

**Universidade do Minho**

Escola de Engenharia

Departamento de Informática

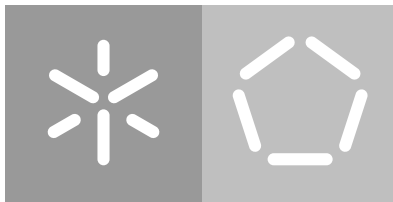
Pedro M. Silva

**First Part of Title**

**Second Part of Title**

**Criação de acompanhamentos e  
e teclados musicais programáveis**

December 2019



**Universidade do Minho**

Escola de Engenharia

Departamento de Informática

Pedro M. Silva

**First Part of Title**  
**Second Part of Title**

**Criação de acompanhamentos e  
e teclados musicais programáveis**

Master dissertation

Master Degree in Computer Science

Dissertation supervised by

**The Supervisor of the thesis**

**The cosupervisor of the thesis**

December 2019

---

## ACKNOWLEDGEMENTS

---

Write acknowledgements here

---

## ABSTRACT

---

Write abstract here (en) or import corresponding file

---

## RESUMO

---

Escrever aqui resumo (pt) ou importar respectivo ficheiro

---

## CONTENTS

---

1	INTRODUCTION	1
2	ESTADO DA ARTE	2
2.1	Trabalho Relacionado	2
2.1.1	Alda	2
2.1.2	ABC Notation	3
2.1.3	Faust	3
2.1.4	Sonic Pi	5
2.2	Gramáticas	5
2.2.1	Diferenças: CFG vs PEG	5
2.3	SoundFonts	7
2.4	Sintetizadores	7
2.4.1	Inicialização	7
2.4.2	Utilização	8
2.5	Resumo	9
3	O PROBLEMA E OS SEUS DESAFIOS	10
3.1	Solução Proposta	10
3.1.1	Gramática da Linguagem	10
3.1.2	Arquitetura do Sistema	11
4	DEVELOPMENT	13
4.1	Decisions	13
4.2	Implementation	13
4.3	Outcomes	13
4.4	Summary	13
5	CASE STUDIES / EXPERIMENTS	14
5.1	Experiment setup	14
5.2	Results	14
5.3	Discussion	14
5.4	Summary	14
6	CONCLUSION	15
6.1	Conclusions	15
6.2	Prospect for future work	15
A	SUPPORT MATERIAL	17

---

## LIST OF FIGURES

---

Figure 1	caption	12
----------	---------	----

---

## LIST OF TABLES

---



---

## INTRODUCTION

---

O objetivo deste trabalho é estudar e prototipar formas de criação de música com recurso a técnicas habitualmente usadas na criação de *software*. Para além de permitir criar música através das notas introduzidas manualmente, a linguagem deve facilitar a geração de música de um modo mais dinâmico, com recurso a operações de combinação e transformação de notas.

O termo música neste contexto é usado num sentido mais amplo que apenas sons. O objetivo desta linguagem é permitir gerar vários *outputs* através do mesmo código fonte, como pautas musicais, ABC, WAV, MIDI, entre outros.

Uma das partes mais críticas relativas à pesquisa e desenvolvimento necessários para a realização da linguagem é a componente temporal implícita em todos os aspetos da linguagem: deve ser possível de um modo intuitivo expressar as várias composições possíveis de notas sem ser necessário expressar os tempos manualmente, tais como notas sequenciais, notas em paralelo, pequenas pausas e grandes pausas, bem como sincronizar partes da música de modo simples.

---

## ESTADO DA ARTE

---

Atualmente a produção de música é realizada utilizando programas com interfaces gráficas, geralmente denominados como [Digital Audio Workstation \(DAW\)](#). A minha abordagem irá consistir em estudar formas de criar e tocar ao vivo músicas (e não só) através de uma [Domain Specific Language \(DSL\)](#), usando técnicas inspiradas nas linguagens de programação e no desenvolvimento de *software*.

### 2.1 TRABALHO RELACIONADO

Existem diversos tipos de linguagens usadas atualmente para produzir ou simplesmente descrever música. Algumas fazem uso do conceito de notas musicais, com recurso a algum sintetizador externo, para gerar os sons, enquanto outras funcionam com base na manipulação direta de ondas de som digitais para criar música. Algumas suportam apenas a descrição estática da música, enquanto outras permitem formas dinâmicas tais como funções, variáveis, estruturas de controlo e repetição, ou até mesmo algoritmos aleatórios que permitem gerar músicas diferentes a cada execução.

