



Universidad Carlos III de Madrid
Escuela Politécnica Superior



Ingeniería Informática

Proyecto Fin de Carrera

Modeling and Indexing Musical Files to allow Music Reuse

Report

Versión:	1.0
Creación:	Diciembre de 2006
Autor:	Julián Urbano Merino
Tutor:	Juan Llorens Morillo

This page is intentionally left blank.

Abstract

The goal of this degree thesis is to provide the necessary mechanisms to allow the music reuse.

The main musical information retrieval methods nowadays are based on text retrieval techniques applied to the music metadata, such as the author, title or musical kind. However, this degree thesis establishes the theoretic and practical bases to allow the music retrieval based on the own musical content, extending the possible scenarios where this kind of information retrieval can be applied in.

On the other hand, the retrieval process is based on the RSHP information representation model and on the CAKE retrieval framework. Thus, this degree thesis also intends to demonstrate the versatility and extension capacity of the RHSP model, applicable to whatever the information domain.

Resumen

El objetivo de este proyecto fin de carrera es proporcionar los mecanismos necesarios para hacer posible la reutilización de música.

Los principales métodos de recuperación de información musical de hoy en día se basan en técnicas de recuperación de texto aplicadas a metadatos de la música, como por ejemplo el autor, título o género musical. Sin embargo, este proyecto fin de carrera sienta las bases teóricas y prácticas para permitir la recuperación de música basada en el propio contenido musical, ampliando los posibles escenarios de este tipo de recuperación de información.

Por otro lado, el proceso de recuperación se basa en el modelo de representación de información RSHP y en el framework de recuperación CAKE. Así, este proyecto fin de carrera pretende también demostrar la versatilidad y capacidad de ampliación del modelo RSHP, aplicable a cualquier dominio de información.

Un resumen completo y en español del proyecto se encuentra en la Parte XII de este documento.

Juan Llorens Morillo, como profesor del Departamento de Informática de la Universidad Carlos III de Madrid, adscrito a la Escuela Politécnica Superior,

CERTIFICA: que el presente Proyecto Fin de Carrera, titulado *Modeling and Indexing Musical Files to allow Music Reuse*, ha sido desarrollado bajo su dirección en la Escuela Politécnica de la Universidad Carlos III de Madrid por Julián Urbano Merino para obtener el título de Ingeniero en Informática.

Considerando este Proyecto Fin de Carrera finalizado, autorizan su presentación y defensa.

Y para que así conste, firma en Leganés a 2 de Enero de 2007.

Fdo.: Juan Llorens Morillo

Agradecimientos

En primer lugar quisiera agradecer a mis padres Julián y Ángela, así como a mis hermanos David, Nuria y Miguel, el apoyo y la dedicación que me han ofrecido siempre durante estos veintitrés años. Gracias por todo. Este proyecto, y el paso que representa, es tanto mío como vuestro.

En segundo lugar quisiera mencionar también a todos mis amigos y colegas, especialmente a Cabezas, Rakel, Vicen y Abraham. Por todos esos buenos momentos vividos y por vivir, y por demostrarme que no hace falta mucha gente si la compañía es buena. No me olvido de la gente de la Universidad, todos esos ratos pasados peleando codo con codo no se olvidan tan fácilmente. Sois muchos, pero gracias especialmente a Luis y Julio.

Gracias también a todas aquellas personas que, directa o indirectamente, han tenido que ver en mi formación educativa. En especial quiero dar las gracias a Félix García y a Alejandro Calderón, del Departamento de Informática de la Universidad Carlos III de Madrid, por darme la oportunidad de trabajar con ellos y por enseñarme la otra cara de la Universidad que tanto me ha gustado.

Muchas gracias a Juan Llorens, mi tutor de proyecto, por las oportunidades brindadas. Por dejarme unir en este trabajo la música y la informática, las dos pasiones de mi vida. Gracias por Finlandia y por lo que viene ahora en Estados Unidos.

Gracias también a toda la gente con la que he pasado los cinco mejores meses de mi vida, como estudiante Erasmus. Víctor, Parti, Pedro, Meji, Julio, Daniel, Seweryn y otros tantos que me dejo en el tintero.

En general, gracias a todos los que me habéis apoyado todo este tiempo porque gracias a vosotros he conseguido lo que me proponía. Y sobretodo gracias a los que no me habéis apoyado, porque habéis conseguido que descubra y logre cosas que ni siquiera pude un día imaginar.

"Without music, life would be a mistake"
Friedrich Nietzsche

This page is intentionally left blank.

Document Status Sheet

This Document Status Sheet (DSS) provides a history of issues and revisions of the document with a comment indicating the reason for the revision.

Modeling and Indexing Musical Files to allow Music Reuse			
Report			
Issue	Revision	Date	Reason for change
1	0	2 Jan 2007	First version of the document

Table 1.1 Document Status Sheet

Table of Contents

Abstract	iii
Agradecimientos	v
Document Status Sheet	viii
Table of Contents	ix
List of Tables.....	xv
List of Figures.....	xvi
Code Listings	xx

Part I: Prologue

1 Purpose of the Document	2
2 Acronyms and Abbreviations	3
2.1 Acronyms.....	3
2.2 Abbreviations.....	3
3 References.....	5
3.1 Main References	5
3.2 Additional References	7
4 Overview of the Document	9

Part II: Musical Theory

1 Musical Notation.....	12
1.1 Origins.....	12
1.2 Enhancement Process.....	13
2 Terminology	14
2.1 Vertical.....	14
2.1.1 The Pentagram	14
2.1.2 Clefs.....	14
2.1.3 Notes.....	14
2.1.4 The Great Pentagram	15
2.2 Horizontal.....	15
2.2.1 Figures	15
2.2.2 Rests or Pauses	16
2.2.3 Note and Figure Writing	16
3 The Bar	18
3.1 Barlines.....	18
3.1.1 Single Barline	18
3.1.2 Double Barline.....	18
3.2 Bar Beats.....	18
3.2.1 The Time Signature.....	19
3.3 Simple and Compound Bars	19
3.3.1 Simple Bar.....	19
3.3.2 Compound Bars	20
3.3.3 Bar Relationship.....	21
3.4 Bar Parts	22
3.5 The Fermata Symbol	22
3.6 Bars in Silence	23
3.7 Incomplete Bars.....	23
3.8 The Tempo.....	23
4 Ties and Rhythm Dots	24
4.1 The Tie	24

4.2	The Rhythm Dot.....	24
4.2.1	Single Rhythm Dot	24
4.2.2	Double Rhythm Dot	25
4.3	Writing Rules	25
5	Alterations.....	27
5.1	The Tuplet	27
5.2	Accidentals	27
5.2.2	Enharmonic Notes.....	28
6	Replay Symbols	30
6.1	Replay Barlines.....	30
6.1.2	Iteration Labels	30
6.2	Navigation Marks.....	31
6.2.1	Dal Segno and Coda.....	31
6.2.2	Da Capo.....	32
7	Tonality	33
7.1	The Base Model	33
7.1.1	Major Scale.....	33
7.2	The Key Signature	34
7.2.1	Cycle of Fifths	34
7.2.2	Cycle of Fourths.....	35
8	Intervals.....	36
8.2	Interval Classification.....	36
9	The Major Mode	38
9.1	Triad Chords.....	38
9.2	Seventh Chords	39
9.3	Other Chords	39

Part III: The MIDI Specification

1	Introduction to MIDI	41
2	Messages	42
3	Voice Messages	44
3.1	Note Off.....	44
3.2	Note On	44
3.3	Aftertouch	45
3.4	Controller	45
3.5	Program Change	46
3.6	Channel Pressure	46
3.7	Pitch Wheel.....	46
4	Other Messages	48

Part IV: Standard MIDI Files 1.0

1	Introduction	50
2	File Block Structure	51
3	Header Chunk.....	52
3.2	Format.....	52
3.3	Number of Tracks.....	53
3.4	Division.....	53
4	Track Chunk.....	54
4.2	Metaevents	54

Part V: The RSHP Model and the CAKE Engine

1	Artifacts Classification and Retrieval	56
----------	---	-----------

1.1	Artifact Information Representation Model r_a	56
1.2	Artifact Indexing Process $I(a)$	56
1.3	Classification Process $C(i_a)$	57
1.4	Artifact Retrieval Process $R(i_q)$	57
2	The RSHP Information Representation Model	58
2.1	Motivations Behind RSHP	58
2.2	Inside RSHP	58
3	The RSHP Metamodel	60
3.2	Artifact	60
3.3	Term	61
3.4	Relationship	62
3.4.1	RSHPSemantics	62
3.5	Information Element	63
3.6	Property	63
4	The CAKE Engine	64
5	XMI Indexing	65
5.1	XMI Parser	65
5.2	Information Storage in Memory	65
5.3	Information Storage in a Database	65
6	XMI Retrieval	66
6.1	UML Query Creation	66
6.2	UML Query Formulation and Resolution	66
6.2.1	Query Inclusion	67
6.2.2	Query Similarity	67
6.3	Topology Measurements	67
6.4	Semantics Measurement	69

Part VI: Definition of the User Requirements

1	Introduction	74
1.1	Purpose	74
1.2	Scope	74
2	General Description	75
2.1	System Perspective	75
2.2	User Characteristics	75
3	General Requirements	76
3.1	CAKE Studio 3.0.0	76
3.2	File Format	76
3.3	Vertical Constraints	76
3.3.1	Octave Equivalence	76
3.3.2	Grade Equality	77
3.3.3	Note Equality	77
3.3.4	Chord Recognition	78
3.4	Horizontal Constraints	78
3.4.1	Time Signature Equivalence	78
3.4.2	Tempo Equality	79
3.4.3	Figure Equality	79
3.4.4	Partial Similarity	79
3.4.5	Time Quantization	80
3.5	Voice Constraints	80

Part VII: General Requirements Analysis and First Solutions

1	Introduction	83
2	File Format	84
2.1	The MIDI lib 2.0.4	85

2.2	SMF Format	87
3	Vertical Constraints.....	88
3.1	Octave Equivalence	88
3.2	Grade Equality	88
3.3	Note Equality.....	89
3.4	Chord Recognition	89
4	Horizontal Constraints	90
4.1	Time Signature Equivalence	90
4.2	Tempo Equality	90
4.3	Figure Equality	91
4.4	Partial Similarity.....	91
5	Voice Constraints.....	93
5.1	Approaches to the Voice Separation.....	93
5.1.1	Split Point Approach.....	93
5.1.2	Rule-based Approach	93
5.1.3	Local Optimization Approach	93
5.1.4	Contig Mapping Approach.....	94
5.1.5	Predicate Approach.....	94
5.2	The Kilian-Hoos Algorithm	94
5.2.1	Preliminaries.....	94
5.2.2	Input Splitting	95
5.2.3	The Cost Function.....	97
	Pitch Distance Penalty C_{pitch}	97
	Gap Distance Penalty C_{gap}	99
	Chord Distance Penalty C_{chord}	100
	Overlap Distance Penalty $C_{overlap}$	101
5.2.4	Cost-Optimized Slice Separation	102

Part VIII: The Mathematical Approach

1	Introduction	105
2	Preliminaries	106
2.1	Domain Normalization	106
2.2	Music As a Mathematical Function	107
3	Comparing Musical Pieces Described as Mathematical Functions	108
3.2	Vertical Comparison.....	109
3.3	Horizontal Comparison.....	110
3.4	Piecewise Comparison	111
3.5	How to Perform the Actual Comparison	112
4	Basic Polynomial Interpolation	113
4.1	The Runge's Phenomenon	113
5	Spline Interpolation.....	115
5.1	Cubic Splines	115
5.2	Choosing a Cubic Spline	116
6	Coping with Chords	118
7	Parametric Curves	119
8	Bézier Curves	120
8.1	Definition	120
8.2	Properties.....	121
9	B-Splines	123
9.1	Definition	123
9.2	Properties.....	125
10	Uniform B-Splines	128
10.1	The Blending Function	128
11	Why Degree 3 Uniform B-Splines	132
11.2	The Knot Spans.....	133
12	Coping with Voices.....	136

Part IX: Translation to the RSHP Metamodel and the CAKE Engine

1	Artifacts' Topology.....	139
2	Single Voices without Chords.....	140
2.1	Final Method to Compare Intervals	140
2.2	How to Represent Information Units	141
3	Single Voices with Chords	142
4	Several Voices without Chords	143
5	Several Voices with Chords	145
6	Representing Numbers with Terms.....	147
6.1	Score Durations	147
6.2	Derivative Values	148
6.2.1	Derivative Values with Voices.....	149
7	Extending the CAKE Engine	151

Part X: Implementation Details

1	The Music Information Retrieval Process	153
2	Implementation in MIKE	155
2.1	Preprocessing	155
2.2	VoiceSeparation	155
2.3	Quantization.....	156
2.4	Minimization.....	157
2.5	Interpolation	158
2.6	RSHP Modeling	158
2.7	The CAKE Studio Manager	159

Part XI: Epilogue

1	Conclusions.....	161
2	Future Work.....	162
3	Project Budget.....	164

Part XII: Source Code

1	MIKE.....	168
1.1	MikeManagerFactory	168
1.2	MIKEIndexerCreator	168
1.3	MIKEManager	168
1.4	MIKEManagerMainForm	173
2	MIKE.Preprocessing	174
2.1	IPreprocessor	174
2.2	PreprocessedMidiSequence	174
2.3	PreprocessedMidiTrack	174
2.4	PreprocessedMidiNote	175
2.5	MIKE.Preprocessing.Silly.....	176
2.5.1	SillyPreprocessor	176
3	MIKE.VoiceSeparation	178
3.1	IVoiceSeparator	178
3.2	SeparatedSequence	178
3.3	SeparatedTrack	178
3.4	SeparatedVoice	179
3.5	SeparatedNote	179
3.6	SeparatedSingleNote	180

3.7	SeparatedChord	180
3.8	MIKE.VoiceSeparation.KilianHoos	181
3.8.1	KilianHoosVoiceSeparator	181
3.8.2	KilianHoosNote	187
3.8.3	KilianHoosSingleNote	188
3.8.4	KilianHoosChord	188
3.9	MIKE.VoiceSeparation.Silly	189
3.9.1	SillyVoiceSeparator	189
4	MIKE.Quantization	190
4.1	IQuantizator	190
4.2	QuantizedSequence	190
4.3	QuantizedStaff	190
4.4	QuantizedVoice	191
4.5	QuantizedDurations	191
4.6	QuantizedNote	192
4.7	QuantizedSingleNote	192
4.8	QuantizedChord	192
4.9	MIKE.Quantization.Silly	193
4.9.1	SillyQuantizator	193
5	MIKE.Interpolation	194
5.1	IInterpolator	194
5.2	InterpolatedSequence	194
5.3	InterpolatedStaff	194
5.4	InterpolatedVoice	195
5.5	InterpolatedSpan	196
5.6	Polynomial	197
5.7	BSplineInterpolator	199
5.8	UniformBSpline	201
6	MIKE.RSHPzation	202
6.1	IRSHPzator	202
6.2	RSHPzator	202

Part XIII: Resumen en Español

1	Notación Musical, MIDI y SMF	207
1.1	Notación Musical	207
1.2	El Estándar MIDI	208
1.3	Standard MIDI Files	208
2	El Modelo RSHP y el CAKE Engine	209
2.2	El CAKE Engine	210
3	Requisitos Generales	211
3.1	Restricciones Verticales	211
3.2	Restricciones Horizontales	212
3.3	Separación de Voces	212
4	El Modelo Matemático	214
4.1	Normalización de Dominios	214

List of Tables

Table 1.1 Document Status Sheet.....	viii
--------------------------------------	------

Part II: Musical Theory

Table 2.1 Note names.....	15
Table 2.2 Note figures and durations.....	16
Table 2.3 Rest figures and durations	16
Table 3.1 Bar relationship.....	22
Table 5.1 The accidentals.....	27
Table 8.1 Intervals in the major scale	36
Table 8.2 Relationships among intervals.....	37
Table 9.1 Relationships among intervals.....	38
Table 9.2 Seventh chords	39

Part III: The MIDI Specification

Table 2.1 MIDI message types	42
------------------------------------	----

Part VIII: The Mathematical Approach

Table 9.1 Degree 0 B-Spline basis functions.....	123
Table 9.2 Degree 1 B-Spline basis functions.....	124
Table 9.3 Degree 2 B-Spline basis functions.....	124

Part IX: Translation to the RSHP Metamodel and the CAKE Engine

Table 6.1 Score durations for notes	147
---	-----

Part XI: Epilogue

Table 3.1 Human effort cost estimation.....	165
Table 3.2 Stuff and documentation cost.....	165
Table 3.3 Erasmus grant estimated cost	166
Table 3.4 Final cost calculation	166

Part XIII: Resumen en Español

Table 1.1 Notas musicales	207
---------------------------------	-----

List of Figures

Part II: Musical Theory

Figure 1.1 Fragment of the Seikilos Epitaph	12
Figure 1.2 Fragment of the Musica Disciplina	12
Figure 2.1 The pentagram	14
Figure 2.2 The Sol clef	14
Figure 2.3 The Fa clef	14
Figure 2.4 Note writing	14
Figure 2.5 Notes with additional lines	15
Figure 2.6 The seven notes in the great pentagram	15
Figure 2.7 Stem and flag writing	17
Figure 2.8 Beamed notes	17
Figure 3.1 Barlines	18
Figure 3.2 The ending double barline	18
Figure 3.3 The dividing double bar	18
Figure 3.4 The most common time signatures	19
Figure 3.5 2/4 bar composition	20
Figure 3.6 3/4 bar composition	20
Figure 3.7 4/4 bar composition	20
Figure 3.8 2/2 bar composition	20
Figure 3.9 6/8 bar composition	21
Figure 3.10 9/8 bar composition	21
Figure 3.11 12/8 bar composition	21
Figure 3.12 6/4 bar composition	21
Figure 3.13 The fermata symbol	22
Figure 3.14 Bars in silence	23
Figure 3.15 Incomplete starting bars	23
Figure 3.16 The tempo	23
Figure 4.1 Tied notes (part I)	24
Figure 4.2 Tied notes (part II)	24
Figure 4.3 The rhythm dot	24
Figure 4.4 The double rhythm dot	25
Figure 4.5 First writing rule	25
Figure 4.6 First exception to the first writing rule	25
Figure 4.7 Second exception to the first writing rule	25
Figure 4.8 Third exception to the first writing rule	26
Figure 4.9 Fourth exception to the first writing rule	26
Figure 4.10 Second writing rule	26
Figure 5.1 Triplet equivalences	27
Figure 5.2 Triplet notes	27
Figure 5.3 The accidentals	28
Figure 5.4 Accidental effect within a bar (part I)	28
Figure 5.5 Accidental effect within a bar (part II)	28
Figure 5.6 Accidental effect within a bar (part III)	28
Figure 5.7 Enharmonic notes	29
Figure 6.1 Repeat barlines (part I)	30
Figure 6.2 Repeat barlines (part II)	30
Figure 6.3 Iteration labels	30
Figure 6.4 The Dal Segno figure	31
Figure 6.5 The Coda figure	31
Figure 6.6 Navigation markers	31
Figure 7.1 Major scale in DO	33
Figure 7.2 Major scale in MI	34
Figure 7.3 Key signature for MI	34
Figure 7.4 Cycle of fifths	34
Figure 7.5 Key signatures (part I)	35
Figure 7.6 Key signatures (part II)	35
Figure 8.1 Intervals among the natural notes	36

Figure 8.2 Melodic and harmonic intervals.....	36
Figure 8.3 Relationships among intervals.....	37
Figure 9.1 Diatonic and chromatic notes.....	38
Figure 9.2 Triad chords.....	38
Figure 9.3 Triad cords in different tonalities.....	39

Part III: The MIDI Specification

Figure 3.1 Pitch Wheel value obtaining.....	47
---	----

Part IV: Standard MIDI Files 1.0

Figure 2.1 SMF chunk format.....	51
Figure 2.2 SMF block structure.....	51
Figure 3.1 Header chunk syntax.....	52
Figure 4.1 Track chunk syntax.....	54

Part V: The RSHP Model and the CAKE Engine

Figure 1.1 Graphical representation of artifacts.....	56
Figure 1.2 Artifact indexing.....	57
Figure 1.3 Artifact classification.....	57
Figure 1.4 Artifact retrieval process.....	57
Figure 3.1 RSHP metamodel.....	60
Figure 3.2 Artifact contents (part I).....	61
Figure 3.3 Artifact contents (part II).....	61
Figure 3.4 Relationships structure and semantics.....	62
Figure 3.5 Information Elements.....	63
Figure 3.6 Properties structure.....	63
Figure 6.1 A target document to retrieve.....	66
Figure 6.2 Vector space model for topology measurement.....	68
Figure 6.3 RSHP difference map.....	70
Figure 6.4 Propagation in the ISO2788 net.....	72

Part VI: Definition of the User Requirements

Figure 3.1 Octave equivalence.....	76
Figure 3.2 Grade equality (part I).....	77
Figure 3.3 Grade equality (part II).....	77
Figure 3.4 Note equality.....	77
Figure 3.5 Time signature equivalence (part I).....	78
Figure 3.6 Time signature equivalence (part II).....	78
Figure 3.7 Tempo equality (part I).....	79
Figure 3.8 Tempo equality (part II).....	79
Figure 3.9 Figure equality.....	79
Figure 3.10 Partial similarity (part I).....	80
Figure 3.11 Partial similarity (part II).....	80
Figure 3.12 Simple voice distinction.....	80
Figure 3.13 Complex voice distinction.....	81

Part VII: General Requirements Analysis and First Solutions

Figure 2.1 Music XML example.....	85
Figure 2.2 MIDI lib static information model (part I).....	86
Figure 2.3 MIDI lib static information model (part II).....	86
Figure 2.4 MIDI lib static information model (part III).....	87
Figure 2.5 MIDI lib static information model (part IV).....	87
Figure 3.1 Grade equality.....	88

Figure 3.2 Interval measurement	89
Figure 4.1 Tempo Equality	90
Figure 4.2 Timeless model (part I)	91
Figure 4.3 Timeless model (part II).....	91
Figure 5.1 Partitioning a piece into slices	95

Part VIII: The Mathematical Approach

Figure 2.1 Note distribution (part I).....	106
Figure 2.2 Note distribution (part II).....	106
Figure 2.3 Note distribution (part III)	107
Figure 3.1 Comparison among musical functions.....	108
Figure 3.2 Comparison among first derivatives.....	109
Figure 3.3 Differences in the time-dimension (part I).....	110
Figure 3.4 Piecewise comparison	111
Figure 3.5 Comparing areas.....	112
Figure 4.1 The Runge's Phenomenon (part I)	114
Figure 4.2 The Runge's Phenomenon (part II)	114
Figure 5.1 Cubic splines	115
Figure 6.1 Paths through chords	118
Figure 7.1 Parametric curves.....	119
Figure 8.1 Bézier curves.....	120
Figure 8.2 Bézier basis functions	121
Figure 8.3 The convex hull property	121
Figure 8.4 The variation diminishing property.....	122
Figure 9.1 Degree 0 B-Spline basis functions	124
Figure 9.2 Degree 1 B-Spline basis functions	124
Figure 9.3 Degree 2 B-Spline basis functions	125
Figure 9.4 B-Spline curves	125
Figure 9.5 Strong convex hull property	126
Figure 9.6 Local modification scheme	126
Figure 10.1 Degree 1 blending function integration intervals	128
Figure 10.2 Degree 1 blending function	129
Figure 10.3 Degree 2 blending function integration intervals	129
Figure 10.4 Degree 2 blending function	130
Figure 10.5 Degree 3 uniform B-Spline basis functions	131
Figure 11.1 Moving control points	133
Figure 11.2 Non-uniform B-Spline spans.....	134
Figure 12.1 3-dimensional interpolating curve	137

Part IX: Translation to the RSHP Metamodel and the CAKE Engine

Figure 1.1 Main artifacts' topology (part I)	139
Figure 1.2 Main artifacts' topology (part II)	139
Figure 2.1 Comparing intervals	140
Figure 2.2 Degree 2 polynomial shapes	141
Figure 2.3 A Concave RSHP	141
Figure 2.4 Sequence with a single voice.....	141
Figure 3.1 Voice with chords (part I).....	142
Figure 3.2 Voice with chords (part II)	142
Figure 3.3 The Span artifact.....	142
Figure 4.1 Several voices without chords (part I)	143
Figure 4.2 Several voice without chords (part II)	143
Figure 4.3 Derivatives with several voices	144
Figure 5.1 Several voices with chords (part I)	145
Figure 5.2 Several voices with chords (part II)	145
Figure 5.3 Several voices with chords (part III).....	146
Figure 5.4 Several voices with chords (part IV).....	146
Figure 6.1 First derivative intervals	149

Part X: Implementation Details

Figure 1.1 The music information retrieval process	153
Figure 2.1 Preprocessed model	155
Figure 2.2 Separated model	156
Figure 2.3 Quantized model	157
Figure 2.4 Interpolated model	158

Part XI: Epilogue

Figure 2.1 Surjective mapping to RHSP	163
---	-----

Part XIII: Resumen en Español

Figure 1.1 El gran pentagrama	207
Figure 1.2 Escala mayor de Do	208
Figure 2.1 El metamodelo RSHP.....	209
Figure 3.1 Equivalencia de octava	211
Figure 3.2 Igualdad de grados	211
Figure 3.3 Igualdad de notas	211
Figure 3.4 Equivalencia de clave de tiempo	212
Figure 3.5 Igualdad de tempo	212
Figure 3.6 Igualdad de figuras.....	212
Figure 3.7 Separación de voces	213
Figure 4.1 Normalización de dominios (parte I)	214
Figure 4.2 Normalización de dominios (parte II)	214
Figure 4.3 Interpolación de canciones	215

Code Listings

Part VII: General Requirements Analysis and First Solutions

Listing 2.1	MusicXML example	85
Listing 5.1	Outline of the Kilian-Hoos algorithm for unquantized input data	97
Listing 5.2	Pitch calculation for voice v with pitchlookback > 0	98
Listing 5.3	Calculation of C_{pitch} for a single voice v in S_i	99
Listing 5.4	Calculation of pitch distance penalty C_{pitch} for slice S_i given previous separation S	99
Listing 5.5	Calculation of gap distance penalty C_{gap} for slice S_i given previous separations S	99
Listing 5.6	Calculation of chord distance penalty C_{chord} for slice S_i	100
Listing 5.7	Calculation of overlap distance penalty $C_{overlap}$ for slice S_i given previous separations S	101
Listing 5.8	Calculation of overlap distance penalty for a single voice	102
Listing 5.9	Randomized iterative algorithm for finding a cost-optimized separation of slice y_i	103

Part I: Prologue

1 Purpose of the Document

The aim of this document is the Final Degree Project whose development is established in the Study Plan for Computing Engineering according to the B.O.E. 07.11.00 as a mandatory requirement, once its defense is accomplished, to have the right to the Computing Engineer Title.

2 Acronyms and Abbreviations

This section provides the definitions of all terms, acronyms and abbreviations, or refers to other documents where the definitions can be found.

2.1 Acronyms

AAAI	American Association for Artificial Intelligence
ASCII	American Standard Code for Information Interchange
BLAST	Basic Local Search Alignment Tool
B.O.E.	From the Spanish “Boletín Oficial del Estado”, Official State Bulletin
BSSC	Board for Software Standardization and Control
CAKE	Computer-Aided Knowledge Engineering
DIN	From the German “Deutsches Institut für Normung”, German Institute for Standardization
DLL	Dynamic Link Library
DNA	Deoxyribonucleic Acid
DOM	Document Object Model
DSS	Document Status Sheet
ESA	European Space Agency
IE	Information Element
IEC	International Electrotechnical Commission
ISMIR	International Symposium on Music Information Retrieval
ISO	International Organization for Standardization
LFO	Low Frequency Oscillation
MIDI	Musical Instrument Digital Interface
MIKE	Music Indexer based on the CAKE Engine
MMA	MIDI Manufacturers Association
MPEG	Moving Picture Experts Group
SMDL	Standard Music Description Language
SMF	Standard MIDI File
SMPTE	Society of Motion Picture and Television Engineers
SQL	Structured Query Language
UML	Unified Modeling Language
UR	User Requirement
VCA	Voltage-Controlled Amplifier
VCF	Voltage-Controlled Filter
WMA	Windows Media Audio
XMI	XML Metadata Interchange
XML	Extensible Markup Language

2.2 Abbreviations

Contig	Contiguous
Hex	Hexadecimal
LXor	Logical XOR

MP3	MPEG-1 Audio Layer 3
Pty	Property
QbyE	Query by Example
RHSP	Relationship
VAT	Value Added Tax
WAV	Waveform audio format
XOR	Exclusive Or

3 References

This section provides a complete list of all the applicable and reference documents, identified by title, author and date if applicable.

3.1 Main References

The list bellow contains all the main references used as main information sources for the current project:

- [Herrera, 1995a] Herrera E., *Teoría Musical y Armonía Moderna vols. I and II*, Antoni Bosch Editor, 1995.
- [Stone] Stone, J. E., *Music, MIDI and Synthesizers*, University of Illinois at Urbana-Champaign, <http://jedi.ks.uiuc.edu/~johns/links/music/>.
- [Borj] *The MIDI Specification*
<http://www.borg.com/~jglatt/tech/midispec.htm>.
- [Knott, 2000] Knott G. D., *Interpolating Cubic Splines*, Birkhäuser, 2000.
- [de Boor, 1978] de Boor C., *A Practical Guide to Splines*, Springer-Verlag, 1978.
- [Llorens, 2003] Llorens J., Morato J. and Génova G., *RSHP: an Information Representation Model based on Relationships*, Special Book on Soft Computing, Berlin, 2003.
- [Llorens, 2002] Llorens J., Fuentes J. M. and Morato J., *A Retrieval Framework for XMI Information*.
- [Kilian, 2004] Kilian J., *Inferring Score Level Musical Information from Low Level Musical Data*, PhD thesis, 2004.
- [ISO, 1986] International Organization for Standardization, *Guidelines for the Establishment and Development of Monolingual Thesauri*, 2nd edition, 11-15 UDC 025.48.ISO2788, Geneva, 1986.
- [Byrd, 2001] Byrd D. and Crawford T., *Problems of Music Information Retrieval in the Real World*, Information Processing and Management.
- [Sharka, 2004] Sharka I., Frederico G. and El Saddik A., *Music Indexing and Retrieval*, Proceedings of the IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems, Boston, MD, USA, 2004.

- [Doraisamy, 2001] Doraisamy S. and Rüger S. M., *An Approach Towards a Polyphonic Music Retrieval System*, Proceedings of the 2nd International Symposium on Music Information Retrieval (ISMIR 2001), pp. 187-193, Indiana University, Bloomington, USA, 2001.
- [Doraisamy, 2004] Doraisamy S. and Rüger S. M., *A Polyphonic Music Retrieval System Using N-Grams*, Proceedings of the 5th International Symposium on Music Information Retrieval (ISMIR 2004), Audiovisual Institute-Universitat Pompeu Fabra, Barcelona, Spain, 2004.
- [Hoos, 2001] Hoos H. H., Renz K. and Görg M., *GUIDO/MIR - An Experimental Musical Information Retrieval System based on GUIDO Music Notation*, Proceedings of the 2nd International Symposium on Music Information Retrieval (ISMIR 2001), pp. 41-50, Indiana University, Bloomington, USA, 2001.
- [Dovey, 2001] Dovey M. J., *A technique for "regular expression" style searching in polyphonic music*, Proceedings of the 2nd International Symposium on Music Information Retrieval (ISMIR 2001), pp. 179-185, Indiana University, Bloomington, USA, 2001.
- [Lebel, 2006] Lebel D., *Voice Separation Summary*, Computer Music Seminar 2 (MUMT-611), Schulich School of Music, 2006.
- [Hoos, 2002] Hoos H. H. and Kilian J., *Voice Separation - A Local Optimization Approach*, Proceedings of the 3rd International Conference on Music Information Retrieval (ISMIR 2002), pp. 39-46, Institut de Recherche et Coordination Acoustique/Musique (IRCAM), Paris, France, 2002.
- [Kirlin, 2005] Kirlin P. B. and Utgoff P. E., *VoiSe: Learning to segregate voices in explicit and implicit polyphony*, Proceedings of the Sixth International Conference on Music Information Retrieval, pp. 552-557, Queen Mary, University of London, 2005.
- [Chew, 2005] Chew E. and Wu X., *Separating Voices in Polyphonic Music: A Contig Mapping Approach*, Proceedings of the International Symposium on Computer Music Modeling and Retrieval, pp. 1-20, Berlin, Germany, 2005.
- [Cambouropoulos, 2000] Cambouropoulos E., *From MIDI to Traditional Musical Notation*, Proceedings of the AAAI Workshop on Artificial Intelligence and Music, Austin, Texas, USA, 2000.
- [Camboutopoulos] Cambouropoulos E., Crochemore M., Iliopoulos C., Mohamed M. and Sagot MF., *A Pattern Extraction Algorithm for Abstract Melodic Representations that Allow Partial Overlapping of Intervallic Categories*,
- [Paiement] Paiement JF., Eck D. and Bengio S., *A Probabilistic Model for Chord Progressions*, Proceedings of the International Symposium on Computer Music Modeling and Retrieval

- [Orio] Orio N. and Neve G., *Experiments on Segmentation Techniques for Music Documents Indexing*, Proceedings of the International Symposium on Computer Music Modeling and Retrieval
- [Yu] Yu Y., Watanabe C. and Joe K., *Towards a Fast and Efficient Match Algorithm for Content-based Music Retrieval on Acoustic Data*, Proceedings of the International Symposium on Computer Music Modeling and Retrieval
- [Nagel, 2005] Nagel C., Evjen B. et al., *Professional C# 2005*, Wrox Press, 2005.
- [Barwell, 2002] Barwell F., Case R. et al., *Professional VB .net 2nd edition*, Wrox Press, 2002.
- ESA software engineering standards.

3.2 Additional References

The list below contains all the additional references where some relevant information can be found about several topics treated in the document:

- [Wikipedia, EN] *The Free Encyclopedia*, <http://en.wikipedia.org>.
- [DIN] *German Institute for Standardization* <http://www.din.de>.
- [SMPTE] *Society of Motion Picture and Television Engineers* <http://www.smpte.org>.
- [MMA] *MIDI Manufacturers Association* <http://www.midi.org>.
- [Rona, 1987] Rona J., *The MIDI Companion: The Ins, Outs and Throughs*, Hal Leonard Publishing Corporation, 1987.
- [Baeza, 1999] Baeza R. and Ribeiro B., *Modern Information Retrieval*, Addison Wesley, 1999.
- [OMG, a] Object Management Group, *Unified Modeling Language (UML)*, <http://www.uml.org/>.
- [OMG, b] Object Management Group, *XML and XMI Resource Page*, <http://www.omg.org/technology/xml/>.
- [W3C] World Wide Web Consortium, *Extensible Markup Language (XML)* <http://www.w3.org/XML/>.
- [Nano] *NanoSounds*, <http://nanosounds.tripod.com>.
- [ICOS] *La Web de los Juegos Locos*, <http://www.losicos.com>.

- [Microsoft] Microsoft, *Microsoft .net Technology*, <http://www.microsoft.com/net/>.
- [dTinf] The Reuse Company, <http://www.reusecompany.com>.
- [ISO/IEC] International Organization for Standardization and International Electrotechnical Commission, *Standard Music Description Language*, ISO/IEC DIS 10743, 1995
- [Recordare] Recordare LLC, *MusicXML Definition*, <http://www.musicxml.org/xml.html>.
- [Toub] Stephen Toub, *.NET, MSDN Magazine and other Adventures in Life*, <http://blogs.msdn.com/toub/>.
- [GotDotNet] GotDotNet, *MIDI for .NET v2.0.4*, <http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=89CDE290-5580-40BF-90D2-5754B2E8137C>.

4 Overview of the Document

The document is divided into 12 main sections that are organized in parts:

- I. Prologue. This part includes a list of acronyms and abbreviations used along the document, as well as the applicable bibliography.
- II. Musical Theory. This section explains in detail the main music theory aspects that are going to be considered for the system.
- III. The MIDI Specification. This time, the MIDI standard is documented so that its main capabilities and features are explained.
- IV. Standard MIDI Files 1.0. This part of the document explains how MIDI files are, their structure and their relationship with the MIDI standard.
- V. The RSHP Model and the CAKE Engine. In this part the RSHP information representation metamodel and the CAKE Engine are explained in detail since they are the pillars of the system.
- VI. Definition of the User Requirements. This section contains the user requirements explained in detail.
- VII. General Requirements Analysis and First Solutions. After section VI ends, this one directly faces all the constraints stated and offers a detailed analysis of them so that a first approximation to their solution is offered.
- VIII. The Mathematical Approach. The basis of MIKE is a strong mathematical model build upon the interpolation bases. This model is explained and detailed in this part of the document with all the details about its improvement.
- IX. Translation to the RSHP Metamodel and the CAKE Engine. Since the project must comply with this metamodel and the engine explained in the Part V, this one is in charge of explaining how to translate to RSHP the mathematical model presented in Part VIII.
- X. Implementation Details. This part contains a detailed explanation about how the model is actually implemented and outlines some suitable future work.
- XI. Epilogue. This almost last part is in charge of presenting the conclusions of the project as well as defining some future work that might be applied to MIKE in order to improve it or simply make it more suitable. Indeed, we will see that some extra work is mandatory. Moreover, this part contains the budget of the project.

- XII. Source Code. This part contains the source code of the system, which is implemented under the Microsoft .net technology in the C# language.
- XIII. Resumen en Español. This is a mandatory part for the report since it is all in the English language. This part contains a brief about the whole project in the Spanish language.

Part II:

Musical Theory

1 Musical Notation

The musical notation is a system to write music [Wikipedia, EN]. Nowadays, this notation is based on a five-line staff with symbols for each note, duration, pitch, clef and whatever related to the music piece represented. However, this has not been the unique notation used along the time.

1.1 Origins

There are some evidences showing that there was any kind of music representation practiced by the Egyptians and others in the Orient in the third millennium BC.

From the sixth century BC to the fourth AC there was another notation used in the Ancient Greece and there still exist some pieces of compositions with this notation surviving nowadays. An important example of this notation is the Seikilos Epitaph which uses some symbols placed above letters. An example of these symbols is depicted below in Figure 1.1.

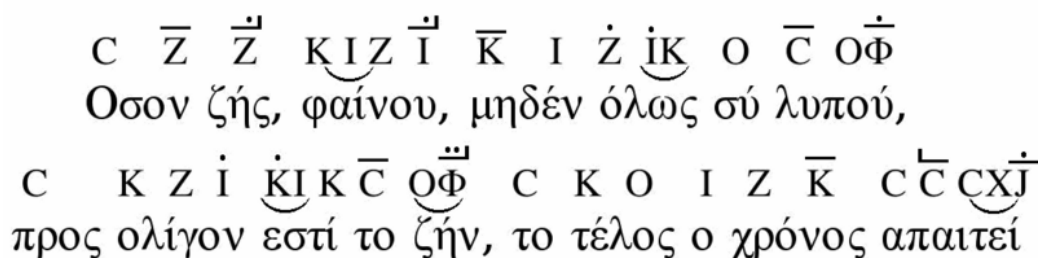


Figure 1.1 Fragment of the Seikilos Epitaph

Knowledge related to this Ancient Greek notation was lost with the fall of the Roman Empire. Then, in the seventh century music theorist Isidore of Seville pointed that it is not possible to notate music. However, in some Gregorian Monasteries, a new notation became to born in the ninth century and was characterized by its symbols, also called neumes. The earliest notation of this kind is the Musica Disciplina of Aurelian of Réôme about year 850. Figure 1.2 shows an example of this notation.

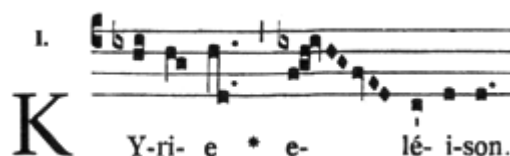


Figure 1.2 Fragment of the Musica Disciplina

There were another notations coming from the Iberian Peninsula known as Visigothic neumes, but unfortunately its few surviving fragments have not been deciphered yet.

Other notations date from China and Japan at the tenth century. In other places, such as India, music was notated by using characters for sounds. On the other hand, in Europe it was tried to create a purely symbolic notation which does not seem to exist anywhere else.

1.2 Enhancement Process

The earliest music notation was encoded using cuneiform scripts in Mesopotamia, dating at the middle of the second millennium BC. There were used several notations from then until the modern one, originated in the Catholic Church. Some of these notations used the neume system but, although this system was capable to express considerable music complexity, it did express neither the time nor the pitch. Therefore, it was impossible to interpret the piece for someone who have never listened the song.

To treat the issue of the pitch, it was introduced a staff consisting only in one single horizontal line, and it was evolving to a four-line system where each pitch had a different height in the staff. This lined system is still used with many variations for different instruments. For instance, there is a notation with six lines used to represent notes in a guitar. Nowadays, it is used a five-line staff which was first adopted in France and became widely used in the sixteenth century.

As mentioned above, the neume system was unable to express the duration of each note. Therefore, by the tenth century arose a new system for representing up to four note lengths. These lengths were relative to the neighboring notes and not absolute. However, it was not until the fourteenth century when a system like the present one became to use fixed note lengths and to split pieces into parts. That way, it was clear, for several staves, which parts must be played at the same time. Finally, nowadays regular measures took place by the seventeenth century.

Nowadays there are several musical notation systems widely used around the globe. There are some ones that use letters, numbers or both them and express the note length with a single number. Others use a wide variety of symbols and even express notes in a staff similar to the instrument with which the piece should be played.

2 Terminology

The representation of musical sounds is made with many different symbols. Some of them define the vertical concept (i.e. the height) and some others the horizontal concept (i.e. the duration).

2.1 Vertical

2.1.1 The Pentagram

The staff used nowadays consists on five horizontal and parallel lines as show in Figure 2.1. This staff is called pentagram and depending on the pitch, each note will be placed in a different height between these lines.

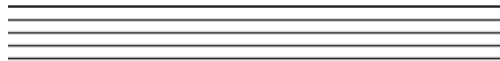


Figure 2.1 The pentagram

2.1.2 Clefs

There are some symbols used to define every note written into the pentagram. These symbols are called clefs and there are mainly two: the 'sol' and the 'fa' one in fourth, which are depicted in Figure 2.2 and Figure 2.3 respectively.



Figure 2.2 The Sol clef



Figure 2.3 The Fa clef

2.1.3 Notes

Notes are represented by symbols written over the spaces or between the pentagram lines as they appear in Figure 2.4. Thus, each musical sound is determined by a note.

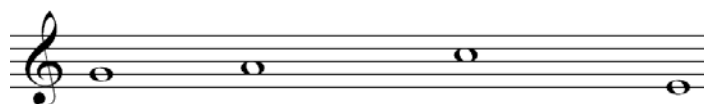


Figure 2.4 Note writing

Some musical sounds can not be represented because of their height, so some additional lines are used. These notes are written with some short and parallel lines equidistant to those of the pentagram, either above or below, as it is depicted in Figure 2.5.



Figure 2.5 Notes with additional lines

2.1.4 The Great Pentagram

Notes placed in the pentagram are seven and are called in the International and the Cipher notation as indicated in Table 2.1:

International notation	Cipher notation
DO	C
RE	D
MI	E
FA	F
SOL	G
LA	A
SI	B

Table 2.1 Note names

The great pentagram is the union of two pentagrams: the highest in Sol clef and the lowest in Fa clef. Notes are written in both pentagrams as shown in Figure 2.6:

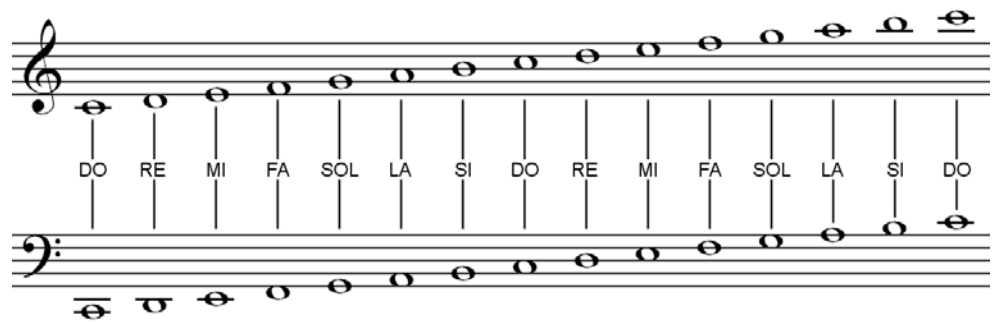


Figure 2.6 The seven notes in the great pentagram

2.2 Horizontal

2.2.1 Figures

Whatever a sound, its duration is determined by the figure of the note it is represented with. There are mainly six figures for representing sound durations: the semibreve, the minim, the crotchet, the quaver, the semiquaver and the demisemiquaver. They are shown in Table 2.2.







Name	Figure	Duration
Semibreve		Taken as unit
Minim		Half a semibreve
Crotchet		Half a minim
Quaver		Half a crotchet
Semiquaver		Half a quaver
Demisemiquaver		Half a semiquaver

Table 2.2 Note figures and durations

There are some other figures, such as the breve, the hemidemisemiquaver or the quasihemidemisemiquaver, that are not actually used either in modern or popular music.

2.2.2 Rests or Pauses

Some times, there arises the need of representing interruptions in staves. Actually, these interruptions are sound lacks, and there are also several figures to represent several rest durations, which are show in Table 2.3:







Note equivalence	Figure
Semibreve rest	
Minim rest	
Crotchet rest	
Quaver rest	
Semiquaver rest	
Demisemiquaver rest	

Table 2.3 Rest figures and durations

2.2.3 Note and Figure Writing

Notes placed over the lines must be crossed by them, and those that are paced between two lines must be touching them.

Every note but semibreves are written with a stem that must be placed on the right hand side when the stem is upwards and on the left hand side when the stem is downwards. Usually, stems are written upwards when the note is placed below the third line of the pentagram and downwards when it is placed over the third line or above.

Flags used for quavers or figures with lower duration are always at the right of the stem, at its extreme and with opposite direction. It is shown in Figure 2.7.



