Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Composição Algorítmica de Fugas ao Estilo de J. S. Bach

José Marcos Alves Menezes

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Aluizio Arcela

Brasília
2008

Universidade de Brasília – UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Guilherme Albuquerque Pinto

Banca examinadora composta por:

Prof. Dr. Aluizio Arcela (Orientador) – CIC/UnB
Prof. Dr. Márcio da Costa Pereira Brandão – CIC/UnB
Prof. Dr. Homero Luiz Piccolo – CIC/UnB

CIP – Catalogação Internacional na Publicação

José Marcos Alves Menezes.

Composição Algorítmica de Fugas ao Estilo de J. S. Bach/ José Marcos Alves Menezes. Brasília : UnB, 2008.
95 p. : il. ; 29,5 cm.

Monografia (Graduação) – Universidade de Brasília, Brasília, 2008.

1. Composição algorítmica, 2. Fuga, 3. Bach,
4. Linguagens para notação musical

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro – Asa Norte
CEP 70910–900
Brasília – DF – Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Composição Algorítmica de Fugas ao Estilo de J. S. Bach

José Marcos Alves Menezes

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Aluizio Arcela (Orientador)
CIC/UnB

Prof. Dr. Márcio da Costa Pereira Brandão
CIC/UnB

Prof. Dr. Homero Luiz Píccolo
CIC/UnB

Prof. Dr. Guilherme Albuquerque Pinto
Coordenador do Bacharelado em Ciência da Computação

Brasília, 27 de junho de 2008

Dedicatória

Dedico este trabalho ao meu orientador Aluizio, minha principal referência, que sempre me incentivou a nunca desistir e me mostrou outras possibilidades musicais.

Agradecimentos

Meus sinceros agradecimentos a Eliza, Yuri, Lehilton, Ígor e Daniel,meus companheiros de jornada, por sempre me acompanhar nesses cinco árduos e longos anos.

Resumo

O presente trabalho implementa a geração computacional automática de fugas, baseada no estilo de compor de J. S. Bach, por crescimento melódico, a partir de transformações melódicas sobre um tema dado, chamado de sujeito da fuga.

A fuga é codificada como um *script* em uma linguagem de notação musical bastante eficiente (*LilyPond*).

A partir desse *script*, utilizando-se o interpretador dessa linguagem, pode-se obter uma representação sônica - por arquivos *MIDI* - ou a partitura gráfica em *PostScript* da fuga gerada.

Palavras-chave: Composição algorítmica, Fuga, Bach, Linguagens para notação musical

Abstract

This project implements the automatic computational generation of fugues, based on the composing style of J. S. Bach, by melodic growth, using melodic transformations on a given theme, called fugue subject.

The fugue is codified as a script using a very efficient musical notation language called *LilyPond*.

From this script, using this language interpreter, we are able to have a sonic representation - through *MIDI files* - or a grafic score in *PostScript* of the generated fugue.

Keywords: Algorithmic composition, Fugue, Bach, Music notation languages

Sumário

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

O presente trabalho aborda a composição algorítmica[9] de fugas, ao estilo de J. S. Bach[1], implementando um programa que realiza o trabalho que um compositor teria para compô-las.

A fuga é considerada por muitos como o mais matemático tipo de composição, principalmente por tratar-se de uma forma musical, ou antes, um roteiro a ser seguido, com várias partes que podem ser combinadas entre si, dependendo para isso somente da inventividade do compositor. Por esse motivo, de ser um "roteiro a ser seguido", um algoritmo, a fuga tem esse grau de abstração tipicamente matemático. Um computador, assim, poderia substituir o homem na tarefa artística[4] de elaborar uma fuga.

Uma fuga é uma composição tipicamente monotemática em que um tema musical (sujeito) é apresentado e desenvolvido, elaborado, variado em toda a extensão da peça[6]. Mesmo não sendo uma regra rígida, todo o material da fuga é derivado do sujeito.

Internamente, a construção da fuga leva em consideração as transformações que seu sujeito pode ter e, externamente, as possíveis concatenações dessas transformações[6].

Essas transformações do sujeito, que em música dá-se o nome de desenvolvimento, são, em última análise, funções que levam o sujeito de um domínio para uma contra-domínio, tendo uma determinada imagem como resultado. Essa analogia de música com matemática norteia todo este trabalho e possibilita, dessa maneira, a implementação de um programa que possa compor satisfatoriamente, computando e concatenando essas transformações.

Levando-se em consideração esses aspectos, a fuga se apresenta como uma forma de arte musical se não de todo, pelo menos em parte computável[3].

Os capítulos que se seguem abordam a constituição de uma fuga; a notação musical utilizada; a analogia do desenvolvolvimento temático com transformações melódicas; o algoritmo desenvolvido e as classe

[1]Compositor alemão que aprimorou e consolidou a forma fuga, sendo considerado por muitos o maior compositor desse estilo, levando a forma às últimas consequências.

implementadas na linguagem JAVA; alguns resultados obtidos e, por fim, a conclusão a que se chegou.

Capítulo 2

Anatomia da Fuga

Fuga, do latim fugare (perseguir) ou fugere (fugir)[8] é um um tipo contrapontístico de composição musical em que o sujeito[1] é apresentado por uma voz isoladamente e, em seguida, é novamente apresentado, em outra tonalidade, por uma segunda voz, enquanto a primeira segue fazendo um contraponto, e assim sucessivamente, por outras vozes, dependendo de seu número[10].

Geralmente uma fuga apresenta de três a cinco vozes, sendo raro, mas não impossível o uso de somente duas ou mais de seis[10]. O termo vozes também pode ser denotado por partes, e não deve ser confundido com vozes humanas. Voz é uma linha melódica da fuga, sendo que um instrumento qualquer pode representá-la e/ou tocá-la.

A fuga é composta por três partes: a exposição, o desenvolvimento e a conclusão.

A exposição é o início da fuga, em que o sujeito será exposto por todas as vozes. É a parte da composição que define a fuga como tal, pois o desenvolvimento e a conclusão podem apresentar diversas formas, combinações, aparências. A exposição tem uma forma fixa, onde cada voz deve expor o sujeito uma vez, necessariamente, enquanto as outras vozes seguem fazendo algum tipo de contraponto.

O tema é chamado de sujeito[8] e apresenta algum interesse rítmico e/ou melódico que será desenvolvido posteriormente e é apresentado na tônica pela primeira voz. Qualquer voz pode iniciar a fuga, mas o sujeito deve, estritamente, na exposição, ser apresentado em todas as vozes.

Ao terminar a declaração do sujeito, a segunda voz entra com a resposta, enquanto a primeira voz faz um contraponto com ela, chamado de contra-sujeito. A resposta é uma imitação do sujeito em outra tonalidade, geralmente a dominante, ocasionalmente a subdominante. A resposta pode ser tonal ou real. Se real, são mantidos, literalmente, os intervalos do sujeito na nova tonalidade; se tonal, são feitas alterações para que fique adequada à nova tonalidade[8].

O contra-sujeito, como visto anteriormente, é um contraponto que

[1] Tema da fuga.

irá acompanhar o sujeito em suas várias declarações, com exceção da primeira, em que o sujeito aparece desacompanhado.

Em uma fuga a três vozes, na exposição ocorrem as entradas sujeito - resposta - sujeito, necessariamente.

A parte central da fuga é chamada de desenvolvimento,onde o sujeito, como o próprio nome sugere, será desenvolvido, elaborado, variado, utilizando-se para isso seus motivos (fragmentos) rítmicos e melódicos.

Episódios são transições modulantes entre uma exposição e outra do sujeito, exposição aqui tomada com o sentido de reexposição temática e não a parte inicial da fuga. Usam de motivos do sujeito em sua composição. Episódios também podem ser chamados de divertimento.

Entra as várias possibilidades de técnicas composicionais, as mais usadas no desenvolvimento de uma fuga são :

- Inversões: os intervalos das notas do sujeito são invertidos, ou seja, o que era ascendente torna-se descendente e vice-versa.
- Movimento retrógrado: o sujeito é declarado de trás pra frente[7].
- Aumento e diminuição: os valores rítmicos das notas do sujeito sofrem um acréscimo ou decréscimo em seus valores, tornando-se mais rápidos ou mais lentos [7].
- Seqüências: padrões rítmicos e melódicos que se repetem em cadeia, geralmente umas três vezes.
- Modulações: mudanças de tonalidade ou de modo (maior-menor).
- Entradas falsas: somente o início do sujeito aparece, dando a impressão de que ele será declarado totalmente.

Não é obrigatório que uma fuga apresente todas as técnicas acima elencadas, podendo, entretanto, apresentar uma combinação dessas técnicas [8].

Por fim, a conclusão pode apresentar um pedal (uma nota longa), geralmente a tônica ou dominante, no baixo (voz mais grave), que soa enquanto as demais vozes encerram a fuga, ou uma recapitulação do sujeito e uma *coda*[2], com uma cadência final, no tom da tônica[3], concluindo, assim, a composição.

[2]Cauda. Finalização de uma peça musical.
[3]Tonalidade em que a peça está escrita.

Capítulo 3

Notação Musical

3.1 Notas

As notas são grafadas como na notação germânica, ou seja, com letras do alfabeto latino. Assim, como correspondência tem-se:

dó = c
ré = d
mi = e
fá = f
sol = g
lá = a
si = b

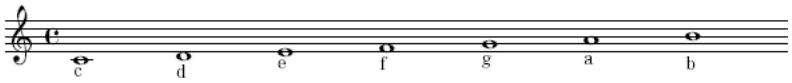

Figura 3.1: *Nomenclatura das notas*

Os sustenidos e bemóis[1] são representados como "is"e "es", respectivamente, e são escritos imediatamente após as notas, sem espaços [1].

Duplos bemóis ou sustenidos[2] são grafados com "isis"ou "eses"[1].

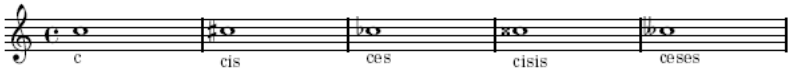

Figura 3.2: *Notação de bemóis e sustenidos*

[1]Os sustenidos e os bemóis são alterações ascendentes e descendentes, respectivamente, nas notas em que são aplicadas. Essas alterações elevam ou diminuem a frequência da nota em um semitom.

[2]Elevam ou diminuem a nota em que está aplicado em um tom.

3.2 Altura

A altura de uma nota é a percepção psicológica do fenômeno físico que denomina-se frequência [2]. Notas com o mesmo nome podem estar em oitavas diferentes. Assim,para diferenciá-las, as oitavas são especificadas colocando-se apóstrofo (”’”) ou vírgula (”,”) após as notas, sendo que cada apóstrofo indica uma oitava superior à central e, cada vírgula, uma inferior [1].

A oitava central é desprovida desses símbolos, e corresponde à do dó 2 do piano, ou seja, o dó do segundo espaço na clave de fá na quarta

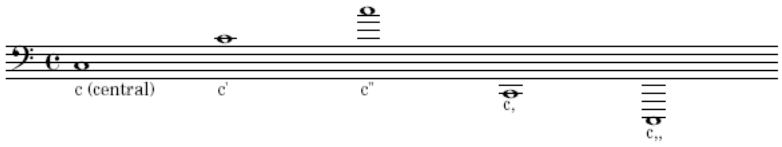

Figura 3.3: *Altura das notas*

3.3 Duração

O duração de cada nota é representada por um número, que é equivalente a alguma figura musical [1]. Assim tem-se, como correspondência:

- 1 = semi-breve
- 2 = mínima
- 4 = semínima
- 8 = colcheia
- 16 = semi-colcheia
- 32 = fusa
- 64 = semi-fusa

O número indicativo de duração deve ser colocado após o nome da nota e altura. A linguagem *LilyPond* permite a omissão dos tempos da nota, adotando para as notas sem número o último valor que porventura tenha aparecido, ou ’4’, como default. Mas adotou-se no presente trabalho a indicação de duração para cada nota.

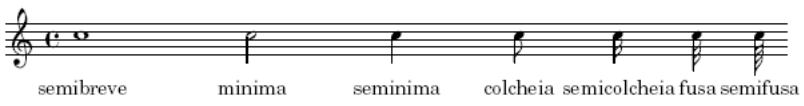

Figura 3.4: *Duração das notas*

3.4 Símbolos

3.4.1 Pausas

As pausas são indicadas com a letra 'r' (rest), adicionadas pelo número indicativo de duração de tempo [1].

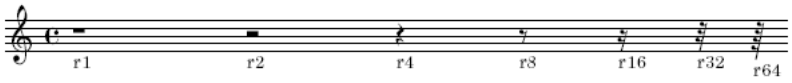

Figura 3.5: *Pausas*

3.4.2 Ligaduras

As ligaduras são indicadas com a o símbolo (~) [1].

Figura 3.6: *Ligadura*

3.4.3 Ponto de aumento

Os pontos de aumento são indicados com ponto simples (.) ou ponto duplo (..) [1].

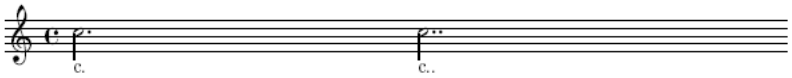

Figura 3.7: *Ponto de Aumento*

3.5 Sobre a linguagem *LiLyPond*

LilyPond é uma linguagem poderosa para notação musical. Permite uma vasta representação de símbolos musicais e extra-musicais, além de possuir um interpretador que permite a geração gráfica de partituras (em *PostScript*) e possibilita, também, a geração de um arquivo *MIDI* com a representação sônica da partitura em questão [1].

3.5.1 O formato .ly

Para que o interpretador da linguagem *LilyPond* funcione corretamente,a música deve ser escrita em um arquivo texto, como o formato apropriado, possuindo a extensão .ly.

Somente alguns aspectos da sintaxe da linguagem *LilyPond* foram usados, mas foram suficientes para poder-se compilar o arquivo gerado pelo programa gerador de fugas e ter como saída a partitura em .pdf e o áudio em .mid.

Para as notas, tem-se que as mesmas devem estar entre chaves ( )[1]. Assim, foi usada uma facilidade da sintaxe *LilyPond* que tem seu funcionamento análogo às atribuições de linguagens de programação, ou seja, pode-se atribuir valores a uma variável e depois usá-la. Assim, as vozes foram definidas como variáveis e as notas geradas pelo programa gerador de fugas foram atribuídas a elas, para clareza do código gerado.

Como exemplo tem-se:
Soprano = ... notas ...
Contralto = ... notas ...
Tenor = ... notas ...

3.5.2 Comandos

O comando \ `score` é o responsável por criar a partitura . Ele "junta"as partes criadas. Dentro dele são adicionados os outros comandos. É análogo à função `main` de algumas linguagens de programação.

O comando \ `midi` permite a geração de um arquivo *MIDI* no momento da compilação do arquivo *.ly*.

Instrumentos podem ser adicionados ao código das vozes para uma maior variedade na execução da música gerada. Esses instrumentos são adicionados pelo comando \ `set Staff.midiInstrument = "nome do instrumento"`.
O arquivo *MIDI* gerado com esse código fará com que a voz a que foi adicionado esse código seja executada com o timbre desse instrumento. Quando não há adição desse comando, por *default* o instrumento para as vozes será o piano.

O interpretador também escolhe, por default, um andamento para a execução do arquivo *MIDI*, assim, para modificar esse valor deve-se adicionar ao código o comando `tempoWholesPerMinute = (ly:make-moment X Y)`, onde X corresponde a quantos Y devem haver em um minuto e Y corresponderia a alguma figura musical.

O comando \ `clef` (clave) adiciona a clave[3] selecionada à voz. As claves usadas foram `treble`, clave de sol, e `bass`, clave de fá na quarta linha.

[3]Claves são símbolos colocados no início das partituras para indicar em qual oitava as notas estão escritas.