Iremos de seguida analisar algumas das soluções disponíveis atualmente, bem como comparar as funcionalidades que cada uma oferece ou não oferece.

#### 2.1.1 *Alda*

A projeto **alda**<sup>[1]</sup> é uma linguagem de música textual desenvolvida em *JAVA* focada na simplicidade: o seu maior ponto de atração é apelar tanto a programadores com pouca experiência musical, bem como a músicos com pouca experiência com programação. Apesar de ser anunciada como direcionada tanto a músicos como a programadores, a linguagem não suporta nenhum tipo de construções dinâmicas, como ciclos ou funções. Este tipo de funcionalidades, se necessário, requer o uso de uma linguagem de programação por cima, que poderia por exemplo, gerar o código *alda* em *runtime* através da manipulação de *strings*.

### Exemplos

O exemplo seguinte demonstra um simples programa escrito em *alda*, demonstrando: a seleção de um instrumento (piano:), a definição da oitava base (o3), um acorde com quatro notas (c1/e/g/>c4) em que a última se encontra uma oitava acima das outras.

```
piano: o3 c1/e/g/>c4 < b a g | < g+1/b/>e
```

Listing 2.1: Exemplo da linguagem alda

É também possível verificar o uso de acidentais (identificados pelos símbolos + ou - a seguir a uma nota) bem como a diferenciação da duração de algumas notas (identificadas pelos números em frente às notas).

#### 2.1.2 ABC Notation

A notação **ABC**[2] é uma notação textual que permite descrever notação musical. É bastante completa, tendo formas de descrever notas, acordes, acidentais, uniões de notas, *lyrics*, múltiplas vozes, entre outros.

Para além da exaustividade de sintaxe que permite descrever quase todo o tipo de música, a popularidade da linguagem também significa que existem já inúmeros conversores de ficheiros ABC para os mais diversos formatos, desde ficheiros MIDI, pautas musicais, ou mesmo ficheiros WAV (gerados através do fornecimento de um ficheiro SoundFont, por exemplo).

A complexidade da notação é uma faca de dois gumes, no entanto: A sua ubiquidade significa que uma maior percentagem de utilizadores já se pode sentir à vontade com a sintaxe, o que não acontece com outras linguagens menos conhecidas. Mas por outro lado, conhecer ou implementar toda a especificação [3] é uma realização bastante difícil.

#### 2.1.3 Faust

A linguagem **Faust**[4] é uma linguagem de programação funcional com foco na sintetização de som e processamento de áudio. Ao contrário das linguagens analisadas até agora, não trabalha com abstrações de notas e elementos musicais. Em vez disso, a linguagem trabalha diretamente com ondas sonoras (representadas como *streams* de números) e através de expressões matemáticas, permite assim manipular o som produzido.

Um dos pontos fortes da linguagem é o facto da sua arquitetura ser construída de raiz para compilar o mesmo código fonte em várias linguagens. De facto, o projeto conta com várias dezenas de *targets*, desde os mais óbvios (C, C++, Java, JavaScript) até alguns mais

especializados (WebAssembly, LLVM Bitcode, instrumentos VST/VSTi). Também permite gerar aplicações *standalone* para as bibliotecas de audio mais comuns[5].

A linguagem vem embutida com uma biblioteca extremamente completa[6] que implementa, entre muitas outras, funções de matemática comuns, filtros audio, funcionalidades extremamente básicas de interfaces gráficas que permitem controlar em tempo real os valores do programa (com botões e *sliders* entre outros).

### Exemplos

A documentação do projeto conta com uma quantidade abundante de exemplos[7] e com um tutorial para iniciantes à [8], do qual irei colocar aqui alguns pequenos pedaços de código que demonstram algumas das capacidades da linguagem.

```
import("stdfaust.lib");  
process = no.noise*0.5;
```