Figure 2.7 Stem and flag writing

When two or more notes which would normally have flags (quaver notes or shorter) appear successively, the flags may be replaced by beams, as shown below in Figure 2.8.



Figure 2.8 Beamed notes

3 The Bar

The bar is the time unit on which a musical piece is split. There are several symbols and notations related to bars that are explained from now on.

3.1 Barlines

Barlines are represented with a perpendicular line that joins the first and the fifth pentagram lines. These lines are used to represent several bar characteristics, such as the end of a piece, a repetition and so on.

3.1.1 Single Barline

Single barlines are used to show the end of a bar and the beginning of the following one, as is depicted in Figure 3.1.



Figure 3.1 Barlines

3.1.2 Double Barline

The double barline consists on two perpendicular lines, being the second one thicker than the first one. This kind of barline indicates the end of the song, as shown in Figure 3.2.



Figure 3.2 The ending double barline

There is also another kind of double barline in which both barlines have the same thickness. This barline is used to indicate the end of one part of the song and the beginning of another. It is depicted in Figure 3.3.

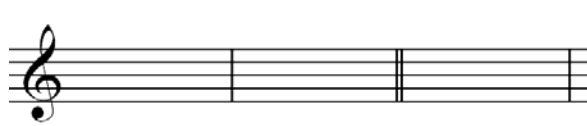


Figure 3.3 The dividing double bar

3.2 Bar Beats

Every bar is divided into periods of time of the same duration known as beats. Thus, the beat is the basic time unit of a music piece.

3.2.1 The Time Signature

A time signature consists in two numbers, one placed above the other immediately after the clef. Its goal is to define the bar. The upper number indicates how many beats a bar has, and the lower number indicates the value of each of those bars related to the unit: the semibreve. The most common time signatures are depicted in Figure 3.4.

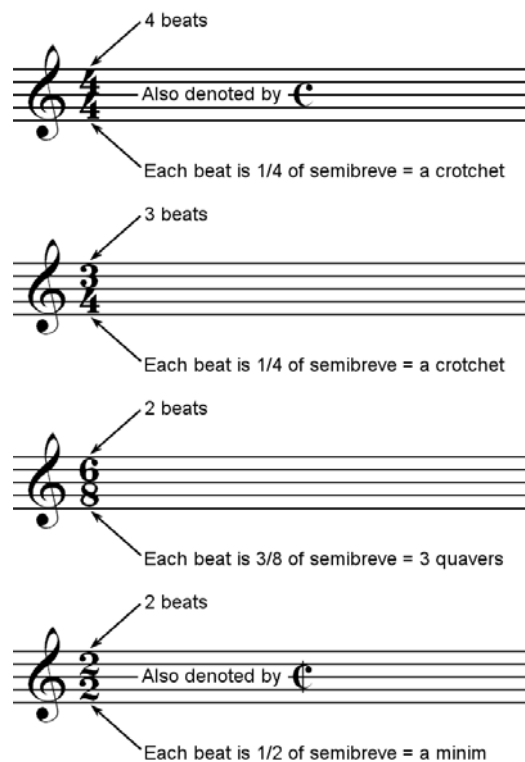


Figure 3.4 The most common time signatures

As show in the third staff, there are some issues about beat and bar composition. It is not as easy as two simple numbers, so next section will clarify it in detail.

3.3 Simple and Compound Bars

There are two kinds of bar: simple and compound. They distinguish one of each other mainly by the number of simple figures their beats are composed by. Simple bar beats are composed by one simple figure, and compound bar beats are composed by three simple figures.

3.3.1 Simple Bar

A simple bar has beats formed by one simple figure. Sometimes, it is said that the beat is actually formed by two figures, but here we will consider only one for simplicity.

The one in Figure 3.5 is a 2/4 bar. Therefore, each bar is formed by two beats, and each beat is formed by a crotchet, since it is the fourth part of a semibreve. Note that these beats can be considered as formed by two quavers.



Figure 3.5 2/4 bar composition

Figure 3.6 depicts a 3/4 bar, so that there are 3 beats per bar, each one formed by one crotchet.



Figure 3.6 3/4 bar composition

The one in Figure 3.7, a 4/4 bar, is the most commonly used nowadays. It is composed by 4 beats, each one formed by a crotchet again.



Figure 3.7 4/4 bar composition

The last one, in Figure 2.2, is a 2/2 bar, which is composed by 2 beats, each one formed by a minim. Note that bars can also be considered as formed by two crotchets.



Figure 3.8 2/2 bar composition

Comparing the first bar in Figure 3.7 and the second one in Figure 3.8, a 4/4 bar can be considered as 2/2 and vice versa. However, in Section 3.4 the difference will be seen.

3.3.2 Compound Bars

A compound bar has beats formed by three simple figures. Sometimes, it is said that the beat is actually formed by one prolonged figure; however, we will consider three simple figures for simplicity. Section 4.2 gives an explanation of what a prolonged note means.

For instance, the one in Figure 3.9 is a 6/8 bar. Each bar is actually formed by 2 beats, so let us consider the upper number as 2. Now, we can see that beats can not be divided into one or two simple figures, but into 3 quavers. Therefore, there are 3 figures whose value is the eighth part of a semibreve (a quaver).

So at the end we have 2 beats formed by 3 figures whose value is 1/8. It is actually 6/8. The figure is said to be a prolonged crotchet (one crotchet and a half).



Figure 3.9 6/8 bar composition

In a 9/8 bar, as the one in Figure 3.10, there are 3 beats, each of one composed by 3 quavers or one prolonged crotchet.

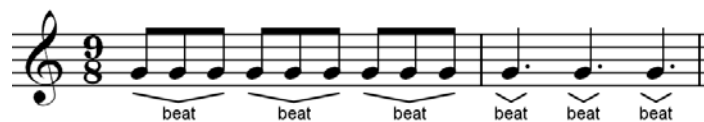


Figure 3.10 9/8 bar composition

The next one in Figure 3.11 is a 12/8 bar, which is formed by 4 beats of 3 quavers each.



Figure 3.11 12/8 bar composition

Finally, Figure 3.12 shows a 6/4 bar which composed by 3 beats, each one formed by 3 crotchets. Here, beats can also be considered as composed by a prolonged minim.



Figure 3.12 6/4 bar composition

3.3.3 Bar Relationship

Focusing on the number of beats per bar, there are mainly three categories: binary bars, ternary bars and quaternary bars. Also, there can be established a correspondence among simple and compound bars according to the number of beats as follows in Table 3.1:

Simple Bar	Compound Bar	Number of Beats
2/4	6/8	2
3/4	9/8	3
4/4	12/8	4
2/2	6/4	2

Table 3.1 Bar relationship

Although there can be defined more time signatures, these ones above are the most common.

3.4 Bar Parts

Depending on the number of beats per bar, each of them is considered as strong, semi-strong or weak.

Thus, parts of a 4 beats bar are:

1 st beat	_____	strong
2 nd beat	_____	weak
3 rd beat	_____	semi-strong
4 th beat	_____	weak

In a 3 beats bar, each one is considered as:

1 st beat	_____	strong
2 nd beat	_____	weak
3 rd beat	_____	weak

And finally, in a 2 beats bar, each part is as follows:

1 st beat	_____	strong
2 nd beat	_____	weak

3.5 The Fermata Symbol

The fermata symbol is a semicircle with a dot inside. It is usually printed above, but occasionally below (upside down), a note or rest. Its intention is to interrupt or prolongate, on purpose, the current bar in the marked note. Since it is a voluntary anomaly, it is often used only in the last note of a piece, as occurs in Figure 3.13.



Figure 3.13 The fermata symbol

3.6 Bars in Silence

Whatever the time signature, if a bar is completely in silence it is indicated only with a semibreve rest. For instance, in a 3/4 time signature like the one in Figure 3.14, a semibreve rest can not fit a bar because its length is greater, but it is printed anyway for a whole silent bar.

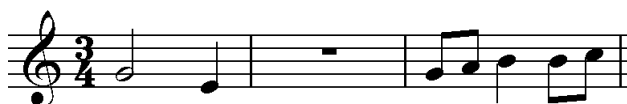


Figure 3.14 Bars in silence

3.7 Incomplete Bars

When the first bar of a piece starts with rests, they are usually omitted, so that only the first note figures appear in that first bar. A simple example is depicted in Figure 3.15.



Figure 3.15 Incomplete starting bars

3.8 The Tempo

An important symbol in every musical piece is the tempo, which is in charge of establishing how many crotchets are played in a minute. Thus, if a tempo mark establishes a tempo of 120, the duration of a crotched would be 500 milliseconds and the duration of a quaver would be 250 milliseconds and so on.

The tempo appears at the beginning of a piece, even though it might change somewhere along the piece.



Figure 3.16 The tempo

4 Ties and Rhythm Dots

These symbols are basically used to prolongate the duration of the note that they are attached to. However, it is not that easy since there are some writing rules that must be followed.

4.1 The Tie

Ties are printed as a curved line joining two or more notes with the same sound (i.e. the same height). Its purpose is to add the value of the figures it is joining. Thus, the following notes in Figure 4.1



Figure 4.1 Tied notes (part I)

are equivalent to those in Figure 4.2.



Figure 4.2 Tied notes (part II)

Note that ties are placed below the notes if their stems are written downwards and above the notes if any of them has stems upwards. Also, ties can join any figure; even though in the figures only appear tied crotchets.

4.2 The Rhythm Dot

When a dot is situated immediately after a note, its function is to prolongate the note value. There are two kinds of rhythm dot: single and double.

4.2.1 Single Rhythm Dot

A single rhythm dot prolongates the note value in a half of its original value. For instance, a crotchet with a rhythm dot is equivalent to three quavers.

Thus, both bars in Figure 4.3 are equivalent:



Figure 4.3 The rhythm dot

4.2.2 Double Rhythm Dot

A double rhythm dot prolongates the note value in a half and a quarter of its original value. For instance, a minim with a double rhythm dot is equivalent to three crotchets and a quaver.

Thus, both bars in Figure 4.4 are equivalent:



Figure 4.4 The double rhythm dot

4.3 Writing Rules

In current music and jazz, the predominant bar is 4/4, and there are some rules to follow for this kind of bar:

- No value can begin in the first half of the bar and prolongate until the second half if it is not by using the tie. Thus, in Figure 4.5 the first bar must be written as the second one:



Figure 4.5 First writing rule

However, there are four exceptions to the rule:

- A semibreve placed in the first beat, as in Figure 4.6:

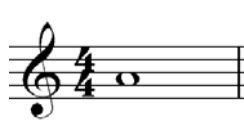


Figure 4.6 First exception to the first writing rule

- A minim with a rhythm dot placed in the first beat, as occurs in Figure 4.7:



Figure 4.7 Second exception to the first writing rule

- A minim placed in the second beat, as in Figure 4.8:



Figure 4.8 Third exception to the first writing rule

- A minim with a rhythm dot placed in the second beat, as occurs in Figure 4.9:



Figure 4.9 Fourth exception to the first writing rule

- Quavers that do not share the same beat can not be beamed. However, it can be done with four quavers sharing the first and the second beat or the third and the fourth ones, as happens in Figure 4.10.



Figure 4.10 Second writing rule

5 Alterations

There are some symbols used to alter either the vertical or the horizontal value of a figure or a set of them. This section explains these alterations.

5.1 The Tuplet

This symbol is used above or below several notes to group them. The value of the whole group without tuplet must be equal to three figures of the same duration. After having put the tuplet, the value of the group becomes as only two figures of the same class. This is depicted in Figure 5.1.



Figure 5.1 Tuplet equivalences

A tuplet is usually composed by a group of three notes, even though two notes is the minimum. Of course, each part of a tuplet can be divided as happens in Figure 5.2:



Figure 5.2 Tupled notes

5.2 Accidentals

The goal of these symbols, presented in Table 5.1, is to modify the height of the note before of which they are situated.

Name	Figure	Meaning
Sharp	#	Increase the height in a semitone
Flat	b	Decrease the pitch in a semitone
Double Sharp	×	Increase the height in a tone
Double Flat	bb	Decrease the height in a tone
Natural	q	Keep the note to its natural sound

Table 5.1 The accidentals

Sections 7 and 8 will give further information about tones and intervals.

Accidentals must be placed at the same line or space which the note is placed at and immediately before them. Some examples are shown in Figure 5.3.



Figure 5.3 The accidentals

A really important point is that an accidental affects not only to the following note, but also to any note else of the same height until the end of the bar. That is why, in Figure 5.4, the second to last note is marked with a natural accidental symbol: to avoid the sharp two notes before.

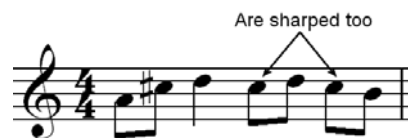


Figure 5.4 Accidental effect within a bar (part I)

Now, let us consider the bar in Figure 5.5 bar:



Figure 5.5 Accidental effect within a bar (part II)

The second note is a DO, but as it is marked with a sharp, it is actually DO#. Therefore, the fourth note is also DO#, since it has the same height and is within the same bar. However, the sixth note is a natural DO, since it has the same name but not the same height (i.e. is lower).

Even though, as said above, the sixth note is a natural DO, a natural accidental symbol is usually placed before to clarify this fact. Figure 5.6 depicts this deed.



Figure 5.6 Accidental effect within a bar (part III)

5.2.2 Enharmonic Notes

Two notes are called enharmonic when they have different names but the same sound. In Figure 5.7 below: DO# and REb, DOb and SI and also MI# and FA.

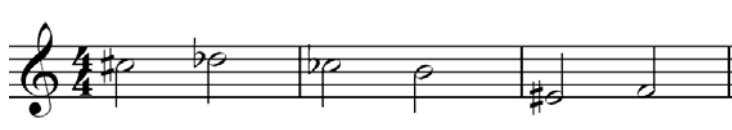


Figure 5.7 Enharmonic notes

6 Replay Symbols

In the current notation used nowadays there are some figures that can be used to mark some parts of a musical piece to be replayed in many manners. Basically, these figures are barlines and symbols placed above them.

6.1 Replay Barlines

The most common format used to indicate a repetition is to place replay barlines between the initial and the final bar to replay. These barlines are formed by two lines and a pair of dots. These two dots must be inside the replayed bars, and the outer line must be thicker than the inner one. In Figure 6.1, bars 3 and 4 must be played twice, so the final arrangement is: 1, 2, 3, 4, 3, 4 and 5.



Figure 6.1 Replay barlines (part I)

When bars to be replayed are the first ones, the starting replay barline is omitted, keeping the second one to mark the last bar to repeat. Therefore, Figure 6.2 indicates that the beginning of the piece must be repeated.



Figure 6.2 Replay barlines (part II)

6.1.2 Iteration Labels

Close to the second barline, there can be some labels marking several bars. These labels indicate that the marked bars must be played only in the iteration they indicate with a number.



Figure 6.3 Iteration labels

In Figure 6.3 above, bars 3 and 4 must be played only in the first iteration. Therefore, the final sequence is: 1, 2, 3, 4, 1, 2 and 5.

6.2 Navigation Marks

Besides the replay barlines, there also several symbols used to indicate what bars must be replayed along the piece. These symbols can, of course, be combined to indicate complex repetitions

6.2.1 Dal Segno and Coda

The Dal Segno symbol in Figure 6.4 is used for representing the beginning of a repetition. It is usually used with the Coda symbol in Figure 6.6 to go back to the beginning of the repetition from different bars.



Figure 6.4 The Dal Segno figure



Figure 6.5 The Coda figure

Although there can be used several textual combinations with Dal Segno and Coda, the most common are the following:

- D. % al Coda: indicates that the piece must be replayed from the bar marked with the Dal Segno figure and then go to the one marked with the Coda figure.
- D. % al fine: indicates that the piece must be replayed from the bar marked with the Dal Segno figure and then continue until the end of the piece.

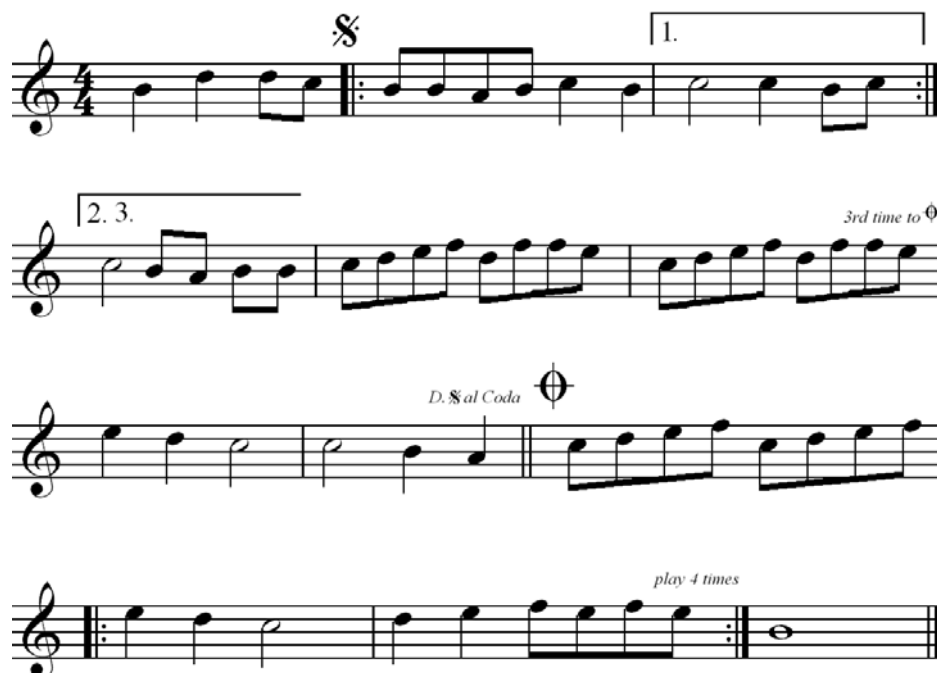


Figure 6.6 Navigation markers

In Figure 6.6 above there appear several of the previous replay figures. First of all, bar 1 is played, followed by bars 2 and 3. Since there is a replay barline, bar number 2 is played again, but not bar 3. Note that there is an iteration label. Instead of from bar 3, the music continues from bar 4. Next ones are bars 5, 6, 7 and 8. There appears a textual figure for coming back to Dal Segno and to continue from Coda. Therefore, the piece continues with bars 2, 4, 5 and 6. Here appears a textual figure indicating that in this bar the piece jumps to Coda in the third iteration of the repetition (which is actually the current one). Thus, after bar 6 the piece continues with bar 9 and later on it enters in a four iterations loop with bars 10 and 11 as is indicated with the textual figure above the end replay barline. Finally, bar 12 is played, finishing the whole piece in that point.

6.2.2 Da Capo

Sometimes, it is needed to come back to the beginning of the piece. For that purpose, it is used the Da Capo figure, which is only a textual label above the barline with letters D.C.

7 Tonality

The tonality of a musical piece defines a set of sounds whose behavior is ruled by a main sound called dominant.

A tonality is based upon seven sounds called degrees and that correspond to the seven note names. These degrees are named using roman numbers, ranging from I to VII, being the first one the dominant note.

Whatever the tonality, it can have several modes; but mainly two:

- Major mode
- Minor mode

About the degrees, they are split into two sets:

- Tonal degrees (I, IV and V) that define the actual tonality.
- Modal degrees (II, III, VI and VII) that define the mode of the tonality.

7.1 The Base Model

In the occidental music it has been taken as basis the tone in major mode. Therefore, given a tonality it should be understood that it is in its major mode if nothing opposite is specified.

The major mode is obtained by arranging degrees so that there is a semitone between the III and the IV and between the VII and the VIII (note that degree VIII is the I one an octave higher) and a tone between the rest of the consecutive degrees.

The basis model takes DO as the first degree of the tonality, and hence as the dominant note.

7.1.1 Major Scale

The major scale is divided into two halves called tetrachords that are formed by four notes each, having a semitone between the III and the IV degree of each tetrachord and being separated by a tone.

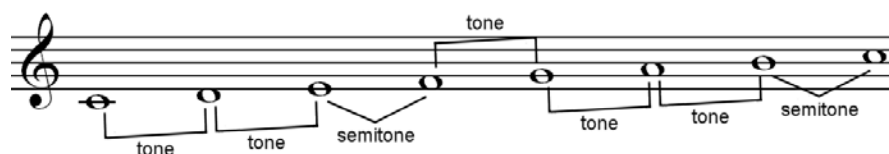


Figure 7.1 Major scale in DO

Whatever the note, it can be build a major scale from it by writing a tetrachord and from there add another tetrachord at a distance of a tone. For example, figure 7.2 shows a major scale in MI.

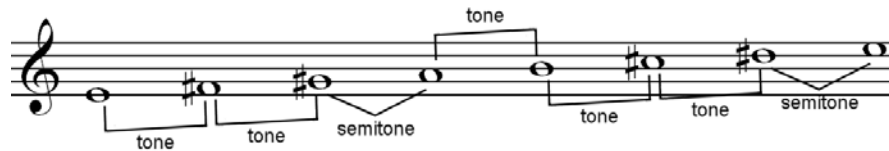


Figure 7.2 Major scale in MI

Section 8 will give more information about intervals between notes so that it will be clear why there is the need of writing accidentals in the figure above.

7.2 The Key Signature

In order to obtain a major scale from a note different than DO, there arises the need of adding accidentals to some degrees. This set of accidentals needed to build a certain major scale is called the key signature.

The key signature is placed just after the clef and its effect is continuous until the end of the piece or a new key signature is defined. For instance, Figure 7.3 shows the key signature of the tonality MI.



Figure 7.3 Key signature for MI

The placement of the accidentals that compose a key signature is determined by the cycle of fifths.

7.2.1 Cycle of Fifths

The cycle of fifths is obtained by placing notes at the same distance one to the previous until there appear the eleven possible notes, such as is done in Figure 7.4.

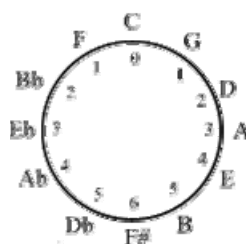


Figure 7.4 Cycle of fifths

Beginning in DO, its major scale needs no accidental, so the key signature for DO does not have any accidental. The next note in the cycle is

SOL and its major tonality needs the note FA#, so the key signature for SOL has an accidental. The third note in the cycle is RE, which needs the FA# and DO#. Thus, the key signature for RE has two accidentals. Basically, each note of the cycle will need every accidental of the previous note plus one more.

The placement of each key signature in the pentagram is as follows in Figure 7.5:



Figure 7.5 Key signatures (part I)

7.2.2 Cycle of Fourths

Taking the cycle of fifths and traversing it in reverse order it can be obtained the cycle of Fourths, which key signatures are shown in Figure 7.6. The key signature for DO remains with no accidentals. The second note, FA, needs a SI flat; and the third one, SIb, will need SIb and MIb.



Figure 7.6 Key signatures (part II)

Non-natural notes have two possible key signatures, one with sharps and another one with flats. The sum of every accidental of enharmonic tones will always be twelve, and it will be taken the one with fewer accidentals. In the case of FA# and SOLb both them has six accidentals, so either can be chosen.

8 Intervals

An interval is the height distance between two musical sounds. In occidental music, the smallest distance between two notes is the semitone. Among the natural notes, distances of a semitone can be found between MI and FA and also between SI and DO. The distance equivalent to two semitones is called tone, and is found between the rest of natural notes. Figure 8.1 shows these intervals.



Figure 8.1 Intervals among the natural notes

Intervals can be divided into melodic or harmonic, as Figure 8.2 shows:

- Melodic intervals if one sound is played after the other.
- Harmonic interval if both notes are played simultaneously.



Figure 8.2 Melodic and harmonic intervals

8.2 Interval Classification

Intervals are measured according to the number of degrees they contain, counting from the lower degree to the higher, both included. Intervals formed between the first degree and the rest ones in a major scale are enumerated in Table 8.1.

Degree	Interval	Height	Abbreviation
II	major second	1 tone	M2
III	major third	2 tones	M3
IV	perfect fourth	2 tones and 1 semitone	P4
V	perfect fifth	3 tones and 1 semitone	P5
VI	major sixth	4 tones and 1 semitone	M6
VII	major seventh	5 tones and 1 semitone	M7

Table 8.1 Intervals in the major scale

On the other hand, intervals can be measured between any two notes. Taking the lower one as if it was the first grade of the tonality and the upper note the other grade, the table below can be used to name the interval according to Table 8.1 and the possible alteration in the upper note's pitch.

-1 tone	-1 semitone	Basis interval	+1 semitone	+1 tone
diminished	minor	major	augmented	double augmented
double diminished	diminished	minor	major	augmented
double diminished	diminished	perfect	augmented	double augmented

Table 8.2 Relationships among intervals

Thus, if we consider the intervals in Figure 8.3, in the left hand side we would consider a tonality of C where the upper note is F, so that the interval is a perfect fourth. On the right hand side, the tonality would be B, and the upper is F so that it would be the fifth grade (a perfect fifth interval) with one semitone less. Therefore, this interval would be a diminished fifth.

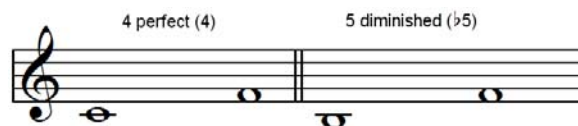


Figure 8.3 Relationships among intervals

9 The Major Mode

According to the major tonality, a major mode is made up with the seven notes of the tonality which are called diatonic notes (with the corresponding alterations depending on the tonality). Moreover, there are another five sounds that correspond to the ten chromatic notes due to the enharmony. For instance, the following Figure depicts the notes that make up the major mode for the tonality of D.



Figure 9.1 Diatonic and chromatic notes

9.1 Triad Chords

When three consecutive diatonic notes of a mode are played simultaneously by means of two harmonic intervals, the whole is called a triad chord. Thus, the seven triad chords in the major mode of D are:

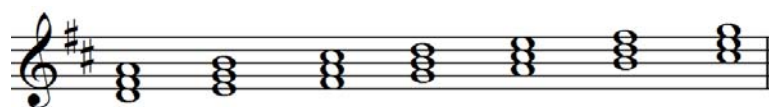


Figure 9.2 Triad chords

Each of these chords is formed starting at a certain grade of the scale. In the tonality of D, the second triad chord starts in E (which is called the root note of the chord). Then the second note has an interval of a third with E (which is a G) and then another third which is called the fifth of the chord (which is a B in this case). Therefore, the second chord of the major mode in D is made up by E, G and B.

The triad chords can be classified in three groups according to the intervals that the third and the fifth notes of the chord make with the root. These three groups are called major, minor and diminished:

Root	Third	Fifth	Group
grades I, IV and V	mayor third	perfect fifth	mayor chord
grades II, III and VI	minor third	perfect fifth	minor chord
grade VII	minor third	diminished fifth	diminished chord

Table 9.1 Relationships among intervals

It is important to note that a certain chord is not exclusive of a single mayor scale. Indeed, it will be in three different scales. For instance, Figure 9.3 depicts how the major chord of C is the first one in the tonality of C, the forth one in the tonality of G and the fifth one in the tonality of F.

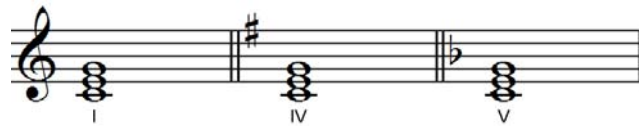


Figure 9.3 Triad cords in different tonalities

On the other hand, the chord made up from the seventh grade is exclusive of a single mayor scale.

9.2 Seventh Chords

If triad chords are made up by three consecutive notes, seventh notes are made up by four consecutive notes. This set of chords is made basically from the triad chords by adding the seventh interval. Thus, we have six seventh chords for each tonality:

Name	Notes
mayor seventh	1, 3, 5, 7
minor seventh	1, $\flat 3$, 5, $\flat 7$
minor seventh diminished fifth	1, $\flat 3$, $\flat 5$, $\flat 7$
dominant	1, 3, 5, $\flat 7$
dominant augmented fifth	1, 3, $\sharp 5$, $\flat 7$
diminished seventh	1, $\flat 3$, $\flat 5$, $\flat \flat 7$

Table 9.2 Seventh chords

9.3 Other Chords

Besides the triad and seventh chords there are other possible chords that are not usually utilized. They are mainly the sixth chords (by adding a sixth interval to the triad chord), ninth, eleventh, thirteenth, quartal and quintal. Moreover, we can also make chords with chromatic notes, and they are called altered chords.

Therefore, a certain performance might have a lot of different chord types. In addition, one can play a chord that is not located in these standard forms, and it might be good for a certain place of the song, thought. Thus, one might have whatever the chord made up by whatever the notes.

Part III:

The MIDI Specification

1 Introduction to MIDI

There are some common things that all musical instruments do, being the first one to make a sound under the control of some musician, so that the instrument starts making a sound whenever the musician wants to. For instance, he or she might push down a piano key, or fret and pick a guitar string. This action of starting a sound can be called 'Note On'.

On the other hand, most instruments also allow stopping the sound at any time. For instance, the musician can release the piano key or release his or her finger from the guitar fret. This action can be called 'Note Off'.

Moreover, most instruments can play distinct octaves. For instance, a piano has 88 keys so that there can be played more than 7 octaves. And even more: many instruments can also play notes at different volumes.

The MIDI standard is born to standardize and normalize all these possibilities for digital score music.

2 Messages

The MIDI protocol is made up of messages consisting of 8-bit bytes. Actually, many message types are defined in the MIDI specification. Even though a message can have unlimited number of bytes, every MIDI message is currently formed with up to 3 bytes. The first byte of the message is called the Status byte, and is important because it is the only one with bit number 7 set, being easy to detect when a message begins just by looking at this bit. Therefore, a message has two kinds of byte:

- Data byte, ranging from 0x00 to 0x7F.
- Status byte, ranging from 0x80 to 0xFF.

Moreover, Status bytes can be broadcasted on any of the 16 MIDI channels, and this is the reason why they are called Voice messages. For these Status bytes, the 8-bit byte is split into 2 nibbles, so that a Status Byte of 0x92 is split into a 0x9 for the higher nibble and a 0x2 for the lower one. The higher nibble tells what type of MIDI message is beginning, and can have one of those values show in Table 2.1:

Value	Meaning
0x8	Note Off
0x9	Note On
0xA	Aftertouch
0xB	Control Change
0xC	Program Change
0xD	Channel Pressure
0xE	Pitch Wheel

Table 2.1 MIDI message types

In the previous example of 0x92, Table 2.1 says that it is a 'Note On' message. The lower nibble tells which channel the message is applicable to. Since there are 16 possible channels in MIDI, 4 bits are needed to map them, so the lower nibble is used for that. Thus, the message 0x92 is a 'Note On' message applicable to channel number 2.

It is important to note that the lower nibble counts from 0 to 15. This means that for the MIDI protocol the first channel is number 0 rather than number 1. However, in most musical software channels are numbered from 1 to 16 since most people begins to count from 1. Therefore, the channel number for a musician would be the one in the Status byte plus 1, so the example 0x92 actually refers to channel 3.

Status bytes ranging from 0xF0 to 0xFF are for messages that do not belong to any particular channel but to all of them. These Status bytes are used to carry information of interest to every MIDI channel, such as synchronizing. These Status bytes are also split into two categories:

- System Common messages, ranging from 0xF0 to 0xF7.

- System Real-Time messages, ranging from 0xF8 to 0xFF.

Actually, some Status bytes are not used in the MIDI Specification [MMA], so if a MIDI device receives them it must just ignore them. For example, Status bytes 0xF4, 0xF5, 0xF9 and 0xFD are not used.

3 Voice Messages

Voice Messages are those that contain the actual performance in a MIDI stream. They are used to play notes or stop them, as well as indicate some variations in the sound such as volume, pitch and many other things. This kind of messages can be broadcasted on any of the 16 MIDI channels containing this type of information about the actual performance by using one among the seven message types listed in Table 2.1 and explained from now on.

3.1 Note Off

This message indicates that a particular note must be released, so that it must stop sounding. However, some patches might have a long VCA release time, needing therefore to slowly fade it out. In addition, some devices might have a Hold Pedal controller being on, and then the note release is postponed until the Hold Pedal is released, although this is done by the actual device.

As seen in Section 2, Status bytes for Note Off messages range from 0x80 to 0x8F, where the lower nibble specifies the MIDI channel.

The Note Off message contains two additional data bytes. The first one indicates the note number the message refers to. In the MIDI Specification there are 128 possible notes, numbered from 0x00 to 0x7F having the Middle C a value of 0x3C.

The second data byte is the velocity, also ranging from 0x00 to 0x7F. This parameter tells how quickly the note must be released (being 0x7F the fastest). It is up to the MIDI device how it uses the velocity parameter, besides some of them always send a value of 0x40 since they are not able to implement velocity features.

3.2 Note On

This message indicates that a particular sound must be played, so that it must start sounding. Once again, some devices might have a long VCA attack time that needs to slowly fade the sound in.

As seen in Section 2, Status bytes for Note On messages range from 0x90 to 0x9F, where the lower nibble specifies the MIDI channel.

The Note On message contains two additional data bytes. The first one indicates the note number the message refers to, having the same possible values as in the Note Off message.

The second data byte is the velocity, also ranging from 0x00 to 0x7F. This parameter tells with how much force the note must be played (being 0x7F the most force). It is up to the MIDI device how it uses the velocity parameter, besides some of them always send a value of 0x40 since they are not able to

perform velocity features. Actually, this parameter is used to tailor the VCA attack time or level, and therefore the overall volume of the note.

When a Note On message carries a value of 0 in the velocity parameter it is actually understood as a Note Off message, since it specifies no volume for the note. Thus, a MIDI device that recognizes Note On messages must be able to recognize both Note Off and Note On messages with velocity parameter set to 0.

In theory, every Note On message should be followed at any point by a Note Off message, even if the note's sound fades out due to some VCA envelope decay and stops sounding before the Note Off message arrives. In case another Note On message is given for a note and a channel that are already sounding, it is up to the device to layer another voice for the same pitch or to cut off the previous one in order to begin with the new one.

3.3 Aftertouch

Many electronic keyboards have some kind of pressure sensing circuitry that can detect how stronger is the pressure the musician applies to the key. Thus, the musician can vary this pressure even though the key is still held down. To exploit this behavior, the device typically generates many such Aftertouch messages while the pressure is varying. Therefore, upon receiving an Aftertouch message, devices vary note's VCA or VCF envelope sustain level or control LFO amount applied to the note's sound, which is the recommended effect.

As seen in Section 2, Status bytes for Aftertouch messages range from 0xA0 to 0xAF, where the lower nibble specifies the MIDI channel.

The Aftertouch message contains two additional bytes. The first one is used once again to indicate the note the effect must be applied to, being the same 128 possibilities as above. The second byte is the pressure amount, ranging from 0x00 to 0x7F, being this last the most pressure.

3.4 Controller

This kind of messages set a particular controller's value. There are 128 possible controllers in a MIDI device, ranging from 0x00 to 0x7F as usual. However, some of these controller numbers are predefined to be assigned to a particular hardware control in a device, such as the Modulation Wheel, which has controller number 1. Nevertheless, some others can be arbitrary assigned to any kind of custom controller.

As well as Table 2.1 shows, Status bytes for Controller messages range from 0xB0 to 0xBF, where the lower nibble specifies the MIDI channel.

After the Status byte in a Controller message, there can be found two additional Data bytes. The first one is used to indicate which controller the message refers to, whilst the second one specifies the value to be used to update the controller, ranging again from 0x00 up to 0x7F.

3.5 Program Change

These messages are used to change to a particular Program (i. e. instrument) within a certain MIDI channel. Most sound modules have several instrumental sounds, such as piano, guitar, trumpet, etc. so that this message can be used to select a program among that ones to obtain a different sound whenever that module interprets a Note On message. Once again, there are up to 128 possible program numbers within a sound module.

However, there are some MIDI devices that do not support several program numbers, such as a Reverb unit. In this case, these messages can be used to swap among different reverb presets. In any other case, messages can be simply ignored.

As was stated in Section 2, Status bytes for Program Change messages range from 0xC0 to 0xCF, being the lower nibble the MIDI channel.

Just one Data byte follows in a Program Change message: the number of program to change to, ranging from 0x00 to 0x7F as usual.

3.6 Channel Pressure

Actually, this kind of message means the same as an Aftertouch message does: specify pressure variations over notes that are sounding. An Aftertouch message gives pressure value just for one note, so that if two notes are being played and the first one receives more pressure than the second one, there will be sent an high aftertouch value for the first one and another message with a lower value for the second one.

However, a Channel Pressure message averages out pressure values for the whole set of notes that are being played. That is to say that only one value is given for the whole set, without an individual control over each key. This is a less powerful feature than aftertouch, but much cheaper to implement.

Usually, Aftertouch and Channel Pressure messages are not sent simultaneously. If a device implements Aftertouch messages there is no need for Channel Pressure values, since the others give even more information. Anyway, a certain device can implement both features if needed.

Values for Channel Pressure Status bytes range from 0xD0 to 0xDF as Table 2.1 says. In this case, the lower nibble is once again the MIDI channel the message refers to.

Just one Data byte follows in a Channel Pressure message, and it indicates the amount of pressure applied to the whole set of notes sounding, ranging again from 0x00 to 0x7F, being this last the most pressure.

3.7 Pitch Wheel

Pitch Wheel messages are used to slide a note's pitch up or down a certain amount. The resolution used for that slide is a fraction of a half-step

called cent. This feature is most common in string instruments such as a guitar, where a string can be bended after having been picked, even though this particular message refers to a tremolo bar which bends the whole bridge and hence the 6 strings. Moreover, some keyboard instruments can simulate this behavior with a wheel, turning it up and down.

This kind of messages has Status bytes ranging from 0xE0 to 0xEF as Table 2.1 shows, being the lower nibble the number of MIDI channel.

Two Data bytes appear in these messages, which have to be joined in order to obtain a 14 bits value which is the one that indicates the amount to be slid. Note that the first bit of each Data byte is useless since it is always set to 0. Thus, the first byte contains the lower 7 bits of the whole 14 bits value, whilst the second one contains the higher 7 bytes of the final value. Figure 2.1 depicts the process.

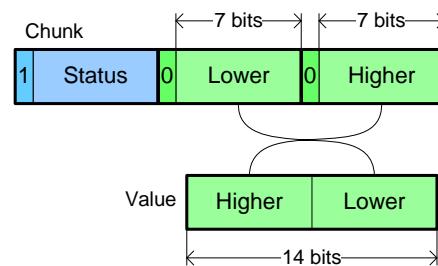


Figure 3.1 Pitch Wheel value obtaining

With 14 bits, the range is established from 0x0000 to 0x3FFF, where higher values transpose the pitch up and lower ones transpose the pitch down, being value 0x2000 the one meaning that the Pitch Wheel is centered and therefore not transposed.

The Pitch Wheel range can usually be adjusted in MIDI devices. For instance, a value of 0x3000 might transpose the pitch up a whole step, whereas in another device it might transpose it up just half a step. However, the General MIDI Specification recommends using the entire range of possible values for a transposition of -2 or +2 whole steps.

4 Other Messages

According to the MIDI specification, there are several more types of message that flow from and to a MIDI device. However, these messages are worthless for our purposes since most of them are just system and control messages used to make all the devices work together.

Thus, only voice messages are going to be considered for MIKE. In particular, only NoteOn and NoteOff messages are taken into account since they are the only ones that carry information just about when notes are played and stopped. The others carry information about how these notes are actually played.

In addition, most of the messages defined in the MIDI standard are not allowed in the SMF specification, so it would just be useless to talk about them.

Part IV:

Standard MIDI Files 1.0

1 Introduction

MIDI is an industry standard that defines each musical note in an electronic musical instrument, precisely and concisely, allowing these instruments and computers to exchange musical data. Note that MIDI does not actually transmit audio, but simply digital data about music. Besides some others, the MIDI standard defines the specification for MIDI files, which is formally known as SMF.

In its version 1.0, MIDI files can contain one or more MIDI streams, with time information for each event. The specification supports several musical entities among the ones shown in Part III of the document, such as sequence, tracks, time and key signatures, etc. Also, some additional information such as track names can be stored in MIDI files.

This version defines files in an 8-bit binary data stream, but data can also be stored in binary files, taking nibbles as units, compressed to 7-bit units for efficient MIDI transmission, converted to Hex ASCII or translated to a printable text file.

From now on, it will be covered only the 8-bit stream version.

2 File Block Structure

MIDI files are made up of chunks. As depicts Figure 2.1, each of these chunks has a 4 character field and a 32-bit number, which is actually the number of data bytes in the chunk, expressed in Motorola Big Endian format. This simple structure allows future new chunks to be introduced in the stream, so that they must be just ignored by a program written before the chunk was introduced. Therefore, a SMF reader program should expect rare chunks to appear in a stream.



Figure 2.1 SMF chunk format

SMF 1.0 defines two kinds of chunk: header chunk and track chunk. A header chunk provides some information about the whole MIDI file whereas a track chunk contains a sequential stream of MIDI data that can contain information for up to 16 MIDI channels.

Therefore, a MIDI file always starts with a header chunk, followed by one or more track chunks, following the structure in Figure 2.2.

MThd	Length	Header Data
MTrk	Length 1	Track 1 Data
MTrk	Length 2	Track 2 Data

Figure 2.2 SMF block structure

3 Header Chunk

This part specifies some basic information about the data in the file. The data section of the chunk contains three 16-bit words, so that the whole chunk follows the syntax in Figure 3.1.

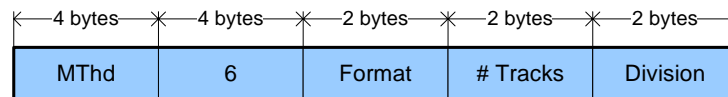


Figure 3.1 Header chunk syntax

As described in Section 2, the chunk starts with the type 'MThd' and the number 6 in 32-bit format (occupying four bytes). Afterwards, there are three additional values: format, number of tracks and division, which are going to be described in the following sections.

3.2 Format

This 16-bit word describes the whole organization of the file. A value of 0 specifies that the file contains only one single track with up to 16 MIDI channels. A value of 1 specifies that the file contains one or more simultaneous tracks assuming that all them starts at time 0, perhaps each one on a single channel. All these tracks, together, are considered a sequence or pattern. A value of 2 means that there are one or more sequentially independent single-track patterns.

Format 0 is the most common one to interchange data. Some programs use MIDI files as input just to apply some effect to a single track, such as mixers, sound effect boxes. Therefore it is desirable to be able to produce such a format, even though Format 1 can be easily converted to some files in Format 1. Moreover, Format 1 can be considered in a vertical one-dimensional form, as a collection of tracks.

MIDI files can express tempo and time signatures, to easily transfer tempo maps from one device to another. For a Format 0 file the tempo is scattered through the track so that the map reader must ignore events. In Format 1 the tempo map must be stored as the first track, starting at 0.04.

Every MIDI file should specify tempo and time signature. If a certain file does not do so, time signature is assumed to be 4/4 and tempo to be 120 beats per minute. In Format 0 these meta-events must appear at the beginning of the track, and in Format 1 they must appear at the beginning of the first track. For Format 2 each track must contain at least initial time signature and tempo information.

3.3 Number of Tracks

This value tells how many tracks are contained within the file. Therefore, it says how many Track chunks are expected afterwards. Of course, a Format value of 0 requires a Number of Tracks value of 1.

3.4 Division

The last value in the Header chunk contains the division of a crotchet note represented by delta-times in the file. However, if the value is negative, it stores the division of a second represented by delta-times in the file, so that events occur in actual time rather than in metrical time.

The upper byte is one among -24, -25, -29 and -30 according to the four standard SMPTE and MIDI time code formats [SMPTE], representing the number of frames per second. The second byte is the resolution within a frame, being typical values 4 (MIDI time code resolution), 8, 10, 80 (bit resolution) or 100. Thus, if it is specified 25 frames per second and a resolution of 40, the overall resolution of the performance is 1 millisecond.

4 Track Chunk

After the Header Chunk there appear as many Track Chunks as defined in the header's number of tracks, so that there is a chunk per track. The chunk is identified by 'Mtrk' and it contains an amount of data that vary depending on the track.

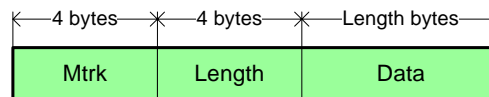


Figure 4.1 Track chunk syntax

As well as in a MIDI flow, a track contains events ordered by time. Once again, this delta time contains the amount of time that should pass between the previous event and the current one. Thus, the first event is assumed to have a delta time 0.

A track chunk contains the same messages as a MIDI flow besides some others that contain specific information about the performance such as the time or key signatures.

There are System Exclusive messages that come also from the MIDI standard messages that can be applied in a SMF file with the event status F0. Another kind of messages are the so called Metaevents, which have a status byte of FF. Note that this status indicates in the MIDI standard a reset message that, on the other hand, would not make any sense in a SMF file. Therefore, the status byte FF is used to indicate a metaevent.

4.2 Metaevents

Besides the voice messages, a SMF file might have other events as seen before. These events can carry information about the sequence number for a certain track, some textual information placed at some point in time, copyright information, the track name, instrument name for each track (that can be changed with a Program Change message), the lyrics of the piece, marks that act as replay symbols (rarely used), devices and port names and so on.

In addition, there are some other metaevents that are totally useful for the system since they indicate the tempo, time and key signature. The first one indicates the number of microseconds per crotchet note. The time signature message contains both the numerator and denominator of the actual signature, whilst the key signature consists on the number of sharps or flats that the signature contains.

However, these last 3 messages are not mandatory in a SMF file. Thus, a default value is assigned in case any of them does not appear. These default values are 120 beats per minute, a 4/4 time signature and a major C key.

Part V:
The RSHP Model and the CAKE
Engine

1 Artifacts Classification and Retrieval

The RSHP model implies a general framework for classifying and retrieving any kind of artifacts, so that it can be applied to whatever information management process.

This framework is divided into four components that are described in the following sections:

- The artifact information representation model r_α
- An artifact indexing process $I(\alpha)$
- A classification process $C(i_\alpha)$
- The artifact retrieval process $R(i_q)$

1.1 Artifact Information Representation Model r_α

A representation model r_α is the actual definition of how every artifact must be stored in a computer system. These describing elements are called descriptors and are usually keywords or natural language words. In a graphical representation like the one in Figure 1.1, every artifact α has a representation in the information-representation map.