3.5.3 Exemplo de código no formado .ly

```
soprano = {

\set Staff.midiInstrument = "church organ"

r2 r2 r2 r2 r2 r4 r4 r2~ r8 r8 r8 r8 a'2 e''2 c''2 a'2
gis'2 a'4 b'4 c''2~ c''8 d''8 c''8 b'8 a'4 a''4 bes''4
c'''4 e''8 f''8 g''8 f''8 f''2~ e''4 d''4 cis''2 d''2
f''2 d''4 a''4 f''4 d''4 cis''4 d''8 e''8 f''4~ f''16
g''16 f''16 e''16 d''2 a''2 f''2 d''2 cis''2 d''4 e''4
f''2~ f''8 g''8 f''8 e''8 g''2 e''2 dis''2 e''4 ges''4
g''2~ g''8 a''8 g''8 ges''8 d'''4 c'''4 b''4 e'''4 d''2
 a''2 f''2 d''2 cis''2 d''4 e''4 f''2~ f''8 g''8 f''8 e''8
r1 r1

}

alto =  {

\set Staff.midiInstrument = "church organ"

d'2 a'2 f'2 d'2 cis'2 d'4 e'4 f'2~ f'8 g'8 f'8 e'8 d'4 e'4
f'4 g'4 b8 c'8 d'8 c'8 c'2~ b4 a4 gis2 a2 c'2 f'2 d'2 cis'2
d'4 e'4 f'2~ f'8 g'8 f'8 e'8 c''4 bes'4 a'4 d'4 e'8 f'8 g'8
f'8 f'2~ e'4 d'4 cis'2 d'2 f'2 a'2 d'2 d'2 a'2 f'2 d'2 cis'2
d'4 e'4 f'2~ f'8 g'8 f'8 e'8 a4 e'4 f'4 g'4 b8 c'8 d'8 c'8 c'2~
b4 a4 gis2 a2 c'2 c'2 a2 gis2 a4 b4 c'2~ c'8 d'8 c'8 b8 g'4 f'4
e'4 a'4

}

```

Código 3.5.1: *Formato . ly*

```
tenor =  {

\set Staff.midiInstrument = "bassoon"

r2 r2 r2 r2 r2 r4 r4 r2~ r8 r8 r8 r8 r2 r2 r2 r2 r2 r4 r4 r2~
r8 r8 r8 r8 d2 a2 f2 d2 cis2 d4 e4 f2~ f8 g8 f8 e8 d1 a1 f1 d1
cis1 d2 e2 f1~ f4 g4 f4 e4 d2 a2 f2 d2 cis2 d4 e4 f2~ f8 g8 f8
e8 r1 r1 r1 r1

}

\score{
        \context StaffGroup <<
                \new Staff <<\clef treble \soprano>>
                \new Staff <<\clef treble  \alto>>
                \new Staff <<\clef bass \tenor>>
        >>
\layout{ }

\midi {

        \context {
                   \Score
                   tempoWholesPerMinute = #(ly:make-moment 120 4)
                }
        }
}
```

Código 3.5.2: *Continuação do formato . ly*

Capítulo 4

Transformações Melódicas

A fuga é uma forma musical extremamente matemática. Uma das possíveis razões para isso talvez seja devido aos momentos temporais das entradas das vozes, que devem estar perfeitamente alinhados. Um outro motivo para esta constatação pode ser devido às transformações que o sujeito sofre ao longo da composição.

Notas musicais podem ser representadas de diversas maneiras possíveis, mas sempre levam em consideração suas três características fundamentais : altura, duração e intensidade.

Em composições musicais é muito comum encontrar-se apenas a notação de altura e duração [5], ficando a intensidade a cargo do intérprete que toca a música. Aqui neste trabalho não foi diferente, a intensidade não foi abordada, por tratar-se de aspecto menos tradicional e observado na composição da fuga.

Com os outros dois parâmetros, altura e duração, é possível toda uma gama de transformações que o sujeito pode sofrer.

No aspecto da duração, as notas podem ser encurtadas ou aumentadas, dilatando ou diminuindo sua duração. Simples multiplicações e divisões são suficientes para acarretar tal transformação. Em tese, o tempo pode ser aumentado ou diminuído em qualquer razão, mas em música o mais comum é a duração ser dobrada ou dividida ao meio, e as fugas usam frequentemente esse tipo de transformação.

Quanto à altura de uma nota, que é o número que representa o valor de sua freqüência, basta multiplicar-se por uma razão para obtermos outra nota, diferente da primeira,ou seja, modula-se essa nota nessa razão.

Pode-se concluir, assim, que as notas podem ser entendidas como tuplas numéricas, e, no caso em questão, entendidas como um par de números, um indicando a freqüência da nota e outro indicando sua duração.

Para fazer-se a transposição das notas num intervalo qualquer, basta multiplicar-se a freqüência da mesma pelo valor da razão do intervalo desejado. Mas para isso seria necessário ter-se esse valor de freqüência para cada nota, o que seria inviável pela notação utilizada. Se assim fosse, cada nota seria representada por um número, indica-

tivo de sua frequência, e não pelos doze nomes que atribuí-se a elas.

Mas essa notação para a representação das notas, qual seja, a utilização de uma cadeia de caracteres com seus respectivos nomes em notação germânica (c, d,e...), permite uma abordagem em aritmética modular para as mesmas, já que pode-se associar um número inteiro a cada nota, e tendo somente 12 notas, pode-se fazer operações mod 12 com as mesmas, pois elas sempre se repetem na mesma ordem. Assim, tem-se para o presente trabalho a seguinte associação [1]:

c = 0
cis-des = 1
d = 2
dis-ees = 3
e = 4
f = 5
fis-ges = 6
g = 7
gis-aes = 8
a = 9
ais-bes = 10
b = 11

Um *array* que guarde essas notas, nessa ordem, tem a vantagem de o índice de cada posição ser o valor que as representa.

As operações em aritmética modular tem a propriedade de sempre retornar o nome da nota, independente da oitava na qual esteja, simplificando assim a multiplicação das notas por uma razão qualquer, que faria o papel das transposições.

O cálculo das oitavas é feito através de somas e subtrações dos apóstrofos e vírgulas, que representam em qual oitava a nota está.

As transformações melódicas mais imediatas são as transposições, em que um trecho musical tem suas freqüências multiplicadas (ou divididas) por uma razão específica.

4.1 Resposta

A resposta, numa fuga, é a transformação que leva o sujeito, no tom da tônica [1], ao tom da dominante [2] ou quinto grau. Todas as notas do sujeito são transpostas, ou seja, multiplicadas, pela razão do quinto grau.

Para isso, basta somar ao valor do índice que representa a nota o valor 7, que corresponde ao quinto grau, tendo assim a transposição ao tom da dominante. Uma operação em aritmética modular (*mod 12*) deve ser feita para que a soma desses valores não ultrapasse a extensão

[1] Tonalidade em que a peça musical está escrita.
[2] Tonalidade do quinto grau da escala na qual a peça está escrita.

do *array* que guarda os valores das notas, e devido a circularidade dessa representação, retorne a nota correta.

4.2 Contra-sujeitos

Os sujeitos das fugas utilizados neste trabalho nunca preenchem o último compasso em que aparecem, por terminarem em um tempo forte do mesmo. Assim, o contra-sujeito, que se inicia imediatamente após a apresentação do sujeito, deve completar os tempos faltantes. Para isso, utilizaram-se as últimas notas do sujeito em movimento retrógrado, ou seja, de trás para frente, e, para não se depender do valor da duração dessas notas, que poderia não completar esse último compasso, seus valores de duração foram modificados para se tornarem semínimas.

Completado esse último compasso, o contra-sujeito 1 foi elaborado com as notas da resposta em movimento retrógrado, de tal forma que estivesse concatenado com a mesma, apresentada por outra voz.

O contra-sujeito 2 foi construído com as notas do contra-sujeito 1 em movimento retrógrado. Como o tamanho do contra-sujeito 1 é menor que o do sujeito, precisa-se, no fim da construção, preencher o contra-sujeito 2 com mais algumas notas, retiradas do sujeito, em movimento retrógrado e com a duração modificada para semínima.

As funções inversas dessas transformações melódicas tentam reverter seu processo de construção, numa tentativa de se obter as partes de que se originaram. No caso do contra-sujeito 1, a resposta, e no caso do contra-sujeito 2, o contra-sujeito 1.

4.3 Aumento

As notas do sujeito tem o valor de suas durações duplicadas.

4.4 Diminuição

As notas do sujeito tem suas durações divididas pela metade.

4.5 Retrógrado

As notas do sujeito são armazenadas de trás para frente.

4.6 Transformações inversas

Somente o sujeito não possui transformação inversa, por ser o gerador de toda a fuga. Assim, as partes da fuga e as transformações

melódicas, como foram definidos, possuem uma transformação inversa, podendo, a partir dos mesmos, obter-se a parte que os gerou.

Capítulo 5

Algoritmo do Programa *FugueMaker*

5.1 Propriedades do sujeito

A única entrada para o programa gerador de fugas é um arquivo texto contendo a representação musical de um sujeito qualquer.Mas, para o correto funcionamento do programa, esse sujeito tem que apresentar certas características, ou seja, ter uma boa construção. A composição de uma fuga deve levar em consideração a estrutura do sujeito, pois nem todos são aptos a tal empreitada.[10]

A boa construção do sujeito significa que o mesmo deve ser em algum compasso binário ou quaternário; não apresentar tempos terciários internamente, tais como tercinas e quiálteras; terminar sempre em tempo forte, ou seja, no primeiro de compassos binários ou primeiro e terceiro de compassos quaternários.

O uso de compassos binários ou quaternários se deve ao fato de tornar o cálculo das partes da fuga mais fácil e suas entradas no tempo e tamanhos ficarem mais simples de se lidar.

Tempos terciários não são incomuns em fugas, mas, também, exigiriam cálculos mais elaborados e complexos, que necessitariam de mais tempo em sua elaboração.

O aspecto de sempre terminar o sujeito em um tempo forte foi constatado a partir de observações em fugas bachianas [5]. Não se pode afirmar tal fato como uma lei, devido ser impossível e inorpotuno para o presente trabalho analisar a totalidade de fugas do compositor alemão, mas tomando algumas peças do *Cravo Bem Temperado e da Arte da Fuga* como guias, levou-se isso em consideração para a escolha dos sujeitos utilizados no presente trabalho. Esse fato observado tornou mais fácil o cálculo do momento de entrada dos sujeitos.

5.2 O algoritmo

A entrada deve ser um arquivo texto com a cadeia de caracteres representando o sujeito da fuga.

A partir dessa entrada são geradas as partes da fuga: sujeito, resposta, contra-sujeito 1 e 2 e suas inversas.

O sujeito, agora, é passado como parâmetro, em um ArryList, para o módulo que gerará as transformações melódicas do mesmo: aumento, diminuição, pausas e retrógrado.

Todas essas operações são auxiliadas por funções implementadas em outras classes.

Essas partes, uma vez geradas, devem ser concatenadas corretamente. A classe responsável por essa concatenação vai adicionando as partes e as transformações melódicas correspondentes a cada voz ao arquivo de saída, e, na medida em que faz isso, utilizando-se de uma pilha, empilha a transformações inversas das mesmas,para posterior desempilhamento.

Uma vez acrescentadas as partes e as transformações melódicas ao arquivo de saída, de tal maneira que a forma da fuga seja obedecida, passa-se ao desempilhamento da aplicação das funções inversas, até que as pilhas de cada voz estejam vazias.

Através da interface gráfica, também são passados valores para que a execução da fuga, através do arquivo MIDI gerado, seja mais variada. Esses valores, códigos da linguagem LilyPond, são adicionados ao arquivo de saída.

Terminado esse processo, o arquivo com a representação da fuga está gerado.

Faz-se uma chamada externa ao programa LilyPond para a compilação do arquivo com a fuga, gerando um arquivo .pdf, com a partitura e outro arquivo .mid, para sua execução.

5.3 Considerações

Esse algoritmo, da forma como foi implementado, garante que a exposição da fuga seja exatamente como sua definição, ou seja, que cada voz exponha o sujeito uma vez. Terminada a exposição, a parte do desenvolvimento da fuga, menos rígida, foi construída adicionando-se uma transformação melódica a cada voz, e, em seguida, desempilhando as funções inversas empilhadas.

Não foi implementada a conclusão da fuga.

Capítulo 6

Implementação das Classe do Programa *FugueMaker*

Foi usada a tecnologia Java para a implementação do programa `FugueMaker` devido às facilidades que propicia no uso de classes já implementadas, exigindo somente o aprendizado das mesmas. Os pacotes gráficos e sônicos que também já existem liberam o programador para se ater somente aos aspectos relevantes de seu projeto.

6.1 Implementação da Classe `Constantes`

Contém as constantes que representam os intervalos de transposição, as oitavas, se acima ou abaixo, e os arrays representando as escalas.

```
1  package fuguegenerator;
2  import java.io.*;
3  import java.util.*;
4
5  public interface Constantes {
6
7      public static final int UNISSONO = 0;
8      public static final int SEGUNDA_MENOR = 1;
9      public static final int SEGUNDA_MAIOR = 2;
10     public static final int TERCA_MENOR = 3;
11     public static final int TERCA_MAIOR = 4;
12     public static final int QUARTA_JUSTA = 5;
13     public static final int TRITONO = 6;
14     public static final int QUINTA_JUSTA = 7;
15     public static final int SEXTA_MENOR = 8;
16     public static final int SEXTA_MAIOR = 9;
17     public static final int SETIMA_MENOR= 10;
18     public static final int SETIMA_MAIOR = 11;
19
20     public static final int OITAVA_ABAIXO = 0;
21     public static final int OITAVA_ACIMA  = 1;
22
23     public static final String[] escala_bemol =
24
25         {"c","des","d","ees","e","f","ges","g","aes","a","bes","b"};
26
27     public static final String[] escala_sustenido =
28
29         {"c","cis","d","dis","e","f","fis","g","gis","a","ais","b"};
30
31 }
```

Código 6.1.1: *Implementação da Classe* `Constantes`

6.2 Implementação da Classe `Arquivo`

Esta classe possui as funções para abrir e gravar arquivos.

```
1  public void abrirArquivo(String nome_arquivo){
2
3          int tamanho_buffer = 0;
4          File entrada  = new File(nome_arquivo);
5          FileInputStream fis = null;
6
7          try {
8                  fis = new FileInputStream(entrada);
9          } catch (FileNotFoundException ex) {
10                 ex.printStackTrace();
11         }
12
13         DataInputStream dis = new DataInputStream(fis);
14
15         try {
16                 tamanho_buffer = dis.available();
17         } catch (IOException ex) {
18                 ex.printStackTrace();
19         }
20
21         buffer = new char[tamanho_buffer];
22
23
24         for(int i = 0; i < tamanho_buffer; i++){
25             try {
26                     buffer[i] = (char)dis.read();
27
28                     System.out.print(buffer[i]);
29                 } catch (IOException ex) {
30                         ex.printStackTrace();
31                 }
32         }
33
34 }
35
36
```

Código 6.2.1: *Implementação do método de abertura de arquivos*

```
1  public void gravarArquivo(String nome_arquivo){
2
3
4          File apaga = new File(".",nome_arquivo);
5
6          apaga.delete();
7
8          File saida  = new File(nome_arquivo);
9          FileOutputStream fos = null;
10
11         try {
12                 fos = new FileOutputStream(saida,true);
13         } catch (FileNotFoundException ex) {
14                 ex.printStackTrace();
15         }
16
17         DataOutputStream dos = new DataOutputStream(fos);
18
19         for(int i = 0; i < buffer.length; i++){
20             try {
21                     dos.write(buffer[i]);
22                 } catch (IOException ex) {
23                         ex.printStackTrace();
24                 }
25         }
26
27         try {
28                 dos.close();
29         } catch (IOException ex) {
30                 ex.printStackTrace();
31         }
32 }
```

Código 6.2.2: *Implementação do método de gravação de arquivos*

6.3 Implementação da Classe `Arquivo_Lily`

Classe para gerar a partitura no formato *LilyPond*.

