Department of Informatics
School of Science and Technology

FINAL YEAR PROJECT
FOR THE AWARD OF
MUSIC INFORMATICS BA
(2009)

Computer Representation and Generation of Karṇāṭak Rhythms

Author:
Arthur John Rupert CARABOTT-TORTELL (53806)

Supervisor:
Dr. Nick COLLINS

Abstract

Computer representations of music are generally concerned with Western styles of music and perhaps consequentially or at least for similar reasons, most research into algorithmic composition is also concerned with these styles. This dissertation gives a background on both Karṇāṭak (South-Indian) music and algorithmic composition, followed by an outline of requirements of a representation and algorithmic composition system for Karṇāṭak rhythmic activity. A system built for these purposes is described and evaluated by musicological and blind-listener comparison with material from professional Karṇāṭak percussionists. The dissertation concludes with possible future extensions and alternate methodologies for the system.

Declaration

This report is submitted as part requirement for the degree of Music Informatics at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Acknowledgements

Roseanna Pollen & Ferdy Carabott,
for their love and support

Suki Ferguson,
for her love and patience

Nick Collins,
for providing inspiration and encouragement

Alex Churchill,
for leading the way

Robin Watson,
for answering so many questions

George Bashi,
for the practical advice

R. N. Prakash,
for his wisdom

Ludwig Pesch and David Nelson,
for lending their expert ears

My evaluators, *Freddy Rayfield, Rafi Rogans-Watson, David Tait, Oliver Levy, Matthew Adamo, Jay Matharu, Arif Driessen, Jamie Bullock, Johnny Wildey, Bopsi Chandramouli, Ben Oliver, Camilo Tirado, Aykut Kekilli*

Contents

Chapter 1

Introduction

In the field of algorithmic composition systems concerned with non-western styles of music are in the minority. While there has been some algorithmic composition of North Indian (Hindustani) percussion music (Bel and Kippen, 1992), only the melodic aspect of South Indian (Karṇāṭak) music has been given any attention (Bel, 1998). To the author's knowledge, there has been no algorithmic composition work focused on Karṇāṭak rhythm.

Despite the infancy of ethnomusicological study of Karṇāṭak music (even in comparison with Hindustani music), the amount of analytical work available, supplemented by instructional material from the musicians themselves makes possible algorithmic composition of the style's fundamental features.

This project is concerned with modelling the rhythm generation processes of the tradition. To do this requires a useful and analogous method of representing Karṇāṭak rhythms, as well as models of the style's many processes for music creation. Once developed, the system would provide an environment for working with Karṇāṭak rhythms, as well as partially and fully automated methods of generating Karṇāṭak rhythms.

While the most abundant source of rhythmic material comes from Karṇāṭak percussionists, a level of abstraction has been adopted so as to separate the rhythms of the tradition from the playing of a particular instrument. This abstraction makes the resulting materials applicable to any Karṇāṭak (or even non-Karṇāṭak) percussion instrument and possibly to melodic instruments.

1.1 Structure of Dissertation

This report gives a description of a system for representing and algorithmically composing Karṇāṭak rhythms with a computer. A background of both algorithmic composition and Karṇāṭak music is given, as well as a review of relevant systems and literature. Specifications and requirements of the research, potential users and the system are outlined and used as the basis for design and implementation. The design of the various elements of the system and their interactions are outlined, with the output evaluated by musicological and listener based comparison with 'real' Karṇāṭak music. Finally, future improvements to the system and alternative methodologies are considered.

Chapter 2

Research Topics

This project is a fusion of algorithmic composition and musicological analysis of Karṇāṭak rhythm. Overviews of both subjects are given with discussion of the most important elements.

2.1 Karṇāṭak Music

Indian classical music is roughly divided into two schools; the Hindustani music of the North and the Karṇāṭak music of the South. While the origins of Karṇāṭak music dates back to (approximately) 500 B.C (Ayyangar, 1972, p.23), the form is a "living, developing phenomenon" (Nelson, 1991, vol.1 p.viii), in a state of continuous evolution. A notable result of this evolution is the recent (early twentieth century) change of performance venue from the religious temple and palace service to the more secular modern concert stage (Viswanathan and Allen, 2004, ch.1).

As with any style of music, in-depth discussion is impossible when talking in general; it becomes necessary to talk about forms within the style. However, it is possible to note a few general concepts that are fundamental to Karṇāṭak music. A feature of Karṇāṭak music often noticeable to the western listener is a lack of functional harmony. Harmony, the simultaneous combination of pitches to create chords, and in turn chord progressions (Carl et al., 2008), is one of the three fundamentals of western tonal music– along with rhythm and melody. Karṇāṭak (as well as Hindustani) music has no concept of a key, or modulation between keys, instead the other fundamentals: melody and rhythm, or Tāḷa and Rāga and their subtleties that "reign supreme" (Viswanathan and Allen, 2004, p.34).

Not without its own trinity of fundamental elements, a third key concept to Karṇāṭak music is one shared with some western styles (such as Jazz); improvisation.

2.1.1 Tāḷa

Tāḷa is somewhat equivalent to time signature in western music in that it defines the number of beats in a clearly definable microstructure of the music. This is only a loose equivalence as the notion of time in western music is linear, and the time signature and tempo may change within a piece. In Karṇāṭak music there is no changing of tempo (Viswanathan and Allen, 2004, p.35), nor changes between Tāḷas or in the structure of Tāḷa; Tāḷa is considered an unchanging, "regularly re-occurring *cycle* of beats" (Nelson, 1991, vol.1 p.6). Tāḷa differs further as (unlike the western time signature) there are no implied accents within a given cycle, instead they are 'generated by musical phrases and the processes applied to them' (Nelson, 2008, p.2). Two examples of tāḷas are given in Figure 2.5. Tāḷas will discussed in more detail as part of the implementation of a Tāḷa Generator in Section 5.4.

2.1.2 Rāga

Rāga can be considered a more sophisticated equivalent to Western scales. While both can be described as a collection of pitch based musical events (notes in the West, svaras in Karṇāṭak music) a Rāga defines many more musical aspects than pitch alone. In a Rāga the pitch of the svaras used may change depending on the direction (ascending or descending), movement between svaras may be stepwise or crooked -starting in one direction, then temporarily reverse before continuing in the original direction

Figure 2.1: Mṛdaṅgamist R. N. Prakash.

Figure 2.2: Karaikudi Mani playing the mṛdaṅgam.

Figure 2.3: Selvaganesh Vinayakram playing the kanjira

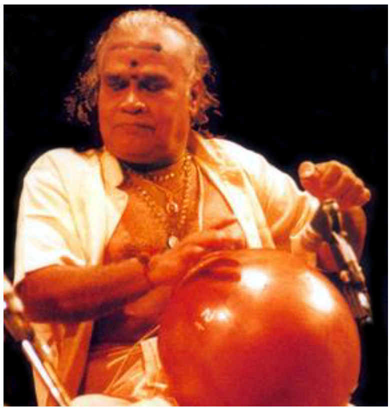


Figure 2.4: T. H. Vikku Vinayakram playing the ghatam

tensvaras may also be of a fixed pitch or produced with a variety of ornamentations (Gamakas) similar to vibrato, glissandos and acciaccatura (Viswanathan and Allen, 2004, p.47).

2.1.3 The Five Families of Rhythm

In Karṇāṭak music there are considered to be five jātis [1] (or families) of rhythm. The numbers that constitute these jātis are four, three, seven, five and nine in order or recognition as a legitimate basis on which to build tāḷa cycles (Viswanathan and Allen, 2004, p.35). These five families are apparent at every level of Karṇāṭak rhythmic thinking; the possible gatis (sub-divisions of the beat), the number of syllables/strokes in the building block phrases of improvisation and composition and the laghu– a variable length aṅgam (section) of the tāḷa. The five families and their names are outlined in Table 2.1.

[1]This term is different from jatis which refers to syllables or groups of syllables. Jāti is also a term for hereditary social grouping, caste (Viswanathan and Allen, 2004, p.35).

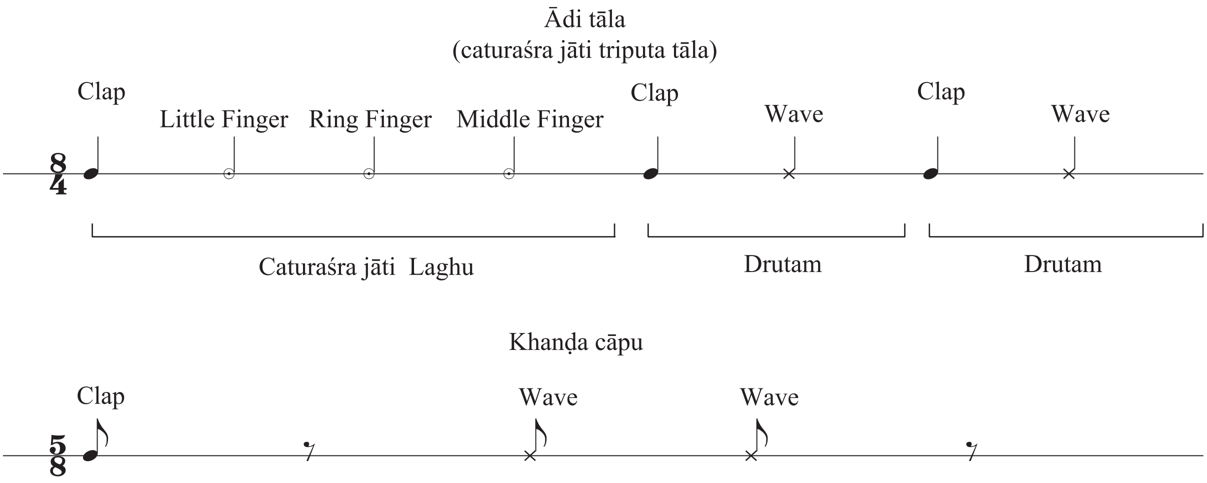


Figure 2.5: Two tāḷas in score notation. The first is Ādi Tāḷa (the short name for caturaśra jāti triputa tāḷa), the most common tāḷa in Karṇāṭak music. The second is khanḍa cāpu. The kriyās (hand gestures) used to keep track of the tāḷa are given, as are the names of the aṅgas (secions, "limbs") of ādi tāḷa. Source: (Pesch and Sundaresan, 1996)

Five Families of Rhythm		
Number	Name	Meaning
4	Caturaśra	"Four-sided"
3	Tiśra	"Three-sided"
7	Miśra	"Mixed"
5	Khanḍa	"Broken"
9	Sankīrṇa	"All mixed up"

Table 2.1: The five families of rhythm and their names. From Viswanathan and Allen (2004, p.35-36)

These five basic units can be expanded by doubling (and in the case of four by halving) to achieve further jāti numbers (Nelson, 1991, vol.1 p.18). It should be noted that numbers are not shared between families; while 12 is a multiple of both three and four, limiting expansion to doubling renders 12 achievable only by (twice) doubling three (Nelson, 1991, vol.1 p.19). However, phrases with pulse totals that do not belong to these families do exist, but are considered to be concatenations; e.g. 15 would be a compound phrase of 5 + 5 + 5 (Da di gi na dom, Da di gi na dom, Da di gi na dom), each belonging to the family of five while the whole phrase does not (Nelson, 1991, vol.1 p.18). To provide clarity it should be noted that a phrase can belong to a rhythmic family while having a *duration* that does not; see Figure 2.6 for an example.

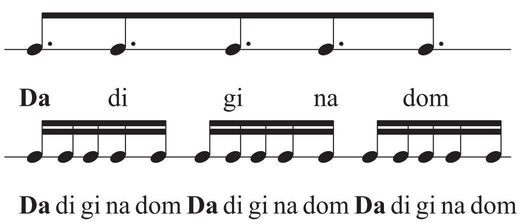


Figure 2.6: Both phrases have an equal duration, yet only the top line is considered part of the five family, the bottom is a compound phrase, of which only the inner phrases belong to the five family.

2.1.4 Improvisation

While there *are* composed pieces of music in the Karṇāṭak tradition, these are transmitted orally from guru to student, who adds their own individual touches. The consequence of oral as opposed to written transmission (and the required reliance on memory) is a concept of composition with a higher level of abstraction than in the west. While the lyrics, tāḷa, rāga and melody may be specified for a composition, the structure and nuances may be altered by the artist; lines may be repeated as many times as desired, with a variety of ornamentations (Viswanathan and Allen, 2004, p.65). This concept of composition is what gives Karṇāṭak compositions their great long-term flexibility and longevity. In the short term (within a performance) the composed core is also considered flexible, as noted by Viswanathan and Allen (2004, p.60) it is this "dialogue between what is fixed and what is created in the moment [that] is at the heart of listeners' enjoyment of Karṇāṭak music."

2.1.5 Instruments

The list of instruments used by Karṇāṭak musicians is ever-growing and changing.

Traditional melodic instruments include the Voice, Veena (a fretted stringed instrument), Violin and the Flute (made from bamboo). The adoption of of foreign instruments has become increasingly common, the European violin approximately two centuries ago (Viswanathan and Allen, 2004, p.29), and the recent additions of Electric Mandolin (Shrinivas, 2007), Electric Guitar (Prasanna, 2003) and Saxophone (Palnath, 2009).

In its current state, Karṇāṭak music's primary percussion instruments are the Mṛdaṅgam– a double headed drum, the main percussion instrument in most Karṇāṭak concerts (Figure 2.7), Kanjira– a small frame drum with a pair of tiny brass jingles (Figure 2.8), Ghatam– a clay pot (Figure 2.9), Morsing– a jaw harp idiophone (Figure 2.10) and the vocal technique Solkaṭṭu/Koṉakkōl (Lockett (2008), p.9-11; Nelson (2008), p.1).

2.1.6 Solkaṭṭu/Koṉakkōl

Solkaṭṭu is a set of percussive-sounding syllables that are learnt in tandem with the strokes and patterns when studying a percussive instrument (Nelson, 2008, p.2). The syllables were chosen for their ability to be smoothly recited successively at fast speeds, a feat that is impossible with English or Tamil numerals (Viswanathan and Allen, 2004, p.36).

Solkaṭṭu was developed into a musical speciality in its own right by Maṉṉārguḍi Pakkiri Piḷḷai (1857-1937) and became known Koṉakkōl (Pesch, 1999, p.47). Koṉakkōl was often featured in performance, particularly tāḷa vādya kaccēri (performances entirely dedicated to rhythm) (Pesch, 1999, Glossary), but has become less common in recent times (Pesch, 1999, p.47). Solkau is used in teaching as the distinctive syllables make the rhythmic groupings very clear. For example a group of four would be spoken 'Ta ka di mi' whereas a group of three would be 'Ta ki tȧ' (Vinayakram and McLaughlin, 2007, ch.2).

There is great variations in the syllables used; usually dependent on the bāṇī (musical style) of the teacher or family, the musical context, or the instrument being played (Pesch, 1999, Glossary). In the lessons of the author's teacher (Mṛdaṅgamist R. N. Prakash) and in Vinayakram's (2007, Disc.2 ch.5) *'Phrases in Ādi Taalam'* the first set of syllables for a group of four are 'Ki ta tha ka', while a more generic set commonly used for non-instrument specific teaching are Ta ka di mi (Pesch and Sundaresan, 1996; Nelson, 2008; Vinayakram and McLaughlin, 2007). Ayyangar (1972, p.309) lists fifteen examples of bi-syllabic solfa (two beat syllables) and ten examples of tri-syllabic solfa, while Brown (1965) lists twenty-six possible stroke combinations for quad-syllabic solfa, demonstrating the extent of variation possible.

An important reason for syllable variation is difference between 'closed' (short, sharp, non-resonant) and 'open' (long, resonant) stokes on a drum. In Karṇāṭak percussion playing phrases may be *rhythmically* identical, while using different sounding strokes. The same is true for Solkaṭṭu/Koṉakkōl where the word 'Ta ka' might be altered to use the arguably more sonorous 'din' to become 'Ta din' (Nelson, 2008, p.22).

A further example of variation in syllable use is shown by the possibility of phrasing a group of five as 'Ta ka, ta ki ṭa' a concatenation of two + three (which could be re-arranged as three + two; 'Ta ki ṭa, ta ka') or using the syllables 'Da di gi na dom' (Vinayakram and McLaughlin, 2007, ch.4).

Figure 2.7: A mṛdaṅgam

Figure 2.8: A collection of kanjiras

Figure 2.9: A ghatam

Figure 2.10: A morsing

Hulzen (2002, p.12) also notes that at second speeds (doubled tempo) groupings often have their own syllables, for example, when the density of 'Ta ka di mi' (first speed) is doubled (second speed) it becomes 'Ta ka di mi, ta ka ju na' and *not* 'Ta ka di mi, ta ka di mi'. The variation here exists as at faster tempos it becomes necessary to use the most efficient syllables, and to alternate between equivalents to prevent fatigue.

The extent of variation should now be clear. For the purposes of this dissertation a standard set of syllables have been adopted and used exclusively (see Table 2.2). The syllables I chose to use are a hybrid of the materials that formed my introduction to Karṇāṭak music and Ko<u>n</u>akkōl; (Vinayakram and McLaughlin, 2007; Pesch and Sundaresan, 1996) and the analytical works that influenced the algorithms; (Nelson, 2008, 1991). The strict use of this vocabulary makes generated material applicable to any instrument, the focus being on the generation of Karṇāṭak rhythms, and not on particular drum strokes.

2.2 Algorithmic Composition

In the paper *AI Methods for Algorithmic Composition: A Survey, a Critical View and Future Prospects* Papadopoulos and Wiggins concatenate David Cope's definitions of the two words ('algorithmic' and 'composition') given during a panel discussion at the 1993 International Computer Music Conference to form the following definition of algorithmic composition: "A sequence (set) of rules (instructions, operations) for solving (accomplishing) a [particular] problem (task) [in a finite number of steps] of combining musical parts (things, elements) into a whole (composition)" (Cope, 1993*b*; Papadopoulos and Wiggins, 1999).

No. Jatis	Jatis
1	Ta
2	Ta ka
3	Ta ki ṭa
4	Ta ka di mi
5	Da di gi na dom
6	Ta ki ṭa ta ki ṭa
7	Ta ka di mi ta ki ṭa
8	Ta ka di mi ta ka ju na
9	Da di gi na dom ta ka di mi

Table 2.2: The set solkaṭṭu words used throughout this project. Note that SCLang doesn't handle accents on letters, so the phonetic 'tah' is used instead of ṭa in system printouts.

2.2.1 Algorithmic Composition Methods

While there are historical examples of non-computational algorithmic compositions such as the Musical Dice Game often (but not entirely accurately) attributed to Mozart, the strengths of computers make them particularly fit for the purpose. While not exhaustive, Papadopoulos and Wiggins provide the following general categories of algorithmic composition methods that have been implemented within the last decade:

- Mathematical Models
- Knowledge Based Systems
- Grammars
- Evolutionary Methods
- Systems Which Learn
- Hybrid Systems

Mathematical Models

Mathematical models of algorithmic composition are outlined as those that use mathematically founded methods such as stochastic processes and Markov chains, as well as chaotic non-linear and iterated functions. There are many examples of systems uses these methods, Ames and Domino's Cybernetic Composer (1992) is noted by Papadopoulos and Wiggins (1999) as being a representative example of these systems.

Knowledge Based Systems

Knowledge based systems (also known as expert systems) are systems which that attempt to symbolically model the knowledge of a human expert using rules or constraints (Papadopoulos and Wiggins, 1999; Raynor, 2000). Ebcioglu's CHORAL (1988), a J.S. Bach style harmonisation system tenis exemplary of this type of system.

Grammars

Grammar systems use grammars derived from music in the same way that linguists derive grammars from language. David Cope's 'Experiments in Musical Intelligence' (Cope, 1991) uses elements of grammar based composition; extracting "signatures" from multiple pieces of a composer's work to use for composition. Bernard Bel's Bol Processor is a completely grammar based system, discussed in greater depth in Section 3.1.1.

Evolutionary Methods

Evolutionary methods are those that use the strengths of Genetic Algorithms (GAs) (dealing with very large search spaces, and providing multiple solutions) for the creation of musical material. Although arguably a search tool, GAs are often regarded as a form of machine learning (Mitchell, 1997, ch.9). There have generally been two approaches employed in regards to the fitness function of these systems; a formally stated and computationally implemented function, and the use of a human function (often referred to as an interactive GA) (Papadopoulos and Wiggins, 1999). A. Biles GenJam is a well known 'genetic algorithm-based model of a novice jazz musician learning to improvise' (Biles, 1994).

Systems Which Learn

This type of system will use various computational methods such as sub-symbolic Artificial Neural Networks (ANN), or symbolic Machine Learning (ML) in an attempt to teach the system how to complete a given task rather than explicitly telling it how (Papadopoulos and Wiggins, 1999). Typically these systems have little or no a priori knowledge of the problem and are taught by example, a method that draws on the ideas of numerous disciplines such as artificial intelligence, probability and statistics, computational complexity, information theory, psychology and neurobiology, control theory and philosophy (Mitchell, 1997). Mozer, a melody generating system that uses an ANN is such a system (Todd, 1989).

Hybrid Systems

Hybrid systems are those that combine the approaches of previously outlined methods. As single-method systems tend to have individual strengths, the hope of a hybrid system is to combine the different assets of various methods into an altogether better system (Papadopoulos and Wiggins, 1999). Biles (1994) turned his GenJam system into a hybrid system when attempting to increase efficiency of evaluation by using an ANN as the fitness function instead of a human (Biles et al., 1996).

2.2.2 Motivations for Algorithmic Composition

There are numerous motivations for the automation of the compositional process, the disambiguation of them has been well achieved by Pearce, Meredith and Wiggins in their paper *Motivations and Methodologies for Automation of the Compositional Process* (Pearce et al., 2002) which outlines the following categories:

- Algorithmic Composition
- Design of Compositional Tools
- Computational Modelling of Musical Styles
- Computational Modelling of Music Cognition

Algorithmic Composition (AC)

Pearce et al. define the motivation behind algorithmic composition programs as being artistic, for the purpose of generating 'novel musical structures, compositional techniques, and even genres of music'. David Cope's Experiments in Musicical Intelligence is an example of an AC system, initially designed to be a composing partner for the purpose of easing composers block (Cope, 1991).

Design of Compositional Tools (DCT)

The motivation for the design of compositional tools is similar to that of algorithmic composition in that the intent is to produce music for practical purposes. While AC systems are usually only used by the authoring composer (almost as an extension of themselves), compositional tools are designed to be used by other composers as part of their own artistic process. Simple examples of compositional tools are found in the notation package Sibelius (Avid, 2008), where functions can be found to create simple harmonies to a given melody, produce retrogrades of material, or to realise figured bass.

Computational Modelling of Musical Styles (CMMS)

Despite producing music, the motivation behind CMMS differs from that of AC and DCT as it 'is not to compose aesthetically pleasing pieces of music nor to design a useful compositional tool but to propose and verify hypotheses about the stylistic attributes defining a body of works' (Pearce et al., 2002). An example of this type of system is HARMONET, developed by Hild, Feulner and Menzel, which focuses on the harmonisation of chorales in the style of J.S. Bach (Hild et al., 1992).

Computational Modelling of Music Cognition (CMMC)

The motivation behind CMMC systems is an increased understanding of 'underlying cognitive process involved in human composition' (Pearce et al., 2002). As with CMMC systems, the production of aesthetically pleasing music or of useful tools is of little interest. The system designed by Steedman (1984) focuses on a generative grammar for jazz chord sequences and is exemplary of this type of program.

2.2.3 System Introduction

Method

The method of algorithmic composition used in this project is entirely knowledge based; Karṇāṭak are analysed for processes which are then turned into explicit algorithms. During the development period experimentation with statistical modelling took place, a method which was later abandoned in favour of the knowledge based approach. The details of the algorithmic composition methods will be discussed in Chapter 5.

Motivation

The motivations for the system are numerous; it can be used as a standalone environment for composition with or without adherence to Karṇāṭak traditions, a use which would fall under the category 'Algorithmic Composition' (Pearce et al., 2002). If used in conjunction with other methods of composition, the system would be classified by Pearce et al. (2002) as a 'Compositional Tool'. As the automation methods of the system have been modelled on processes and structures found in Karṇāṭak music, exclusive use of these methods would result in a 'Computational Modelling of Musical Style' (Pearce et al., 2002).

Chapter 3

Review of Relevant Work

Algorithmic Composition is a field with many methodologies (Papadopoulos and Wiggins, 1999) and motivations (Pearce et al., 2002) with application to any form of music. While certain musical forms have been the subject of many attempts at automation e.g. chorales (Cope, 1991; Ebcioglu, 1988; Hild et al., 1992), generation of Karṇāṭak rhythms are (to the author's knowledge) as of now un-automated.

This lack of closely related systems has resulted in investigation into work slightly further afield, as well as looking to the musical 'formulae' discovered by Indian and non-Indian musicologists.

3.1 Computational Systems

There are few examples of computer music researchers working with Karṇāṭak music. Two notable practitioners are Arvindh Krishnaswamy, who has been prolific in his work on the tracking, measurement, perception, analysis, transcription and modelling of pitch and rāgas in Karṇāṭak music (Krishnaswamy, 2003*a*,*b*,*c*, 2004*b*,*e*,*a*,*c*,*d*), and M.Subramanian whose work on synthesising gamakams (melodic inflections) (Subramanian, 2002, 1999) has lead to his Rasika-Gaayaka educational software (Subramanian, 2008).

As is evident, the preoccupation of these researchers is pitch and melody, (with the exception of the Tāḷa counting lessons in Subramanian's software) and even then the work has not touched upon the topic of Karṇāṭak algorithmic composition.

While computational algorithmic composition of Western musical styles is a flourishing field (Cope, 2005; Roads, 1996; Miranda, 2001; Ames and Domino, 1992; Ebcioglu, 1988; Biles, 1994), there has been far less work focused on Indian music, let alone Karṇāṭak (South Indian) music. To the author's knowledge there exists only one example of Karṇāṭak algorithmic composition work; grammars constructed for Bernard Bel's Bol Processor 2 (Bel, 2006). Again, the focus has been on melody generation, with grammars for modelling melodic improvisation and a number of highly constrained grammars for specific compositions. However, the Bol Processor 2 has been used extensively for the modelling of Hindustani (North Indian) Tabla improvisation of the *qa'ida* form (Kippen and Bel, 1992) (see Figure 3.1), making it a relevant system to this project.

3.1.1 Bol Processor

Note: This review of the Bol Processor is largely re-written from an essay by the author, submitted for the Generative Creativity course, as part of the Music Informatics (BA) undergraduate degree at the University of Sussex. Referenced as (Carabott, 2009).

The Bol Processor (BP) is a hybrid system (Papadopoulos and Wiggins, 1999) by Bernard Bel named after its original purpose; the transcription of *bols* (Tabla strokes) at performance speed (Bel, 1998) (a system commissioned by ethnomusicologist Jim Kippen).