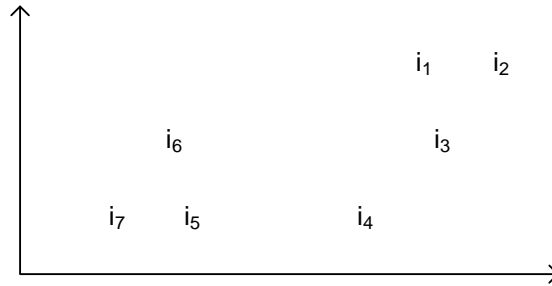


Figure 1.1 Graphical representation of artifacts

1.2 Artifact Indexing Process $I(\alpha)$

The indexing process describes the way a new artifact α is placed into the representation map by generating an artifact index i_α

$$\forall \alpha \exists r_\alpha / I(\alpha) = i_\alpha \quad [1.1]$$

where i_α is the computer-based description of α within the representation map. This process is called indexing and is represented in Figure 1.2.

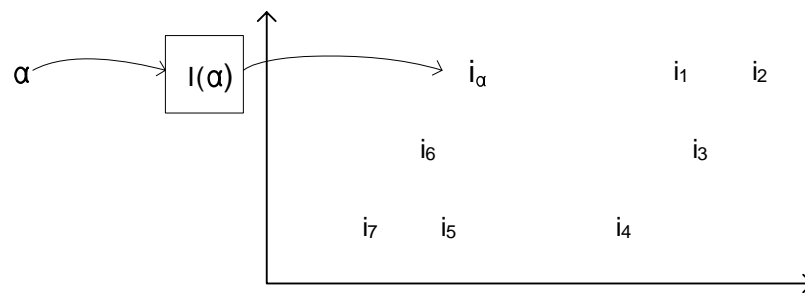


Figure 1.2 Artifact indexing

1.3 Classification Process $C(i_α)$

This process assigns a class or classes to $i_α$. A class basically groups all the artifact representations i_i considered similar. Even though this functionality is usually not implemented, it makes easier the retrieval process if it is based on the previously made classification.

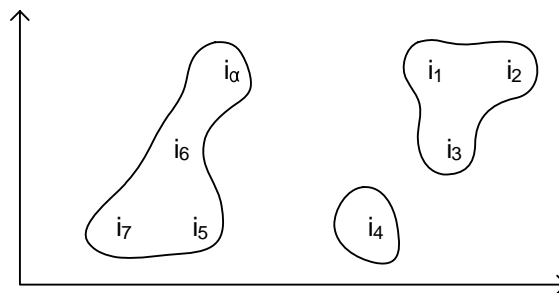


Figure 1.3 Artifact classification

1.4 Artifact Retrieval Process $R(i_q)$

With this process it can be determined which artifact representations are relevant to a query. Therefore, given a query q the retrieval process returns artifacts which present some similarity with the query q , which is also treated as an artifact i_q .

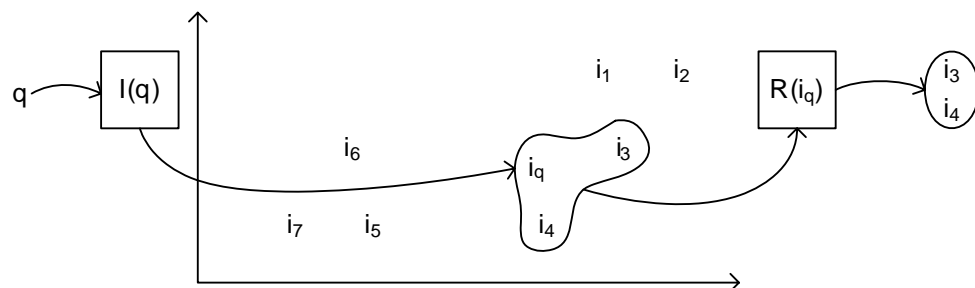


Figure 1.4 Artifact retrieval process

2 The RSHP Information Representation Model

This section introduces the RSHP information representation model that is used by the CAKE engine and therefore will be used to carry on with the project objective.

2.1 Motivations Behind RSHP

In order to allow information reuse it is necessary to find an information representation model capable to cope with any kind of artifact types. The RSHP model can be used as such representation model since it faces the main problems that the rest of representation models can not solve totally:

- Term-based models are good for textual artifacts but do not work properly with structured artifacts such as software object model. Moreover, two main problems arise from the usage of these models: selection and classification can not be automated.
- Element-data models work fine for data retrieval such as source code, but not so well for the rest of artifact types.
- Database and Software-design models are becoming standard representation paradigms for software artifacts and maybe for some other modeling kinds. However, the use of these models is almost impossible for textual artifacts.
- Behavioral models are good for dynamic artifacts but not for static artifacts out of that scope.
- Formal methods are designed for the kind of information they are intended to model and there is no generalized formal representation model yet.
- Knowledge-based models have not been defined to represent artifacts content and thus there is no general model for artifact classification.

2.2 Inside RSHP

The main idea behind RSHP is the fact that information is, in essence, related facts. Therefore, it is considered mandatory to focus on relationships as the most important asset. Relational data modeling, object oriented models, processes and even UML itself can be modeled as elements linked by relationships. Text information can also be represented as relationships between terms since it actually uses the sentence structure as base.

Thus, the RSHP information representation model is based on this principle [Llorens, 2003]: “In order to represent information, the main description element to be found within an artifact should be the relationship. This relationship is in charge of linking information elements”. Thus, the atomic information components are information elements that will be linked by relationships. These elements are defined by concepts that are represented by a normalized term and can also be treated as artifacts.

Therefore, the representation of whatever artifact is as simple as

$$i_{\alpha} = \{f(RSHP_1), f(RSHP_2), \dots, f(RSHP_n)\} \quad [2.1]$$

where

$$RSHP = \{IE \text{ describing the dynamics of the relationship}, \\ IE_1, IE_2, \dots, IE_n\}^{RSHP \text{ type}} \quad [2.2]$$

Properties are introduced into the model to allow metadata modeling, and also Information Elements to allow an artifact to contain other artifacts by using them as descriptors. Thus, the general model is:

$$\forall \alpha, l(\alpha) = i_{\alpha} = \{f(RSHP_1), f(RSHP_2), \dots, f(RSHP_n), \\ f(Pty_1), f(Pty_2), \dots, f(Pty_m), \\ f(IE_1), f(IE_2), \dots, f(IE_p)\} \quad [2.3]$$

having

$$Pty = \{IE \text{ describing the tag of the property}, IE_1, IE_2, \dots, IE_n\} \quad [2.4]$$

$$IE = \{Term / < term \text{ describing artifact } >_{Artifact \text{ type}}\} \quad [2.5]$$

Also, $f(RSHP)$ is whatever function that receives the Information Elements that form the relationship, applying the same to $f(Pty)$ and $f(IE)$. Therefore, the textual artifact “Computers are machines and they have processor” and considering $f(RSHP)=RSHP$, the artifact might be represented as:

$$i_{\alpha} = \{RSHP_1, RSHP_2\} \quad [2.6]$$

where

$$RSHP_1 = \{to \text{ be, computer, machine}\}^{Hierarchy} \\ RSHP_2 = \{to \text{ have, computer, processor}\}^{Aggregation} \quad [2.7]$$

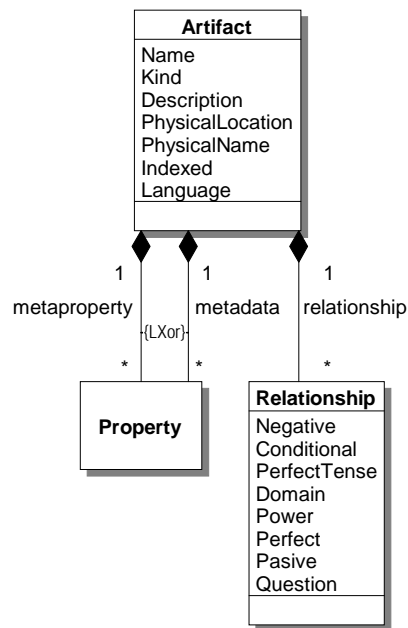


Figure 3.2 Artifact contents (part I)

Despite an artifact is defined for relationships, it can also be represented by means of single terms through Information Elements. Moreover, an artifact might aggregate other sub-artifacts thanks to a simple generalization.

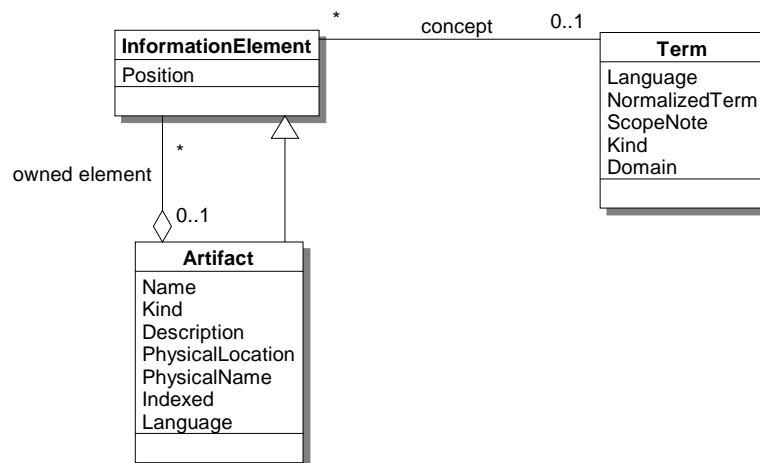


Figure 3.3 Artifact contents (part II)

3.3 Term

In Section 2 was introduced a controlled vocabulary that Information Elements must use, and this vocabulary is actually composed by Terms. According to the restrictions of the RSHP model, an implicit 1 to 1 relationship among a concept and its representation term must be considered. Thus, every single noun, keyword, artifact, etc., must be modelled with terms, so that a given term is the normalized representation of a unique concept.

In addition, two different kinds of terms are considered:

- Dynamic actions as the actual meaning of relationships.
- Static concepts as the elements linked by relationships and dynamic actions.

3.4 Relationship

As seen in Section 2, relationships are intended to be the descriptors of all types of artifacts' information. Whatever the relationship, it is defined by some occurrences of concepts called Information Elements.

Relationships can be or not symmetric, so that these concepts may have a determined order. In addition, a relationship can have an information element naming its dynamic action.

Finally, a RSHP links Information Elements by means of fuzzy measurements, assigning a quantifiable worth to the relationship. Moreover, indexing and retrieval algorithms for RSHP are very dependant on the relationship kinds they work with in the artifacts, so every found relationship must be typed. This is accomplished with RSHPSemantics, even though some properties in the Relationship class are used when modeling textual information.

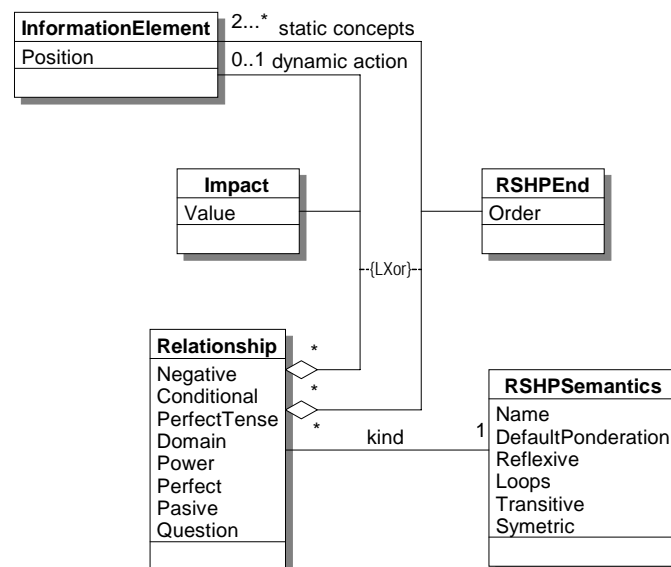


Figure 3.4 Relationships structure and semantics

3.4.1 RSHPSemantics

This class is used to model relationships information. The three properties of a mathematical equivalence relationship can be used: reflexivity, symmetry and transitivity, as well as loops and default weighting for query propagation and domain generation. Thus, RSHPSemantics handles two aspects:

- It qualifies existing relationships in the model.

- It qualifies the dynamic terms to allow indexers identify relationship types.

3.5 Information Element

An Information Element is the minimal information unit when it does not aggregate other IEs. According to the RSHP metamodel in Figure 3.1 an Information Element can be either a term within an artifact (at an optional position) or an artifact itself. In addition, an IE can be part of a relationship or a property. As Figure 3.5 depicts, an Information Element is simply the occurrence of a concept labelled with a certain Term.

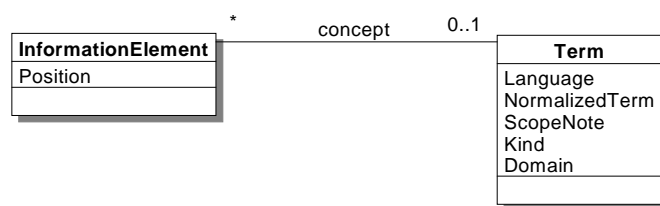


Figure 3.5 Information Elements

3.6 Property

In previous versions of the RSHP metamodel, only relationships were used for modelling, and artifact's metadata were represented by a particular metadata relationship. However, in the version used for this final project these relationships are separated because they have obvious different semantics.

Every property has one IE naming the tag and one or more IE's as actual values.

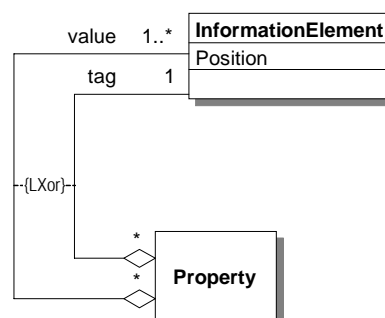


Figure 3.6 Properties structure

4 The CAKE Engine

The CAKE Engine is basically a retrieval framework that uses the RSHP Information Representation Model as main tool [Llorens, 2002]. Even though it was born to cope with Software Reuse based on XMI documents [OMG, b], it might be used for whatever domain thanks to the use of the RSHP model.

It has been demonstrated as a good choice for Software reuse, as well as for textual information and even spreadsheets. Now, the main goal of the current final project is to check whether it can be used or not for music reuse.

The following sections will give a brief about the CAKE Engine by means of Software Reuse and XMI artifacts, so that their indexing and retrieval processes can be clearly understood since they are the most important tools for the RSHP artifacts reuse.

5 XMI Indexing

In order to index XMI files three processes are required:

- Parse the XMI file and gather its information.
- Transform that information into UML information [OMG, a].
- Translate the UML representation to the database model.

5.1 XMI Parser

Since XMI is a well defined standard, a XMI parser can check whether the file is well formed or not. In such a case, two steps must be performed later:

- Identify every UML model elements in the document, first looking for those ones that are not derived from the UML Relationship meta-class.
- Identify every relationship among the model elements.

5.2 Information Storage in Memory

The information gathered with the XMI parser is introduced into an information structure that represents the UML metamodel, so that two main goals are achieved:

- Offer the possibility to graphically visualize the information by means of graphs.
- Prepare it to be stored in persistent systems in the third phase.

5.3 Information Storage in a Database

Once the XMI documents have been parsed and their information extracted and represented with the UML metamodel, it must be stored into a database to provide persistence and a common access for the retrieval process.

6 XMI Retrieval

Thanks to the XMI indexing process, UML documents are stored and classified into the repository. Now the framework must provide an accurate retrieval that allows the reutilization of UML software artifacts.

6.1 UML Query Creation

One of the most important differences that RSHP introduces is that the user interface must be radically different than the existing ones, say, graphical. Moreover, using the QbyE paradigm as the ground technique for creating UML techniques allows the framework to provide a concrete solution to the concept-term mapping problem. Therefore, when formulating a UML query the user must select the names of the artifacts from a controlled vocabulary so that:

- People involved in different projects share the same vocabulary.
- Better results will be obtained as the queries better represent the semantics wanted to define.

6.2 UML Query Formulation and Resolution

Considering the information depicted in the class diagram in Figure 6.1



Figure 6.1 A target document to retrieve

to be retrieved, it can be represented by means of RSHP in the following way:

$$RSHP_1 = \{ \text{"no name"}, < \text{Computer} >_{\text{class}}, < \text{Processor} >_{\text{class}} \}^{\text{Association}} \quad [6.1]$$

where RSHP1 is an association relationship, and $< \text{Computer} >_{\text{class}}$ and $< \text{Processor} >_{\text{class}}$ are sub-artifacts typed as UML classes. Thus, the representation of the $< \text{Computer} >_{\text{class}}$ sub-artifact is as follows:

$$\begin{aligned} IE_1 &= \{ < \text{SerialNumber} >_{\text{attribute}} \} \\ IE_2 &= \{ < \text{Start} >_{\text{method}} \} \end{aligned} \quad [6.2]$$

and the representation of the $< \text{Processor} >_{\text{class}}$ sub-artifact is:

$$IE_1 = \{ < \text{Name} >_{\text{attribute}} \} \quad [6.3]$$

Finally, the $< \text{Name} >_{\text{attribute}}$ has additional information, so it is also represented as a sub-artifact:

$$RSHP_1 = \{\text{"no name", Data Type, Integer}\}^{\text{Property Qualification}} \quad [6.4]$$

According to the representation model above, the retrieval framework must provide two different retrieval capabilities:

- Query inclusion.
- Query similarity.

6.2.1 Query Inclusion

With this query type they can be searched models that fully include the query, element by element and relationship by relationship. Since all the artifacts indexed are stored in a database according to the indexing process in Section 5, a single (and very complex) SQL SELECT statement might be used, with many WHERE clauses including joins and sub-SELECT clauses for every artifact.

6.2.2 Query Similarity

This kind of query is based on a similarity function between an artifact a and a query q that must be calculated for every artifact in the repository, considering then the performance as an important topic. Considering that UML models are being compared, this measure implies to compare two diagrams and gives a value for how similar they are.

For this purpose, two main aspects are considered:

- Model Topology.
- Model Semantics.

Thus, the similarity function between an artifact a and a query q is:

$$F_{\text{Similarity}}(a, q) = K_T \cdot F_T(a, q) + K_S \cdot F_S(a, q) \quad [6.5]$$

where $F_T(a, q)$ measures the similarity among two models based on the relationships they have, while $F_S(a, q)$ compares the type of the relationships they have in common. On the other hand, K_T and K_S are constants for fine tuning the whole function.

6.3 Topology Measurements

The $F_T(a, q)$ function from [6.5] gives values according to the relationship types found in the artifact and in the query. For this purpose is used a vector space model with one dimension per relationship type, so that the vector space has as many dimensions as relationship types.

Two vectors representing the artifact and the query are created in such a vector space, each of them including for every dimension the number of relationships found of the corresponding type as Figure 6.2 depicts:

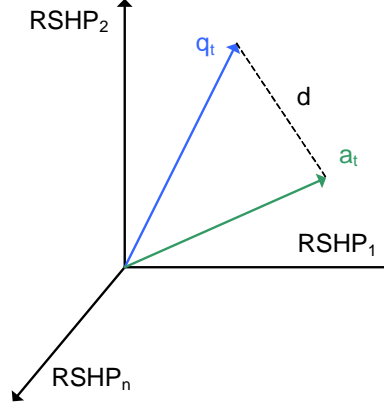


Figure 6.2 Vector space model for topology measurement

where n is the total number of different relationship types and:

$$\begin{aligned} a_t &= (\text{Number of RSHPs}_1 \text{ in Artifact,} \\ &\quad \text{Number of RSHPs}_2 \text{ in Artifact,} \\ &\quad \dots, \\ &\quad \text{Number of RSHPs}_n \text{ in Artifact}) = (a_{t1}, a_{t2}, \dots, a_{tn}) \\ q_t &= (\text{Number of RSHPs}_1 \text{ in Query,} \\ &\quad \text{Number of RSHPs}_2 \text{ in Query,} \\ &\quad \dots, \\ &\quad \text{Number of RSHPs}_n \text{ in Query}) = (q_{t1}, q_{t2}, \dots, q_{tn}) \end{aligned} \quad [6.6]$$

The similarity among these two vectors is measured by means of the Euclidean distance. In addition, in order to ponder the case where one model fully includes the other, the $F_T(a, q)$ function is moderated by a sign-based variable, say s , that takes its value from the following:

$$\text{sign}_i := \text{sign}(a_{ti} - q_{ti}) \quad [6.7]$$

and

$$s = \frac{\sum_{i=1}^n (\text{positive sign}_i)}{n} \quad [6.8]$$

Therefore, the topology function is formulated as:

$$F_T(a, q) = \sqrt{(a_{t1} - q_{t1})^2 + (a_{t2} - q_{t2})^2 + \dots + (a_{tn} - q_{tn})^2} \cdot C_{\text{inclusion}} \quad [6.9]$$

where

$$C_{\text{inclusion}} = \left[1 - \left(K_{\text{inclusion}} \cdot ((3k_i + 1)s^2 - 4k_i s + k_i) \right) \right] \quad [6.10]$$

having:

- $0 \leq K_{\text{inclusion}} \leq 1$ where a value of 0 means that no inclusion effect is taken into account.
- $0 \leq k_i \leq 1$ with the following meaning:
 - $k_i = 0 \rightarrow F_T(a, q) = 0$ only if $a \subset q$
 - $k_i = 1 \rightarrow F_T(a, q) = 0$ if $a \subset q \vee q \subset a$

6.4 Semantics Measurement

The semantics distance $F_S(a, q)$ among an artifact and a query is measured considering two aspects:

- The common concepts.
- The common RSHPs.

Therefore, the semantics function is as follows:

$$F_S(a, q) = K_{\text{IEs}} \cdot F_{\text{IEs}}(a, q) + K_{\text{RSHPs}} \cdot F_{\text{RSHPs}}(a, q) \quad [6.11]$$

where K_{IEs} and K_{RSHPs} are constants for fine tuning and $F_{\text{IEs}}(a, q)$ is:

$$F_{\text{IEs}}(a, q) = \frac{\text{Number of IEs from Query in Artifact}}{\text{Number of IEs from Query}} \quad [6.12]$$

in case the query has less IEs than the artifact, otherwise the function is calculated as

$$F_{\text{IEs}}(a, q) = \frac{\text{Number of IEs from Artifact in Query}}{\text{Number of IEs from Artifact}} \quad [6.13]$$

$F_{\text{RSHPs}}(a, q)$ measures the distance between all the artifact's RSHPs and all the query's RSHPs by comparing the distance of every single RSHP from one document (the one with less RSHPs) with all the RSHPs from the other one, one by one using a function $\Delta_{\text{RSHP}}(\text{RSHP}_1, \text{RSHP}_2)$ and selecting the combination with the minimum distance. Therefore, and assuming $|a| \leq |q|$:

$$\begin{aligned}
 F_{\text{RSHPs}}(a, q) = c_o \mid \\
 c_o \in C \wedge \\
 C := \{c_i \mid c_i = \{(r_1, s_{i1}), \dots, (r_n, s_{in})\} \mid \\
 a = \{r_1, \dots, r_n\} \wedge \\
 \forall j, k \in \{1, \dots, n\} : s_{ij}, s_{ik} \in q \wedge s_{ij} \neq s_{ik}\} \} \wedge \\
 \forall c_k \in C : \sum_{i=1}^n \Delta_{\text{RSHP}}(r_i, s_{ki}) \geq \sum_{i=1}^n \Delta_{\text{RSHP}}(r_i, s_{oi})
 \end{aligned} \tag{6.14}$$

Once every possible distance is calculated, it is selected the minimum possible value for all the combinations. For instance, in Figure 6.3 function $F_{\text{RSHPs}}(a, q)$ might be formed by pairs RSHP_{a1} - RSHP_{q2} and RSHP_{a2} - RSHP_{q1} with a distance value of 0.2 or by pairs RSHP_{a1} - RSHP_{q1} and RSHP_{a2} - RSHP_{q2} with a distance value of 0.15.

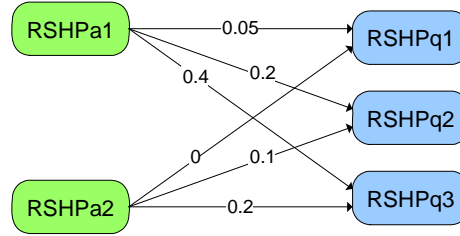


Figure 6.3 RSHP difference map

Therefore, the semantic distance between two RSHPs is calculated with the function $\Delta_{\text{RSHP}}(\text{RSHP}_1, \text{RSHP}_2)$, which takes two aspects into account:

- The total distance between all the IEs in both RSHPs.
- The difference of both RSHPs according to their dynamic concept (action) by means of an IE.

Thus, the function is defined as follows:

$$\begin{aligned}
 \Delta_{\text{RSHP}}(\text{RSHP}_1, \text{RSHP}_2) = K_{\text{RSHP IEs}} \cdot F_{\text{RSHP IEs}}(\text{IEs}_1, \text{IEs}_2) + K_{\text{Action}} \cdot \Delta_{\text{Action}}(A_1, A_2) \mid \\
 \text{IEs}_i := \{\text{IE}_k \mid \text{IE}_k \in \text{RSHP}_i\} \wedge A_i = \text{Action}(\text{RSHP}_i)
 \end{aligned} \tag{6.15}$$

Function $F_{\text{RSHP IEs}}(\text{IEs}_1, \text{IEs}_2)$ gives the difference between two sets of Information Elements by comparing every single IE from one set (the one with less elements) with every IE from the other set by means of a function $\Delta_{\text{IE}}(\text{IE}_1, \text{IE}_2)$, selecting later the combination with the minimum distance. Therefore, and assuming $|\text{IEs}_1| \leq |\text{IEs}_2|$:

$$\begin{aligned}
 F_{\text{RSHP}|\text{IEs}}(\text{IEs}_1, \text{IEs}_2) &= \sum_{i=1}^n \Delta_{\text{IE}}(r_i, s_{oi}) \mid \\
 &\quad c_o \in C \wedge \\
 C &:= \{c_i \mid c_i = \{(r_1, s_{i1}), \dots, (r_n, s_{in})\} \mid \\
 &\quad \text{IEs}_1 = \{r_1, \dots, r_n\} \wedge \\
 &\quad \forall j, k \in \{1, \dots, n\} : s_{ij}, s_{ik} \in \text{IEs}_2 \wedge s_{ij} \neq s_{ik}\} \} \wedge \\
 \forall c_k \in C : \sum_{i=1}^n \Delta_{\text{IE}}(r_i, s_{ki}) &\geq \sum_{i=1}^n \Delta_{\text{IE}}(r_i, s_{oi})
 \end{aligned} \tag{6.16}$$

In order to compare both sets, two aspects are considered to control the roles that IEs play in both RSHPs:

- The order of every IE in the relationship.
- The symmetry of every RSHP.

In order to moderate the possibility when two RSHPs are different even when they have the same Action and the same IEs but in different order, a K_{punish} constant punishes the distance between two IEs when they are not in the same side of their asymmetric relationships:

$$\Delta_{\text{IE}}(\text{IE}_1, \text{IE}_2) \cdot (1 + K_{\text{punish}}) \tag{6.17}$$

The value of this function in charge of comparing two IEs works as follows:

$$\Delta_{\text{IE}}(\text{IE}_1, \text{IE}_2) = \begin{cases} 0 & \text{IE}_1 = \text{IE}_2 \\ K_{\text{syn}} & \text{IE}_i \in \text{Equi}(\text{IE}_j) \mid i \neq j \\ n & \text{IE}_i \cap \text{Prop}^n(\text{IE}_j) \neq \emptyset \mid i \neq j \wedge n \text{ is minimal} \\ K_{\text{syn}} + n & \text{Equi}(\text{IE}_i) \cap \text{Prop}^n(\text{IE}_j) \neq \emptyset \mid i \neq j \\ K_{\text{punish}} & \text{Equi}(\text{IE}_i) \cap \text{Prop}^n(\text{IE}_j) = \emptyset \mid i \neq j \end{cases} \tag{6.18}$$

where:

- n is an integer representing the propagation level in a ISO2788 net.
- N is the maximum propagation level so that $n \leq N$.
- $\text{Equi}(\text{IE}_i)$ is the set of synonyms of IE_i .
- $\text{Prop}^n(\text{IE}_i)$ is the set of related IEs for propagation level n . For instance, upon the net depicted in Figure 6.4 might be applied $\text{Prop}^1(\text{Workstation}) = \{\text{Computer}, \text{IBM-PC}\}$ as well as $\text{Prop}^3(\text{Laptop}) = \{\text{IBM-PC}\}$.

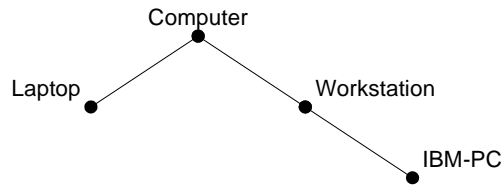


Figure 6.4 Propagation in the ISO2788 net

- $0 \leq K_{\text{syn}} \leq 1$ where a value of 0 means that synonyms are treated as IEs, while a value of 1 means that synonyms are slightly punished: $K_{\text{punish}} = 3N$.

On the other hand, the function $\Delta_{\text{Action}}(A_1, A_2)$ from [6.15] measures the difference among two RSHPs according to their action:

$$\Delta_{\text{Action}}(A_1, A_2) = \begin{cases} \Delta_{\text{IE}}(\text{IE}_1, \text{IE}_2) & \text{RSHP}_1 = \text{RSHP}_2 \\ K_{\text{punish}} & \text{RSHP}_1 \neq \text{RSHP}_2 \end{cases} \quad [6.19]$$

Part VI: Definition of the User Requirements

1 Introduction

This Section provides an overview of the User Requirements Definition phase and the scope of the system to develop.

1.1 Purpose

The main purpose for this phase of the project is to establish and detail all the features the final user expects from the system. Previous parts of this document have given a background about the musical domain and the CAKE environment. Thus, the User Requirements Definition phase will group all them to show the actual needs and the problems and risks that might arise from them.

This part of the report is addressed to those readers that want to know what the system is about and the requirements it must comply to, avoiding much technical stuff and focusing only on the actual needs.

1.2 Scope

The main goal of the current project is to develop a system that allows music reuse and, hence, some other capabilities derived from it such as musical comparisons in a quantitative way.

The system functionality can be divided into three main parts:

- Index musical MIDI files according to their contents, discarding if needed some useless information.
- Retrieve MIDI files similar to another one given as a query by means of similarity or inclusion.
- Present the results of the retrieval according to the specific needs of the actual user.

The first two parts shape the core of the project, while the third one can change depending on the actual user needs.

2 General Description

This section describes the general factors that affect the system and its requirements, making them easier to understand.

2.1 System Perspective

First of all, it is important to note that the CAKE Engine introduced in Part V of the document must be used along with the RSHP Information Representation Model. This requirement establishes an operational environment that the system must comply with.

Since it is mandatory to use the CAKE Engine, a model for musical information must be developed according to the RSHP specification. Moreover, the system must be included in the CAKE Studio software by implementing some required extensions.

2.2 User Characteristics

The main users of the system will be musicians or, at least, people who has some musical background.

However, lastly it seems to be necessary some training on the MIKE internal process since some feedback from the user would be necessary in the process. As we will see in the Part X of the document, the voice separation process needs some information from the user about how good are the intermediate separations so that the system can adjust some penalization constants.

Therefore, this background might be necessary. On the other hand, this interaction with the user is not needed in the current state of the system, but will be necessary later on in final versions though.

3 General Requirements

This section enumerates the general constraints that affect the development of the system, from the technical constraints to those derived from the actual musical domain.

3.1 CAKE Studio 3.0.0

The main constraint for the system is that it must run under the CAKE Studio 3.0.0 [dTinf] and, therefore, under Microsoft .net technology [Microsoft]. In particular, a Manager for the CAKE Studio must be created in order to cope with musical files indexing and retrieval and it must run under the .net Framework 1.1 4322 sp 1.

Moreover, the CAKE Engine and therefore the RSHP Information Representation Model must be used as a result of the CAKE Studio usage as stated before.

3.2 File Format

The basic way to retrieve musical information is to provide to the CAKE Studio an artifact to use as query. However, this artifact must be actually a file of the same type that those that were already indexed previously. The CAKE Studio allows more than one file format per indexer, so a decision must be taken about the supported file formats.

3.3 Vertical Constraints

By vertical constraints are considered those that affect to the height of the notes, tonalities, grades, key signatures and so on.

3.3.1 Octave Equivalence

Figure 3.1 depicts a melody written in the fourth octave first and then in the fifth one. They represent the same performance but played in different octaves.



Figure 3.1 Octave equivalence

Anyway they are the same, so the system must ignore the octave issue since it lacks of importance. Actually, each instrument has a different number of possible octaves, so it makes no sense to compare them as a whole.

3.3.2 Grade Equality

Figure 3.2 shows part of the main riff of the song Layla, from Derek and the Dominoes, in its original key signature of major F. As it depicts, the riff starts with the third grade of the tonality, followed by the fifth one, the sixth and so on.

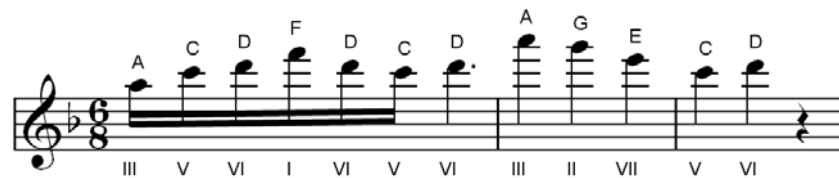


Figure 3.2 Grade equality (part I)

If it is considered now the riff in Figure 3.3, it can be seen that the same performance is represented. Third grade of the tonality starts, followed by the fifth one, the sixth and so on.



Figure 3.3 Grade equality (part II)

However, in this case the song is in the key signature of major Bb (7 semitones down). Therefore, even though the grade progression is maintained, the resulting notes are totally different. It is pretty clear that the riffs are the same, so the system must find out similarity among them.

3.3.3 Note Equality

Figure 3.4 shows exactly the same performance that Figure 3.2 does. This means that exactly the same notes make it up: A followed by C, D and so on.



Figure 3.4 Note equality

However, in this case the riff is in the tonality of major C, so the grades will be different. In Figure 3.2 grades are III followed by V, VI and so on. Nevertheless, in Figure 3.4 they are VI followed by I, II and so on. Since both riffs are exactly the same, the system must find out again the similarity among them.

3.3.4 Chord Recognition

Another desired feature for the system is the ability to recognize chords or part of a performance. Moreover, the best solution to the chord recognition would be the ability to recognize not only a certain chord as a whole, but also part of it. For instance, in a triad chord (made up by the root, the third and the fifth) one might only recognize two notes (typically the root and the fifth). However, one might recognize another two or simply just the root note of the chord.

Actually, a typical technique to recognize chords is to start playing them just with the root note and then, progressively, figure out the other ones and add them to the chords.

Therefore, the system should be able to recognize chords wholly and partially. Of course, two identical chords should have a semantic distance of zero whilst a chord and another one that is part of the first one should give a distance between 0 and 1, but close to 0 at some extent depending on how many notes are included.

3.4 Horizontal Constraints

By horizontal constraints are considered those that affect to the length of the notes, time signatures, bars, etc.

3.4.1 Time Signature Equivalence

Figure 3.5 depicts a simplified version for the start of op. 81 no. 10 from S. Heller in major E with the original 2/4 time signature. This means that the piece is split into 4 bars of 2 beats each, being each beat equivalent to a crotchet.



Figure 3.5 Time signature equivalence (part I)

However, if a 4/4 time signature is considered, the piece would be split into 2 bars of 4 beats each, being each beat equivalent to a crotchet as well.



Figure 3.6 Time signature equivalence (part II)

Actually, as shows Part II, Section 3.4 the only difference among the two previous staves is how hard each note must be played. However, in essence, both them are exactly the same, so the system must ignore somehow the time signatures in cases like this.

3.4.2 Tempo Equality

Let us consider the staff in Figure 3.7. It depicts a simple melody with a tempo of 60 crotchets per minute and a key signature of 4/4.



Figure 3.7 Tempo equality (part I)

If the same melody is played at twice the speed (120 crotchets per minute) or half the speed (30 crotchets per minute) but maintaining the actual time, result in the two staves depicted in Figure 3.8.



Figure 3.8 Tempo equality (part II)

In the first one, as the tempo is twice, every note has twice length as well. Likewise, the second one has half the tempo, so every note has half its length. It is clear that the three staves are equal, but with different tempos, note lengths, time signatures or number of bars. Anyway, the system must consider them as exactly or almost equal.

3.4.3 Figure Equality

If the melody in Figure 3.6 is played slower or quicker by means of a tempo variation, the result would be like the two staves in Figure 3.9.



Figure 3.9 Figure equality

This case is quite similar to the tempo perfect equality, but with a difference of actual time. Anyhow, the system must consider them as similar melodies or even the same one.

3.4.4 Partial Similarity

Sometimes, a melody is altered by changing only the duration of a note. For instance, if the melody in Figure 3.7 is changed by means of note length, a melody like the one in Figure 3.10 could be formed.



Figure 3.10 Partial similarity (part I)

Both melodies are quite similar but not identical. Therefore, the system must consider them as similar but never equal. The same consideration should be made if notes length is altered along with the whole tempo, like Figure 3.11 depicts.



Figure 3.11 Partial similarity (part II)

3.4.5 Time Quantization

Sometimes, MIDI files are recorded in such a way that there is no explicit information about the onset time and duration of each note, so that the only data available is the amount of milliseconds.

The system should deal with these particular cases so that it would be nice to compare performances by score and real time.

3.5 Voice Constraints

Figure 3.12 depicts the original start of op. 81 no. 10 from S. Heller. As it can be seen, actually two voices make it up, colored in blue and green. The instrument that plays this performance is a piano, and as it uses to happen with many instruments (mainly keyboard instruments) there are two voices (one per hand).



Figure 3.12 Simple voice distinction

These melodies work together as a whole, but can also be treated individually. Indeed, if this performance is played with a melodic instrument (flute for instance), the musician will only be able to play one of them at the same time. Therefore, it would be necessary to compare the whole piece with only the blue melody or maybe only the green one.

Thus, the system must be able to treat voices separately but also together. Moreover, the voice distinction might be done not only for one staff with the Sol clef and another voice for the Fa clef. Indeed, as Figure 3.13 depicts, several voices might be found on a single staff.

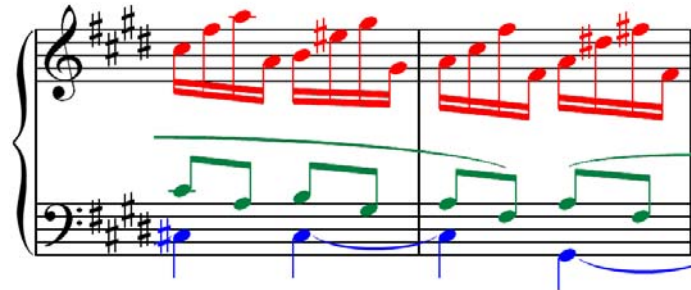


Figure 3.13 Complex voice distinction

Actually, a single staff can have up to 4 voices, even though the usual number is just one or two. Thus, usually the maximum number of voices is two per staff, so that two staves make it up to 4 possible voices. However, whenever a single staff has two voices it is almost sure that the other one will have only one, so it can be stated a maximum number of 3 simultaneous voices at a time.

Anyway, the final idea and desire is to distinguish voices and have the possibility of comparing only one, two, three or even four together at a time.

Part VII:
General Requirements
Analysis and First Solutions

1 Introduction

The aim of this part of the document is to analyze the constraints given in the User Requirements Definition and offer a first approximation to their solution.

2 File Format

Nowadays there are several file formats to store musical information, from WAV, MP3 or WMA to SMF, MusicXML [Recordare] or SMDL [ISO/IEC]. However, the main difference among the first three formats and the second three formats is that the first ones do not maintain the actual information about the performance's staves. This makes necessary to the user to convert his or her query staves to one of these binary file formats.

Therefore, only file formats maintaining information about the actual staves must be considered. Moreover, it would be desirable to allow a widely accepted file format, whilst to allow more than one file format would also be great.

MusicXML format might be considered as the best choice, since a parser development would be an easy to achieve task thanks to frameworks like Microsoft's DOM. However, it has a particular drawback that makes it unfeasible: the huge amount of information that takes into account. In a MusicXML file are contained both actual musical information and layout information for displaying it. Since the main disadvantage of a XML format is that it might be unfeasible because of the amount of useless information it contains (markup data), and even if this format has useless data information (layout information), MusicXML is clearly discarded.

In order to offer a more precise look at this problem, Listing 2.1 shows an example of a MusicXML file.

```
<?xml version="1.1" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE score-partwise PUBLIC
"-//Recordare//DTD MusicXML 1.1 Partwise//EN"
"http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
  <part-list>
    <score-part id="P1">
      <part-name>Music</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>1</divisions>
        <key>
          <fifths>0</fifths>
        </key>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
        <clef>
          <sign>G</sign>
          <line>2</line>
        </clef>
      </attributes>
      <note>
        <pitch>
          <step>C</step>
          <octave>4</octave>
        </pitch>
        <duration>4</duration>
        <type>whole</type>
      </note>
    </measure>
  </part>
</score-partwise>
```



```
</note>  
</measure>  
</part>  
</score-partwise>
```

Listing 2.1 MusicXML example

This example consists only in a staff like the one in Figure 2.1, where appears just a C note in the 4th octave. As it can be seen on Listing 2.1, every time a note needs to be placed on the staff, a whole `<note>` element must be put into the MusicXML file along with its sub-elements, increasing the file size up to an unfeasible threshold.



Figure 2.1 Music XML example

On the other hand, SMF is a widely deployed standard since many years ago, and does not sin of having layout information. Every musical program will for sure allow working with SMF format, so it will be the chosen one for the system. The main drawback of SMF is that a parser construction will be harder to perform since it is a binary format.

However, as the main advantage of being a standard since many years ago, it has a lot of frameworks, utilities, tools and whatever could be wished to handle SMF files. Indeed, Microsoft Windows itself has a DLL (winmm.dll) that offers all the needed functionality to play SMF files and use MIDI devices. Nevertheless, the system must be developed under the .net technology, so Interop services will be needed in case of using one of these functionalities.

Nonetheless, Stephen Toub, from the Microsoft MSDN Magazine [Toub], has a framework built under C# and supported over the winmm.dll library that offers an easy way to handle SMF files, with a static information model to access the data. This framework is called MIDI lib and is currently on its version 2.0.4 [GotDotNet].

Therefore, only SMF 1.0 files will be accepted with its two possible file extensions: .midi and .mid.

2.1 The MIDI lib 2.0.4

Once it has been decided to use the MIDI lib 2.04, the first and most important step is to understand the static information model it uses to handle SMF files, and it will be done by means of UML class diagrams.

The first diagram in Figure 2.2 shows the main classes that define a SMF file: a `MidiSequence` containing several `MidiTracks`, which use a `MidiEventCollection` in order to store every `MidiEvent` within the track.

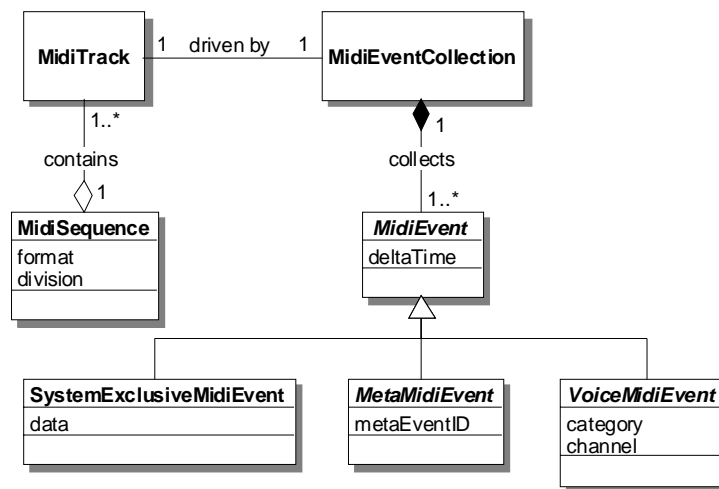


Figure 2.2 MIDI lib static information model (part I)

Figure 2.3 depicts classes used to handle MIDI meta-events, basically by creating a particular class for every case and making it a child of **MetaMidiEvent**.

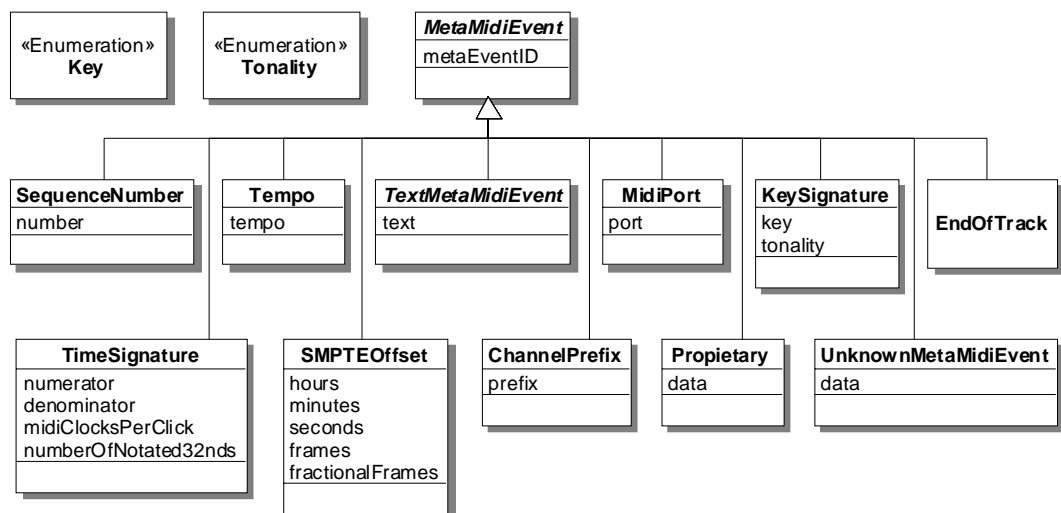


Figure 2.3 MIDI lib static information model (part II)

The most important meta-events here are the **KeySignature** (together with the **Key** and **Tonality** enumerations), **Tempo** and **TimeSignature**. However, the rest of classes might be used to store information about authors and so on, so that more precise information can be offered to the user in a retrieval process.

