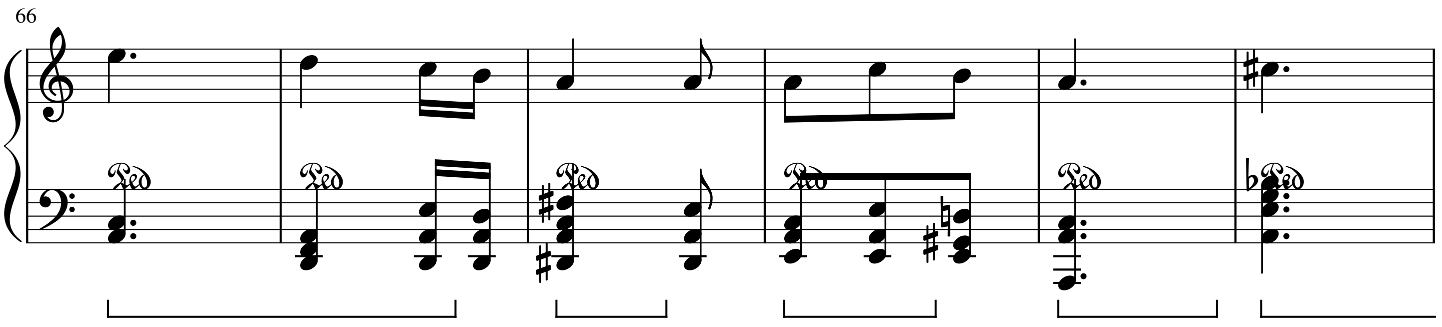
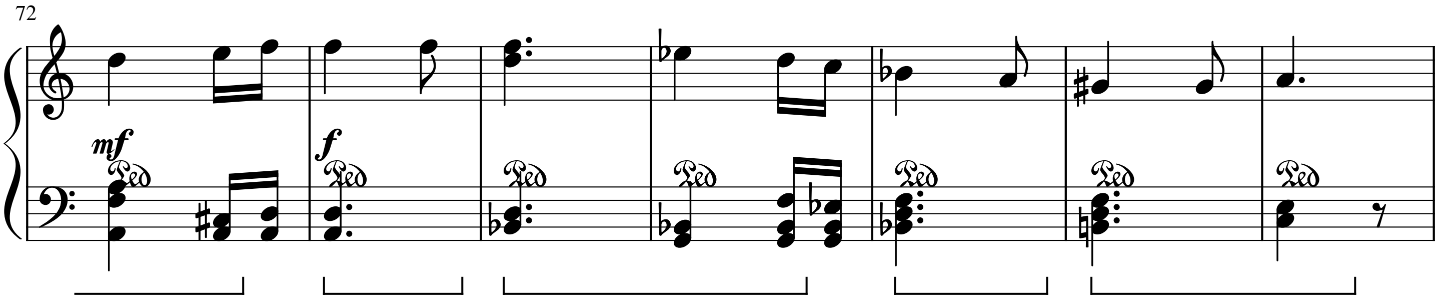
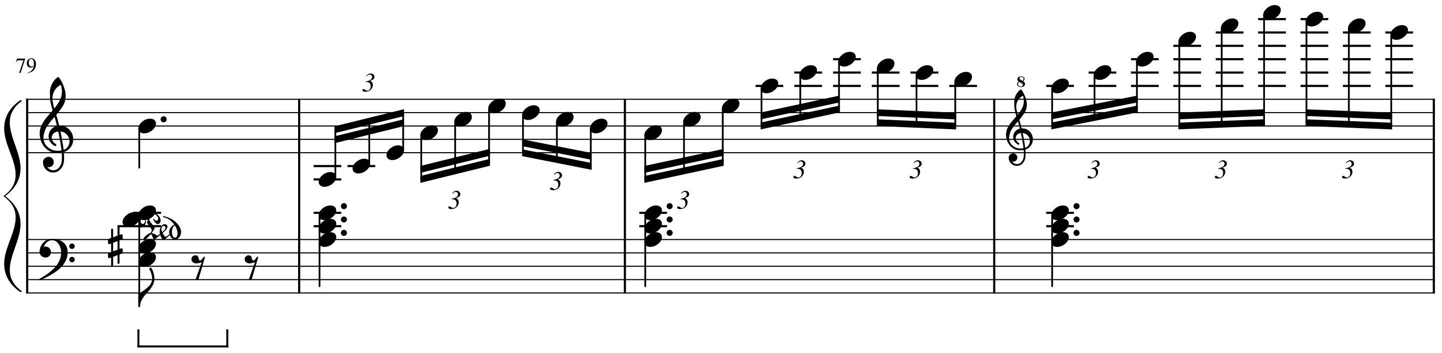
Appendix II: Altered Version of Ludwig Van Beethoven’s Für Elise

Für Elise

L. V. Beethoven




30
34
p
37
42
48
Ped
Ped
Ped


54

60
f
Ped

66
Ped

72
mf
f
Ped

79
3
8
Ped