Initially the system resembled a customised word-processor with the *bol* vocabulary mapped to a keyboard. Soon an *inference engine* based on Chomsky's linguistics was added, enabling Tabla music generation from user-defined grammars (Bel, 1998). The inference engine combined generative, context-free and generalising grammars as well as pattern rules (Bel, 1992*a*). A template matching parsing module was built for classifying variations on a theme with a 'membership test' (Bel, 1992*a*). The aim of the research was to 'create a human-computer interaction where musicians themselves respond to the output of BP grammars, and grammars are in turn modified to account for the input of musicians' (Kippen and Bel, 1992).

The second Bol Processor (BP2) was the result of interaction with Western musicians and featured numerous additions. The most interesting addition to the system was the symbolic machine-learning system QAVAID (Question Answer Validated Analytical Inference Device) (Kippen and Bel, 1989). QAVAID was given only knowledge of the grammar format not musical structure, and used an incremental learning strategy modelled on traditional teaching methods; "an implicit model is transmitted by means of sequences of positive instances of the 'language'. Negative instances composed by students are rejected or corrected." (Kippen and Bel, 1992). The use of a machine-learning algorithm removed the musicologist (and their individual assumptions) from the generation of grammars, making the process more analogous to the traditional teacher and student situation.

The *inference engine* was improved with the addition of remote contexts, repetition patterns, logic-numeric flags and meta-grammars (Kippen and Bel, 1992; Bel, 1992*b*). The language of Tabla *Bols* was replaced with 'sound-objects' that contained any number of MIDI messages (ranging from a simple NoteOn/NoteOff pair to a whole stream altering velocity, modulation, aftertouch etc). The move from *Bols* to MIDI based 'sound-objects' maintained the ability to work with Tabla, opened up the system to Western music and eased human computer interaction (via MIDI instruments).

Note: The following material is original work not found in (Carabott, 2009)

Between 1995 and 1997 work was conducted with BP2 and Karṇāṭak musicians (Bel, 1998). Some of this work was used as a general demonstration of BP2 at the 2006 Virtual Gamelan Graz Symposium (Bel, 2006) but has not been the focus of any academic publications. However, a number of Karṇāṭak grammars that were produced are included with BP2, available from the Bol Processor Homepage (Bel, 2009). These grammars include original compositions by Kumar S. Subramanian (see Figure 3.2) and Nadaka, a model of melodic improvisation and several well known compositions.

Unfortunately for this project, all of the Karṇāṭak grammars are for melodic materials. While rhythmic information is inherent in these melodies, they are not attempting to model the rhythms played by Karṇāṭak percussionists.

```
GRAM#6 [8]  LEFT  A'6 + <-> dhageteenakena+
GRAM#6 [9]  LEFT  A'8 + <-> dhatidhageteenakena+
GRAM#6 [10]  LEFT  A4 + <-> dheenagena+
GRAM#6 [11]  LEFT  C6 + <-> dhagedheenagena+
GRAM#6 [12]  LEFT  A8 + <-> dhatidhagedheenagena+
```

Figure 3.1: Part of a BP2 grammar for Hindustani Tabla playing. Note the use of bols (stroke names) as input

```
GRAM#2[1] <8-2> X --> sa6 re6
GRAM#2[2] <8-2> X --> re6 ga6
GRAM#2[3] <8-1> X Y --> ga6 pa6 dha6 X X
GRAM#2[4] <8-1> X X --> sa7 dha6 pa6
```

Figure 3.2: Part of a BP2 grammar for an original composition Karṇāṭak by Kumar S. Subramanian. Note the Karṇāṭak svara names used as input (sa, re, ga, pa, dha)

3.1.2 SwarShala

SwarShala is a proprietary software package for 'learning, practising and composing' Indian music (both Hindustani and Karṇāṭak) by Swar Systems (*Swar Systems*, 2009). While offering no methods algorith-

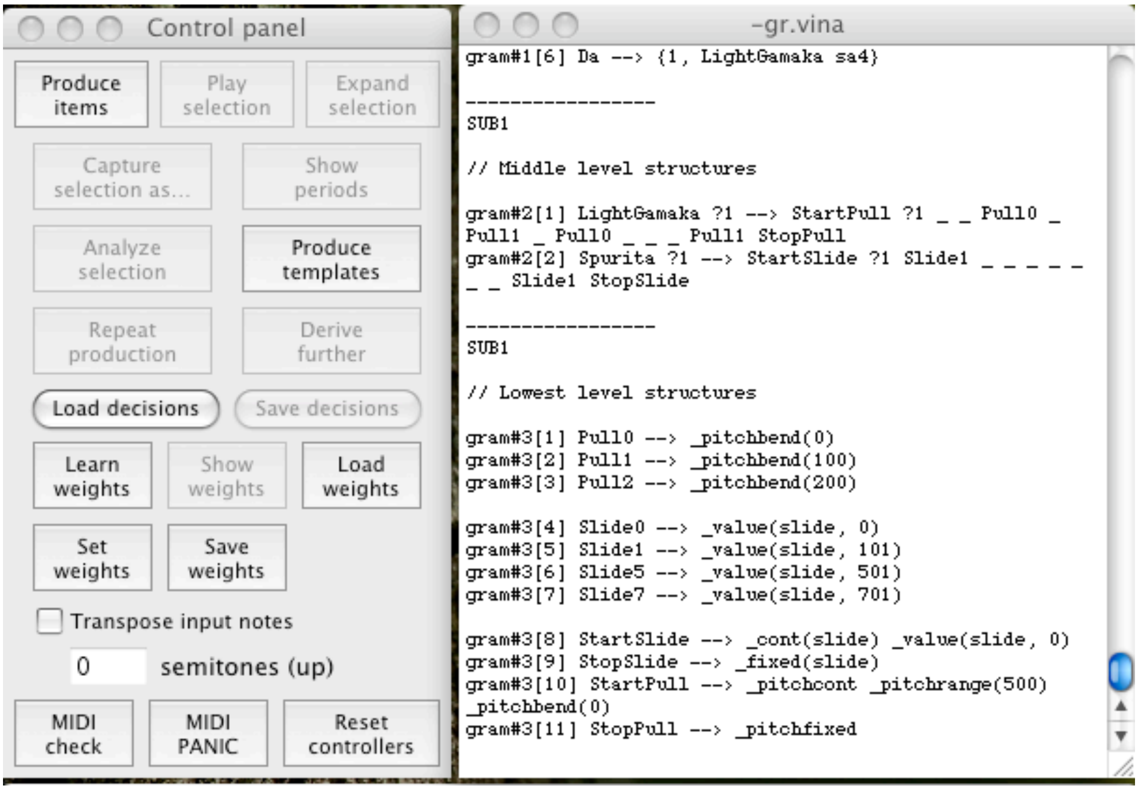


Figure 3.3: Bernard Bel's Bol Processor 2 with a grammar for a vina piece.

mic composition, the package does provide a means of representation; a MIDI sequencer with stoke or svara names appended to note objects, viewable either as a piano roll or in tāḷas (see Figure 3.4).

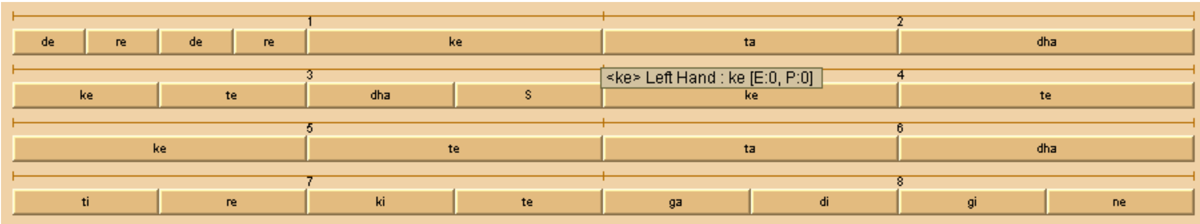


Figure 3.4: The tāḷa view of a sequence of Tabla Bols in SwarShala.

3.2 Literature

While there is a lack of computational algorithms for generating Karṇāṭak rhythms, there are published works of theory by Western ethnomusicologists and Indian music theorists alike. The most influential texts are outlined below. Another useful resource not discussed in depth is Hulzen's dissertation (Hulzen, 2002). The alternative perspective provided by Hulzen saved this project from over reliance on Nelson (1991, 2008) and Brown (1965), the former using the latter as the foundation of his work.

3.2.1 Robert Brown

Robert Brown's thesis (Brown, 1965) was a ground-breaking work that 'revealed to non-Indian musicologists for the first time, the structure of mṛdaṅgam lessons' (Nelson, 1991, vol.1 p.iv). It introduced Karṇāṭak concepts of rhythm, described the physical properties and cultural relevance of the mṛdaṅgam, provided sonograms of the various strokes, introduced solkaṭṭu and provided 152 mṛdaṅgam lessons.

Brown's work is an excellent resource for general knowledge of Karṇāṭak percussion as well as being a plethora of documented Karṇāṭak rhythmic materials. Brown provides the most in depth discussion of the problem of syllable and word grouping, one of the biggest analytical difficulties of the musical style. Brown's work also contains some useful rhythmic analysis work, notably suffix substitutions (Brown, 1965, p.149) for sarvalaghu (time-keeping) phrases and the beginnings of an algorithm for mōrās, providing the groundwork for Nelson's (1991) more analysis focused work.

3.2.2 David Nelson

David Nelson's thesis (Nelson, 1991) uses Brown's work as a starting point, but focuses on the performance of the tani āvartanam (drum solo) on the mṛdaṅgam. The materials used in the study were five tanis performed by mṛdaṅgamists considered to be at the top of the profession (Palghat R. Raghu, Trichy S. Sankraran, T. K. Murthy, Vellore G. Ramabhadran and Karaikudi R. Mani) so can be taken as an accurate survey of the tani āvartanam circa 1991 (Nelson, 1991). The tanis were filmed, transcribed, analysed and discussed with the performer so as to get a professional opinion on the posited theories.

This thesis (Nelson, 1991) and the resulting book (Nelson, 2008) have been crucial resources because of the depth of analysis and wealth of transcribed materials. In particular, the differentiation between time-keeping phrases (sarvalaghu) and calculated phrases (kannakus), as well as the general formula for mōrās and explanation of compound mōrās that would likely have been missed or misunderstood when working with transcriptions alone.

3.2.3 S. Rajagopala Iyer and R. Krishna Murthy

Sangeetha Akshara Hridaya (A New Approach to Tāḷa Calculations) by Iyer (2000) is a theory book aimed at the Karṇāṭak percussionist that provides various practical methods for rhythmic calculation and clarification of terms. The book focuses on various methods of reaching the eḍuppu (starting beat of the melody) from any point in the cycle. Although under different names, the methods coincide with Nelson's theories (Nelson, 1991, 2008), confirming their suitability for implementation.

3.3 Review

To the author's knowledge there has been no algorithmic composition work focused on Karṇāṭak rhythm, with melody only slightly less overlooked (Bel, 2006). Fortunately there is a relative abundance of both analytical (Iyer, 2000; Ayyangar, 1972; Viswanathan and Allen, 2004; Hulzen, 2002; Nelson, 1991, 2008; Pesch and Sundaresan, 1996; Pesch, 1999) and instructional (Vinayakram and McLaughlin, 2007; Vinayakram, 2007; Lockett, 2008; Prakash, 2009) material to forge the beginnings of algorithmic composition of Karṇāṭak rhythms.

Chapter 4

Specifications and Requirements Analysis

The building of any software system necessitates requirements analysis, defined by Abran et al. as "the elicitation, analysis, specification, and validation of software requirements" (Abran et al., 2004, ch.2). These activities are advised as "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly." (Abran et al., 2004, ch.2).

The building of a system for working with Karṇāṭak rhythmic materials will benefit greatly from a formal analysis of research, user and system requirements as well as attention to professional considerations.

4.1 System Objectives

4.1.1 Research Requirements

Provide a suitable representation for Karṇāṭak concepts of rhythm

For Karṇāṭak musicians the tradition of oral teaching (Viswanathan and Allen, 2004, p.60) makes an additional form of representation unnecessary (Nelson, 1991, vol.3, p.1), yet this has not stopped attempts to establishing one.

Song texts have been recorded in physical form; dried palm leaves for centuries until mass printing arrived in the mid-nineteenth century, but contain only the required rāga and tāḷa without melodic or rhythmic notation (Viswanathan and Allen, 2004, p.25).

Not wishing to use western score notation (Nelson, 1991, vol.3, p.1) ethnomusicologists developed their own methods of transcribing Karṇāṭak rhythms, but as Hulzen's (2002) adoption of, and Nelson's (1991) rejection of Brown's (1965) notation demonstrates, a standard is yet to be set. On the other hand, musicians such as Lockett (2008), Vinayakram and McLaughlin (2007) and Vinayakram (2007) have been happy to adapt score notation to their needs.

With any form of representation there will be loss of information; a vibrato mark on a score does not give precise timings for the motion, nor does an orally transmitted melody produce the timbre of the instrument it is intended to be played on. The ubiquitous computer representation of music, MIDI, may be reasonably suited to representing keyboard music, but shows its weaknesses when used for wind instruments. Even digital recordings suffer a loss of information, albeit (usually) inaudible information.

The focus of this project being Karṇāṭak rhythms, a suitable representation must indicate rhythmic groupings (the koṉakkōl word e.g. Ta ka di mi) and timing information interpretable by the computer. For the purpose of this project information present in the source material used (percussion performances) such as timbre and dynamics are superfluous so can be lost in representation.

For practical purposes the representation must be clearly analogous to Karṇāṭak materials to ease the construction of, and experimentation with the system. For the evaluation of the system it is also necessary that the representation can be converted to an audio signal, with printing/scoring as a useful addition.

It will also be necessary to represent the higher level concepts of Karṇāṭak rhythm, namely Tāḷas and complete performances.

Develop methods to generate rhythmic phrases in the Karṇāṭak tradition

Modelling the traditions of Karṇāṭak rhythm will necessitate methods to generate new material according to a given context; tāḷa, gati (beat division) and laya (tempo). The main use of this new material will be the modelling of sarvalaghu.

Nelson (1991, vol.1 p.28) describes sarvalaghu as 'the shaping of time', while Sankaran uses the term 'flow patterns' (Sankaran (1977) in; Nelson (1991)). The purpose of sarvalaghu is to reinforce the flow of the tāḷa (Viswanathan and Allen, 2004, p.68) in a manner that is primarily propulsive, drawing attention to 'the flow of rhythmic time rather than to its design possibilities' (Nelson, 1991, vol.1 p.29). A western synonym for sarvalaghu might be 'groove' or 'feel', and the embellishment of these. Sarvalaghu is the style of play that will occupy most of the percussionists stage time, especially when accompanying a soloist (Nelson, 1991, vol.1 p.28).

These methods will be used to build archetypal structures from which variations can be generated to create sarvalaghu passages of coherent but varied phrases. Generated phrases will also be of use when generating rhythmic cadences.

Implement methods to generate the formalised rhythmic cadences of Karṇāṭak music

An essential element of Karṇāṭak drumming is the use of rhythmic cadences known as kaṇakkus. Kaṇakku is a form in which the percussionist creates patterns that disrupts the perceived flow of the tāḷa (Viswanathan and Allen, 2004, p.68). Kaṇakku is used for cadential purposes, and is a broad term for the various loosely defined forms mōrā, arudi, tīrmānam, kōrvai, and ta din gi ṇa tom (Nelson, 1991, vol.1 p.43).

Adapting the formulas provided by musicologists (Nelson, 1991, 2008; Iyer, 2000) to computational methods to re-produce these cadences will be critical to this project.

Identify and implement material manipulation techniques employed in Karṇāṭak composition and improvisation

A natural part of a Karṇāṭak drummer's performance is motivic development and embellishment (Nelson, 1991, vol.1 p.36). In order to model this it will be necessary to identify the various processes used and implement computational methods to reproduce them.

4.1.2 User Requirements

As the focus of the project is research-orientated there are no intended end-users but for the researchers themselves. However, consideration of hypothetical users would serve to guide the development in a direction that may be of use in a number of fields. While these potential uses are to be taken into account, for practical use some extension or customisation of the system is likely to be necessary.

Musicologists

A suitable way of representing Karṇāṭak rhythms would be of great use to musicologists, with the facility to make examples audible ensuring against ambiguities or misinterpretation. Transcription could be greatly eased by a mapping of solkaṭṭu word representations to an input device in a manner similar to the Bol Processor and Tabla bols (Bel, 1998). The mapping of whole words to single input keys would enable transcription at performance speed of music far beyond the instrumental technique of the musicologist.

Note: (The transcription of human examples for the evaluation (Chapter 6.1) was completed using shorthand names for the syllables/words, which were then replaced with `KonaWord.new` *method calls using the text editor's find and replace function).*

Karṇāṭak Teachers

Combining their own input with the methods for generating variation, percussion teachers could quickly provide students with a vast number of variations on an idea. The ability to play back the variations as audio would keep with the tradition of oral teaching (Viswanathan and Allen, 2004, p.60).

Karṇāṭak Musicians

Composition The automated methods of material generation and manipulation could be used as a source of new ideas for composition. Parameters such as tāḻa, gati and laya could be used to generate new material free from the composer's predispositions.

Experimentation outside the tradition is increasing among Karṇāṭak musicians; the group *Shakti* has always been a combination of jazz guitarist John McLaughlin and Hindustani Tabla player Zakir Hussain with various Karṇāṭak musicians, while musicians Selvaganesh Vinayakram and U. Shrinivas have both released 'fusion' albums (Vinayakram, 2004; Shrinivas, 2007) that display their upbringing in the Karṇāṭak tradition, but would not be considered a part of the classical form. For the inclined musician the system would provide familiar materials with the precision of computer, which could be exploited to explore and realise new and complex musical ideas outside the tradition and beyond the musician's ability

Accompaniment The system may be of use to the practising musician as an accompaniment tool; the user could decide on tāḻa, laya, and gati leaving the system to generate accompaniment material with the continuous, suble variations made by humans.

Non-Karṇāṭak Musicians

For non-Karṇāṭak musician the system could function as a dynamic library of new improvisational or compositional ideas. Interesting ideas may emerge from the cross-pollination of non-Karṇāṭak music with Karṇāṭak structures or methods of generating variation.

4.2 System Requirements

Having considered what will be required of the system, it is worth considering what will be required *by* the system.

4.2.1 A platform for symbolic representations and their manipulation

As there are no ready made, entirely suitable methods for representing Karṇāṭak rhythms on the computer it will be necessary to create one, for which a programming language is well suited. As previously mentioned (4.1.1) the representation will need to contain multiple pieces of information (e.g a koṉakkōl word, timing information etc), making a new class in an object-orientated programming language ideal.

SCLang

SCLang is the programming language and interpreter of SuperCollider (*SuperCollider Homepage*, 2009). As well as being a full object-orientated programming language it makes encoding of time-based routines trivial and has been built specifically to communicate with sound synthesis engines such as SCSynth and Max/MSP via Open Sound Control. SCLang also provides classes for interfacing with MIDI which will be of use when using Swar Systems' (2009) Karṇāṭak virtual instruments for evaluation.

4.2.2 An audio synthesis engine for playback of materials

In order to produce koṉakkōl audio examples it will be necessary to playback recordings of the various words. To avoid a vast number of recordings it will be necessary to vary playback rate so that overlap is avoided at fast tempos.

SCSynth

SCSynth is the synthesiser half of SuperCollider (*SuperCollider Homepage*, 2009) and is easily capable of variable rate playback of samples. The third party JoshUGens library by Joshua Parmenter (bundled as an extra with SuperCollider) provides the ability to change the rate of playback while maintaining pitch which will be useful for the sake of aesthetics.

4.3 Professional Considerations

4.3.1 Analysis of Material Under Copyright

In order to derive rules for generating Karṇāṭak music it is necessary to analyse music from the style. While some transcriptions are available from academic works (Nelson, 1991; Hulzen, 2002) the use of transcriptions under copyright (Vinayakram and McLaughlin, 2007; Nelson, 2008) as well as original transcriptions of music under copyright (Vinayakram, 2007) is necessary in ensuring a spectrum of playing styles is accounted for.

Article 3 of the British Computer Society's (BCS) Code of Conduct states "You shall ensure that within your professional field/s you have knowledge and understanding of relevant legislation, regulations and standards, and that you comply with such requirements." (BCS, 2008*a*). In this instance the relevant legislation is chapter 3 (Acts Permitted in relation to Copyright Works), section 29 (Research and private study), article oneof the Copyright, Designs and Patents Act 1988 (OPSI, 1988) which declares "Fair dealing with a literary, dramatic, musical or artistic work for the purposes of research or private study does not infringe any copyright in the work or, in the case of a published edition, in the typographical arrangement."

4.3.2 System Evaluation

In conclusion of this project it will be critical to adhere to article 3 of the BCS code of good practice; "Honestly summarise the mistakes made, good fortune encountered and lessons learned. Recommend changes that will be of benefit to later projects" (BCS, 2008*b*). It would be unrealistic to expect the system (or any, ever) to provide a 'general solution' for Karṇāṭak rhythm, especially as it is a continually growing art-form. As a result all possible improvements will be noted as potential future extensions.

4.3.3 Listener Evaluation

Article 9 of the BCS code of conduct begins "You shall not misrepresent or withhold information on the performance of products, systems or services...". In accordance with this the quality of output will not be tampered with before being presented to the listener.

The article continues "...or take advantage of the lack of relevant knowledge or inexperience of others." As a relatively esoteric area of research it is likely that the knowledge of Karṇāṭak music possessed by evaluators will vary. So that this can be taken into account, the evaluators exposure to Karṇāṭak music will be noted.

In order to provide more critical evaluation of material a number of experts in the field (musicologists Nelson and Pesch and Sundaresan and mṛdaṅgamist R.N. Prakash) will included in the evaluation.

Chapter 5

Design and Implementation

The system that has been designed and implemented provides a SuperCollider environment for working with Karṇāṭak rhythmic material. Three representation classes have been built; `KonaWord` for representing individual ko<u>n</u>akkōl words, `KonaTime` for grouping multiple `KonaWord` objects and `KonaTani` which can be considered a complete performance. The class `KonaGenerator` was built to provide methods for generating and manipulating `KonaWord` and `KonaTime` objects, storing them in a `KonaTani` instance.

As discussed in Section 2.2.3 the method of the system can be classified as Knowledge based (Papadopoulos and Wiggins, 1999), while the numerous motivations include 'Algorithmic Composition', 'Design of Compositional Tools' and 'Computational Modelling of Musical Styles' (Pearce et al., 2002).

5.1 Class Structure and Interaction

In a typical working environment an instance of `KonaTani` is created, providing the necessary `SynthDefs` and `TempoClock` for playback, as well as creating a `KonaGenerator` instance. Material can then be generated either by hand (instantiating `KonaWords` and `KonaTimes`) or using the `KonaGenerator` instance's generation methods and then mutated using the `KonaGenerator` mutation methods. `KonaWord` and `KonaTime` instances can be played directly, or added to the `KonaTani` to be played back with the tāḷa being clapped. The conceptual layout of the system is shown in Appendix B.

5.2 Representation Classes

The design of suitable representations was one of the most difficult challenges of the project. There were some considerable grey areas, notably the classification of the basic unit of Karṇāṭak rhythm, represented by the `KonaWord` class.

5.2.1 KonaWord

Each `KonaWord` represents the basic unit of Karṇāṭak rhythm; a number of pulses/syllables grouped as a single word sometimes known as a Tattakhara (Iyer, 2000, p.12). Each instance also includes specification of the number of jatis (syllables), the karve (relative duration of jatis), the gati (how the beat is subdivided) and the number of mātras it occupies (the number of sub-divisions), as well as a Routine for playback and a method for printing.

Basic Units While existence of basic units is commonly discussed (Brown, 1965; Nelson, 1991, 2008; Pesch and Sundaresan, 1996; Pesch, 1999; Vinayakram and McLaughlin, 2007; Viswanathan and Allen, 2004; Iyer, 2000), these studies have not required classification decisions of the same degree as necessitated by a computer representation. The most in depth of discussion of the ambiguities of these building blocks and their relationship to mṛdaṅgam fingerings can be found in Brown (1965, ch. XIV).

Important to this discussion is an understanding of the role of the basic unit in Karṇāṭak rhythm. As previously mentioned (Section 2.1.1), the tāḷa and gati do not imply a particular accent structure, any pulse may be accented as "accents are generated by phrase groupings" (Nelson, 1991, p.19). The ramifications of this are twofold; the first pulse of a phrase grouping is always accented and any accent implies the beginning of a phrase grouping, see Figure 5.1 for an example.

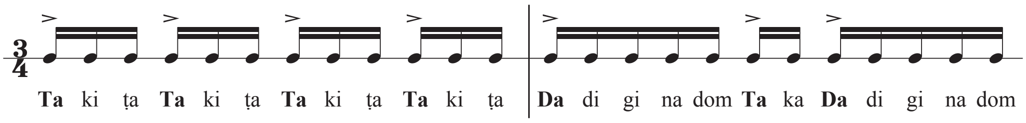


Figure 5.1: Two bars in a three beat tāḷa. The accents all result from the use of basic units, resulting in clearly different accent structures. From Vinayakram and McLaughlin (2007, ch.5)

There is an example found in Vinayakram and McLaughlin (2007, ch.3) that might seem to disprove this rule, however upon deeper inspection it is easily accounted for (see Figure 5.2 for the notation).

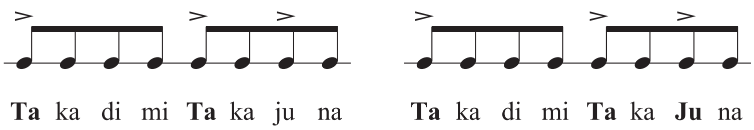


Figure 5.2: A common phrase alluding to two groups of four (as on the left), in fact consisting of a group of four with two groups of two (as on the right). From Vinayakram and McLaughlin (2007, ch.3)

From an initial assessment of this example it is possible to conclude that the single word Ta ka ju na has two accents (Ta and ju). This is understandable as Ta ka ju na is commonly used with a single accent in alternation with Ta ka di mi (Viswanathan and Allen, 2004, p.39). Brown (1965, p.236) notes that a group of four such as Ta ka di mi "can be analyzed as consisting of two pairs, Ta ka and Di mi, both of which also occur in isolation, but the combination of four syllables is at the same time an entity in itself." The same is true of Ta ka and Ju na, listed by Brown (1965, p.238) as possible groups of two or combined as a group of four. It should be noted that in Vinayakram and McLaughlin (2007) Ju na does not appear as a group of two outside of this context, what we are actually seeing is a context sensitive, four pulse phrase made up of two, two pulse phrases. [1]

So why not use less ambiguous Taka Taka? In fact this more distinct possibility crops up later in the chapter (Vinayakram and McLaughlin, 2007, ch.3) during an improvisation, however the context is clearly different (see Figure 5.3 for the notation). By looking at other bars in the improvisation it is clear that variation is being generated by permuting four groups of three and two groups of two. The separation of the two groups of two in bar 17 highlights the fact that these are separate entities, a fact that remains in bar 10 despite their proximity. This counter-example proves that there is more to the previous Ta ka Ju na (Figure 5.2) than just two groups of two; it is in fact two groups of two bound together as a four pulse phrase, unlike the two groups of two found in bar 10 of Figure 5.3. This is the only example found where a (usually single) word appears to have multiple accents, a five pulse phrase accents on the first and third pulse is never given as **Da** di **Gi** na dom but always as **Ta** ka **Ta** ki ṭa.