Even deeper in meta-events, Figure 2.4 shows classes used to handle text MIDI meta-events, such as Copyright, lyrics, Instruments and so on. As well as with the previous elements in Figure 2.3, this information might be used to offer more precise information to the user.

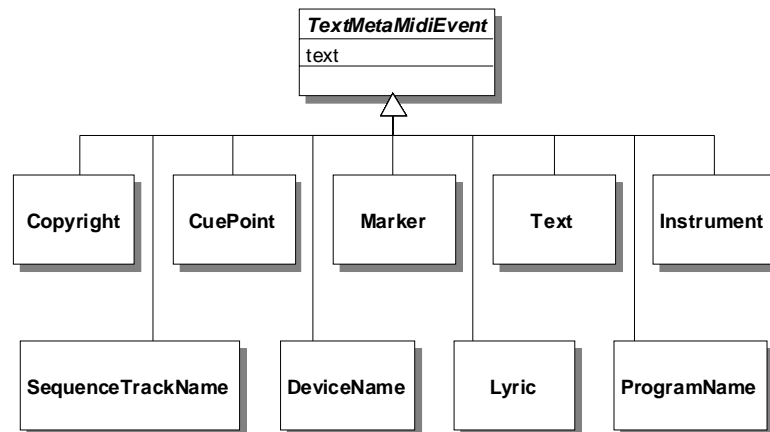


Figure 2.4 MIDI lib static information model (part III)

And finally, Figure 2.5 depicts every class needed to handle voice events. Moreover, several enumerations are added for having more accurate and precise information about each event.

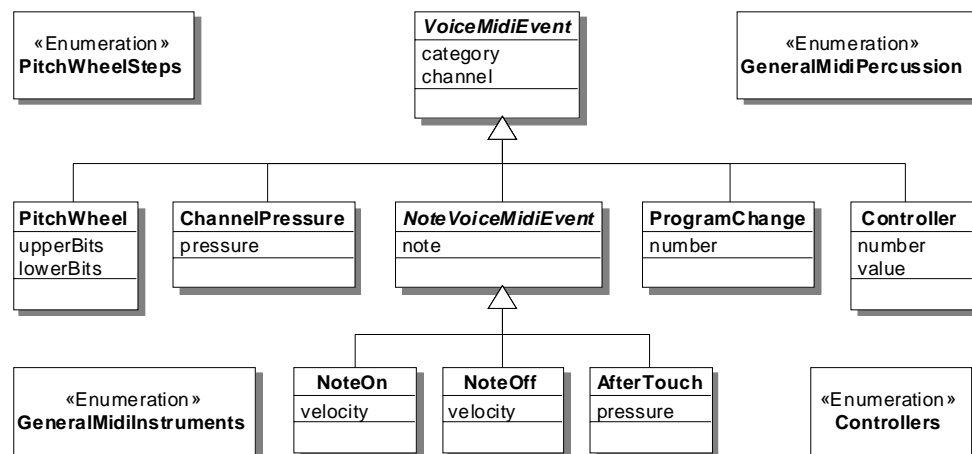


Figure 2.5 MIDI lib static information model (part IV)

Moreover, the MIDI lib offers some additional classes that might be used to play and generate SMF files. Therefore, it would be possible to generate SMF data from a retrieval result in RSHF and then play it so that the user can compare results without the need of retrieving the whole MIDI file linked to the retrieved result.

2.2 SMF Format

Only SMF formats 0 and 1 are going to be considered since format 2 allows asynchronous and sequentially independent tracks. Moreover, as Section 2.2 explains in the document's Part X, only files with a single staff will be allowed with its voices in different tracks. Therefore, to allow a file with format 2 does not make any sense.

3 Vertical Constraints

In the User Requirements Definition were introduced some problems related to the height of each note. This section now analyzes these vertical problems and their possible solution.

3.1 Octave Equivalence

As stated in the Requirements Definition, octave numbers must be ignored as a whole, considering as equal two melodies with the same notes but in different octaves. Therefore, the octave number will be ignored in a first approach.

3.2 Grade Equality

It is really usual to make a different version of a given song or performance and, also usual, to make it changing the key signature. For instance, the original key signature of Knocking on Heavens Door from Bob Dylan is major G, as well as the version from Eric Clapton. However, the version from Gun's n' Roses is in minor G, and the one from Avril Lavigne is in major C.

Therefore, actual notes can not be compared. Instead, the grades of the tonality can be compared to decide whether two songs are the same or not. Thus, actual notes are going to be discarded.

Grades might be compared directly with the number of grade. However, it is not enough in cases like Figure 3.1 where the second note is the same and hence also the grade. But they are in different octaves, breaking thus the equality.



Figure 3.1 Grade equality

Therefore, the number of octave must be taken into account for the comparison even though in Section 3.1 it was decided to ignore them as a whole. Thus, some other technique must be used to avoid the Octave Equivalence problem since the comparison must be in a way like $\{V_4, II_5, VII_4\}$ against $\{V_4, II_4, VII_4\}$, where X_Y means the Xth grade in the Yth octave.

However, the key signature is not mandatory in a MIDI file as it was explained in Section 4.2 of document's Part IV, so it must be figured out. Nevertheless, it is difficult to figure out the key signature of a song just from its notes, and in some cases it is even impossible due to ambiguity issues (see

the Note Equality problem), so another solution must be applied to the Grade Equality problem

3.3 Note Equality

At this point it is clear that octaves must be ignored as a whole but taken into account within a particular melody. Moreover, actual notes can not be compared, conceding place to the tonality grades.

However, the Note Equality problem presented in the Requirements Definition does not allow the comparison only by means of scale grades. Thus, the only way to compare two melodies is to compare the intervals between notes. For instance, in the first bar on Figure 3.1 it would be said that there is an interval of +4 notes (from G4 to D5) and then another interval of -2 notes (from D5 to B4). On the right staff it would be said that there is an interval of -3 notes (from G4 to D4) and then of +5 notes (from D4 to B4).

The problem is that in some cases melodies have non-natural notes, so that the interval can not be evaluated as +3 or +4 notes, but maybe +3.5. Therefore, intervals must be measured by means of number of tones or even semitones like Figure 3.2 depicts. In addition, an interval of 1 note might be 2 notes in depending on the tonality of the whole song.



Figure 3.2 Interval measurement

With this method, the Octave Equivalence problem is solved since both melodies will have the same intervals. The Grade Equality problem is also solved since the same progression of grades will have the same intervals between notes, independently of the tonality. Even more, the Note Equality problem is also solved since two melodies with the same notes will have the same intervals even though they have different key signatures.

Thus, this method of comparing melodies by means of interval progression solves every vertical problem proposed in the User Requirements Definition.

3.4 Chord Recognition

The issue about the chord recognition and comparison will be treated in detail in the Part VIII of the document, where the mathematical model is explained. Trying to give a solution right now, without the needed mathematical background would be worthless.

4 Horizontal Constraints

Some other constraints affect to the horizontal meaning of a composition. The Definition of the User Requirements stated some horizontal problems and constraints that the system must comply with, and they are analyzed and solved in the following sections.

4.1 Time Signature Equivalence

When Musical Theory was introduced, it was stated that the only actual difference among two different time signatures is how strong certain notes must be played. Therefore, time signatures will be simply ignored.

However, it might be good to know where a bar finishes in order to split the performance in several parts or artifacts, or simply to analyze melodies easily and figure out the riffs. Anyway, the use of the time signature will only affect to the preliminary analysis of the song and not anymore. Moreover, it is not mandatory to include a time signature in a MIDI file as Section 4.2 says in the Part IV of the document. Therefore, the system must not depend in a hundred percent on the time signature.

4.2 Tempo Equality

In a first approach to this problem, the only way to solve it is to consider actual time instead of musical time. For instance, in Figure 4.1 the first note can be considered to start at time 0 ms. Since a minute has 60 crotchets, a single crotchet will last $\frac{60s/min \cdot 1000ms/s}{60crotchet/min} = 1000ms/crotchet$. Thus, a quaver will last 500 ms.



Figure 4.1 Tempo Equality

Therefore, the second quaver will start at 500ms, the third one will start at 1000ms and the fourth one at 1500ms. The fifth one will start at 2000ms and will last until 3000ms because it is a crotchet.

Considering this measurement for the time, the Tempo Equality problem is easily solved, so it can be considered as enough so far.

loss of information as it ignores the note length, so any other additional technique must be proposed to avoid this loss.

5 Voice Constraints

The first step to solve the Voice Problem is to recognize voices. In a MIDI file with voices, no additional information is added in order to separate them, but there is just a bunch of notes ordered by time. Therefore, an algorithm must be developed for this first step.

5.1 Approaches to the Voice Separation

The main challenge when separating voices is to differentiate among chords and different voices when notes have same duration and onset time. Many algorithms have been proposed to achieve a solution for this problem given as input a stream of notes. In the following subsections, these approaches are described [Lebel, 2006].

5.1.1 Split Point Approach

The simplest solution consists on splitting the range of all possible pitch values into disjoint sets so that each set corresponds to a single voice. Although this algorithm is extremely simple, it does not assign necessarily the correct voice to each note since assumes a fixed number of voices and that voices do not share any pitch value. Moreover, this algorithm does not support chords.

5.1.2 Rule-based Approach

Another proposed solution is to take advantage of the voice-leading rules used by the actual composers, such as limiting the number of voices, prefer small intervals between successive notes or avoid overlapping voices. The problem with this approach is that the number of rules can increase unfeasibly and they might be not widely accepted. Moreover, some rules such as the fixed number of voices might be proper just for some parts of the performance, yielding to a erroneous result.

5.1.3 Local Optimization Approach

In this case, the proposed algorithm uses a heuristic algorithm. The idea is to apply the iterative process to the input stream, splitting it into small slices containing overlapping notes and then assigning these notes to a single voice by using a randomized local search that optimizes a cost function [Hoos, 2002].

Even tough this solution does not find sometimes the correct voice separation, offers a reasonable one. In addition, the major advantage is that it recognizes chords properly, and this is a really valuable feature of the algorithm.

5.1.4 Contig Mapping Approach

The main difference among this algorithm and the local optimization is that it aims to provide the correct separation rather than a proper one for transcription by running an algorithm similar to those used with DNA processing. As [Chew, 2005] describes, the input stream is segmented into collections of overlapping pieces and then adjacent contigs are connected by using a shortest distance method.

5.1.5 Predicate Approach

As the Contig Mapping approach does, the Predicate approach aims to find the correct separation for the performance's voices. The algorithm is implemented with a learned decision tree to decide whether two notes are or not in the same voice, delegating in another algorithm the task of assigning notes to voices by considering many aspects concerning distance and rhythm. As explained in [Kirlin, 2005], the algorithm does not have a fixed number of voices. However, and has many learned algorithms sin of, it never produces error-free results.

5.2 The Kilian-Hoos Algorithm

So far, no algorithm has been proposed for the Voice Separation problem producing a perfect result. Thus, the Local Optimization approach has been chosen since it correctly detects chords, and this is a really valuable feature since songs are made up by them and to acquire a decent set of songs free of chords is almost impossible.

5.2.1 Preliminaries

The input to the algorithm is given as a list of notes sorted by onset time. Note number i -th in the list is represented by a vector $m_i = (o_i, d_i, p_i)$, where o_i , d_i , and p_i are, respectively, the onset time, the duration and the note's pitch. These properties are also acceded by $\text{onset}(m_i)$, $\text{duration}(m_i)$ and $\text{pitch}(m_i)$. Moreover, two integers, v_i and c_i , are also linked to a note m_i in order to denote the voice and chord that the note is currently associated with. Likewise, these properties are also acceded by $\text{voice}(m_i)$ and $\text{chord}(m_i)$. In addition, a function is utilized to indicate the end point of a note m_i .

$$\text{offset}(m_i) = \text{onset}(m_i) + \text{duration}(m_i) \quad [5.1]$$

Moreover, some relations among notes are defined:

$$\begin{aligned} m_i \leq m_j &:= \text{onset}(m_i) \leq \text{onset}(m_j) \\ m_i = m_j &:= \text{onset}(m_i) = \text{onset}(m_j) \end{aligned} \quad [5.2]$$

Furthermore, a function $\text{overlap}(m_i, m_j)$ tells whether notes m_i and m_j overlap in time:

$$\begin{aligned} \text{overlap}(m_i, m_j) := & \text{onset}(m_i) \leq \text{onset}(m_j) \leq \text{offset}(m_i) \vee \\ & \text{onset}(m_j) \leq \text{onset}(m_i) \leq \text{offset}(m_j) \end{aligned} \quad [5.3]$$

By means of these definitions, the input of the algorithm is formally written as:

$$M = (m_1, \dots, m_l) \mid \forall i \in \{1, \dots, l-1\} : m_i \leq m_{i+1} \quad [5.4]$$

In the output voice separation, two notes with the same onset time can only be in different voices or be in the same one but grouped within a chord:

$$\begin{aligned} m_i = m_j \Rightarrow & \text{voice}(m_i) \neq \text{voice}(m_j) \vee \\ & (\text{voice}(m_i) = \text{voice}(m_j) \wedge \text{chord}(m_i) = \text{chord}(m_j)) \end{aligned} \quad [5.5]$$

For this algorithm, in case the input is quantized, chords are restricted only to notes with the same onset time. If input is not quantized overlapping notes with different onset times should be allowed to form a chord. Thus, in the original implementation, chords with small overlaps or other inaccuracies are eliminated in a preprocessing phase.

5.2.2 Input Splitting

The first step in the algorithm is to split the input M into slices y_i of consecutive overlapping notes (m_k, \dots, m_{k+p}) so that there is an overlap among any pair of notes within each slice and that between two consecutive slices, say y_i and y_{i+1} , there are at least two notes that do not overlap, as Figure 5.1 depicts. Therefore, every note in y_i but the one with the smallest offset time may overlap with notes in y_{i+1} .

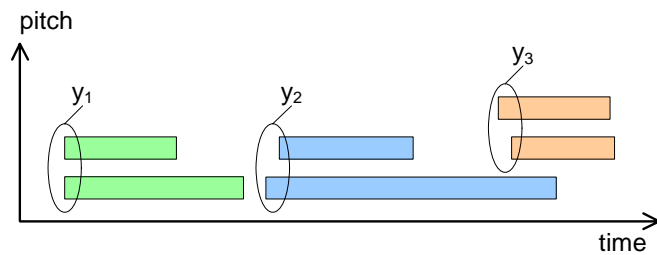


Figure 5.1 Partitioning a piece into slices

Formally, the splitting of M is defined as the set B containing indexes of notes in M that become the first notes of the slices y_1, \dots, y_n :

$$\begin{aligned} B := & \{b_1, \dots, b_n \mid \forall i \in \{1, \dots, n\} : b_i \in \mathbb{N} \wedge \\ & b_1 = 1 \wedge b_n \leq l \wedge \\ & (\forall q, r \in \{b_i, \dots, b_{i+1} - 1\} : \text{overlap}(m_q, m_r)) \wedge \\ & (\neg \exists s \geq b_{i+1} \mid \forall q \in \{b_i, \dots, b_{i+1} - 1\} : \text{overlap}(m_q, m_s))\} \end{aligned} \quad [5.6]$$

Thus, and based on B, the definition of the set of slices y_i is as follows:

$$Y := (y_1, \dots, y_n) \mid y_i = \{m_{b_i}, \dots, m_{b_{i+1}-1}\} \quad [5.7]$$

and splits the input M into slices as formula [5.5] describes.

$$M = (\underbrace{m_1, \dots, m_{b_2-1}}_{y_1}, \underbrace{m_{b_2}, \dots, m_{b_3-1}}_{y_2}, \dots, \underbrace{m_{b_n}, \dots, m_l}_{y_n}) \quad [5.8]$$

From now on, the voice separation for a slice $y_i = (m_{b_i}, \dots, m_{b_{i+1}-1})$ is denoted by $S(y_i)$, and even just S_i for simplicity. The vector S_i is made up by the voice and chord that the q -th note of slice y_i belongs to:

$$S_i = (s_{i1}, \dots, s_{ip}) \mid \forall q \in \{1, \dots, p\} : s_{iq} = (\text{voice}(m_{b_i+q-1}), \text{chord}(m_{b_i+q-1})) \quad [5.9]$$

In Addition, the set of all possible combinations of $S(y_i)$ for a slice y_i is denoted by S_i^* . Thus, any full voice separation S for the input is made up by separations of slices y_i so that $S = (S_1, \dots, S_n)$, where the set of all possible combinations for S and a given input M is S^* .

For a given slice y_i , the number of possible combinations (size of set S_i^*) depends on the number of notes $|y_i|$ and on the maximum number of voices in the desired output, say $n\text{Voices}$. In particular, when any subset of y_i can be combined into a chord, there are at least $n\text{Voices}^{|y_i|}$ possible voice separations. Therefore, the number of possible separations for M is exponential in $|M|$. This means that the algorithm might be unfeasible for most of the inputs. Therefore, a heuristic function is used in an iterative process that constructs the voice separation based on a stochastic local search that optimizes the partition.

The idea behind the algorithm is to construct a voice separation for M based on local optimized separations for each slice. This optimization is based on a cost function C assessing the quality of a separation S_i given the previous separations (S_1, \dots, S_{i-1}) .

Listing 5.1 shows the outline of the algorithm for an unquantized input. In case the input is quantized, the last two steps might be avoided.

```

procedure voiceSeparation(M, k)
  input:
    sorted list of notes M
    maximal number of voices k
  output:
    voice separation S
  begin
    segment M into slices  $y_1, \dots, y_n$ 
    S := ();
    for i := 1 to n do
       $S_i := \text{separateSlice}(y_i, S)$ 
      S := S +  $S_i$ 
      eliminate overlaps within voices of S
      regularize chords where required
    end
  end

```

Listing 5.1 Outline of the Kilian-Hoos algorithm for unquantized input data

5.2.3 The Cost Function

The cost function assessing the quality of a voice separation in slice S_i with previous separation S_1, \dots, S_{i-1} is the weighted sum of several features:

$$C(S_i, S) = K_{\text{pitch}} C_{\text{pitch}}(S_i, S) + K_{\text{gap}} C_{\text{gap}}(S_i, S) + K_{\text{chord}} C_{\text{chord}}(S_i) + K_{\text{overlap}} C_{\text{overlap}}(S_i, S) \quad [5.10]$$

where C_{pitch} penalizes large pitch intervals between successive notes, C_{gap} penalizes large gaps (rests) between successive notes, C_{chord} penalizes large pitch distances between the highest and the lowest note and C_{overlap} penalizes overlap between successive notes in the same voice. On the other hand, K_{pitch} , K_{gap} , K_{chord} and K_{overlap} are constants to fine tuning the result by giving different weights to each feature.

Pitch Distance Penalty C_{pitch}

The pitch distance penalty increases with each interval between two successive notes in the same voice, giving to the first one a fixed penalty for starting a new voice. In some cases, like with melodies with short sequences of large pitch intervals, a lookback mechanism can be used to calculate the pitch interval not only with the last one, but with the last l notes on the voice.

In order to compare the pitch interval between a note m_j and a pitch p_i a function $\text{cPitch}(m_j, p_i)$ is defined so that it returns $\text{pitch}(m_j)$ in case m_j does not belong to a chord, and the pitch of the note within the chord closest in pitch to p_i otherwise.

$$\text{cPitch}(m_j, p_i) = \begin{cases} \text{pitch}(m_j) & \neg \exists \text{chord}(m_j) \\ \text{pitch}(m_c) \mid m_c \in \text{chord}(m_j) & \exists \text{chord}(m_j) \\ \neg \exists m_k \in \text{chord}(m_j) \mid | \text{pitch}(m_k) - p_i | < | \text{pitch}(m_c) - p_i | & \exists \text{chord}(m_j) \end{cases} \quad [5.11]$$

Thus, if no lookback is used, the pitch distance among a voice v and a pitch value p_i is:

$$\text{cPitch}(v, p_i) = \text{cPitch}(\text{LOnset}(v), p_i) \quad [5.12]$$

where $\text{LOnset}(v)$ is the note within v with the latest onset time:

$$\text{LOnset}(v) = m_{\text{latest}} \in v \mid \forall m_i \in v : \text{onset}(m_i) \leq \text{onset}(m_{\text{latest}}) \quad [5.13]$$

Otherwise, if lookback is used, the algorithm in Listing 5.2 is used to calculate the pitch distance value.

```
function cPitch(v, l, p_j)
input:
voice index v
```

```

lookback size 1
pitch pj of note j
output:
average pitch p of voice v for comparison to pj
begin
prevNote := prev(lonset(v), pj)
p := cPitch(lonset(v), pj)
i := 1
while i ≤ 1 do
p := 0.8 · p + 0.2 · cPitch(prevNote, pj)
prevNote := prev(prevNote, pj)
i := i + 1
end
return p
end

```

Listing 5.2 Pitch calculation for voice v with pitchlookback > 0

On that algorithm, function $\text{prev}(m_j, p_i)$ returns the note directly preceding m_j within the same voice and not belonging to the same chord as m_j . In case the preceding figure is a chord, it will be returned the note within the chord that is closest in pitch to p_i . Constant values 0.8 and 0.2 were empirically found by the authors as the best ones.

$$\text{prev}(m_j, p_i) = \begin{cases} m_p \in \text{voice}(m_j) \mid \text{onset}(m_p) < \text{onset}(m_j) \wedge \\ \neg \exists m_k \in \text{voice}(m_j) \mid \text{onset}(m_p) < \text{onset}(m_k) < \text{onset}(m_j) & \neg \exists \text{chord}(m_p) \\ m_c \in \text{voice}(m_j) \mid m_j \notin \text{chord}(m_c) \wedge \\ \text{onset}(m_c) < \text{onset}(m_j) \wedge & [5.14] \\ \neg \exists m_k \in \text{chord}(m_c) \mid \text{pitch}(m_k) - p_i < \text{pitch}(m_c) - p_i \wedge & \exists \text{chord}(m_c) \\ \neg \exists m_k \in \text{voice}(m_j) \mid m_k \notin \text{chord}(m_c) \wedge \\ \text{onset}(m_c) < \text{onset}(m_k) < \text{onset}(m_j) \end{cases}$$

Therefore, the pitch distance penalty C_{pitch} for a voice v is calculated as shown in Listing 5.3:

```

function Cpitch(Si, S, v)
input:
slice separation Si
separation S for previous slices
voice index v
output:
pitch distance pVD
begin
mp := first note in yi // m de b de i
prevNote := prev(mp, pitch(mp))
pVD := 0
foreach note mj in Si do
if voice(mj) = v then
pDist := |cPitch(prevNote, pitch(mj)) - pitch(mj)| / 128
pVD := pVD + (1 - pVD) · pDist
if chord(prevNote) ≠ chord(mj) then
prevNote := mj
end
end
end
return pVD

```

```
end
```

Listing 5.3 Calculation of C_{pitch} for a single voice v in S_i

and the final C_{pitch} function for a separation S_i given the previous separations S is shown in Listing 5.4:

```
function Cpitch(Si, S,)
input:
    slice separation Si
    separation S for previous slices
output:
    pitch distance penalty pD
begin
    pD := 0
    foreach voice v used in Si do
        pD := pD + (1 - pD) · cpitch(Si, S, v)
    end
    return pD
end
```

Listing 5.4 Calculation of pitch distance penalty C_{pitch} for slice S_i given previous separation S

Gap Distance Penalty C_{gap}

Some studies quoted in [Hoos, 2002] show that listeners prefer voices with few and short distances, and this axiom is taken into account for the C_{gap} function, as Listing 5.5 shows.

```
function Cgap(Si, S)
input:
    slice separation Si
    separation S for previous slices
output:
    gap distance penalty gD
begin
    gD := 0
    cNotes := 0
    foreach voice v used in Si do
        m := earliest note in Si with voice(m) = v
        gD := gD + cGap(m, v)
        cnotes := cnotes + 1
    end
    gD := gD / cnotes
    return gD
end
```

Listing 5.5 Calculation of gap distance penalty C_{gap} for slice S_i given previous separations S

Therefore, a gap penalty arises whenever a note is added to a voice in such a way that it introduces a gap. Moreover, the bigger the gap duration is, the bigger the penalty is as well. In case a note starts a new voice, the gap distance between it and the beginning of the composition m_1 is also penalized.

Function $C_{gap}(m_i, v)$ among a note m_i and a voice v penalizes the appending of note m_i to voice v by returning the gap distance introduced by m_i divided by the maximal gap length that m_i would introduce to any voice, returning thus a value between 0 and 1:

$$C_{\text{gap}}(m_i, v) = \frac{\text{onset}(m_i) - \text{offset}(\text{IOnset}(v))}{\text{onset}(m_i) - \text{offset}(\text{IOnset}(v_{\text{max}}))} \mid \forall q \in \{1, \dots, n\text{Voices}\} : \quad [5.15]$$

$$\text{onset}(m_i) - \text{offset}(\text{IOnset}(v_{\text{max}})) \geq \text{onset}(m_i) - \text{offset}(\text{IOnset}(v_q))$$

Chord Distance Penalty C_{chord}

When notes are combined within a chord, small ranges are preferred. On the other hand, it would be expected that all notes belonging to a chord have the same onset time and length in case of quantized data. Therefore, the chord distance penalty increases with the range of a chord, durations of notes and differences between onset times. Listing 5.6 outlines the steps of the function.

```
function Cchord(Si)
  input:
    slice separation Si
  output:
    chord distance penalty CD
begin
  CD := 0
  foreach chord c in Si do
    p := pDuration(c) + (1 - pDuration(c)) · pRange(c)
    p := p + (1 - p) · pOnset(c)
    CD := CD + (1 - CD) · p
  end
  return CD
end
```

Listing 5.6 Calculation of chord distance penalty C_{chord} for slice S_i

The range penalty $p\text{Range}$ for a given chord c is defined as the difference, in semitones, between the highest and the lowest notes in the chord:

$$p\text{Range}(c) = \min \left\{ \frac{p_{\text{highest}} - p_{\text{lowest}}}{24}, 1 \right\} \mid$$

$$c = \{m_1, \dots, m_n\} \wedge \quad [5.16]$$

$$p_{\text{highest}} = \max \{ \text{pitch}(m_1), \dots, \text{pitch}(m_n) \} \wedge$$

$$p_{\text{lowest}} = \min \{ \text{pitch}(m_1), \dots, \text{pitch}(m_n) \}$$

but in a range from 0 to 1, where ranges of more than two octaves receive the same penalty.

Likewise, the $p\text{Duration}$ penalty depends on the shortest and latest notes in the chord c . Thus, the function returns a value from 0 to 1, where a value of zero means that all durations are the same.

$$\begin{aligned}
 pDuration(c) &= 1 - \frac{d_{\text{shortest}}}{d_{\text{longest}}} \\
 c &= \{m_1, \dots, m_n\} \wedge \\
 d_{\text{shortest}} &= \max\{\text{duration}(m_1), \dots, \text{duration}(m_n)\} \\
 d_{\text{longest}} &= \min\{\text{duration}(m_1), \dots, \text{duration}(m_n)\}
 \end{aligned}
 \tag{5.17}$$

Lately, the pOnset penalty depends on the earliest and latest notes in c , as well as on the longest one. Thus, the function returns a value of 0 when every note within c has the same onset time. Moreover, since every couple of notes in c overlap, the penalty can never be greater than 1.

$$\begin{aligned}
 pOnset(c) &= \frac{o_{\text{latest}} - o_{\text{earliest}}}{d_{\text{longest}}} \\
 c &= \{m_1, \dots, m_n\} \wedge \\
 o_{\text{latest}} &= \max\{\text{onset}(m_1), \dots, \text{onset}(m_n)\} \wedge \\
 o_{\text{earliest}} &= \min\{\text{onset}(m_1), \dots, \text{onset}(m_n)\} \\
 d_{\text{longest}} &= \max\{\text{duration}(m_1), \dots, \text{duration}(m_n)\}
 \end{aligned}
 \tag{5.18}$$

With these three penalty functions, the chord distance penalty calculates a value by combining them in such a way that if one of them is large, the overall penalty will be large as well.

Overlap Distance Penalty C_{overlap}

Depending on the instrument used to play a certain performance, and sometimes also on the style, there can be some overlaps among consecutive notes within a voice (mainly in melodic lines) that can not be avoided with a preprocessing phase. Moreover, in case of a not quantized performance, there will arise, for sure, many overlaps that are not supposed to appear, and they appear due to the fact that the performance is played by a human being.

Therefore, overlaps between notes that do not belong to the same chord are allowed, and function C_{overlap} is introduced in order to penalize the overlapping amount for a given separation S_i :

```

function Coverlap( $S_i$ ,  $S$ )
input:
  slice separation  $S_i$ 
  separation  $S$  for previous slices
output:
  overlap distance penalty  $oD$ 
begin
   $oD := 0$ 
  foreach voice  $v$  used in  $S_i$  do
     $oDist := Coverlap(S_i, S, v)$ 
     $oD := oD + (1 - oD) \cdot oDist$ 
  end
  return  $oD$ 
end

```

Listing 5.7 Calculation of overlap distance penalty C_{overlap} for slice S_i given previous separations S

and delegates on the following function to compute the overlap penalty for a given voice:

```
function Coverlap(Si, S, v)
  input:
    slice separation Si
    separation S for previous slices
    voice v
  output:
    overlap distance penalty ovd
begin
  prevNote := onset (v)
  ovd := 0
  foreach note mj in yi do
    if voice(mj) = v then
      oDist := Coverlap(prevNote, mj)
      ovd := ovd + (1 - ovd) * oDist
      if chord(mj) ≠ chord(prevNote) then
        prevNote := mj
      end
    end
  end
  return ovd
end
```

Listing 5.8 Calculation of overlap distance penalty for a single voice

Thus, a new function C_{overlap} will return the overlap distance between two successive notes in the following way:

$$C_{\text{overlap}}(m_j, m_k) = \begin{cases} 1 - \frac{\text{onset}(m_k) - \text{onset}(m_j)}{\text{duration}(m_j)} & \text{overlap}(m_j, m_k) \\ 0 & \neg \text{overlap}(m_j, m_k) \end{cases} \quad [5.19]$$

returning a value between 0 and 1 as well.

5.2.4 Cost-Optimized Slice Separation

Based on the cost function defined in formula 5.10, and given a separation of slices $S=y_1, \dots, y_{i-1}$, a stochastic local search is used to find out the optimal voice separation S_i for slice y_i .

Starting with an initial separation $S_i := S_i^0$, a randomized iterative process tries to find out the best separation by assigning a note to a different voice, storing the combination that minimizes the cost function. This process is limited to a fixed number of θ steps:

$$\theta = 3 \cdot |y_i| \cdot n\text{Voices} \quad [5.20]$$

As listing 5.9 shows, the algorithm begins with an initial separation S_i^0 , where every note of y_i is assigned to the first voice, grouping into chords those notes with equal onset times. Then, the algorithm moves to a neighbor separation, understanding S_i and S'_i as neighbor separations if they are valid and differ in the voice and/or chord assignment of exactly one note. A separation is considered as valid if and only if any notes with identical onset times within the same voice are combined into a chord.

```
function separateSlice(yi, S)
  input:
    slice yi
    separations S for previous slices
  output:
    optimized selection Siopt
begin
  obtain Si by setting all notes of yi to voice 0 and
  combining all notes with equal onset times into chords
  Siopt := Si
  noImpr := 0
  while noImpr < |yi| · nvoices · 3 do
    with probability 0.8 do
      Si := neighbour Si' of Si with minimal cost C(S', S)
    otherwise
      Si := randomly selected neighbour of Si
    end
    if C(Si, S) < C(Siopt, S) then
      Siopt := Si
      noImpr := 0
    else
      noImpr := noImpr + 1
    end
  end
  return Siopt
end
```

Listing 5.9 Randomized iterative algorithm for finding a cost-optimized separation of slice y_i

The actual next step is chosen randomly: with a fixed probability of 0.8 (chosen by the authors as the best for empirical results) the neighbor separation with minimal cost is selected; otherwise, a random neighbor separation is used. That way, the search process will not get stuck into local minima of the cost function.

Part VIII:

The Mathematical Approach

1 Introduction

This part of the document provides a detailed description of the mathematical approach that is finally applied to the system. In particular, the mathematics subfield of numerical analysis is used to model the musical files as functions that define curves in several dimensions. After this modeling is performed, the curve is split into several pieces that will later correspond to relationships between artifacts.

The main advantage of this solution is that it is efficient and solves all the general requirements shown in Section 3 of the document's Part VI.

2 Preliminaries

Some initial work must be done upon the input sequence of notes before applying the mathematical model to it. This section explains this process.

2.1 Domain Normalization

The first step in this approach is to distribute notes within a single staff maintaining a distance between two successive notes that is a multiple of the minimum considered duration for a single note. For instance, if the shortest note is a demisemiquaver, a distance of length 1 unit can be considered as the minimum distance for convenience.

Therefore, the following staff in Figure 2.1



Figure 2.1 Note distribution (part I)

will distribute its notes as Figure 2.2 depicts, where every gap between two successive dashed lines has a length of 1 unit (i. e. a demisemiquaver).

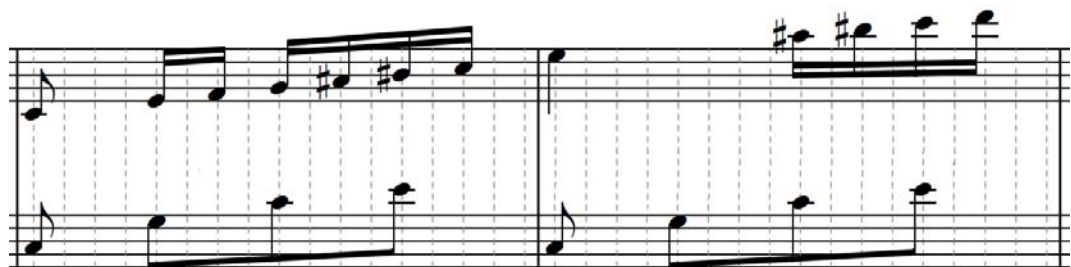


Figure 2.2 Note distribution (part II)

Thus, a crotchet will have a separation of length 8 units with its following note, and a quaver will have a separation of 4 units.

The next step, once the time-dimension is normalized, is to normalize the pitch-dimension. To do so, the pitch range will be considered as $[0, 127]$, since the MIDI specification allows only 128 possible values for a note's pitch. Using a musical notation, these values range from C0 until G10.

It is important to note that in this new representation system, accidentals are not needed, since two notes that share the same height are

committed to have the same pitch. Therefore, if a certain note has a height h , its sharp note will have height $h+1$, whilst the flat note will have $h-1$.

Figure 2.3 depicts the same performance in Figure 2.1, but with normalized time and pitch dimensions. Note also that every note is depicted with a filled circle, avoiding stems and whatever the characteristic that might make it different. Moreover, barlines are removed since the time signature is no longer useful.

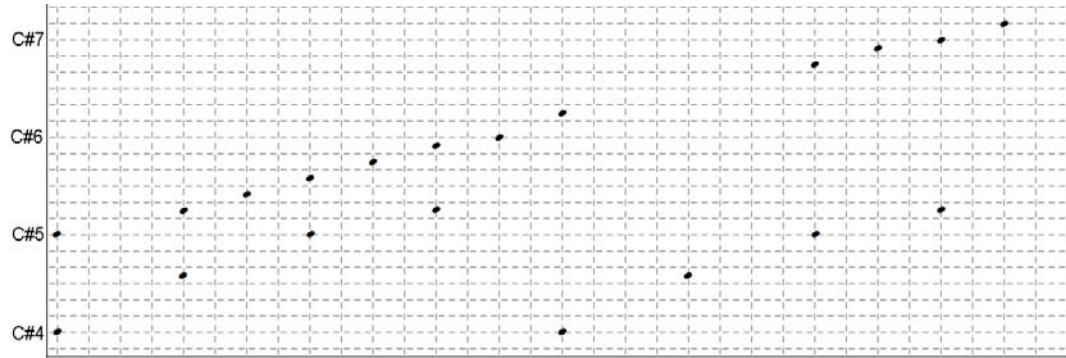


Figure 2.3 Note distribution (part III)

Now that the performance is normalized, an analytical process starts so that the whole piece can be specified by a mathematical function.

2.2 Music As a Mathematical Function

The main point of this mathematical approach is to consider every piece as a function $C_i(t)$ that defines a curve mapping positions in time with the corresponding pitch values.

$$C_i : \mathbb{R} \rightarrow [0,127] \quad [2.1]$$

The best way to obtain this function is to interpolate the points generated after the normalization phase, but distinguishing among voices. For instance, in Figure 2.3 there appear two different voices. Considering that the performance starts at $t=0$, the set of interpolating points (t_i, p_i) for the lower voice is:

$$\{(0,49),(4,56),(8,61),(12,64),(16,49),(20,56),(24,61),(28,64)\} \quad [2.2]$$

From this set of points, it can be defined a function $C_2(t)$ as the interpolating function of that set, which defines the curve that shapes the pitch variation, as the time changes, for the second voice.

From now on, only pieces with a single voice will be considered for simplicity. The case of a piece with several voices will be treated later on, in Section 12 with more detail.

3 Comparing Musical Pieces Described as Mathematical Functions

Now that every musical piece is defined as a function $C(t)$ over time, a way to compare two of them must be elaborated. Let us imagine, for instance, that the following two pieces in Figure 3.1 have to be compared.

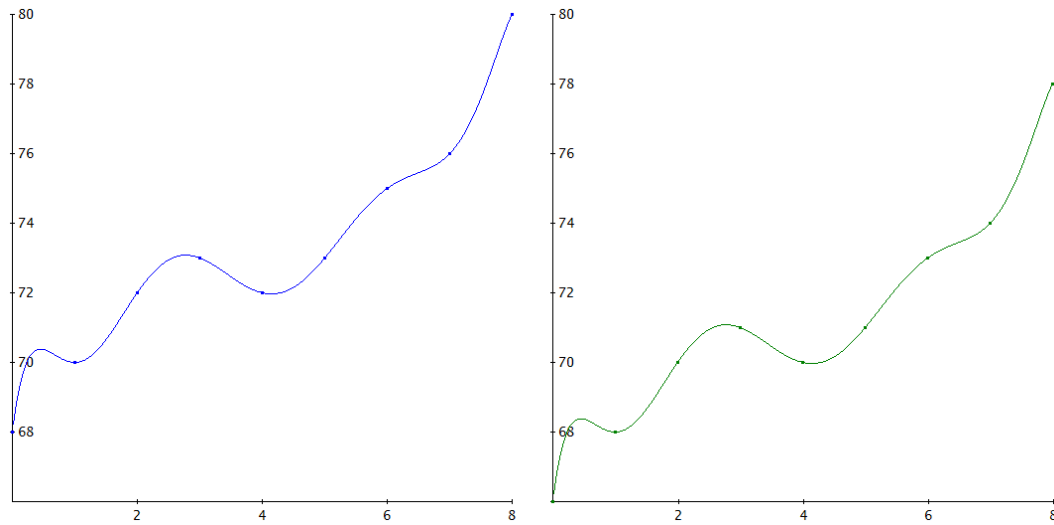


Figure 3.1 Comparison among musical functions

It is pretty clear that both curves have the same shape and, therefore, that they are exactly the same musical piece. However, the only difference among them is that the green one is shifted 2 pitch units downwards (2 semitones). Therefore, if the blue curve represents a piece in the tonality of D, the green one represents the same piece but in the tonality of C. Thus, if both curves are described as a polynomial like

$$C(t) = a_n t^n + a_{n-1} t^{n-1} + \dots + a_1 t + a_0 = \sum_{i=0}^n a_i t^i \quad [3.1]$$

the only difference between them is the constant a_0 that actually defines the height difference. Hence, a good transformation that might be done to every function $C(t)$ is to utilize its first derivative, throwing the original away. Doing so, the blue and green curves in Figure 3.1 will have the same first derivatives, so that the comparison can be done easily and accurately.

Therefore, the previous musical performances will be treated as Figure 3.2 depicts, where it is clear that both functions are exactly the same, even with the same eight.

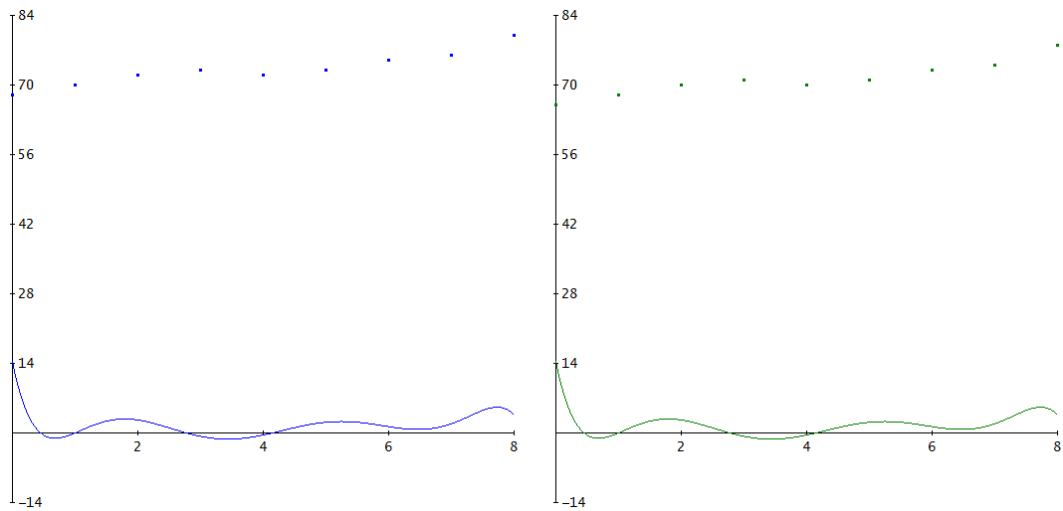


Figure 3.2 Comparison among first derivatives

It is important to note that, even though deriving the original function, no loss of information arises. Indeed, the only loss is the function's pitch eight, which is useless as seen before. If the derivative function $C'(t)$ is integrated, we obtain the original function $C(t)$ plus an integration constant, say K_{int} , which actually measures the tonality difference and can, therefore, be ignored:

$$C'(t) = \frac{d(C'(t))}{dt} = na_nt^{n-1} + (n-1)a_{n-1}t^{n-2} + \dots + 2a_2t + a_1 \quad [3.2]$$

$$\int C'(t)dt = C(t) + \cancel{K_{int}} = a_nt^n + a_{n-1}t^{n-1} + \dots + a_2t^2 + a_1t + \cancel{K_{int}}$$

3.2 Vertical Comparison

Due to the decision of comparing the first derivatives instead of the original functions, every vertical constraint defined in Section 3 of the User Requirements Definition is fulfilled.

First of all, the Octave Equality problem is not a situation because of the use of the first derivative. The integration constant that represents the pitch difference, would be a multiple of 12 in the case of the Octave Equality, since a whole octave has 12 semitones. Thus, since this constant is not taken into account, the problem is not such anymore.

The Grade Equality problem is not an issue anymore because it is the same case as the Octave Equality problem where the integration constant can have whatever the value.

About the Note Equality problem, the initial solution given before was to consider pitch differences between two successive notes instead of their actual pitch values. And this is just the idea behind the first derivative of a function. While in a first approach there were considered entire pitch differences such as +4, 0 or -2, the first derivative leads us to a better and

more accurate comparison with real numbers that, moreover, depends not only on the two successive notes, but also on the adjacent ones. This is an important characteristic that will be discussed later on, in the Part IX.

3.3 Horizontal Comparison

Due to the time-dimension normalization time signatures are not considered once the interpolation phase is achieved, so that the Time Signature Equivalence problem is not an issue.

Both the Figure Equality and the Tempo Equality problem need more considerations in this mathematical model if we want to solve them. The way to compare two melodies that share the same pitch progression but differ in the time-dimension is simply by applying a linear transformation to the curves so that they coincide. For instance, the melody in Figure 3.3 (actually it is already a first derivative), is the same as the one in Figure 3.2 but with a difference in the time-dimension.

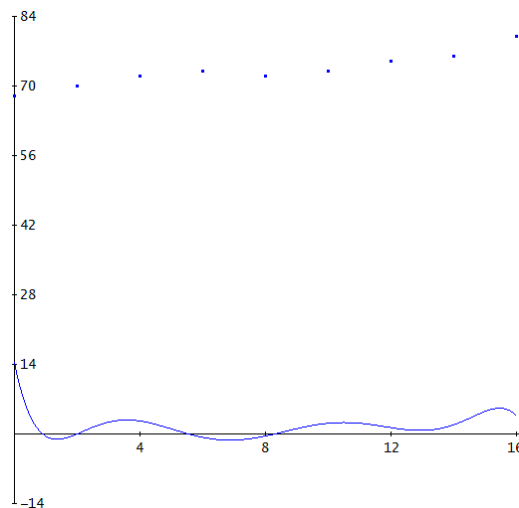


Figure 3.3 Differences in the time-dimension (part I)

If the curve in Figure 3.2 is denoted as $C'(t)$ and the one above in Figure 3.3 is named $D'(t)$, we can indeed say that the following equality is true:

$$D'(t) = C'\left(\frac{t}{2}\right) \quad [3.2]$$

That is to say that if two melodies are alike but differ in the time-dimension, it can be applied a transformation upon the variable t so that, at the end, both melodies will be exactly the same and the comparison should be trivial.

3.4 Piecewise Comparison

Previous section showed us how to compare two identical melodies that suffer the Figure Equality or the Tempo Equality problem. Simply by applying a transformation to the time-dimension we can have identical copies.

However, the same method can not be used with the Partial Equality problem, since a transformation to the time-dimension will modify the whole curve and, hence, the comparison will not be feasible. In order to cope with this problem, the initial approach outlined in Section 4 of the document's Part VII is now considered again.

The idea behind the final approach is to split the curve in pieces that start and end exactly in the point where a note starts or ends. That is to say that whenever a note starts (its onset time) the curve is split in that point and will be split again whenever the note ends (its offset time). By doing so, we will have at the end a piecewise curve where we will be able to apply a linear transformation like before but piece to piece.