Apresenta os códigos dos instrumentos musicais, para que o arquivo `.mid` execute a fuga com o timbre escolhido associado a cada voz.

```
1  public String violin = "\\set Staff.midiInstrument = \"violin\" \n";
2
3  public String viola = "\\set Staff.midiInstrument = \"viola\" \n";
4
5  public String cello = "\\set Staff.midiInstrument = \"cello\" \n";
6
7  public String flute = "\\set Staff.midiInstrument = \"flute\" \n";
8
9  public String clarinet = "\\set Staff.midiInstrument = \"clarinet\" \n";
10
11 public String bassoon = "\\set Staff.midiInstrument = \"bassoon\" \n";
12
13 public String xylophone = "\\set Staff.midiInstrument = \"xylophone\" \n";
14
15 public String oboe = "\\set Staff.midiInstrument = \"oboe\" \n";
16
17 public String glockenspiel = "\\set Staff.midiInstrument = \"glockenspiel\" \n";
18
19 public String church_organ = "\\set Staff.midiInstrument = \"church organ\" \n";
20
21 public String pizzicato_strings = "\\set Staff.midiInstrument =
22                                                 \"pizzicato strings\" \n";
23
24 public String harpsichord = "\\set Staff.midiInstrument = \"harpsichord\" \n";
25
26 public String dulcimer = "\\set Staff.midiInstrument = \"dulcimer\" \n";
27
28 public String trumpet = "\\set Staff.midiInstrument = \"trumpet\" \n";
```

Código 6.3.1: *Códigos dos instrumentos musicais utilizados*

```
1  public String textoMidiAbre = "\\midi {\n";
2
3  public String textoMidiFecha = " }\n"+
4                                    "}";
5
6  public String tempoMidiAbre = "\\context { \n"+
7                                   "\\Score \n"+
8                                   "tempoWholesPerMinute = #(ly:make-moment ";
9
10 public String tempoMidiFecha = " 4) \n"+
11                                  } ";
```

Código 6.3.2: *Texto para geração do arquivo .mid*

Possui uma variável (`arquivoPrincipal`) na qual toda inserção de código vai sendo colocada através do método `adicionarTexto(String s)`.

```
1  private String arquivoPrincipal = new String();
```

Código 6.3.3: *Variável que guarda o texto do arquivo .ly a ser gerado*

```
1  public void adicionarTexto(String s){
2
3          arquivoPrincipal += ("\n"+s);
4
5  }
```

Código 6.3.4: *Implementação do método para adicionar texto à variável `arquivoPrincipal`*

6.4 Implementação da Classe `TocadorMidi`

Classe criada para tocar o arquivo .mid. De simples implementação, somente abre o arquivo .mid e o toca e/ou pára a execução que porventura esteja em andamento.

```
package fuguegenerator;

import javax.sound.midi.*;
import java.io.*;


public class TocadorMidi {

    private static Sequencer    sequencer = null;
    private static Synthesizer  synthesizer = null;
    private static Sequence     sequence = null;
    private String nome_arq_Midi = new String();

    public TocadorMidi(String nome_arq) {

        this.nome_arq_Midi = nome_arq;
        inicializaMidi();
    }

    public void inicializaMidi(){

        String  strFilename = nome_arq_Midi;
        File    midiFile = new File(strFilename);
        MidiFileFormat   fileFormat = null;



        try {
                fileFormat = MidiSystem.getMidiFileFormat(midiFile);
        } catch (InvalidMidiDataException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
```

Código 6.4.1: *Implementação da Classe* `TocadorMidi`

```
        try
        {
                sequence = MidiSystem.getSequence(midiFile);

        }
        catch (InvalidMidiDataException e)
        {

                e.printStackTrace();
                System.exit(1);
        }
        catch (IOException e)
        {
                e.printStackTrace();
                System.exit(1);
        }

        try
        {
                sequencer = MidiSystem.getSequencer();

        }
        catch (MidiUnavailableException e)
        {
                e.printStackTrace();
                System.exit(1);
        }

        try
        {
                sequencer.open();
        }
        catch (MidiUnavailableException e)
        {
                e.printStackTrace();
                System.exit(1);
        }
```

Código 6.4.2: *Continuação da implementação da Classe* `TocadorMidi`

```
            try
            {
                    sequencer.setSequence(sequence);
            }
            catch (InvalidMidiDataException e)
            {
                    e.printStackTrace();
                    System.exit(1);
            }


            if (! (sequencer instanceof Synthesizer))
            {
                    try
                    {
                            synthesizer = MidiSystem.getSynthesizer();
                            synthesizer.open();
                            Receiver        synthReceiver = synthesizer.getReceiver();
                            Transmitter     seqTransmitter = sequencer.getTransmitter();
                            seqTransmitter.setReceiver(synthReceiver);
                    }
                    catch (MidiUnavailableException e)
                    {
                            e.printStackTrace();
                    }

            }

        }

        public void toca(){
            sequencer.start();
        }

         public void para(){
            sequencer.stop();
        }
    }
```

Código 6.4.3: *Continuação da implementação da Classe* `TocadorMidi`

6.5 Implementação da Classe `FuncoesAuxiliares`

Esta classe possui os métodos para auxiliar todos os processos envolvidos na geração da fuga.

O método `getLiteral(String s)` retorna a cadeia de caracteres que representa a nota que foi passada como parâmetro, sem qualquer símbolo adicional ou número indicativo de duração.

O método `getTempoNota(String s)` retorna o valor numérico da duração da nota passada como parâmetro.

```
public static String getLiteral(String s){

    String literal = new String();

    for(int k = 0; k < s.length(); k++){
        if(s.charAt(k) == '0' ||
           s.charAt(k) == '1' ||
           s.charAt(k) == '2' ||
           s.charAt(k) == '3' ||
           s.charAt(k) == '4' ||
           s.charAt(k) == '5' ||
           s.charAt(k) == '6' ||
           s.charAt(k) == '7' ||
           s.charAt(k) == '8' ||
           s.charAt(k) == '9' ||
           s.charAt(k) == '~' ||
           s.charAt(k) == '.' ){/*nao faz nada*/}
        else literal += s.charAt(k);
    }
    return literal;
}
```

Código 6.5.1: *Implementação do método `getLiteral()`*

```
public static double getTempoNota(String s){

        String numeral = new String();
        int ponto = 0;
        int denominador = 0;
        double retorno;

        for(int k = 0; k < s.length(); k++){
            if(s.charAt(k) == '0' ||
               s.charAt(k) == '1' ||
               s.charAt(k) == '2' ||
               s.charAt(k) == '3' ||
               s.charAt(k) == '4' ||
               s.charAt(k) == '5' ||
               s.charAt(k) == '6' ||
               s.charAt(k) == '7' ||
               s.charAt(k) == '8' ||
               s.charAt(k) == '9' ){

                numeral += s.charAt(k);
            }
            else if (s.charAt(k) == '.'){
                ponto++;
            }
        }

        denominador = Integer.parseInt(numeral);
        retorno = (1 / (double)denominador);
        if(ponto > 0){
            if(ponto == 1){
                 retorno += (1 / ( (double)denominador * 2));
            }
            if(ponto == 2){
                retorno += (1 / ((double)denominador * 2));
                retorno += (1 / ((double)denominador * 2 * 2));
            }
        }

        return retorno;

}
```

Código 6.5.2: *Implementação da método* `getTempoNota()`

Os métodos `getNumCompassos(ArrayList l)`, `getCompassosCheios(ArrayList l)` e `getValorPreencherCompasso(ArrayList l)` retornam o número de compassos com a parte fracionária, se o último compasso não estiver completo; o número de compassos cheios efetivamente, ou seja, sem a parte fracionária; e o valor de duração que falta para preencher o último compasso, se incompleto, dos `ArrayList` de entrada, respectivamente.

```
    public static double getNumCompassos(ArrayList l){

        double retorno = 0.0;

        for(int i = 0; i < l.size(); i++){

            retorno += getTempoNota((String) l.get(i));
        }
        return  retorno;
    }
```

Código 6.5.3: *Implementação do método* `getNumCompassos()`

```
public static int getCompassosCheios(ArrayList l){

        return (int) getNumCompassos(l);
}
```

Código 6.5.4: *Implementação do método* `getCompassosCheios()`

```
public static double getValorPreencherCompasso(ArrayList l){

        double retorno = 0.0;
        double resto = 0.0;


        resto = getNumCompassos(l) - getCompassosCheios(l);

        if(resto == 0.0){
            retorno = 0.0;
        }
        else{
            retorno = (1 - resto);
        }

        return retorno;
}
```

Código 6.5.5: *Implementação do método* `getValorPreencherCompasso()`

O método `getPosicaoNota(String s)` retorna o valor numérico da posição da nota passada como parâmetro na escala, ou seja, o índice do `array` que representa a escala que está na classe Constantes.

```
public static int getPosicaoNota(String s){

        int retorno = 0;
        String literal = new String();

        for(int k = 0; k < s.length(); k++){
            if(s.charAt(k) == '0' ||
               s.charAt(k) == '1' ||
               s.charAt(k) == '2' ||
               s.charAt(k) == '3' ||
               s.charAt(k) == '4' ||
               s.charAt(k) == '5' ||
               s.charAt(k) == '6' ||
               s.charAt(k) == '7' ||
               s.charAt(k) == '8' ||
               s.charAt(k) == '9' ||
               s.charAt(k) == '\''||
               s.charAt(k) == ',' ||
               s.charAt(k) == '~' ||
               s.charAt(k) == '.' ){/*nao faz nada*/ }
            else literal += s.charAt(k);
        }


        for(int posicao = 0; posicao < 12; posicao++){


            if(literal.equals(Constantes.escala_bemol[posicao])){
                retorno = posicao;
            }

            if(literal.equals(Constantes.escala_sustenido[posicao])){
                retorno = posicao;
            }
        }
        return retorno;
}
```

Código 6.5.6: *Implementação do método* `getPosicaoNota()`

O método `calculaInversao(int a, int b)` recebe como parâmetros a posição de duas notas e calcula a inversão da segunda em relação à primeira, ou seja, retorna o índice da nota invertida na escala, ou de bemóis ou de sustenidos, da classe Constantes, na qual esteja.

```
public static int calculaInversao(int a, int b){

        int vetorPosicao[] = new int[12];
        int posicaoSegundaNota;
        int posicaoInvertida = a;

        // tabela com a posicao 'a' valendo zero

        for(int i = 0; i < 12; i++){
            if(i >= a){
                vetorPosicao[i] = (i - a);
            }
            else if (i < a){
                vetorPosicao[i] = ((12 - a) + i);
            }
        }

        posicaoSegundaNota = vetorPosicao[b];

        for(int i = 0; i < 12; i++){

            if( vetorPosicao[i] == ((12 - posicaoSegundaNota) % 12)){
                posicaoInvertida = i;
            }
        }


        return posicaoInvertida;
}
```

Código 6.5.7: *Implementação do método* `calculaInversao()`

O métodoo `imprime(ArrayList l)` imprime em tela as `strings` contidas no `ArrayList` de entrada.

```
    public static void imprime(ArrayList l){

        System.out.print("\nImprimindo:\n");
        for(int i = 0; i < l.size(); i++){
           System.out.print(l.get(i) +" ");
        }
        System.out.println();
    }
```

Código 6.5.8: *Implementação do método* `imprime()`

O método `modificaUltimaNota(ArrayList l)` altera o valor numérico do tempo da última nota para 4, ou seja, transformando-a em uma semínima.

```
public static void modificaUltimaNota(ArrayList l){

        String notaModificada = new String();
        String literal = new String();
        String numeral = "4";

        int posicao = l.size();
        String s = new String();

        s = (String) l.get(posicao - 1);

        for(int k = 0; k < s.length(); k++){
            if(s.charAt(k) == '0' ||
               s.charAt(k) == '1' ||
               s.charAt(k) == '2' ||
               s.charAt(k) == '3' ||
               s.charAt(k) == '4' ||
               s.charAt(k) == '5' ||
               s.charAt(k) == '6' ||
               s.charAt(k) == '7' ||
               s.charAt(k) == '8' ||
               s.charAt(k) == '9' ||
               s.charAt(k) == '~' ||
               s.charAt(k) == '.' ){/*nao faz nada*/}
            else literal += s.charAt(k);
        }

        l.remove(posicao - 1);
        notaModificada = literal + numeral;
        l.add(notaModificada);
 }
```

Código 6.5.9: *Implementação do método* `modificaUltimaNota()`

O método `retiraUltimaNota(ArrayList l)`, como o nome sugere, retira a última nota do ArrayList passado.

```
public static void retiraUltimaNota(ArrayList l){

        l.remove(l.size() - 1);
}
```

Código 6.5.10: *Implementação do método* `retiraUltimaNota()`

O método `copiaLista(ArrayList l)` retorna um `ArrayList` idêntico ao passado como parâmetro.

```
public static ArrayList copiaLista(ArrayList l){

        ArrayList retorno = new ArrayList();

        for(int i = 0; i < l.size(); i++){

            retorno.add(i, l.get(i));
        }

        return retorno;
}
```

Código 6.5.11: *Implementação do método* `copiaLista()`

O método `completaComPausas(ArrayList l)` completa o `ArrayList` passado com pausa até que seu último compasso esteja completo.

```
public static void completaComPausas(ArrayList l){

        double faltaPreencher = FuncoesAuxiliares.getValorPreencherCompasso(l);

        while (faltaPreencher != 0){
            l.add("r32");
            faltaPreencher -= 0.03125;
        }
}
```

Código 6.5.12: *Implementação do método* `completaComPausas()`

O método `retornaMaiorValor(double a, double b, double c)` retorna o maior valor dos três argumentos passados.

```
public static double retornaMaiorValor(double a, double b, double c){

        if(a > b){
            if(a > c){
                return a;
            }
            else{
                return c;
            }
        }
        else{
            if(b > c){
                return b;
            }
            else{
                return c;
            }
        }
    }
}
```

Código 6.5.13: *Implementação do método* `retornaMaiorValor()`

6.6 Implementação da Classe `PartesDaFuga`

Classe que contém as principais partes da fuga: sujeito, resposta, contra-sujeito 1, contra-sujeito 2 e as inversões dos contra-sujeito 1 e 2. Essas partes foram implementadas com `ArrayList`, onde cada posição do array corresponde a uma nota, ou melhor, à string que a representa.

```
private ArrayList sujeito = new ArrayList();
private ArrayList resposta = new ArrayList();
private ArrayList contra_sujeito_1 = new ArrayList();
private ArrayList func_inv_contra_sujeito_1 = new ArrayList();
private ArrayList contra_sujeito_2 = new ArrayList();
private ArrayList func_inv_contra_sujeito_2 = new ArrayList();
```

Código 6.6.1: *`ArrayList` com as Partes da Fuga*

O método `setSujeito(char[] buffer)` recebe os caracteres lidos do arquivo de entrada com a representação do sujeito e coloca cada nota em uma posição do `ArrayList` que o representa. A última nota do sujeito é modificada para que tenha como tempo o número 4, ou seja, para que seja uma semínima.

```
public void setSujeito(char[] buffer){

        char[] buffer_sujeito = new char[buffer.length];;
        StringTokenizer tokenSujeito;

        for(int i = 0; i < buffer.length; i++){
           buffer_sujeito[i] = buffer[i];
        }

        String bufferString = new String(buffer_sujeito);
        tokenSujeito = new StringTokenizer(bufferString," ");

        while(tokenSujeito.hasMoreTokens()){
            sujeito.add(tokenSujeito.nextToken());
        }

        FuncoesAuxiliares.modificaUltimaNota(sujeito);
        FuncoesAuxiliares.imprime(sujeito);

}
```

Código 6.6.2: *Implementação do método* `setSujeito()`

A método `setResposta(ArrayList l)` tem como parâmetro um `ArrayList`, que deve conter as notas do sujeito, que será todo transposto uma quinta justa acima. Retorna um `ArrayList` contendo essa transposição.

```
public void setResposta(ArrayList l){

        String t = new String();
        String normal = new String();
        int posicao = 0;
        for(int i = 0; i < sujeito.size(); i++){
                t = TransformacoesAlgebricas.transpoeToken((String) l.get(i),
                                                Constantes.QUINTA_JUSTA);

                posicao = FuncoesAuxiliares.getPosicaoNota((String) l.get(i));
                if((posicao + Constantes.QUINTA_JUSTA) < 12){
                    resposta.add(t);
                }
                else{
                    normal = TransformacoesAlgebricas.normalizaToken(t,
                                                Constantes.OITAVA_ACIMA);
                    resposta.add(normal);
                }
        }
}
```

Código 6.6.3: *Implementação do método* `setResposta()`

O método `setContraSujeito_1(ArrayList l)` tem como parâmetro um `ArrayList`, que deve conter as notas da resposta. Primeiramente o método coloca no ArrayList de retorno notas que completem o compasso do sujeito, já que este termina em um tempo forte e não fecha um compasso. Essas notas são obtidas a partir do próprio sujeito, em movimento retrógrado. Em seguida, são colocadas as notas do parâmetro de entrada, ou seja, da resposta, em movimento também retrógrado, até que a saída fique do tamanho do mesmo, sem sua