While this level of disambiguation may not be necessary in most situations, it has been vital to the design of the `KonaWord` class. Such discussion may also be relevant to the mṛdaṅgamist, for whom Ta ka and Ju na would indicate different fingerings (Brown, 1965, p.238).

Common ground Despite the possible ambiguities of construction for a group of four, most theoretical or solkaṭṭu focused examples (Pesch and Sundaresan, 1996; Pesch, 1999; Nelson, 2008; Iyer, 2000; Lockett, 2008; Hulzen, 2002; Ayyangar, 1972) use clearly distinct words for low numbered (1-4) phrase groupings. While there may be alternatives used (e.g. Ta ka di na or Ta ri gi du for Ta ka di mi) these are kept distinct from concatenations of smaller groups by associating syllables exclusively (e.g. always using di

[1]A pattern regarding capitalisation of solkaṭṭu words in this study may have been noticed by the reader. This is fully explained later in the paragraph relating the chosen solution to the ambiguities of these words

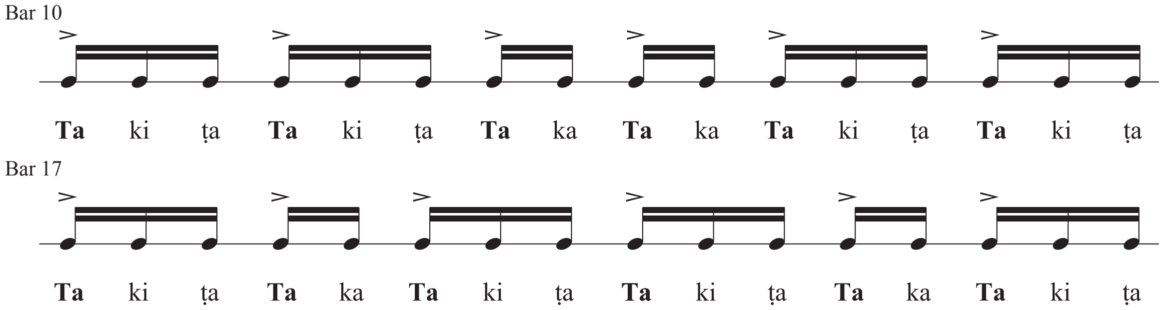


Figure 5.3: Two bars from an improvisation from Vinayakram and McLaughlin (2007, ch.3). The theme apparent in both bars is the use of permutations of four groups of three with two groups of two. Bar 10 uses Ta ka Ta ka in contrast to Ta ka Ju na of Figure 5.2.

na in the context of Ta ka di na, never as a group of two) with the previously exception of Ta ka Ju na in Vinayakram and McLaughlin (2007). Such ambiguities are usually confined to mṛdaṅgam playing, where the choice of syllable is more significantly weighted.

Variation Variation and difficulties begin to occur with numbers larger than four. For example Pesch and Sundaresan (1996, p.13) give the word for five as Ta dhi ki na tom [2] with an alternative of Ta ka ta ki ṭa. The first word is distinctly a group of five, while the second word could be mistaken for a group of two (Ta ka) and three (Ta ki ṭa) which would have a different accent structure to the first (as shown in Figure 5.4). This ambiguity is more apparent with numbers such as six for which Nelson (2008, p.15) gives as Ta ka di mi ta ka (4 + 2) or Ta ki ṭa ta ki ṭa (3 + 3) but as Pesch and Sundaresan (1996, p.13) point out could just as easily be Ta ka ta ka dhi na (2 + 4). Clearly all of these examples represent six pulse phrases, however, always implementing a group of six as a concatenation would prevent the possibility of a six pulse phrase with a single accent on the first beat.

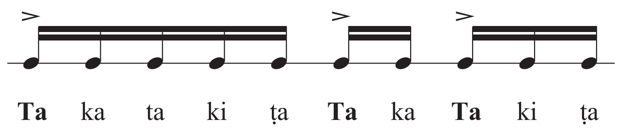


Figure 5.4: An ambiguity; a word made from concatenating Ta ka and Ta ki ṭa to get Ta ka ta ki ṭa, and a *phrase* made up of two words Ta ka and Ta ki ṭa.

Possible Solutions The most common solution that is at least acknowledged by all sources is to generate these longer words by concatenation, but treat them as a single word; placing an accent on the first pulse. Usually with this method as few groups are used for concatenation as possible e.g. Iyer (2000, p.13) and Lockett (2008, p.20) both give a group of nine as Ta ka di mi ta di gi na dom (4 + 5). While sources such as Viswanathan and Allen (2004, p.36) and Nelson (2008, p.15) offer the possibility of Ta ka di mi ta ka ta ki ṭa (4 + 2 + 3) (almost certainly because of their use of Ta ka ta ki ṭa for five) nowhere is a group of nine given as Ta ka ta ka ta ka ta ki ṭa (2 + 2 + 2 + 3) or similar, without it being regarded as a grouping of phrases as opposed to a single phrase.

Another solution is to use unique words for these high numbered groupings. The new words result from extending syllables of Da di gi na dom, as in Figure 5.5.

This idea is employed up to groups of seven syllables by Vinayakram and McLaughlin (2007) and up to nine by Pesch and Sundaresan (1996, p.13). While both sources accept the possibility of creating large numbers by concatenation, Vinayakram and McLaughlin (2007) choose to use these unique words exclusively while Pesch and Sundaresan (1996) give them as alternatives. This solution has two main

[2] This unfamiliar spelling for a group of five (usually Da di gi na dom) is exemplary of the previously mentioned (2.1.6) variation found in Solkaṭṭu words

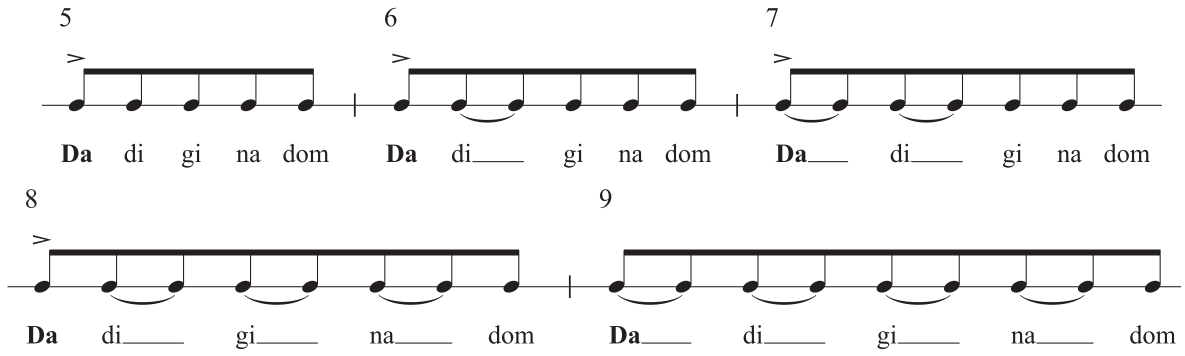


Figure 5.5: Da di gi na dom and various extensions for groups of six to nine. From Vinayakram and McLaughlin (2007, ch.4) and Pesch and Sundaresan (1996, p.13)

advantages; the unique word avoids ambiguity and the shared root word Da di gi na dom significantly eases transitions between the groups (e.g. groups of five to six) in recitation Vinayakram and McLaughlin (2007, ch.4). While this method is successful in disambiguating phrase groupings (and thus, accents) the use of two different syllable durations breaks the mould of these building blocks as it creates a sub-structure, more akin to a phrase made up of words than a low-level word itself. For example the group of seven in Figure 5.5 has a substructure of 2 + 2 + 3, which is *rhythmically* different from the first solution's Ta ka di mi ta ki ṭa. This grey area between words and phrases is highlighted by Brown's (1965, p.241) inclusion of the phrase Ta lan - da (commonly Ta lan - gu) in his list of four syllable groups because "the middle syllable is *always* long, and because it is always treated as a quadripartite structure." See Figure 5.6.

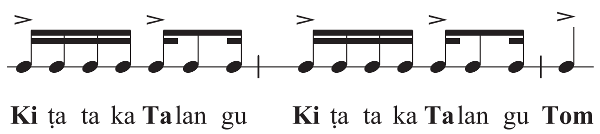


Figure 5.6: A quintessential phrase of fours. From Karaikudi R. Mani's tani āvartanam in Nelson (1991, vol.3 p.58)

Chosen Solution The solution chosen for this project has resulted from the prioritising of using one pulse duration for each word so as to avoid sub-structures. This decision means that for numbers above six a concatenation of words is used to create *new* words that are considered distinct from a concatenation of word objects. For example, Ta ki ṭa ta ki ṭa is preferable to Da di - gi na dom because of its single pulse duration, and is considered distinct from a phrase made up of two instances of Ta ki ṭa i.e. Ta ki ṭa, Ta ki ṭa.

For the purposes of this dissertation these basic building block words are made distinct via an enlargement of the first letter of the first syllable, and in the case of a phrase containing multiple words a comma is used to separate them. The use of a hyphen '-' indicates an extension of the previous syllable by an equal duration e.g Ta - ka - and Ta ka di mi are of equal durations. In score notation the first syllable is made bold and given an accent. Beaming is generally an indication of word grouping but is occasionally to group phrases where appropriate as with anomalies such as Ta lan - gu. See Figure 5.7 for an example. As the rhythms generated by words with more than one duration (e.g. Da - di - gi na dom) are a phenomenon of Karṇāṭak music, they will be accounted for in a different way (discussed in 5.2.2). The list of basic units and their jatis used for this dissertation can be found in Table 2.2.

Maximum Size In theory words of any length could be constructed in this way (e.g. a 13 pulse word with a single accent being Ta ka di mi ta ka ju na da di gi na dom), in practice this is not the case. The maximum number of syllables for a single word is commonly given as nine (Nelson, 2008; Lockett,

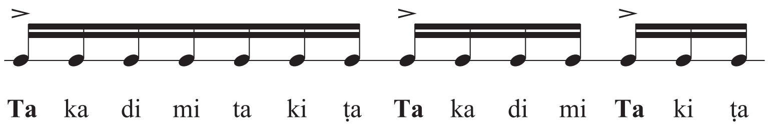


Figure 5.7: A score representation of the phrase Ta ka di mi ta ki ṭa, Ta ka di mi, Ta ki ṭa. Notice how the enlargement of letters and use of commas correspond to the beginning of words/groupings and the resulting accents and bold text. as well as the use of beaming (where possible) and bold text. Also notice how the word for the first group of seven is a concatenation of the words for four Ta ka di mi and three Ta ki ṭa, yet this is considered a group of seven. This is in contrast to the following groups of four and three which clearly separate entities.

2008; Iyer, 2000) which ties in with the largest of the five rhythmic families 2.1.3. As Nelson (2008, p.15) notes "longer phrases usually include rests", which would push such phrases out of this project's definition of a word (`KonaWord`) and into the category of a phrase (discussed later in 5.2.2)

Word / Jatis

Each `KonaWord` instance contains an array of jatis (syllables, stored as symbols) which are combined to form a word. The instantiation method `new` has an argument for defining the number of syllables, which determines which the of syllables and ultimately the word to be used. Storing the syllables as symbols makes them useful for printing and playback buffer selection. The `KonaTani` class (5.2.3) contains an array of all possible syllables and an array with their corresponding audio buffers. A `KonaWord` instance can use the `indexOf` method of the syllables array with each syllable symbol to find the correct buffer for playback.

Gati

The gati (sub-division, sometimes called Nadai) is defined at instantiation and is a determining factor of each syllables' duration in relation to the beat. The gati is limited to the values of the five rhythmic families (2.1.3) four, three, seven, five and nine (see Table 2.1). The sub-divisions generated by the gati are called mātras, so one beat in tiśra (three) gati will have three mātras [3]. For an illustration on the role of the gati see Figure 5.8.

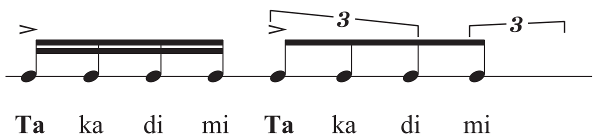


Figure 5.8: Notation of two four syllable `KonaWord` instances, the first with a gati of four, the second with a gati of three.

Karve

The Karve is the number of mātras each jati of a word 'occupies' (Iyer, 2000, p.11). For example a caturaśra gati word with a karve of two would be double the duration of a caturaśra gati word with a karve of one, as illustrated in Figure 5.9

[3]There is a disagreement in the literature as to the definition of mātra. Nelson (1991, vol.1 glossary) uses the term in regards to the number of beats per akṣara (count) of the tāḷa, while Pesch and Sundaresan (1996, glossary) and Iyer (2000, p.11) use the term to describe the sub-divisions created by the gati

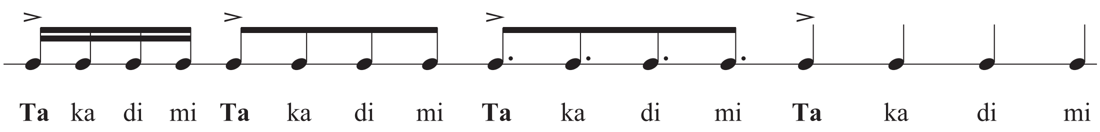


Figure 5.9: Notation of four caturaśra gati `KonaWord` instances, with karves of one to four respectively.

Routine

Each `KonaWord` instance contains a Routine for playback via a number of possible methods. When run the Routine will simultaneously play and print each syllable/hit of the word as well as its duration relative to the beat, as in Figure 5.10. The default method of playback is to use the `konaHit` SynthDef to play time stretched recordings (stored in buffers) of each syllable via the `PV_PlayBuf` UGen to avoid overlap. Alternatively, routines of MIDI signals can be sent to control a synthesiser, sampler or virtual instrument outside of SuperCollider.

```
Ta 0.25 ka 0.25 di 0.25 mi 0.25
```

Figure 5.10: An example of printing from the Routine of a four syllable caturaśra gati `KonaWord` with a karve of one.

Printing

Printing of a word is possible outside of a Routine via the `postWord` method. `postWord` is capable of printing the syllables of the word, the duration of each syllable as a decimal relative to the beat, and the duration as a fractional value relative to western concepts of beat values (e.g. a quaver is 1/8, a semi-quaver is 1/16 etc). Any or all of these print options can be omitted via arguments. An example printout is shown in Figure 5.11.

```
[ Da       , di       , gi       , na       , dom      ]
[ 0.333    , 0.333    , 0.333    , 0.333    , 0.333    ]
[ [ 1, 12 ], [ 1, 12 ], [ 1, 12 ], [ 1, 12 ], [ 1, 12 ] ]
```

Figure 5.11: A printout of a five syllable tiśra gati `KonaWord` with a karve of one, courtesy of the `postWord` method with all printing options enabled.

5.2.2 KonaTime

The `KonaTime` class was built to group together `KonaWord` as well as `KonaTime` instances to create musical structures ranging from a phrase to a whole piece. Rather than use a set of more specific classes to represent periods of time (e.g. a class for a phrase, another for a tāḷa cycle etc) the more generic `KonaTime` was built. This generic class allows for all materials to be treated in the same way, with the same methods (especially as a `KonaTime` can hold just a single `KonaWord`). The use of a generic class as structure of time allows for unconventional treatments of material, e.g. building a traditional cadential structure (see 5.3.3) out of a whole piece of music rather than a single phrase.

Design

`KonaTime` is a subclass of `List`, making available all of SCLang's `List`, `SequenceableCollection` and `Collection` methods for traditional (see 5.3.3) and untraditional manipulations. The availability of these methods also eased the development process and makes the material more accessible to those familiar with SCLang.

`Konatime` stores and makes accessible the following information; the words contained, the combined and individual duration(s) and jatis of contained objects and the duration in number of tāḷas. As with `KonaWord` a Routine for playback and a method for printing are included as well as a convenience concatenation method.

Use

`KonaWord` instances can be added to a `KonaTime` in the same way as any object can be added to a List. Adding multiple `KonaWords` will result in a short musical phrase, which may in turn be added to another `KonaTime` representing a musical passage.

The ambiguous words/phrases outlined in Section 5.2.1 such as Ta lan - gu and Da - di - gi na dom can be created using a `KonaTime` and multiple `KonaWords` as in Figure 5.12. While the syllables may be different, the rhythm is the same and the hierarchy of syllable accents is kept inline with real `KonaWords` by stacking an additional accent on the first syllable if all items in the `KonaTime` are `KonaWords`. This is obviously a form of compromise that could be resolved as a future extension, see 7.1.

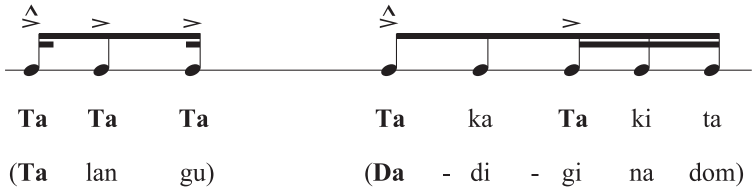


Figure 5.12: Phrases Ta lan - gu and Da - di - gi na dom as represented by `KonaWords` grouped in a `KonaTime` notice the additional accents, imitating the accent structure of `KonaWords`.

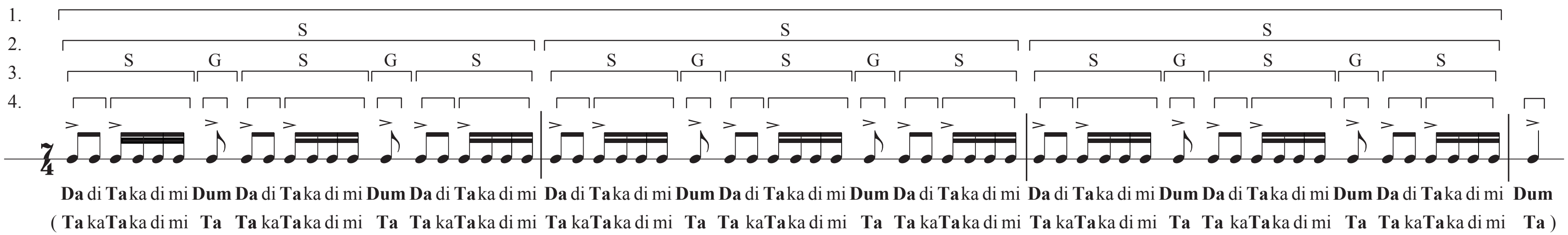


Figure 5.13: A compound mōrā from Vinayakram and McLaughlin (2007, ch.6 Finale). Key: 1. `KonaTime` instance grouping the compound mōrā. 2. `KonaTime` instances the compound mōrā statement phrases. 3. `KonaTime` instances grouping the constituent mōrā statement and gap phrases. 4. `KonaWord` instances.

5.2.3 KonaTani

The `KonaTani` class represents a complete piece of music. Although it was mentioned (5.2.2) that `KonaTime` was capable of this task (indeed one is used inside a `KonaTani` for storage), `KonaTani` accounts for more musical aspects, which are required for the automated features of the system.

Design

Specifiable attributes include the laya (tempo), tāḷa, starting gati and gatis to change to (a change of gati is a ubiquitous feature of Karṇāṭak drum solos (Nelson, 1991, vol.1 p.90)) and the `SynthDef` to use for playback; `konaHit` for Ko̱nakkōl syllables or `MIDIPlay` for MIDI playback.

As with `KonaWord` and `KonaTime`, `KonaTani` features a routine for playback, but also contains a `TempoClock` object on which the routine is played. Also stored is a second routine for clapping the tāḷa using a `SynthDef` by Magnusson (2009).

As only one instance of `KonaTani` will be created per piece of music (unlike `KonaWord`), the `konaHit` SynthDef and buffers for playback are loaded and allocated during `KonaTani` instantiation. Instantiation also results in the creation of a `KonaGenerator` object, discussed in Section 5.3.

Use

A `KonaTani` can be instantiated with user specified settings with the `new` method, or with randomly selected settings with the `rand` method. If the `new` method is used a piece can be created by hand using `KonaWord` and `KonaTime` instances with the `KonaGenerator` instance.

5.3 Generation and Manipulation Class and Methods

The `KonaGenerator` class was built to handle all of the generation and manipulation requirements of the system. The methods were not written in an attempt to model the mental processes of the Karṇāṭak musician, they are the result of analysis and as such model the output of these processes. Many of the generation methods make use of the mutation methods, the decision was made to keep these processes separate to maintain coherency; creating an archetypal phrase and altering it rather than continuously creating new phrases. Almost all techniques have been implemented as methods that require manual specification of arguments with an additional probabilistic method for automation. This separation gives the system great flexibility, making it of use to many users (4.1.2) for many purposes (2.2.2).

5.3.1 Generation Overview

Integer Partitioning - ZS2 Algorithm

Integer partitioning has been incredibly useful for generating Karṇāṭak music. Given a number of beats and the gati it is trivial to calculate the total number of pulses. This value may be halved or doubled (2.1.3) and then partitioned, with the resulting parts used to create phrase groupings which in turn create accents ("accents are generated by phrase groupings" (Nelson, 1991, vol.1 p.19)) and a musical structure (see Figures 5.14 and 5.15 for examples).

Integer partitioning in this system is courtesy of the ZS2 algorithm (Figure 5.16) by Zoghbi and Stojmenović (1998). ZS2 is an unrestricted integer partition algorithm (no limitation on part size) that produces partitions in lexicographic order with constant average delay (Zoghbi and Stojmenović, 1998). ZS2 was implemented in SCLang as `allPartitions`. The method has an optional minimum and maximum part size with a cap on the maximum at nine to match the maximum size of a `KonaWord`. Additionally the `randomPartition` method was written return a single random partition.

For integers greater than 40 the ZS2 implementation started to slow significantly, with a partition of 100 averaging `54.92 seconds`. As a workaround the arrays for 40-100 were stored as files and are read instead of calculated. Numbers up to 39 are kept as calculations as the `allPartitions` method has the option to exclude certain partitions (e.g. those with too great a number of unique parts) which is more efficient than removing such partitions afterwards, which is necessary with arrays loaded from files.

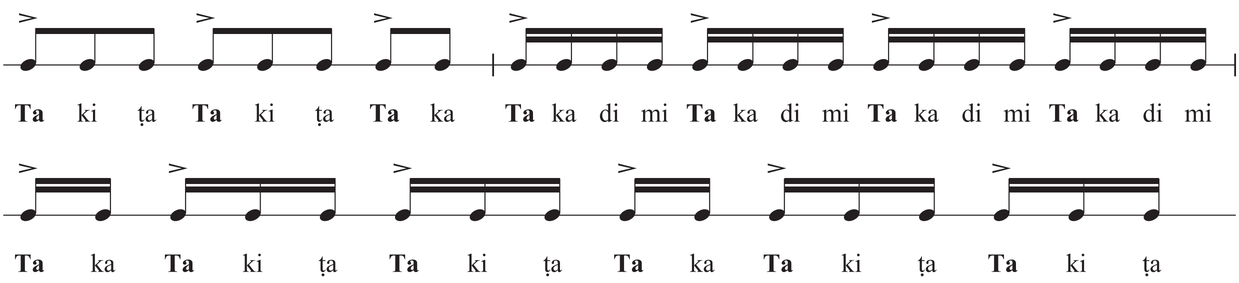


Figure 5.14: Three examples of pulse partitioning of four beats of caturaśra gati. The first bar sees the 16 pulses treated as eight with twice the duration, partitioned into 3 + 3 + 2. The second is a partition of the 16 into 4 + 4 + 4 + 4. The third is a partition of 2 + 3 + 3 + 2 + 3 + 3, which is more likely a partitioning of eight pulses into 2 + 3 + 3 with a repetition. From Vinayakram and McLaughlin (2007, ch.2)

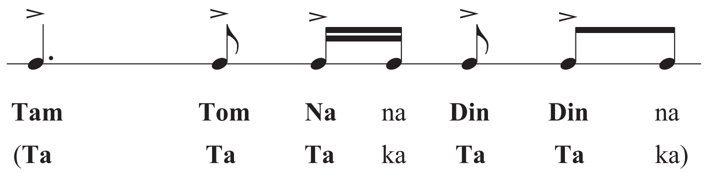


Figure 5.15: A four beat caturaśra phrase partitioned into 6 + 2 + 2 + 2 + 4. Notice how the part value is used as the number of mātras (sub-division parts) for each word and not necessarily the number of pulses, e.g. the second word Tom and the third Na na have the same duration in mātras but a different number of pulses. From Karaikudi R. Mani's tani āvartanam (aksharas one and two from cycle three) in Nelson (1991, vol.3 p.58). Nelson's solkaṭṭu is given as the top line, with this project's underneath. Note: As Nelson does not use western notation or any formatting in regards to capitalisation, the conventions adopted for this project have been applied to both lines of solkaṭṭu..

Permutation

Once a number of pulses has been partitioned, so long as there is variation in part size, a number of variations up to the factorial of the integer can be generated by permutation. See Figure 5.17. A method to return all permutations (`allPerms`) and a method to return a single (random) permutation (`randomPerm`) were written.

Pruning

Number of unique parts After experimentation with partitioning and permutation for material generation it was felt that the partitions would sometimes lack an identity. Comparison with real-world highlighted the fact that typically no more than three unique part sizes were used, examples from Vinayakram and McLaughlin (2007, ch.6, ch.2); Figure 5.13 is up of phrases of 4 + 4 + 2 (two unique parts), Figure 5.14 uses 6 + 6 + 4 (three unique parts, could be seen as 3 + 3 + 2), 4 + 4 + 4 + 4 (one unique part) and 2 + 3 + 3 + 2 + 3 + 3 (two unique parts), Figure 5.17 uses permutations of 3 + 3 + 2 (two unique parts), an example by Mani from Nelson (1991, vol.3 p.58); Figure 5.15 uses 6 + 2 + 2 + 2 + 4 (three unique parts).

As a result of this observation the `removeGreaterThan` method was written which will remove from a given collection any partition with more unique parts than an given value. A weight parameter is included with a default of 0.97 to occasionally allow partitions with a high number of unique parts.