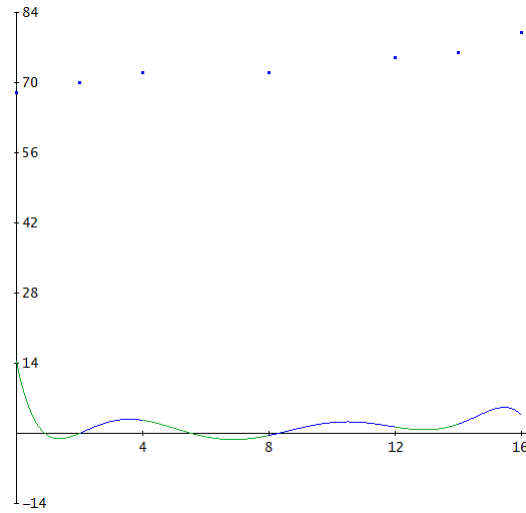


Figure 3.4 Piecewise comparison

Therefore, the curve $C(t)$ and hence its first derivative $C'(t)$ will be a piecewise function defined by intervals. Thus, given a melody defined by a set of notes ordered by onset time

$$M = (m_0, m_1, \dots, m_l) \mid \forall i \in \{0, \dots, l-1\} : \text{onset}(m_i) < \text{onset}(m_{i+1}) \quad [3.4]$$

a function, say $c_i(t)$ is defined for each interval $[\text{onset}(m_i), \text{onset}(m_{i+1})]$, so that the final curve is defined as:

$$C(t) = \begin{cases} c_0(t) & t_0 \leq t \leq t_1 \\ c_1(t) & t_1 \leq t \leq t_2 \\ \vdots & \vdots \\ c_{l-1}(t) & t_{l-1} \leq t \leq t_l \end{cases} \quad [3.5]$$

where $t_i := \text{onset}(m_i)$, notation that will be used from now on for simplicity.

Now, with this piecewise function, we can shrink or stretch each piece by multiplying or dividing the variable t by a certain number so that we can have identical copies for each piece to compare.

Of course, there can be applied some penalizations each time a transformation is needed in the time-dimension so that, at the end, the comparison will not provide a distance of 0 since the melodies are not the same actually.

3.5 How to Perform the Actual Comparison

Let us imagine that we are about to compare two pieces $c'_i(t)$ and $d'_i(t)$ as Figure 3.5 depicts. The best way to compare them is by calculating the area between them in the interval, which corresponds with the filled area in grey between the curves.

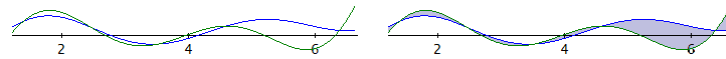


Figure 3.5 Comparing areas

Thus, the area between both curves, in the interval $[\text{onset}(m_i), \text{onset}(m_{i+1})]$ is defined as Δ :

$$\Delta = \int_{t_i}^{t_{i+1}} |c'_i(t) - d'_i(t)| dt \quad [3.6]$$

Since this way to calculate the difference between the curves can lead us to a huge range of possible values, the best to do is to normalize that error in such a way that is proportional to the curves being compared. A good approach is to divide that error by the maximum area defined by the curves:

$$\Delta = \frac{\int_{t_i}^{t_{i+1}} |c'_i(t) - d'_i(t)| dt}{\max \left(\int_{t_i}^{t_{i+1}} |c'_i(t)| dt, \int_{t_i}^{t_{i+1}} |d'_i(t)| dt \right)} \quad [3.7]$$

Moreover, with this division the error will be in the range $[0,1]$ which is the range that the CAKE Engine uses to quantify semantic distances.

4 Basic Polynomial Interpolation

The most typical way to interpolate a set of $l+1$ points in \mathbb{R}^2 is to use the Lagrange Interpolating Polynomial of degree l or less:

$$C(t) = q_0(t)p_0 + \dots + q_l(t)p_l \quad \text{where} \quad [4.1]$$

$$q_i(t) := \prod_{\substack{0 \leq j \leq l \\ j \neq i}} \frac{t - t_j}{t_i - t_j}$$

where $p_i := \text{pitch}(m_i)$, notation that will be used from now on for simplicity.

The best way to obtain the solution is to solve the following $l+1$ equations put in matrix-form and that make up a linear system that has exactly a unique solution:

$$\begin{bmatrix} t_0^l & t_0^{l-1} & \dots & t_0 & 1 \\ t_1^l & t_1^{l-1} & \dots & t_1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ t_l^l & t_l^{l-1} & \dots & t_l & 1 \end{bmatrix} \begin{bmatrix} a_l \\ a_{l-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_l \end{bmatrix} \quad [4.2]$$

The main characteristic of this interpolation method is that the generated curve passes through the interpolating points (the notes). On the other hand, it generates a polynomial of degree l for a given input of $l+1$ notes. It is pretty obvious that this interpolating method is not feasible if we want to compare curves by the area between them as seen on Section 3.5, since it involves to solve a equation of degree l .

Since nowadays there are known direct formulas to calculate the roots of polynomials with a degree up to four, a new method is needed. An approximation method such as the Bisection method or the Newton one might be used. However, this will lead as to an efficiency penalization because this comparison must be done tons of times.

4.1 The Runge's Phenomenon

A drawback of this common interpolation method is that the curve oscillates a lot as long as the number of points, and hence the degree of the polynomial, increases. This is called the Runge's Phenomenon and the main disadvantage is not only that the curve oscillates, but that it does it in such an unpredictable way that the comparison will not make any sense.

For instance, Figure 4.1 shows two melodies that differ only in the pitch of 3 notes (marked up with a red circle). We can see how the interpolating curves differ a lot and how the area between both them increases a lot.

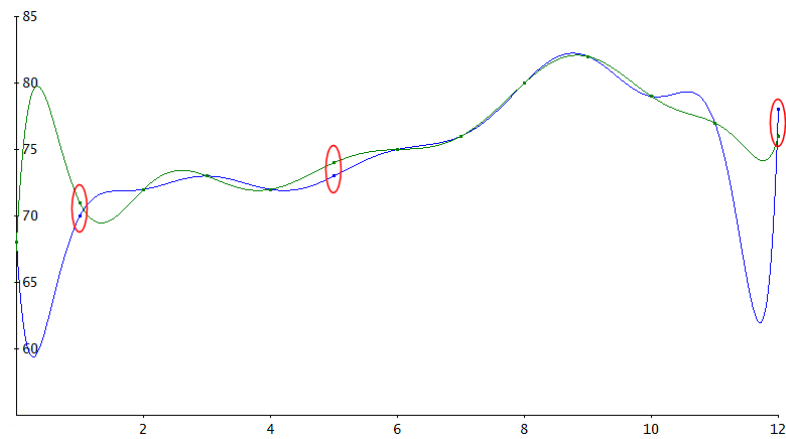


Figure 4.1 The Runge's Phenomenon (part I)

On Figure 4.2 the corresponding first derivatives of the curves above are depicted. These are the actual curves that are going to be compared, and we can see that the result that this comparison would lead to is not good enough. Indeed, we can see how the curves have a huge value for the derivative in the boundaries.

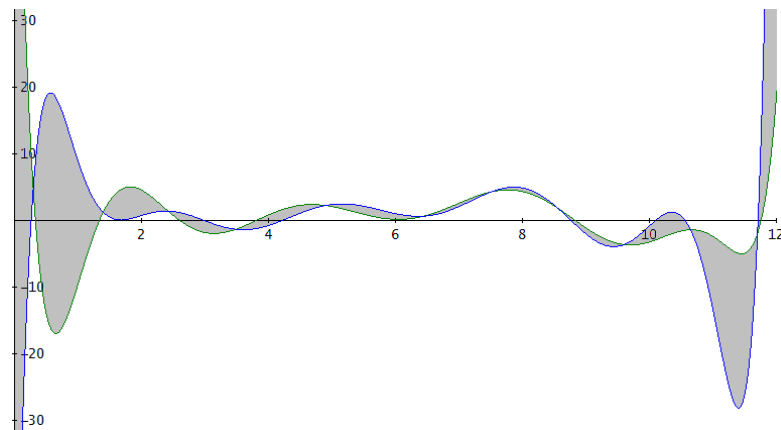


Figure 4.2 The Runge's Phenomenon (part II)

Moreover, we can see how by changing a single note the whole curve will change to some extent. Of course, this is an effect that should be avoided because a change in a single note might mess up the whole comparison.

5 Spline Interpolation

The most feasible solution to the Runge's phenomenon is to interpolate using splines instead of interpolating polynomials such as the Lagrange's one.

At a glance, a spline function is a piecewise interpolating function that is defined with a single function for each interval between two points. On the other hand, the basic polynomial interpolation defined a unique function for the whole set of points. However, now with this interpolation method the curve is already split into pieces as we wanted from Section 3.4.

As the spline function is piecewise it offers a much more close approximation to the points and is much smoother. Another advantage upon the basic interpolation is that the degree of the polynomials that define the curve can be chosen so that it does not have a certain value depending on the number of points. Indeed, the most typical case is to utilize polynomials of degree 3, so that the final curve is so smooth that its second derivative is continuous in the whole domain. Therefore, since the curve is C^2 it implies that it is also curvature continuous.

5.1 Cubic Splines

The most typical case of spline interpolation is the cubic spline, which uses polynomials of degree 3. The same example as in Figure 4.1 is depicted in Figure 5.1 but using cubic splines.

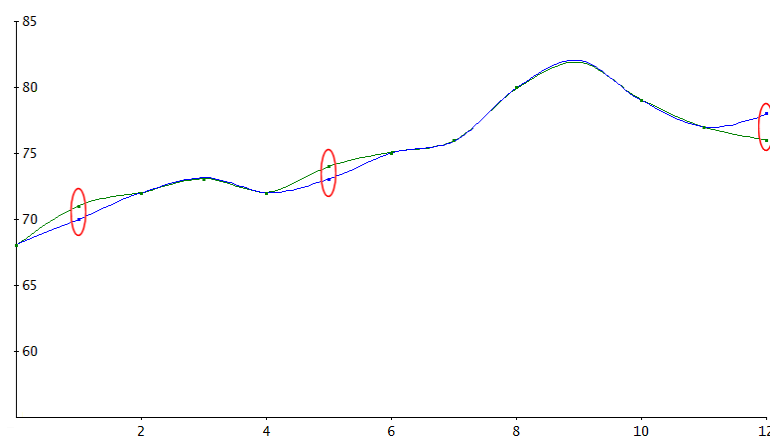


Figure 5.1 Cubic splines

We can see how both curves follow the points much more accurately than with a Lagrange interpolation. Indeed, the curve seems to change only in the interval close to the point where a single note is changed (circled in red). However, even though it seems so the curve changes everywhere. The difference is obviously bigger in those points, but changes all along the curve. Thus, a change in a single note will change the whole curve since the whole curve depends on all the interpolating points (the actual notes).

5.2 Choosing a Cubic Spline

There are a few different kinds of cubic spline curves, depending on the conditions that the spline must comply with. Starting with the same musical input as before

$$M = (m_0, m_1, \dots, m_l) \mid \forall i \in \{0, \dots, l-1\} : \text{onset}(m_i) < \text{onset}(m_{i+1}) \quad [5.1]$$

we are about to define a spline curve that is defined in intervals as before:

$$C(t) = \begin{cases} c_0(t) & t_0 \leq t \leq t_1 \\ c_1(t) & t_1 \leq t \leq t_2 \\ \vdots & \vdots \\ c_{l-1}(t) & t_{l-1} \leq t \leq t_l \end{cases} \quad [5.2]$$

In this particular case, we agree that $c_i(t) \in \mathbb{P}_3$, where \mathbb{P}_k is the linear space of polynomials of degree k . Each of these polynomial pieces must satisfy the following conditions:

$$\begin{aligned} c_i(t_i) &= p_i \\ c_i(t_{i+1}) &= p_{i+1} \\ c'_i(t_i) &= s_i \\ c'_i(t_{i+1}) &= s_{i+1} \end{aligned} \quad [5.3]$$

where the slopes s_i are free parameters. Thus, the curve passes through the interpolating points and its first derivative agree at these points regardless of the value for the slopes.

In order to compute the coefficients for the i -th polynomial the Newton form is used:

$$c_i(t) = a_i(t - t_i)^3 + b_i(t - t_i)^2 + c_i(t - t_i) + d_i : \forall t \in [t_i, t_{i+1}] \quad [5.4]$$

Since there are l polynomials, each of them with 4 parameters, we need $4l$ independent conditions to find them. We have $2l$ conditions since the curve passes through the points: $c_i(t_i) = p_i \wedge c_i(t_{i+1}) = p_{i+1}$ (conditions 1 and 2 in [5.3]). Moreover, with the third and fourth conditions in [5.3] we can force another $2l - 2$ conditions since $c'_i(t_i) = c'_{i-1}(t_i)$.

Therefore, we have $2l + 2l - 2 = 4l - 2$ conditions and two more are needed. Spline kinds such as the Hermite spline, Bessel spline, Akima's spline or the Clamped spline are not valid since all of them need the values for the slopes. These kinds of splines are used mainly for approximating a non-polynomial function, and hence the slopes can be known by deriving it, but not in our case.

On the other hand, the most common cubic spline is called natural and can be defined with the following two restrictions needed to have the $4l$

conditions: $c''_0(t_0) = 0 \wedge c''_{l-1}(t_l) = 0$. A trivial linear system can be made to solve the $4l$ coefficients and be solved very efficiently.

After all, what we have is a cubic spline defined with l polynomials of degree 3. Moreover, the first derivative is continuous so that we can compare melodies by calculating the area among them. In addition, the second derivative is continuous, what makes the first one smooth. Thus, the first derivative will not have huge variations and will be more accurate.

6 Coping with Chords

As seen in the User Requirements Definition, one of the most desired features of the system is the capability to compare chords. However, with the current approximation by interpolating curves it is not feasible to compare chords since at a given point on time only one note can be defined. That is to say that the current model does allow melodic intervals, but not harmonic ones.

As we saw with the General Requirements, the idea behind chords is that there might be different possible paths to go from one note to another by crossing the chords' notes. For instance, let us consider the linear interpolation in Figure 6.1 where some random chords are inserted in a melodic line.

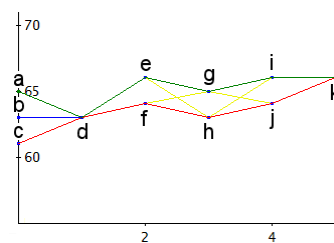


Figure 6.1 Paths through chords

The desired comparison should be able to recognize each of those paths across the chords. Moreover, it should allow not only a single path through all the chords, but also a set of them. For instance, a query may be the green path \overline{adehik} or the red one \overline{cdfhjk} .

However, a better one would contain the green and the red ones. The point here is that, if the green and the red paths are provided, the segments in yellow can be inferred. Actually, what is provided is not the set of paths but the actual notes. Therefore, we could have said that the green path would be \overline{adehik} and that the red one would be \overline{cdfgjk} . Thus, since all these combinations could be possible, all of them should be considered.

This example shows the basic idea with the chord comparison, but it has been seen with linear interpolation (splines of degree 1). Since the degree of the interpolating splines will probably be 3, we will come back to this issue later on since the general case is more complex.

7 Parametric Curves

So far, every interpolating method was of the form $C(t): \mathbb{R} \rightarrow \mathbb{R}$, but for the next interpolating methods the curve will not be defined in that way. On the other hand, it will be defined as a parametric curve. In essence, a parametric function is defined as

$$C(u): [0,1] \rightarrow (C_1(u), C_2(u), \dots, C_n(u)) \in \mathbb{R}^n \quad [7.1]$$

Therefore, $C(u)$ maps a value in the interval $[0,1]$ to a point in the space \mathbb{R}^n . Actually, the parameter u can be evaluated in whatever the interval, but for convenience it is always evaluated between 0 and 1.

Thus, if we want a parametric curve to interpolate the notes in the pitch-time plane, it will have the following form:

$$C(u): [0,1] \rightarrow (\text{time}(u), \text{pitch}(u)) \in \mathbb{R}^2 \quad [7.2]$$

This kind of functions is mostly used for curves defined in a three dimensional space. For instance, if we define the following parametric curve:

$$C(u) := (u, u^2, u^3) \mid u \in [-1,1] \quad [7.3]$$

in the interval $[-1,1]$ we will have a curve defined in the space, in a box from point $(-1,0,-1)$ to point $(1,1,1)$, as Figure 7.1 depicts.

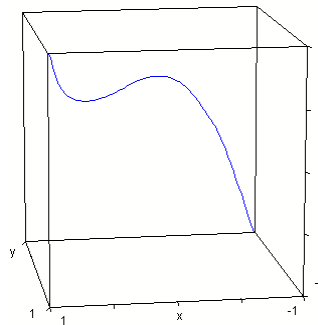


Figure 7.1 Parametric curves

Therefore, a parametric curve defines actually n functions, one per dimension. The only point is the domain for u , since an approximation method should be used to obtain the point in space that corresponds to a certain value in a certain axis. This makes some problems arise, but they will be avoided thanks to another solution that will be discussed later on.

8 Bézier Curves

Another type of spline interpolation, which will give us some interesting properties, is the Bézier curves. These curves will give us the basic ideas for the final interpolating spline used in the system since they are based on the Bézier curves and hence share their properties besides some others.

8.1 Definition

Given $l+1$ notes m_0, m_1, \dots, m_l , called the control points, the Bézier curve defined by these control points is

$$C(u) = \sum_{i=0}^l B_{l,i}(u) m_i \quad [8.1]$$

where the coefficients, known as Bézier basis functions or Bernstein polynomials, are calculated as follows:

$$B_{l,i}(u) = \frac{l!}{i!(l-i)!} u^i (1-u)^{l-i} \quad [8.2]$$

The variable u is evaluated within the interval $[0,1]$ and therefore all basis functions are non-negative. It is important to note that the curve is defined as parametric, since the multiplication by each point m_i results in two functions: one for the time-component and another one for the pitch-component of the plane.

In Figure 8.1 a Bézier curve with 11 control points is depicted in blue. The red polygon is called the control polygon or control polyline, depending on the literature, and it connects all the control points. In addition, the green circle marks the point in the curve that corresponds to $u = 0.4$.

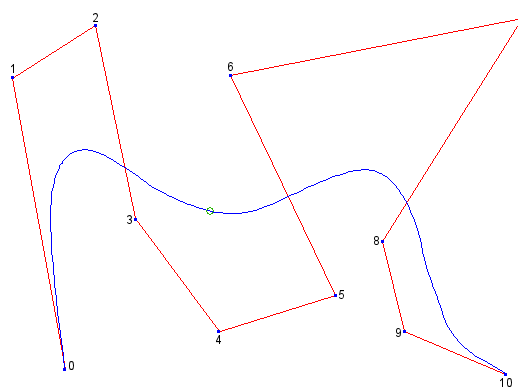


Figure 8.1 Bézier curves

8.2 Properties

Bézier curves have some interesting properties that we must consider before starting the interpolation phase:

- The degree of a Bézier curve defined by $l + 1$ control points is l .
- $C(u)$ passes through m_0 and m_l .
- Non-negativity: all basis functions are non-negative.
- Partition of unity: the sum of all basis functions at a fixed u is 1. Figure 8.2 shows a typical distribution for basis functions (in this case with 5 control points). According to [8.1] each of these functions is multiplied by the corresponding point so that the sum leads to the actual point for the curve.

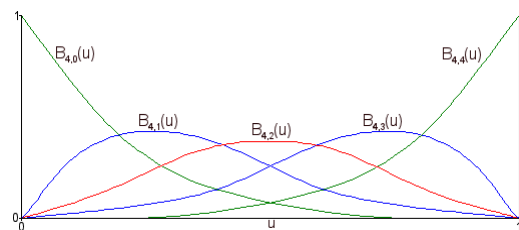


Figure 8.2 Bézier basis functions

- Convex hull: the Bézier curve lies completely in the convex hull of the given control points. This convex hull is actually the area contained within the outer points of the control polygon. Thus, in Figure 8.3 appears the same curve as Figure 8.1 depicted, and due to this property the curve lies completely in the grey area.

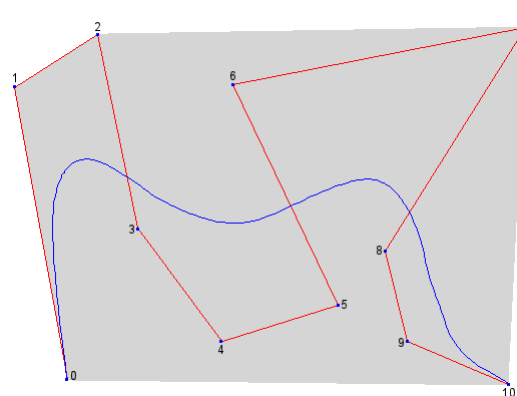


Figure 8.3 The convex hull property

- Variation diminishing: if the curve is in a plane, this means that no straight line intersects the curve more times than it intersects the curve's control polyline. For instance, in Figure 8.4 the blue Bézier curve is not right because the green straight line intersects it four times and only 2 times the control polyline.

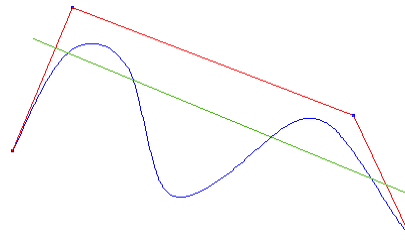


Figure 8.4 The variation diminishing property

- Affine invariance: if an affine transformation is applied to a Bézier curve, the resulting curve can be constructed from the affine images of its control points. Thus, since a linear translation of the time-dimension is needed in order to compare spans, this property assures that the curve will not change its shape with different span lengths or note durations. Indeed, the pitch component of the curve does not change as long as the notes have the same pitch.

All these properties, besides some others unique for the B-Splines, will help us to solve the General Requirements and will also lead us to a final mathematical model that solves the whole music information retrieval issue.

9 B-Splines

Bézier basis functions were used as weights for the construction of Bézier curves. In the case of a B-Spline, there are also basis functions used in the same way, but they are much more complex: the domain is subdivided by knots and each basis functions is non-zero on a few adjacent intervals so that B-Spline basis functions are quite local.

9.1 Definition

Let U be a set of $l+1$ non-decreasing numbers $u_0 \leq u_1 \leq \dots \leq u_l$. The u_i 's are called knots and the set U is called the knot vector, whilst the half-open interval $[u_i, u_{i+1})$ the i -th knot span. In our case, we are going to consider that the sequence is monotone increasing so that $u_0 < u_1 < \dots < u_l$ and every knot is called a simple knot.

These knots can be considered as division points that subdivide the interval $[u_0, u_l]$ into knot spans. Even though the domain can be chosen like it was in the Bézier curves, for convenience the interval $[0,1]$ is chosen as the domain for u .

Another characteristic of the B-Spline basis functions is that they can have whatever the degree, say p , so that each basis function is defined with the Cox-de Boor recursion formula:

$$N_{i,0}(u) = \begin{cases} 1 & u_i \leq u \leq u_{i+1} \\ 0 & \text{otherwise} \end{cases} \quad [9.1]$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)$$

Therefore, if we consider the knot vector $U = (0, 0.25, 0.5, 0.75, 1)$, we know that each $N_{i,0}(u)$ will be 1 in $[u_i, u_{i+1})$, so that the basis functions will be as Figure 9.1 depicts and Table 9.1 shows.

Basis Function	Span	Equation
$N_{0,0}(u)$	$[0, 0.25)$	1
$N_{1,0}(u)$	$[0.25, 0.5)$	1
$N_{2,0}(u)$	$[0.5, 0.75)$	1
$N_{3,0}(u)$	$[0.75, 1)$	1

Table 9.1 Degree 0 B-Spline basis functions

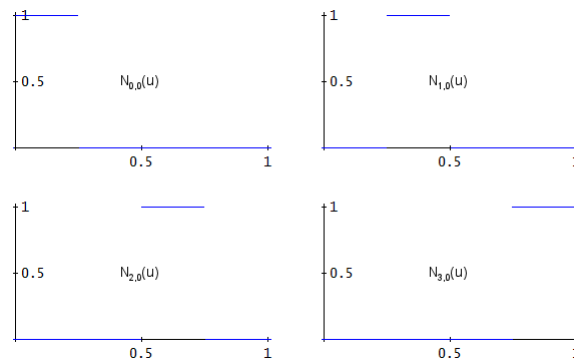


Figure 9.1 Degree 0 B-Spline basis functions

The following shows the equations for the degree 1 basis functions.

Basis Function	Span	Equation
$N_{0,1}(u)$	$[0, 0.25)$	$4u$
	$[0.25, 0.5)$	$2 - 4u$
$N_{1,1}(u)$	$[0.25, 0.5)$	$4u - 1$
	$[0.5, 0.75)$	$3 - 4u$
$N_{2,1}(u)$	$[0.5, 0.75)$	$4u - 2$
	$[0.75, 1)$	$4 - 4u$

Table 9.2 Degree 1 B-Spline basis functions

which are depicted in Figure 9.2:

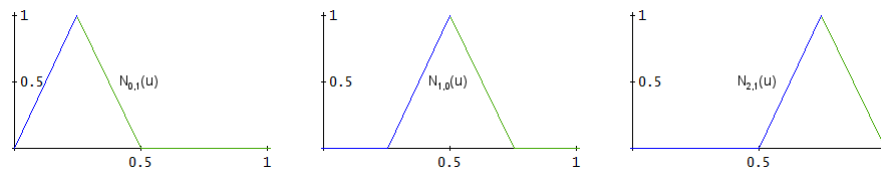


Figure 9.2 Degree 1 B-Spline basis functions

In order to calculate the degree 2 functions, we only need to follow the recursion formula in [9.1] with the equations already calculated in Table 9.2. This leads us to the following equations for the degree 2:

Basis Function	Span	Equation
$N_{0,2}(u)$	$[0, 0.25)$	$8u^2$
	$[0.25, 0.5)$	$-16u^2 + 12u - 1.5$
	$[0.5, 0.75)$	$8u^2 - 12u + 4.5$
$N_{1,2}(u)$	$[0.25, 0.5)$	$8u^2 - 4u + 0.5$
	$[0.5, 0.75)$	$-16u^2 + 20u - 5.5$
	$[0.75, 1)$	$8u^2 - 16u + 8$

Table 9.3 Degree 2 B-Spline basis functions

Each of these functions is defined in three knot spans as we can see in Figure 9.3:

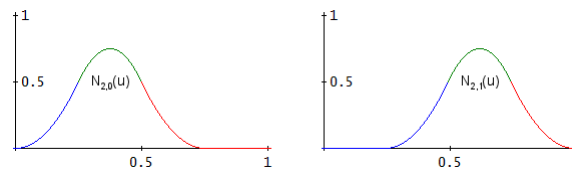


Figure 9.3 Degree 2 B-Spline basis functions

Once the basic functions are calculated, the B-Spline curve is defined pretty similarly to the Bézier curves:

$$C(u) = \sum_{i=0}^l N_{i,p}(u)m_i \quad [9.2]$$

where p is the degree for the polynomials. Unlike a Bézier curve, a B-Spline involves more information: a set of $k + 1$ knots and a degree p . However, these parameters must satisfy the following:

$$k = l + p + 1 \quad [9.3]$$

That is to say that, if we want a B-Spline curve of degree 3 with 10 control points, we must provide 14 knots that will give us 13 knot spans. This is the case, for instance, of the curve depicted in Figure 9.4:

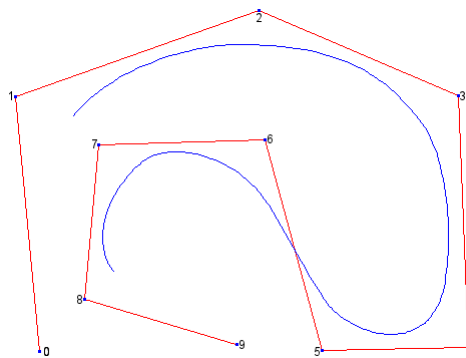


Figure 9.4 B-Spline curves

9.2 Properties

B-Spline curves share many important properties with Bézier curves, because the former is a generalization of the latter. In addition, B-Spline curves have unique properties that make them the most suitable for our purposes.

- A B-Spline is a piecewise curve with each component a curve of degree p . This allows us to design complex shapes with lower degree polynomials since the curve approximates to the control polyline as the degree decreases.

- The equality $k = l + p + 1$ must be satisfied.
- Strong convex hull: the same property of the convex hull is present in B-Splines but even stronger. If a point is defined for a given u in knot span $[u_i, u_{i+1})$, then the point will be in the hull made up by control points $m_i, m_{i-1}, \dots, m_{i-p}$.

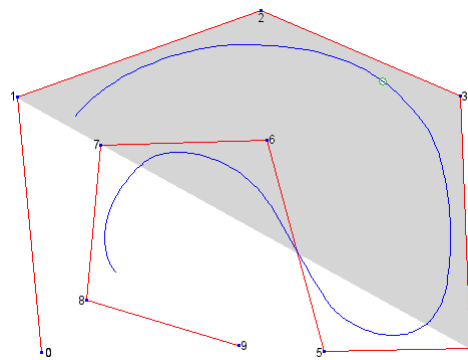


Figure 9.5 Strong convex hull property

- Affine invariance: the same property of the Bézier curves appears with B-Splines.
- Local modification scheme: changing the position of a control point m_i only affects the curve on the interval $[u_i, u_{i+p+1})$. Since every basis function $N_{i,p}(u)$ is non-zero only on interval $[u_i, u_{i+p+1})$, changing its corresponding control point will change only that interval. This means that if we change a note, the curve will change only in an interval close to it. Moreover, this interval is deterministic. In the figure below, the control point number 5 is changed and we can see how the curve changes only in its proximity.

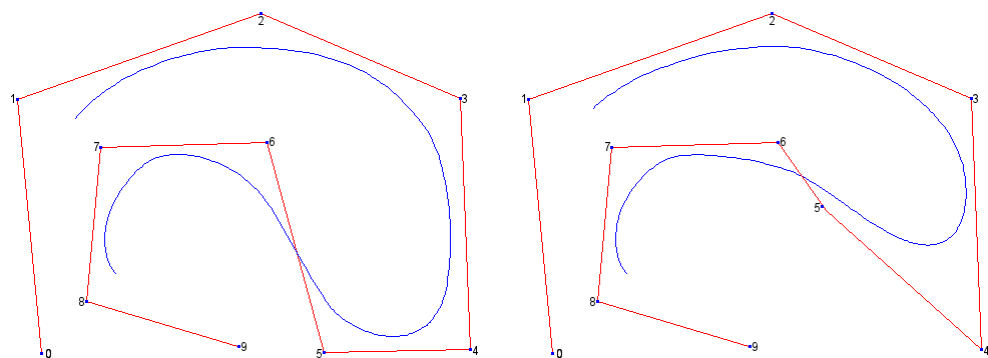


Figure 9.6 Local modification scheme

- $C(u)$ is C^{p-h} continuous at a knot of multiplicity h . If u is not a knot, the curve is infinitely differentiable at that point because it is a polynomial of degree p .

- Variation diminishing: if the curve is in a plane, this means that no straight line intersects the curve more times than it intersects the curve's control polyline.

10 Uniform B-Splines

When the knot vector of a B-Spline contains knots that are equidistant, the B-Spline is called uniform because every knot span has the same length. This means that the basis functions $N_{i,p}(u)$ are all translates of a single blending function $N_p(u)$ where

$$N_{i,p}(u) = N_p(u - i) \quad [10.1]$$

This blending function can be defined by convolution of blending functions of lower degree and assuming a fixed span length, say 1.

10.1 The Blending Function

The uniform B-Spline blending function of degree p is defined recursively by:

$$\begin{aligned} N_0(u) &= \begin{cases} 1 & u_i \leq u \leq u_{i+1} \\ 0 & \text{otherwise} \end{cases} \\ N_p(u) &= (N_{p-1} * N_0)(u) \end{aligned} \quad [10.2]$$

This convolution can be seen to be the integral

$$N_p(u) = (N_{p-1} * N_0)(u) = \int_{-\infty}^{\infty} N_{p-1}(x) N_0(u - x) dx = \int_{u-1}^u N_{p-1}(x) dx \quad [10.3]$$

Thus, to obtain the blending function of degree 1, we have to calculate this defined integral as

$$N_1(u) = \int_{u-1}^u N_0(x) dx \quad [10.4]$$

and evaluate it by intervals. The two possible intervals are depicted in Figure 10.1, where every interval has a length of 1.

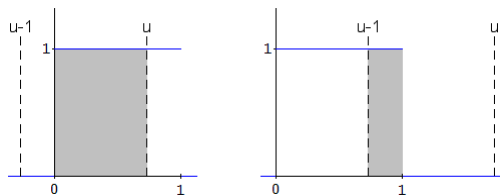


Figure 10.1 Degree 1 blending function integration intervals

Therefore, the blending function is defined in two intervals, and the equations are solved by solving both integrals.

$$N_1(u) = \begin{cases} \int_0^u dx & 0 \leq u \leq 1 \\ \int_{u-1}^1 dx & 1 \leq u \leq 2 \end{cases} \quad [10.5]$$

$$= \begin{cases} u & 0 \leq u \leq 1 \\ 2 - u & 1 \leq u \leq 2 \end{cases}$$

The blending function is quite the same as the ones in Figure 9.2:

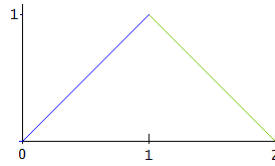


Figure 10.2 Degree 1 blending function

Now that we have the blending function of degree 1, we can apply the same formulas and obtain the degree 2 blending function. The intervals are:

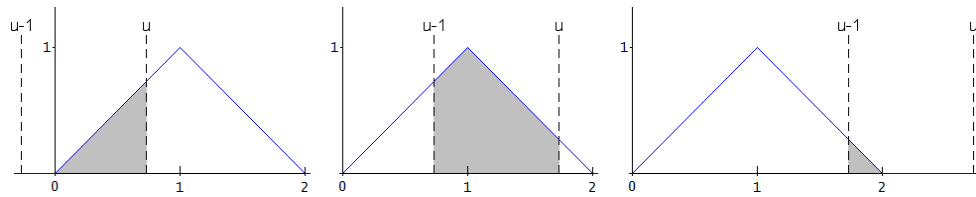


Figure 10.3 Degree 2 blending function integration intervals

and hence, the blending function is calculated as:

$$N_2(u) = \begin{cases} \int_0^u x dx & 0 \leq u \leq 1 \\ \int_{u-1}^1 x dx + \int_1^u 2 - x dx & 1 \leq u \leq 2 \\ \int_{u-1}^2 2 - x dx & 2 \leq u \leq 3 \end{cases} \quad [10.6]$$

$$= \begin{cases} \frac{u^2}{2} & 0 \leq u \leq 1 \\ -\frac{2u^2 - 6u + 3}{2} & 1 \leq u \leq 2 \\ \frac{u^2 - 6u + 9}{2} & 2 \leq u \leq 3 \end{cases}$$

These equations lead us to a blending function that is, once again, the same function in essence as the one seen before in Figure 9.3:

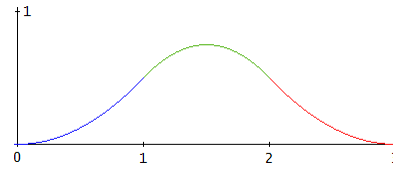


Figure 10.4 Degree 2 blending function

If we want blending functions of a higher order, we only have to keep on going with the convolutions. For instance, if we continue until a degree 3 (which will be the one used in the system), we have the following blending function:

$$N_3(u) = \begin{cases} \int_0^u \frac{x^2}{2} dx & 0 \leq u \leq 1 \\ \int_{u-1}^1 \frac{x^2}{2} dx + \int_1^u -\frac{2x^2 - 6x + 3}{2} dx & 1 \leq u \leq 2 \\ \int_{u-1}^2 -\frac{2x^2 - 6x + 3}{2} dx + \int_2^u \frac{x^2 - 6x + 9}{2} dx & 2 \leq u \leq 3 \\ \int_{u-1}^3 \frac{x^2 - 6x + 9}{2} dx & 3 \leq u \leq 4 \end{cases} \quad [10.7]$$

$$= \begin{cases} \frac{u^3}{6} & 0 \leq u \leq 1 \\ -\frac{3u^3 - 12u^2 + 12u - 4}{6} & 1 \leq u \leq 2 \\ \frac{3u^3 - 24u^2 + 60u - 44}{6} & 2 \leq u \leq 3 \\ -\frac{u^3 - 12u^2 + 48u - 64}{6} & 3 \leq u \leq 4 \end{cases}$$

The point now is that each of these functions must be translated so that they are evaluated in the interval $[0,1]$. For instance, the second equation is evaluated in $[1,2]$, but it should be in $[0,1]$, so that the equation is translated to the left by changing the variable u for $u+1$:

$$-\frac{3(u+1)^3 - 12(u+1)^2 + 12(u+1) - 4}{6} = -\frac{3u^3 - 3u^2 - 3u - 1}{6} \quad [10.8]$$

Thus, the third and fourth equations would be

$$\begin{aligned} \frac{3(u+2)^3 - 24(u+2)^2 + 60(u+2) - 44}{6} &= \frac{3u^3 - 6u^2 + 4}{6} \\ -\frac{(u+3)^3 - 12(u+3)^2 + 48(u+3) - 64}{6} &= -\frac{u^3 - 3u^2 + 3u - 1}{6} \end{aligned} \quad [10.9]$$

Later on, multiplying each of them by its corresponding control point and then calculating the sum, we have the parametric function that represents

the curve in the given interval. This function is usually put in matrix form with the blending functions and the fixed set of points that define the interval:

$$c_i(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} m_{i-1} \\ m_i \\ m_{i+1} \\ m_{i+2} \end{bmatrix} \quad | \quad u \in [0,1] \quad [10.10]$$

It is important to note that each piece of the uniform B-Spline is evaluated between 0 and 1 and there is no global domain for the whole curve. Thus, each basis function is defined there and, once again, the sum of all them is equals to 1 in the whole interval $[0,1]$, so that each of them is a weight for each control point. As Figure 8.2, they are depicted in Figure 10.5:



Figure 10.5 Degree 3 uniform B-Spline basis functions

11 Why Degree 3 Uniform B-Splines

So far, we have seen many interpolating methods with many interesting properties for the music information retrieval. Afterwards, the chosen method uses degree 3 uniform B-Splines for some reasons that are discussed from now on.

- Since it is a spline curve, it does not suffer of the Runge's phenomenon, so that the curve will not oscillate between points.
- Moreover, it has the variation diminishing property, so that the oscillation between points is minimal.
- A B-Spline is a piecewise curve, so that the curve is already split into pieces between control points. These pieces will be the basic information unit for RSHPs. A discussion about the boundaries of each piece will be seen later on in Section 11.2.
- Each piece of the curve is a parametric polynomial of degree 3, so that the curve is differentiable. In addition, since every knot has a multiplicity of 1, the curve is $C^{p-h} = C^2$ continuous. We need at least C^1 continuity, but a C^2 curve even makes the first derivative smooth.
- Thanks to the convex hull property (and therefore the strong convex hull one), the curve will not go beyond the limits of the domain. In other words, since every control point is a note with a pitch value in the interval $[0,127]$, the interpolating curve will not go beyond. Therefore, the curve complies with the pitch domain. In addition, it is easy to realize that it also complies with the time domain defined by the initial and end notes.
- Thanks to the local modification scheme, and making each knot span start and finish at the time-component of control points, the curve will not change globally whenever a note is changed. Particularly, since the curve has degree 3, changing a note m_i changes the curve in the interval $[t_{i-2}, t_{i+2}]$. This is a mandatory property if we want to cope with chord comparison, since every possible path through the points must be compared. With the other methods of interpolation the curves changed globally, but with B-Splines it changes around. Therefore, since the number of affected spans is deterministic, every single path through the points can be calculated and compared.
- Since the curve does not pass through the actual notes and it complies with the strong convex hull property, no large picks are going to be created. This makes the curve not-sensible to large changes in pitch so that we can have accuracy according to the pitch

difference between notes. In other words, the curve does not change linearly as the pitch changes.

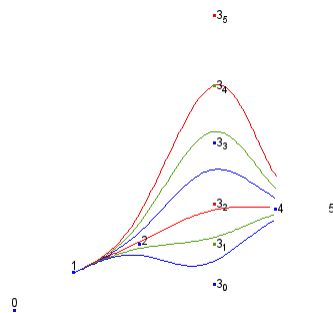


Figure 11.1 Moving control points

- Since the curve is parametric, we have a single function for the pitch-dimension and another one for the time-dimension. Therefore, no linear transformation is needed to the time-dimension in order to compare two intervals. Actually, the time-dimension function is useless since every span has length 1 (i. e. the piece is timeless).
- Thanks to the affine invariance property the curve will not change its shape whenever the span lengths needs to change.
- The lower the degree is, the fewer the number of possible paths we have for each chord. However, the lower the degree, the lower the quality of the curve and the lower also the continuity. Therefore, a midpoint should be chosen for the degree, and this midpoint is 3. Consider that on each onset time, say t_i the song has k_i harmonic notes. Thus, for a degree p , every single span, say s_i , depends on the points $m_i, m_{i+1}, \dots, m_{i+p}$, so that there will be $\prod_{j=i}^{i+p} k_j$ possible single paths in the span and hence $(p+1) \prod_{j=i}^{i+p} k_j$ possible whole paths in the whole interval of $p+1$ spans affected by m_i . At the end, a degree 3 is chosen since it guarantees the minimal number of possible paths between notes with the smoothness of the first derivative.
- A uniform B-Spline is easy and efficient to calculate since it needs only 4 multiplications of a polynomial by a number.
- Some other properties of the degree 3 uniform B-Splines are discussed in Section 6 of the following document's Part.

11.2 The Knot Spans

There are some issues with the knot spans lengths that must be considered carefully because they might ruin the whole thing.

In case the knot span is not uniform or the distance between successive notes is not uniform either, the time-dimension single function of the curve will not start and finish between the notes. In other words:

$$\begin{aligned} |t_{i+1} - t_i| < |t_{i+2} - t_{i+1}| &\rightarrow c_i(1) = t' = c_{i+1}(0) \mid t' \in (t_{i+1}, t_{i+2}) \\ |t_{i+1} - t_i| > |t_{i+2} - t_{i+1}| &\rightarrow c_i(1) = t' = c_{i+1}(0) \mid t' \in (t_i, t_{i+1}) \end{aligned} \quad [11.1]$$

This fact is depicted in Figure 11.2, where we can see the effect of having different separation between successive notes.

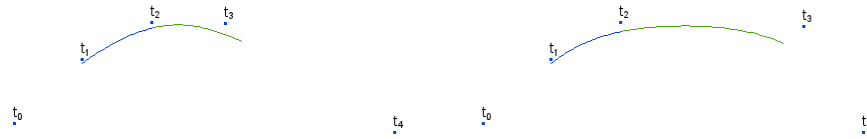


Figure 11.2 Non-uniform B-Spline spans

Since the lowest-level comparison is among the polynomials between two successive notes, these intervals must have the same length. Therefore, if the fact of having different note durations affects to the length of the intervals, the knot vector will be uniform as well as the set of control points.

It is obvious that by doing so we are wasting the time information, so it has to be used in some other way so that the system realizes also about the time-dimension.

Having a uniform knot vector we can use the basis functions in [10.10]. Then, since the distance between two successive notes is the same, say d , we have for the time-dimension:

$$d \frac{u^3}{6} - 2d \frac{3u^3 - 3u^2 - 3u - 1}{6} + 3d \frac{3u^3 - 6u^2 + 4}{6} - 4d \frac{u^3 - 3u^2 + 3u - 1}{6} = d(3 - u) \quad [11.2]$$

We can see that the polynomials for the time-dimension have degree 1, so that we do not need to approximate the value for u in case we want to knot the exact pitch for a certain time value.

We might move the knots so that the distance between them is somehow according to the corresponding note duration, but practical experience shows that modifying knot positions is neither predictable nor satisfactory.

It is important also to note that we are only changing the time-dimension but not the pitch one. The only thing we are doing upon the curve is shrinking or stretching it in the time-dimension. Actually this is the linear transformation needed upon the time-dimension, but is done from the beginning in this case so that every interval has the same length.

Lastly, if we want to have a function defined like [5.2], these spans must be translated in time so that the first point of the interval, $c_i(0)$, should be placed in $t = t_i$. Therefore, the curve is defined for the time-dimension as:

$$C_{\text{time}}(t) = \begin{cases} c_{\text{time},0} \left(\frac{t-t_1}{t_2-t_1} \right) + t_1 & t_0 \leq t \leq t_1 \\ c_{\text{time},1} \left(\frac{t-t_2}{t_3-t_2} \right) + t_2 & t_1 \leq t \leq t_2 \\ \vdots & \vdots \\ c_{\text{time},l-p} \left(\frac{t-t_{l-p+1}}{t_{l-p+2}-t_{l-p+1}} \right) + t_{l-p+1} & t_{l-p} \leq t \leq t_{l-p+1} \end{cases} \quad [11.3]$$

and for the pitch dimension as:

$$C_{\text{pitch}}(t) = \begin{cases} c_{\text{pitch},0} \left(\frac{t-t_1}{t_2-t_1} \right) & t_0 \leq t \leq t_1 \\ c_{\text{pitch},1} \left(\frac{t-t_2}{t_3-t_2} \right) & t_1 \leq t \leq t_2 \\ \vdots & \vdots \\ c_{\text{pitch},l-p} \left(\frac{t-t_{l-p+1}}{t_{l-p+2}-t_{l-p+1}} \right) & t_{l-p} \leq t \leq t_{l-p+1} \end{cases} \quad [11.4]$$

The last issue about the degree 3 B-Spline, and particularly related to its derivatives, will be discussed in the Part IX of the document, when the mathematical model is translated into the RSHP metamodel.

12 Coping with Voices

So far, we have been talking about the mathematical model assuming that the performances have only one single voice. However, we must deal with voices as it is a general constraint to the system. The whole idea of normalizing the domain, interpolating by using B-Splines and then compare each piece of the curve will be kept with voices.

