Automatic Generation of Etude for Violin Players by Means of Quantum Circuits

You Zou

zou@campus.tu-berlin.de

ABSTRACT

Etudes are a type of sheet music that is studied by learners of the violin. Etudes are usually simple in style and relatively even in tempo and are suitable for violin learners to use as warm-up exercises before playing larger and more complicated pieces. In this article, a software based on the automatic generation of etudes pieces by quantum circuits will be introduced. Using this software, the user can generate a random piece of music at a time. The piece of music generated by this software is simple but has some regularity as well as randomness. The software is written in Python and uses the *qiskit* library for quantum computation, the *music21* and *pygame* libraries to match notes and to generate and play music. The code and instructions for using this software can be found at the following link: *https://github.com/zouyou1998/quantummusic*

1. INTRODUCTION

In music learning, an etude is a particular type of sheet music. They are usually short, with a steady rhythm and no major variations. For violin learners, etudes are a good way for beginners to familiarize themselves with musical notation and fingering. They can also be used by professional violinists to warm up before a performance or a larger piece of music. For example, the Chinese amateur violin examination is divided into three parts: scales, etudes and large pieces. Scales, as one of the most elementary types of music, allow the examiner to know the candidate's understanding of intonation, while etudes allow the examiner to perceive not only the candidate's understanding of intonation, but also rhythm and tempo, which are also important. Candidates can also prepare for the third part (the large piece) by playing the Etudes first.

Historically, there are some very famous etudes, such as the Minuet in G, BWV Anh. 114 by Bach[1]. This is an appropriate piece to play as an etude piece, as there are many simple scales in the piece. Chopin's Étude Op. 10, No. 2, on the other hand, is relatively complex and very fast, but a close analysis of the score shows that there are still many adjacent notes and a large number of scales. From a technical point of view, it is not difficult to play for mature players.

[1] The definition of an etude varies from region to region. In China, for example, the Minuet is classified as a type of etude.

In this article, a software based on the automatic generation of sheet music by quantum circuits will be designed. Each run of the software will produce a regular score with randomization. The score will correspond to the notes within the range of the violin, and the generated score will be played at the end.

2. SOFTWARE

This section of the paper describes the software in detail. 2.1 describes some of the ideas that went into the conception of the software, and 2.2 describes the design logic of each part of the software in detail.

2.1 The idea of the overall software

The software can be designed using quantum circuits because the etudes are characterized by "usually simple styles, relatively uniform rhythms and multiple scales". An initial note will be given first, and then the quantum circuit calculates what the next note will be based on the passage of four quantum bits. The probability that the next note will be different from the previous note is only up to three degrees, and the probability that it will be one degree different is inversely proportional to the degree difference. In other words, it is less likely that the next note will be three notes away from the previous note than it is that it will be one note away. This rule was set up to give a "scale-like" score, in keeping with the style of the etudes. Additional rules have been added, such as the upper and lower limits of notes and the probability of repeated notes. These rules are explained in section 2.2.1. The etudes generated by the software will be rhythmically uniform (each note stays the same length) and each note will be close to each other (scale-like characteristics).

This software is divided into three parts, designed in Python. The first part is quantum circuits, and using the *qiskit* library in Python, a list of numbers will be randomly generated by quantum computation according to certain rules. This list of numbers is the next step in generating the prototype of notes that have real meaning. In the second step, this list of numbers is mapped to the corresponding MIDI numbers for normalization, so that the final representation has a consistent musical style. Finally, the standardized

sequence of numbers is used to play the music using Python's *pygame* and *music21* libraries and can be saved as a *.mid* or *.wav* file. The overall structure is shown in figure 1 below. The next section will describe in detail how each part works.

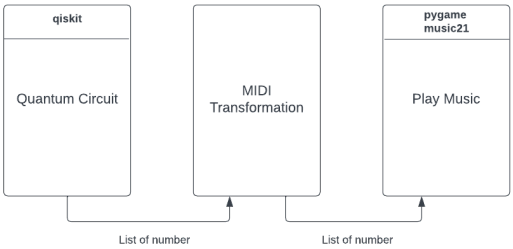

Figure 1. The overall structure of the software, this software is divided into three parts: Quantum Circuit, MIDI Transformation and Play Music

2.2 The detail explanation

2.2.1 The quantum circuits

The first part deals with quantum circuits, and the idea of realizing this part is to first introduce four quantum bits, add Hadamard gates to each of them, and then measure the results, which will produce a binary four-bit number. There are a total of sixteen possibilities for a four-bit binary number, and these sixteen possibilities are mapped into seven different offsets, each of which corresponds to the offset of the next note from the previous note. In quantum computing, Hadamard gates are a very common gate operation that plays an important role in quantum circuits. The main function of Hadamard gates is to convert quantum bits (qubits) from the classical states of 0 and 1 to a state called a "superposition state". It is impossible to predict which state a single-bit system will be in after passing through an H-gate, because in classical computing, a bit can only be in either a 0 or a 1 state [1]. Therefore, the Hadamard gate introduces a kind of "randomness" into quantum computing. It is important to note that this randomness is not random, but due to the properties of superposition states. When a bit is in a superposition state, it does not have a definite value until it is measured but is in a 0 or 1 state with a certain probability. This probability is determined by the principle of superposition of wave functions in quantum mechanics. functions in quantum mechanics. Finally, each note after the offset is calculated to give a preliminary result [2].