Part sizes Although it has been said that a tāḷa and gati have no inherent accent structure (Section 2.1.1) there are often cases where it is desirable to highlight the current environment, e.g. sarvalaghu patterns, on which Nelson (1991, vol.1 p.29) comments "phrases are arranged in patterns that bear an integral relation with the given akṣara structure. These patterns are arranged into groups that bear an

Algorithm ZS2

```
for i ← 1 to n do x[i] ← 1; output x[1..n];
x[0] ← −1; x[1] ← 2; h ← 1; m ← n − 1; output x[1..n];
while x[1] ≠ n do
            if m − h > 1
               then h ← h + 1; x[h] ← 2; m ← m − 1;
               else  j ← m − 2;
                      while x[j] = x[m − 1] do x[j] ← 1; j ← j − 1;
                      h ← j + 1; x[h] ← x[m − 1] + 1;
                      r ← x[m] + x[m − 1](m − h − 1); x[m] ← 1;
                      if m − h > 1
                         then x[m − 1] ← 1;
                      m ← h + r − 1
            output x[1..m]
```

Figure 5.16: Pseudo code for the ZS2 algorithm by Zoghbi and Stojmenović (1998).

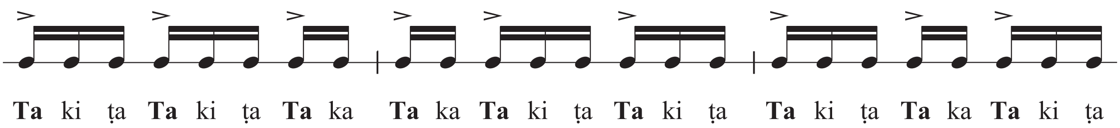


Figure 5.17: Three permutations of the 3 + 3 + 2 partition of eight. From Vinayakram and McLaughlin (2007, ch.3)

analogous relationship with the tāḷa structure.” One way to achieve this might be to use groupings of sizes that are strongly related to the structure, e.g. groups of two, four and eight in ādi tāḷa (8 beats) caturaśra gati. However, as Nelson (1991, vol.1 p.31) points out it is possible to “use a contrasting organization of pulses, thereby generating a more complicated relationship with a beat or pair of beats” with which he gives the example of a 3 + 3 + 2 pattern for two beats of caturaśra gati. The extent to which it is desirable to use patterns that contrast with the structure of the environment is context dependant, Nelson (1991, vol.1 p.40) gives the example in Figure 5.18 as being “closer to the kaṇakku end of the spectrum”.

To account for both of these situations the method `removeThoseContaining` was written which will remove from a collection of partitions all those which contain specified part sizes. An optional argument is provided so that each value can be given a probability of removal.

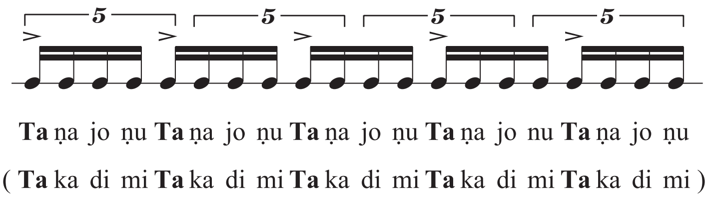



Figure 5.18: A four akṣara khaṇḍa gati pattern made up of groups of four. An example of how groupings outside of the rhythmic family of the gati can be used. From Nelson (1991, vol.1 p.40)

Converting to `KonaWords`

As the initial generation methods work with arrays of integers, the method `partsToWords` was written to convert these into `KonaWords`. In addition to the partition array parameter, two boolean parameters are included for control over the way `KonaWords` are generated. The first parameter determines whether or not `KonaWords` can be created with a single jati (Ta) and a karve equal to the partition size, the second specifies the possibility of the opposite; a `KonaWord` with a number of jatis equal to the partition size and a karve of one (Figure 5.19). If both (or neither) parameter is set to true, each possibility is given a 50% chance.

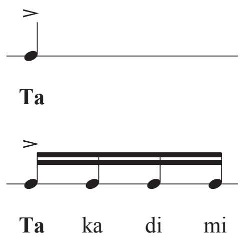


Figure 5.19: Two possible conversions of the part size four into a `KonaWord`. The top line with one jati and a karve of four, and the bottom line with four jatis and a karve of one.

Combining

As the length of phrases and speed increases so to does the use of larger phrase groupings. In Figure 5.20 we can see a 2 + 2 + 2 + 2 + 4 structure, while both examples in Figure 5.21 use larger groupings (predominantly fours in the first example and eight and six in the second) and are twice the speed.

The reasons for this have not been written about, however Vinayakram and McLaughlin (2007, ch.5) give the opinion that a subdivision as in Figure 5.20 is ‘boring’. In recitation it is also noticeably easier to work with larger groupings at faster tempos, the reader is invited to compare repetition of Ta ka Ta ka and Ta ka di mi at 160bpm. The use of larger groupings also reduces the total number of mental objects being dealt with, easing memorisation (Miller, 1956) which is key to the oral tradition of Karṇāṭak teaching (Viswanathan and Allen, 2004, p.60).

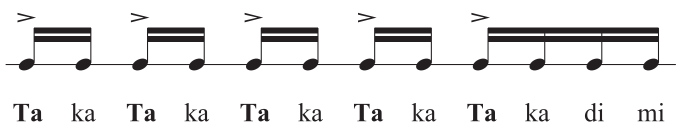


Figure 5.20: An phrase with an uncommon use of many groups of two as opposed to larger groupings, yet still possible to perform at 90bpm. From Vinayakram and McLaughlin (2007, ch.5)

To account for this phenomena the method `combine` was written to regroup the total number of pulses in a phrase (`KonaTime`) into as few `KonaWords` as possible. Additionally `combineSimilar` was written to combine adjacent identical `KonaWords`, with arguments for the maximum combination size, the maximum number of `KonaWords` to combine and a probability (default of one) determining combination success.

5.3.2 Sarvalaghu Generation

As previously discussed (4.1.1) sarvalaghu patterns are the time-keeping structures that reinforce the flow of the tāḷa (Viswanathan and Allen, 2004, p.68) and form the majority of accompaniment playing. Unlike the calculated forms of playing (5.3.3) for which formulae have been documented, accompaniment is considered “extremely difficult to teach, and to this day is learned primarily by example and absorption.” (Nelson, 1991, vol.1 p.iv). Sarvalaghu patterns vary between musicians and contexts. For example, in the vilamba kāla stages of a tani āvartanam the patterns tend to use two mātra syllables/strokes with accents on the beat or half beat, while in the madhyama kāla section one mātra syllables/strokes are used with accents on any pulse in the beat (Nelson, 1991, vol.1 p.37) (see Figure 5.22).

The sarvalaghu generation in this project has focused on the vilamba kāla sarvalaghu styles of the five tani āvartanams in Nelson (1991). As these tani āvartanams were all at a similar tempo and in

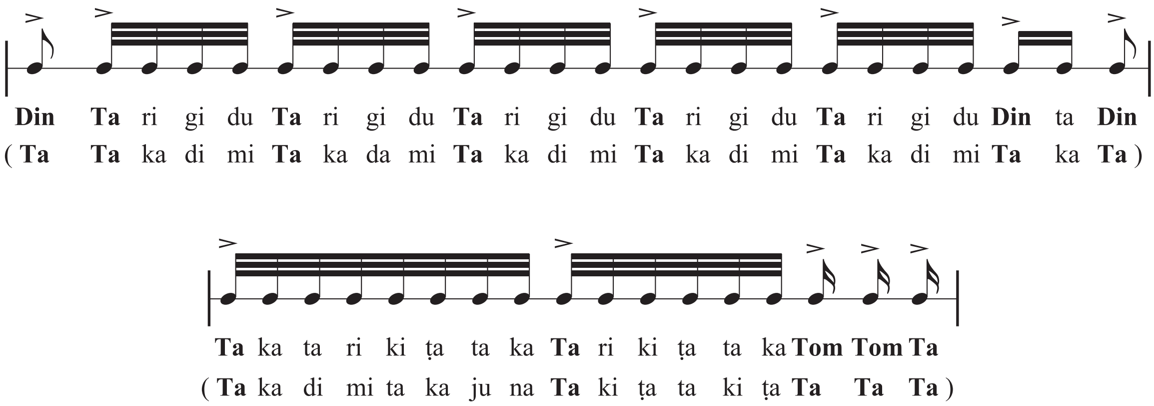



Figure 5.21: Two fast phrases using large groupings such as four, six, and eight as opposed to small groupings of two or three. This tends to be the norm at speeds such as this (performed at 85bpm). The first phrase is from an exposition of a composition by Palani Subramania Pillai by Vellore G. Ramabhadran as part of his tani āvartanam in Nelson (1991, vol.3 p.9) . The second phrase is from Trichy S. Sankaran's tani āvartanam in Nelson (1991, vol.3 p.162)

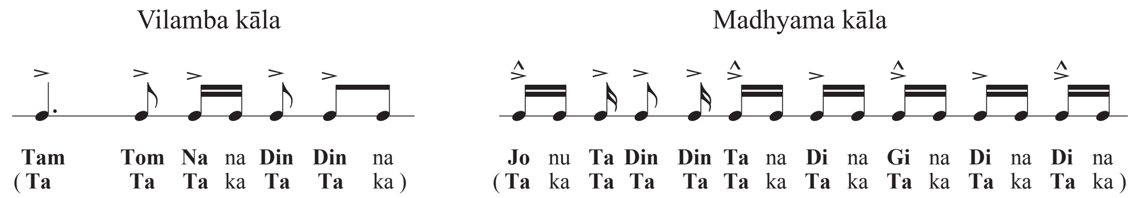



Figure 5.22: Two phrases from Karaikudi R. Mani's tani āvartanam in Nelson (1991, vol.3 p.58, p.65), highlighting the difference in sarvalaghu patterns between sections of an improvisation. The vilamba kāla example primarily uses words with jatis of at least two mātras, with accents only on the whole or half beat. The madhyama kāla example primarily uses words with one mātra jatis and contains accents on quarter beats.

the same tāḷa and gati, additional patterns in other settings were transcribed from Vinayakram (2007) (Appendix C).

Analysis revealed that sarvalaghu patterns were not simply improvisation within a setting, instead an archetype phrase is set up and used as the basis for improvisation (see Figure 5.23). In some cases a new phrase is developed from the archetype, and adopted as the new foundation for improvisation (see Figure 5.24).

Phrase Length A prevalent feature of these phrases was a neat fitting of repetitions within the tāḷa, as a result attempts were made to calculate phrase length as a division of the tāḷa. This model had problems accounting for phrases at faster tempos which often occupied entire tāḷas (Figure 5.25). To take tempo into account the length of phrases were considered in terms of absolute time (see Figure 5.26), where a striking similarity was found between phrases; they were all between 2 and 3.375 seconds, fitting well with the perceptual present and short-term memory of humans within which most musical and spoken phrases fit (London, 2004, ch.2) (Collins, 2008, p.3).

`vSarvaPhraseLength` was written to calculate a suitable duration (in beats) for a sarvalaghu phrase. The maximum number of beats for a phrase is calculated using the absolute time for one beat and the maximum cognitive time for a phrase. Regardless of absolute time a limit of five beats is enforced as most phrases stayed below this, possibly because of information load (Collins, 2008, p.3). `vSarvaPhraseLength` then find the longest phrase length that fits neatly into the tāḷa once or in multiples of two.

Phrase Generation Phrase generation is split into two methods `vSarvaPhrase` and `vSarvaPhraseAuto`; the former creating phrases according to a given phrase duration in mātras, and the later simply call-

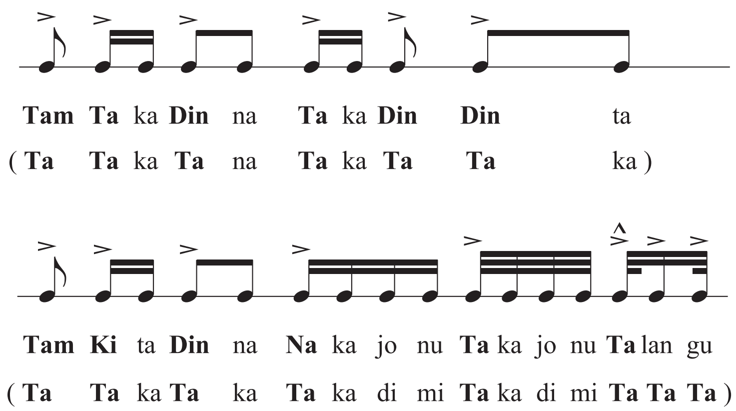



Figure 5.23: Two phrases from from the vilamba kāla of Trichy S. Sankaran's tani āvartanam in Nelson (1991, vol.3 p.162). The second phrase can clearly be seen to be a variation on the first, with density alteration of the last three syllables.

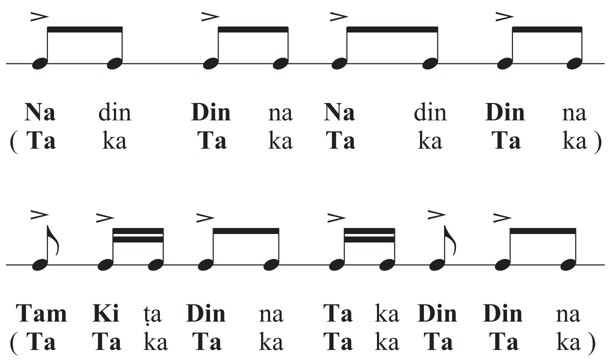


Figure 5.24: Two phrases from from the vilamba kāla of Trichy S. Sankaran's tani āvartanam in Nelson (1991, vol.3 p.162). The first is a common basic sarvalaghu pattern (Nelson, 1991, vol.1 p.32) which is transformed into the second phrase via minor adjustments. The second phrase is then adopted as an archetype phrase for the rest of the vilamba kāla (Nelson, 1991, vol.3 p.162-171).

ing `vSarvaPhrase` passing in the results of the `vSarvaPhraseLength` method multiplied by the gati. `vSarvaPhrase` was written to generate fundamental sarvalaghu phrases from which variations and new motifs can be created. In accordance with Nelson's (1991, vol.1 p.37) observation that patterns in vilamba kāla primarily use words with two mātra jatis, the minimum part size (for partitioning) is set to two. The maximum value is set to the gati if the phrase is long, and the phrase duration if not, an array is created from these values e.g. `[2,3,4,5]` for khaṇḍa gati. An array of weights for each part size is constructed with bias for part sizes that are included in the gati's rhythmic family (2.1.3). For khaṇḍa (five) and misra (seven) gati part sizes two and three are given an additional bias because of

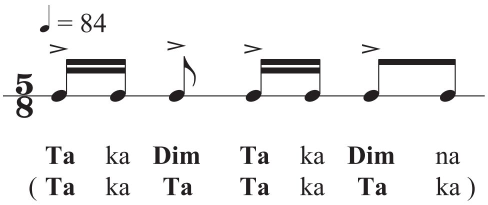
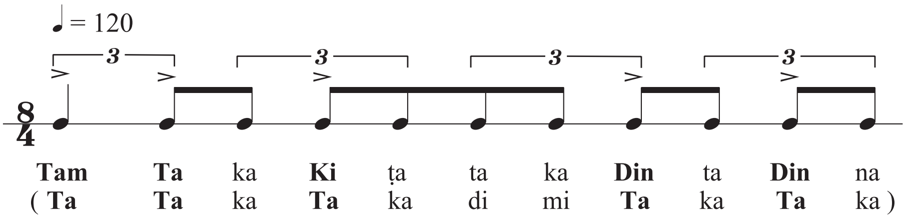
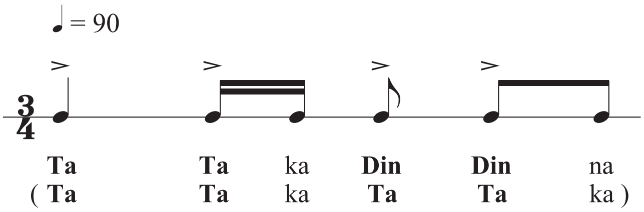


Figure 5.25: A khaṇḍa cāpu sarvalaghu phrase by Vinayakram (2007, Disc.1 ch.6) occupying the entire cycle.

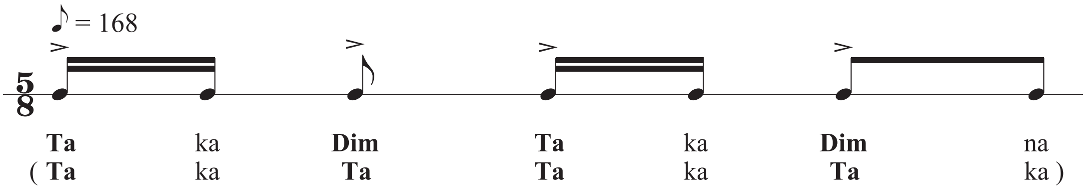
Adi tāḷa tiśra gati 120 bpm from Vinayakram (2007, Disc 1, ch.6).
Time: 2 seconds

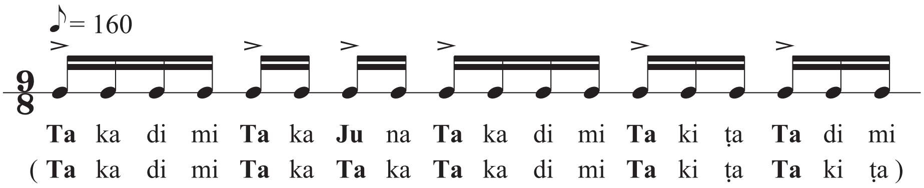


Rupaka tāḷa caturaśra gati 90 bpm from Vinayakram (2007, Disc 1, ch.6).
Time: 2 seconds



Khanḍa cāpu caturaśra gati 168 bpm from Vinayakram (2007, Disc 1, ch.6).
Time: 1.8 seconds



Sankīrṇa cāpu caturaśra gati 160 bpm from Vinayakram (2007, Disc 1, ch.6).
Time: 3.375 seconds

Adi tāḷa caturaśra gati 96 bpm from Trichy Sankaran's tani āvartanam in Nelson (1991, vol.3 p.162).
Time: 2.5 seconds

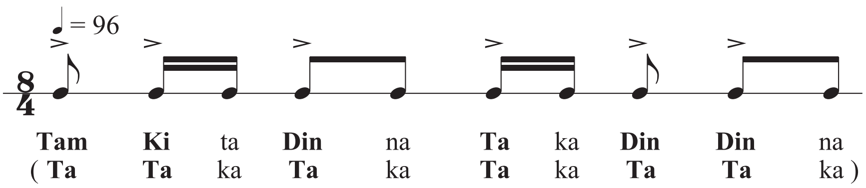


Figure 5.26: Sarvalaghu phrases in various contexts, with absolute times all between 2 and 3.375 seconds.

their suitability for the gati.

The `phraseMatras` value is partitioned with `allPartitions` (passing in the maximum part size), with any partition with more than four parts being removed by `removeGreaterThan`. Partitions with particular part sizes are probabilistically removed by passing the array of part sizes and weights to `removeThoseContaining`. A partition is randomly selected and finally permuted with `randomPerm`. A custom transformation from integers to `KonaWords` then takes place; any even values greater than two are given a 75% chance of being halved in terms of jatis and doubled in terms of karve (e.g. a four mātra TA becomes a two mātra per jati TA KA), all other values become single jati (TA) `KonaWords` with the part number used for the karve. Finally the phrase is passed into `combineSimilar` with a maximum combination size of four and some syllables are probabilistically muted to create variation (see **Muting Jatis** in Section 5.3.4). See Figure 5.27 for an example of this process.

Phrase Development As previously mentioned (Section 5.3.2) it is common for a phrase to be varied enough that it becomes a separate entity, which may then be adopted as a primary phrase from which further variations are created (see Figure 5.24). This is achieved using the `mutatePhrase` method which calls upon a number of sub-routines. This process is discussed in detail in Section 5.3.5.

Phrase Suffixes As Nelson (1991, vol.1 p.88) notes it is very common for sarvalaghu phrases to be concluded with a 'suffix' that may also serve to provide variety as well as introduce cadences (see the discussion of formal cadences in Section 5.3.3). See Figure 5.28 for an example.

From analysis of the suffixes in Nelson's (1991, vol.3) five tani āvartanam it was apparent that the most typical feature was an increase of jati density; using a greater volume of shorter notes. The `addSuffix` method was written to add a suitable suffix to the end of a given phrase. This method collects the jatis that make up the last quarter of the phrase and increases the density; double for the first few jatis, and possible quadruple for the middle and last jatis. For a more detailed explanation of jati density alteration see **Altering Jati Density** in Section 5.3.4.

Statistical Generation

Statistical generation of sarvalaghu patterns were briefly experimented with; a model was made of the first four cycles of Trichy S. Sankaran's tani āvartanam in Nelson (1991, vol.3 p.162). This was implemented in as the method `vSarvaStat`, but was not developed beyond a basic state.

5.3.3 Kannaku Generation

Simple Mōrās

A mōrā is "the fundamental cadential structure of Karṇāṭak music" (Nelson, 1991, vol.1 glossary). Nelson came up with a simple formula for mōrās (Figure 5.29) and notes "a mōrā usually consists of a phrase or statement repeated three times with separations that may be articulated. Its structure sets up a temporary tension with that of the tāḷa that is usually resolved at an important structural point in the cycle." While it is necessary for the statement to be at least one pulse in duration, the gap may be 0 (Nelson, 1991, vol.1 p.46). This allowance neatly combines two of Iyer's (2000, p.68, p.79) mōrā formulas (Figure 5.30).

As the gap and the final resolving beat often use the same sound (as in Figure 5.31) the structure of some Mōrās with a gap greater than 0 is easy to misinterpret as just three statements (as Brown (1965, vol.1 p.151) did), as shown in Figure 5.31. While this example is understandable, with a mōrā such as Figure 5.32 the interpretation is clearly invalid.

```
Call: vSarvaPhraseAuto
    Call: vSarvaPhraseLength
        return  4                              //Phrase duration in beats

    Call: vSarvaPhrase(vSarvaPhraseLength*gati: 16)    //Total number of matras

        Call: allPartitions
            return   [ [ 4, 3, 3, 3, 3 ], [ 4, 4, 2, 2, 2, 2 ],
                       [ 4, 4, 3, 3, 2 ], [ 4, 4, 4, 2, 2 ],
                       [ 4, 4, 4, 4 ].... ]
        Call: removeGreaterThan
            return   [ [ 4, 3, 3, 3, 3 ], [ 4, 4, 2, 2, 2, 2 ],
                       [ 4, 4, 3, 3, 2 ], [ 4, 4, 4, 2, 2 ],
                       [ 4, 4, 4, 4 ]... ]
        Call: removeThoseContaining
            return   [ [ 4, 4, 2, 2, 2, 2 ], [ 4, 4, 4, 2, 2 ],
                       [ 4, 4, 4, 4 ] ]

            Call .choose
                        return  [ 4, 4, 4, 2, 2 ]

        Call: randomPermutation
            return  [ 4, 2, 4, 4, 2 ]

        Conversion to KonaWords:
            [ Ta , ka , Ta , Ta , ka , Ta , ka , Ta  ]
            [ 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]

        Call: combineSimilar
            return
            [ Ta , ka , Ta , Ta , ka , di , mi , Ta  ]
            [ 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]

        Call: randomMuteJati
            return
            [ Ta , ka ,  - , Ta , ka , di , mi , Ta  ]
            [ 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]

        return
            [ Ta , ka ,  - , Ta , ka , di , mi , Ta  ]
            [ 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]
```

Figure 5.27: An example of the phrase generation process in Ādi tāḷa at 80bpm.

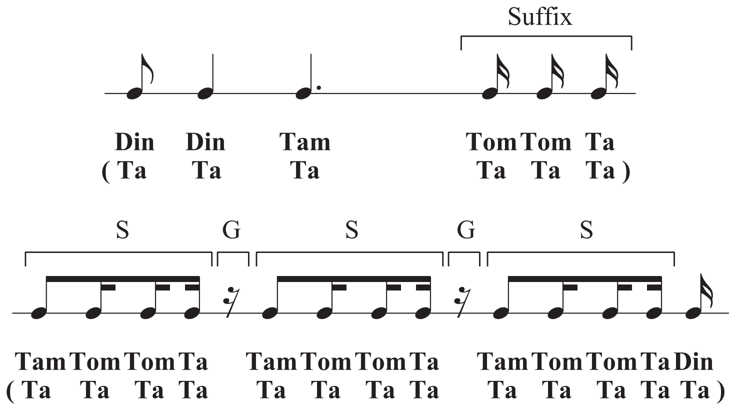


Figure 5.28: A phrase with a suffix from Palghat R. Raghu's tani āvartanam (Nelson, 1991, vol.3 p.104-105) (top line), and a mōrā that appears later using the suffix for the statement sections.

$$xyxyx \tag{5.1}$$

or

$$(Statement)(Gap)(Statement)(Gap)(Statement) \tag{5.2}$$

Figure 5.29: Nelson's (1991, vol.1 p.46) mōrā formula

$$(Jathi)(Jathi)(Jathi) \tag{5.3}$$

$$(Jathi)(Karve)(Jathi)(Karve)(Jathi) \tag{5.4}$$

Figure 5.30: Two of Iyer's (2000, p.68, p.79) mōrā formulas that can be accounted for with Nelson's one (5.29). Note Iyer's different terminology; he uses Jathi to mean syllable or group of syllables (which is distinct from words, for which he uses Thatthakaras). He also gives karve two meanings; the first in accordance with use in this study, and the second to mean 'an independent Jathi to seperate two Jathis or groups of Jathis in a mukthayam [mōrā]'.

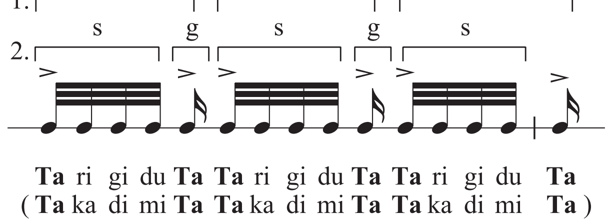


Figure 5.31: Two possible interpretations of a mōrā by Karaikudi R. Mani's tani āvartanam in Nelson (1991, vol.3 p.59). While the first might seem plausible, it is only because the gaps and the concluding beat are identical. See Figure 5.32 for a situation where such an interpretation is made impossible.

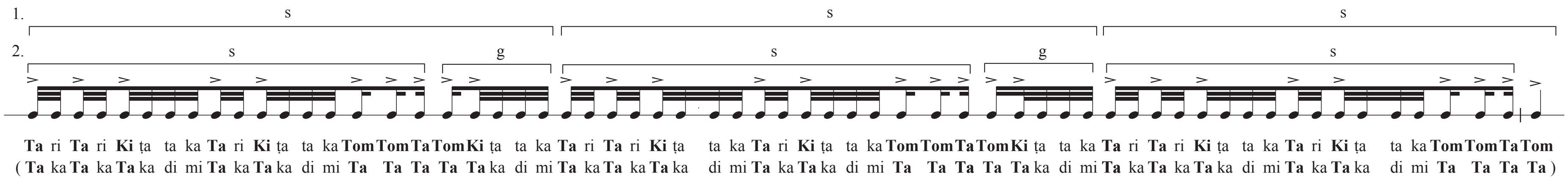


Figure 5.32: A long mōrā from Trichy S. Sankaran's tani āvartanam in Nelson (1991, vol.3 p.164). While the use of only three statements was plausible in Figure 5.31, in this case it is clearly invalid as the statements are not identical. The second grouping conforms to the formulas of Nelson (1991) and Iyer (2000).