Listing 2.2: Geração de ruído aleatório com volume a metade

No primeiro exemplo, podemos ver a estrutura mais básica de um programa escrito em *Faust*. Na primeira linha é importada a biblioteca *standard* da linguagem. Na segunda linha podemos ver a *keyword* **process**, que representa o *input* e *output* audio do nosso programa. Finalmente, em frente a essa *keyword* podemos ver a expressão `no.noise*0.5`. Isto demonstra a utilização de construções da biblioteca *standard*, como o gerador de ruído, bem como a utilização de operadores matemáticos usuais (neste caso a multiplicação) para manipular o audio, e diminuir o volume para metade.

```
import("stdfaust.lib");  
ctFreq = 500;  
q = 5;  
gain = 1;  
process = no.noise : fi.resonlp(ctFreq,q,gain);
```

Listing 2.3: Geração de ruído aleatório com um filtro *low-pass*

Neste exemplo, estamos a usar o operador `:` para canalizar o output do gerador de ruído para um filtro *low-pass*, que filtra todas as frequências acima de um valor de corte (a variável `ctFreq`). Aumentar esta variável resulta num som mais agudo, enquanto que ao diminui-la obtemos um som mais grave (pois o valor de corte é mais baixo, apenas os sons abaixo desse valor são passados).

```
import("stdfaust.lib");
```

```

ctFreq = hslider("[0]cutoffFrequency",500,50,10000,0.01);
q = hslider("[1]q",5,1,30,0.1);
gain = hslider("[2]gain",1,0,1,0.01);
t = button("[3]gate");
process = no.noise : fi.resonlp(ctFreq,q,gain)*t;

```

Listing 2.4: Geração de ruído aleatório com um filtro *low-pass* controlada por uma interface

Por fim podemos ver um exemplo igual ao anterior, mas em vez de ter os valores das variáveis estáticos (guardados nas variáveis `ctFreq`, `q` e `gain`), estes são controlados em tempo real pela interface definida pelas chamadas à função `hslider`. Foi também adicionada uma variável `t` com um botão "*gate*". Este produz o valor 0 quando está solto, e o valor 1 quando está pressionado, valor que quando multiplicado pelo resto da expressão serve efetivamente como um *on-off switch* para todo o sistema.

#### 2.1.4 *Sonic Pi*

d

## 2.2 GRAMÁTICAS

Para além dos aspetos técnicos da geração e reprodução de música já abordados neste relatório, existe também um componente fulcral relativo à análise e interpretação da linguagem que irá controlar a geração dos sons. Uma das primeiras decisões a ser tomada diz respeito à escolha do *parser*, e possivelmente, do tipo de gramática que irá servir de base para a geração do mesmo.

Tradicionalmente, as gramáticas mais populares no campo de processamento de texto tendem a ser [Context Free Grammar \(CFG\)](#), que são usadas como *input* nos geradores de *parser* mais populares (Bison/YACC, ANTLR). Existem no entanto alternativas, algumas mais recentes, como as [Parsing Expression Grammar \(PEG\)](#), que trazem consigo diferenças que podem ser consideradas por alguns como vantagens ou desvantagens.

### 2.2.1 *Diferenças: CFG vs PEG*

A diferença com maiores repercussões práticas entre as duas classes de gramáticas deve-se à semântica atribuída ao operador de escolha, e a consequente **ambiguidade** (ou falta dela) na gramática. Nas gramáticas [PEG](#), o operador é ordenado, o que significa que a ordem porque as alternativas aparecem é relevante durante o *parse* do *input*. Isto contrasta com a semântica nas [CFG](#), onde a ordem das alternativas é irrelevante. Isto pode no entanto levar

a ambiguidades, onde o mesmo *input*, descrito pela mesma gramática, pode resultar em duas árvores de *parsing* diferentes. Isto é, as CFG podem por essa razão ser ambíguas.

Tomemos como exemplo o famoso problema do *dangling else*[9] descrito nas duas classe de gramáticas:

```
if (a) if (b) f1(); else f2();
```

Listing 2.5: Gramática

```
statement = ...
  | conditional_statement

conditional_statement = ...
  | IF ( expression ) statement ELSE statement
  | IF ( expression ) statement
```

No caso de uma CFG, sabendo que o operador de escolha | é comutativo, o seguinte *input* será ambíguo, podendo resultar num *if-else* dentro do *if* ou num *if* dentro de um *if-else*.

Mas no caso de uma PEG, o resultado é claro: um *if-else* dentro de um *if*. Quando a primeira regra do condicional chega ao statement, este vai por sua vez chamar o não terminal conditional\_statement, que por sua vez irá consumir o *input* até ao fim. Deste modo, quando a execução voltar ao primeiro conditional\_statement, esta irá falhar por não conseguir ler o *else* (uma vez que já consumimos todo o texto de entrada). Deste modo irá usar a segunda alternativa, dando então o resultado previsto.