This is accomplished by first creating a 4-bit quantum circuit, then adding Hadamard gates to each bit in turn, and then measuring the results. The structure of the circuit can be found in figure 2. The software is set to generate 300 notes in total (the length can be changed at any time). Next, create an empty list called *music_number*, this list will later generate 300 four-bit binary numbers. Next start the simulator for a total of 300 simulations, each set to shot, because in this software it doesn't matter how many shots versus the result, because of the random nature of the highlighted results. For each simulation result, the generated result (e.g., 0010, 1001, 0101 ....) is added to the previously generated music_number.

The grade of the notes is then defined. Because this software is mainly aimed at violin learners. Therefore, firstly, considering that the range of violin is G3-A7, and considering that the practice piece is mainly for warm-up purpose, so the violinist's grip shifting (grip shifting refers to the violinist not only shifting fingers but moving the entire wrist in contact with the instrument in order to expand the range) is not considered, the range is shortened to G3-B5, and then considering the unity of the generated music style, the C major scale is adapted here. There are 17 kinds of notes in the range of G3-B5 with C Major Scale, so now creating a gradient named notes, and add the notes to the generated music_number. Now creating a list named *notes* and assign the value 0-16. then use python's random package to create an initial note (number) *current_note*, the next note (number) will be determined by the current note together with the binary 4-bit number in *music_number* with the following rules:

- If the result of 4 bits from *music_number* is 0000, the new note is 3 steps below the previous note (1/16 probability)
- If the result of 4 bits from *music_number* is 0001, 0010, the new note is 2 steps below the previous note (1/8 probability)
- If the result of 4 bits from *music_number* is 0011, 0100, 0101, 0110, the new note is 1 step lower than the previous note (1/4 probability).
- If the result of 4 bits from *music_number* is 0111, 1000, then the new note is the same as the previous one (1/8 probability)
- If the result of 4 bits is from *music_number* 1001, 1010, 1011, 1100, then the new note is 1 step higher than the previous note (1/4 probability)
- If the result of 4 bits is from *music_number* 1101, 1110, then the new note is 2 steps higher than the previous one (1/8 probability)
- If the result of 4 bits is from *music_number* 1111, then the new note is 3 steps higher than the previous note (1/16 probability)

This shows that except for the 1/8 probability that the next note is the same as the previous one, in all other situation,

the new note is less likely to occur if it is more different than the previous one. The figure 3 below expresses the probability of the next note occurring. If the next note is lower than 0 (G3), the next note will automatically be 0 (G3). If the next note is higher than 16 (B5), the next note will automatically be 16 (B5) if the boundary factor is taken into account. After traversing the results of 300 simulations and stacking them in order, all the results are stored in the list *generated_notes*. This list stores 300 integer numbers from 0 to 16. Each number represents a note corresponding to the C major scale.

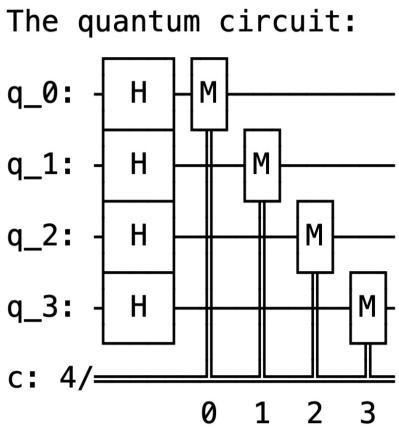

Figure 2. The quantum circuit consists of 4 quantum bits, then adding Hadamard gates to each bit in turn, and then measuring the results

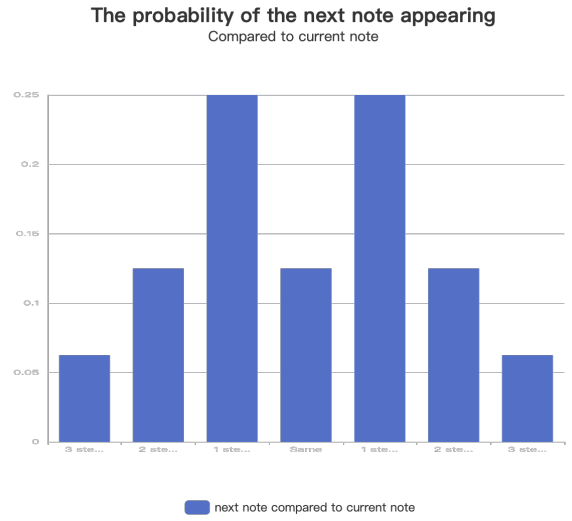

Figure 3. the probability of the next note occurring compared to current note. Except for the 1/8 probability that the next note is the same as the previous one, in all other situation, the new note is less likely to occur if it is more different than the previous one.