The task of generating mōrās is split into a number of methods; `moraStatement, moraGap`, and `moraOffset` for generating their respective parts from a given duration in mātras, gati and karve, `createSimpleMora` for combining the parts into the mōrā structure, `randomMoraValues` for calculating suitable durations for each section of a mōrā from a given total duration in mātras, `randomMora` for generating random mōrās from a given duration in mātras, gati and karve and `moraFrom` for creating mōrās from a given `KonaWord/KonaTime` instance for the statement and a maximum duration in mātras.

`moraStatement, moraGap and moraOffset` work in a similar manner; given a number of pulses, gati and karve they will generate either a single jati `KonaWord` or one or more poly-jati `KonaWords`, filling the given duration. The weights for the methods differ, with gaps given a greater weighting for single jati `KonaWords`, and only offsets given the possibility of being entirely rests.

Given a duration in mātras, gati and karve, `randomMora` will `randomMoraValues` to calculate the durations for each section of the mōrā and `createSimpleMora` with `moraStatement, moraGap`, and `moraOffset` methods to generate a simple mōrā. If the duration is less than seven `randomMora` is called recursively with the duration doubled but the karve halfed; the mōrā has the same duration but double the density. The reason for this is Nelson's (2008, p.23) observation that "if a mora statement is shorter than five pulses, its gap will nearly always be at least two pulses", any duration below seven makes this impossible. This possibility of increased density is also given a 12.5% chance of occurring regardless of the duration. Nelson's (2008, p.23) observation is also taken into account when deciding statement and gap durations; if the total duration is less than 15 the gap minimum is set to two. Once possible gap and statement durations have been calculated and randomly selected, the required offset can be calculated. Finally all three values are turned into `KonaWords` with their respective methods and returned in a `KonaTime`. As mōrās are cadences always resolved with a strong beat (or at least never a rest) the `makePostMora` method was written to ensure that the material that comes after the mōrā does not start with a rest; converting rests if necessary.

The `moraFrom` method allows mōrās to be generated using a given phrase, which could for example be a phrase featured earlier in a composition, a common practice made apparent through analysis of the five tani āvartanams in Nelson (1991, vol.3). This phenomenon is noted by Nelson (1991, vol.1 p.89) in his discussion of phrase suffixes, which "intrdouce rhythmic phrases that always hvae the potential to become formal cadences". The method also has the option of passing in a pre-made gap and/or offset, any section that is not passed in is generated.

Compound Mōrās

A compound mōrā is a mōrā in which the statements themselves are also of the mōrā form (Nelson, 1991, vol.1 p.53) (Figure 5.29), see Figure 5.13 for an example.

The `randomSamaCompoundMora` method was written to generate compound mōrās with a sama (equal) yati (shape); where all three statements are identical (there are other shapes such as gopucca– 'cow's tail' which contracts with increasingly smaller statements and srotovaha– 'river-mouth' the opposite of gopucca). This method uses the `randomMoraValues` method to determine section values, `randomMora` to create the simple mōrā that will comprise the statements of the compound mōrāand `moraFrom` to create the missing parts (offset and gaps if necessary) and build the mōrā structure.

Gati Changes

A change of gati is crucial to the tani āvartanam, and is usually carried out in the middle section; the madhyama kāla (Nelson, 1991, vol.1 p.89-90). Often a whole section of music will be transposed from the original gati to a new one, in a tani āvartanam by Palghat R. Raghu performs a 180 pulse kōrvai (form of composition) in caturaśra, khaṇḍa and tiśra gati (Nelson, 1991, vol.1 p.19, vol.3 p.107-108) see Figure 5.33.

To model this process `wordAtGati` and `phraseAtGati` were written. `wordAtGati` takes an existing `KonaWord` and desired gati and karve, from which it returns a new `KonaWord` equal in jatis but with altered gati and karve. As `phraseAtGati` has to keep relative the karves of multiple `KonaWords` it takes a gati parameter and an expansion parameter instead of karve. A distinction is made between `KonaWords` and `KonaTimes`, the former returns a new `KonaWord` with the input object's karve multiplied by the expansion value, the later results in a recursive call on all contained objects until a `KonaWord` is being dealt with.

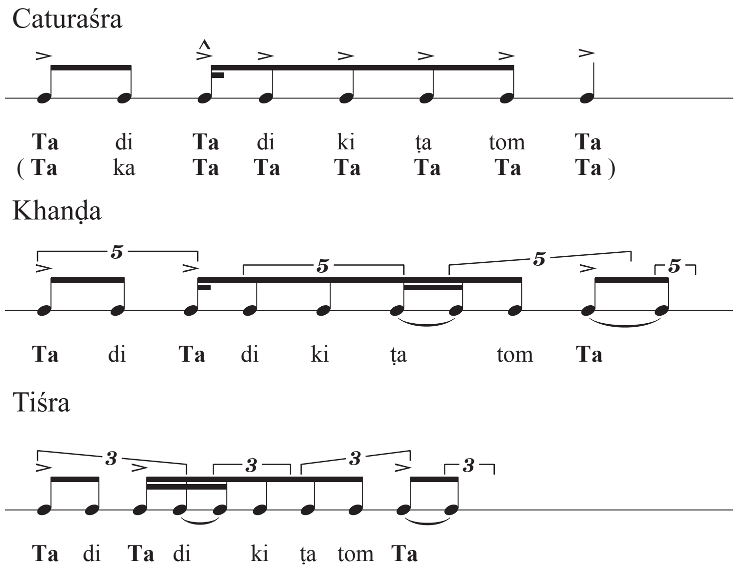


Figure 5.33: An example of gati change. The first phrase from a kōrvai in Palghat R. Raghu's tani āvartanam (Nelson, 1991, vol.3 p.107-108) in caturaśra, khaṇḍa and tiśra gati.

5.3.4 Micro Mutation

A number of methods have been implemented for altering material at the micro level; the jatis of individual `KonaWords`.

Altering Jati Density

A simple method of variation from an existing pattern is to increase the 'density' of one or more of the jatis as shown in Figure 5.34. The `densityJati` method was written to accomplish this, taking as its parameters a `KonaWord`, the index of the jati to be altered and the density multiplier (see Figure 5.35 for example output). Additionally the `randomDensityJati` method was written to automate this process, randomly selecting an index and choosing from acceptable multipliers for the gati (e.g. three is unsuitable for caturaśra gati). As this process appeared frequently during analysis `randomDensityJati` is given a chance of recursion which decreases with each recursive call.

This process is only designed to increase the density of a jati as decreasing the density would return a phrase of greater duration than the input word. A different approach was used to solve this problem (see **Extending Jatis** below).

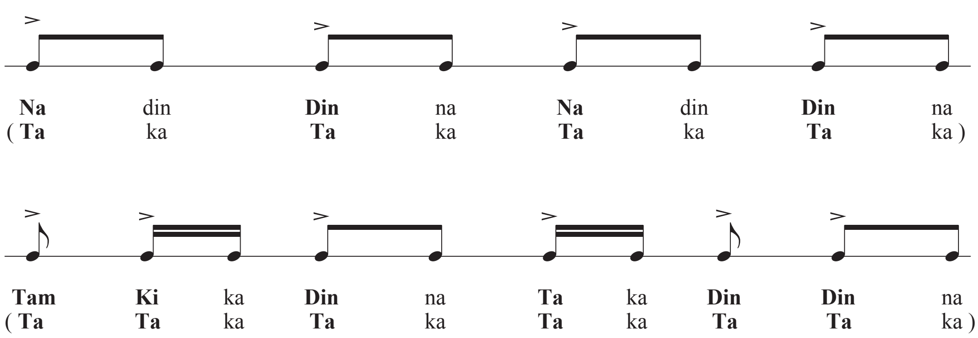


Figure 5.34: The first four beats from cycles two and three of Trichy S. Sankaran's tani āvartanam in Nelson (1991, vol.3 p.162). The first line is a common basic sarvalaghu pattern (Nelson, 1991, vol.1 p.32), the second is a variation easily produced using the `densityJati` method on index one of the first word with a multiplier of two, and on index 0 of the third word with a multiplier of two.

`postWord` method printout:

```
[ Ta      , ka      ]
[ 0.5     , 0.5     ]
[ [ 1, 8 ], [ 1, 8 ] ]
Becomes
[ Ta      , Ta       , ka       ]
[ 0.5     , 0.25     , 0.25     ]
[ [ 1, 8 ], [ 1, 16 ], [ 1, 16 ] ]
```

```
[ Ta       , ki       , tah       ]
[ 0.333    , 0.333    , 0.333     ]
[ [ 1, 12 ], [ 1, 12 ], [ 1, 12 ] ]
Becomes
[ Ta       , Ta       , ka       , di       , mi       , Ta       ]
[ 0.333    , 0.083    , 0.083    , 0.083    , 0.083    , 0.333    ]
[ [ 1, 12 ], [ 1, 48 ], [ 1, 48 ], [ 1, 48 ], [ 1, 48 ], [ 1, 12 ] ]
```

Figure 5.35: Example output from `jatiDensity`. In the first example a two pulse, caturaśra gati, two karve `KonaWord` has been passed in with the index of mutation as one and a density multiplier of two. In the second example a three pulse, tiśra gati, one karve `KonaWord` is passed in with index of one and a multiplier of four.

Extending Jatis

To solve the problem of decreasing the density of a jati in a `KonaWord` (as in Figure 5.36) the `extendJati` method was written. Instead of requiring a multiplier this method takes the number of jatis to extend a given jati by, a check is included to make sure that the extension stays within the duration of the `KonaWord`. The `randomExtendJati` method provides automation of this process.

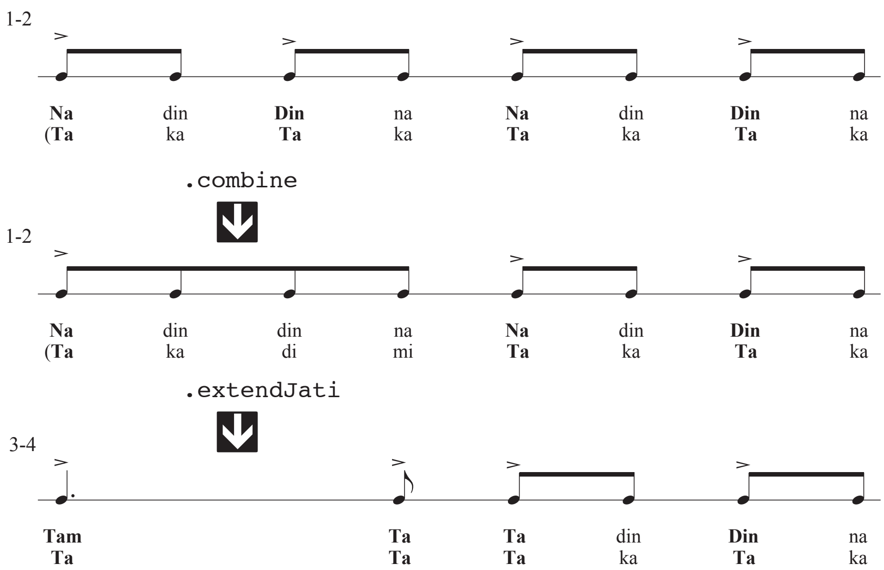


Figure 5.36: An example of jati contraction. The top line shows akṣaras one and two of the second cycle of Trichy S. Sankaran's tani āvartanam (in Nelson (1991, vol.3 p.162)). The second line shows this phrase altered with the `combine` method (5.3.1 **Combining**). The third line is the result of `extendJati` used on index 0 of the first word with an extension of two.

Muting Jatis

As an alternative method of achieving similar results as those in Figure 5.36 the `muteJati` method was written. Given a `KonaWord` and an index this method will turn an audible jati into a rest. Despite the difference in representation (see Figures 5.37 and 5.38) the results are identical. In Karṇāṭak music the term kārvai, which is almost analogous to the Western ‘rest’ (Nelson, 1991, vol.2 p.162) yet differs as “unlike our rest, [kārvai] *includes* the syllable immediately preceding it, and in fact may be said to flow from it as an extension”. This is probably one of the reasons that Nelson chose not to use staff notation.

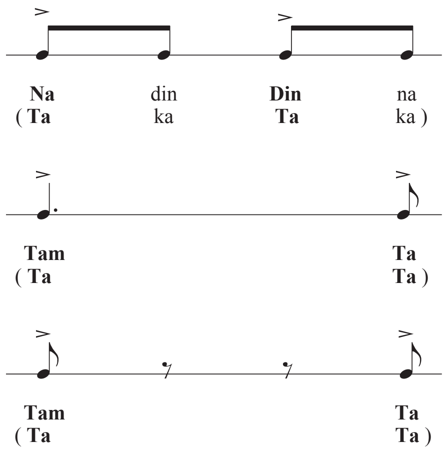


Figure 5.37: Notation and `KonaWord` `postWord` output of a phrase (line one) from Trichy S. Sankaran’s tani āvartanam (in Nelson (1991, vol.3 p.162)) altered by combination and extension (line two) and muting (line three).

```
[ Ta     , ka     , Ta     , ka      ]
[ 0.5    , 0.5    , 0.5    , 0.5     ]
[ [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ] ]

[ Ta     , Ta      ]
[ 1.5    , 0.5     ]
[ [ 3, 8 ], [ 1, 8 ] ]

[ Ta     , -      , -      , Ta      ]
[ 0.5    , 0.5    , 0.5    , 0.5     ]
[ [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ] ]
```

Figure 5.38: Output from calling `postWord` on each of the `KonaTime` instances in Figure 5.37.

Partitioning

Just as partitioning can be used to generate new phrases, it can also be used to create variations by partitioning existing `KonaWords`. While many other processes feature a manual and automated method, for this process a single optionally manual method is provided. This is due to the large number of possible partitions for most values, if specific values were desired they could be instantiated by hand. `partitionWord` takes as parameters a `KonaWord`, minimum and maximum part sizes, if no minimum or maximum sizes are provided they are chosen randomly. `partitionWord` will randomly partition and permute the total number of mātras (jatis * karve) of the given `KonaWord`, and return them as new `KonaWords` in a `KonaTime` (Figures 5.39 and 5.40).

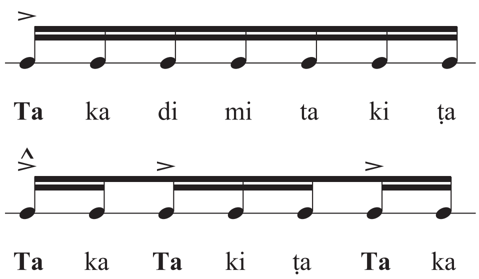


Figure 5.39: A seven jati, caturaśra gati, one karve `KonaWord` (Line one) partitioned into a `KonaTime` with three `KonaWords`, all caturaśra gati, one karve with 2 + 3 + 2 jatis respectively.

```
[Ta , ka , di , mi , ta , ki , tah ]
[0.25 , 0.25 , 0.25 , 0.25 , 0.25 , 0.25 , 0.25 ]
[[1, 16], [1, 16], [1, 16], [1, 16], [1, 16], [1, 16], [1, 16]]

[Ta , ka , Ta , ki , tah , Ta , ka ]
[0.25 , 0.25 , 0.25 , 0.25 , 0.25 , 0.25 , 0.25 ]
[[1, 16], [1, 16], [1, 16], [1, 16], [1, 16], [1, 16], [1, 16]]
```

Figure 5.40: Output from calling `postWord` on each of the `KonaTime` instances in Figure 5.39.

5.3.5 Macro Mutation

For altering large groups of material additional methods have been implemented that make use of the automated micro mutation methods (5.3.4).

Altering Phrase Density

Just as the density of individual jatis can be altered with `densityJati`, a method has been written to alter the density of `KonaWords` or collections of them. `atDensity` takes as its parameters either a `KonaWord` or `KonaTime` as well as a density multiplier. So that relative densities can be maintained within a phrase, `atDensity` works recursively until it deals with individual `KonaWords`. If the altered number of jatis is an integer and can fit into a single word (i.e. is under nine) a new `KonaWord` is returned to the stack parent. If the new jati number is not an integer (e.g. a nine jati word is multiplied by 0.5) a single jati word with a karve equal to the jatis of the input word is returned. If more than one `KonaWord` is required they are returned in a `KonaTime`. An automated version of this method is implemented as `randomAtDensity` which chooses randomly from a list of valid density multipliers for the gati, with a maximum resulting karve of 0.5.

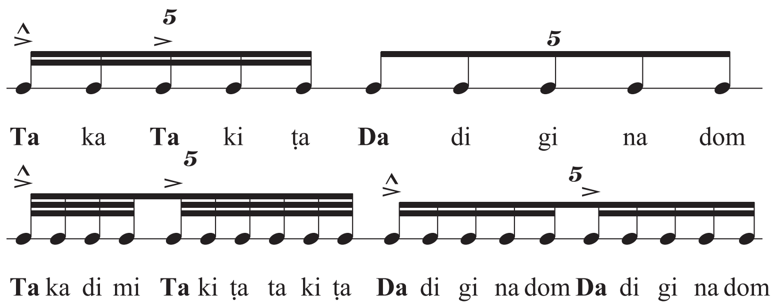


Figure 5.41: Altering phrase density. A khaṇḍa gati phrase (top line) is doubled in density (bottom line) using `atDensity` with a multiplier of two. The phrase consists of a sub-phrase of `KonaWords` TA KA + TA KI ṬA and a `KonaWord` DA DI GI NA DOM. Notice how in the mutation the density is kept relative when doubled.

```
[ Ta       , ka       , Ta       , ki       , tah      , Da       , di       , gi       , na       , dom       ]
[ 0.2      , 0.2      , 0.2      , 0.2      , 0.2      , 0.4      , 0.4      , 0.4      , 0.4      , 0.4       ]
[ [ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 10 ], [ 1, 10 ], [ 1, 10 ], [ 1, 10 ], [ 1, 10 ] ]
```

```
[ Ta       , ka       , di       , mi       , Ta       , ki       , tah      , ta       , ki       , tah      ,
[ 0.1      , 0.1      , 0.1      , 0.1      , 0.1      , 0.1      , 0.1      , 0.1      , 0.1      , 0.1      ,
[ [ 1, 40 ], [ 1, 40 ], [ 1, 40 ], [ 1, 40 ], [ 1, 40 ], [ 1, 40 ], [ 1, 40 ], [ 1, 40 ], [ 1, 40 ], [ 1, 40 ],

Da       , di       , gi       , na       , dom      , Da       , di       , gi       , na       , dom       ]
0.2      , 0.2      , 0.2      , 0.2      , 0.2      , 0.2      , 0.2      , 0.2      , 0.2      , 0.2       ]
[ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 20 ], [ 1, 20 ] ]
```

Figure 5.42: The SuperCollider `postWord` output for Figure 5.41 (the mutated phrase is spread across two lines).

Phrase Permutation

In order to create permutations of a phrase the method `permutePhrase` was written with an optional parameter for permutation number, which will be random if not set. See Figures 5.43 and 5.44 for an example.

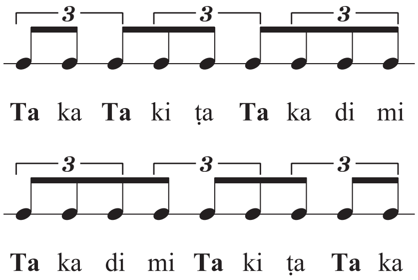


Figure 5.43: An example of phrase permutation. A tiśra gati phrase consisting of Ta ka (2) + Ta ki ṭa (3) + Ta ka di mi (3) (top line) is permuted into `Ta ka di mi` (4) + Ta ki ṭa (3) + Ta ka (2) (bottom line).

```
[Ta    ,ka    ,Ta    ,ki    ,tah   ,Ta    ,ka    ,di    ,mi    ]
[0.333 ,0.333 ,0.333 ,0.333 ,0.333 ,0.333 ,0.333 ,0.333 ,0.333 ]
[[1,12],[1,12],[1,12],[1,12],[1,12],[1,12],[1,12],[1,12],[1,12]]

[ Ta    ,ka    ,di    ,mi     ,Ta    ,ki    ,tah   ,Ta    ,ka    ]
[ 0.333,0.333 ,0.333,0.333  ,0.333 ,0.333 ,0.333 ,0.333 ,0.333 ]
[[1,12],[1,12],[1,12],[1,12],[1,12],[1,12],[1,12],[1,12],[1,12]]
```

Figure 5.44: Output from calling `postWord` on each of the `KonaTime` instances in Figure 5.43.

Phrase Mutation

To automate mutation of a phrase with multiple processes the `mutatePhrase` method was written, which makes use of most of the previously mentioned automated mutation methods (`randomAtDensity, randomExtendJati, randomMuteJati, randomDensityJati, partitionWord`). `mutatePhrase` loops through the items in a phrase calling itself recursively for `KonaTime` instances and for `KonaWords` making probabilistic decisions as to whether or not mutation should occur and in what form. Finally the phrase is either returned or recursively mutated according to a probability, which if successful is halved for the recursive call. See Figure 5.45 for examples.

Word Mutation

The more complex generation processes (5.3.2) generally require partitioning, permutation and mutation to move from an phrase archetype to something more interesting. The methods `partitionWord` (which includes permutation) and `mutatePhrase` have been combined to achieve this in the `randomPartitionMutate` method. An optional probability (defaults to 0.5) can be passed to determine whether partitioning and permutation should take place, mutation is guaranteed (Figure 5.46 for examples).

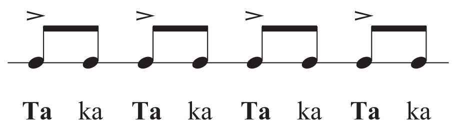
5.4 Tāḷa Generator

A rudimentary tāḷa generator was built separately to the main system, which constructs a routine based on a set of parameters for the tāḷa. The tāḷa generator could account for the seven elementary tāḷas (sūḷādi sapta tāḷas) as well as khaṇḍa and miśra cāpu tāḷas. Each of the sūḷādi sapta tāḷas is comprised of a combination of kriyās; laghu– a clap followed by a variable number of finger counts (notated as `In` or a vertical line followed by a number, where n is the total number of beats e.g. I4 is a clap followed by three finger counts), drutam– a clap followed by a wave of the hand (notated as `O` or a full circle), and anudrutam– a single clap (notated as `U` or a half circle) (Pesch and Sundaresan, 1996, p.15-16).

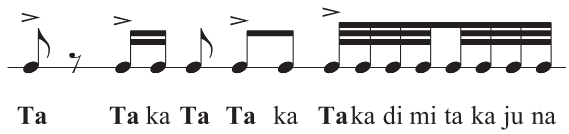
The tāḷa generator has a function for each of these tāḷas with parameters for the variables such as laghu duration and gati, which generates the tāḷas technical name as well as a routine which sends messages to *Processing Homepage* (2009) to display images of the relevant hand signals.

5.5 Critique of Design and Implementation

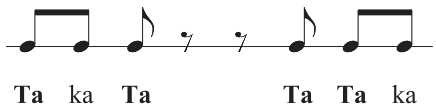
The specifications laid out (in Chapter 4) have been well met by the system; representation classes have been built that can successfully represent the *rhythmic* aspects of Karṇāṭak percussion playing, (albeit with some with some small compromises, (see 5.2.1 and 5.2.2) and methods for generating rhythmic phrases and cadences as well as methods to mutate them have been implemented. As well as the automated methods that form the 'Computational Modelling of Musical Style' (Pearce et al., 2002) side




```
[ Ta      , ka      , Ta      , ka      , Ta      , ka      , Ta      , ka      ]
[ 0.5     , 0.5     , 0.5     , 0.5     , 0.5     , 0.5     , 0.5     , 0.5     ]
[ [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ] ]
```




```
[ Ta  ,-   ,Ta     ,ka     , Ta  ,Ta   ,ka   ,Ta    ,ka    ,di    ,mi    ,ta    ,ka    ,ju    ,na    ]
[ 0.5 ,0.5 ,0.25   ,0.25   , 0.5 ,0.5  ,0.5  ,0.125 ,0.125 ,0.125 ,0.125 ,0.125 ,0.125 ,0.125 ,0.125 ]
[[1,8],[1,8],[1,16],[1,16],[1,8],[1,8],[1,8],[1,32],[1,32],[1,32],[1,32],[1,32],[1,32],[1,32],[1,32]]
```



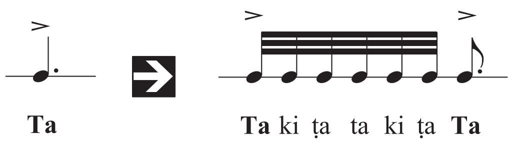

```
[ Ta      , ka      , Ta      , -       , -       , Ta      , Ta      , ka      ]
[ 0.5     , 0.5     , 0.5     , 0.5     , 0.5     , 0.5     , 0.5     , 0.5     ]
[ [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ], [ 1, 8 ] ]
```

Figure 5.45: Examples of `mutatePhrase` in notation and `postWord` output. The first line is a basic sarvalaghu pattern (Nelson, 1991, vol.1 p.32), the second and third are variations created with `mutatePhrase`.

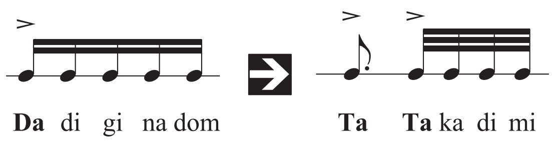
of this project, most elements of the system can be used manually or with semi-automation, making it useful to musicologists, Karṇāṭak Teachers and musicians, and even-non Karṇāṭak musicians.

The parameters for the representation classes as well as generation and mutation methods all use values and terms that should be familiar to those versed in Karṇāṭak music and at least be understandable to those who are not. While some of the design decisions, especially regarding the boundaries `KonaWord` class might not suit all potential users, the areas in which there have been compromises make no impingement on the representation of the *rhythmic* aspects of Karṇāṭak music.

The system being divided into classes, eases the use or development of individual elements. While some loose coupling of classes has been used to ease development, this is easily reversible.




```
[ Ta       ]      [ Ta        , ki        , tah       , ta        , ki        , tah       , Ta        ]
[ 1.5      ]  ->  [ 0.125     , 0.125     , 0.125     , 0.125     , 0.125     , 0.125     , 0.75      ]
[ [ 3, 8 ] ]      [ [ 1, 32 ], [ 1, 32 ], [ 1, 32 ], [ 1, 32 ], [ 1, 32 ], [ 1, 32 ], [ 3, 16 ] ]
```




```
[ Da     , di      , gi      , na      , dom      ]      [ Ta     , Ta      , ka      , di      , mi       ]
[ 0.25   , 0.25    , 0.25    , 0.25    , 0.25     ]  ->  [ 0.75   , 0.125   , 0.125   , 0.125   , 0.125    ]
[ [1, 16], [1, 16], [1, 16], [1, 16], [1, 16] ]      [ [3, 16], [1, 32], [1, 32], [1, 32], [1, 32] ]
```

Figure 5.46: Two examples of the `randomPartitionMutate` method in notation and `postWord` output. Notice the equal handling of the two forms of `KonaWord` used in primitive generation; a single jati word with high karve and a poly-jati word with single karve.