última nota. Este fato faz com que o contra-sujeito 1 seja menor que o sujeito em 2 semínimas.

```
public void setContraSujeito_1(ArrayList l){

        double faltaPreencher = FuncoesAuxiliares.getValorPreencherCompasso(l);
        int cont = 2;
        double tamanho_resposta = 0.0;
        double tamanho_contrasujeito_1 = 0.0;


        String s = new String();
        String normal = new String();



        while (faltaPreencher != 0){
            s = FuncoesAuxiliares.getLiteral((String) sujeito.get(sujeito.size()
                                                                  - cont)) +"4";
            contra_sujeito_1.add(s);
            cont++;
            faltaPreencher -= 0.25;
        }

        for(int i = 0; i < l.size(); i++){
            tamanho_resposta += FuncoesAuxiliares.getTempoNota((String) l.get(i));
        }

        for(int i = 0; i < contra_sujeito_1.size(); i++){
            tamanho_contrasujeito_1 += FuncoesAuxiliares.getTempoNota((String)
                                                    contra_sujeito_1.get(i));
        }

        cont = 2;

        while((tamanho_contrasujeito_1 < (tamanho_resposta - 0.5))){

                s = (String) l.get(l.size() - cont);
                normal = TransformacoesAlgebricas.normalizaToken(s,
                                                Constantes.OITAVA_ABAIXO);
                contra_sujeito_1.add(normal);
                cont++;
                tamanho_contrasujeito_1 += FuncoesAuxiliares.getTempoNota(s);

        }

}
```

Código 6.6.4: *Implementação do método* `setContraSujeito_1()`

O método `setContraSujeito_2(ArrayList l)` recebe o `ArrayList` com as notas do contra-sujeito 1 e as coloca no ArrayList de saída em movimento retrógrado. Como o contra-sujeito 1 é menor que o tamanho do sujeito, há ainda uma complementação do contra-sujeito 2 com notas do sujeito, em movimento retrógrado, até que aquele tenha o tamanho deste.

```
public void setContraSujeito_2(ArrayList l){

        int cont = 1;
        int posicao = 0;
        String s = new String();
        String t = new String();
        String normal = new String();
        double tamanho_sujeito = 0.0;
        double tamanho_contrasujeito_2 = 0.0;

        for(int i = 0; i < l.size(); i++){
          s = (String) l.get(l.size() - cont);
          t = TransformacoesAlgebricas.transpoeToken(s, Constantes.QUARTA_JUSTA);
          posicao = FuncoesAuxiliares.getPosicaoNota(s);
          if((posicao + Constantes.QUARTA_JUSTA) < 12){
                 contra_sujeito_2.add(t);
          }
          else{
                normal = TransformacoesAlgebricas.normalizaToken(t,
                                                        Constantes.OITAVA_ACIMA);
                contra_sujeito_2.add(normal);
            }
            cont++;
        }

        for(int i = 0; i < sujeito.size(); i++){
            tamanho_sujeito += FuncoesAuxiliares.getTempoNota((String)
                                                                sujeito.get(i));
        }

        for(int i = 0; i < contra_sujeito_2.size(); i++){
            tamanho_contrasujeito_2 += FuncoesAuxiliares.getTempoNota((String)
                                                         contra_sujeito_2.get(i));
        }

        cont = 1;

        while( tamanho_contrasujeito_2 < (tamanho_sujeito - 0.25)){

          s = FuncoesAuxiliares.getLiteral((String) sujeito.get(sujeito.size()
                                                                - cont)) +"4";
          contra_sujeito_2.add(s);
          tamanho_contrasujeito_2 += 0.25;
          cont++;
        }
}
```

Código 6.6.5: *Implementação do método `setContraSujeito_2()`*

Os métodos `setFuncInversaContraSujeito_1(ArrayList l)` e `setFuncInversaContraSujeito_2(ArrayList l)` recebem como entrada o contra-sujeito 1 e o contra-sujeito 2, respectivamente, e tentam reverter o processo de suas criações, retornando suas entradas originais, quais sejam, a resposta e o contra-sujeito 1, nesta ordem.

```
public void setFuncInversaContraSujeito_1(ArrayList l){

        int cont = 1;
        String s = new String();
        double faltaPreencher;


        while(cont <= l.size()){
            s = (String) l.get(l.size() - cont);
            func_inv_contra_sujeito_1.add(s);
            cont++;
        }

        cont = 1;

        faltaPreencher = FuncoesAuxiliares.
                         getValorPreencherCompasso(func_inv_contra_sujeito_1);

        while (faltaPreencher != 0){
           s = (FuncoesAuxiliares.getLiteral((String) resposta.get(resposta.size()
                                                                    - cont)) +"4");
           func_inv_contra_sujeito_1.add(s);
           cont++;
           faltaPreencher -= 0.25;
        }

}
```

Código 6.6.6: *Implementação do método* `setFuncInversaContraSujeito_1()`

```
public void setFuncInversaContraSujeito_2(ArrayList l){


        int posicao = 0;
        ArrayList aux = new ArrayList();
        ArrayList auxNorm = new ArrayList();
        String s = new String();
        String t = new String();
        String normal = new String();

        for(int cont = 1; cont <= l.size(); cont++){
            s = (String) l.get(l.size() - cont);
            t = TransformacoesAlgebricas.transpoeToken(s,
                                                Constantes.QUINTA_JUSTA);
            posicao = FuncoesAuxiliares.getPosicaoNota(s);
            if((posicao + Constantes.QUINTA_JUSTA) < 12){
                 aux.add(t);

            }
            else{
                normal = TransformacoesAlgebricas.normalizaToken(t,
                                                        Constantes.OITAVA_ACIMA);
                aux.add(normal);

            }

        }
        auxNorm = TransformacoesAlgebricas.retornaNormalizacao(aux,
                                                        Constantes.OITAVA_ABAIXO);

        for(int i = 0; i < auxNorm.size(); i++){
            func_inv_contra_sujeito_2.add((String) auxNorm.get(i));

        }
}
```

Código 6.6.7: *Implementação do método* `setFuncInversaContraSujeito_2()`

Por fim, o método `setPartesDaFuga()` inicializa todas os `ArrayList` desta classe, que tem como entrada o array de char com as notas do sujeito, do qual tudo é derivado.

```
public void setPartesDaFuga(){

        setSujeito(buffer_sujeito);
        setResposta(sujeito);
        setContraSujeito_1(resposta);
        setContraSujeito_2(contra_sujeito_1);
        setFuncInversaContraSujeito_1(contra_sujeito_1);
        setFuncInversaContraSujeito_2(contra_sujeito_2);
}
```

Código 6.6.8: *Implementação do método `setPartesDaFuga()`*

6.7 Implementação da Classe `TransformacoesAlgebricas`

Classe semelhante à classe `PartesdaFuga`, pois contém as transformações melódicas do sujeito da fuga: aumento, diminuição, retrógrado e inversão, além das pausas correspondentes às suas notas, armazenadas cada uma em um `ArrayList`.

```
private ArrayList pausas = new ArrayList();
private ArrayList aumento = new ArrayList();
private ArrayList diminuicao = new ArrayList();
private ArrayList inversao = new ArrayList();
private ArrayList retrogrado = new ArrayList();
```

Código 6.7.1: *`ArrayList` com as Transformações Melódicas*

O método `normalizaToken(String s, int posicao)` recebe como parâmetros o `token` a ser normalizado e a posição para normalização e funciona fazendo com que a `string` de retorno esteja uma oitava abaixo ou acima da inicial, dependendo do valor da posição. O método trabalha com as vírgulas e apóstrofos que porventura estejam na `string`. Se a posição é para subir uma oitava, o método acrescenta um apóstrofo ou diminui uma vírgula, da string original; se é para descer uma oitava, o método diminui um apóstrofo ou acrescenta uma vírgula ao token de retorno. Às pausas, que independem de oitavas, não são feitas quaisquer alterações. Para aplicar essa operação a todas as notas de um ArrayList, é utilizada o método `retornaNormalizacao(ArrayList l, int posicao)`, que retornará um `ArrayList` com as modificações feitas.

```
public static String normalizaToken(String s, int posicao){

    String retorno = new String();
    String literal = new String();
    String numeral = new String();
    String apostrofo = new String();
    String virgula = new String();

    int cont_apostrofo = 0;
    int cont_virgula = 0;

    if(s.charAt(0) == 'r'){
        return s;
    }
    for(int k = 0; k < s.length(); k++){
        if(s.charAt(k) == '0' ||
           s.charAt(k) == '1' ||
           s.charAt(k) == '2' ||
           s.charAt(k) == '3' ||
           s.charAt(k) == '4' ||
           s.charAt(k) == '5' ||
           s.charAt(k) == '6' ||
           s.charAt(k) == '7' ||
           s.charAt(k) == '8' ||
           s.charAt(k) == '9' ||
           s.charAt(k) == '~' ||
           s.charAt(k) == '.' ){

            numeral += s.charAt(k);
        }
        else
        if (s.charAt(k) == '\''){
            apostrofo += s.charAt(k);
            cont_apostrofo++;
        }
        else
        if(s.charAt(k) == ',' ){
            virgula += s.charAt(k);
            cont_virgula++;
        }
        else literal += s.charAt(k);
    }
```

Código 6.7.2: *Implementação do método* `normalizaToken()`

```
        switch(posicao){
            case(Constantes.OITAVA_ABAIXO):{
                retorno += literal;
                if((cont_apostrofo == 0) && (cont_virgula == 0)){
                    retorno += ",";
                }
                else if(cont_apostrofo != 0){
                        for(int i = 0; i < cont_apostrofo - 1; i++){
                            retorno += "\'";
                        }
                }
                else if(cont_virgula != 0){
                        for(int i = 0; i < cont_virgula + 1; i++){
                            retorno += ",";
                        }
                }
                retorno += numeral;
                break;

            }
            case(Constantes.OITAVA_ACIMA):{
                retorno += literal;
                if((cont_apostrofo == 0) && (cont_virgula == 0)){
                    retorno += "\'";
                }
                else if(cont_apostrofo != 0){
                        for(int i = 0; i < cont_apostrofo + 1; i++){
                            retorno += "\'";
                        }
                }
                else if(cont_virgula != 0){
                        for(int i = 0; i < cont_virgula - 1; i++){
                            retorno += ",";
                        }
                }
                retorno += numeral;
                break;
            }
        }
        return retorno;
    }
```

Código 6.7.3: *Continuação da implementação do método* `normalizaToken()`

O método `transpoeToken(String s, int t)` transpõem a nota passada como parâmetro pela posição t. Para isso, depois de localizado em qual escala a nota está, o valor do índice da posição dessa nota nessa escala é somado ao valor de t e é feita a operação em aritmética modular para se achar o índice da posição da nota transposta.

```
public static String transpoeToken(String s, int t){

        String retorno = new String();
        String literal = new String();
        String numeral = new String();

        for(int k = 0; k < s.length(); k++){
            if(s.charAt(k) == '0' ||
               s.charAt(k) == '1' ||
               s.charAt(k) == '2' ||
               s.charAt(k) == '3' ||
               s.charAt(k) == '4' ||
               s.charAt(k) == '5' ||
               s.charAt(k) == '6' ||
               s.charAt(k) == '7' ||
               s.charAt(k) == '8' ||
               s.charAt(k) == '9' ||
               s.charAt(k) == '\''||
               s.charAt(k) == ',' ||
               s.charAt(k) == '~' ||
               s.charAt(k) == '.' ){

                numeral += s.charAt(k);
            }
            else literal += s.charAt(k);
        }
        for(int i = 0; i < 12; i++){

            if(literal.equals(Constantes.escala_bemol[i])){
                retorno = Constantes.escala_bemol[(i + t) % 12];
                if(numeral.length() > 0){
                  retorno += numeral;
                }
                return   retorno;
            }

            if(literal.equals(Constantes.escala_sustenido[i])){
                retorno = Constantes.escala_sustenido[(i + t) % 12];
                if(numeral.length() > 0){
                  retorno += numeral;
                }

                return   retorno;
            }
        }
        return s;
}
```

Código 6.7.4: *Implementação do método* `transpoeToken()`

O método `pausificaToken(String s)` transforma a string de entrada em sua pausa correspondente, ou seja, retorna a pausa que tem o mesmo tamanho, em relação ao tempo, da string. Para que essa transformação seja atribuída à variável pausas da classe, é chamada o método `setPausas(ArrayList l)`, que tem como entrada o ArrayList com as notas do sujeito.

```
public String pausificaToken(String s){

        String retorno = new String();
        String numeral = new String();

        for(int k = 0; k < s.length(); k++){
               if(s.charAt(k) == '0' ||
                  s.charAt(k) == '1' ||
                  s.charAt(k) == '2' ||
                  s.charAt(k) == '3' ||
                  s.charAt(k) == '4' ||
                  s.charAt(k) == '5' ||
                  s.charAt(k) == '6' ||
                  s.charAt(k) == '7' ||
                  s.charAt(k) == '8' ||
                  s.charAt(k) == '9' ||
                  s.charAt(k) == '~' ||
                  s.charAt(k) == '.' ){
                     numeral += s.charAt(k);
                  }
           }
           retorno = "r";
           if(numeral.length() > 0){
               retorno += numeral;
           }
           return retorno;
}
```

Código 6.7.5: *Implementação do método* `pausificaToken()`

O método `aumentaToken(String s)` tem como retorno uma `string` com a duplicação do tempo da string de entrada. Para aplicar essa transformação a um `ArrayList`, o método `setAumento(ArrayList l)` é chamada, e o valor é atribuído à variável aumento da classe.

```
public static String aumentaToken(String s){

        String retorno = new String();
        String literal = new String();
        String numeral = new String();
        String simbolo = new String();

        int numero = 0;

        for(int k = 0; k < s.length(); k++){
            if(s.charAt(k) == '0' ||
               s.charAt(k) == '1' ||
               s.charAt(k) == '2' ||
               s.charAt(k) == '3' ||
               s.charAt(k) == '4' ||
               s.charAt(k) == '5' ||
               s.charAt(k) == '6' ||
               s.charAt(k) == '7' ||
               s.charAt(k) == '8' ||
               s.charAt(k) == '9' ){

               numeral += s.charAt(k);
            }
            else if (s.charAt(k) == '.' || s.charAt(k) == '~'){
                     simbolo += s.charAt(k);
                }
                else{
                       literal += s.charAt(k);
                }
        }

        if(numeral.length() > 0){

            numero = Integer.parseInt(numeral);
            numero = (numero / 2);
            retorno += literal;
            retorno += numero;
            retorno += simbolo;
            return retorno;
        }
        else{
            return s;
        }
}
```

Código 6.7.6: *Implementação do método `aumentaToken()`*

O método `diminuiToken(String s)` tem como retorno uma string com a metade do tempo da string de entrada. Para aplicar essa transformação a um `ArrayList`, o método `setDiminuicao(ArrayList l)` é chamada, e o valor é atribuído à variável diminuicao da classe.

```
public static String diminuiToken(String s){

        String retorno = new String();
        String literal = new String();
        String numeral = new String();
        String simbolo = new String();

        int numero = 0;

        for(int k = 0; k < s.length(); k++){
            if(s.charAt(k) == '0' ||
               s.charAt(k) == '1' ||
               s.charAt(k) == '2' ||
               s.charAt(k) == '3' ||
               s.charAt(k) == '4' ||
               s.charAt(k) == '5' ||
               s.charAt(k) == '6' ||
               s.charAt(k) == '7' ||
               s.charAt(k) == '8' ||
               s.charAt(k) == '9' ){

               numeral += s.charAt(k);
            }
            else if (s.charAt(k) == '.' || s.charAt(k) == '~'){
                     simbolo += s.charAt(k);
                }
                else{
                       literal += s.charAt(k);
                }
        }

        if(numeral.length() > 0){

            numero = Integer.parseInt(numeral);
            numero = (numero * 2);
            retorno += literal;
            retorno += numero;
            retorno += simbolo;
            return retorno;
        }
        else{
            return s;
        }
}
```

Código 6.7.7: *Implementação do método `diminuiToken()`*

O método `inverteToken(String s, int t)` tem como retorno a `string` que está na posição t na escala das notas, da classe Constantes, sendo que t é o índice da nota invertida, calculada previamente. Assim, é aparentemente desnecessária a passagem da string s, mas isso é feito para verificar em qual oitava a nota original está e fazer a necessária normalização, se preciso, e também retorna a nova nota com os possíveis símbolos que a nota original apresente.

```
public static String inverteToken(String s, int t){

        String retorno = new String();
        String literal = new String();
        String numeral = new String();
        int apostrofo = 0;
        int virgula = 0;

        for(int k = 0; k < s.length(); k++){
            if(s.charAt(k) == '0' ||
               s.charAt(k) == '1' ||
               s.charAt(k) == '2' ||
               s.charAt(k) == '3' ||
               s.charAt(k) == '4' ||
               s.charAt(k) == '5' ||
               s.charAt(k) == '6' ||
               s.charAt(k) == '7' ||
               s.charAt(k) == '8' ||
               s.charAt(k) == '9' ||
               s.charAt(k) == '~' ||
               s.charAt(k) == '.' ){

                numeral += s.charAt(k);
            }
            else if(s.charAt(k) == ','){
                virgula++;
            }
            else if(s.charAt(k) == '\''){
                apostrofo++;
            }
            else literal += s.charAt(k);
        }
        for(int i = 0; i < 12; i++){

            if(literal.equals(Constantes.escala_bemol[i])){

                retorno = Constantes.escala_bemol[t];
                if(numeral.length() > 0){
                  retorno += numeral;
                }
            }
```