Com este exemplo de *backtracking* podemos também verificar um problema aparente nas gramáticas PEG. Falhando a primeira alternativa na produção conditional\_statement, a segunda irá ser testada. Mas é evidente, olhando para a gramática que a segunda alternativa é exatamente igual à parte inicial da primeira alternativa (que neste caso também corresponde á parte que teve sucesso). Em vez de voltar a testar as regras de uma forma *naïve*, as *Parsing Expression Grammar* guardam antes em *cache* os resultados de testes anteriores, permitindo assim uma pesquisa em tempo linear relativamente ao tamanho do *input*, à custa de uma maior utilização de memória.

### Resumo

Em resumo, as três principais diferenças entre as tradicionais *Context Free Grammar* (CFG) e as mais recentes *Parsing Expression Grammar* (PEG) são:

**Ambiguidade.** O operador de escolha ser comutativo nas CFG resulta em gramáticas que podem ser ambíguas para o mesmo *input*. As PEG são determinísticas, mas exigem mais cuidado na ordem das produções, uma vez que tal afeta a semântica da gramática.

**Memoization** Para evitar *backtracking* exponencial, as PEG utilizam *memoization* que lhes permite guardar em *cache* resultados parciais durante o processo de *parsing*. Isto reduz o tempo dispendido, pois evita fazer o *parse* do mesmo texto pela mesma regra duas vezes. Mas também aumenta o consumo de memória, pois os resultados parciais têm de ser guardados até a análise terminar por completo.

**Composição** As *Parsing Expression Grammar* também têm a vantagem de oferecerem uma maior facilidade de composição. Em qualquer parte da gramática é possível trocar um terminal por um não terminal. Isto é, é extremamente fácil construir gramáticas mais modulares e compô-las entre si.

## 2.3 SOUND FONTS

O formato *SoundFont* foi originalmente desenvolvido nos anos 90 pela empresa E-mu Systems para ser usado inicialmente pelas placas de som Sound Blaster. Ao longo dos anos o formato sofreu diversas alterações, encontrando-se atualmente na versão 2.04, lançada em 2005[10]. Atualmente existem diversos sintetizadores de software *cross platform* e open source capazes de converterem eventos *MIDI* em som usando ficheiros *SoundFont*, dispensando a necessidade de uma placa de som compatível com o formato. Alguns destes projetos são TiMidity++, WildMIDI e FluidSynth.

Um ficheiro de *SoundFont* é constituído por um ou mais bancos (*banks*) (até um máximo de 128). Cada banco pode por sua vez ter até 128 *presets* (por vezes também chamados instrumentos ou programas).

## 2.4 SINTETIZADORES

A biblioteca FluidSynth é um *software* sintetizador de áudio em tempo real que transforma dados *MIDI* em sons, que podem ser gravados em disco ou encaminhados diretamente para um *output* de áudio. Os sons são gerados com recurso a *SoundFonts*[10] (ficheiros com a extensão *.sf2*) que mapeiam cada nota para a gravação de um instrumento a tocar essa nota.

Os *bindings* da biblioteca para C# foram baseados no código *open source* do projeto NFluidSynth[11], com algumas modificações para compilar com a versão da biblioteca em Linux.

### 2.4.1 Inicialização

Para utilizar a biblioteca FluidSynth, existem três objetos principais que devem ser criados: *Settings* (*fluid\_settings\_t\**), *Synth* (*fluid\_synth\_t\**) e *AudioDriver* (*fluid\_audio\_driver\_t\**).

O objecto **Settings**[12] é implementado com recurso a um dicionário. Para cada chave (por exemplo, “audio.driver”) é possível associar um valor do tipo inteiro (`int`), *string* (`str`) ou *double* (`num`). Alguns valores podem ser também booleanos (`bool`), no entanto eles são armazenados como inteiros com os valores aceites sendo apenas 0 e 1.

O objeto **Synth** é utilizado para controlar o sintetizador e produzir os sons. Para isso é possível enviar as mensagens MIDI tais como `NoteOn`, `NoteOff`, `ProgramChange`, entre outros.