Chapter 6

Evaluation

Throughout this project the development of the system has benefited from an ongoing evaluation process; Karṇāṭak rhythm has been practised and studied via instructional materials (Vinayakram and McLaughlin, 2007; Lockett, 2008; Vinayakram, 2007; Nelson, 2008; Prakash, 2009) and traditional mṛdaṅgam lessons under R. N. Prakash (see Section 6.2.3). As concepts became clearer through practise, their counterpart computer implementations were been refined. The theoretical basis for these implementations was then discussed with Prakash, forming an iterative, agile development. The lessons were typically with a group of musicians, with emphasis on interaction. The material included traditional sarvalaghu patterns and mōrās, as well methods on adapting material for different tāḷas. Prakash also shared the names of influential musicians to listen to, as well as the details of a concert in London featuring the legendary mṛdaṅgam maestro Guru Kaaraikkudi Mani, the equally acclaimed vocalist Shri T. M. Krishna, violinist H. N. Bhaskar and kanjeera player N. Amrit. The concert was a rare opportunity to witness highest level Karṇāṭak music first hand.

6.1 Evaluation Method

As a means of evaluating the output of the system a discrimination test was set up in which participants were asked to distinguish musical examples used as the basis for the system and examples from the system itself. This method was largely inspired by one part of Pearce and Wiggins's (2001) paper *Towards a Framework for the Evaluation of Machine Compositions*. While similar to a Turing test (Turing (1950) in; Pearce and Wiggins (2001)), Pearce and Wiggins make the distinction that while Turing test is designed to test for machine-thinking via interaction, the form of test being used here "simply determines the (non-)membership of a machine composition in a set of human composed pieces of music" (Pearce and Wiggins, 2001).

As mentioned in the beginning of this dissertation (see Chapter 1) an abstraction was made from the playing style of any particular instrument, with a focus on rhythmic content. This created a problem during the evaluation; while the examples weren't specified for a particular instrument, playback via MIDI control of a sampled drums was vastly more pleasant to listen to and comprehend than time-stretched playback of recorded koṉakkōl syllables. While the human examples could be replicated stroke for stroke, the computer examples contained no stroke information. While a stroke-management system might have been a useful solution, it was not considered core to the project and so was not developed. A possible solution would have been to use random, or semi-random strokes for both human and computer examples, as it would would remove the advantage the human examples had of expert stroke choices. The problem with such an approach is that it compromises the human examples with computer interference, which would likely disturb expert listeners. To solve this problem the original strokes were used for human examples and the strokes were set by hand for the computer examples, in the same way Harold Cohen would hand colour drawings by early versions of AARON (Boden, 2004, p.314-315). It was made clear at the beginning of the questionnaire that while examples may have been composed by either a computer or human, all examples were *performed* by a computer.

The experts and lay listeners were both given the same set of examples, which were all generated single run of their respective methods to avoid cherry-picking.

- 18 examples of basic, undeveloped vilamba sarvalaghu phrases, with computer examples generated by `vSarvaPhraseAuto` using the same tāḷa, laya and gati settings as the human examples.
- 14 examples of developments of basic phrases. The purpose was to evaluate the ability to create variations from a given phrase. The computer developments all resulted from using `mutatePhrase` on the same original phrases used in the human examples of development.
- Four examples of longer developments. A basic phrase was played followed by a development featuring two variations and a suffix. The computer examples used `mutatePhrase` and `addSuffix` on the original human basic phrase.
- 16 examples of mōrās, including compound mōrās. The duration in mātras, the gati, karve and laya of human mōrās were passed into `randomMora` and `randomSamaCompoundMora` to create the computer examples.
- Three basic structures. These were short compositions composed entirely by the computer, all following the same structure. A method `basicStructure` was written into `KonaGenerator` to make it easier to produce multiple examples. The structure of the examples was as follows:

  Cycle 1 Basic phrase, developments using `mutatePhrase` with a suffix.

  Cycle 2 Developments generated using `randomDensityJati` on an already developed phrase from Cycle 1.

  Cycle 3 Half cycle of phrase developments followed by a half cycle mōrā.

  Cycle 4 Phrase developments using `mutatePhrase`.

  Cycles 5 and 6 A compound mōrā.

  Cycles 7 to n-1 The whole composition again in a new gati.

  Cycle n A short concluding mōrā to fill any remaining beats.

Some basic information was requested of participants; age, whether or not they were a musician, details of musical training (if any) and exposure to/knowledge of Indian music and Karṇāṭak music on a scale from 1 (none) to 10 (expert). For each example listeners were asked to decide whether they thought the example was written by a human or a computer, as well being asked for comments on any example that felt particularly obvious either way. Additional comments were requested at the end of each set of examples and at the end of the questionnaire, where participants were asked to rate how difficult they found the discrimination of examples on a scale from 1 (very easy) to 10 (very difficult). The majority of participants completed the test using an online questionnaire, a small number of evaluations were carried out in person including the evaluation by R. N. Prakash (6.2.3).

6.2 Expert Listeners

6.2.1 Ludwig Pesch

Biography *From Ludwig Pesch's homepage Pesch (2009)*

> Ludwig Pesch taught music and performed with improvisation ensembles while studying music and musicology at the Government Music College and University Freiburg (Germany). For 15 years, he was a pupil of the late Ramachandra Shastry, musical heir to the great traditions of Tyagaraja and Sarabha Sastrigal.
>
> A scholar under the Indo-German Cultural Exchange Programme and the German Academic Exchange Service, he completed his Diploma Course (five years) in Carnatic Music (First Class) at Kalakshetra. He is the co-founder of a major music documentation centre and archive in Chennai, Sampradaya.
>
> For many years, he performed alongside his guru. While receiving advanced training as a Carnatic flautist in the Post-Diploma Course for two years at Kalakshetra, he also had numerous opportunities to give solo-concerts of his own. Since then, he has performed all over South India (Tamil Nadu, Kerala, Andhra Pradesh and Karnataka).

Pesch is also the author of *The Oxford Illustrated Companion to South Indian Classical Music* (Pesch, 1999), Eloquent Percussion (Pesch and Sundaresan, 1996) as well as a number of other books on Karṇāṭak music.

In the questionnaire, Pesch ranked his knowledge/exposure of both Indian and Karṇāṭak music as 10/10 (expert) and gave as his musical background "Western music college and musicology; South Indian (Carnatic) music diploma in Madras".

Undeveloped Phrases Of the 18 undeveloped phrases, nine (50%) were correctly identified, of the incorrectly identified examples four computer examples were thought to be human. Pesch provided many constructive criticisms, of correctly identified computer generated example he said "Toy music feel in spite of underlying complexities", "Lack of sense of direction" and "Too plain, without apparent direction". Correctly identified human examples were noted for their "sense of anticipation engendered here", "building up [of] interest, differentiation" and for being a "grand statement (spacing and colouring) as those loved by seniour drummers". There were a number of positive comments on computer examples misidentified as being human such as "Sense of deliberate friction to heighten interest" and "solicits attention, as would be the case in a drum solo performance as it 'takes off'."

Of the examples as a whole Pesch said "Remarkable bits of music; some of the guesses pertain to the computer's 'training' or reference and assignment; and naturally from the brevity of samples (out of context)".

Phrase Developments Out of 14 examples of phrase development eight (57.14%) were correctly identified, of which four computer examples were mistaken to be human. There were a number of positive remarks attributed to computer examples (albeit always when mistaken to be human) including "organic extension", "pleasing" and "sense of fingering".

Longer Developments Three of the four (75%) longer developments were correctly identified, with the only mistaken example being a computer development thought to be human and described as "More captivating than others". Pesch commented that he found these examples hard to distinguish.

Mōrās Out of 16 examples 10 (62.5%) were correctly identified, of those misidentified four were computer generated. The computer examples were correctly identified for being "too smooth", having a "contived feel" and being "mechanical" and "monotonous". The only misidentified computer generated example given a comment was said to have a "lack of beauty", but was still considered human. Of the set of examples Pesch noted that "The cerebral nature of drumming makes the lines blur", a statement backed up by his correct classification of a human example despite it noting it would be "technically challenging".

Basic Structures All but one of the three (66.66%) basic structures were correctly identified, with one (correctly identified) computer example noted as being "less appealing than the others".

Summary Pesch rated the difficulty of distinguishing examples a 9 out of 10, commenting that "In the competitive world of Carnatic drumming, calculated patterns, often of great complexity and in need of virtuosity, have come to stay." This comment as well as the previous note that "The cerebral nature of drumming makes the lines blur" (paragraph on **Mōrās**) seems indicative of an opinion that certain aspects of Karṇāṭak drumming are becoming more obviously calculable, while other aspects such as sarvalaghu can still only be described in more abstract terms (**Undeveloped Phrases** paragraph).

The evaluation process as a whole was described as "Well done and challenging, with plenty of room for doubt whether computers can conceive of the same differentiation in drumming as an ambitious drummer's mind, or even more. Musical context therefore matters most; greater complexity makes for greater interchangeability between 'man and machine'."

For a summary of Ludwig's results see Table 6.1.

6.2.2 David Nelson

Biography *See also see 3.2.2.*
From David Nelson's homepage (Nelson, 2009):

> David Nelson has been performing and teaching South Indian drumming since 1975. From his principal teacher, the renowned T. Ranganathan, he learned to accompany a wide range of styles, including Bharata Natyam, South India's classical dance. He has a Ph.D. in Ethnomusicology from Wesleyan University, where he is Artist in Residence in South Indian drumming. He has accompanied well-known artists throughout the United States, Europe, India, and China. He has also written extensively on South Indian drumming, including a major article in the Garland Encyclopaedia of World Music.

In the questionnaire, Nelson gave his knowledge of Indian music as 8/10, and knowledge of Karṇāṭak music as 10/10, musical training was given as "early training in classical and band (brass instruments, piano), Karṇāṭak music since 1970, voice and mṛdaṅgam, several years of jazz drumset".

Undeveloped Phrases For the undeveloped phrases Nelson correctly identified 13/18 (72.2%) examples, of the incorrectly identified examples four were computer examples labelled as human and one was the opposite. Many examples were commented upon, the most frequent being that examples "could be either". For a number of the correctly identified computer examples comments regarding the mechanical execution were given, demonstrating some difficulty in separation of composition from performance. The comments for computer examples incorrectly identified as human included "it's plausible, but could be either", "pattern human, strokes and execution mechanical" and "like something a player might do...", suggesting a lack of strong conviction for the given answer.

When commenting on the examples as a whole Nelson said "They're too short and out of context to be really convincing either way, even the ones that don't necessarily sound like someone might play them might be okay as part of something longer that on the whole is convincing." This comment raises an important issue regarding example durations that will be discussed in the critique of the evaluation method (Section 6.5).

Phrase Developments For the phrase developments Nelson correctly identified 5/14 (35.7%) examples, of the nine incorrectly identified five were computer generated. Fewer comments were given than for the undeveloped phrases; for the misidentified computer examples the comments "sounds pretty typically human", "probably human, but not obvious" and "sounds believable" were given, a misidentified human example was described as "shapeless, bland". The comment given for this section as a whole was "Mostly ambiguous, these could be either".

Longer developments Nelson correctly identified all four longer developments. The two computer examples garnered contrasting responses; "Not convicing, it sounds random" and "Somebody might do this, but I'm sceptical". Both human examples were given the same comment; "Not a style I know, but somebody might play like this", which raises the issue of performance and execution again as they are both examples taken from the first few cycles of a tani āvartanam analysed in Nelson's thesis (Nelson, 1991, vol.3 Trichy S. Sankaran p.162).

Mōrās Nelson correctly identified 9/16 (56.25%) mōrā examples, of the seven incorrectly identified five were computer generated. There were many comments in regard to the execution, including a human example described as "pretty obviously mechanical", while a computer example was commented as "execution sound human". Two computer examples were subject to the extremely critical comment "I *hope* it's a computer, this one's awful." Additionally, one example was noted as being a pattern played by Nelson himself.

Basic Structures All three basic structures were identified as being computer generated, with the following comments "I don't think anybody plays like this", "The beginning doesn't make musical sense to me", "Fingers, phrases, don't make sense to me", with the additional general comment "If a human made these compositions, I really dislike the style". These comments seem to be more focused on musical style than capability, with the compositions compared to a musician with an unfavourable or poorly formed style.

Summary In response to the evaluation as a whole Nelson said "None of this sounded like any playing I recognize, whether computer or not.". When it was noted that many of the human examples were taken from transcriptions in his thesis, he replied that "The material itself may have come from my work, but the strokes and execution were so unfamiliar as to be distracting. I guess I listen to those qualities as much as to the ideas as such. A real musician would play 'in a style', meaning that there would be more internal coherence." Despite complaints about the effects of computer performance, human examples were correctly identified 74.61% of the time on average, perhaps indicating that they were less influential factors than Nelson believed. The area in which the most computer generated examples were incorrectly identified (phrase development) was also the area in which the most human examples were incorrectly identified. Considering the greater success of human identification in other areas of the test, it is likely that this was a particularly difficult process to judge accurately. Nelson gave the overall difficulty of identification a rating of 5/10. It should be noted that Nelson had the advantage of prior exposure to many of the human examples (whether he was aware of it or not) as all ādi tāḷa caturaśra gati examples were from the transcriptions of tani āvartanam on which his thesis was based. For a summary of David's results see Table 6.2.

6.2.3 Sri R. N. Prakash

Biography *From the South Asian Arts website (South Asian Arts UK, 2009):*

> Sri R.N.Prakash is a disciple of Vidvan K.N.Krishnamurthy of Bangalore. His talents have been promptly recongnised in India by his elevation to the prestigious grade 'A' artiste status at a very young age by All India Radio.
>
> Currently he is the resident Mṛdaṅgam teacher at the London School of Carnatic Music based at the London Sivan Kovil in Lewisham. He is a popular Mṛdaṅgam and Ghatam artiste in the UK as well as abroad. Practically every Carnatic concert in London relies on his Ghatam accompaniment. Although his expertise is in Indian classical music, he is versatile, adventurous and open minded. His fusion work with western pop and jazz groups, especially with 'Massive Attack', illustrates his interest in building musical bridges to other cultures.

Undeveloped Phrases Prakash correctly identified 9/18 undeveloped phrases, of which eight were computer examples thought to be human. During this section Prakash expressed an interest in the variety of performers the human examples were drawn from, noting after the evaluation that certain musical settings were unusual for certain performers. For example, the fact that the miśra gati examples were by Vikku Vinayakram was a surprise, as this is apparently a rare (and thus, unfamiliar) example of his playing. As a result, there was an increased degree of openness to unfamiliar playing.

Phrase Developments Four of the 14 phrase developments were correctly identified, with six computer examples being mistaken as human. During the discussion of the system that took place after the evaluation, when discussing the generation of variations, Prakash said that it was fingerings that can be 'wrong' but not variations as long as they add up properly as "variation is just mathematics".

Longer Developments All four longer developments were thought to be computer generated, despite the two human examples.

Mōrās Of the 16 examples of mōrās six were correctly identified, of which five computer examples were misidentified as human.

Basic Structures Of the three, all computer generated compositions, two were considered plausible as human.

Summary In discussion of the evaluation process as a whole Prakash commented that "it can be hard to recognise [examples as] human or computer as [they are] played on the computer... what makes Indian music is a character, you can't create the character." As with Nelson (6.2.2) it seems that execution plays too important a role to make the identification trivial. On the topic of computer performance Prakash commented "for Karṇāṭak music you need a lot of feel... [the computer's lack of feel] is the reason that computer music has not gotten into Indian or Karṇāṭak music." On a scale from 1-10 Prakash gave the difficulty of identification a 6. For a summary of Prakash's results see Table 6.3.

System Discussion Uniquely for a professional listener, the evaluation process was carried out in person, allowing the various components of the system to be demonstrated and discussed. In discussion of the applications of such a system Prakash was very positive; while convinced that computers can not yet play a role in Karṇāṭak performance, he saw great potential in the system for composition and teaching. On the topic of teaching noted that students could be provided with a good reference of a pattern or idea, as well as saving time spent construction permutations and combinations. The automation of permutation and combination was also thought useful for composition, saying that "for people who already have the knowledge of these rhythms, it can take [composition] to a different level", but also warning against treating it as a shortcut; that "you should have very good ears to appreciate the [generated] rhythms". It was also noted that the combination of very precise playback with `phraseAtGati` had particularly useful applications when dealing with translating material to difficult gatis such as khaṇḍa (seven sub-divisions per beat) and sankīrṇa (nine sub-divisions per beat), or between gatis that are very close in timing (e.g. six to seven).

Of the system as a whole Prakash said "I think it has been very good project, where computer music can give a lot more phrases and gives a very strong rhythm sense. It gives an easy way to work out permutations and combinations."

6.3 Lay Listeners

In addition to the expert listeners the evaluation was carried out by 17 other participants, of which 15 rated their exposure to/knowledge of Karṇāṭak music as below 5/10 ('some'), with an average of 2.13/10. The two participants with exposure to/knowledge of Karṇāṭak music higher than 5/10 gave themselves 7/10 and 8/10 but scored no higher than the average lay participant. Of these participants, 11 were musicians with a variety of backgrounds, including "8 years trumpet, 6 years in a progressive rock band, 1 in a metal band, Numerous years Jazz Big Band, Grade 5 Theory distinction, AS Level B Music. Studying for a Masters in Digital Music & Sound Art 5 years mixing in a digital environment.", "Doctoral composer, jazz/pop/classical musician", "Death metal drummer for 8 years" and "Self-taught. No particular style".

In the following discussion of results the answers always refer to the average identification.

Undeveloped Phrases Only three of the 18 examples of undeveloped phrases were misidentified, with only one computer examples mistaken as human. Many participants commented that the computer performance and lack of knowledge made the task very difficult, and that intuition was heavily relied upon.

Phrase Developments Of the 14 phrase developments, only four were correctly identified. Of those misidentified, five computer examples were thought to be human and were given comments such as "Form seemed typical of human. Interesting Variation" and "Felt more creative". One computer example was regarded as human by 14/17 participants (as well as 2/3 experts), with the comments "Too groovy for a computer, I felt very definite about this choice" and "Has human flare". The correctly identified human examples were often said to be too systematic and mechanical.

One participant (a non-musician with an 8/10 rating for exposure to Karnatak music) noted that these types of variations should not be complicated, making programming of suitable methods reasonably achievable and thus harder to distinguish results.

Correctly identified human examples	Incorrectly identified human examples	Correctly identified computer examples	Incorrectly identified computer examples
Undeveloped Phrases			
5/9 **55.56%**	4/9 **44.44%**	4/9 **44.44%**	5/9 **55.56%**
Phrase Development			
5/7 **71.43%**	2/7 **28.57%**	3/7 **42.86%**	4/7 **57.14%**
Long Developments			
2/2 **100%**	0/2 **0%**	1/2 **50%**	1/2 **50%**
Mōrās			
3/6 **50%**	3/6 **50%**	7/10 **70%**	3/10 **30%**
Basic Structures			
0/0 **n/a%**	0/0 **n/a%**	2/3 **66.6%**	1/3 **33.34%**

Table 6.1: A summary of Ludwig Pesch's evaluation

Correctly identified human examples	Incorrectly identified human examples	Correctly identified computer examples	Incorrectly identified computer examples
Undeveloped Phrases			
8/9 **88.89%**	1/9 **11.11%**	8/9 **88.89%**	1/9 **11.11%**
Phrase Development			
3/7 **42.86%**	4/7 **57.14%**	2/7 **28.57%**	5/7 **71.43%**
Long Developments			
2/2 **100%**	0/2 **0%**	2/2 **100%**	0/2 **0%**
Mōrās			
4/6 **66.67%**	2/6 **33.33%**	5/10 **50%**	5/10 **50%**
Basic Structures			
0/0 **n/a%**	0/0 **n/a%**	3/3 **100%**	0/3 **0%**

Table 6.2: A summary of David Nelson's evaluation

Correctly identified human examples	Incorrectly identified human examples	Correctly identified computer examples	Incorrectly identified computer examples
Undeveloped Phrases			
8/9 **88.89%**	1/9 **11.11%**	1/9 **11.11%**	8/9 **88.89%**
Phrase Development			
3/7 **42.86%**	4/7 **57.14%**	1/7 **14.29%**	6/7 **85.71%**
Long Developments			
0/2 **0%**	2/2 **100%**	2/2 **100%**	0/2 **0%**
Mōrās			
1/6 **16.67%**	5/6 **83.33%**	5/10 **50%**	5/10 **50%**
Basic Structures			
0/0 **n/a%**	0/0 **n/a%**	1/3 **33.34%**	2/3 **66.66%**

Table 6.3: A summary of R. N. Prakash's evaluation

Correctly identified human examples	Incorrectly identified human examples	Correctly identified computer examples	Incorrectly identified computer examples
Undeveloped Phrases			
7/9 **77.78%**	2/9 **22.22%**	8/9 **88.89%**	1/9 **11.11%**
Phrase Development			
3/7 **42.86%**	4/7 **57.14%**	1/7 **14.29%**	6/7 **85.71%**
Long Developments			
1/2 **50%**	1/2 **50%**	1/2 **50%**	1/2 **50%**
Mōrās			
4/6 **66.67%**	2/6 **33.33%**	6/10 **60%**	4/10 **40%**
Basic Structures			
0/0 **n/a%**	0/0 **n/a%**	1/3 **33.34%**	2/3 **66.66%**

Table 6.4: A summary of the lay listener's evaluations.

Longer Developments The longer developments had a fairly close balance of votes for both possibilities, resulting in half of both the computer and human examples being misidentified. One Computer example was thought to be "Too groovy and cheeky for a computer generation" as well as having "a very human feel to the pattern", while another was said to be "very obviously computer".

Mōrās Six of the 16 mōrās were incorrectly identified, four of which were computer examples with all but one being a close result. The inherent complexity of the mōrās was found by many to make discrimination difficult.

Basic Structures Of the three basic structures only one was correctly identified as being computer generated, and even then only by a 40/60 divide. The change of gati in the correctly identified computer example was said to be too awkward and sparse to be human. The computer example most voted as human was said to have "the true structure of human composition".

Summary The average rating of difficulty of discrimination was 8.71 with the lowest being 7/10 and five participants giving 10/10. The reasons for difficulty that cropped up numerous times were the computer playback, lack of prior knowledge and the number of examples. For the summary results of the lay listeners see Table 6.4.

6.4 Summary

Judging the overall success of the generational aspects of the system is difficult; the results of the experts were varied and often contradictory (they all agreed on only 20/55 examples) and in the lay listener evaluation many examples were identified with only a one or two vote difference. Subjectivity was abundant with participants praising and harshly criticising both computer and human examples, as was prejudice, with positive comments only given to examples thought to be human.

Strengths

Looking at the percentage of misidentified computer examples it is clear that the system performs best at creating variations on material, as noted by Prakash (6.2.3) variations are a computationally inclined area of Karṇāṭak music. Of the 55 examples there were three computer examples (all short phrase developments) that the experts and lay listeners mistook to be human, as well as two basic phrases and one mōrā that the experts (but not the lay listeners) also mistaken.

Of the three experts only Prakash was able to have the system demonstrated in person to provide feedback on the representations and compositional tools. Thoughtful of the potential applications of the system he was very enthusiastic, citing a number of potential uses (see **System Discussion** in 6.2.3).

Weaknesses

The areas in which the system performed poorly were the longer developments and basic structures, with worse than hoped for results in undeveloped phrase generation. A potential explanation for some of these weaknesses is that while the system was based primarily on ādi tāḷa caturaśra gati examples, a large proportion of the evaluation examples used different tāḷas and gatis.

6.5 Critique of Evaluation Method

A number of problems arose during the evaluation process which hindered the quality of feedback.

6.5.1 Computer Performance

The complaint from participants that appeared most frequently was the difficulty of discrimination due to computer performance. The problem with performance strokes was discussed as part of the evaluation process (Section 6.1), but in brief was that the system worked only with koṉakkōl syllables, without a stroke management system. The restriction to these syllables (Table 2.2) did not limit the capabilities of this system, when discussed with Prakash he said that all rhythms could be represented by them, and easily interpreted into strokes by a performer. A similar complaint was the rigidity of timing of the performance, an issue that might have been lessened by the use of a MIDI humanise function.

If time had permitted an ideal solution to both problems would have been to adhere to the advice of David Cope:

> If one employs synthetic sounds and inflexible rhythms while replicating music, stylistically valid works may not be recognized. Proper timbral choices, use of live performers and attention to performance practice are highly recommended. (Cope, 1993*a*, p.408)

Yet it is not difficult to imagine this process being criticised because of the performer's contribution to the material.

6.5.2 Example Duration and Context

Another common comment on the difficulty of discrimination was that many of the examples were short and out of context. This was a real problem for many of the processes being evaluated; isolating the individual processes while giving the participants enough material to listen to. For these short examples they were often played twice in succession, and then repeated after a short break. Still this did not solve the problem of context, a number of participants commented that they found it easier to discriminate with the longer examples which had more of a sense of context.

6.5.3 Participant Uncertainty

Participants both expert and lay expressed uncertainty with a number of examples. One lay participant commented that were there an "I don't know" option or similar, they would have used it for a great deal of examples. An alternative to this would have been a rating of confidence of decision for each choice, although this would have been time consuming.

6.5.4 Pride and Prejudice: The problem with discrimination

For the purposes of evaluating Computational Modelling of Musical Styles the discrimination test (Pearce and Wiggins, 2001) is, in theory, well suited. However, in practice there were a number of significant issues.

There was a sense that participants felt that they, as opposed to the system, were being tested, one musician participant commented "[the test] was very hard...[it] can hurt people's pride". This issue is more significant for expert participants as decisions may be thought of as a test of their expertise; about which any uncertainty might raise insecurities. The underlying reason for this issue seems to be prejudices in regards to the ability of computer and the value of human creativity. Indeed the notion that something of artistic value might be created by a computer is deemed impossible by some (Boden, 2004, p.321). There were a large number of comments for examples thought to be human that were expressions of value, while examples thought to be generated by the computer were often met with derogatory remarks. Instead it may have been wise to have concealed the involvement of a computer at all, either asking for just qualitative feedback or pretending the computer examples were by a human student.

Chapter 7

Conclusion

The system achieved the objective of representing Karṇāṭak rhythms well as well as interesting results from the discrimination test of the generative features. The representations and methods for manipulating them were felt to be useful for a number of applications by a high level musician of the style. A number of computer generated examples were of high enough quality to fool three experts and a majority of lay listeners into believing they were of human origin, and a number of other mistaken examples were positively commented upon.