In previous sections we have seen that the curve for a certain performance was of the form:

$$C(t) = \begin{cases} c_0(t) & t_0 \leq t \leq t_1 \\ c_1(t) & t_1 \leq t \leq t_2 \\ \vdots & \vdots \\ c_{l-1}(t) & t_{l-1} \leq t \leq t_l \end{cases} \quad [12.1]$$

This function interpolates the set of notes over a time-pitch plane. The issue about voices is that they have to be compared independently and together, as we saw in the General Requirements. Therefore, the best solution is to use an additional dimension for each voice, maintaining the time-dimension.

Doing so, a performance with two voices will be defined in a space of 3 dimensions and, in general, a performance with v voices will be defined in a space with $v+1$ dimensions. By doing so, if we want to compare two performances, say $C(t)$ and $D(t)$, we have to consider that each of them will return a vector in a voice1-voice2 plane:

$$C(t) = (\text{pitch}_1, \text{pitch}_2) \quad [12.2]$$

Therefore, if we consider the staff in Figure 2.1, after having normalized it and interpolated, the final curve would be like Figure 12.1 depicts, where the red and green curves are the single voices and the blue one is calculated by adding the other two as $C(t) = (C_1(t), C_2(t))$:

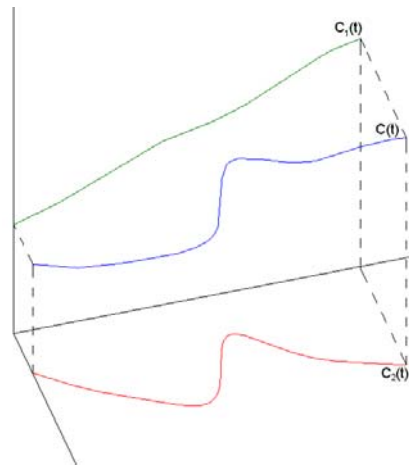


Figure 12.1 3-dimensional interpolating curve

With this kind of curves, we can compare $C(t)$ and $D(t)$ like seen before, calculating the area between them in a certain interval. On the other hand, if we want to compare only single voices, say $C_i(t)$ and $D_i(t)$, we can compare their single components, which lead us to a comparison among their partial derivatives:

$$\frac{\partial C(t)}{\partial \text{pitch}_i} = C_{\text{pitch}_i}(t) \text{ and } \frac{\partial D(t)}{\partial \text{pitch}_j} = D_{\text{pitch}_j}(t) \quad [12.3]$$

This way, we can compare the pitch variation of both curves together so that we can focus only on the partial derivative of one of them but also get some information about the pitch variation of the others so that shifted copies in time will be detected.

In the following part of the document we will see how this multi-dimensional model is translated into the RSHP metamodel and how we will be able to compare voices separately and together. The main lack that the RSHP metamodel sins of is that it is not able to represent mathematical expressions like functions, polynomials, derivatives and so on, so that the transformation is not trivial if we want to guarantee a successful comparison.

Part IX:
Translation to the RSHP
Metamodel and the CAKE
Engine

1 Artifacts' Topology

To begin the RSHP modeling, we have to define first the topology of the main artifact and their subartifacts, as well as the relationships that each of them contains. Thus, the main artifact's type is going to be a Sequence. In addition, this Sequence artifact will have an artifact Staff for each of them. That way, we can compare, for instance, the piano of a Sequence with the guitar of another one.

Even though this version of the system allows only one staff per sequence, the model presented here is valid for more staves. However, in Section 2 of Part XI we will see some future work that would be suitable at the time of adding more than one single staff per sequence.

Later on, each of these Staff artifacts will contain a subartifact of type Voice that will contain the actual musical information about a single voice in the staff. Doing so, we will be able to compare voices separately.

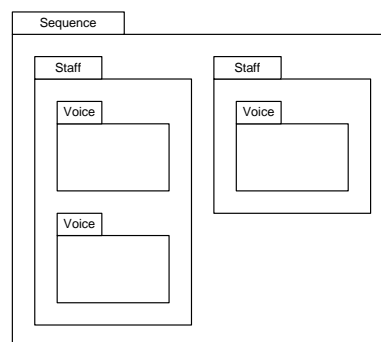


Figure 1.1 Main artifacts' topology (part I)

In the above Figure 1.1 appears a Sequence artifact that contains two Staff subartifacts: one of them has two Voice subartifacts and the other one has a single Voice subartifact. This topology corresponds to a performance like the one depicted in Figure 1.2.

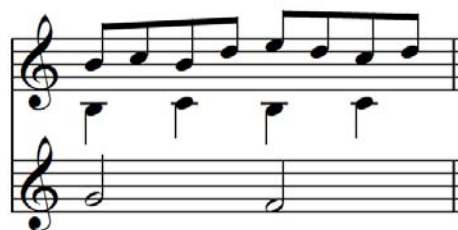


Figure 1.2 Main artifacts' topology (part II)

Now that each voice is defined separately, let us consider, from now on, that coping only with staves that have one single voice.

2 Single Voices without Chords

Here we have a Voice artifact that must be filled up with the information about a single curve $C_i(t)$ that has no chord. This function is actually defined by intervals between two successive notes, so we have to add information about each of these single pieces.

We saw in Section 3.5 of the document's Part VIII that the comparison among two intervals will be made by calculating the area between both curves and then dividing it by the maximum area of the two intervals. Note that we are comparing the first derivatives instead of the actual interpolating curves.

Even though this comparing method seems to be the best one for our purposes, it is quite difficult to translate that mathematical model into the RSHP metamodel. And not only this translation, but also the way the CAKE engine compares artifacts and RSHPs. After having considered many ways to represent polynomials with the RSHP metamodel and considering the results they would yield with the CAKE Engine, another comparison method is proposed taking advantage of the B-Spline interpolation method chosen.

2.1 Final Method to Compare Intervals

After having discarded the comparison method by calculating the area between curves, a new method is proposed by comparing punctual values of the first derivative along with the interval's duration and the shape of the curve in that interval. This information is going to be called information unit.

For instance, considering the first derivative in Figure 2.1, the information units to represent in the RSHP metamodel is (note that the values for the first derivatives are known):

- A piece of duration 4 that has a convex shape. The derivative value at the beginning is $C'(0)$ and the value at the end is $C'(4)$.
- A piece of duration 2 that has a concave shape. The derivative value at the beginning is $C'(4)$ and the value at the end is $C'(6)$.
- A piece of duration 2 that has a convex shape. The derivative value at the beginning is $C'(6)$ and the value at the end is $C'(8)$.

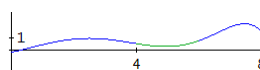


Figure 2.1 Comparing intervals

Later on, we can compare these information units by considering the shape, the duration and where are they placed in the pitch-dimension by means of the derivative values. Firstly, let us see how to represent this in the RSHP metamodel.

2.2 How to Represent Information Units

Reminding the RSHP metamodel seen in the document's Part V, there seems to be a clear way to represent information units. First of all, we have to consider the point that every RSHP must have a type, an action that gives semantics to that specific type of relationship and one or more information elements to connect by the RSHP.

Since we are comparing the derivatives of degree 3 polynomials, we have degree 2 curves. In addition, a degree 2 polynomial can have only three possible shapes: concave, convex or flat despite of whether they have maxima or minima points or not.

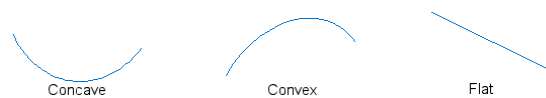


Figure 2.2 Degree 2 polynomial shapes

Thus, we are going to have three kinds of RSHP: Concave, Convex and Flat.

On the other hand, the action of the RSHP will be the duration of the interval, whilst the left IE will be a Term containing the derivative at the beginning (with concept order 1) and the right IE will be another Term containing the derivative value at the end (with concept order 2).

Therefore, if we have an interval of duration 6, with a concave polynomial that goes from a derivative value of 4 to a value of 3, the relationship is going to be depicted as Figure 2.3 shows (note that the line linking the IEs has a concave shape due to the RSHP's type):

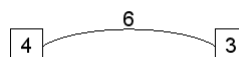


Figure 2.3 A Concave RSHP

Therefore, if we have a sequence that contains only the curve in Figure 2.1, the resulting Sequence Artifact would be like the one in Figure 2.4:

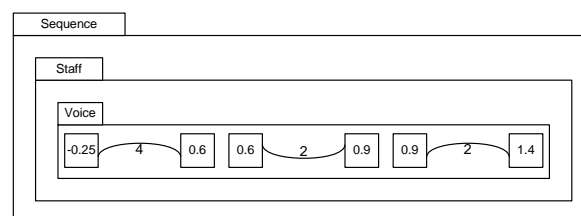


Figure 2.4 Sequence with a single voice

In addition, each of these RSHP is marked up with a position in the Voice artifact so that the sequence is somehow modeled too.

3 Single Voices with Chords

Now that we have seen how to model a basic melody with RSHP, let us see how to model a melody with some harmony, i. e. chords. For instance, let us consider the first derivative depicted in Figure 3.1:

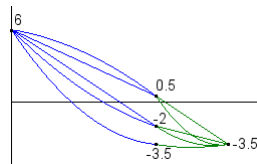


Figure 3.1 Voice with chords (part I)

The point is that a query might have only a few of the possible paths contained in the original artifact, like the following one:

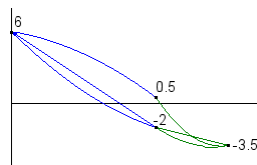


Figure 3.2 Voice with chords (part II)

Moreover, the semantic of the chord is that all the paths must be together. That is to say that the blue curves, for example, must appear together in the same time span. If they appear in different spans they will not be a chord, so that we must group all the possible paths of a chord (or a melody) into a bigger information unit containing all of them. This information unit is, of course, an artifact.

Therefore, each of the previous spans in the query would be grouped into a single artifact of type Span that is a subartifact of the Voice artifact:

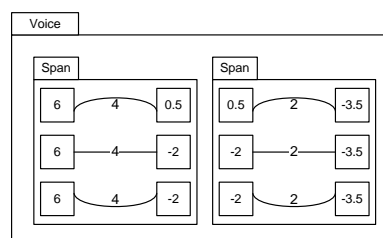


Figure 3.3 The Span artifact

Note that all the previous examples would be modeled also as a set of Span artifacts containing a single RSHP. The thing is that if the artifact in Figure 3.3 is compared with the corresponding artifact in Figure 3.1, a comparison by inclusion would lead as to a semantic distance of 0, since we have some paths of the chord in the query's artifact.

Like before, each Span subartifact has a position within the Voice artifact, and the RSHPs have now the position in the Span subartifact.

with concept order 2 that stores the value for the derivative at the end of the value. However, we need some information about how the other voices are changing in the interval, how their derivatives evolve.

The change consists on introducing also the values of the other voice's derivative at both limits of the interval. Let us take a look at the first derivative of the interpolating curve for Figure 4.1:

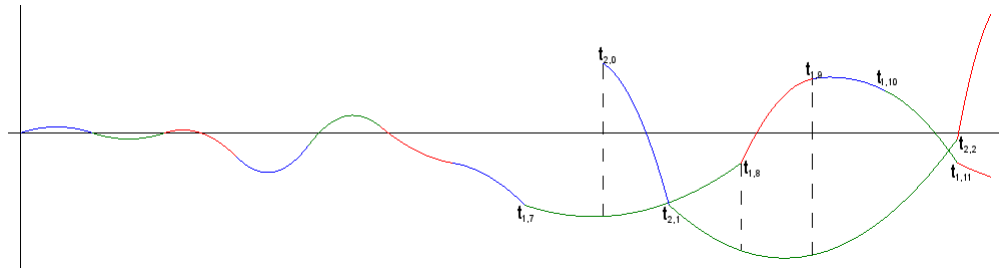


Figure 4.3 Derivatives with several voices

The idea is to include in the RSHP information about the other derivative, which will be the values at the beginning (with concept order 3) and the end of the interval (with concept order 4). Thus, a single RSHP for the interval $[t_{1,8}, t_{1,9}]$ as a type Convex, a duration of 4 and a concept order 1 of $C_1'(t_{1,8})$ as well as a concept order 2 of $C_1'(t_{1,9})$. Now, we are going to add information about the other curve, so we add a Term with concept order 3 containing $C_2'(t_{1,8})$ and a Term with concept order 4 containing $C_2'(t_{1,9})$.

Likewise, the RSHP for the interval $[t_{2,0}, t_{2,1}]$ is Concave as well, with a duration of 4 units and a concept order 1 of $C_2'(t_{2,0})$, concept order 2 of $C_2'(t_{2,1})$, concept order 3 of $C_1'(t_{2,0})$ and concept order 4 of $C_1'(t_{2,1})$. Note that some RSHPs might not have concept order 3 or 4, like it happens with the one modeling the interval $[t_{1,7}, t_{1,8}]$, which does not have a value for the concept order 3 but it does have the value for the concept order 4.

Thus, if we have translated copies of the voices, the concept orders 3 and 4 will change accordingly even though the original concept orders 1 and 2 will remain unchanged. Note that if another voice is present, there will be another pair of Terms with concept orders 3 and 4.

Therefore, with some additional considerations, comparisons between single voices will still work if the number of voices differs, since the concept orders 1 and 2 are the same. On the other hand, comparisons of the whole staff (all the voices) will realize of the time-translations and will give semantic distances accordingly.

5 Several Voices with Chords

The last scenario for a staff is the case with several voices containing chords. Let us consider the staff in Figure 5.1



Figure 5.1 Several voices with chords (part I)

that has the following first derivatives:

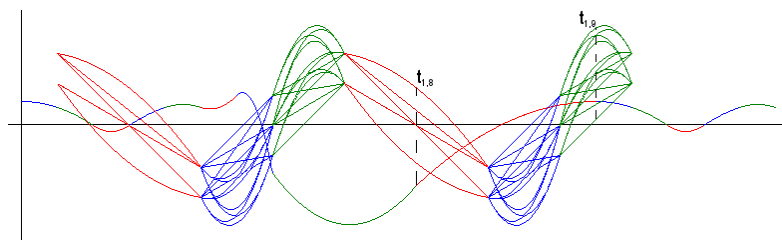


Figure 5.2 Several voices with chords (part II)

The point here is that in the interval $[t_{1,8}, t_{1,9}]$, for instance, there are several values for the derivatives that should be placed in the concept orders 3 and 4. We might put all their corresponding Terms with the corresponding concept order, but this would lead us to an error.

Let us imagine that we have three voices and we are modeling a RSHP for the first of them. The second voice provides concept order 3 with values a and b. Moreover, the third voice contributes with values c and d. Therefore, the RSHP would have 4 Terms with concept order 3.

Now consider a query with two voices, where the corresponding RSHP has the same concept orders 1 and 2, the same action and the same duration. That is, the same interval. Imagine also that the second derivative provides Terms with concept order 3 and values a and c. Both RSHP are represented in Figure 5.3.

The comparison among them would give a semantic distance of zero since the query is contained in the repository. But it is pretty clear that the semantic comparison is not correct because we are comparing voices 2 and 3 of the repository with the voice 2 of the query, and we must compare them separately. That is to say that we have to compare the voice 2 of the repository with the second of the query and later on the last with the third of the repository.

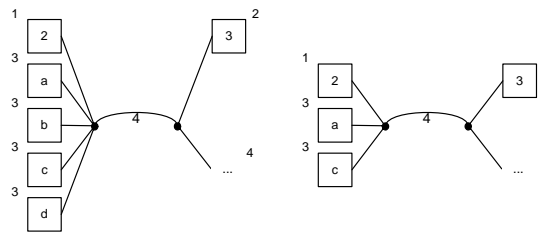


Figure 5.3 Several voices with chords (part III)

Thus, we have to group the derivative values for each of the derivatives so that we do not mix their values. Doing so, a new subartifact type arises, which is called Derivatives and will appear at both ends of the RSHPs, which are contained in subartifacts of type Span. Therefore, the final representation will be:

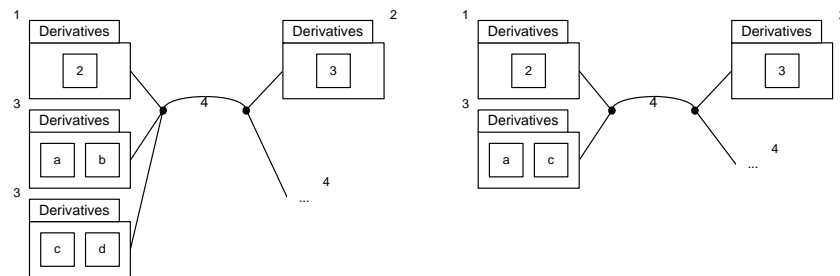


Figure 5.4 Several voices with chords (part IV)

This way we split the semantic of the other derivatives and separate them according to their voice.

6 Representing Numbers with Terms

The problem we must deal now with is the impossibility of representing real numbers in the RSHP metamodel. However, we can have some approximations so that we can define a closed domain for the possible note durations and the possible derivative values.

6.1 Score Durations

Right now, the real-time is not considered so that we only deal with score durations. As seen in Section 2.1 of the last document's Part, a minimum time unit must be defined so that we can normalize the time-dimension. The decision taken there is to consider a minimum length of demisemiquaver, so that a quaver would be 4 units for instance.

However, we must deal with the possibility of having triplets so that a demisemiquaver in the triplet would have duration of $\frac{\text{unit}}{3}$. Thus, minimum duration is going to be 3 units so that the demisemiquaver in a triplet will have duration of 2 units. Doing so, the standard notes have the following durations:







Name	Figure	Duration	In triplet
Semibreve		96	64
Minim		48	32
Crotchet		24	16
Quaver		12	8
Semiquaver		6	4
Demisemiquaver		3	2

Table 6.1 Score durations for notes

Therefore, we can use these numbers as the Terms used in the action of the relationships. However, we must consider the possibility of having ties and rhythm dots (see Section 4 of the Part II), so that a crotchet might be tied with a quaver and a rhythm dot. This would lead us to a duration of $24 + 12 + 12/2 = 42$ which can not be modeled with Terms.

We might define a huge number of Terms, each of them for the possible values in increments of 1 by 1, but this would be unfeasible at the end because we do not know where to stop. If we consider again ties, let us imagine a tie between two semibreves where the resulting duration would be 192 units. And even more, imagine three tied semibreves with a crotchet plus a semiquaver in a triplet.

The point is that we can not know, at the time of creating the domain, what are the possible values for the duration of notes. Therefore, the solution taken by now is to consider intervals between the values in Table 6.1. That is, imagine all the values ordered in a vector:

$$(2,3,4,6,8,12,16,24,32,48,64,96) \quad [6.1]$$

We are about to define intervals around each of these numbers, from the midpoint towards the previous number until the midpoint towards the following one. For instance, the interval containing 24 is $[20,28)$, whilst the one containing the 8 is $[7,10)$. On the other hand, since we know that no note will have a duration less than 2, the first interval is $[2,3)$. In addition, the last interval to be considered is $[80,+\infty)$, where every note with a duration longer than 80 is going to be grouped under this Term.

Thus, we have 12 Terms that define intervals for the possible durations in score time and that are defined as so in the domain of the RSHP repository. In addition, these Terms are linked in the domain so that a certain interval, say L_j , is linked with an Association RSHP to the intervals L_{j-1} and L_{j+1} .

6.2 Derivative Values

Something similar to the case of the score durations happens here, but let us see, first of all, why B-Splines of degree 3 were chosen.

As seen in the previous Section 10, the pitch-component of a span is calculated as:

$$\frac{u^3}{6}p_{i-1} + \frac{-3u^3 + 3u^2 + 3u + 1}{6}p_i + \frac{3u^3 - 6u^2 + 4}{6}p_{i+1} + \frac{-u^3 + 3u^2 - 3u + 1}{6}p_{i+2} \quad [6.2]$$

where p_j denotes the pitch of the note m_j . However, what we are about to compare is the first derivative of this function, which is

$$\frac{u^2}{2}p_{i-1} + \frac{-3u^2 + 2u + 1}{2}p_i + \frac{3u^2 - 4u}{2}p_{i+1} + \frac{-u^2 + 2u - 1}{2}p_{i+2} \quad [6.3]$$

Moreover, we are obtaining the value of the first derivative evaluated at $u = 0$, so that the values to put in the RSHPs are calculated as:

$$\frac{p_i - p_{i+2}}{2} \quad [6.4]$$

That is to say that the derivative evaluated in 0 depends only on the notes m_i and m_{i+2} . On the other hand, the value at $u = 1$ is the same as the value of the following span evaluated in 0, so that it is

$$\frac{p_{i+1} - p_{i+3}}{2} \quad [6.5]$$

Thus, the values at both ends of the RSHP depend on the 4 points that define the interval. Moreover, these values depend on the difference in pitch between two notes, and this is actually the idea behind the first derivative. That is to say that every pair of points $p_i - p_{i+2}$ or $p_{i+1} - p_{i+3}$ that keeps the pitch difference is going to have the same derivatives. And if we think on it, this is the case of a given interval in different tonalities:

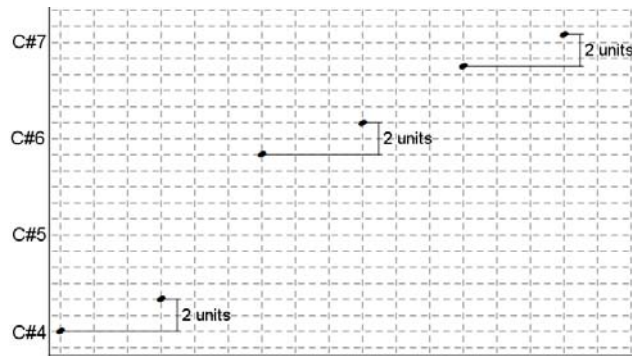


Figure 6.1 First derivative intervals

Therefore, the two values put in a RSHP can not correspond to different intervals in two curves. Since the left value depends on the interval between 2 notes and the right value depends on the interval between the other 2 notes, no other sequence of notes is possible with these values.

This is what gives us the sense of sequence in our representation of music. Remind that the RSHP metamodel does not have any mechanism that might help us with the issue of taking into account the sequence of notes. However, with B-Splines of degree p , we assure that a polynomial is unique of a fixed sequence of $p + 1$ points (actually the derivative of the curve). This acts like a sliding window that compares the curve by pieces of 4 notes.

Moreover, there is another good point from equations [6.4] and [6.5]. We know that every pitch value is in the interval $[0,127]$, so that the difference is -127 as minimum and 127 as maximum, which leads us to 255 possible values.

Thus, we have the domain entirely defined with the interval $[-127,127] \in \mathbb{Z}$, so that we might create a Term for the domain with each of the values in the interval.

6.2.1 Derivative Values with Voices

We have seen so far that the values for the first derivatives can be normalized with 255 Terms in the domain. However, this technique is not valid with staves containing more than a single voice. For instance, considering again the curve in Figure 4.3, the values having concept order 3 and 4 are not necessary integers.

The point is that, when calculating the concept order 3 by obtaining the derivative value $C_2'(t_{1,8})$, the polynomial has to be evaluated in $u = 0.25$, and later on $C_2'(t_{1,9})$ will evaluate the polynomial in $u = 0.5$. In general, the curves might be evaluated in any value for u between 0 and 1, and this makes it impossible to define a domain by extension so that a domain by definition of intervals is proposed again.

With the examples considered so far, no derivative value has been greater than 14 or smaller than -14. Therefore, intervals are chosen by now with length 1 between -14 and 14, so that integer numbers are in the middle of each interval and hence we will not be penalized for distances of, for instance, 3 and 3.2.

Thus, the domain defines Terms for the derivative values that are distributed as follows, so that we have 31 intervals:

$$(-\infty, -14.5), [-14.5, -13.5), \dots, [-0.5, 0.5), \dots, [13.5, 14.5), [14.5, \infty) \quad [6.6]$$

Like before these Terms are linked with an Association RSHP in the domain, so that every interval is associated with the previous and the following ones. That way, the CAKE Engine can penalize distances in the derivatives.

7 Extending the CAKE Engine

The current implementation of the CAKE Engine is not suitable at all for MIKE and for mathematical models in general. As seen in Section 6 there are some problems with the representation and comparison of numbers.

Even though Part XI of the document outlines a possible solution to the problem, the point is that that solution might not be the best at all.

Moreover, there are some problems with the comparison of the information elements contained in the RSHPs. As seen in previous sections, the RSHPs have five concept orders right now:

- Duration of the interval as concept order 0.
- Value of the derivative at the beginning of the span with concept order 1.
- Value of the derivative at the end of the span as concept order 2.
- Values of the other voices' derivatives at the beginning of the span as concept order 3.
- Values of the other voices' derivatives at the end of the span as concept order 4.

The right way to compare two RSHP is to compare only information elements with the same concept order. That is to say that if two elements in two RSHP are being compared, their semantic distance will be automatically 1 in case they do not have the same concept order in their respective relationships.

Therefore, the CAKE Engine has been modified so that an extension is possible. This extension allows a particular engine to implement the function in charge of calculating the semantic distance among two information elements. Once there, the concept orders can be compared and the distance returned accordingly.

Moreover, since the current model has more or less random intervals for the possible derivative values as well as for the possible durations in score time. Thus, the current version of the system uses a deprecated field in the Information Elements that is precisely a floating point number which is used to store the actual value of the derivative.

Therefore, another extension is made so that, provisionally, the semantic distance between two terms can be calculated by dividing their difference by the maximum of them. Thus, a relative distance is given between 0 and 1.

Part X:

Implementation Details

1 The Music Information Retrieval Process

The music information retrieval process is a complex procedure that is divided into several phases that need to be executed in sequence, one after the other.

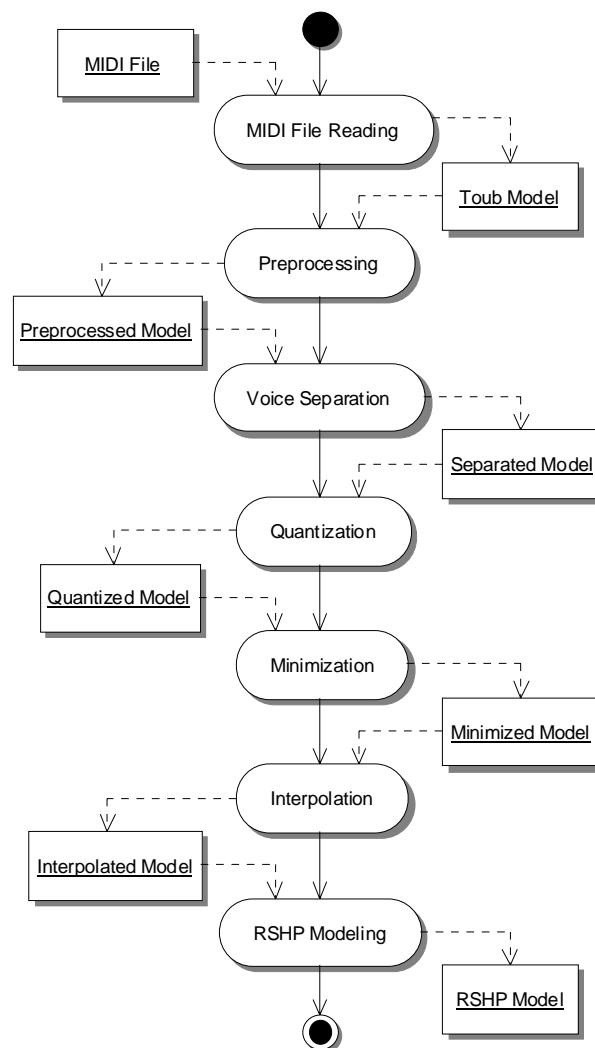


Figure 1.1 The music information retrieval process

As seen in Figure 1.1, there are mainly 7 phases executed in sequence, each of which returns an intermediate model that serves as input for the next phase.

- **Midi File Reading.** In this first phase of the process, the MIDI file is read and modeled with the Toub library [Toub].

- **Preprocessing.** In this part the file is preprocessed to clean it and remove useless information. For instance, the event-driven model is translated to a score-driven model where there are instances of notes with a certain onset time, duration and pitch. All the data but the NoteOn and NoteOff events, as well as the division, unit are removed.
- **Voice Separation.** In this third stage, the preprocessed model is split into several voices if applicable. This phase yields a figure-driven model where chords appear grouped.
- **Quantization.** In this phase the staff is quantized so that the onset time and the duration given in milliseconds are expressed in score units such as crotchet or minim. Moreover, the tempo is detected in this phase.
- **Minimization.** Once the input is quantized, it can be minimized by obtaining the possible repetitions in the performance such as riffs.
- **Interpolation.** Applying the technique seen in the mathematical approach, the performance is interpolated so that the curves that describe its pitch change for each voice are calculated.
- **RSHP Modeling.** In this final stage, the mathematical model is translated to artifacts as it was described before. This generated RSHP model can then be used for indexing or for accomplish a query.

These stages are implemented in MIKE with some aspects that need to be considered.

2 Implementation in MIKE

Each of the previous phases in the music information retrieval process is implemented in MIKE as a different package, nesting from a parent package named MIKE.

2.1 Preprocessing

In this first package the preprocessing task is implemented. The package defines a `IPreprocessor` interface that is going to be implemented by a particular preprocessor. The input to this phase is a Toub model (an instance of class `MidiSequence`) that is event-driven and, after the phase is executed, a preprocessed model is returned.

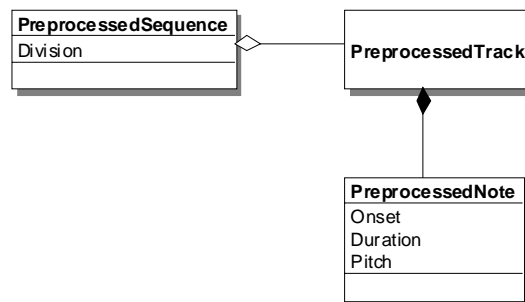


Figure 2.1 Preprocessed model

The implemented preprocessor used in this version is called `SillyPreprocessor` since the only task that it is going to perform is to remove useless events from the stream. Therefore, the preprocessor is going to deal only with `NoteOn` and `NoteOff` messages. For the output model, the preprocessor creates instances of notes that do have a certain onset time and duration, so that the event-driven model is no longer used.

2.2 VoiceSeparation

As well as the Preprocessing package, this one also defines an interface `IVoiceSeparator` that can be implemented in order to provide the voice separation functionality. This phase is also in charge of detecting chords, so that the output model is slightly changed. A chord is going to be considered as a collection of single notes. Thus, an interface `ISeparatedNote` is declared to deal with single notes and chords.

In this version of the system, a `SillyVoiceSeparator` is implemented even though the Kilian-Hoos algorithm should be used. The point is that the Kilian-Hoos algorithm needs some feedback from the user in order to adjust the penalization weights so that the input can be split correctly.

Thus, the Kilian-Hoos algorithm can not be used right now because the process does not have any kind of interaction with the user. Instead, the SillyVoiceSeparator is used to generate a separated model.

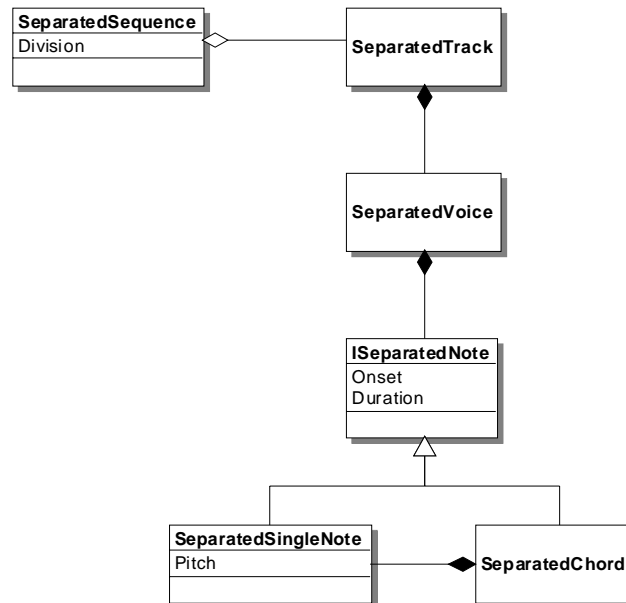


Figure 2.2 Separated model

The point is that the SillyVoiceSeparator actually does not do anything but recognizing and grouping chords. Thus, the input must have a special format so that, by now, the voices are already separated and stored in different tracks. In other words, if a certain staff has more than a single voice, it is separated into several tracks, one per voice. Therefore, the current version allows only one staff per sequence.

Anyway, the KilianHoos algorithm is implemented waiting for an extension to the current version that could use it so that the voice separation is actually performed and the input does not need to be already separated.

2.3 Quantization

So far, the onset time and duration of notes are expressed in milliseconds, and the quantization phase intends to recognize the tempo and hence the score onset time and score duration of notes.

Once again, the package defines an interface name IQuantizator that declares a method to quantize a given instance of the SeparatedModel. Like before, a SillyQuantizator is implemented in this version for several reasons. First of all, this version only deals with mechanical or metrical sequences, where the onset and duration of notes is precisely defined accordingly to the division field of the file. This means that the sequence can be quantized simply by division and multiplication.

As seen in Section 6 of the document's Part VIII, the crotched has assigned a score duration of 24, so that for a given delta time the score time is

$\frac{24 \cdot \text{delta}}{\text{division}}$

[2.1]

On the other hand, with the tempo detection and quantization happens more or less the same as with the voice separation: none of the existing algorithms generates a precise output. There are some approaches to the tempo detection based on rules, probabilistic models, multiple agents, oscillators and many more. For the quantization there are also some approaches based on grids, rules, transcriptions, vector models, cellular automata and more. In [Kilian, 2004] some hybrid techniques are proposed so that they might be used in a future.

On the other hand, the most typical way to record a live performance is with a clicktrack model where the tempo is defined before so that the quantization can be performed in real-time while playing. Thus, the quantization phase is used in some particular cases.

After all, the model generated in this phase is basically identical as the one generated in the voice separation. This time, the duration and onset time of the notes is stored in score time, but in the future it would be nice to store also the real time values so that both them can be used when comparing.

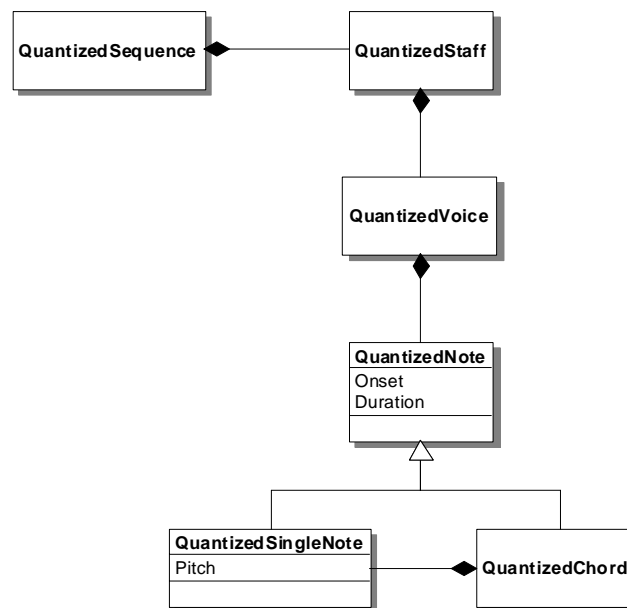


Figure 2.3 Quantized model

2.4 Minimization

This phase is neither implemented in the current version. Therefore, no minimization is applied upon the sequences so that they are not optimized. This is, however, an important issue for the efficiency of the system since the number of artifacts to create can be huge. Some algorithms such as BLAST can be applied here to recognize patterns in the sequence. Indeed, in [Kilian, 2004] an extension called MusicBLAST is introduced for the pattern induction and segmentation of sequences.

Therefore, no minimized model is created and the quantized one is directly forwarded to the Interpolating phase.

2.5 Interpolation

This package uses a class name Polynomial used to perform operations with polynomials such as additions or differentiation. This Polynomial class is used from class UniformBSpline in order to generate the corresponding polynomial functions for the interpolating curves.

Basically, an interface IInterpolator is again defined so that the actual interpolation, from a quantized model, is executed in a class named BSplineInterpolator. Here, the input is separated into normalized spans and then every possible path is calculated according to the number of simultaneously notes and the degree of the curves, which is 3 currently.

Once the sequence is interpolated the polynomials are stored in an interpolated model as follows:

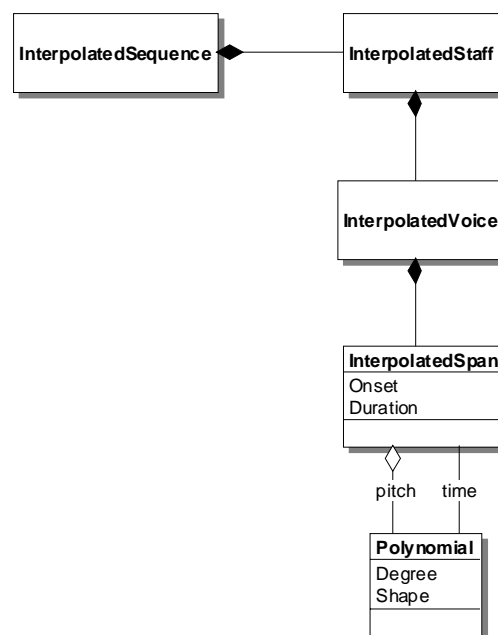


Figure 2.4 Interpolated model

Note that, since a certain span might have more than one path due to the existence of a close chord, each span has a collection of polynomials for the pitch component of the curve.

2.6 RSHP Modeling

The last phase in the music modeling process is to generate an instance of the RSHP metamodel that represents the original input sequence. An interface IRSHPzator is defined so that the current implementation in RSHPzator performs the model translation. After loading the domain from the

given repository, the RSHPzator generates the Sequence Artifact as explained in Part IX.

2.7 The CAKE Studio Manager

In the parent package, named MIKE, there are some classes used to incorporate MIKE to the CAKE Studio. Basically there is a Manager that declares the kind of indexing. It is the one in charge of starting the whole MIR process and storing the artifacts in the repository as well as creating some others for queries. In addition, the MIKEManager is in charge of creating the domain in the repository in the first time it is executed.

Part XI: Epilogue

1 Conclusions

Considering the initial objectives proposed it can be affirmed that all of them have been achieved.

Even though the project leaves many points where an extension and future work can be done, the main objective of providing a metamodel for music information in the RSHP information representation metamodel that allows the music reuse with the CAKE Engine has been achieved.

Moreover, the whole thing has a solid mathematical basis that solves all of the common problems in the music information retrieval process. The main advantage of MIKE is that it solves, in a single system, all these problems. Nowadays there are many approaches to the music information retrieval, but none of them is as extensive as MIKE at least about the comparison capabilities.

The main disadvantage of the current methods of music information retrieval is that they are not thought for the actual music information. There are many proposals based on probabilistic models that do not offer reliability. Some others are based on text information retrieval techniques applied to some textual representation of the music such as the GUIDO music notation. Others apply regular expressions searches to these textual representations, but the issue is the same: music is not text.

On the other hand, MIKE is thought for music since the beginning so that every decision made was focused only on music information. It uses a mathematical model that solves in a single system all the General Requirements considered and it is easily extensible to include more information thanks to the versatility of the RSHP metamodel.

This project opens many interesting doors for the music information retrieval by establishing the theoretical basis for the whole process. Much more work is remaining as seen in Part X, so that some other contributions to MIKE would lead to an important and strong reference point in the field of music information retrieval.

2 Future Work

As seen in Section X, many parts of the music information retrieval process are not yet implemented. Phases such as the Quantization or the Minimization would be suitable for a future work upon MIKE.

On the other hand, an important extension to consider is to definitely include the Kilian-Hoos voice separation algorithm (or an improved one). The point is that the algorithm needs some kind of feedback from the user in order to assign a value to each of the penalization constants (see Section 5.2.3 of the Part VII). Therefore, a good future work would be to provide some kind of staff visualization window for the CAKE Studio so that the user can see the intermediate voice separations and change the penalization constants accordingly. A good point would be to offer the possibility of fixing the separation note by note with a single action like clicking on it and swapping the voice assigned.

Once the voice separation is fully integrated, it would be possible to model into a single Sequence artifact several Staff subartifacts so that several instruments can be included in a single piece. Therefore, it could be possible to compare instruments one by one or as a whole, like the case with voices.

Another point is to add some extra information to the current model such as metadata about the piece, instruments or whatever. Another good point is to offer the possibility of comparing pieces not only by score time like now, but also by real time. The thing would be to change the action of the RSHP so that it becomes an artifact containing terms for both cases.

However, the most important and critical applicable work to MIKE is about the domain definition for the duration and derivative values. Right now the values are split into intervals that are linked one to each other so that the CAKE Engine penalizes distances. However, this interval definition is almost random right now.

The point is that intervals with a lot of occurrences should have more accuracy than those with a few occurrences. Therefore, the best solution would be to do a statistic study with a large MIDI database so that we could see how many occurrences appear for each derivative and duration value. Thus, with a histogram it would be possible to perform an optimal distribution of the domain.

In addition, the current model is open to modifications. A possible modification would be to add the information about the other voices not for each RSHP in a single span, but in the own Span artifact. Another extension is to add some values of the second derivative in the RSHPs so that the function that maps a polynomial to a RSHP would become surjective.

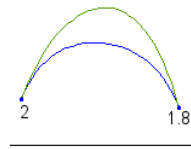


Figure 2.1 Surjective mapping to RHSP

Right now, both curves (they are already the first derivatives) in Figure 2.1 are treated as equal because the current model does not have collect information about the curvature. However, it is clear that they are not the same, the green one changes much more than the blue one. It can be easily demonstrated that by adding the value of the second derivative evaluated in $u = 0$ is enough to make the mapping surjective.

If the derivative polynomial is in the form $au^2 + bu + c$, the RSHP already contains the value of c since it is evaluated in 0. On the other hand, it has the value of $a + b + c$ when evaluated in $u = 1$. This is why the curvature is not taken into account. However, the second derivative has the form $2au + b$, so that evaluating it in $u = 0$ would give the value of b and hence the value of a . This way, the polynomial is totally defined in the RSHP.

This change would mean a new artifact type to add or the usage of new concept orders in the RSHPs, but will assure that there will not be false positives.

Therefore, several extensions can be applied upon MIKE, and all of them are welcome.

3 Project Budget

In order to specify the budget of the current herein described, there are established some assumptions:

- 8 full working hours per day.
- 20 days per month.

Moreover, the project can be developed by two different kinds of workers:

- Senior engineer with experience and research skills, responsible of the research area and schedule. A salary of 155 €/day is established¹.
- Junior engineer responsible of the programming tasks, with a salary of 105 €/day.

In addition, the identified tasks needed to accomplish the project are:

- Documentation about the MIDI standard and the SMF format.
- Documentation about the RSHP metamodel and the CAKE Engine.
- Documentation about the state of the art in the field of music information retrieval.
- Goals and software scope establishment.
- Analysis.
- Research in the field of numerical analysis.
- Design.
- Implementation.
- Tests.
- Documentation that yields to the current report.

Therefore, the effort can be calculated as follows:

¹ Salaries containing all applicable taxes like 40% for Social Security.

Task	Worker	Estimated Days	Cost
Documentation on MIDI and SMF	Senior	7	1,085€
Documentation on RSHP and CAKE	Senior Junior	12	3,120€
Documentation on State of the art	Senior	16	2,480€
Goals establishment	Senior	3	465€
Analysis	Senior	7	1,085€
Research in numerical analysis	Senior	36	5,580€
Design	Senior Junior	7	1,830€
Implementation	Junior	15	1,575€
Test	Senior Junior	3	780€
Documentation	Senior Junior	8	2,080€
Total		114	20,080€

Table 3.1 Human effort cost estimation

Moreover, during the elaboration of the project there were needed some stuff spending as well as the purchase of books and related documentation stuff.

Stuff	Quantity	Unitary Cost	Cost
Notebook Samsung NP-X20	1 during 114 days	1,277€	199€
Printer Samsung ML-2510	1 during 114 days	86€	13€
Microsoft Windows XP Professional	1	Student License	0€
Microsoft Visual Studio .net 2003	1	Student License	0€
Microsoft Office 2003 Professional	1	Student License	0€
Reuse Studio with Indexing License	1	3,480€	3,480€
Paper	2000	0.01€	20€
Toner ink	1	64€	64€
CD-ROM	5	0.20€	1€
Internet connection	6 months	20€	120€
Books and related documentation	-	-	82€
Total			3,979€

Table 3.2 Stuff and documentation cost

Moreover, the project was developed as part of an Erasmus grant awarded during 5 months in the city of Mariehamn, in Finland. Thus, the applicable cost resulting from the grant is:

Concept	Quantity	Unitary Cost	Cost
Travel to and from Mariehamn	2	120€	240€
Housing	5 months	Included in the grant	0€
Subsistence allowance	5 months	450€	2,250€
Total			2,490€

Table 3.3 Erasmus grant estimated cost

Therefore, the total cost of the project before taxes is:

$$20,080 + 3,979 + 2,490 = 26,549€ \quad [3.1]$$

Finally, by applying taxes and the corresponding risk and profit percentages:

Concept	Cost
Total cost	26,549€
Risk (8%)	2,124€
Profit (14%)	3,717€
Subtotal	32,390€
VAT (16%)	5,182€
Total	37,572€

Table 3.4 Final cost calculation

Therefore, the total final cost of the project focused on a potential client would be 37,572€.

