A Fugue Generator

Emilie Klefbom, Fredrik Hallhagen,
Johanna Warnqvist, Viktoria Löfgren

May 2020

1 Introduction

A fugue is a type of composition popular during the baroque era. It is one of the techniques J.S. Bach is most known for. It uses counterpoint, a composing technique for multiple voices, where each voice has a meaningful melody, but still creates a pleasant sounding piece when played together. The fugue is characterized by having a motive, the *subject*, that returns throughout the piece.

We have found a short guide on musikteoriskolan[1] on how to write a simple two-voice fugue, with some of the typical parts. Sadly the web page has expired but we managed to save the textual content. The form of our composition is mainly based on that guide. It includes typical elements such as the *exposition*, the beginning of the piece, where the voices are introduced one by one, playing the subject. It also has parts where the subject is inverted, played "backwards" and in different keys. In the end of the piece, there is a *stretto*, which is a canon on the subject.

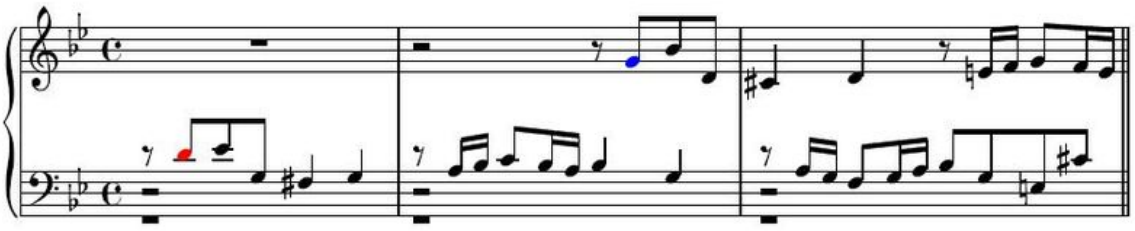

Figure 1: Exposition of Bach's Fugue no 16 in G minor, from The Well-Tempered Clavier [5]

In figure (1) we see a typical exposition. The lower voice introduces the subject, starting at the red note. The higher voice *answers* by playing the subject transposed, starting at the blue note.

2 Method

Our intentions during the project was to first build a simple generator from start to finish and then add to and improve to this model as much as time would allow us. We did this by splitting the work up into the part described below.

2.1 How we represent notes

Our project is built on an open source python package called Mingus[2]. Initially we started with our own object-oriented solution, but due to the time restriction of the project we decided to switch to a pre-built package to minimize the risk of complications later in the project. We choose Mingus since it was comprehensive and had a lot of useful classes for different note objects and also had support for

LilyPond[3] and midi in/output. In hindsight Mingus has probably created more issues than it solved since the LilyPond and midi support didn't work and some of the functions in Mingus are not up to date. We later fixed the midi output on our own and only used LilyPond for testing. We exported midi files into Musescore[4] for better scores in the final report.

2.2 How we generate music

The fugue generation is built up of several parts but can be divided into a pure mathematical part and the evolutionary generator. These parts are combined in our fugue generator to create a fugue either from a given subject or from a randomly generated one. We used the guide from musikteoriskolan[1] to determine the structure and content of the fugue. For example, we always start the first bar with the exposition of the subject[1] in the first voice with pause in the second voice. The second bar then plays the dominant answer[2] and the counter-subject. Throughout the fugue we invert, transpose and reverse. This is done by our Track Functions. These are purely mathematical functions that uses the subject to calculate the correct output. The counter-subject, harmonies and modulation between parts are generated by the evolutionary generator. These are less restricted and more random than the track functions. Some parts of the fugue generator are random selections of predetermined conditions. For example the ending and the generation of a random subject follow some constraints but are much simpler and less "creative" than the evolutionary generator.

2.3 The Evolutionary generator

The core of the project is the evolutionary generator. Evolutionary generation attempts to keep parts of the best melodies in previous generations. To determine which melodies are the best we use a fitness function to assign a score to every melody in a generation. The melodies with the highest fitness scores are then spliced and mutated[3] to form the next generation. The process is then repeated with the new generation being tested, scored, best individuals from that generation being selected, combined and mutated to create the next generation and so on. The probability of mutations and similar to happen in each generation where set in the program inspired by other evolutionary programs, and the population size[4] we used was 100 melodies. The number of generations[5] could be decided for each run as an input.