Código 6.7.8: *Implementação do método* `inverteToken()`

```
            if(literal.equals(Constantes.escala_sustenido[i])){

                retorno = Constantes.escala_sustenido[t];
                if(numeral.length() > 0){
                  retorno += numeral;
                }
            }
        }
        return   retorno;
    }
```

Código 6.7.9: *Continuação da implementação do método* `inverteToken()`

O método `setInversao(ArrayList l)` calcula, a partir da primeira nota, que será a base para a inversão, as inversões de todas as restantes do ArrayList passado como parâmetro e adiciona esses valores à variável inversão da classe.

```
public void setInversao(ArrayList l){

        String notaInvertida = new String();
        int posicaoPrimeiraNota =FuncoesAuxiliares.getPosicaoNota((String)
                                                                    l.get(0));
        int posicaoNota = 0;
        int posicaoInvertida = 0;
        // a primeira nota da inversao eh igual a primeira nota do sujeito
        inversao.add(l.get(0));

        //para calcular a inversao, fixamos a primeira nota e, a partir dela,
        //fazemos as inversoes dos intervalos

        for(int i = 1; i < l.size(); i++){

            posicaoNota  = FuncoesAuxiliares.getPosicaoNota((String) l.get(i));
            posicaoInvertida = FuncoesAuxiliares.
                               calculaInversao(posicaoPrimeiraNota,posicaoNota);
            notaInvertida = inverteToken((String) l.get(i), posicaoInvertida);
            inversao.add(notaInvertida);
        }

}
```

Código 6.7.10: *Implementação do método* `setInversao()`

O método `setRetrogrado(ArrayList l)` inverte as posições do `ArrayList` passado como parâmetro e atribui esses valores à variável inversão da classe.

```
public void setRetrogrado(ArrayList l){

         // i tem que comecar com o valor 2 para
        // nao pegar o valor da ultima nota

        for(int i = 2; i <= l.size(); i++){

           retrogrado.add(l.get(l.size() - i));
        }
}
```

Código 6.7.11: *Implementação do método `setRetrogrado()`*

O método `setTransformacoesAlgebricas(ArrayList l)` inicializa todos os ArryList da classe, que tem como entrada o ArrayList contendo as notas do sujeito, ou seja, as transformações melódicas derivam diretamente do sujeito da fuga.

```
public void setTransformacoesAlgebricas(ArrayList l){

        setPausas(l);
        setAumento(l);
        setDiminuicao(l);
        setInversao(l);
        setRetrogrado(l);

}
```

Código 6.7.12: *Implementação do método* `setTransformacoesAlgebricas()`

6.8 Implementação da Classe `GeraFuga`

É a classe principal do programa gerador de fugas, onde as partes da fuga e as transformações melódicas serão concatenadas, de forma a gerar a fuga.

Os `ArrayList voz_1, voz_2 e voz_3` armazenam as notas obtidas na construção das partes da fuga e das transformações melódicas, na ordem em que devem aparecer.

As pilhas `pilha_1, pilha_2 e pilha_3` armazenam as transformações inversas das partes da fuga na medida em que estas vão sendo geradas, para posterior desempilhamento na ordem inversa em que foram empilhadas.

Os contadores de tempo `contadorTempo_1, contadorTempo_2 e contadorTempo_3` são variáveis numéricas que vão sendo incrementadas dos tempos das notas na medida em que estas são adicionadas nos ArrayList das vozes correspondentes, para o cálculo do tamanho dessas vozes enquanto estão sendo geradas e/ou para uso posterior do valor desse tamanho.

As `strings textoVoz_1, textoVoz_2 e textoVoz_3` recebem as strings que foram armazenadas nos `ArrayList voz_1, voz_2 e voz_3`, respectivamente, para serem acrescentadas no arquivo texto que representará a fuga no formato `LyliPond`.

```
1  private ArrayList voz_1 = new ArrayList();
2  private ArrayList voz_2 = new ArrayList();
3  private ArrayList voz_3 = new ArrayList();
4
5  private Stack pilha_1 = new Stack();
6  private Stack pilha_2 = new Stack();
7  private Stack pilha_3 = new Stack();
8
9  private double contadorTempo_1 = 0.0;
10 private double contadorTempo_2 = 0.0;
11 private double contadorTempo_3 = 0.0;
12
13 private String textoVoz_1 = new String();
14 private String textoVoz_2 = new String();
15 private String textoVoz_3 = new String();
```

Código 6.8.1: *Variáveis da Classe* `GeraFuga`

Os métodos `adicionarVoz1(ArrayList l), adicionarVoz2(ArrayList l) e adicionarVoz3(ArrayList l)` adicionam o conteúdo das posições do `ArrayList` passado ao `ArrayList` da voz correspondente e incrementa o contador dessas vozes com os tempos das notas passadas.

```
1  public void adicionarVoz1(ArrayList l){
2
3          for(int i = 0; i < l.size(); i++){
4              voz_1.add(l.get(i));
5              contadorTempo_1 += FuncoesAuxiliares.getTempoNota((String) l.get(i));
6          }
7  }
8
9  public void adicionarVoz2(ArrayList l){
10
11         for(int i = 0; i < l.size(); i++){
12             voz_2.add(l.get(i));
13             contadorTempo_2 += FuncoesAuxiliares.getTempoNota((String) l.get(i));
14         }
15 }
16
17 public void adicionarVoz3(ArrayList l){
18
19         for(int i = 0; i < l.size(); i++){
20             voz_3.add(l.get(i));
21             contadorTempo_3 += FuncoesAuxiliares.getTempoNota((String) l.get(i));
22         }
23 }
```

Código 6.8.2: *Implementação dos métodos* `adicionarVoz`

O método `setFugaPrimeiraParte()` adiciona às vozes correspondentes as partes da fuga e suas transformações melódicas, na ordem correta de aparecimento, e, concomitantemente, empilha os valores obtidos da aplicação da função inversa a essas partes, se são derivações de outras partes da fuga. Como todas as partes que compõem a fuga derivam direta ou indiretamente do sujeito, somente este não possuirá função inversa.

```
public void setFugaPrimeiraParte(){

        ArrayList aux = new ArrayList();
        ArrayList auxNorm = new ArrayList();
        String s = new String();
        String t = new String();
        String normal = new String();
        int posicao = 0;

//************************** Voz_2 ***************************
        // Segunda voz, onde o tema aparece primeiro
        ArrayList aux2Pilha1;
        ArrayList aux2Pilha2;
        ArrayList aux2Pilha3;

        adicionarVoz2(partesFuga.getSujeito());
/* 1 */ adicionarVoz2(partesFuga.getContraSujeito_1());
/* 2 */ adicionarVoz2(partesFuga.getContraSujeito_2());
/* 3 */ adicionarVoz2(transformacoes.getRetrogrado());

        // empilhamento das funcoes inversas

/* 1 */ aux2Pilha1 = FuncoesAuxiliares.copiaLista(partesFuga.
                                getFuncInversaContraSujeito_1());
        pilha_2.push(aux2Pilha1);

/* 2 */ aux2Pilha2 = FuncoesAuxiliares.copiaLista(partesFuga.
                                getFuncInversaContraSujeito_2());
        pilha_2.push(aux2Pilha2);

/* 3 */ aux.clear();
        aux = FuncoesAuxiliares.copiaLista(partesFuga.getSujeito());
        FuncoesAuxiliares.retiraUltimaNota(aux);
        aux2Pilha3 = FuncoesAuxiliares.copiaLista(aux);
        pilha_2.push(aux2Pilha3);

/************************** Voz_1 **************************/
        ArrayList aux1Pilha1;
        ArrayList aux1Pilha2;
        ArrayList aux1Pilha3;

        auxNorm.clear();
        auxNorm = TransformacoesAlgebricas.retornaNormalizacao(aux,
                                                Constantes.OITAVA_ACIMA);
        aux1Pilha2 = FuncoesAuxiliares.copiaLista(auxNorm);
        pilha_1.push(aux1Pilha2);
```

Código 6.8.3: *Implementação do método* `setFugaPrimeiraParte()`

```
/* 3 */ auxNorm.clear();
        auxNorm = TransformacoesAlgebricas.
                        retornaNormalizacao(partesFuga.getSujeito(),
                                        Constantes.OITAVA_ACIMA);
        FuncoesAuxiliares.retiraUltimaNota(auxNorm);
        aux1Pilha3 = FuncoesAuxiliares.copiaLista(auxNorm);
        pilha_1.push(aux1Pilha3);


//*************************** Voz_3 ***************************/
        ArrayList aux3Pilha1;

        adicionarVoz3(transformacoes.getPausas());
        adicionarVoz3(transformacoes.getPausas());

        auxNorm.clear();
        auxNorm = TransformacoesAlgebricas.
                        retornaNormalizacao(partesFuga.getSujeito(),
                                        Constantes.OITAVA_ABAIXO);
        FuncoesAuxiliares.retiraUltimaNota(auxNorm);
        adicionarVoz3(auxNorm);

        //a ordem esta invertida para nao ser preciso fazer outra normalizacao
/* 1 */ aux3Pilha1 = FuncoesAuxiliares.copiaLista(auxNorm);
        pilha_3.push(aux3Pilha1);

/* 1 */ auxNorm.clear();
        auxNorm = TransformacoesAlgebricas.
                  retornaNormalizacao(transformacoes.getAumento(),
                                  Constantes.OITAVA_ABAIXO);
        adicionarVoz3(auxNorm);


}
        adicionarVoz1(transformacoes.getPausas());
/* 1 */ adicionarVoz1(partesFuga.getResposta());
```

Código 6.8.4: *Continuação da implementação do método* `setFugaPrimeiraParte()`

```
        aux.clear();
/* 2 */ for(int i = 0; i < partesFuga.getContraSujeito_1().size(); i++){
                t = TransformacoesAlgebricas.
                   transpoeToken((String) partesFuga.getContraSujeito_1().get(i),
                                                 Constantes.QUARTA_JUSTA);
                posicao = FuncoesAuxiliares.getPosicaoNota((String)
                                          partesFuga.getContraSujeito_1().get(i));
                if((posicao + Constantes.QUARTA_JUSTA) < 12){
                    aux.add(t);
                }
                else{
                    normal = TransformacoesAlgebricas.normalizaToken(t,
                                                          Constantes.OITAVA_ACIMA);
                    aux.add(normal);
                }

        }
        auxNorm.clear();
        auxNorm = TransformacoesAlgebricas.retornaNormalizacao(aux,
                                                          Constantes.OITAVA_ACIMA);
        adicionarVoz1(auxNorm);

/* 3 */ auxNorm.clear();
        auxNorm = TransformacoesAlgebricas.
                                retornaNormalizacao(transformacoes.getDiminuicao(),
                                                    Constantes.OITAVA_ACIMA);
        adicionarVoz1(auxNorm);

/* 1 */ auxNorm.clear();
        auxNorm = TransformacoesAlgebricas.
                  retornaNormalizacao(partesFuga.getSujeito(),
                                      Constantes.OITAVA_ACIMA);
        FuncoesAuxiliares.retiraUltimaNota(auxNorm);
        aux1Pilha1 = FuncoesAuxiliares.copiaLista(auxNorm);
        pilha_1.push(aux1Pilha1);
```

Código 6.8.5: *Continuação da Implementação do método* `setFugaPrimeiraParte()`

```
2       aux.clear();
        for(int i = 0; i  partesFuga.getFuncInversaContraSujeito_1().size(); i++){
           t = TransformacoesAlgebricas.transpoeToken((String)
                                partesFuga.getFuncInversaContraSujeito_1().get(i),
                                Constantes.QUINTA_JUSTA);
        posicao = FuncoesAuxiliares.
                  getPosicaoNota((String)
                                partesFuga.getFuncInversaContraSujeito_1().get(i));
        if((posicao + Constantes.QUINTA_JUSTA)  12){
                    aux.add(t);
        }
        else{
                normal = TransformacoesAlgebricas.normalizaToken(t,
                                                    Constantes.OITAVA_ACIMA);
                aux.add(normal);
        }
        }
```

Código 6.8.6: *Continuação da Implementação do método* `setFugaPrimeiraParte()`

O método `setFugaSegundaParte()` apenas desempilha as inversas das transformações e as adiciona às vozes correspondentes.

```
1  public void setFugaSegundaParte(){
2
3
4            while(!pilha_2.empty()){
5                adicionarVoz2((ArrayList) pilha_2.pop());
6            }
7
8            while(!pilha_1.empty()){
9                adicionarVoz1((ArrayList) pilha_1.pop());
10           }
11
12           while(!pilha_3.empty()){
13               adicionarVoz3((ArrayList) pilha_3.pop());
14           }
15
16 }
```

Código 6.8.7: *Implementação do método `setFugaSegundaParte()`*

O método `completarFugaComPausas()` adiciona pausas às vozes menores até que estejam do mesmo tamanho da maior das três, fechando, assim, de maneira correta a fuga.

```
1  public void completarFugaComPausas(){
2
3          double faltaPreencherCompasso;
4          double faltaPreencherMusica;
5          double maiorValor = 0.0;
6          int cont = 2;
7
8          // se falta preencher o compasso entra no while
9
10         faltaPreencherCompasso = FuncoesAuxiliares.
11                                  getValorPreencherCompasso(voz_1);
12         while (faltaPreencherCompasso > 0){
13
14             voz_1.add("r32");
15             contadorTempo_1 += 0.03125;
16             faltaPreencherCompasso -= 0.03125;
17
18         }
19
20         faltaPreencherCompasso = FuncoesAuxiliares.
21                                  getValorPreencherCompasso(voz_2);
22
23         while (faltaPreencherCompasso > 0){
24             voz_2.add("r32");
25             contadorTempo_2 += 0.03125;
26             faltaPreencherCompasso -= 0.03125;
27
28         }
29
30         faltaPreencherCompasso = FuncoesAuxiliares.
31                                  getValorPreencherCompasso(voz_3);
32
33         while (faltaPreencherCompasso > 0){
34             voz_3.add("r32");
35             contadorTempo_3 += 0.03125;
36             faltaPreencherCompasso -= 0.03125;
37
38         }
39
40         maiorValor = FuncoesAuxiliares.retornaMaiorValor(contadorTempo_1,
41                                                          contadorTempo_2,
42                                                          contadorTempo_3);
43
```

Código 6.8.8: *Implementação do método* `completarFugaComPausas()`

```
1           if(maiorValor == contadorTempo_1){
2               faltaPreencherMusica = contadorTempo_1 - contadorTempo_2;
3               while (faltaPreencherMusica > 0){
4
5                   voz_2.add("r1");
6                   contadorTempo_2 += 1.0;
7                   faltaPreencherMusica -= 1.0;
8
9               }
10
11              faltaPreencherMusica = contadorTempo_1 - contadorTempo_3;
12
13              while (faltaPreencherMusica > 0){
14
15                     voz_3.add("r1");
16                     contadorTempo_3 += 1.0;
17                     faltaPreencherMusica -= 1.0;
18
19              }
20          }
21          else
22          if(maiorValor == contadorTempo_2){
23              faltaPreencherMusica = contadorTempo_2 - contadorTempo_1;
24              while (faltaPreencherMusica > 0){
25                  voz_1.add("r1");
26                  contadorTempo_1 += 1.0;
27                  faltaPreencherMusica -= 1.0;
28
29              }
30
31              faltaPreencherMusica = contadorTempo_2 - contadorTempo_3;
32              while (faltaPreencherMusica > 0){
33                   voz_3.add("r1");
34                   contadorTempo_3 += 1.0;
35                   faltaPreencherMusica -= 1.0;
36
37              }
38          }
```

Código 6.8.9: *Continuação da implementação do método* `completarFugaComPausas()`

```
1           else
2           if(maiorValor == contadorTempo_3){
3               faltaPreencherMusica = contadorTempo_3 - contadorTempo_1;
4               while (faltaPreencherMusica > 0){
5                   voz_1.add("r1");
6                   contadorTempo_1 += 1.0;
7                   faltaPreencherMusica -= 1.0;
8
9               }
10
11              faltaPreencherMusica = contadorTempo_3 - contadorTempo_2;
12              while (faltaPreencherMusica > 0){
13                   voz_2.add("r1");
14                   contadorTempo_2 += 1.0;
15                   faltaPreencherMusica -= 1.0;
16
17              }
18          }
19
20  }
```

Código 6.8.10: *Continuação da implementação do método* `completarFugaComPausas()`

O método `setTextoVozes()` transfere as strings que estão nas posições dos `ArrayList` das vozes para as variáveis de texto que as representam e os métodos `getVoz_1(), getVoz_2() e getVoz_3()` retornam esses textos armazenados.

```
1  public void setTextoVozes(){
2
3          for(int i = 0; i < voz_2.size(); i++){
4
5              textoVoz_2 += (voz_2.get(i) +" ");
6
7          }
8
9          for(int i = 0; i < voz_1.size(); i++){
10
11             textoVoz_1 += (voz_1.get(i) +" ");
12
13         }
14
15         for(int i = 0; i < voz_3.size(); i++){
16
17             textoVoz_3 += (voz_3.get(i) +" ");
18
19         }
20 }
21
22 public String getVoz_1(){
23
24         return  textoVoz_1;
25
26 }
27
28 public String getVoz_2(){
29
30         return  textoVoz_2;
31
32 }
33
34 public String getVoz_3(){
35
36         return  textoVoz_3;
37
38 }
```

Código 6.8.11: *Implementação do método* `setTextoVozes()`

6.9 Implementação da Classe `Janela`

Esta classe fornece a interface gráfica para facilitar a execução do programa gerador de fugas e adicionar algumas funcionalidades que tornam a música gerada mais palatável ao ouvinte, como a escolha de instrumentos para cada voz e o tempo da execução sônica.