7.1 Future Improvements

Representation The representation of the 'words' of Karṇāṭak rhythm (Section 5.2) required a number of design choices that resulted in a compromise in quality of representation of some words; see Section 5.2.2 and Figure 5.12 for the discussion of words such as Ta lan - gu and Da - di - gi na dom. There were also well-known (Brown, 1965) difficulties in distinguishing words *made by concatenation* from *concatenated words* (see 5.2.1). A future revision of this project could combine the `KonaWord` and `KonaTime` classes into a single `KonaPhrase` class, which could have user specified syllables/strokes, their durations and accents. This would provide a more flexible representation, able to account for the variety found in solkaṭṭu/koṉakkōl, but would require greater effort on the part of the user.

As mentioned under Musicologists in the User Requirements (Section 4.1.2) the representations could be mapped to an input device for faster transcription by musicologists. This could be developed further by mapping stroke representations to an instrument equipped with sensors for automatic transcription from a musician's performance, as done by Kapur (2008, ch.5) with his Hindustani E-Tabla, E-Dolak and E-Sitar.

Generation The generation and mutation methods in this version of the system only represent the tip of the iceberg of Karṇāṭak rhythm, and are only representative of the musicians and contexts studied. There are countless other structures in Karṇāṭak rhythm that time has not permitted consideration. Drumming is also an essential feature of Karṇāṭak dances such as the bharata nāṭya (Pesch and Sundaresan, 1996, Glossary), which includes another set of traditions completely untouched in this project.

Selection Currently much of the selection process is random (e.g. partition choice), or weighted random. Where non-weighted randomness is used, heuristics could be introduced (e.g. symmetry for partition choice), and where weighted randomness is used the weights could be data-mined from musical examples, as in the current version they have generally been set loosely and tweaked intuitionally.

Mutation There is much room for improvement of the automated mutation methods. While the results of the processes may closely match those found in human examples, the automation of them has been achieved by trial and error. A better solution would be to implement an analysis system that would use input material to create statistical and contextual models of the use of the processes.

A particular mutation method that could be further developed is `phraseAtGati`. At the moment it returns a brute conversion from one gati to another. In real situations a conversion such as this might require some alteration to the output to make it fit with the tāḷa (e.g. a 16 pulse phrase in will fill four beats in caturaśra gati, but only 3.2 in khaṇḍa). Automation of this alteration process would make it easy to try out compositions in different gatis and tāḷas, which may be especially useful to the Karṇāṭak percussion student who is not yet fully capable of calculating this transitions.

Playback The focus on rhythms as opposed to particular playing strokes means that when output is with Karṇāṭak instruments, an indication of whether a stroke is open (resonant) or closed (non-resonant) is lacking. For the Karṇāṭak musician this is not likely to be a problem, as solkaṭṭu to stroke interpretation is common (Brown, 1965, 135). For computer playback however this does pose a problem, as was encountered with listener evaluation of the system (Chapter 6).

A future solution would be an interpreter module that would translate assign suitable strokes to the `KonaWord` jatis.

Technical The ZS2 integer partitioning algorithm (Zoghbi and Stojmenović, 1998) implemented in SCLang for generation rhythm generation (Section 5.3.1) can take considerable amounts of time for larger numbers (partitioning 100 with a minimum part size of two and a maximum of nine averaged `54.92 seconds` on an an Apple Dual 1.8GHz G5 with 4 GB of ram), which is reduced by reading from arrays stored as files (the same partition averaged `5.73 seconds` with this approach).

Implementation in a language such as `C` could decrease this time, Zoghbi and Stojmenović (1998) reported their `C` implementation on a SUN machine (circa 1998) averaged `10.3 seconds` for a partition of 75 (Zoghbi and Stojmenović, 1998), the same procedure with the SCLang implementation took `8.37 seconds` with a relatively contemporary machine.

7.2 Alternative Methodologies

The system used a completely symbolic, knowledge based approach which Papadopoulos and Wiggins (1999) outline as having the following difficulties:

- - *Knowledge elicitation is difficult and time consuming, especially in subjective domains such as music.*
  - This has certainly been the case with this project, with large amounts of time spent modelling a relatively small number of processes.
- - *Since [knowledge based systems] do what we program them to do they depend on the ability of the "expert", who in many cases is not the same as the programmer, to clarify concepts, or even find a flexible representation.*
  - For the purposes of this project the expert can be considered to be the analytical and instructional materials with the author as an intermediary. In the instance of original transcription and analysis work. This creates a chain of dependence that includes the quality of the musician's performance and teaching, the musicologist's analysis and the author's understanding.
- - *[Knowledge based systems] become too complicated if we try to add all the "exceptions to the rule" and their preconditions, something necessary in this domain.*
  - For the sake of accomplishing a variety of tasks at a reasonable level, not all exceptions have been taken into account, limiting the output of the system.

Alternatives to this symbolic, knowledge based approach include using grammars, evolutionary models or machine learning techniques (Section 2.2.1). Bernard Bel's success with grammars to model tabla playing (Section 3.1.1), (Bel, 2006) indicates that this path is likely to be fruitful for generating Karṇāṭak rhythms. The mutational or 'germinal' (Brown, 1965) nature of Karṇāṭak music makes an evolutionary approach appealingly suitable. A machine learning approach has the advantage of eliminating the dependence on the analyst(s) ability, with the system instead dealing with real Karṇāṭak rhythms directly.

7.3 Contribution

As previously mentioned (Chapter 1) this project is the first to focus on computer representation and generation of Karṇāṭak rhythms. It is hoped that this preliminary work should serve as a useful starting point for future work, as well as highlighting some of the areas of Karṇāṭak rhythm in which computational processes would be most welcomed and beneficial. Hopefully this work will play a role in raising the profile of Karṇāṭak music in the field of computer music, the results of which may in turn feed back into the Karṇāṭak music community.

Bibliography

Abran, A., Moore, J. W., Bourque, P. and Dupuis, R., eds (2004), *Guide to the Software Engineering Body of Knowledge*, IEEE Computer Society.

Ames, C. and Domino, M. (1992), Cybernetic composer: An overview, *in* M. Balaban, K. Ebcoglu and O. Laske, eds, 'Understanding Music with AI', AAAI Press, pp. 186–205.

Avid (2008), 'http://www.sibelius.com/'.

Ayyangar, R. R. (1972), *History of South Indian (Carnatic) Music : From Vedic Times to the Present*, R. Rangaramanuja Ayyangar, Madras (Chennai).

BCS (2008*a*), 'Code of Conduct'.

BCS (2008*b*), 'Code of Good Practice'.

Bel, B. (1992*a*), "Modelling Improvisatory and Compositional Processes", *Languages of Design, Formalisms for Word, Image and Sound* , Vol. 1, pp. 11–26.

Bel, B. (1992*b*), Symbolic and Sonic Representations of Sound-Object Structures, *in* M. Balaban, K. Ebcioglu and O. Laske, eds, 'Understanding Music with AI : Perspectives on Music Cognition', MIT Press, pp. 64–109.

Bel, B. (1998), "Migrating Musical Concepts - An Overview of the Bol Processor", *Computer Musical Journal* , Vol. 22, pp. 56–64.

Bel, B. (2006), The Bol Processor Project: Musicological and Technical Issues, *in* 'Virtual Gamelan Graz: Rules - Grammars - Modelling Symposium', Institute of Ethnomusicology, Graz, Austria.

Bel, B. (2009), 'Bol Processor Homepage'.
URL: *http://bolprocessor.sourceforge.net/*

Bel, B. and Kippen, J. (1992), Bol Processor Grammars, *in* M. Balaban, K. Ebcioglu and O. Laske, eds, 'Understanding Music with AI : Perspectives on Music Cognition', MIT Press, pp. 366–400.

Biles, J. A. (1994), 'GenJam: A Genetic Algorithm for Generating Jazz Solos'.

Biles, J., Anderson, P. G. and Loggi, L. W. (1996), Neural Network Fitness Functions for a Musical IGA, Technical report, Rochester Institute of Technology.

Boden, M. A. (2004), *The Creative Mind*, second edn, Routledge, Abingdon.

Brown, R. (1965), The Mṛdaṅga: A Study of Drumming in South India, PhD thesis, Department of Music, University of California.

Carabott, A. (2009), Bernard Bel's Bol Processor: An Authoritative and Critical Study. Submitted for the Generative Creativity course, as part of the Music Informatics (BA) undergraduate degree at the University of Sussex.

Carl, D., Julian, A., Charles, W., Richard, C. and Brian, H. (2008), 'Harmony'.

Collins, N. (2008), 'Perception of Rhythm Lecture Slides'.

Cope, D. (1991), *Computers and Musical Style*, Oxford University Press, Oxford.

Cope, D. (1993*a*), A Computer Model of Music Composition, *in* S. M. Schwanauer and D. A. Levitt, eds, 'Machine Models of Music', MIT Press, Cambridge, Massachusetts.

Cope, D. (1993*b*), Panel Discussion, *in* 'Proceedings of the International Computer Music Conference'.

Cope, D. (2005), *Computer Models of Musical Creativity*, The MIT Press. 1213294.

Ebcioglu, K. (1988), "An Expert System for Harmonizing Four-part Chorals", *Computer Musical Journal* , Vol. 12, pp. 43–51.

Hild, H., Feulner, J. and Menzel, D. (1992), "HARMONET: a neural net for harmonising chorales in the style of J.S. Bach", *Advances in Neural Information Processing* , Vol. 4, pp. 267–274.

Hulzen, R. v. (2002), Improvisation and its Guiding Principles in Percussion Playing in South Indian Classical Music, PhD thesis.

Iyer, S. R. (2000), *Sangeetha Akshara Hridaya (A New Approach to Tāḷa Calculations)*, Gaana Rasika Mandali.

Kapur, A. (2008), Digitizing North Indian Music: Preservation and Extension using Multimodal SensorSystems, Machine Learning and Robotics, PhD thesis, University of Victoria.

Kippen, J. and Bel, B. (1989), "The Identification and Modelling of a Percussion 'Language', and the Emergence of Musical Concepts in a Machine-Learning Experimental Set-Up", *Computers and the Humanities* , Vol. 23, pp. 199–214.

Kippen, J. and Bel, B. (1992), Modelling Music With Grammars: Formal Language Representation In The Bol Processor, *in* A. Marsden and A. Pople, eds, 'Computer Representations and Models in Music', Academic Press, London, pp. 207–38.

Krishnaswamy, A. (2003*a*), Application of Pitch Tracking to South Indian Classical Music, *in* 'IEEE ICASSP'.

Krishnaswamy, A. (2003*b*), On the Twelve Basic Intervals in South Indian Classical Music, *in* 'Audio Engineering Society'.

Krishnaswamy, A. (2003*c*), Pitch Measurements Versus Perception of South Indian Classical Music, *in* 'SMAC'.

Krishnaswamy, A. (2004*a*), Inflexions and Microtonality in South Indian Classical Music, *in* 'FRSM'.

Krishnaswamy, A. (2004*b*), Melodic Atoms For Transcribing Carnatic Music, *in* 'ISMIR'.

Krishnaswamy, A. (2004*c*), Multi-Dimensional Musical Atoms in South Indian Classical Music, *in* 'ICMPC'.

Krishnaswamy, A. (2004*d*), Results in Music Cognition and Perception and Their Application to Indian Classical Music, *in* 'FRSM'.

Krishnaswamy, A. (2004*e*), Towards Modeling, Computer Analysis and Synthesis of Indian Ragams, *in* 'FRSM'.

Lockett, P. (2008), *Indian Rhythms For Drumset*, Hudson Music.

London, J. (2004), *Hearing in time: Psychological aspects of musical meter*, Oxford University Press, USA.

Magnusson, T. (2009), 'Clapping SynthDef', SC-Users Mailing List Contribution.
URL: *http://www.cogs.susx.ac.uk/users/thm21/*

Miller, G. (1956), "The magical number seven, plus or minus two", *Psychological review* , Vol. 63, pp. 81–97.

Miranda, E. (2001), *Composing music with computers*, Focal Press.

Mitchell, T. (1997), *Machine Learning*, McGraw Hill.

Nelson, D. (1991), Mṛdaṅgam Mind: The Tani Āvartanam in Karṇāṭak Music, PhD thesis.

Nelson, D. (2009), 'David Nelson Homepage'.
URL: *http://dpnelson.web.wesleyan.edu/index.html*

Nelson, D. P. (2008), *Solkattu Manual : An Introduction to the Rhythmic Language of South Indian Music*, Wesleyan University Press, Middletown, Connecticut.

Office of Public Sector Information (1988), 'Copyright, Designs and Patents Act 1988', Public Act.

Palnath, K. (2009), 'Kadrigo Palnath Homepage'.
URL: *http://www.kadrigopalnath.com/*

Papadopoulos, G. and Wiggins, G. (1999), AI methods for algorithmic composition: a survey, a critical view and future prospects, *in* 'AISB Symposium on Musical Creativity', pp. 110–117.

Pearce, M., Meredith, D. and Wiggins, G. (2002), "Motivations and methodologies for automation of the compositional process", *Musicæ Scientiæ* , Vol. 2, pp. 200–2.

Pearce, M. and Wiggins, G. (2001), 'Towards a framework for the evaluation of machine compositions'. 22–32.

Pesch, L. (1999), *The Illustrated Companion to South Indian Classical Music*, Oxford University Press, New Dehli, India.

Pesch, L. (2009), 'Profile: Ludwig Pesch'.
URL: *http://home.planet.nl/ pesch082/Ludwig/English/index.html*

Pesch, L. and Sundaresan, T. (1996), *Eloquent Percussion: A Guide to South Indian Rhythm*, ekaóègrata, Amsterdam.

Prakash, R. N. (2009), 'South Indian Rhythm Fest', DVD.

Prasanna (2003), 'Ragamorphism', DVD.

Processing Homepage (2009).
URL: *http://www.processing.org/*

Raynor, W. (2000), *The International Dictionary of Artificial Intelligence*, Lessons Professional Publishing.

Roads, C. (1996), *The Computer Music Tutorial*, MIT Press.

Sankaran, T. (1977), *The Art of Drumming: South Indian mrdangam*, Private Edition, Toronto.

Shrinivas, U. (2007), 'Samjanitha', Audio CD.

South Asian Arts UK (2009), 'Sri R. N. Prakash - Artist Profile'.
URL: *http://www.saa-uk.org.uk/index.php?id=7914*

Steedman, M. (1984), "A generative grammar for jazz chord sequences", *Music Perception* , Vol. 2, pp. 52–77.

Subramanian, M. (1999), "Synthesizing Carnatic Music with a Computer", *Sangeet Natak - Journal of Sangeeet Natak Akademi* , Vol. 133-134, pp. 16–24.

Subramanian, M. (2002), Analysis of Gamakams of Carnatic Music using the Computer, *in* 'Sangeet Natak - Journal of Sangeeet Natak Akademi', Vol. 37, pp. 26–47.

Subramanian, M. (2008), 'Carnatic Music Software - Rasika & Gaayaka'.
URL: *http://carnatic2000.tripod.com/*

SuperCollider Homepage (2009).
URL: *http://supercollider.sourceforge.net/*

Swar Systems (2009).
URL: *http://www.swarsystems.com*

Todd, P. M. (1989), "A Connectionist Approach to Algorithmic Composition", *Computer Musical Journal* , Vol. 13, pp. 27–43.

Turing, A. (1950), "Computing machinery and intelligence", *Mind* , Vol. 59, pp. 433–460.

Vinayakram, S. (2004), 'Soukha', CD.

Vinayakram, S. G. and McLaughlin, J. (2007), 'The Gateway to Rhythm'.

Vinayakram, T. H. V. (2007), 'The Language and Technique of South Indian Percussion'.

Viswanathan, T. and Allen, M. H. (2004), *Music in South India*, Global Music Series, Oxford University Press, New York.

Zoghbi, A. and Stojmenović, I. (1998), "Fast algorithms for genegrating integer partitions", *International Journal of Computer Mathematics* , Vol. 70, Taylor & Francis, pp. 319–332.

Appendix A

Glossary

- Ādi Tāḷa: An eight beat tāḷa, the most predominant tāḷa in Karṇāṭak music.
- Akṣara: Individual beat or count (a component of tāḷa).
- Aṅga: "Limb"; a group of akṣaras, component of the tāḷa.
- Arudi: A rhythmic cadence or ending phrase.
- Bāṇī: A musical style or family tradition.
- Bhajan: Devotional Song.
- Caturaśra: A member of the five families of rhythm; four or "Four-sided".
- Druta Kāla: The third degree of speed. Sometimes used to refer to the third section of the tani āvartanam.
- Gamakas: Ornamentation.
- Gati: The beat subdivision, 'gait'.
- Ghatam: Clay pot percussion instrument. The main supporting percussion instrument of Karṇāṭak music.
- Jati: A solkaṭṭu/koṉakkōl syllable.
- Jāti: "Variety"; (a) social grouping based on birth, (b) the five important numerical varieties of Karṇāṭak rhythm.
- Jāvaḷi: A genre of dance music.
- Kaccēri: The Karṇāṭak concert.
- Kanjira: Small lizardskin frame drum with metal jingles attached to a wooden shell.
- Karṇāṭak (often Carnatic) Music: The classical music system of South India.
- Khanḍa: A member of the five families of rhythm; five or "broken".
- Kriti: Three part compositional, central to the Karṇāṭak concert.
- Kaṇakku: Calculation. Tension creating rhythmic figures, contrasts Sarvalaghu.
- Koṉakkōl: Solkaṭṭu performed in a concert setting.
- Kōrvai: A complex rhythmic design, ending with a mōrā.
- Koraippu: A section of the tani āvartanam for two or more percussionists in which progressively shorter groups of phrases are traded between musicians.

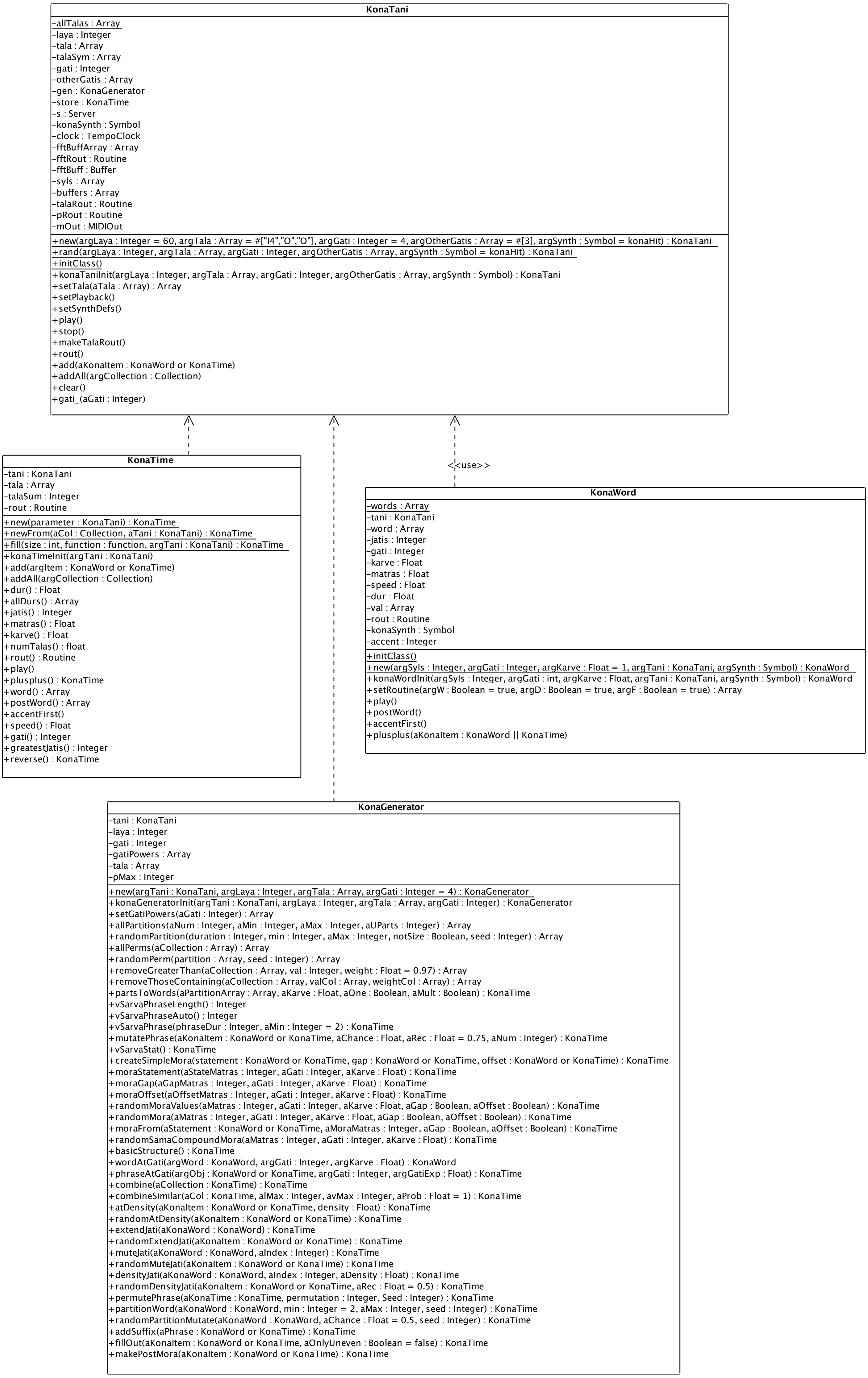
- Madhyama Kāla: Second degree of speed. Sometimes used to as the name of the middle section of the tani āvartanam.
- Mṛdaṅgam: A two headed drum, the primary percussion instrument in Karṇāṭak music.
- Miśra: A member of the five families of rhythm; Seven or "Mixed".
- Mōrā: A rhythmic ending or cadential figure.
- Morsing: Jaw harp idiophone.
- Padam: A dance music genre.
- Periya Mōrā: The penultimate mōrā composition in a tani āvartanam.
- Rāga: A recognizable and unique melodic entity.
- Sankīrṇa: A member of the five families of rhythm; nine or "All mixed up".
- Sarvalaghu: Rhythm patterns that carry the flow of musical time.
- Solkaṭṭu: The South Indian system of spoken syllables and tāḷa hand gestures.
- Svara: A musical note with pitch.
- Svara Kalpana: Improvised singing or playing of svaras.
- Ta Din Gi Na Tom: A concluding section within a piece.
- Tāḷa: The cyclical meter in Karṇāṭak music.
- Tāna Varṇam: Musical form which gives a summary of rāga features.
- Tani Āvartanam: The percussion solo in a Karṇāṭak music concert.
- Thatthakaras: Jatis grouped into a word.
- Tillāna: A dance music genre,.
- Tīrmānam: A set composition for classical dance accompanied by solkaṭṭu.
- Tiśra: A member of the five families of rhythm; three or "Three-sided".
- Vilamba Kāla: The first degree of speed. Sometimes used as the name of the first section of the tani āvartanam.

(Ayyangar, 1972; Nelson, 2008; Pesch and Sundaresan, 1996; Pesch, 1999; Viswanathan and Allen, 2004; Iyer, 2000).

Appendix B

System Diagram

A class diagram for the system can be found on the next page.


KonaTani
-allTalas : Array
-laya : Integer
-tala : Array
-talaSym : Array
-gati : Integer
-otherGatis : Array
-gen : KonaGenerator
-store : KonaTime
-s : Server
-konaSynth : Symbol
-clock : TempoClock
-fftBuffArray : Array
-fftRout : Routine
-fftBuff : Buffer
-syls : Array
-buffers : Array
-talaRout : Routine
-pRout : Routine
-mOut : MIDIOut
+new(argLaya : Integer = 60, argTala : Array = #["I4","O","O"], argGati : Integer = 4, argOtherGatis : Array = #[3], argSynth : Symbol = konaHit) : KonaTani
+rand(argLaya : Integer, argTala : Array, argGati : Integer, argOtherGatis : Array, argSynth : Symbol = konaHit) : KonaTani
+initClass()
+konaTaniInit(argLaya : Integer, argTala : Array, argGati : Integer, argOtherGatis : Array, argSynth : Symbol) : KonaTani
+setTala(aTala : Array) : Array
+setPlayback()
+setSynthDefs()
+play()
+stop()
+makeTalaRout()
+rout()
+add(aKonaItem : KonaWord or KonaTime)
+addAll(argCollection : Collection)
+clear()
+gati_(aGati : Integer)
KonaTime
-tani : KonaTani
-tala : Array
-talaSum : Integer
-rout : Routine
+new(parameter : KonaTani) : KonaTime
+newFrom(aCol : Collection, aTani : KonaTani) : KonaTime
+fill(size : int, function : function, argTani : KonaTani) : KonaTime
+konaTimeInit(argTani : KonaTani)
+add(argItem : KonaWord or KonaTime)
+addAll(argCollection : Collection)
+dur() : Float
+allDurs() : Array
+jatis() : Integer
+matras() : Float
+karve() : Float
+numTalas() : float
+rout() : Routine
+play()
+plusplus() : KonaTime
+word() : Array
+postWord() : Array
+accentFirst()
+speed() : Float
+gati() : Integer
+greatestJatis() : Integer
+reverse() : KonaTime
<<use>>
KonaWord
-words : Array
-tani : KonaTani
-word : Array
-jatis : Integer
-gati : Integer
-karve : Float
-matras : Float
-speed : Float
-dur : Float
-val : Array
-rout : Routine
-konaSynth : Symbol
-accent : Integer
+initClass()
+new(argSyls : Integer, argGati : Integer, argKarve : Float = 1, argTani : KonaTani, argSynth : Symbol) : KonaWord
+konaWordInit(argSyls : Integer, argGati : int, argKarve : Float, argTani : KonaTani, argSynth : Symbol) : KonaWord
+setRoutine(argW : Boolean = true, argD : Boolean = true, argF : Boolean = true) : Array
+play()
+postWord()
+accentFirst()
+plusplus(aKonaItem : KonaWord || KonaTime)
KonaGenerator
-tani : KonaTani
-laya : Integer
-gati : Integer
-gatiPowers : Array
-tala : Array
-pMax : Integer
+new(argTani : KonaTani, argLaya : Integer, argTala : Array, argGati : Integer = 4) : KonaGenerator
+konaGeneratorInit(argTani : KonaTani, argLaya : Integer, argTala : Array, argGati : Integer) : KonaGenerator
+setGatiPowers(aGati : Integer) : Array
+allPartitions(aNum : Integer, aMin : Integer, aMax : Integer, aUParts : Integer) : Array
+randomPartition(duration : Integer, min : Integer, aMax : Integer, notSize : Boolean, seed : Integer) : Array
+allPerms(aCollection : Array) : Array
+randomPerm(partition : Array, seed : Integer) : Array
+removeGreaterThan(aCollection : Array, val : Integer, weight : Float = 0.97) : Array
+removeThoseContaining(aCollection : Array, valCol : Array, weightCol : Array) : Array
+partsToWords(aPartitionArray : Array, aKarve : Float, aOne : Boolean, aMult : Boolean) : KonaTime
+vSarvaPhraseLength() : Integer
+vSarvaPhraseAuto() : Integer
+vSarvaPhrase(phraseDur : Integer, aMin : Integer = 2) : KonaTime
+mutatePhrase(aKonaItem : KonaWord or KonaTime, aChance : Float, aRec : Float = 0.75, aNum : Integer) : KonaTime
+vSarvaStat() : KonaTime
+createSimpleMora(statement : KonaWord or KonaTime, gap : KonaWord or KonaTime, offset : KonaWord or KonaTime) : KonaTime
+moraStatement(aStateMatras : Integer, aGati : Integer, aKarve : Float) : KonaTime
+moraGap(aGapMatras : Integer, aGati : Integer, aKarve : Float) : KonaTime
+moraOffset(aOffsetMatras : Integer, aGati : Integer, aKarve : Float) : KonaTime
+randomMoraValues(aMatras : Integer, aGati : Integer, aKarve : Float, aGap : Boolean, aOffset : Boolean) : KonaTime
+randomMora(aMatras : Integer, aGati : Integer, aKarve : Float, aGap : Boolean, aOffset : Boolean) : KonaTime
+moraFrom(aStatement : KonaWord or KonaTime, aMoraMatras : Integer, aGap : Boolean, aOffset : Boolean) : KonaTime
+randomSamaCompoundMora(aMatras : Integer, aGati : Integer, aKarve : Float) : KonaTime
+basicStructure() : KonaTime
+wordAtGati(argWord : KonaWord, argGati : Integer, argKarve : Float) : KonaWord
+phraseAtGati(argObj : KonaWord or KonaTime, argGati : Integer, argGatiExp : Float) : KonaTime
+combine(aCollection : KonaTime) : KonaTime
+combineSimilar(aCol : KonaTime, aIMax : Integer, avMax : Integer, aProb : Float = 1) : KonaTime
+atDensity(aKonaItem : KonaWord or KonaTime, density : Float) : KonaTime
+randomAtDensity(aKonaItem : KonaWord or KonaTime) : KonaTime
+extendJati(aKonaWord : KonaWord) : KonaTime
+randomExtendJati(aKonaItem : KonaWord or KonaTime) : KonaTime
+muteJati(aKonaWord : KonaWord, aIndex : Integer) : KonaTime
+randomMuteJati(aKonaItem : KonaWord or KonaTime) : KonaTime
+densityJati(aKonaWord : KonaWord, aIndex : Integer, aDensity : Float) : KonaTime
+randomDensityJati(aKonaItem : KonaWord or KonaTime, aRec : Float = 0.5) : KonaTime
+permutePhrase(aKonaTime : KonaTime, permutation : Integer, Seed : Integer) : KonaTime
+partitionWord(aKonaWord : KonaWord, min : Integer = 2, aMax : Integer, seed : Integer) : KonaTime
+randomPartitionMutate(aKonaWord : KonaWord, aChance : Float = 0.5, seed : Integer) : KonaTime
+addSuffix(aPhrase : KonaWord or KonaTime) : KonaTime
+fillOut(aKonaItem : KonaWord or KonaTime, aOnlyUneven : Boolean = false) : KonaTime
+makePostMora(aKonaItem : KonaWord or KonaTime) : KonaTime

Appendix C

Transcriptions

As part of this project a number of transcriptions of Karṇāṭak percussionists were made. A variety of examples follow.