But the thing that controls the ultimate efficiency and output of the evolutionary generator is the fitness functions. These determine which melodies are good or bad and give it a comparative score.To do this we created test functions that tested different aspects of a melody. Some examples of aspects are: How many notes are in a given scale? How much repetition is there within the melody? Is there a good contrapunctional motion in the melody? and many more. The fitness function combines all of these functions and multiplies them with "weights" that determine how important the particular feature is to generate a "good" fugue. It then goes on to calculate the score for each of these test functions and combine them into a final result or fitness score for a melody. This is then used in the evolutionary generator to pick out the good melodies from a generation. The functions and weight of the fitness functions varies for the different generated parts. For example the modulation fitness function does not care about harmonizing with another melody but instead cares more about the intervals within a melody whereas the harmony fitness function cares mostly about aspects that compares the generated melody and the melody that it will be harmonized with.

[1] The melody we generate the fugue from.
[2] The subject transposed to the dominant.
[3] Random chance of changing the duration or pitch of single notes in a melody.
[4] The number of melodies in each generation that are tested and can be combined.
[5] How many iterations of testing-combining-mutating we do.

3 Optimizing the parameters

After writing the code, the main challenge with the projects was to decide on how to weigh the tests in the fitness functions. An advantage of letting a program compose contrapuntal music is that there already are relatively strict rules on how to do it.

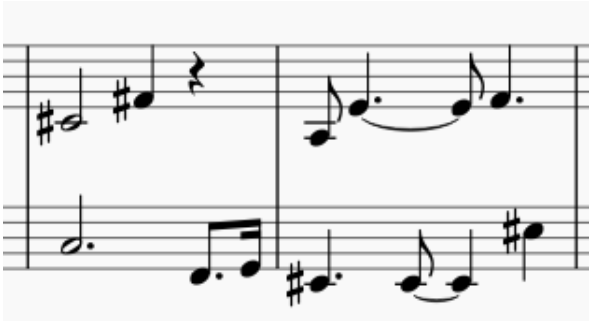

Figure 2: Consonant intervals, both staffs notated in G clef, key C major.

One of the main rules is that the voices have to have consonant intervals between them, preferably thirds and sixths, which are perceived as "rich" intervals. So, one of our tests test if the intervals between the voices are good. In figure (2) we see a snippet from a piece created with heavy weight on this test. It certainly did create consonant intervals, but not in scale. This test does not take the harmonic context into consideration. We have another test that gives positive feedback if the notes are in key, but it has been overridden by the heavy weight on the consonance test.

Another example on a test in the fitness functions is the note duration test. It looks at the note values of the generated parts. The evolution generator is able to generate note values from a sixteenth to a whole note. In the fitness functions we can weigh the points for each note value towards what we consider suitable. We have noticed that in this style of music, quarters, eighths and sixteenths are common so we might want to steer the evolution toward those values. In figure (3) we see an example of music created with heavy weights on those note values. Notice that it is done somewhat at the expense of other qualities, there are many note pitches that aren't in scale, and some weird leaps in the melodies.

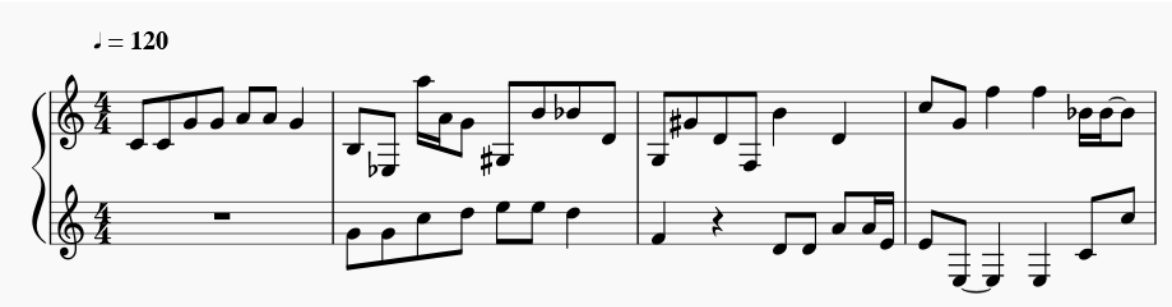

Figure 3: Part of generated fugue when short note values are favored.