Os instrumentos podem ser aplicados a cada voz separadamente antes da geração do arquivo que será compilado, se isso não for feito, o instrumento default será o piano.

O tempo também deve ser escolhido antes da geração do arquivo, e está em batidas por minuto.

O botão Gerar fuga permite a escolha do arquivo texto que contém o sujeito, a partir do qual a fuga será gerada, e gera o arquivo no formato `LilyPond`, para posterior compilação.

```
1  private void gerarFugaActionPerformed(java.awt.event.ActionEvent evt) {
2
3
4          String nome_arq = new String();
5          String nome_lily = new String();
6          String temp = new String("");
7          Arquivo arq = new Arquivo();
8          Arquivo_Lily arq_lily = new Arquivo_Lily();
9          GeraFuga fuga;
10         int opcaoInstrumento_1 = 0;
11         int opcaoInstrumento_2 = 0;
12         int opcaoInstrumento_3 = 0;
13
14
15         if (fc == null) {
16             fc = new JFileChooser();
17         }
18
19
20         if(tempo.getText() != null){
21             temp = tempo.getText();
22         }
23
24         opcaoInstrumento_1 = instrumentoVoz_1.getSelectedIndex();
25         opcaoInstrumento_2 = instrumentoVoz_2.getSelectedIndex();
26         opcaoInstrumento_3 = instrumentoVoz_3.getSelectedIndex();
27
28         int valorRetorno = fc.showDialog(Janela.this,"Gerar Fuga");
29
30         if(valorRetorno == JFileChooser.APPROVE_OPTION){
31
32             File file = fc.getSelectedFile();
33
34             if( getExtensao(file).equals("txt")){
35                 nome_arq = file.getName();
36                 AreaTexto.append(nome_arq+"\n");
37
38                 arq.abrirArquivo(nome_arq);
39                 arq.imprimeBuffer();
40
41                 fuga = new GeraFuga(arq);
42
43                 arq_lily.adicionarTexto("soprano = {");
44                 if(opcaoInstrumento_1 != 0){
45                     arq_lily.adicionarTexto(retornaInstrumento(opcaoInstrumento_1));
```

Código 6.9.1: *Implementação do método* `gerarFuga()`

```
                }
                arq_lily.adicionarTexto(fuga.getVoz_1());
                arq_lily.adicionarTexto("}");

                arq_lily.adicionarTexto("alto =  {\n");
                if(opcaoInstrumento_2 != 0){
                    arq_lily.adicionarTexto(retornaInstrumento(opcaoInstrumento_2));
                }
                arq_lily.adicionarTexto(fuga.getVoz_2());
                arq_lily.adicionarTexto("}");

                arq_lily.adicionarTexto("tenor =  {");
                if(opcaoInstrumento_3 != 0){
                    arq_lily.adicionarTexto(retornaInstrumento(opcaoInstrumento_3));
                }
                arq_lily.adicionarTexto(fuga.getVoz_3());

                arq_lily.adicionarTexto("}");
                arq_lily.adicionarTexto(arq_lily.fim);

                arq_lily.adicionarTexto(arq_lily.textoMidiAbre);

                if(!temp.equals("")){
                   arq_lily.adicionarTexto(arq_lily.tempoMidiAbre);
                   arq_lily.adicionarTexto(temp); // colocar aqui o tempo
                   arq_lily.adicionarTexto(arq_lily.tempoMidiFecha);
                }

                arq_lily.adicionarTexto(arq_lily.textoMidiFecha);

                arq_lily.imprimeTexto();

                nome_lily = nome_arq.replaceAll(".txt","");
                arq_lily.gravarArquivoTexto(nome_lily+".ly");

                AreaTexto.append(arq_lily.getTexto());

            }
            else{
                 AreaTexto.append("\n Formato de arquivo invalido \n");
            }
        }
        else {
            AreaTexto.append("\n Erro na abertura do arquivo \n");
        }

        fc.setSelectedFile(null);


}
```

Código 6.9.2: *Continuação da implementação do método* `gerarFuga()`

O botão Compila fuga faz uma chamada externa ao interpretador `LilyPond`, que compilará o arquivo gerado pelo programa gerador de fugas e gerará o arquivo .pdf com a partitura que representa a fuga e o arquivo .mid, para tocá-la.

```
1       private void compilarFugaActionPerformed(java.awt.event.ActionEvent evt) {
2
3           String nome_arq = new String();
4           Runtime rt = Runtime.getRuntime();
5           Process p;
6
7           if (fc == null) {
8               fc = new JFileChooser();
9           }
10
11          int valorRetorno = fc.showDialog(Janela.this,"Compilar Fuga");
12
13
14          if(valorRetorno == JFileChooser.APPROVE_OPTION){
15
16              File file = fc.getSelectedFile();
17              if( getExtensao(file).equals("ly")){
18
19                  nome_arq = file.getName();
20                  try {
21                          p = rt.exec("lilypond "+nome_arq);
22                          try {
23                                  if( (p.waitFor()) == 0){
24                                      AreaTexto.append("\n Compilacao do
25                                                arquivo com sucesso \n");
26                                  }
27                          } catch (InterruptedException ex) {
28                                  ex.printStackTrace();
29                          }
30
31
32                  } catch (IOException ex) {
33                          ex.printStackTrace();
34                  }
35              }
36              else{
37                   AreaTexto.append("\n Formato de arquivo invalido \n");
38              }
39          }
40          else{
41               AreaTexto.append("\n Erro na abertura do arquivo \n");
42          }
43
44          fc.setSelectedFile(null);
45
46
47
48      }
```

Código 6.9.3: *Implementação do método* `compilarFuga()`

Os botões Tocar fuga e Parar execução são meras adições de funcionalidade, para não ser preciso outro programa externo para executar e parar a execução do arquivo .mid gerado.

```
1  private void tocarFugaActionPerformed(java.awt.event.ActionEvent evt) {
2
3          String nome_arq = new String();
4
5          if (fc == null) {
6              fc = new JFileChooser();
7          }
8
9          int valorRetorno = fc.showDialog(Janela.this,"Tocar Fuga");
10
11
12         if(valorRetorno == JFileChooser.APPROVE_OPTION){
13
14             File file = fc.getSelectedFile();
15             if( getExtensao(file).equals("midi") || getExtensao(file).equals("mid")){
16                  nome_arq = file.getName();
17                  AreaTexto.append(nome_arq+"\n");
18                  tocador = new TocadorMidi(nome_arq);
19                  tocador.toca();
20             }
21             else{
22                  AreaTexto.append("\n Formato de arquivo invalido \n");
23             }
24
25         }
26         else {
27             AreaTexto.append("\n Erro na abertura do arquivo \n");
28         }
29
30         fc.setSelectedFile(null);
31
32 }
```

Código 6.9.4: *Implementação do método* `tocarFuga()`

6.9.1 Interface Gráfica

Figura 6.1: *Interface Gráfica*

6.10 Implementação da Classe `ProgramaFinal`

Classe principal, que inicializa a execução do programa.

```
1  package fuguegenerator;
2
3  import javax.swing.*;
4
5  public class ProgramaFinal extends JPanel {
6
7
8          private static void createAndShowGUI() {
9
10             JFrame frame = new JFrame(" Gerador de Fugas ");
11             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12             frame.add(new Janela());
13             frame.pack();
14             frame.setVisible(true);
15         }
16
17     public static void main(String[] args) {
18         //Schedule a job for the event dispatch thread:
19         //creating and showing this application's GUI.
20         SwingUtilities.invokeLater(new Runnable() {
21             public void run() {
22                 //Turn off metal's use of bold fonts
23                 UIManager.put("swing.boldMetal", Boolean.FALSE);
24                 createAndShowGUI();
25             }
26         });
27     }
28
29 }
```

Código 6.10.1: *Implementação da Classe* `ProgramaFinal`

Capítulo 7

Resultados Obtidos

Como exemplo para o funcionamento do programa, apresentamos quatro sujeitos, ou entradas, as saídas correspondentes e a partitura que as representa.

7.1 Exemplo 1: O Cravo Bem Temperado, Fuga I em Dó Maior (BWV 846)

7.1.1 Sujeito (entrada do programa)

A notação para este sujeito é:

r8 c'8 d'8 e'8 f'8. g'32 f'32 e'8 a'8 d'8 g'8~ g'16 a'16 g'16 f'16 e'16

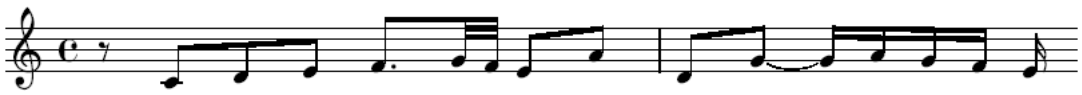

Figura 7.1: *Sujeito da Fuga*

7.1.2 Fuga (saída do programa)

```
soprano = {
r8 r8 r8 r8 r8. r32 r32 r8 r8 r8 r8~ r16 r16 r16
r16 r8 g’8 a’8 b’8 c’’8. d’’32 c’’32 b’8 e’’8 a’8
d’’8~ d’’16 e’’16 d’’16 c’’16 b’4 bes’’4 f’’16 g’’16
a’’16 g’’16 g’’8~ d’’8 a’’8 e’’8 f’’32 g’’32 f’’8. r16
c’’16 d’’16 e’’16 f’’16. g’’64 f’’64 e’’16 a’’16 d’’16
g’’16~ g’’32 a’’32 g’’32 f’’32 r8 c’’8 d’’8 e’’8 f’’8.
g’’32 f’’32 e’’8 a’’8 d’’8 g’’8~ g’’16 a’’16 g’’16 f’’16
g’’8. a’’32 g’’32 ges’’8 b’’8 e’’8 a’’8~ a’’16 b’’16 a’’16
g’’16 c’’’4 ges’’’4 g’’’4 a’’’4 r8 c’’8 d’’8 e’’8 f’’8. g’’32
f’’32 e’’8 a’’8 d’’8 g’’8~ g’’16 a’’16 g’’16 f’’16 r32 r32 r32
r32 r32 r32 r32 r32 r32 r32 r32 r32 r32 r32 r32 r32 r32 r32 r32
r32 r32 r32 r32 r32
}
alto =  {

r8 c’8 d’8 e’8 f’8. g’32 f’32 e’8 a’8 d’8 g’8~ g’16 a’16
g’16 f’16 e’4 f’4 c’16 d’16 e’16 d’16 d’8~ a8 e’8 b8 c’32
 d’32 c’8. f’8. g’32 f’32 e’8 a’8 d’8 g’8~ g’16 a’16 g’16
 f’16 bes’4 e’4 f’16 g’16 a’16 g’16 g’8~ d’8 a’8 e’8 f’32
 g’32 f’8. e’8 d’8 c’8 r8 r8 c’8 d’8 e’8 f’8. g’32 f’32 e’8
a’8 d’8 g’8~ g’16 a’16 g’16 f’16 b4 f’4 c’16 d’16 e’16 d’16
d’8~ a8 e’8 b8 c’32 d’32 c’8. c’8. d’32 c’32 b8 e’8 a8 d’8~
d’16 e’16 d’16 c’16 f’4 b’4 c’’4 d’’4
}
tenor =  {
r8 r8 r8 r8 r8. r32 r32 r8 r8 r8 r8~ r16 r16 r16 r16
r8 r8 r8 r8 r8. r32 r32 r8 r8 r8 r8~ r16 r16 r16 r16
r8 c8 d8 e8 f8. g32 f32 e8 a8 d8 g8~ g16 a16 g16 f16
r4 c4 d4 e4 f4. g16 f16 e4 a4 d4 g4~ g8 a8 g8 f8 r8 c8
 d8 e8 f8. g32 f32 e8 a8 d8 g8~ g16 a16 g16 f16 r1 r1
}
\score{
\context StaffGroup <<
\new Staff <<\clef treble \soprano>>
\new Staff <<\clef treble  \alto>>
\new Staff <<\clef bass \tenor>>
>>
\layout{ }
\midi {

 }
}
```