2.2.2 MIDI Transformation

The main function of the second part is to correspond the result of the *generated_notes* generated by the first part to the MIDI pitch, and generate a result that conforms to the MIDI standard. This step can be called "normalization" of the notes. As mentioned above, in order to unify the style in this software, C Major Scale will be used to compose the music. And because of the violin range limitation and violin performer's grip shifting limitation, the pitch will be limited between G3 to B5, corresponding to 56 to 84 in the MIDI system. The specific corresponding rules are as shown in Figure 4. The data in *generated_notes* is traversed and mapped, and the final result is stored in a new list called *output_list*.

number in *generated_notes*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
number in *output_list* (MIDI normalization)	56	58	60	61	63	65	67	68	70	72	73	75	77	79	80	82	84

Figure 4. The mapping from *generated_notes* to *output_list* according to C Major Scale in violin range without grip shifting

2.2.3 Play music

The last part converts the generated digital MIDI sequences into *.mid* files and plays the generated music online. Two Python packages are used here: *music21* and *pygame*. First, *pygame* is initialized, then a new music stream object is created with stream from *music21*. Next, *instrument* in *music21* will be used to set up a violin sound. Then loop through the list of notes, using C4 as the base pitch, creating one new note object at a time, and then adding that note object to the music stream object. Next, add the violin sound to the stream object, write it to a MIDI file and save it. The result is a file called *"violin_music.mid"* that contains all the information.

If the user wants to play the music, use the *pygame* package to load the *.mid* file you just created. Then use the *pygame.mixer* function to play the generated music.

3. DISCUSSION

The previous section explained how all three parts work. In this section, 3.1 explains the problems, difficulties and limitations encountered in the design of this software. This is followed in 3.2 by an explanation of how the development of the software can be continued at a later stage.

3.1 Limitations

A number of problems and limitations arose when designing the software. Firstly, the software strictly defines the range and the C major scale that must be used, so if you want to convert to another scale, you have to modify the entire code and re-reference the MIDI correspondences for each note. Second, when a note comes to a critical point, it may stay longer at the critical point because the code takes the critical condition into account, which may limit the final generated music to a small part of the range.

3.2 Outlooks

There are a number of things that could be done to improve the program in the future. First of all, multiple scales could be set up to allow the users to choose which scale they would like to use to produce music. The designer could also create visualization windows that would allow the user to select and produce music more easily.

Second, multifunctional options can be added to allow the user to choose the length of the music, the time range, and the beat control. Finally, conversion can also be set up to convert *.mid* to *.wav* function online. Although there are many online conversion tools in the market, users will save a lot of time by converting directly in this software. The audio files generated by this software are all converted according to the *.mid* generated by the software after the online website (*audio.online-convert.com*). In addition, the developer can also modify the notes generated at the end, such as making every fourth note legato (each note of the violin is one note, but if every fourth note is made legato, it will be played four times at the same time, and this purpose will make the melody more beautiful).

4. SUMMARY

Etudes are a type of sheet music that is studied by learners of the violin. These musical compositions serve as valuable tools for violinists to refine their technical skills and musicality. Typically characterized by their simplicity in style and consistent tempo, etudes offer violin learners the opportunity to engage in focused practice sessions before tackling more complex pieces. In this article, an innovative software that revolves around the automatic generation of etude pieces using quantum circuits is developed. This groundbreaking software introduces a novel approach to music creation, enabling users to generate unique and random compositions at their convenience. The resulting musical pieces produced by this software exhibit a beautiful balance between simplicity and intricacy, incorporating elements of both regularity and randomness. The software has its ability to generate etudes with a touch of randomness opens up new possibilities for violin learners, fostering creativity and musical exploration.

The code and project files required for this software can be found at the following website, which contains this article, the python code and 5 *.mid* files and 5 corresponding music examples converted to *.wav* format:

https://github.com/zouyou1998/quantummusic

5. REFERENCE

.

[1] Shor, P. W. (1998). Quantum computing. Documenta Mathematica, 1(1000), 1.

[2] Steane, A. (1998). Quantum computing. Reports on Progress in Physics, 61(2), 117.