If we instead favor longer note values, the program generates pieces like the one seen in figure (4). In these six bars we also see clearly what parts are pre-determined and what parts are generated. The first two bars is the exposition, where the subject (twinkle twinkle little star) is introduced in both voices. In bar five and six, the exposition is repeated but transposed to A minor. These parts are always only dependent on the input subject while the accompanying voices to these parts and both voices in bars three to four are generated with the evolution algorithm.

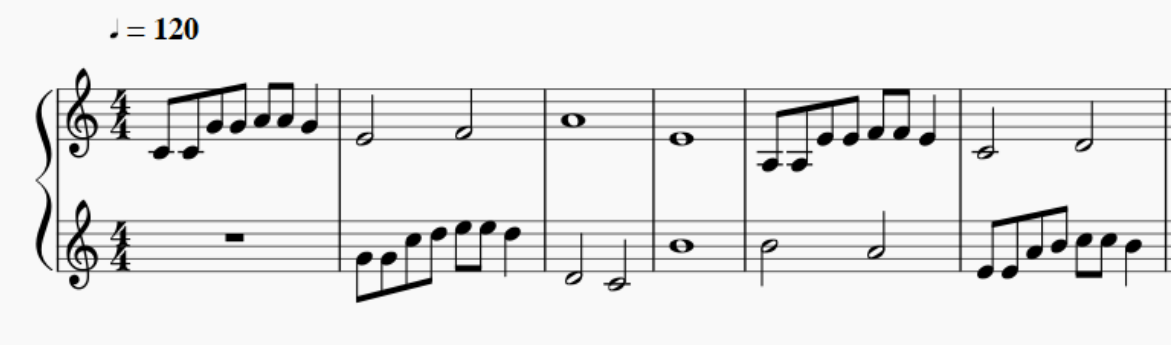

Figure 4: Part of generated fugue when long note values are favored.

Deciding what tests to run and how to weigh them is really the hardest part in creating a good fugue. It is hard to fine tune them so that the generator produces music with exactly the right amount of "creativity". We had an idea to use already existing fugues and see what they would score in our tests, but we had troubles with getting them into our program. That would maybe have given us some insight in how to set the parameters, but we would still not know if our choice of tests is reasonable.

4 Are the fugues from the Fugue Generator "good"?

What is a good fugue, and have we managed to create one? That is a question that have become an important part of our project. A fugue is supposed to follow a lot of rules to be called a fugue, and in our program we have implemented many such rules and asked our generator to try to create something that follows these rules as good as possible. The program will do as we tell it to, but what we have learned during this project is that music is not all about rules. There is no such thing as objective beauty. Composing is a lot about creativity, and also a lot about subjective opinions about what sounds good. Following the given rules too much will not lead to a desired result. A human composer would never follow all of these rules exactly through a whole piece. It is the small imperfect parts that makes the whole piece more perfect.

5 Improvements for Fugue Generator 2.0

The generator as it is, is quite unpredictable. Generating something conventionally pleasant sounding can be hard. We've found this to be mainly due to issues with the evolutionary generator. The evolutionary model we have built is also sensitive to over-fitting[6] when running for too long. If the project was to be redone it could be interesting to use a machine learning approach instead. This would take away the need for a fitness function and problems like over-fitting could easier be solved with for example drop out filters.

6 Final reflection

Our fugues might not be something that a composer would create if told to write something beautiful. But that is exactly the reason why our generator has a value. It is unique in the sense that it has more freedom than a machine learning approach. Unlike machine learning it is not trying to replicate a certain music piece but instead tries to follow these common aspects of music and is free to do so

[6]Over-fitting is the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably - Wikipedia.

in a more unexpected unconventional way. It's not bound by a specific style or composer but instead creates a new questionable genre every time the code is run.

There are already humans that can compose pieces that makes people feel feelings and truly enjoy listening to the pieces. But the generator gives interesting results that makes the listener question our question. Should we really ask if our fugues are good? Or should we just see this new kind of fugues as a way to question the old ways and come up with new, exciting ways of combining notes in a non-traditional and unreasonable way that yet somehow becomes great.

References

[1] Musikteoriskolans guide to create a simple fugue www.musikteoriskolan.se (expired) A saved version from 22 May

[2] Mingus: Open souce python music package http://bspaans.github.io/python-mingus/

[3] LilyPond: Open source score generating software for python https://lilypond.org/

[4] Musescore: Free software that generates music scores https://musescore.org/en

[5] Wikipedia: *Fugue* https://en.wikipedia.org/wiki/Fugue