O terceiro objeto **AudioDriver** encaminha automaticamente os sons para algum *audio output*, seja ele colunas no computador ou um ficheiro em disco. Os seguintes *outputs* são suportados pela biblioteca:

LINUX: jack, alsa, oss, PulseAudio, portaudio, sdl2, file

WINDOWS: jack, PulseAudio, dsound, portaudio, sdl2, file

MAX OS: jack, PulseAudio, coreaudio, portaudio, sndman, sdl2, file

ANDROID: opensles, oboe, file

#### 2.4.2 Utilização

Com os objetos necessários inicializados, é necessário ainda especificar qual (ou quais) a(s) *SoundFont(s)* a utilizar. Para isso podemos chamar o método `Synth.LoadSoundFont` que recebe dois argumentos: uma *string* com o caminho em disco do ficheiro *SoundFont* a carregar, seguido dum booleano que indica se os *presets* devem ser atualizados para os da nova *SoundFont* (isto é, atribuir os instrumentos da *SoundFont* aos canais automaticamente).

A função `Synth.NoteOn` recebe três argumentos: um inteiro a representar o canal, outro inteiro entre 0 e 127 a representar a nota, e finalmente outro inteiro também entre 0 e 127 a representar a velocidade da nota.

O canal (**channel**) representa qual o instrumento que vai reproduzir a nota em questão. Cada canal está atribuído a um programa da *SoundFont*, e é possível a qualquer momento mudar o programa atribuído a qualquer canal através do método `Synth.ProgramChange`. Caso se tenha carregado mais do que uma *SoundFont*, é possível usar o método `Synth.ProgramSelect`, que permite especificar o id da *SoundFont* e do banco do instrumento a atribuir.

A chave (**key**) representa a nota a tocar. Sendo este valor um inteiro entre 0 e 127, é necessário saber como mapear as tradicionais notas musicais neste valor. Para isso, basta colocarmos as *pitch classes* e os seus respetivos acidentais *sharp* numa lista ordenada (C, C#, D, D#, E, F, F#, G, G#, A, A#, B) e associar a eles os inteiros entre 0 e 11 (inclusive). Depois apenas temos de somar a esse número a multiplicação da oitava da nota (a começar em 0) por 12. Podemos deste modo calcular, por exemplo, que a *key* do C central



(C4) é igual a 48 ( $0 + 4 * 12$ ). Assim, podemos generalizar que para uma oitava  $O$  e para um tom de nota  $N$ , obtemos a chave aplicando a fórmula:

$$N + O * 12$$

A velocidade (**velocity**) é também um valor entre 0 e 127. Relacionando a velocidade com um piano físico, esta representa a força (ou velocidade) com que a tecla foi premida. Velocidades maiores geram sons mais altos, enquanto que velocidades mais baixas geram sons mais baixos, permitindo assim ao músico dar ou tirar ênfase a uma nota relativamente às restantes. De notar que um valor igual a zero é o equivalente a invocar o método `Synth.NoteOff`.

A método `Synth.NoteOff`, por sua vez, recebe apenas dois argumentos (canal e chave), e deve ser chamada passado algum tempo para terminar a nota. Podemos deste modo construir a analogia óbvia que o método `NoteOn` corresponde a uma tecla de piano ser premida, e `NoteOff` corresponde a essa tecla ser libertada.

## 2.5 RESUMO

---

## O PROBLEMA E OS SEUS DESAFIOS

---

The problem and its challenges.

### 3.1 SOLUÇÃO PROPOSTA

In this section, it is presented various ways to display an image.

#### 3.1.1 Gramática da Linguagem

A gramática de expressões ou acompanhamentos musicais tem como base fundamental os seguintes blocos: notas, pausas e modificadores. As notas são identificadas pelas letras A até G, seguindo a notação de *Helmholtz*<sup>[13]</sup> para denotar as respectivas oitavas. Podem também ser seguidas de um número ou de uma fração, indicando a duração da nota.

##### *Exemplos de notas*

C, , C, C c c' c'' c''' c'/4 A1/4 B2

As notas podem depois ser compostas sequencialmente (como demonstrado em cima) ou em paralelo (separados por uma barra vertical |). Devemos notar que o operador paralelo tem a menor precedência de todos, pelo que não é necessário agrupar as notas com parênteses quando se usa. Isto é, as duas expressões seguintes são equivalentes.