Part XII: Source Code

1 MIKE

1.1 MikeManagerFactory

```
using System;

using CAKE;
using CAKE.Managers;

namespace MIKE {
    public class MIKEManagerFactory : CAKEManagerFactory{
        public override string ModuleNamespace{
            get{
                return "MIKE";
            }
        }
        protected override void AddManagerNames(){
            base.AddManager(MIKEManager.DisplayName, MIKEManager.Description,
                typeof(MIKEManager).Name, MIKEManager.Version, MIKEManager.Projects);
        }
        public override CAKEManager CreateManagerByName(string managerName){
            if(managerName == typeof(MIKEManager).Name){
                return new MIKEManager(this);
            }
            return null;
        }
        public override CAKELicenseInfo GetLicenseInfo(string managerName){
            return null;
        }
        protected override void RegisterIndexers(){
            base.RegisterIndexer(MIKEManager.KnownExtensions,
                new MIKEIndexerCreator());
        }
    }
}
```

1.2 MIKEIndexerCreator

```
using System;

using CAKE;
using CAKE.Managers;
using CAKE.IndexingServices;

namespace MIKE {
    public class MIKEIndexerCreator : IFileIndexerCreator {
        public IFileIndexer CreateFileIndexer(IndexingManager indexerManager) {
            return new MIKEManager(indexerManager);
        }
    }
}
```

1.3 MIKEManager

```
using System;
using System.ComponentModel;
using System.Windows.Forms;
using System.Collections;
using WindowDockerLib;

using CAKE;
using CAKE.Managers;
using CAKE_Indexer;
using CAKE.IndexingServices;
```

```
using Toub.Sound.Midi;
using MIKE.Preprocessing;
using MIKE.Preprocessing.Silly;
using MIKE.VoiceSeparation;
using MIKE.VoiceSeparation.Silly;
using MIKE.Quantization;
using MIKE.Quantization.Silly;
using MIKE.Interpolation;
using MIKE.RSHPzation;

namespace MIKE {
    public class MIKEManager : CAKEManagerIndexable{

        protected IPreprocessor _prep;
        protected IVoiceSeparator _sep;
        protected IQuantizator _quant;
        protected IInterpolator _inte;
        protected IRSHPzator _rshp;

        internal const string DisplayName = "MIKE";
        internal const string Description =
            "Music Indexer based on the CAKE Engine";
        internal static Version Version{
            get{
                return new Version(1, 0);
            }
        }
        internal static ProjectInfoCollection Projects{
            get{
                ProjectInfoCollection projects = new ProjectInfoCollection();
                projects.Add(new ProjectInfo("MIKE Project"));
                return projects;
            }
        }
        internal static String[] KnownExtensions{
            get{
                return new string[] { "midi", "mid" };
            }
        }

        private bool _isDestroyed;
        public override bool IsDestroyed{
            get{
                return _isDestroyed;
            }
        }

        protected internal MIKEManager(MIKEManagerFactory f) : base(f){
            this._isDestroyed = false;
            _prep = new SillyPreprocessor();
            _sep = new SillySeparator();
            _quant = new SillyQuantizator();
            _inte = new BSplineInterpolator(3);
            _rshp = new RSHPzator();
        }
        protected internal MIKEManager(IndexingManager idxManager) :
            base(idxManager){
            _prep = new SillyPreprocessor();
            _sep = new SillySeparator();
            _quant = new SillyQuantizator();
            _inte = new BSplineInterpolator(3);
            _rshp = new RSHPzator();
        }
        protected override void AddManagerForms(){
            this.AddForm(typeof(MIKEManagerMainForm).FullName);
        }
        protected override CAKEItem OnCreate(ProjectInfo projectType, string name){
            CreateNewProject(name);
            return null;
        }
        protected override void OnConnect(){
            return;
        }
        protected override void OnDisconnecting(CancelEventArgs e){

```

```

        return;
    }
    protected override void OnDisconnect(){
        return;
    }
    protected override void OnDestroy(){
        if(!this.IsDestroyed){
            _isDestroyed = true;
            if(base.MainForm != null){
                base.MainForm.Close();
                base.MainForm.Dispose();
            }
            base._mainForm = null;
        }
    }
    protected override bool OnLoad(string location){
        return true; /** @todo OnLoad */
    }
    public override bool IsDependentForm(DockableForm formInstance){
        return formInstance.Equals(base.MainForm);
    }
    public override Form CreateFormByName(string formTypeName){
        return this.CreateDockedFormByName(formTypeName);
    }
    public override DockableForm CreateDockedFormByName(string formTypeName){
        if(formTypeName == typeof(MIKEManagerMainForm).FullName){
            return this.GetMainForm();
        }
        return null;
    }
    public override bool Save(string location){
        return true; /** @todo Save */
    }
    internal void CreateNewProject(string name){
        this.GetMainForm();
        base.GetDocker().Add(base.MainForm, true, DockStyle.Left);
        MIKEManagerMainForm frmChild = new MIKEManagerMainForm(this);
        frmChild.MdiParent = base.GetDocker().DockingForm;
    }
    internal MIKEManagerMainForm GetMainForm(){
        if(base.MainForm != null && !base.MainForm.IsDisposed){
            return (MIKEManagerMainForm)base.MainForm;
        }
        base._mainForm = new MIKEManagerMainForm(this);
        return (MIKEManagerMainForm)base.MainForm;
    }
    public override string[] AvailableExtensions{
        get{
            return KnownExtensions;
        }
    }
    protected override Artifact OnBatchIndex(IndexingManager idxManager,
        string fullName){
        CAKEEngine engine = idxManager.Repository;
        /** Start the MIR process
        MidiSequence mSeq = MidiSequence.Import(fullName);
        PreprocessedMidiSequence pSeq = _prep.Preprocess(mSeq);
        SeparatedSequence sSeq = _sep.SeparateVoices(pSeq);
        QuantizedSequence qSeq = _quant.Quantize(sSeq);
        InterpolatedSequence iSeq = _inte.Interpolate(qSeq);
        Artifact rSeq = _rshp.RSHPTize(iSeq, engine, fullName, false);

        rSeq.Save(false);
        return rSeq;
        */
    }
    protected override Artifact OnBatchQuery(IndexingManager idxManager,
        string fullName){
        CAKEEngine engine = idxManager.Repository;
        /** Start the MIR process
        MidiSequence mSeq = MidiSequence.Import(fullName);
        PreprocessedMidiSequence pSeq = _prep.Preprocess(mSeq);
        SeparatedSequence sSeq = _sep.SeparateVoices(pSeq);

```

```

        QuantizedSequence qSeq = _quant.Quantize(sSeq);
        InterpolatedSequence iSeq = _inte.Interpolate(qSeq);
        Artifact rSeq = _rshp.RSHPtize(iSeq, engine, fullName, true);

        return rSeq;
    }
    protected override Artifact OnIndex(Indexer idx){
        CAKEEngine eng = idx.Repository;
        return null;
    }
    protected override Artifact OnQuery(Indexer idx){
        return null;
    }
    public void CreateDomain(CAKEEngine Engine) {
        Engine.Clear();
        Engine.Load(true);
        Engine.DeleteAndSaveArtifacts();

        // The language for MIKE
        Language music = new Language(Engine, "MI", "MIKE", "MIKE Language",
            null, 30000, 0);

        // Artifact Types
        ArtifactType atSequence = new ArtifactType(Engine, "Sequence", 0, null,
            30000, 0);
        ArtifactType atStaff = new ArtifactType(Engine, "Staff", 0, null,
            30001, 0);
        ArtifactType atVoice = new ArtifactType(Engine, "Voice", 0, null,
            30002, 0);
        ArtifactType atSpan = new ArtifactType(Engine, "Span", 0, null,
            30003, 0);
        ArtifactType atDerivative = new ArtifactType(Engine, "Derivatives", 0,
            null, 30004, 0);

        // Types of Term
        TermSemanticItem tsiDerivatives = newTermSemanticItem(Engine,
            "Derivatives", 30000, 0);
        TermSemanticItem tsiDurationsScore = newTermSemanticItem(Engine,
            "Score Durations", 30001, 0);

        // Types of RSHP
        SemanticItem siAssociation = Engine.SemanticItemFromJC(300);
        if(siAssociation == null){
            siAssociation = new SemanticItem(Engine, "Association", null, false,
                1, true, true, true, true, "Related", "Related", false, "RT",
                "RT", 300, 0);
        }
        SemanticItem siConcave = newSemanticItem(Engine, "Concave", 30000, 0);
        SemanticItem siConvex = newSemanticItem(Engine, "Convex", 30001, 0);
        SemanticItem siFlat = newSemanticItem(Engine, "Flat", 30002, 0);

        // Intervals for the 1st derivatives as Terms
        ArrayList derivativeIntervals = new ArrayList();
        derivativeIntervals.Add(newTerm(Engine, "(-inf, -14.5)", tsiDerivatives,
            music, null, 0));
        for(int i = 13; i >= 0; i--) {
            derivativeIntervals.Add(newTerm(Engine, "[-"+(i+1)+".5, -"+i+".5)",
                tsiDerivatives, music, null, 0));
        }
        derivativeIntervals.Add(newTerm(Engine, "[-0.5, 0.5)", tsiDerivatives,
            music, null, 0));
        for(int i = 0; i < 14; i++) {
            derivativeIntervals.Add(newTerm(Engine, "["+i+".5, "+(i+1)+".5)",
                tsiDerivatives, music, null, 0));
        }
        derivativeIntervals.Add(newTerm(Engine, "[14.5, +inf)", tsiDerivatives,
            music, null, 0));

        // Durations for the notes in score time
        ArrayList scoreDurationIntervals = new ArrayList();
        scoreDurationIntervals.Add(newTerm(Engine, "[2, 3)",
            tsiDurationsScore, music, null, 0));
        scoreDurationIntervals.Add(newTerm(Engine, "[3, 4)",

```

```

tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[4, 5)",
tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[5, 7)",
tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[7, 10)",
tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[10, 14)",
    tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[14, 20)",
tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[20, 28)",
tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[28, 40)",
tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[40, 56)",
tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[56, 80)",
tsiDurationsScore, music, null, 0));
scoreDurationIntervals.Add(newTerm(Engine, "[80, +inf)",
tsiDurationsScore, music, null, 0));

// Domain creation
Artifact domain = Engine.DomainArtifact;
if(domain != null) {
    // Relationships among 1st derivative value terms
    for(int i = 0; i < derivativeIntervals.Count-1; i++) {
        RSHP rshpT = new RSHP(domain, siAssociation, false, false,
            0, false, false, false, false, false, 0, 0);
        KE ket1 = new KE(rshpT, derivativeIntervals[i] as Term,
            1, 0, 0, 0, 0);
        KE ket2 = new KE(rshpT, derivativeIntervals[i+1] as Term,
            2, 0, 0, 0, 0);
    }
    // Relationships among score time terms
    for(int i = 0; i < scoreDurationIntervals.Count-1; i++) {
        RSHP rshpT = new RSHP(domain, siAssociation, false, false, 0,
            false, false, false, false, false, 0, 0);
        KE ket1 = new KE(rshpT, scoreDurationIntervals[i] as Term,
            1, 0, 0, 0, 0);
        KE ket2 = new KE(rshpT, scoreDurationIntervals[i+1] as Term,
            2, 0, 0, 0, 0);
    }
} else {
    MessageBox.Show("There was an error creating the domain.",
        "Domain Not Created", MessageBoxButtons.OK, MessageBoxIcon.Error);
    return;
}
Engine.Save(false);
Engine.CalculateTermDistances(
    CalculateDistanceType.CalculateOnlyOneLevel, false, domain);
MessageBox.Show("The domain was created successfully.",
    "Domain Created", MessageBoxButtons.OK, MessageBoxIcon.Information);
}
// Auxiliar functions to create the domain with default parameters
protected SemanticItem newSemanticItem(CAKEEngine engine, string category,
    int JC, int databaseCode) {
    return new SemanticItem(engine, category, null, false, 180, false,
        false, false, false, null, null, false, null, null, JC, databaseCode);
}
protected TermSemanticItem newTermSemanticItem(CAKEEngine engine,
    string description, int JC, int databaseCode) {
    return new TermSemanticItem(engine, description,
        engine.TermSemanticItemFromJJC(100), false, 1, 0, 0, 0.00001, 0.00001,
        false, false, JC, databaseCode);
}
protected Term newTerm(CAKEEngine engine, string normalizedTermName,
    TermSemanticItem kind, Language language, SemanticItem semanticItem,
    int databaseCode) {
    return new Term(engine, normalizedTermName, kind, language, semanticItem,
        false, 0, 0, null, null, null, null, null, null, null, true,
        0, 0, null, 0, 0, false, false, 0, 0, 0.00001, 0.00001, databaseCode);
}
}
}

```



```
}
```

1.4 MIKEManagerMainForm

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

using WindowDockerLib;
using CAKE.Managers;
using CAKE;

namespace MIKE {
    public class MIKEManagerMainForm : DockableForm {
        private System.ComponentModel.Container components = null;
        private System.Windows.Forms.Button button1;

        private readonly MIKEManager _manager;

        public MIKEManagerMainForm(MIKEManager workingManager) : base() {
            _manager = workingManager;
            InitializeComponent();
        }
        protected override void Dispose( bool disposing ) {
            if( disposing ) {
                if(components != null) {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        #region Código generado por el Diseñador de windows Forms
        private void InitializeComponent() {
            this.button1 = new System.Windows.Forms.Button();
            this.SuspendLayout();

            this.button1.Location = new System.Drawing.Point(8, 8);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(88, 32);
            this.button1.TabIndex = 0;
            this.button1.Text = "Create Domain";
            this.button1.Click += new System.EventHandler(this.button1_Click);

            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(292, 273);
            this.ControlBox = false;
            this.Controls.Add(this.button1);
            this.Name = "MIDIManagerMainForm";
            this.Text = "MIDIManagerMainForm";
            this.ResumeLayout(false);
        }
        #endregion

        private void button1_Click(object sender, System.EventArgs e) {
            if(_manager.IndexingManager != null) {
                button1.Enabled = false;
                _manager.CreateDomain(_manager.IndexingManager.Repository);
                button1.Enabled = true;
            }
            else{
                MessageBox.Show("You must be first connected to the repository",
                    "No Repository Open", MessageBoxButtons.OK,
                    MessageBoxIcon.Information);
            }
        }
    }
}
```

2 MIKE.Preprocessing

2.1 IPreprocessor

```
using System;
using Toub.Sound.Midi;

namespace MIKE.Preprocessing {
    public interface IPreprocessor {
        PreprocessedMidiSequence Preprocess(MidiSequence seq);
    }
}
```

2.2 PreprocessedMidiSequence

```
using System;
using System.Collections;
using System.IO;

namespace MIKE.Preprocessing {
    public class PreprocessedMidiSequence : IEnumerable{
        protected ArrayList _tracks;
        protected int _division;

        public int Count {
            get { return this._tracks.Count; }
        }
        public int Division {
            get { return _division; }
        }
        public PreprocessedMidiSequence(int division){
            _division = division;
            this._tracks = new ArrayList();
        }

        public void Add(PreprocessedMidiTrack trk) {
            this._tracks.Add(trk);
        }
        public PreprocessedMidiTrack this[int index] {
            get { return (PreprocessedMidiTrack)(this._tracks[index]); }
            set { this._tracks[index] = value; }
        }
        public IEnumerator GetEnumerator() {
            return this._tracks.GetEnumerator();
        }
    }
}
```

2.3 PreprocessedMidiTrack

```
using System;
using System.Collections;
using System.IO;

namespace MIKE.Preprocessing {
    public class PreprocessedMidiTrack : IEnumerable{
        protected ArrayList _notes;

        public int Count {
            get { return this._notes.Count; }
        }
    }
}
```

```
public PreprocessedMidiTrack() {
    this._notes = new ArrayList();
}

public void Add(PreprocessedNote fig) {
    this._notes.Add(fig);
}

internal void SortByOnset() {
    this._notes.Sort(new PreprocessedNote.OnsetComparer());
}

public PreprocessedNote this[int index] {
    get { return (PreprocessedNote)(this._notes[index]); }
    set { this._notes[index] = value; }
}

public virtual IEnumerator GetEnumerator() {
    return _notes.GetEnumerator();
}
}
}
```

2.4 PreprocessedMidiNote

```
using System;
using System.Collections;

namespace MIKE.Preprocessing {
    public class PreprocessedNote {
        private long _onset;
        private int _duration;
        private byte _pitch;

        public long Onset {
            get { return this._onset; }
        }
        public int Duration {
            get { return this._duration; }
        }
        public byte Pitch {
            get { return this._pitch; }
        }
        public long Offset {
            get { return this._onset + this._duration; }
        }

        public PreprocessedNote(long onset, int duration, byte pitch) {
            this._onset = onset;
            this._duration = duration;
            this._pitch = pitch;
        }

        public static bool operator <= (PreprocessedNote n1, PreprocessedNote n2) {
            return n1.Onset <= n2.Onset;
        }
        public static bool operator >= (PreprocessedNote n1, PreprocessedNote n2) {
            return n1.Onset >= n2.Onset;
        }

        public bool Overlap(PreprocessedNote n2) {
            return (this.Onset <= n2.Onset && n2.Onset < this.Offset) ||
                (n2.Onset <= this.Onset && this.Onset < n2.Offset);
        }

        public override string ToString() {
            return this.Onset + "\t" + this.Duration + "\t" + this.Pitch;
        }

        public static string NoteName(byte note) {
            // Get the octave and the pitch within the octave
            int octave = note / 12;
            int pitch = note % 12;

            // Translate the pitch into a note name
            string name;
            switch(pitch) {
                case 0: name = "C"; break;
            }
        }
    }
}
```

```

        case 1: name = "C#"; break;
        case 2: name = "D"; break;
        case 3: name = "D#"; break;
        case 4: name = "E"; break;
        case 5: name = "F"; break;
        case 6: name = "F#"; break;
        case 7: name = "G"; break;
        case 8: name = "G#"; break;
        case 9: name = "A"; break;
        case 10: name = "A#"; break;
        case 11: name = "B"; break;
        default: name = ""; break;
    }
    // Append the octave onto the name
    return name + octave;
}
public class OnsetComparer : IComparer {
    public int Compare(object x, object y) {
        PreprocessedNote X = x as PreprocessedNote;
        PreprocessedNote Y = y as PreprocessedNote;

        // Compare the onset times
        return X.Onset.CompareTo(Y.Onset);
    }
}
}
}
}

```

2.5 MIKE.Preprocessing.Silly

2.5.1 SillyPreprocessor

```

using System;
using Toub.Sound.Midi;
using MIKE.Preprocessing;

namespace MIKE.Preprocessing.Silly {
    public class SillyPreprocessor: IPreprocessor {
        public PreprocessedMidiSequence Preprocess(MidiSequence seq) {
            if(seq.Format != 0 && seq.Format != 1)
                throw new ArgumentException(
                    "SMF format "+ seq.Format + " not supported");

            PreprocessedMidiSequence pSeq =
                new PreprocessedMidiSequence(seq.Division);
            foreach(MidiTrack trk in seq) {
                PreprocessedMidiTrack pTrk = new PreprocessedMidiTrack();

                long[] onsets = new long[128];
                long onset = 0;
                foreach(MidiEvent e in trk.Events) {
                    onset += e.DeltaTime;
                    if(e is NoteVoiceMidiEvent) {
                        NoteVoiceMidiEvent v = e as NoteVoiceMidiEvent;
                        if(IsNoteOff(v)) {
                            PreprocessedNote n = new PreprocessedNote(
                                onsets[v.Note], (int)(onset - onsets[v.Note]), v.Note);
                            pTrk.Add(n);
                        } else if(IsNoteOn(v)) {
                            onsets[v.Note] = onset;
                        }
                    }
                }
                pTrk.SortByOnset();
                if(pTrk.Count != 0){
                    pSeq.Add(pTrk);
                }
            }
            return pSeq;
        }
        protected bool IsNoteOff(MidiEvent e) {

```

```
        return (e is NoteOff || (e is NoteOn && ((NoteOn)e).Velocity == 0));
    }
    protected bool IsNoteOn(MidiEvent e) {
        return (e is NoteOn);
    }
    protected void DeltasToTotals(MidiTrack trk) {
        // update all delta times to be total times
        MidiEventCollection evs = trk.Events;
        long total = evs[0].DeltaTime;
        for(int i = 1; i < evs.Count; i++) {
            total += evs[i].DeltaTime;
            evs[i].DeltaTime = total;
        }
    }
}
}
```

3 MIKE.VoiceSeparation

3.1 IVoiceSeparator

```
using System;
using MIKE.Preprocessing;

namespace MIKE.VoiceSeparation {
    public interface IVoiceSeparator {
        SeparatedSequence SeparateVoices(PreprocessedMidiSequence pSeq);
    }
}
```

3.2 SeparatedSequence

```
using System;
using System.Collections;
using System.IO;

namespace MIKE.VoiceSeparation {
    public class SeparatedSequence : IEnumerable{

        protected ArrayList _tracks;
        protected int _division;

        public int Count {
            get { return this._tracks.Count; }
        }
        public int Division {
            get { return _division; }
        }
        public SeparatedSequence(int division){
            _division = division;
            this._tracks = new ArrayList();
        }

        public void Add(SeparatedTrack sTrk) {
            this._tracks.Add(sTrk);
        }
        public SeparatedTrack this[int index] {
            get { return (SeparatedTrack)(this._tracks[index]); }
            set { this._tracks[index] = value; }
        }
        public IEnumerator GetEnumerator() {
            return this._tracks.GetEnumerator();
        }
    }
}
```

3.3 SeparatedTrack

```
using System;
using System.Collections;
using System.IO;

namespace MIKE.VoiceSeparation {
    public class SeparatedTrack : IEnumerable{
        protected ArrayList _voices;

        public int Count {
            get { return _voices.Count; }
        }
    }
}
```

```
public SeparatedTrack() {
    _voices = new ArrayList();
}

public int Add(SeparatedVoice svoice) {
    return _voices.Add(svoice);
}

public SeparatedVoice this[int index] {
    get { return (SeparatedVoice)(_voices[index]); }
    set { _voices[index] = value; }
}

public virtual IEnumerator GetEnumerator() {
    return _voices.GetEnumerator();
}
}
```

3.4 SeparatedVoice

```
using System;
using System.Collections;
using System.IO;
using MIKE.Preprocessing;

namespace MIKE.VoiceSeparation {
    public class SeparatedVoice : IEnumerable {
        protected ArrayList _notes;

        public int Count {
            get { return this._notes.Count; }
        }

        public SeparatedVoice() {
            this._notes = new ArrayList();
        }

        public void Add(SeparatedNote sNote) {
            _notes.Add(sNote);
            SortByOnset();
        }

        public void Remove(SeparatedNote sNote) {
            _notes.Remove(sNote);
            SortByOnset();
        }

        public int IndexOf(SeparatedNote sNote) {
            return this._notes.IndexOf(sNote);
        }

        internal void SortByOnset() {
            this._notes.Sort(new MIKE.VoiceSeparation.OnsetComparer());
        }

        public SeparatedNote this[int index] {
            get { return (SeparatedNote)(this._notes[index]); }
            set { this._notes[index] = value; }
        }

        public virtual IEnumerator GetEnumerator() {
            return _notes.GetEnumerator();
        }
    }
}
```

3.5 SeparatedNote

```
using System;
using System.Collections;
using MIKE.Preprocessing;

namespace MIKE.VoiceSeparation {
    public interface SeparatedNote {
        long Onset {
```

```

        get;
    }
    int Duration {
        get;
    }
    long Offset {
        get;
    }
    bool IsChord {
        get;
    }
}

public class OnsetComparer : IComparer {
    public int Compare(object x, object y) {
        SeparatedNote X = x as SeparatedNote;
        SeparatedNote Y = y as SeparatedNote;
        // Compare the onset times
        return X.Onset.CompareTo(Y.Onset);
    }
}
}

```

3.6 SeparatedSingleNote

```

using System;
using System.Collections;
using MIKE.Preprocessing;

namespace MIKE.VoiceSeparation {
    public class SeparatedSingleNote : SeparatedNote {
        private long _onset;
        private int _duration;
        private byte _pitch;

        public long Onset {
            get { return _onset; }
        }
        public int Duration {
            get { return _duration; }
        }
        public byte Pitch {
            get { return _pitch; }
        }
        public long Offset {
            get { return _onset + _duration; }
        }
        public bool IsChord {
            get { return false; }
        }

        public SeparatedSingleNote(long onset, int duration, byte pitch) {
            this._onset = onset;
            this._duration = duration;
            this._pitch = pitch;
        }
    }
}

```

3.7 SeparatedChord

```

using System;
using System.Collections;
using System.IO;
using MIKE.Preprocessing;

namespace MIKE.VoiceSeparation {
    public class SeparatedChord : SeparatedNote, IEnumerable {
        protected ArrayList _notes;
    }
}

```



```
private long _onset;
private int _duration;

public long Onset {
    get { return this[0].Onset; }
}
public int Duration {
    get { return (int)(this[Count-1].Offset-this[0].Onset); }
}
public long Offset {
    get { return this[Count-1].Offset; }
}
public bool IsChord {
    get { return true; }
}

public int Count {
    get { return _notes.Count; }
}

public SeparatedChord() {
    _notes = new ArrayList();
}

public void Add(SeparatedSingleNote sNote) {
    _notes.Add(sNote);
    SortByOnset();
}
public void Remove(SeparatedSingleNote sNote) {
    _notes.Remove(sNote);
    SortByOnset();
}
protected void SortByOnset() {
    _notes.Sort(new MIKE.VoiceSeparation.OnsetComparer());
}
public SeparatedSingleNote this[int index] {
    get { return (SeparatedSingleNote)(_notes[index]); }
    set { _notes[index] = value; }
}
public virtual IEnumerator GetEnumerator() {
    return _notes.GetEnumerator();
}
}
}
```

3.8 MIKE.VoiceSeparation.KilianHoos

3.8.1 KilianHoosVoiceSeparator

```
using System;
using System.Collections;
using MIKE.Preprocessing;

namespace MIKE.VoiceSeparation.KilianHoos {
    public class KilianHoosVoiceSeparator : IVoiceSeparator {
        protected const int N_VOICES = 2;
        protected const double K_pitch = 0.3;
        protected const double K_gap = 0.8;
        protected const double K_chord = 0.3;
        protected const double K_overlap = 0.1;

        public SeparatedSequence SeparateVoices(PreprocessedMidSequence pSeq) {
            return null;
        }

        public SeparatedSequence SeparateVoices(PreprocessedMidSequence pSeq) {
            ArrayList Ss = new ArrayList();
            foreach(PreprocessedMidTrack pTrk in pSeq) {
```

```

        if(pTrk.Count != 0){
            ArrayList B = CalculateVectorB(pTrk);
            ArrayList Y = CalculateSlicesY(B, pTrk);
            ArrayList S = new ArrayList();
            foreach(ArrayList y_i in Y) {
                ArrayList S_i = SeparateSlice(y_i, S);
                S.Add(S_i);
                // Remove overlaps and regularize chords
            }
            Ss.Add(S);
        }
    }
    return Ss;
}

#region Main Algorithm's Procedures

protected ArrayList SeparateSlice(ArrayList y_i, ArrayList S) {
    ArrayList S_i = InitializeS_i(y_i);
    ArrayList S_i_opt = S_i;
    int noImpr = 0;

    Random r = new Random();
    while(noImpr < y_i.Count*N_VOICES*3) {
        ArrayList neighbors = CalculateNeighbors(S_i);
        if(r.Next(10) <= 8) {
            if(neighbors.Count != 0){
                ArrayList S_min = neighbors[0] as ArrayList;
                foreach(ArrayList S_j in neighbors) {
                    if(C(S_j, S) < C(S_min, S)){
                        S_min = S_j;
                    }
                }
                S_i = S_min;
                Console.WriteLine(" Minima con C="+C(S_i, S)+" : ");
                foreach(KilianHoosSingleNote m in S_i) {
                    Console.WriteLine(m+", ");
                }
                Console.WriteLine();
            }else{
                noImpr = Int32.MaxValue-1;
            }
        }else{
            if(neighbors.Count != 0){
                S_i = neighbors[r.Next(neighbors.Count-1)] as ArrayList;
                Console.WriteLine(" Random con C="+C(S_i, S)+" : ");
                foreach(KilianHoosSingleNote m in S_i) {
                    Console.WriteLine(m+", ");
                }
                Console.WriteLine();
            }else{
                noImpr = Int32.MaxValue-1;
            }
        }
        if(C(S_i, S) < C(S_i_opt, S)) {
            S_i_opt = S_i;
            noImpr = 0;
        }else{
            noImpr++;
        }
    }
    return S_i_opt;
}

private ArrayList InitializeS_i(ArrayList y_i) {
    ArrayList S_i = new ArrayList();
    ArrayList temp = new ArrayList();
    // Add them to a temporal list, without chords
    foreach(KilianHoosSingleNote m in y_i) {
        KilianHoosSingleNote m2 = m.Clone();
        m2.Voice = 0;
        temp.Add(m2);
    }
    // Group into chords when possible
    KilianHoosNote last = null;

```

```

        foreach(KilianHoosSingleNote m in temp) {
            if(last != null && m.Onset == last.Onset) {
                // If has the same onset and duration as another one,
                // group them into a chord
                if(!last.IsChord) {
                    KilianHoosChord c = new KilianHoosChord();
                    c.Add(last as KilianHoosSingleNote);
                    last = c;
                }
                (last as KilianHoosChord).Add(m);
            }
            S_i.Add(m);
            last = m;
        }
        return S_i;
    }
    private ArrayList CalculateNeighbors(ArrayList S_i) {
        ArrayList neighbors = new ArrayList();
        // Create neighbors by changing voices note by note
        for(int i = 0; i < S_i.Count; i++){
            KilianHoosSingleNote m_i = S_i[i] as KilianHoosSingleNote;
            // Create a neighbor for each of the remaining voices of note m_i
            for(int v = m_i.Voice+1; v < N_VOICES; v++) {
                ArrayList neighbor = new ArrayList();
                // Add previous notes to the neighbor maintaining voice
                // and with no chord
                for(int j = 0; j < i; j++) {
                    KilianHoosSingleNote m_j = S_i[j] as KilianHoosSingleNote;
                    KilianHoosSingleNote m_j_2 = m_j.Clone();
                    m_j_2.Voice = m_j.Voice;
                    neighbor.Add(m_j_2);
                }
                // Add the current note with another voice
                KilianHoosSingleNote m_i_2 = m_i.Clone();
                m_i_2.Voice = v;
                neighbor.Add(m_i_2);
                // Add following notes to the neighbor maintaining voice
                // and with no chord
                for(int j = i+1; j < S_i.Count; j++){
                    KilianHoosSingleNote m_j = S_i[j] as KilianHoosSingleNote;
                    KilianHoosSingleNote m_j_2 = m_j.Clone();
                    m_j_2.Voice = m_j.Voice;
                    neighbor.Add(m_j_2);
                }
                // Group notes with same onset and voice into chords
                ArrayList grouped = new ArrayList();
                for(int j = 0; j < neighbor.Count; j++) {
                    if(!grouped.Contains(j as object)) {
                        KilianHoosSingleNote m_j = neighbor[j]
                        as KilianHoosSingleNote;
                        // Check the following notes
                        for(int k = j+1; k < neighbor.Count; k++) {
                            KilianHoosSingleNote m_k = neighbor[k]
                            as KilianHoosSingleNote;
                            if(m_j.Voice == m_k.Voice && m_j.Onset == m_k.Onset){
                                if(!m_j.IsChord){
                                    m_j.Chord = new ArrayList();
                                }
                                m_k.Chord = m_j.Chord;
                                grouped.Add(k as object);
                                grouped.Add(j as object);
                            }
                        }
                    }
                }
                neighbors.Add(neighbor);
            }
        }
        return neighbors;
    }
    protected double C(ArrayList S_i, ArrayList S) {
        double p = K_pitch*C_pitch(S_i, S);
        double g = K_gap*C_gap(S_i, S);
        double c = K_chord*C_chord(S_i);
    }

```

```

        double o = K_overlap*C_overlap(S_i, S);
        /*double r = p + (1-p)*g;
        r = r + (1-r)*c;
        r = r + (1-r)*o;
        return r;*/
        return p+g+c+o;
    }

#endregion

#region Input Splitting

protected ArrayList CalculateVectorB(PreprocessedMidiTrack pTrk) {
    ArrayList B = new ArrayList();
    // First index is the first note's
    B.Add((object)0);
    for(int i = 1; i < pTrk.Count; i++) {
        int j = (int)B[B.Count-1];
        // Check that overlaps with all previous notes
        // within the current slice
        while(j < i && pTrk[i].Overlap(pTrk[j])) {
            j++;
        }
        // If not, start new slice with that index
        if(i != j) {
            B.Add(i as object);
        }
    }
    return B;
}

protected ArrayList CalculateslicesY(ArrayList B,
PreprocessedMidiTrack pTrk) {
    ArrayList Y = new ArrayList();
    ArrayList y = null;
    for(int i = 0, b = 0; i < pTrk.Count; i++) {
        if(B.GetRange(b, B.Count-b).Contains(i as object)) {
            // If the note index is beyond the current slice,
            // create a new one
            y = new ArrayList();
            Y.Add(y);
            y.Add(new KilianHoossSingleNote(pTrk[i]));
            b++;
        }else{
            // If not, add it to the current slice
            y.Add(new KilianHoossSingleNote(pTrk[i]));
        }
    }
    return Y;
}

#endregion

#region Pitch Distance Penalty

protected byte cPitch(KilianHoossSingleNote m_j, byte p_l) {
    return cPitchNote(m_j, p_l).Pitch;
}

private KilianHoossSingleNote cPitchNote(KilianHoossSingleNote m_j,byte p_l){
    if(!m_j.IsChord) {
        // If m_j does not belong to a chord, return itself
        return m_j;
    }else{
        // If not, return the note within the chord that is
        // closest in pitch to p_l
        KilianHoossSingleNote m_c = m_j;
        foreach(KilianHoossSingleNote m_k in m_j.Chord) {
            if(Math.Abs(m_k.Pitch - p_l) < Math.Abs(m_c.Pitch - p_l)) {
                m_c = m_k;
            }
        }
        return m_c;
    }
}

protected KilianHoossSingleNote lonset(int v, ArrayList S) {

```

```

        KilianHoosSingleNote m_latest = null;
        int i = S.Count-1;
        // Search within the previous separations
        while(i >= 0 && m_latest == null){
            ArrayList S_i = S[i] as ArrayList;
            int j = S_i.Count-1;
            // Search within the notes of each separation
            while(j >= 0 && m_latest == null){
                KilianHoosSingleNote m_j = S_i[j] as KilianHoosSingleNote;
                if(m_j.Voice == v) {
                    m_latest = m_j;
                }
                j--;
            }
            i--;
        }
        return m_latest;
    }
    protected double C_pitch(ArrayList S_i, ArrayList S,
        KilianHoosSingleNote m_0) {
        // Take the last note in S with voice v
        KilianHoosSingleNote prevNote = lonset(m_0.Voice, S);
        double pVD = 0;
        if(prevNote == null) {
            // m_0 starts a new voice
            prevNote = m_0;
            pVD = 0.3;
        }
        foreach(KilianHoosSingleNote m_j in S_i) {
            if(m_j.Voice == m_0.Voice) {
                double pDist = (double)Math.Abs(cPitch(prevNote, m_j.Pitch) -
                    m_j.Pitch) / 128;
                pVD += (1-pVD)*pDist;
                if(!prevNote.IsChord || !m_j.IsChord ||
                    !prevNote.Chord.Equals(m_j.Chord)) {
                    prevNote = m_j;
                }
            }
        }
        return pVD;
    }
    protected double C_pitch(ArrayList S_i, ArrayList S) {
        double pD = 0;
        ArrayList usedVoices = new ArrayList();
        foreach(KilianHoosSingleNote m_j in S_i) {
            if(!usedVoices.Contains(m_j.Voice)){
                usedVoices.Add(m_j.Voice as object);
                pD += (1-pD)*C_pitch(S_i, S, m_j);
            }
        }
        return pD;
    }
}
#endregion

#region Gap Distance Penalty

protected double cGap(KilianHoosSingleNote m_g, ArrayList S) {
    if(lonset(m_g.Voice, S) == null) {
        // m_g starts a new voice
        return 0.1; // Penalize accordingly
    }
    ArrayList calculatedVoices = new ArrayList();
    KilianHoosSingleNote m_max = m_g;
    KilianHoosSingleNote m_max_v = null;
    int i = S.Count-1;
    // Search within the previous separations
    while(i >= 0 && calculatedVoices.Count != N_VOICES){
        ArrayList S_i = S[i] as ArrayList;
        int j = S_i.Count-1;
        // Search within the notes of each separation
        while(j >= 0 && calculatedVoices.Count != N_VOICES){
            KilianHoosSingleNote m_j = S_i[j] as KilianHoosSingleNote;
            if(!calculatedVoices.Contains(m_j.Voice as object)) {
                calculatedVoices.Add(m_j.Voice as object);
            }
        }
    }
}

```

```

        if(m_j.Voice == m_g.Voice) {
            m_max_v = m_j;
        }
        if(m_max.Offset > m_j.Offset) {
            m_max = m_j;
        }
    }
    j--;
}
i--;
}
if(m_g.Onset != m_max.Offset){
    return (m_g.Onset - m_max_v.Offset) / (m_g.Onset - m_max.Offset);
}else{
    return 0;
}
}
protected double C_gap(ArrayList S_i, ArrayList S) {
    double gD = 0;
    int cNotes = 0;
    ArrayList usedVoices = new ArrayList();
    foreach(KilianHoosSingleNote m_j in S_i) {
        if(!usedVoices.Contains(m_j.Voice)){
            usedVoices.Add(m_j.Voice as object);
            gD += cGap(m_j, S);
            cNotes++;
        }
    }
    gD /= cNotes;
    return gD;
}

#endregion

#region Chord Distance Penalty

protected double C_chord(ArrayList S_i) {
    double CD = 0;
    ArrayList usedChords = new ArrayList();
    foreach(KilianHoosSingleNote m in S_i) {
        if(m.IsChord && !usedChords.Contains(m.Chord)){
            usedChords.Add(m.Chord);
            double p = pDuration(m.Chord) +
                (1-pDuration(m.Chord))*pRange(m.Chord);
            p += (1-p)*pOnset(m.Chord);
            CD += (1-CD)*p;
        }
    }
    return CD;
}

protected double pRange(ArrayList c) {
    KilianHoosSingleNote p_highest = c[0] as KilianHoosSingleNote;
    KilianHoosSingleNote p_lowest = c[0] as KilianHoosSingleNote;
    foreach(KilianHoosSingleNote m_i in c) {
        if(m_i.Pitch > p_highest.Pitch) {
            p_highest = m_i;
        }else if(m_i.Pitch < p_lowest.Pitch) {
            p_lowest = m_i;
        }
    }
    double r = Math.Min((double)(p_highest.Pitch - p_lowest.Pitch) / 24, 1);
    return r;
}

protected double pDuration(ArrayList c) {
    KilianHoosSingleNote d_longest = c[0] as KilianHoosSingleNote;
    KilianHoosSingleNote d_shortest = c[0] as KilianHoosSingleNote;
    foreach(KilianHoosSingleNote m_i in c) {
        if(m_i.Duration > d_longest.Duration) {
            d_longest = m_i;
        }else if(m_i.Duration < d_shortest.Duration) {
            d_shortest = m_i;
        }
    }
    double r = 1 - (double)(d_longest.Duration / d_shortest.Duration);
}

```

```

        return r;
    }
    protected double pOnset(ArrayList c) {
        KilianHoosSingleNote o_earliest = c[0] as KilianHoosSingleNote;
        KilianHoosSingleNote o_latest = c[0] as KilianHoosSingleNote;
        KilianHoosSingleNote d_longest = c[0] as KilianHoosSingleNote;
        foreach(KilianHoosSingleNote m_i in c) {
            if(m_i.Onset > o_latest.Onset) {
                o_latest = m_i;
            } else if(m_i.Onset < o_earliest.Onset) {
                o_earliest = m_i;
            }
            if(m_i.Duration > d_longest.Duration) {
                d_longest = m_i;
            }
        }
        double r = (double)(o_latest.Onset - o_earliest.Onset) /
            d_longest.Duration;
        return r;
    }
}
#endregion

#region Overlap Distance Penalty

protected double C_overlap(ArrayList S_i, ArrayList S) {
    double oD = 0;
    ArrayList usedVoices = new ArrayList();
    foreach(KilianHoosSingleNote m in S_i) {
        if(!usedVoices.Contains(m.Voice as object)){
            usedVoices.Add(m.Voice as object);
            double oDist = C_overlap(S_i, S, m);
            oD += (1-oD)*oDist;
        }
    }
    return oD;
}

protected double C_overlap(ArrayList S_i, ArrayList S,
    KilianHoosSingleNote m_0) {
    KilianHoosSingleNote prevNote = lOnset(m_0.Voice, S);
    if(prevNote == null) {
        // m_0 starts the voice
        prevNote = m_0;
    }
    double oVD = 0;
    foreach(KilianHoosSingleNote m_j in S_i) {
        if(m_j.Voice == m_0.Voice) {
            double oDist = cOverlap(prevNote, m_j);
            oVD += (1-oVD)*oDist;
            if(!prevNote.IsChord || !m_j.IsChord ||
                !prevNote.Chord.Equals(m_j.Chord)) {
                prevNote = m_j;
            }
        }
    }
    return oVD;
}

protected double cOverlap(KilianHoosSingleNote m_j,
    KilianHoosSingleNote m_k) {
    if(m_j.Onset != m_k.Onset && m_j.Overlap(m_k)) {
        return 1-((double)(m_k.Onset-m_j.Onset)/m_j.Duration);
    } else {
        return 0;
    }
}
}
#endregion
}
}

```

3.8.2 KilianHoosNote

using System;

```
namespace MIKE.VoiceSeparation.KilianHoos {
    public interface KilianHoosNote {
        long Onset {
            get;
        }
        bool IsChord {
            get;
        }
        int Voice {
            get;
            set;
        }
    }
}
```

3.8.3 KilianHoosSingleNote

```
using System;
using System.Collections;
using MIKE.Preprocessing;

namespace MIKE.VoiceSeparation.KilianHoos {
    public class KilianHoosSingleNote : PreprocessedNote, KilianHoosNote {
        private ArrayList _chord;
        private int _voice;

        public ArrayList Chord {
            set {
                if (this.IsChord) {
                    this._chord.Remove(this);
                }
                this._chord = value;
                if (this.IsChord) {
                    this.Chord.Add(this);
                }
            }
            get { return this._chord; }
        }
        public bool IsChord {
            get { return this.Chord != null; }
        }
        public int Voice {
            set { this._voice = value; }
            get { return this._voice; }
        }
        public KilianHoosSingleNote(PreprocessedNote pNote) :
            base(pNote.Onset, pNote.Duration, pNote.Pitch) {
            this._chord = null;
            this._voice = -1;
        }
        protected KilianHoosSingleNote(long onset, int duration, byte pitch) :
            base(onset, duration, pitch) {
            this._chord = null;
            this._voice = -1;
        }
        public KilianHoosSingleNote Clone() {
            KilianHoosSingleNote m = new KilianHoosSingleNote(
                this.Onset, this.Duration, this.Pitch);
            return m;
        }
    }
}
```

3.8.4 KilianHoosChord

```
using System;
using System.Collections;

namespace MIKE.VoiceSeparation.KilianHoos {
    public class KilianHoosChord : KilianHoosNote, IEnumerable {
        private int _voice;
        public int Voice {
            get { return this._voice; }
        }
    }
}
```



```

        set{ this._voice = value; }
    }
    public long onset {
        get { return this[0].Onset; }
    }
    public bool IsChord {
        get { return true; }
    }
    private ArrayList _notes;

    public KilianHoosChord() {
        this._notes = new ArrayList();
    }
    public void Add(KilianHoosSingleNote m){
        this._notes.Add(m);
    }
    public KilianHoosSingleNote this[int index] {
        get { return this._notes[index] as KilianHoosSingleNote; }
    }
    public IEnumerator GetEnumerator() {
        return this._notes.GetEnumerator();
    }
}
}

```

3.9 MIKE.VoiceSeparation.Silly

3.9.1 SillyVoiceSeparator

```

using System;
using System.Collections;
using MIKE.Preprocessing;
using MIKE.VoiceSeparation;

namespace MIKE.VoiceSeparation.Silly {
    public class SillySeparator : IVoiceSeparator {

        public SeparatedSequence SeparateVoices(PreprocessedMidiSequence pSeq) {
            SeparatedSequence sSeq = new SeparatedSequence(pSeq.Division);
            SeparatedTrack sTrk = new SeparatedTrack();
            foreach(PreprocessedMidiTrack pVoice in pSeq) {
                SeparatedVoice sVoice = new SeparatedVoice();

                SeparatedNote prev = null;
                foreach(PreprocessedNote m in pVoice){
                    SeparatedSingleNote ssn = new SeparatedSingleNote(
                        m.Onset, m.Duration, m.Pitch);
                    if(prev != null && m.Onset == prev.Onset){
                        if(!prev.IsChord){
                            SeparatedChord c = new SeparatedChord();
                            c.Add(prev as SeparatedSingleNote);
                            c.Add(ssn);
                            prev = c;
                        }else{
                            (prev as SeparatedChord).Add(ssn);
                        }
                    }else{
                        if(prev != null){
                            sVoice.Add(prev);
                        }
                        prev = ssn;
                    }
                }
                sVoice.Add(prev);
                sTrk.Add(sVoice);
            }
            sSeq.Add(sTrk);
            return sSeq;
        }
    }
}

```

4 MIKE.Quantization

4.1 IQuantizator

```
using System;
using MIKE.VoiceSeparation;

namespace MIKE.Quantization {
    public interface IQuantizator {
        QuantizedSequence Quantize(SeparatedSequence sSeq);
    }
}
```

4.2 QuantizedSequence

```
using System;
using System.Collections;
using System.IO;
using MIKE.VoiceSeparation;

namespace MIKE.Quantization {
    public class QuantizedSequence : IEnumerable{
        protected ArrayList _stoffs;

        public int Count {
            get { return _stoffs.Count; }
        }

        public QuantizedSequence(){
            _stoffs = new ArrayList();
        }

        public void Add(QuantizedStaff qStaff) {
            _stoffs.Add(qStaff);
        }

        public QuantizedStaff this[int index] {
            get { return (QuantizedStaff)(_stoffs[index]); }
            set { _stoffs[index] = value; }
        }

        public IEnumerator GetEnumerator() {
            return _stoffs.GetEnumerator();
        }
    }
}
```

4.3 QuantizedStaff

```
using System;
using System.Collections;
using System.Drawing;

namespace MIKE.Quantization {
    public class QuantizedStaff : IEnumerable {
        protected ArrayList _voices;

        public int Count {
            get { return _voices.Count; }
        }

        public QuantizedStaff() {
            _voices = new ArrayList();
        }

        public void Add(QuantizedVoice voice) {
```

4.4 QuantizedVoice

4.5 Quantized Durations

page 191