Código 7.1.1: *Arquivo gerado pelo programa*

7.1.3 Partitura

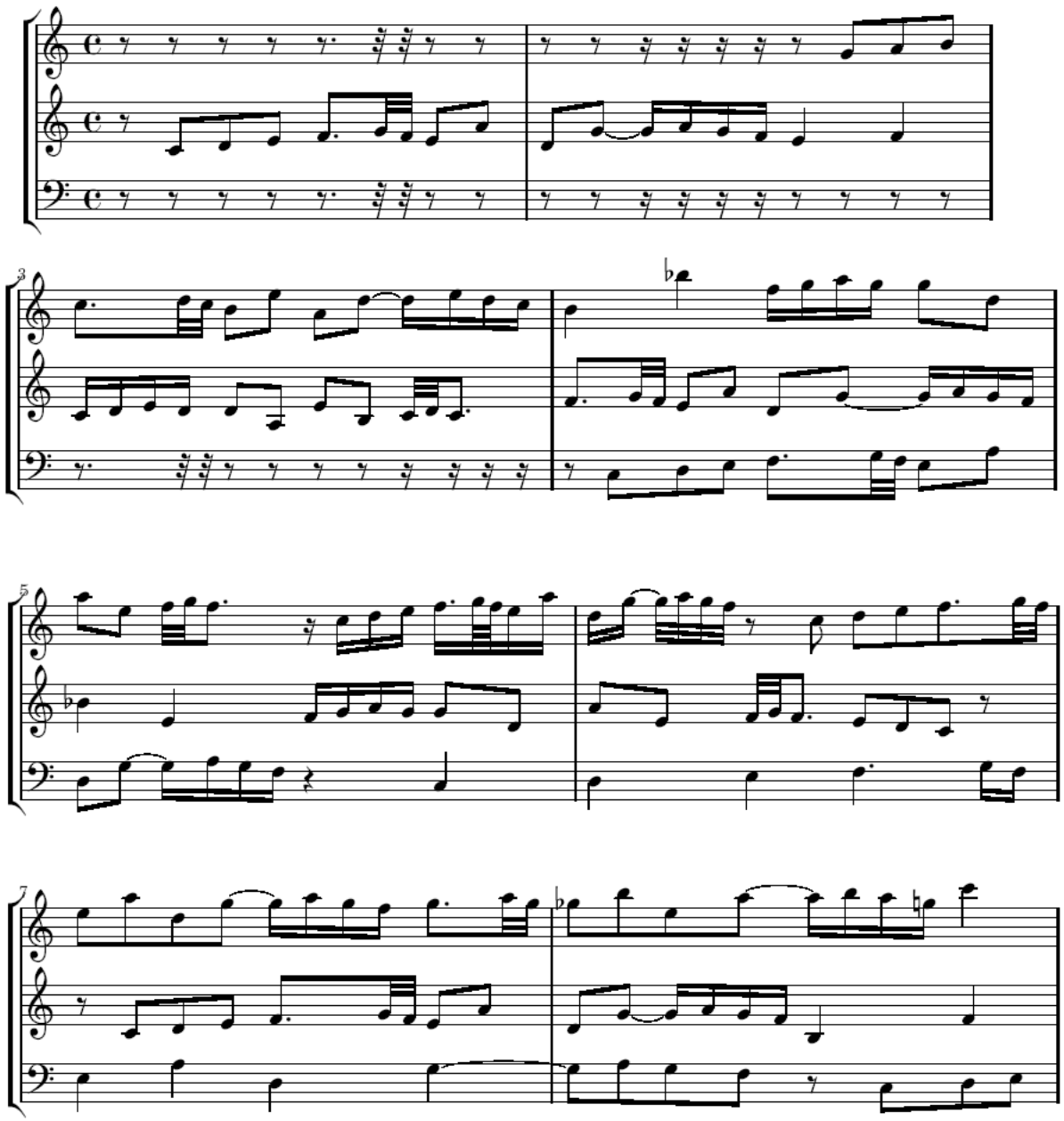

Figura 7.2: *Partitura*

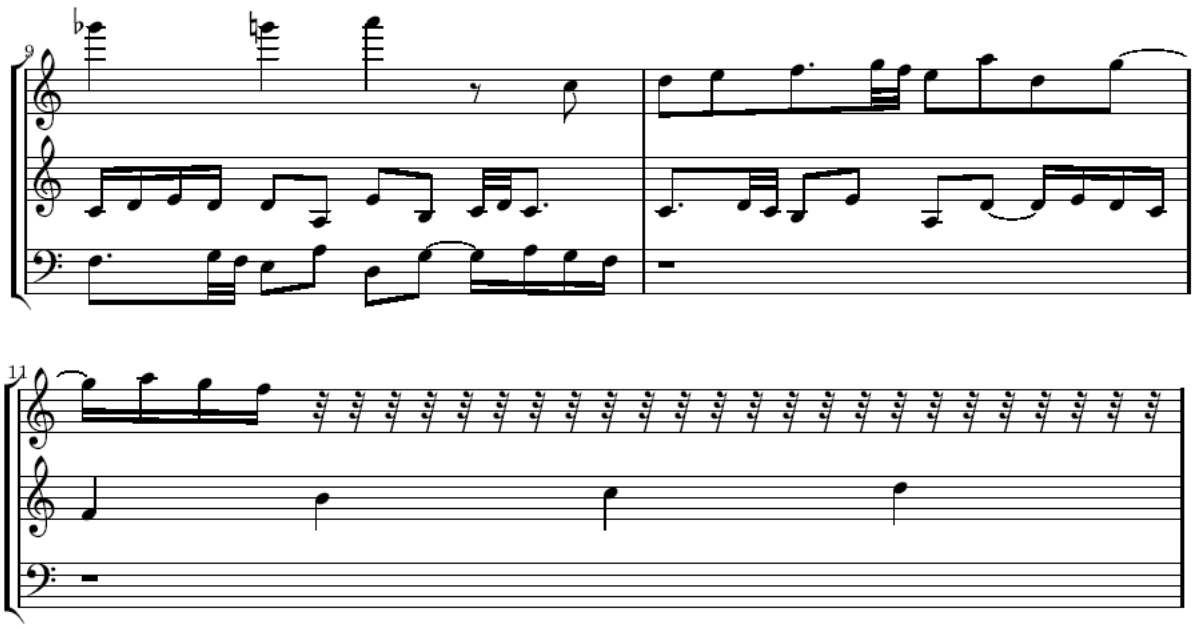

Figura 7.3: *Continuação da Partitura*

7.2 Exemplo 2: O Cravo Bem Temperado, Fuga II em Dó Menor (BWV 847)

7.2.1 Sujeito (entrada do programa)

A notação para este sujeito é:

r8 c”16 b’16 c”8 g’8 aes’8 c”16 b’16 c”8 d”8 g’8 c”16 b’16 c”8 d”8 f’16 g’16 aes’4 g’16 f’16 ees’16

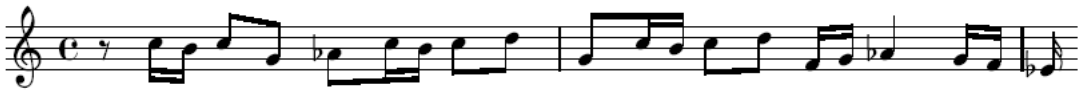

Figura 7.4: *Sujeito da Fuga*

7.2.2 Fuga (saída do programa)

```
soprano = {
r8 r16 r16 r8 r8 r8 r16 r16 r8 r8 r8 r16 r16 r8 r8 r16 r16
r4 r16 r16 r8 g’’16 ges’’16 g’’8 d’’8 ees’’8 g’’16 ges’’16
g’’8 a’’8 d’’8 g’’16 ges’’16 g’’8 a’’8 c’’16 d’’16 ees’’4 d’’16
 c’’16 bes’4 bes’’4 c’’’4 des’’’4 f’’16 g’’16 aes’’4 g’’16 f’’16
d’’’8 c’’’8 b’’16 c’’’16 g’’8 r16 c’’’32 b’’32 c’’’16 g’’16 aes’’16
c’’’32 b’’32 c’’’16 d’’’16 g’’16 c’’’32 b’’32 c’’’16 d’’’16 f’’32 g’’32
aes’’8 g’’32 f’’32 r8 c’’’16 b’’16 c’’’8 g’’8 aes’’8 c’’’16 b’’16
c’’’8 d’’’8 g’’8 c’’’16 b’’16 c’’’8 d’’’8 f’’16 g’’16 aes’’4 g’’16
f’’16 a’’8 d’’’16 des’’’16 d’’’8 e’’’8 g’’16 a’’16 bes’’4 a’’16 g’’16
ees’’’4 d’’’4 c’’’4 f’’’4 r8 c’’’16 b’’16 c’’’8 g’’8 aes’’8 c’’’16 b’’16
c’’’8 d’’’8 g’’8 c’’’16 b’’16 c’’’8 d’’’8 f’’16 g’’16 aes’’4 g’’16 f’’16 r1
}
alto =  {

r8 c’’16 b’16 c’’8 g’8 aes’8 c’’16 b’16 c’’8 d’’8 g’8 c’’16 b’16 c’’8 d’’8
 f’16 g’16 aes’4 g’16 f’16 ees’4 f’4 g’4 aes’4 c’16 d’16 ees’4 d’16 c’16 a’8
g’8 ges’16 g’16 d’8 g’8 c’’16 b’16 c’’8 d’’8 f’16 g’16 aes’4 g’16 f’16 des’’4
c’’4 bes’4 ees’4 f’16 g’16 aes’4 g’16 f’16 d’’8 c’’8 b’16 c’’16 g’8 d’’8 c’’8
b’16 c’’16 aes’8 g’8 c’’8 b’16 c’’16 r8 r8 c’’16 b’16 c’’8 g’8 aes’8 c’’16 b’16
c’’8 d’’8 g’8 c’’16 b’16 c’’8 d’’8 f’16 g’16 aes’4 g’16 f’16 bes4 f’4 g’4 aes’4
c’16 d’16 ees’4 d’16 c’16 a’8 g’8 ges’16 g’16 d’8 d’8 g’16 ges’16 g’8 a’8 c’16
d’16 ees’4 d’16 c’16 aes’4 g’4 f’4 bes’4
}
tenor =  {
r8 r16 r16 r8 r8 r8 r16 r16 r8 r8 r8 r16 r16 r8 r8 r16 r16 r4 r16 r16 r8 r16 r16
r8 r8 r8 r16 r16 r8 r8 r8 r16 r16 r8 r8 r16 r16 r4 r16 r16 r8 c’16 b16 c’8 g8
aes8 c’16 b16 c’8 d’8 g8 c’16 b16 c’8 d’8 f16 g16 aes4 g16 f16 r4 c’8 b8 c’4
 g4 aes4 c’8 b8 c’4 d’4 g4 c’8 b8 c’4 d’4 f8 g8 aes2 g8 f8 r8 c’16 b16 c’8 g8
aes8 c’16 b16 c’8 d’8 g8 c’16 b16 c’8 d’8 f16 g16 aes4 g16 f16 r1 r1
}
\score{
\context StaffGroup <<
\new Staff <<\clef treble \soprano>>
\new Staff <<\clef treble  \alto>>
\new Staff <<\clef bass \tenor>>
>>
\layout{ }
\midi {

 }
}
```

Código 7.2.1: *Arquivo gerado pelo programa*

7.2.3 Partitura

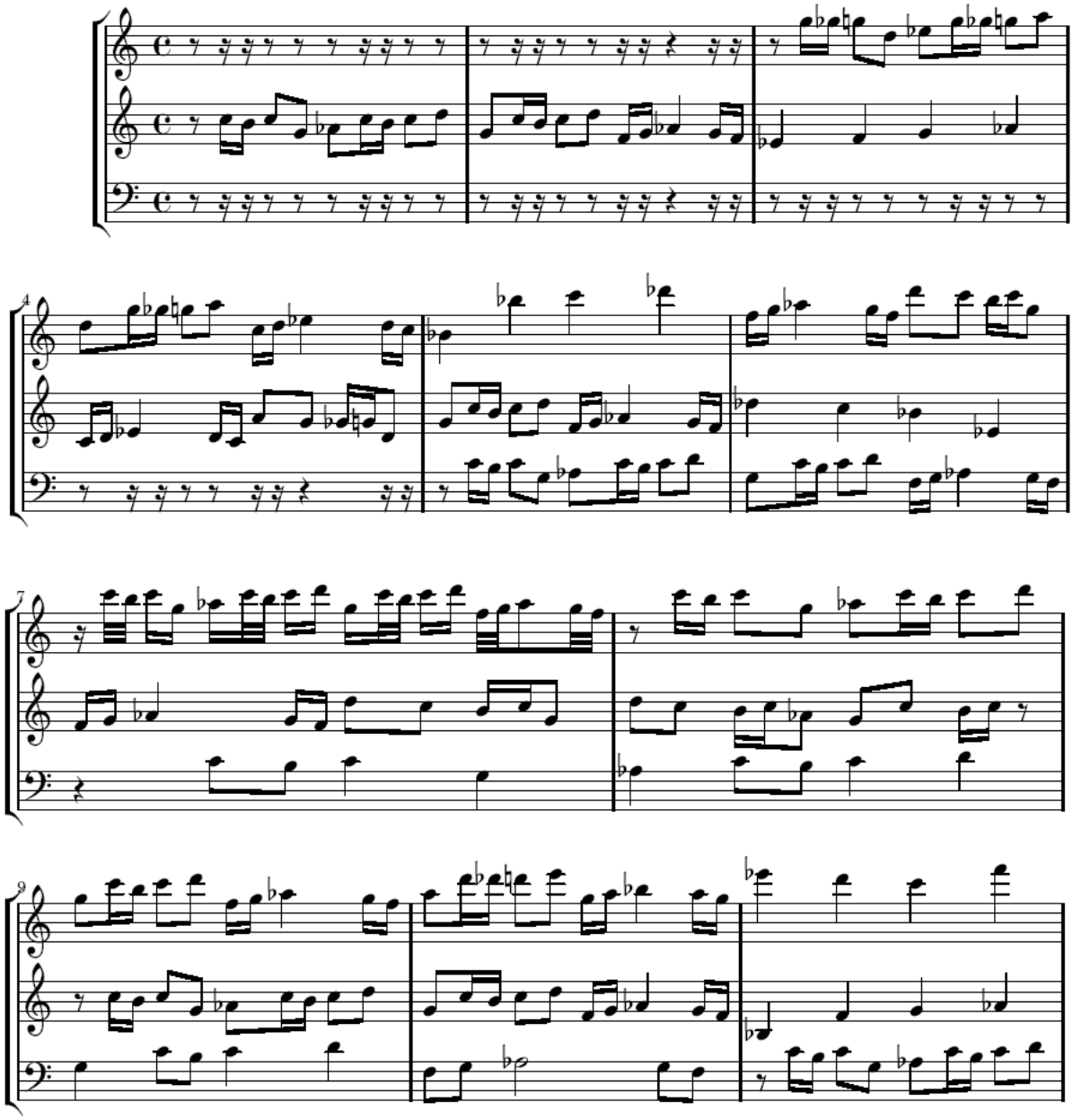

Figura 7.5: *Partitura*

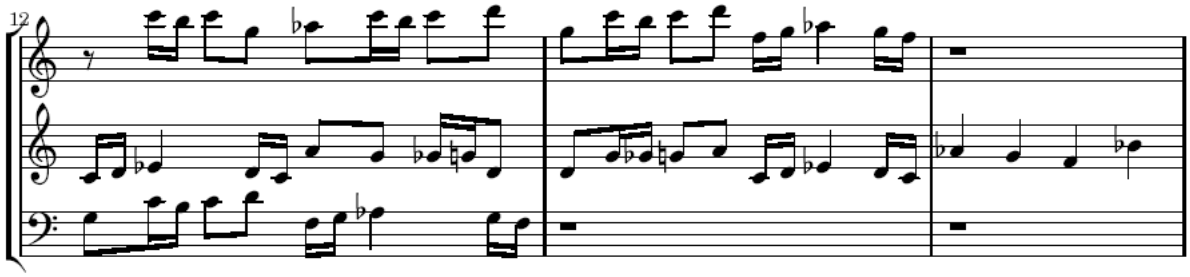

Figura 7.6: *Continuação da Partitura*

7.3 Exemplo 3: A Arte da Fuga, Contrapunctus I em Ré Menor (BWV 1080)

7.3.1 Sujeito (entrada do programa)

A notação para este sujeito é:
d'2 a'2 f'2 d'2 cis'2 d'4 e'4 f'2 f'8 g'8 f'8 e'8 d'4

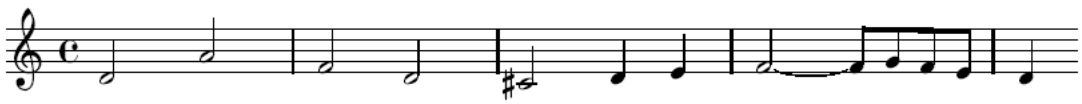

Figura 7.7: *Sujeito da Fuga*

7.3.2 Fuga (saída do programa)

```
soprano = {
r2 r2 r2 r2 r2 r4 r4 r2~ r8 r8 r8 r8 a'2 e''2 c''2 a'2
gis'2 a'4 b'4 c''2~ c''8 d''8 c''8 b'8 a'4 a''4 bes''4
c'''4 e''8 f''8 g''8 f''8 f''2~ e''4 d''4 cis''2 d''2 f''2
d''4 a''4 f''4 d''4 cis''4 d''8 e''8 f''4~ f''16 g''16 f''16
e''16 d''2 a''2 f''2 d''2 cis''2 d''4 e''4 f''2~ f''8 g''8
f''8 e''8 g''2 e''2 dis''2 e''4 ges''4 g''2~ g''8 a''8 g''8
ges''8 d'''4 c'''4 b''4 e'''4 d''2 a''2 f''2 d''2 cis''2 d''4
e''4 f''2~ f''8 g''8 f''8 e''8 r1 r1
}
alto =  {

d'2 a'2 f'2 d'2 cis'2 d'4 e'4 f'2~ f'8 g'8 f'8 e'8 d'4 e'4 f'4
g'4 b8 c'8 d'8 c'8 c'2~ b4 a4 gis2 a2 c'2 f'2 d'2 cis'2 d'4 e'4
f'2~ f'8 g'8 f'8 e'8 c''4 bes'4 a'4 d'4 e'8 f'8 g'8 f'8 f'2~ e'4
d'4 cis'2 d'2 f'2 a'2 d'2 d'2 a'2 f'2 d'2 cis'2 d'4 e'4 f'2~ f'8
g'8 f'8 e'8 a4 e'4 f'4 g'4 b8 c'8 d'8 c'8 c'2~ b4 a4 gis2 a2 c'2
c'2 a2 gis2 a4 b4 c'2~ c'8 d'8 c'8 b8 g'4 f'4 e'4 a'4
}
tenor =  {
r2 r2 r2 r2 r2 r4 r4 r2~ r8 r8 r8 r8 r2 r2 r2 r2 r2 r4 r4 r2~ r8 r8
r8 r8 d2 a2 f2 d2 cis2 d4 e4 f2~ f8 g8 f8 e8 d1 a1 f1 d1 cis1 d2 e2
f1~ f4 g4 f4 e4 d2 a2 f2 d2 cis2 d4 e4 f2~ f8 g8 f8 e8 r1 r1 r1 r1
}
\score{
\context StaffGroup <<
\new Staff <<\clef treble \soprano>>
\new Staff <<\clef treble  \alto>>
\new Staff <<\clef bass \tenor>>
>>
\layout{ }
\midi {

 }
}
```