A B C | D E F  
( A B C ) | ( D E F )

É também possível agrupar estes blocos com recurso a parênteses. Os grupos herdam o contexto da expressão superior, mas as modificações ao seu contexto permanecem locais.

Isto permite, por exemplo, modificar configurações para apenas um conjunto restrito de notas. No exemplo seguinte, a velocidade da nota C é 70, mas para o grupo de notas A B a velocidade é 127.

```
v70 (v127 A B) C
```

Os modificadores disponíveis são:

**VELOCITY** A velocidade das notas, tendo o formato `[vV][0-9]+`.

**DURAÇÃO** A duração das notas, tendo o formato `[lL][0-9]+` ou `[lL][0-9]+/[0-9]+`.

**TEMPO** O número de batidas por minuto (BPM) que definem a velocidade a que as notas são tocadas, tendo o formato `[tT][0-9]+`.

**ASSINATURA DE TEMPO** Define a assinatura de tempo, que define o tipo de batida da música e o comprimento de uma barra na pauta musical. Tem o formato `[sS][0-9]+/[0-9]`.

É também possível definir qual o instrumento a ser utilizado para as notas. Todas as notas pertencentes ao mesmo contexto depois do modificador utilizarão esse instrumento.

```
(:cello A F | :violin A D)
```

Para além destas funcionalidades, também existe algum açúcar sintático para algumas das tarefas mais comuns na construção de acompanhamentos, como tocar acordes ou repetir padrões.

```
( [BG]*2 [B2G2] ) *3
```

### 3.1.2 Arquitetura do Sistema

A block diagram of the planned system / approach Here we have an example of inserting an image between the text paragraphs.






Figure 1: caption

Here we have how an image can be wrapped into the text without having surrounding space, and taking advantage of the space to be disposed on the side, without breaking the text readability.

This approach also benefits from the fact that the text will be related implicitly to the image on its side, although it should be referenced on the text anyway, otherwise, it should be consulting to perceive to which paragraph the image is related to.



Here is how we place an image as a floating body. Take in attention that the image is displayed on the next page, because there's no more room in this page.

You can also use an image as an icon, eg. , in the main text. Click on it to visit the website. It is also listed in the list of terms. Another example of an item to appear in the term index:

---

## DEVELOPMENT

---

### 4.1 DECISIONS

### 4.2 IMPLEMENTATION

### 4.3 OUTCOMES

Main result(s) and their scientific evidence

### 4.4 SUMMARY

---

## CASE STUDIES / EXPERIMENTS

---

Application of main result (examples and case studies)

5.1 EXPERIMENT SETUP

5.2 RESULTS

5.3 DISCUSSION

5.4 SUMMARY

---

## CONCLUSION

---

Conclusions and future work.

### 6.1 CONCLUSIONS

### 6.2 PROSPECT FOR FUTURE WORK

---

## BIBLIOGRAPHY

---

- [1] alda. <https://alda.io/>.
- [2] Abc notation. <http://abcnotation.com/>.
- [3] Abc notation standard v2.1, 2011.
- [4] Faust. <https://faust.grame.fr/>.
- [5] Faust targets. <https://faust.grame.fr/doc/manual/#a-quick-tour-of-the-faust-targets>.
- [6] Faust libraries. <https://faust.grame.fr/doc/libraries/>.
- [7] Faust examples. <https://faust.grame.fr/doc/libraries/>.
- [8] Faust quick start. <https://faust.grame.fr/doc/examples/index.html>.
- [9] Dangling else. [https://en.wikipedia.org/wiki/Dangling\\_else](https://en.wikipedia.org/wiki/Dangling_else).
- [10] Soundfont technical specification. <http://www.synthfont.com/sfspec24.pdf>, February 2006.
- [11] Atsushi Eno. Nfluidsynth. <https://github.com/atsushieno/nfluidsynth>, 2019.
- [12] Fluidsynth settings. <http://www.fluidsynth.org/api/fluidsettings.xml>.
- [13] Helmholtz pitch notation. [https://en.wikipedia.org/wiki/Helmholtz\\_pitch\\_notation](https://en.wikipedia.org/wiki/Helmholtz_pitch_notation).





---

## SUPPORT MATERIAL

---

Auxiliary results which are not main-stream; or

Details of results whose length would compromise readability of main text; or

Specifications and Code Listings: should this be the case; or

Tooling: Should this be the case.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.