```
        TupletMinim = 32,  
        Minim = 48,  
        TupletSemibreve = 64,  
        Semibreve = 96,  
    }  
}
```

4.6 QuantizedNote

```
using System;  
  
namespace MIKE.Quantization {  
    public interface QuantizedNote {  
        int Onset {  
            get;  
        }  
        bool IsChord {  
            get;  
        }  
    }  
}
```

4.7 QuantizedSingleNote

```
using System;  
  
namespace MIKE.Quantization {  
    public class QuantizedSingleNote : QuantizedNote {  
        protected byte _pitch;  
        protected int _onset;  
        protected int _duration;  
  
        public byte Pitch {  
            get { return _pitch; }  
        }  
        public int Onset {  
            get { return _onset; }  
        }  
        public bool IsChord {  
            get { return false; }  
        }  
        public QuantizedSingleNote(byte pitch, int onset, int duration) {  
            _pitch = pitch;  
            _onset = onset;  
            _duration = duration;  
        }  
    }  
}
```

4.8 QuantizedChord

```
using System;  
using System.Collections;  
  
namespace MIKE.Quantization {  
    public class QuantizedChord : QuantizedNote, IEnumerable {  
        protected ArrayList _notes;  
  
        public int Onset {  
            get { return ((QuantizedSingleNote)_notes[0]).Onset; }  
        }  
        public bool IsChord {  
            get { return true; }  
        }  
        public int Count {  
            get { return _notes.Count; }  
        }  
    }  
}
```

```

    public QuantizedChord(){
        _notes = new ArrayList();
    }
    public void Add(QuantizedSingleNote note) {
        _notes.Add(note);
    }
    public QuantizedSingleNote this[int index] {
        get { return _notes[index] as QuantizedSingleNote; }
    }
    public IEnumerator GetEnumerator() {
        return _notes.GetEnumerator();
    }
}
}

```

4.9 MIKE.Quantization.Silly

4.9.1 SillyQuantizator

```

using System;
using MIKE.VoiceSeparation;

namespace MIKE.Quantization.Silly {
    public class SillyQuantizator : IQuantizator {
        public QuantizedSequence Quantize(SeparatedSequence sSeq) {
            int division = sSeq.Division;

            QuantizedSequence qSeq = new QuantizedSequence();
            foreach(SeparatedTrack sTrack in sSeq){
                QuantizedStaff qStaff = new QuantizedStaff();
                foreach(SeparatedVoice sVoice in sTrack){
                    QuantizedVoice qVoice = new QuantizedVoice();
                    foreach(SeparatedNote sNote in sVoice){
                        if(sNote.IsChord) {
                            QuantizedChord qChord = new QuantizedChord();
                            foreach(SeparatedSingleNote m in (sNote as SeparatedChord)){
                                int onset = ((int)(m.Onset*
                                    (int)QuantizedDurations.Crotchet)/division);
                                int duration = ((int)(m.Duration*
                                    (int)QuantizedDurations.Crotchet)/division);
                                QuantizedSingleNote qNote = new QuantizedSingleNote(
                                    m.Pitch, onset, duration);
                                qChord.Add(qNote);
                            }
                            qVoice.Add(qChord);
                        }else{
                            SeparatedSingleNote m = sNote as SeparatedSingleNote;
                            int onset = ((int)(m.Onset*
                                (int)QuantizedDurations.Crotchet)/division);
                            int duration = ((int)(m.Duration*
                                (int)QuantizedDurations.Crotchet)/division);
                            QuantizedSingleNote qNote = new QuantizedSingleNote(
                                m.Pitch, onset, duration);
                            qVoice.Add(qNote);
                        }
                    }
                    qStaff.Add(qVoice);
                }
                qSeq.Add(qStaff);
            }
            return qSeq;
        }
    }
}
}

```

5 MIKE.Interpolation

5.1 IInterpolator

```
using System;
using MIKE.Quantization;

namespace MIKE.Interpolation {
    public interface IInterpolator {
        InterpolatedSequence Interpolate(QuantizedSequence qSeq);
    }
}
```

5.2 InterpolatedSequence

```
using System;
using System.Collections;
using System.IO;
using MIKE.Quantization;

namespace MIKE.Interpolation {
    public class InterpolatedSequence : IEnumerable{
        protected ArrayList _stoffs;

        public int Count {
            get { return _stoffs.Count; }
        }

        public InterpolatedSequence(){
            _stoffs = new ArrayList();
        }

        public void Add(InterpolatedStaff iStaff) {
            _stoffs.Add(iStaff);
        }

        public InterpolatedStaff this[int index] {
            get { return (InterpolatedStaff)(_stoffs[index]); }
            set { _stoffs[index] = value; }
        }

        public IEnumerator GetEnumerator() {
            return _stoffs.GetEnumerator();
        }
    }
}
```

5.3 InterpolatedStaff

```
using System;
using System.Collections;
using System.Drawing;

namespace MIKE.Interpolation {
    public class InterpolatedStaff : IEnumerable {
        protected ArrayList _voices;

        public int Count {
            get { return _voices.Count; }
        }

        public InterpolatedStaff() {
            _voices = new ArrayList();
        }

        public void Add(InterpolatedVoice voice) {
```

```

        _voices.Add(voice);
    }
    public InterpolatedVoice this[int index] {
        get { return _voices[index] as InterpolatedVoice; }
    }
    public IEnumerator GetEnumerator() {
        return _voices.GetEnumerator();
    }
    public ArrayList GetDerivativesAt(int scoreTime) {
        ArrayList l = new ArrayList();
        foreach(InterpolatedVoice iVoice in _voices) {
            l.Add(iVoice.GetDerivativesAt(scoreTime));
        }
        return l;
    }
    public void Paint(Graphics g, int ss, int offset) {
        Pen[] pens = new Pen[]{ new Pen(Color.Blue, 1),
                                new Pen(Color.Green, 1),
                                new Pen(Color.Red, 1)};

        float s = 6f;
        offset = 1000;
        g.DrawLine(new Pen(Color.Gray, 1), 0, offset/2, 3000, offset/2);
        g.DrawLine(new Pen(Color.Gray, 1), 0, offset, 3000, offset);

        int e=0;
        foreach(InterpolatedVoice iVoice in _voices){
            PointF dp_1 = new PointF(0, 0);
            PointF dp_2 = new PointF(0, 0);
            foreach(InterpolatedSpan iSpan in iVoice) {
                foreach(Polynomial poly in iSpan) {
                    double d = 1f/20;
                    for(int j = 0; j <= 20; j++){
                        float x = (float)(iSpan.ScoreOnset+iSpan.ScoreDuration*j*d);
                        float dy = (float)(poly.Evaluate(j*d));
                        dp_2 = new PointF(x*s, offset/2-dy*s*2);
                        g.DrawLine(pens[e%pens.Length], dp_1, dp_2);
                        dp_1 = dp_2;
                        if(j%5==0)
                            g.DrawString(Math.Round(dy, 3)+"",
                                new Font("Arial Narrow", 6), Brushes.Black, dp_2);
                    }
                }
                e++;
            }
        }
    }
}

```

5.4 InterpolatedVoice

```

using System;
using System.Collections;

namespace MIKE.Interpolation {
    public class InterpolatedVoice : IEnumerable {
        protected ArrayList _spans;

        public int Count {
            get { return _spans.Count; }
        }
        public InterpolatedVoice() {
            _spans = new ArrayList();
        }
        public void Add(InterpolatedSpan span) {
            _spans.Add(span);
        }
        public InterpolatedSpan this[int index] {
            get { return _spans[index] as InterpolatedSpan; }
        }
        public IEnumerator GetEnumerator() {
            return _spans.GetEnumerator();
        }
    }
}

```

```

    }
    public InterpolatedSpan GetSpanAt(int scoreTime) {
        return GetSpanAt(scoreTime, 0, Count-1);
    }
    private InterpolatedSpan GetSpanAt(int scoreTime, int left, int right) {
        if(right < left){
            return null;
        }
        int iMid = (left+right)/2;
        InterpolatedSpan mid = _spans[iMid] as InterpolatedSpan;
        if(mid.EvaluateAt(0) <= scoreTime && scoreTime <= mid.EvaluateAt(1)){
            return mid;
        }
        if(mid.EvaluateAt(0) > scoreTime){
            return GetSpanAt(scoreTime, left, iMid-1);
        }else{
            return GetSpanAt(scoreTime, iMid+1, right);
        }
    }
    public ArrayList GetDerivativesAt(int scoreTime) {
        ArrayList l = new ArrayList();
        InterpolatedSpan iSpan = GetSpanAt(scoreTime);
        if(iSpan != null) {
            l = iSpan.GetDerivativesAt(scoreTime);
        }
        return l;
    }
}
}
}

```

5.5 InterpolatedSpan

```

using System;
using System.Collections;
using MIKE.Quantization;

namespace MIKE.Interpolation {

    public class InterpolatedSpan : IEnumerable {

        protected Polynomial _x;
        protected ArrayList _polys;
        protected int _scoreOnset;
        protected int _scoreDuration;

        public int ScoreOnset {
            get { return _scoreOnset; }
        }
        public int ScoreDuration {
            get { return _scoreDuration; }
        }
        public int Count {
            get { return _polys.Count; }
        }

        public InterpolatedSpan(Polynomial x, int scoreOnset, int scoreDuration) {
            _x = x;
            _polys = new ArrayList();
            _scoreOnset = scoreOnset;
            _scoreDuration = scoreDuration;
        }
        public void Add(Polynomial poly) {
            _polys.Add(poly);
        }
        public Polynomial this[int index] {
            get { return _polys[index] as Polynomial; }
        }
        public IEnumerator GetEnumerator() {
            return _polys.GetEnumerator();
        }
        public ArrayList GetDerivativesAt(int scoreTime) {
            double u = ((double)(scoreTime-ScoreOnset))/ScoreDuration;

```



```

        ArrayList l = new ArrayList();
        foreach(Polynomial p in _polys) {
            l.Add(p.Evaluate(u) as object);
        }
        return l;
    }
    public double EvaluateXAt(double u) {
        return _x.Evaluate(u)*ScoreDuration+ScoreOnset;
    }
}
}

```

5.6 Polynomial

```

using System;
using System.Collections;
using System.Text;

namespace MIKE.Interpolation {
    public class Polynomial {

        public static readonly Polynomial ZERO = new Polynomial(new double[] {0});
        public static readonly Polynomial ONE = new Polynomial(new double[] {1});

        protected double[] _coefs;
        public int Degree {
            get{ return _coefs.Length-1; }
        }

        public PolynomialShapes Shape {
            get {
                Polynomial ddd = Derivative().Derivative();
                if(ddd.Evaluate(0)<0)
                    return PolynomialShapes.Convex;
                else if(ddd.Evaluate(0)==0)
                    return PolynomialShapes.Flat;
                else
                    return PolynomialShapes.Concave;
            }
        }
        protected Polynomial(int degree) {
            this._coefs = new double[degree+1];
        }
        public Polynomial(double[] coefs) {
            _coefs = new double[coefs.Length];
            for(int i = 0; i < coefs.Length; i++){
                _coefs[i] = coefs[i];
            }
            Narrow();
        }

        public Polynomial Add(Polynomial p) {
            Polynomial r = new Polynomial(Math.Max(Degree, p.Degree));
            for(int i = 0; i <= Degree; i++){
                r._coefs[i] = _coefs[i];
            }
            for(int i = 0; i <= p.Degree; i++){
                r._coefs[i] += p._coefs[i];
            }
            r.Narrow();
            return r;
        }
        public Polynomial Subtract(Polynomial p) {
            Polynomial r = new Polynomial(Math.Max(Degree, p.Degree));
            for(int i = 0; i <= Degree; i++) {
                r._coefs[i] = _coefs[i];
            }
            for(int i = 0; i <= p.Degree; i++){
                r._coefs[i] -= p._coefs[i];
            }
            r.Narrow();
        }
    }
}

```

```

        return r;
    }
    public Polynomial Multiply(Polynomial p) {
        Polynomial r = new Polynomial(Degree+p.Degree);
        for(int i = Degree; i >= 0; i--){
            for(int j = p.Degree; j >= 0; j--){
                r._coefs[i+j] += _coefs[i]*p._coefs[j];
            }
        }
        r.Narrow();
        return r;
    }
    public Polynomial Multiply(double d) {
        Polynomial r = new Polynomial(_coefs);
        for(int i = this.Degree; i >= 0; i--){
            r._coefs[i] *= d;
        }
        return r;
    }
    public Polynomial Divide(double d) {
        Polynomial r = new Polynomial(_coefs);
        for(int i = this.Degree; i >= 0; i--){
            r._coefs[i] /= d;
        }
        return r;
    }
    public Polynomial Derivative() {
        if(Degree == 0){
            return Polynomial.ZERO;
        }
        Polynomial r = new Polynomial(Degree-1);
        for(int i = Degree; i > 0; i--){
            r._coefs[i-1] = _coefs[i]*i;
        }
        return r;
    }
    public double Evaluate(double x) {
        double r = _coefs[0];
        for(int i = 1; i <= Degree; i++){
            r += _coefs[i]*Math.Pow(x, i);
        }
        return r;
    }
    protected void Narrow() {
        int i = Degree;
        while(i > 0 && _coefs[i] == 0){
            i--;
        }
        if(i >= 0 && i != Degree){
            double[] aux = new double[i+1];
            for(i = i; i >= 0; i--){
                aux[i] = _coefs[i];
            }
            _coefs = aux;
        }
    }
    public ArrayList Solve(double val) {
        if(Degree == 3) {
            return SolveDegree3(val);
        } else if(Degree == 2) {
            return SolveDegree2(val);
        } else {
            return SolveDegree1(val);
        }
    }
    private ArrayList SolveDegree3(double val) {
        ArrayList res = new ArrayList();
        double d = _coefs[0] - val;
        double c = _coefs[1];
        double b = _coefs[2];
        double a = _coefs[3];

        double s1 = 2*Math.Sqrt(b*b-3*a*c);
        double s2 = ((27*a*a*d-9*a*b*c+2*b*b*b)*Math.Sign(a))/

```

```

        (2*(b*b-3*a*c)*Math.Sqrt(b*b-3*a*c));
double s3 = 3*Math.Abs(a);
double s4 = b/(3*a);

double s5 = -b*b+3*a*c;
double s6 = -2*b*b*b+9*a*b*c-27*a*a*d;
double s7 = Math.Pow(s6+Math.Sqrt(4*Math.Pow(
    s5, 3)+Math.Pow(s6, 2)),1d/3);

double r0 = -b/(3*a)-(Math.Pow(2, 1d/3)*s5)/(3*a*s7)+s7/
    (3*a*Math.Pow(2, 1d/3));
double r1 = (s1*Math.Cos(Math.Acos(-s2)/3))/s3-s4;
double r2 = (-s1*Math.Sin( Math.Asin(s2)/3 + Math.PI/3))/s3-s4;
double r3 = (s1*Math.Sin(Math.Asin(s2)/3))/s3-s4;

res.Add(r0 as object);
res.Add(r1 as object);
res.Add(r2 as object);
res.Add(r3 as object);
return res;
}
private ArrayList SolveDegree2(double val) {
    ArrayList res = new ArrayList();
    double d = _coefs[0] - val;
    double c = _coefs[1];
    double b = _coefs[2];

    double r1 = -c+Math.Sqrt(c*c-4*b*d)/(2*b);
    double r2 = -c-Math.Sqrt(c*c-4*b*d)/(2*b);

    res.Add(r1 as object);
    res.Add(r2 as object);
    return res;
}
private ArrayList SolveDegree1(double val) {
    ArrayList res = new ArrayList();
    double d = _coefs[0] - val;
    double c = _coefs[1];

    double r1 = -d/c;

    res.Add(r1 as object);
    return res;
}
public override string ToString() {
    StringBuilder r = new StringBuilder();
    for(int i = Degree; i > 0; i--){
        if(_coefs[i] != 0){
            r.Append(Math.Round(_coefs[i], 4)+"x^"+i+" ");
        }
    }
    if(this.Degree == 0 || _coefs[0] != 0){
        r.Append(Math.Round(_coefs[0], 4));
    }
    return r.ToString();
}
}
}
}

```

5.7 BSplineInterpolator

```

using System;
using System.Drawing;
using System.Collections;
using MIKE.Quantization;

namespace MIKE.Interpolation {
    public class BSplineInterpolator : IInterpolator {
        protected UniformBSpline _bSpline;

        public int Degree {
            get{ return _bSpline.Degree; }
        }
    }
}

```

```

    }

    public BSplineInterpolator(int degree) {
        _bspline = new UniformBSpline(degree);
    }

    protected ArrayList ExpandChords(QuantizedVoice qvoice) {
        ArrayList points = new ArrayList();
        for(int i = 0; i < qvoice.Count; i++){
            QuantizedNote n = qvoice[i];
            if(n.IsChord){
                QuantizedChord c = n as QuantizedChord;
                // Duplicate previous unfinished polynomials n_j
                for(int j = 1; j < Degree+1 && (i-j) >= 0; j++){
                    ArrayList l_n_j = points[i-j] as ArrayList;
                    // Create c.Count-1 copies of l_n_j
                    ArrayList copies = new ArrayList();
                    for(int k = 1; k < c.Count; k++){
                        foreach(ArrayList l_n_j_k in l_n_j){
                            ArrayList copy = new ArrayList();
                            foreach(Point n_j in l_n_j_k){
                                copy.Add(n_j);
                            }
                            copy.Add(new Point(c[k].Onset, c[k].Pitch));
                            copies.Add(copy);
                        }
                    }
                    // Append c[0] to previous polynomials
                    foreach(ArrayList l_n_j_k in l_n_j){
                        l_n_j_k.Add(new Point(c[0].Onset, c[0].Pitch));
                        copies.Add(l_n_j_k);
                    }
                    // Swap previous l_n_j for copies
                    points[i-j] = copies;
                }
                // Create a new ArrayList for polynomials starting at n_i
                ArrayList l_n_i = new ArrayList();
                foreach(QuantizedSingleNote n_i in c) {
                    ArrayList l_n_i_0 = new ArrayList();
                    l_n_i_0.Add(new Point(n_i.Onset, n_i.Pitch));
                    l_n_i.Add(l_n_i_0);
                }
                points.Add(l_n_i);
            }else {
                QuantizedSingleNote n_i = n as QuantizedSingleNote;
                // Append n_i to previous unfinished polynomials n_j
                for(int j = 1; j < Degree+1 && (i-j) >= 0; j++){
                    ArrayList l_n_j = points[i-j] as ArrayList;
                    foreach(ArrayList l_n_j_k in l_n_j){
                        l_n_j_k.Add(new Point(n_i.Onset, n_i.Pitch));
                    }
                }
                // Create a new ArrayList for polynomials starting at n_i
                ArrayList l_n_i = new ArrayList();
                ArrayList l_n_i_0 = new ArrayList();
                l_n_i_0.Add(new Point(n_i.Onset, n_i.Pitch));
                l_n_i.Add(l_n_i_0);
                points.Add(l_n_i);
            }
        }
        // Remove Last #Degree polynomials
        points.RemoveRange(points.Count-Degree, Degree);
        return points;
    }

    public InterpolatedSequence Interpolate(QuantizedSequence qSeq) {
        InterpolatedSequence iSeq = new InterpolatedSequence();
        InterpolatedStaff iStaff = new InterpolatedStaff();
        foreach(QuantizedStaff qStaff in qSeq){
            foreach(QuantizedVoice qvoice in qStaff) {
                InterpolatedVoice ivoice = new InterpolatedVoice();
                ArrayList points = ExpandChords(qvoice);
                // Create a new InterpolatedSpan foreach list l_i
                foreach(ArrayList l_i in points){
                    ArrayList l_i_0 = l_i[0] as ArrayList;

```

```

        Point l_i_0_0 = (Point)l_i_0[l_i_0.Count/2-1];
        Point l_i_0_1 = (Point)l_i_0[l_i_0.Count/2];
        InterpolatedSpan iSpan = new InterpolatedSpan(
            _bSpline.ComputeX(l_i[0] as ArrayList),
            l_i_0_0.X, l_i_0_1.X-l_i_0_0.X);
        // Add a polynomial foreach combination l_i_j
        foreach(ArrayList l_i_j in l_i) {
            // Interpolate points and get the blending polynomial
            Polynomial p_i_j = _bSpline.ComputeY(l_i_j).Derivative();
            iSpan.Add(p_i_j);
        }
        iVoice.Add(iSpan);
    }
    iStaff.Add(iVoice);
}
iSeq.Add(iStaff);
}
return iSeq;
}
}
}

```

5.8 UniformBSpline

```

using System;
using System.Drawing;
using System.Collections;

namespace MIKE.Interpolation {
    public class UniformBSpline {
        protected int _degree;

        public int Degree {
            get { return _degree; }
        }

        protected static readonly Polynomial[][] Blendings =
            new Polynomial[][] {
                new Polynomial[] {
                    new Polynomial(new double[] {0, 0, 0, 1}).Divide(6),
                    new Polynomial(new double[] {1, 3, 3, -3}).Divide(6),
                    new Polynomial(new double[] {4, 0, -6, 3}).Divide(6),
                    new Polynomial(new double[] {1, -3, 3, -1}).Divide(6)},
                new Polynomial[] {
                    new Polynomial(new double[] {0, 0, 0, 0, 1}).Divide(24),
                    new Polynomial(new double[] {1, 4, 6, 4, -4}).Divide(24),
                    new Polynomial(new double[] {11, 12, -6, -12, 6}).Divide(24),
                    new Polynomial(new double[] {11, -12, -6, 12, -4}).Divide(24),
                    new Polynomial(new double[] {1, -4, 6, -4, 1}).Divide(24)}
            };

        protected Polynomial[] Blending {
            get { return Blendings[Degree-3]; }
        }

        public UniformBSpline(int degree) {
            _degree = degree;
        }

        public Polynomial ComputeY(ArrayList points) {
            Polynomial y = Polynomial.ZERO;
            Polynomial[] b = Blendings[Degree-3];
            for(int i = 0; i <= Degree; i++){
                Point p_i = (Point)points[Degree-i];
                y = y.Add(b[i].Multiply(p_i.Y));
            }
            return y;
        }

        public Polynomial ComputeX(ArrayList points) {
            return new Polynomial(new double[] {0, 1});
        }
    }
}

```

6 MIKE.RSHPzation

6.1 IRSHPzator

```
using System;
using System.Collections;
using MIKE.Interpolation;
using CAKE;

namespace MIKE.RSHPzation {
    public interface IRSHPzator {
        Artifact RSHPtize(InterpolatedSequence iSeq,
            CAKEEngine Engine, string fullName, bool isQuery);
    }
}
```

6.2 RSHPzator

```
using System;
using MIKE.Interpolation;
using System.Collections;

using CAKE;

namespace MIKE.RSHPzation {
    public class RSHPzator : IRSHPzator {
        protected bool _domainLoaded;

        protected Language _language;

        protected ArtifactType _atSequence;
        protected ArtifactType _atStaff;
        protected ArtifactType _atVoice;
        protected ArtifactType _atSpan;
        protected ArtifactType _atDerivatives;

        protected TermSemanticItem _tsiDerivatives;
        protected TermSemanticItem _tsiScoreDurations;

        protected SemanticItem _siConcave;
        protected SemanticItem _siConvex;
        protected SemanticItem _siFlat;

        protected ArrayList _tNegativeDerivatives;
        protected ArrayList _tPositiveDerivatives;
        protected ArrayList _tScoreDurations;

        public RSHPzator() {
            _domainLoaded = false;
        }

        protected void LoadDomain(CAKEEngine Engine) {
            _language = Engine.LanguageFromJC(30000);

            _atSequence = Engine.ArtifactTypeFromJC(30000);
            _atStaff = Engine.ArtifactTypeFromJC(30001);
            _atVoice = Engine.ArtifactTypeFromJC(30002);
            _atSpan = Engine.ArtifactTypeFromJC(30003);
            _atDerivatives = Engine.ArtifactTypeFromJC(30004);

            _tsiDerivatives = Engine.TermSemanticItemFromJC(30000);
            _tsiScoreDurations = Engine.TermSemanticItemFromJC(30001);

            _siConcave = Engine.SemanticItemFromJC(30000);
            _siConvex = Engine.SemanticItemFromJC(30001);
        }
    }
}
```

```

_siFlat = Engine.SemanticItemFromJC(30002);

_tNegativeDerivatives = new ArrayList();
_tNegativeDerivatives.Add(Engine.Term("[ -0.5, 0.5)", _tsiDerivatives,
false));
for(int i = 1; i <= 14; i++) {
    _tNegativeDerivatives.Add(Engine.Term("[ -"+i+".5, -"+(i-1)+"+.5)",
        _tsiDerivatives, false));
}
_tNegativeDerivatives.Add(Engine.Term("(-inf, -14.5)", _tsiDerivatives,
false));

_tPositiveDerivatives = new ArrayList();
_tPositiveDerivatives.Add(Engine.Term("[ -0.5, 0.5)", _tsiDerivatives,
false));
for(int i = 0; i <= 13; i++) {
    _tPositiveDerivatives.Add(Engine.Term("[ "+i+".5, "+(i+1)+"+.5)",
        _tsiDerivatives, false));
}
_tPositiveDerivatives.Add(Engine.Term("[14.5, +inf)", _tsiDerivatives,
false));

_tScoreDurations = new ArrayList();
Term t = Engine.Term("[2, 3)", _tsiScoreDurations, false);
for(int i = 0; i < 3; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[3, 4)", _tsiScoreDurations, false);
for(int i = 3; i < 4; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[4, 5)", _tsiScoreDurations, false);
for(int i = 4; i < 5; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[5, 7)", _tsiScoreDurations, false);
for(int i = 5; i < 7; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[7, 10)", _tsiScoreDurations, false);
for(int i = 7; i < 10; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[10, 14)", _tsiScoreDurations, false);
for(int i = 10; i < 14; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[14, 20)", _tsiScoreDurations, false);
for(int i = 14; i < 20; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[20, 28)", _tsiScoreDurations, false);
for(int i = 20; i < 28; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[28, 40)", _tsiScoreDurations, false);
for(int i = 28; i < 40; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[40, 56)", _tsiScoreDurations, false);
for(int i = 40; i < 56; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[56, 80)", _tsiScoreDurations, false);
for(int i = 56; i < 80; i++)
    _tScoreDurations.Add(t);
t = Engine.Term("[80, +inf)", _tsiScoreDurations, false);
_tScoreDurations.Add(t);
}

public Artifact RSHPtize(InterpolatedSequence iSeq, CAKEEngine Engine,
    string fullName, bool isQuery){
    if(!_domainLoaded) {
        LoadDomain(Engine);
    }
    Artifact rSeq = new Artifact(Engine, fullName, _atSequence,    fullName,
        null, _language, false, "Sequence", isQuery, 0);

    for(int staff = 0; staff < iSeq.Count; staff++){
        InterpolatedStaff iStaff = iSeq[staff];
        Artifact rStaff = new Artifact(Engine, rSeq, fullName+": "+staff,
            _atStaff, null, null, _language, 0, staff, 0, 0, null, false,
            "Staff "+staff, isQuery, 0);
    }
}

```

```

for(int voice = 0; voice < iStaff.Count; voice++){
    InterpolatedVoice iVoice = iStaff[voice];
    Artifact rVoice = new Artifact(Engine, rStaff, fullName+": "+
        staff+": "+voice, _atVoice, null, null, _language, 0, voice, 0,
        0, null, false, "Voice "+voice, isQuery, 0);

    for(int span = 0; span < iVoice.Count; span++){
        InterpolatedSpan iSpan = iVoice[span];
        Artifact rSpan = new Artifact(Engine, rVoice,
            fullName+": "+staff+": "+voice+": "+span,
            _atSpan, null, null, _language, 0, span, 0, 0, null, false,
            "Span "+span, isQuery, 0);

        for(int poly = 0; poly < iSpan.Count; poly++){
            Polynomial iPoly = iSpan[poly];
            RSHP rshp = new RSHP(rSpan, GetSemanticItem(iPoly), false,
                false, 0, false, false, false, false, 0, 0);
            KE action = new KE(rshp, GetScoreDuration(
                iSpan.ScoreDuration), 0, 0, 0, 0, 0);
            Artifact dy0 = new Artifact(Engine, rshp, fullName+": "+
                staff+": "+voice+": "+span+"dy0", _atDerivatives, null,
                null, _language, 1, 0, 0, 0, GetDerivative(
                    iPoly.Evaluate(0)), false, "dy0", isQuery, 0);
            Artifact dy1 = new Artifact(Engine, rshp, fullName+": "+
                staff+": "+voice+": "+span+"dy1", _atDerivatives, null,
                null, _language, 2, 0, 0, 0, GetDerivative(
                    iPoly.Evaluate(1)), false, "dy0", isQuery, 0);
            // Add Concept Orders 3
            ArrayList co3 = iStaff.GetDerivativesAt(iSpan.ScoreOnset);
            for(int i = 0; i < co3.Count; i++){
                ArrayList co3_ = co3[i] as ArrayList;
                if(i != voice && co3_.Count != 0) {
                    Artifact dy0_ = new Artifact(Engine, rshp, fullName+
                        ": "+staff+": "+voice+": "+span+"dy0_", _atDerivatives,
                        null, null, _language, 3, 0, 0, 0, null, false,
                        "dy0_", isQuery, 0);
                    foreach(double dy0_d in co3_) {
                        KE dy0_k = new KE(dy0_, GetDerivative(dy0_d), 0,
                            0, 0, dy0_d, 0);
                    }
                }
            }
            // Add Concept Orders 4
            ArrayList co4 = iStaff.GetDerivativesAt(iSpan.ScoreOnset+
                iSpan.ScoreDuration);
            for(int i = 0; i < co4.Count; i++){
                ArrayList co4_ = co4[i] as ArrayList;
                if(i != voice && co4_.Count != 0) {
                    Artifact dy1_ = new Artifact(Engine, rshp, fullName+
                        ": "+staff+": "+voice+": "+span+"dy1_", _atDerivatives,
                        null, null, _language, 4, 0, 0, 0, null, false,
                        "dy1_", isQuery, 0);
                    foreach(double dy1_d in co4_) {
                        KE dy1_k = new KE(dy1_, GetDerivative(dy1_d), 0,
                            0, 0, dy1_d, 0);
                    }
                }
            }
        }
    }
}
return rSeq;
}
protected Term GetDerivative(double dy) {
    if(dy > 0){
        double dy2 = Math.Round(Math.Round(dy, 3));
        return _tPositiveDerivatives[(int)Math.Min(dy2,
            _tPositiveDerivatives.Count-1)] as Term;
    }else{
        double dy2 = Math.Round(Math.Round(dy, 3));
        dy2 = Math.Sign(dy2)*dy2;
        return _tNegativeDerivatives[(int)Math.Min(dy2,
            _tNegativeDerivatives.Count-1)] as Term;
    }
}

```



```
    }  
  }  
  protected Term GetScoreDuration(int duration) {  
    if(duration >= 80) {  
      return _tScoreDurations[_tScoreDurations.Count-1] as Term;  
    }else{  
      return _tScoreDurations[duration] as Term;  
    }  
  }  
  protected SemanticItem GetSemanticItem(Polynomial poly){  
    switch(poly.Shape){  
      case PolynomialShapes.Concave: return _siConcave;  
      case PolynomialShapes.Convex: return _siConvex;  
      default: return _siFlat;  
    }  
  }  
}  
}
```

Part XIII:

Resumen en Español

1 Notación Musical, MIDI y SMF

El presente proyecto fin de carrera se basa en el estándar MIDI y en SMF para representación de archivos musicales. En esta Sección se presenta una muy breve introducción a los conceptos básicos de la notación musical así como de MIDI y SMF 1.0.

1.1 Notación Musical

En el sistema actual de representación de música se usan pentagramas como los de la Figura 1.1, en el que aparece el llamado gran pentagrama, con claves de Sol y de Fa.

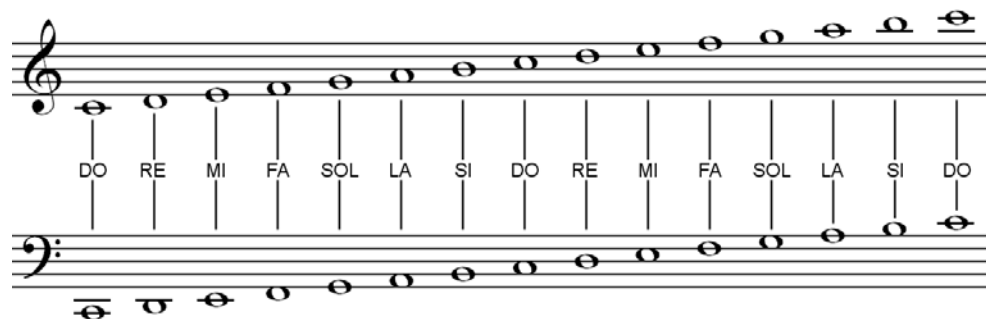


Figure 1.1 El gran pentagrama

Las notas se representan con símbolos como los de la figura, de tal forma que cada uno de ellos representa una duración distinta de la nota y la altura en la que esté representará el tono (cuanto más alto más agudo).






Nombre	Figura	Duración
Redonda		Taken as unit
Blanca		La mitad de una redonda
Negra		La mitad de una blanca
Corchea		La mitad de una negra
Semicorchea		La mitad de una corchea
Fusa		La mitad de una semicorchea

Table 1.1 Notas musicales

Estas notas, según la tonalidad elegida para una determinada canción, tienen de forma natural unos intervalos entre notas sucesivas. Estos intervalos definen la escala del tono y el modo del mismo. Por ejemplo, la Figura 1.2 muestra una escala mayor en la tonalidad de do.

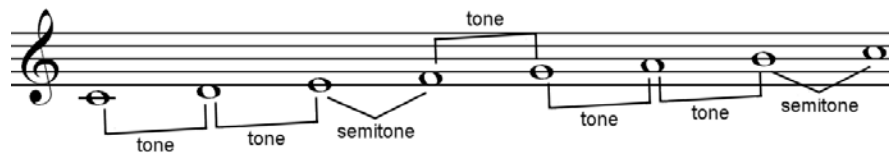


Figure 1.2 Escala mayor de Do

1.2 El Estándar MIDI

MIDI es un estándar de representación de información musical dirigida por eventos. Básicamente, un flujo de datos MIDI consiste en una secuencia ordenada en el tiempo de eventos.

Estos eventos pueden ser de muy variados tipos, pero principalmente interesan los eventos NoteOn y NoteOff, que son los que indican cuando una nota debe sonar o detenerse.

Hay otros tipos de información en un flujo MIDI, pero carecen de interés para el objetivo de este proyecto.

1.3 Standard MIDI Files

SMF es una especificación acerca de cómo almacenar contenido musical en archivos. Básicamente contiene también eventos al igual que un flujo MIDI. Sin embargo, un archivo SMF puede contener mucha más información adicional como la tonalidad, tiempo, letra, copyright, etc. Además, en un archivo SMF los eventos se separan por tracks, que esencialmente se pueden corresponder con distintos instrumentos que aparecen en la composición.

Para la realización de este proyecto fin de carrera se ha elegido MIDI como medio de representación musical porque está muy extendido actualmente (es un estándar de hace dos décadas), es sencillo, ampliable y contiene toda la información necesaria.

2 El Modelo RSHP y el CAKE Engine

RSHP es un metamodelo de representación de información que permite modelar cualquier dominio de información de una única forma. Está basado en relaciones entre elementos de información, que a su vez se pueden agrupar en artefactos. La Figura 2.1 muestra un diagrama UML del modelo.

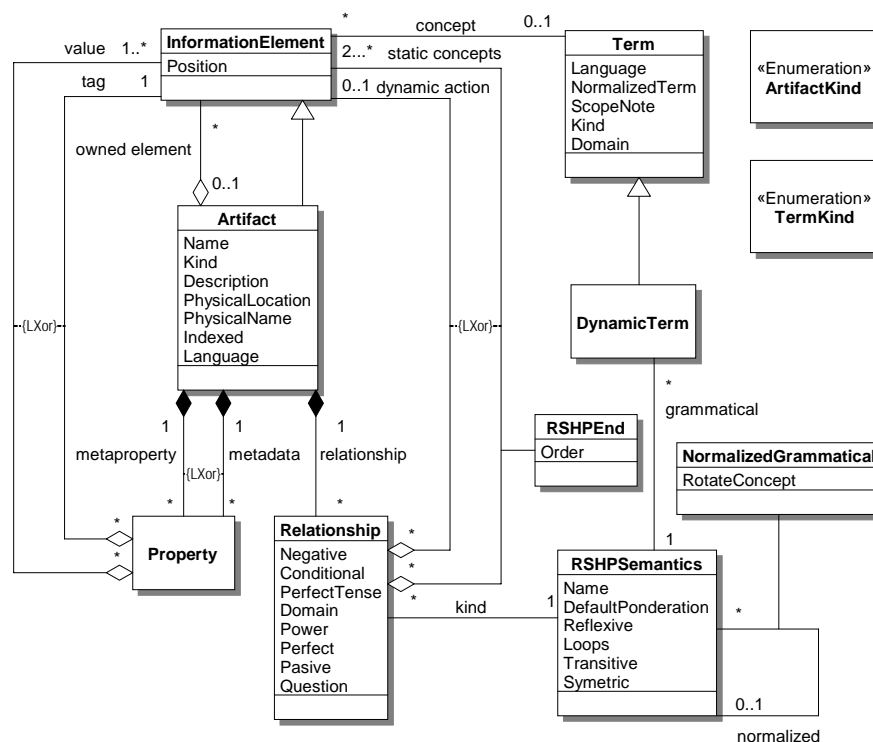


Figure 2.1 El metamodelo RSHP

Aunque RSHP se originó para recuperación de información textual, lo cierto es que su diseño permite el modelado de cualquier dominio de información (salvo algunos pequeños aspectos), y en este caso se va a aplicar a la representación de información musical.

La unidad de comparación son los artefactos, que pueden ser comparados entre sí de acuerdo a su tipo. Un artefacto a su vez puede tener más subartefactos y relaciones RSHP entre, de nuevo, artefactos o términos.

Cada RSHP tiene también un tipo y se le asocia un elemento de información llamado acción y que recoge la semántica dinámica de la relación, mientras que el tipo recoge la semántica estática. A su vez, cada RSHP conecta elementos de información entre sí. Así, por ejemplo, para decir que un ordenador es un tipo que hereda desde el tipo máquina, podemos establecer una relación jerarquía de la siguiente forma:

$$RSHP_1 = \{\text{ser, ordenador, máquina}\}^{\text{Jerarquía}} \quad [2.1]$$

2.2 El CAKE Engine

Por otro lado, el metamodelo RSHP sólo sirve para representar información. Cuando se quieran comparar artefactos para realizar búsquedas, debe incluirse un framework que realice tal tarea. Aquí es donde entra el CAKE Engine, encargado de ofrecer procesos de indexación y recuperación sobre artefactos modelados con RSHP.

El CAKE Engine ofrece consultas por similitud o por inclusión. Realiza dos tipos de comparación en cualquier caso:

- Toma en cuenta la topología de los artefactos, de forma que considera aspectos como número de subartefectos, su estructura, tipo, etc.
- Distancias semánticas entre elementos de información, de forma que permite establecer relaciones y distancias entre términos de una red semántica.

Los datos obtenidos se ponen en forma vectorial sobre un espacio n-dimensional en el que cada artefacto base está representado por un vector. Después, una distancia euclídea sirve para determinar la distancia semántica entre una query y un artefacto en el repositorio.

El CAKE Engine se encuentra integrado en una herramienta llamada Software Reuse o CAKE Studio. Así, el proyecto deberá estar integrado a su vez con esta herramienta, estableciendo así el entorno operativo del mismo.

3 Requisitos Generales

En esta Sección se presentan una serie de requisitos generales con los que el sistema debe cumplir. Se dividen en requisitos verticales y horizontales, atendiendo a la dimensión musical que tratan.

3.1 Restricciones Verticales

Primeramente, el sistema no debe ser sensible a cambios de octava. Es decir, que las dos piezas en la Figura 3.1 deben ser consideradas como iguales al tratarse aisladamente.



Figure 3.1 Equivalencia de octava

Por otro lado, tampoco debe ser sensible a cambios de tonalidad. Esto significa que si se comparan dos piezas musicales iguales pero en distinta tonalidad, el sistema deberá retornar una distancia semántica cero.

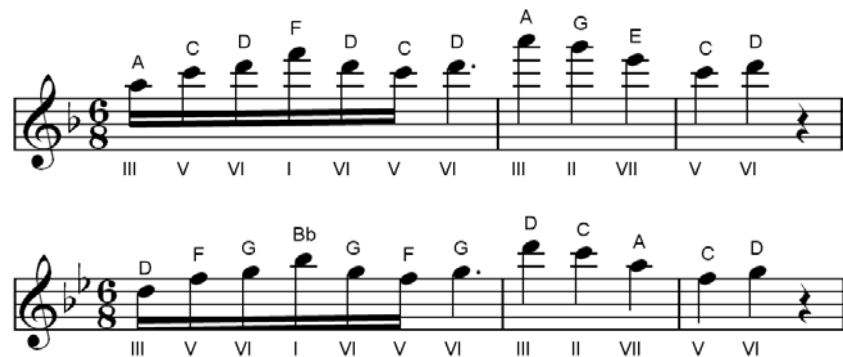


Figure 3.2 Igualdad de grados

Sin embargo, también debe reconocer como iguales las secuencias en las que aparezcan las mismas notas, aunque se trate de grados distintos en distintas tonalidades.



Figure 3.3 Igualdad de notas

Otra característica deseable del sistema es la capacidad de reconocer acordes y armonía en general. Particularmente, debería ser posible la

3.2 Restricciones Horizontales

3.3 Separación de Voces

page 212

Así, piezas como la de la Figura 3.7 deberían separar las voces azul y verde:



Figure 3.7 Separación de voces

4 El Modelo Matemático

La solución propuesta para el proceso de recuperación de información musical pasa por el uso del análisis numérico para interpolar la secuencia de notas en un espacio multidimensional de forma que los elementos a comparar sean las derivadas de las curvas generadas.

4.1 Normalización de Dominios

Primeramente, debe realizarse un proceso de normalización de los dominios temporal y de tono. Aprovechando el intervalo de tonos posibles de MIDI [0,127], y estableciendo una unidad mínima de tiempo para las notas, una pieza como la de la Figura 4.1



Figure 4.1 Normalización de dominios (parte I)

se convertiría en una secuencia normalizada de puntos como la siguiente:

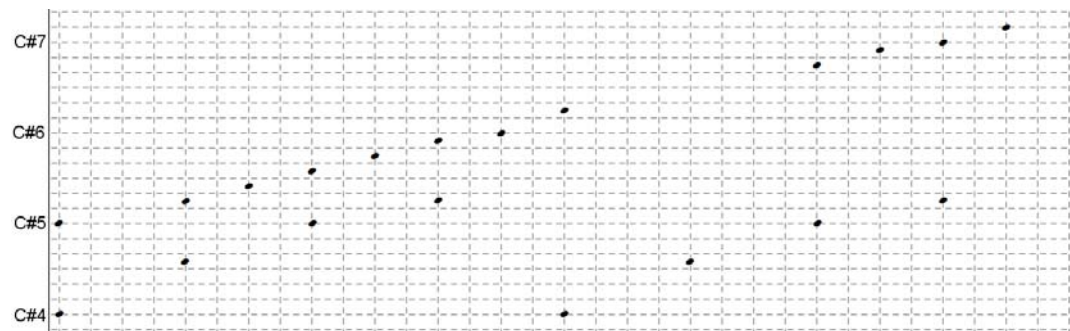


Figure 4.2 Normalización de dominios (parte II)

Una vez hecho esto, se podría empezar la interpolación de los puntos para generar las curvas. Entre los distintos tipos de interpolación existentes, se ha elegido interpolación por B-Splines paramétricos debido a una gran cantidad de propiedades que los hacen destacar sobre el resto de opciones:

- No sufre el fenómeno de Runge, por lo que no habrá oscilaciones de la curva entre dos puntos.
- La curva se define por partes, justamente para poder comparar el intervalo entre dos notas de forma más precisa.

- Cada parte de la curva es un polinomio de grado 3, por lo que es derivable y además tiene continuidad geométrica.
- Los B-Spline aseguran que la curva se mantendrá en un dominio deseado de valores, por lo que el intervalo $[0,127]$ de MIDI se mantendrá siempre.
- El cambio de una de las notas sólo afecta a la curva en un intervalo determinista, por lo que los cambios locales no repercutirán globalmente y hacen además posible la comparación de acordes.
- Son muy fáciles de calcular ya que solo necesitan cuatro multiplicaciones de un polinomio por una constante.

Así, la pieza musical de la Figura 4.1 quedaría, lista para comparación, como sigue (con una voz simple por cada dimensión):

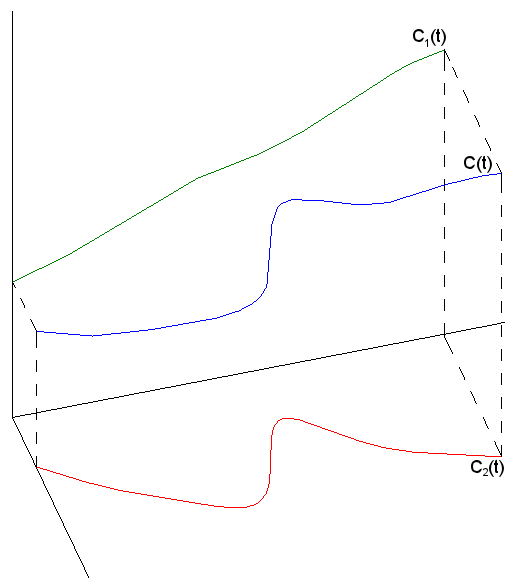


Figure 4.3 Interpolación de canciones

This page is intentionally left blank.

Julián Urbano Merino, autor del presente Proyecto Fin de Carrera titulado *Modeling and Indexing Musical Files to allow Music Reuse*, autoriza a que la información y documentación contenida en esta memoria pueda ser utilizada para la realización de otros Proyectos Fin de Carrera así como cualquier otra actividad docente.

Fdo.: Julián Urbano Merino