Código 7.3.1: *Arquivo gerado pelo programa*

7.3.3 Partitura

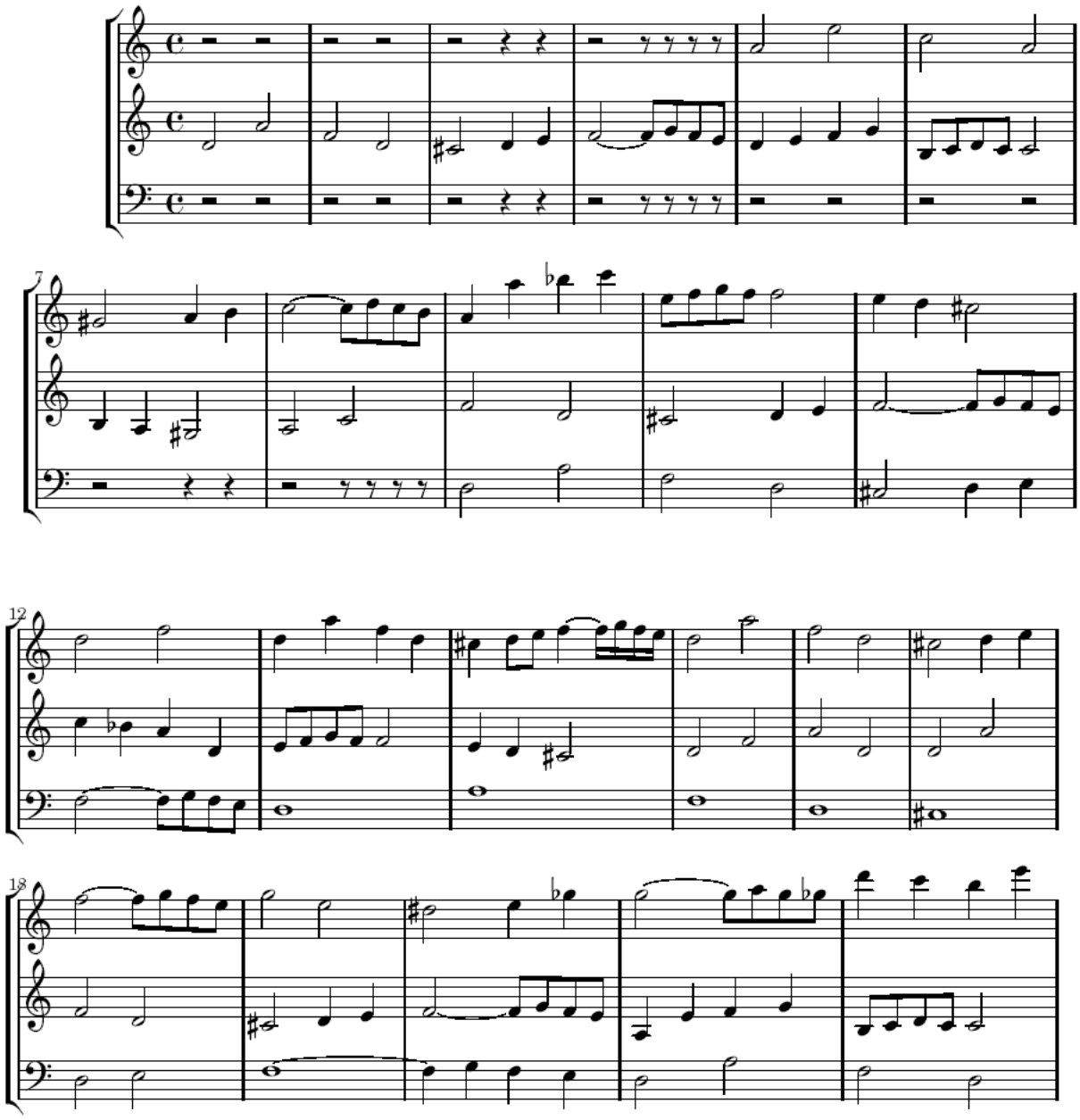

Figura 7.8: *Partitura*

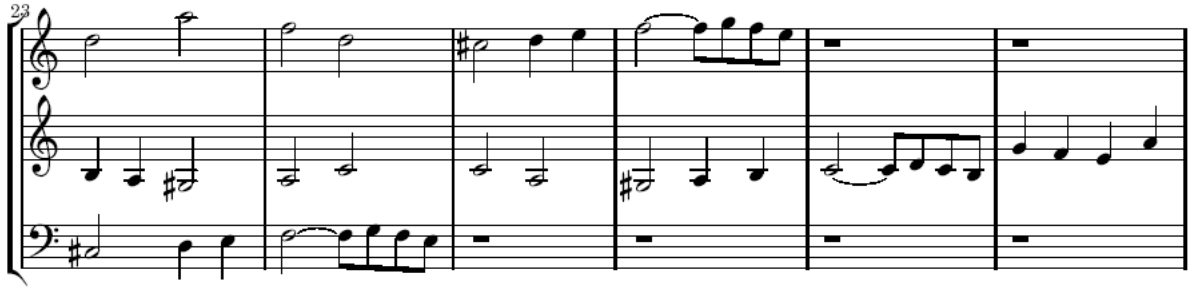

Figura 7.9: *Continuação da Partitura*

7.4 Exemplo 4: Tema do Desenho Animado "Os Simpsons"

7.4.1 Sujeito (entrada do programa)

A notação para este sujeito é:
c'4. e'8 e'8 fis'4 a'8 g'4. e'8 r8 c'8 r8 a8 fis8 fis8 fis8 g8 g4

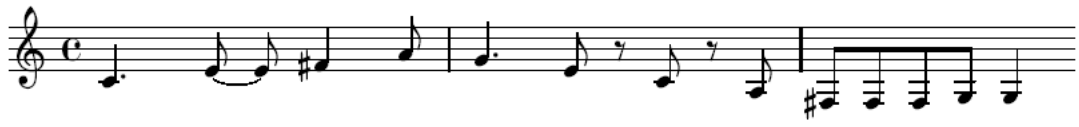

Figura 7.10: *Sujeito da Fuga*

7.4.2 Fuga (saída do programa)

```
soprano = {
r4. r8~ r8 r4 r8 r4. r8 r8 r8 r8 r8 r8 r8 r8 g'4. b'8~
b'8 cis''4 e''8 d''4. b'8 r8 g'8 r8 e'8 cis'8 cis'8 cis'8
d'8 d'4 c''4 g'8 fis'8 fis'8 fis'8 a'8 r8 c''8 r8 e''8 g''4.
a''8 fis''4 e''8 c''8. e''16~ e''16 fis''8 a''16 g''8. e''16
r16 c''16 r16 a'16 fis'16 fis'16 fis'16 g'16 c''4. e''8~ e''8
fis''4 a''8 g''4. e''8 r8 c''8 r8 a'8 fis'8 fis'8 fis'8 g'8 ges''8
gis''4 b''8 a''4. ges''8 r8 d''8 r8 b'8 gis'8 gis'8 gis'8 a'8 d''4
a''4 a''4 gis''4 c''4. e''8~ e''8 fis''4 a''8 g''4. e''8 r8 c''8 r8
a'8 fis'8 fis'8 fis'8 g'8 r32 r32 r32 r32 r32 r32 r32 r32 r1
}
alto =  {

c'4. e'8~ e'8 fis'4 a'8 g'4. e'8 r8 c'8 r8 a8 fis8 fis8 fis8 g8
g4 g4 d8 cis8 cis8 cis8 e8 r8 g8 r8 b8 d'4. e'8 cis'4 b8 e'8 fis'4
a'8 g'4. e'8 r8 c'8 r8 a8 fis8 fis8 fis8 g8 c'4 g4 g8 fis8 fis8 fis8
a8 r8 c'8 r8 e'8 g'4. a'8 fis'4 e'8 e'8~ c'4. c'4. e'8~ e'8 fis'4 a'8
g'4. e'8 r8 c'8 r8 a8 fis8 fis8 fis8 g8 d4 g4 d8 cis8 cis8 cis8 e8 r8
g8 r8 b8 d'4. e'8 cis'4 b8 b8 cis'4 e'8 d'4. b8 r8 g8 r8 e8 cis8 cis8
cis8 d8 g4 d'4 d'4 cis'4
}
tenor =  {
r4. r8~ r8 r4 r8 r4. r8 r8 r8 r8 r8 r8 r8 r8 r4. r8~ r8 r4 r8 r4.
r8 r8 r8 r8 r8 r8 r8 r8 c4. e8~ e8 fis4 a8 g4. e8 r8 c8 r8 a,8 fis,8
fis,8 fis,8 g,8 c2. e4~ e4 fis2 a4 g2. e4 r4 c4 r4 a,4 fis,4 fis,4 fis,4
g,4 c4. e8~ e8 fis4 a8 g4. e8 r8 c8 r8 a,8 fis,8 fis,8 fis,8 g,8 r1 r1 r1
}
\score{
\context StaffGroup <<
\new Staff <<\clef treble \soprano>>
\new Staff <<\clef treble  \alto>>
\new Staff <<\clef bass \tenor>>
>>
\layout{ }
\midi {

 }
}
```

Código 7.4.1: *Arquivo gerado pelo programa*

7.4.3 Partitura

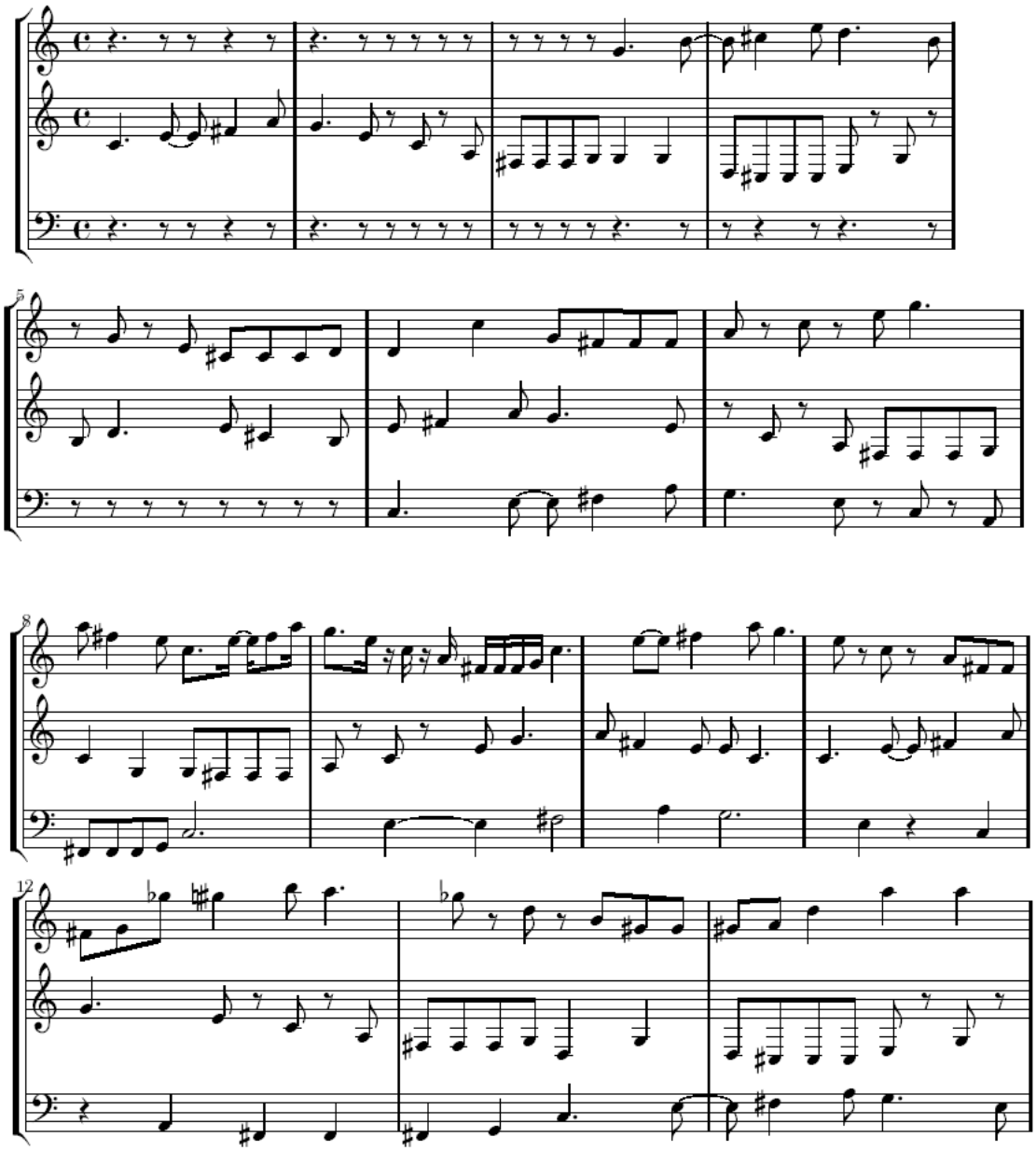

Figura 7.11: *Partitura*

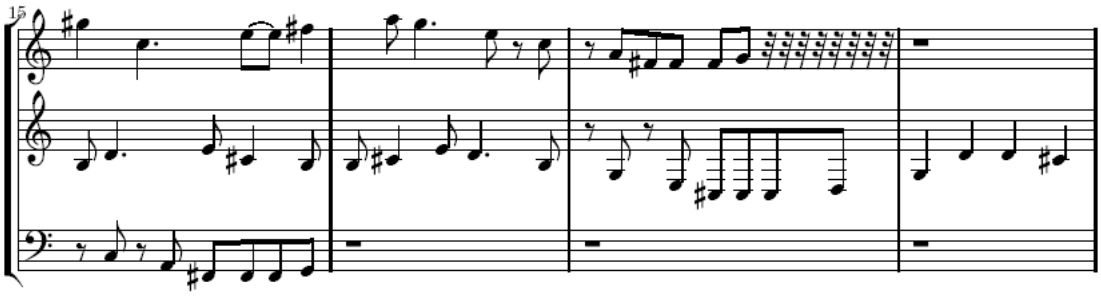

Figura 7.12: *Continuação da Partitura*

Capítulo 8

Conclusões e Trabalhos Futuros

O presente trabalho mostrou-se satisfatório na composição de uma fuga usando uma implementação computacional de um algoritmo. O julgamento estético dos resultados fica a cargo dos ouvintes das fugas geradas.

O uso desse algoritmo, do modo como foi implementado, permitiu somente uma única estruturação para as músicas geradas, resultando, assim, em um aspecto intrínseco das composições [11], mas nem por isso perdeu-se a variedade nos resultados obtidos.

O próprio ato de implementar o algoritmo gerador de fugas tornou-se uma forma de compor fugas. Mas uma vez pronto o programa, os resultados ficaram intimamente associados às entradas, ou seja, aos sujeitos.

Muitas modificações ainda podem ser feitas para aperfeiçoar o programa gerador de fugas, que ainda apresenta-se incompleto. Pode-se melhorá-lo para poder aceitar uma gama maior de sujeitos, ritmos ternários, entre outras coisas.

A parte de conclusão da fuga não foi implementada e seria o mais natural prosseguimento do trabalho.

Referências Bibliográficas

[1] http://lilypond.org/doc/v2.10/Documentation/, 2008.

[2] http://pt.wikipedia.org/wiki/Altura$(m\%C3\%BAsica)$, 2008.

[3] Andres Garay Acevedo. Fugue composition with counterpoint melody generation using genetic algorithms. In *Computer Music Modeling and Retrieval*. Springer, 2005.

[4] Harry B. Lincoln. The computer and music. Cornell University Press, Ithaca, New York, first edition edition, 1970.

[5] Johann Sebastian Bach. *Das Wohltemperierte Klavier*. G. Henle Verlag, München, 1960.

[6] Marcel Bitsch and Jean Bonfils. *La Fugue*. Éditions Combre, Paris, 1993.

[7] Amando Blanquer. *Tecnica del Contrapunto*. Real Musical S. A., Madri, 1975.

[8] Stephan Krehl. *Fuga*. Editorial Labor S. A., Barcelona, primera edición edition, 1930.

[9] Fred Lerdahl and Ray Jackendoff. *A Generative Theory of Tonal Music*. The MIT Press, Cambrige, Massachusets, 1996.

[10] Alfred Mann. *The Study of Fugue*. Faber and Faber, 24 Russell Square, London, 1958.

[11] Martin Supper. A few remarks on algorithmic composition. *Computer Music Journal*, pages 48–53, 2001.