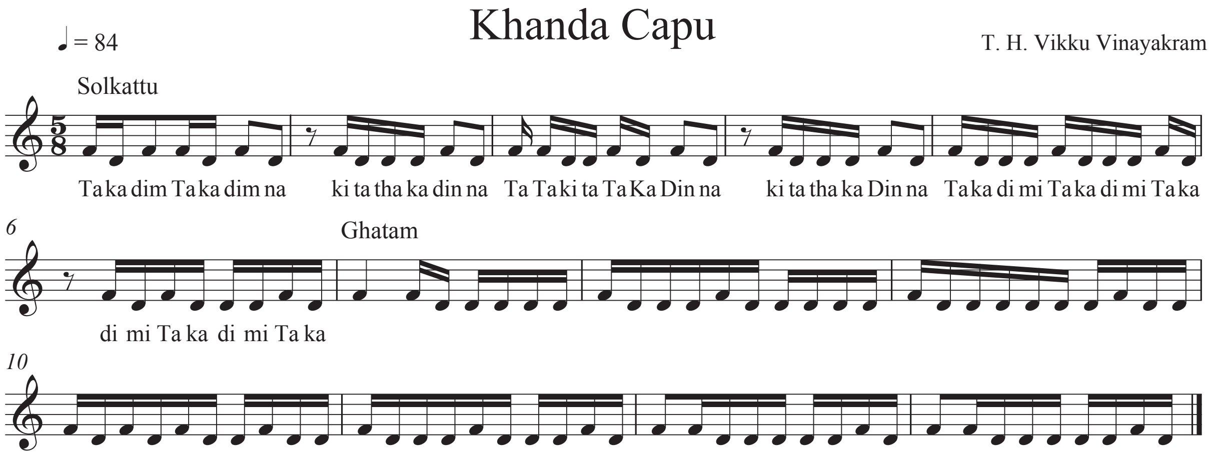


Figure C.1: From Vinayakram (2007, Disc.2 ch.)

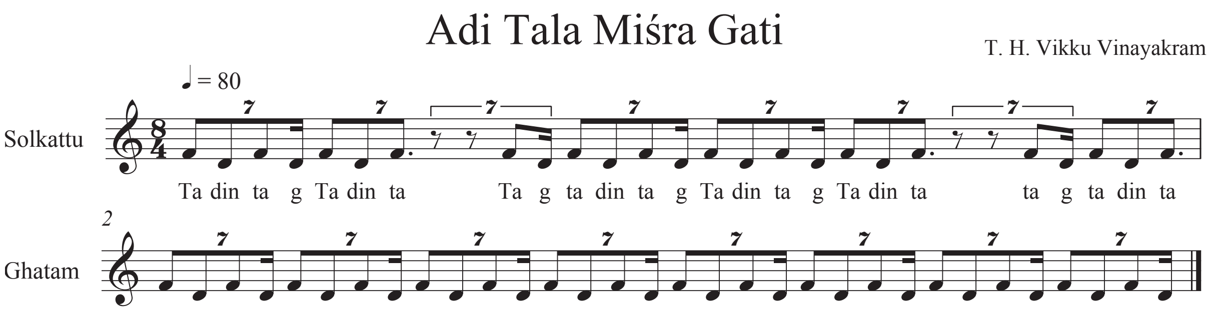



Figure C.2: From Vinayakram (2007, Disc.1 ch.6)

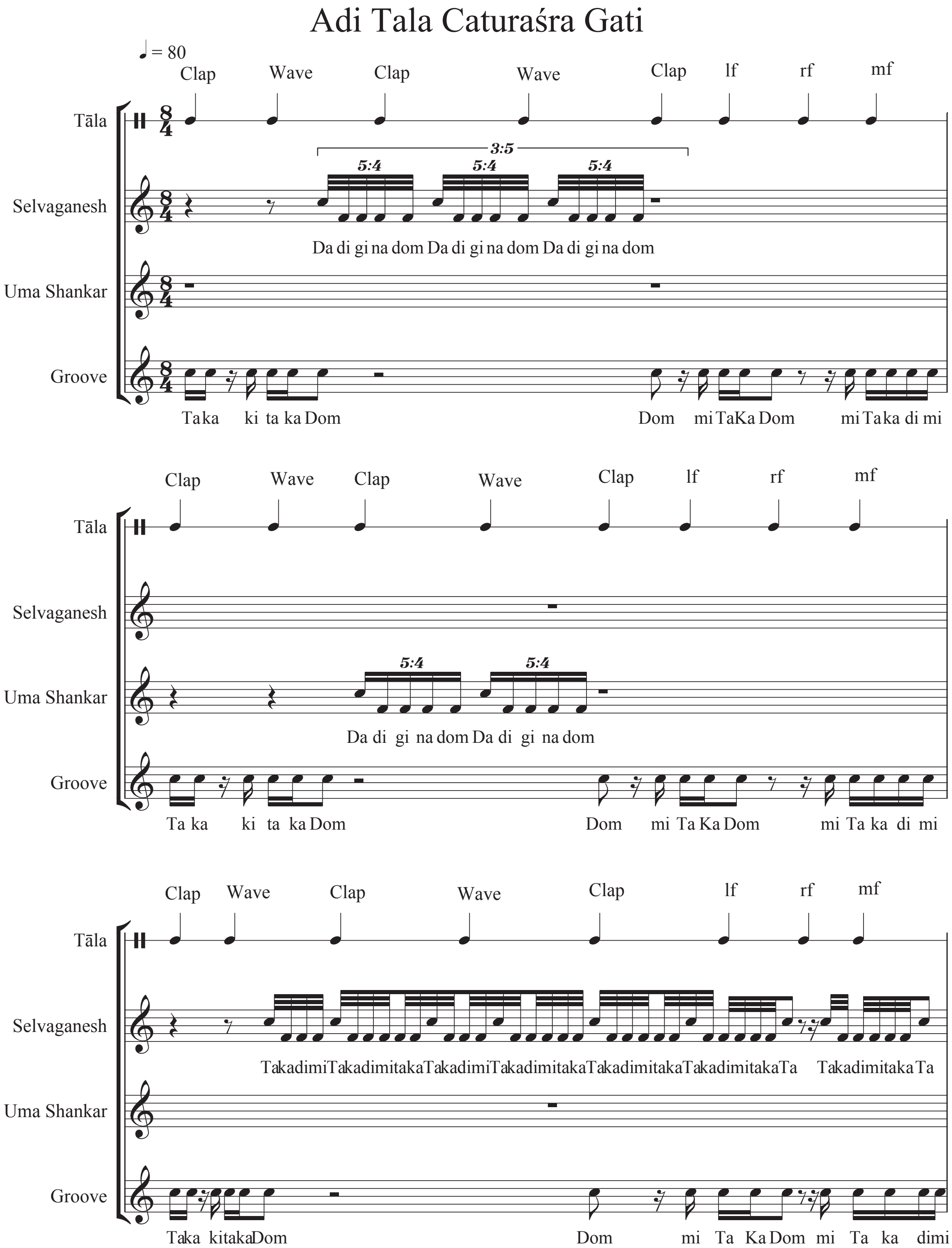



Figure C.3: From Vinayakram (2007, Disc.1 ch.5)

Adi Tala Tisra Gati

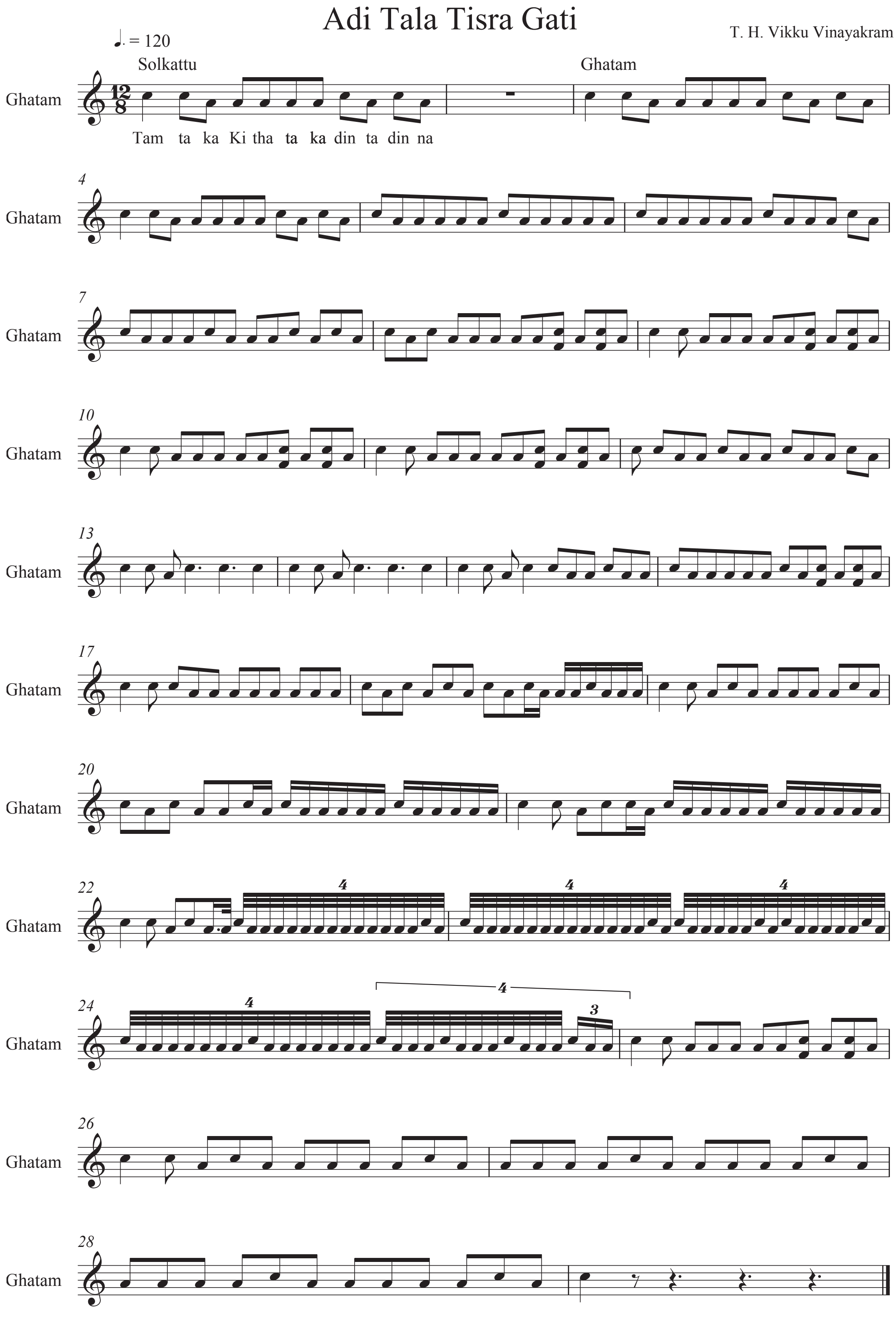


Figure C.4: From Vinayakram (2007, Disc.1 ch.6)

Appendix D

Source Code

The source code begins on the next page.

```
/* ========================================================================== */
/* = KonaWord Class - Represents a a single Konakkol word made up of jatis = */
/* ========================================================================== */

KonaWord {
    classvar words; //Lookup table for syllables to use

    var <tani;      //The Tani that this word belongs to
    var <word;      //The word the instance represents, an Array of symbols
    var <jatis;     //The number of syllables in the word
    var <gati;      //The subdivision of the beat.
    var <karve;     //The number of matras each jati should occupy.
    var <matras;    //The number of pulses/sub-divisions in the word;
    var <speed;     //The duration wait between syllables
    var <dur;       //The duration of the word (the jatis * karve)
    var <val;       //Jatis, dur and word in an array for comparison
    var <rout;      //The routine for playing this phrase
    var konaSynth;  //Symbol of the SynthDef to use
    var <accent;    //Additional accent on the first syllable

    *initClass {
        //Set up lookup table for syllables
        words = Array.newClear(10);
        words[0] = ['-'];
        words[1] = ['Ta'];
        words[2] = ['Ta', 'Ka'];
        words[3] = ['Ta', 'Ki', 'Tah'];
        words[4] = ['Ta', 'Ka', 'Di', 'Mi'];
        words[5] = ['Da', 'Di', 'Gi', 'Na', 'Dom'];
        words[6] = ['Ta', 'Ki', 'Tah', 'Ta', 'Ki', 'Tah'];
        words[7] = ['Ta', 'Ka', 'Di', 'Mi', 'Ta', 'Ki', 'Tah'];
        words[8] = ['Ta', 'Ka', 'Di', 'Mi', 'Ta', 'Ka', 'Ju', 'Na'];
        words[9] = ['Da', 'Di', 'Gi', 'Na', 'Dom', 'Ta', 'Ka', 'Di', 'Mi'];

    }

    *new {|argSyls, argGati, argKarve=1, argTani, argSynth|

        //Check arguments aren't nil
        if( (argSyls==nil) || (argGati == nil),
            {^"arguments not set\n Provide (numSyllables, gati)"}
        );

        //Check specified group is within bounds, the gati is legit
        if( argSyls<=9 && ([4,3,5,7,9].includes(argGati)),
            {^super.new.konaWordInit(argSyls, argGati, argKarve, argTani,
argSynth) },
            {^"Bad Size or Gati"}
        );
    }

    konaWordInit { |argSyls, argGati, argKarve, argTani, argSynth|
        tani = argTani;
        word = words[argSyls];
        jatis = word.size;
        gati = argGati;
        karve = argKarve;
        matras = jatis*karve;
        speed = ((1/gati)*karve);
        dur = speed * jatis;
        val = [jatis, dur, word];
        accent = 0;

        if(tani!=nil) {
            konaSynth = tani.konaSynth
        } {
            if(argSynth!=nil) {
                konaSynth = argSynth
            } {
                konaSynth = \beep
            };
        };

        this.setRoutine;
    }

    //Method to set the routine for this word. Stored in a function for re-use
    setRoutine {
        var ind;
        var rate;
        var amp;

        //MIDI variables
        var bOne;               //MIDI note for first beat (always an open sound)
        var bOthers;            //Chosen MIDI notes for other beats;
        var othersComplete;     //Possible MIDI notes for other beats
        var othersTemp;         //Storage for next 'other beat' MIDI note.
        var note;               //Chosen MIDI pitch.
        var val;                //Temporary storage of chosen MIDI note.
        var vel;                //Chosen velocity for MIDI note.

        switch (konaSynth)
            {nil} {
                rout = Routine {
                    word.size.do {|i|
                        amp = 0.2;
                        case
                        {word[i]=='-'}  {amp=0}
                        {i==0}  {amp=0.4};

                        tani.s.bind{Synth(konaSynth, [\amp,
(amp+(accent/10)).min(1)])};
                        word[i].postln;
                        speed.wait;
                    };
                    yieldAndReset(nil);
                };
            }

            {\konaHit}  {
                rout = Routine {
                    word.size.do { |i|
                        //Index of the syllable to be played
                        ind = tani.syls.indexOf(word[i]);
                        if(i==0) {(amp=0.8+(accent/10)).min(1)} {amp=0.6};
                        if(word[i]!='-') {
                            tani.s.bind {
                                Synth(\konaHit, [\out, 0, \bufnum, tani.fftBuff,
\recBuf, tani.buffers[ind], \rate, ((tani.laya/60)*(0.25/speed)).max(1);]);
                            };
```

```
                        tani.fftRout.next;
                        if(i==0) {
                            word[i].post;
                        } {
                            word[i].asString.toLower.post;
                        };
                        " ".post;
                        speed.post; " ".post;
                        speed.wait;

                    } {
                        word[i].post; " ".post;
                        speed.post; " ".post;
                        speed.wait
                    };
                };
                "".postln;
                yieldAndReset(nil);
            };

        }
        {\MIDITranscribe} {
            rout = Routine {
                word.size.do { |i|
                    if(word[i]!='-') {
                        if(i==0)
                            {note = 48; vel =
((70..100).choose+accent).min(127)}
                            {note = 52; vel = (100+accent).min(127)};
                        tani.mOut.noteOn(0, note, vel);
                            if(i==0) {
                                word[i].post;
                            } {
                                word[i].asString.toLower.post;
                            };
                        " ".post;
                        speed.post; " ".post;
                        speed.wait;
                        tani.mOut.noteOff(0, note, vel);
                    } {
                        word[i].post; " ".post;
                        speed.post; " ".post;
                        speed.wait
                    };
                };
                yieldAndReset(nil);
            };
        }

        //Automated mapping of strokes for Kanjira virtual instrument
        {\MIDIPlay} {
                bOne = [36, 37, 38, 39, 45, 46, 47].choose;
                othersComplete = (48..55);
                othersTemp = Array.newFrom(othersComplete);
                bOthers = Array.newClear(word.size-1);
                (word.size-1).do { |i|
                    val = othersTemp.choose;
                    bOthers[i] = val;
                    othersTemp = Array.newFrom(othersComplete);
                    othersTemp.remove(val);
                };
            rout = Routine {
                word.size.do { |i|
                    if(word[i]!='-') {
                        if(i==0) {
                            note = bOne;
                            vel = (100+accent).min(127);
                            word[i].post; " ".post;

                        } {
                            note = bOthers[i-1];
                            vel = (70..100).choose;
                            word[i].asString.toLower.post; " ".post;

                        };
                        tani.mOut.noteOn(0, note, vel);
                        speed.post; " ".post;
                        speed.wait;
                        tani.mOut.noteOff(0, note, vel);
                    } {
                        word[i].post; " ".post;
                        speed.post; " ".post;
                        speed.wait
                    };
                };
                " ".postln;
                yieldAndReset(nil);
            };


        };

}

play {
    this.rout.play(tani.clock);
}

//Concatonation method to return a new KonaTime with both KonaItems
++ { |aKonaItem|
    var newTime = KonaTime.new();
    newTime.add(this).addAll(aKonaItem);
    ^newTime
}

//Printing method for timing information of the word.
postWord {|argW=true, argD=true, argF=true|
    var words, decimals, fractions;
    var maxItemLength;

    words = Array.newClear(jatis);
    decimals = Array.newClear(jatis);
    fractions = Array.newClear(jatis);

    jatis.do { |i|
        words[i] = word[i].asString;
        decimals[i] = speed.asString[0..4];
        fractions[i] = (speed/4).asFraction(100,false).asString;
    };
```

```
        maxItemLength = [words.maxItem, decimals.maxItem, fractions.maxItem].maxItem.size;

        jatis.do { |i|
            var numSpaces;
            numSpaces = maxItemLength - words[i].size;
            numSpaces.do {
                words[i] = words[i] ++ " ";
            };
            numSpaces = maxItemLength - decimals[i].size;
            numSpaces.do {
                decimals[i] = decimals[i] ++ " ";
            };
            numSpaces = maxItemLength - fractions[i].size;
            numSpaces.do {
                fractions[i] = fractions[i] ++ " ";
            };
        };

        if(argW) {
            words.postln;
        };
        if(argD) {
            decimals.postln;
        };
        if(argF) {
            fractions.postln;
        };

        ^[words, decimals, fractions];

    }

    //For adding additional accents to the first syllable
    accentFirst {
        accent = accent + 10;
    }

}
```

```
/* ======================================================================== */
/* = KonaTime Class - Collection class that represents phrases of music  = */
/* ======================================================================== */

KonaTime : List {

    var <tani;      //  The tani this belongs to
    var <tala;      //  The tala of the tani
    var <talaSum;   //  The sum of the tala beats
    var dur;        //  The duration of this instance
    var jatis;      //  The total number of Jatis in this instance
    var matras;     //  The total number of matras in this instance
    var numTalas;   //  The number of talas this instance represents
    var rout;       //  The playback routine
    var gati;       //  The gati of the first object. In most cases
                    //  the gati will be the same for all objects,
                    //  but for experimental work this might be mixed.

    *new {|argTani|
        ^super.new.konaTimeInit(argTani);
    }

    //Create a new instance from a given collection of KonaItems
    *newFrom{|aCol, aTani|
        var tani;
        var ret;

        tani = aTani;
        ret  = this.new(tani);

        aCol.size.do { |i|
            ret.add(aCol[i])
        };

        ^ret
    }

    *fill{ |size, function, argTani|
        var newCollection = KonaTime.new(argTani);
        size.do { |i|
            newCollection.add(function.())
        };
        ^newCollection;
    }

    konaTimeInit {|argTani|
        tani = argTani;
        if(tani!=nil,
            {
                tala = tani.tala;
                talaSum = tala.sum;
            }
        );

    }

    //Add an item
    add {|argItem|
        //Ensure against non-KonaItems
        if(argItem.class==KonaWord || (argItem.class==KonaTime)) {
            if(argItem==this) {
                super.add(KonaTime.newFrom(argItem, argItem.tani))
            } {
                super.add(argItem);
            };
        };
        //Accent the first item in the phrase
        if(this.size==1) {
            this.accentFirst;
            gati = this[0].gati;
        };
    }

    //Add a collection
    addAll { |argCollection|
        super.addAll(argCollection);
    }

    //Duration getter method
    dur {
        var ret = 0;
        this.do { |item, i|
            ret = ret + item.dur;
        };
        ^ret;
    }

    //Duration getter method for all contained objects
    allDurs {
        allDurs = Array.newClear(this.size);

        this.do{ |item, i|
            if(item.class==KonaWord) {
                allDurs[i] = Array.fill(item.jatis, item.speed);
            } {
                allDurs[i] = item.allDurs
            };
        }

        ^allDurs;
    }

    //Jatis getter method
    jatis {
        jatis = 0;
        this.size.do { |i|
            jatis = jatis + this[i].jatis;
        };
        ^jatis
    }

    //Matras getter method
    matras {
        matras = 0;
        this.size.do { |i|
            matras = matras + this[i].matras;
        };
        ^matras;
    }
```

```
    //Karve getter method, returns the mode karve of the instance.
    karve {
        var karves = this.collect({|item, i| item.karve});

        ^karves.maxItem({|item, i| karves.occurrencesOf(item)})
    }

    //Returns the number of cycles this instance occupies
    numTalas {
        numTalas = 0;
        ^numTalas = this.dur/this.talaSum;
    }

    //The playback routine for this instance
    rout {
        rout = Routine.new({});
        this.do { |item, i|
            rout = rout++item.rout
        };
        ^rout;
    }

    //Play the instance routine to the parent KonaTani's clock.
    play {
        this.rout.play(tani.clock);
    }


    //Concatonation method for combining KonaItems
    ++ { |aKonaItem|
        var newTime = KonaTime.new(tani);
        newTime.addAll(this).addAll(aKonaItem);
        ^newTime
    }

    //Getter method for the word of all of the contained objects
    word {
        word = List[];
        this.do({|item, i| word.add(item.word)});
        ^word.asArray;
    }

    //Method for printing the word and timing of all contained objects
    postWord {|argW=true, argD=true, argF=true|
        var words, decimals, fractions;
        var get;
        this.do {|item, i|
            get = item.postWord(false, false, false);
            words = words ++ get[0];
            decimals = decimals ++ get[1];
            fractions = fractions ++ get[2];
        };

        if(argW) {
            words.postln;
        };
        if(argD) {
            decimals.postln;
        };
        if(argF) {
            fractions.postln;
        };
        "".postln;
        ^[words, decimals, fractions];
    }

    //Method to accent the first syllable of the first item in the KonaWord
    //Works recursively on KonaTimes
    accentFirst {
        if(this[0]!=nil) {
            this[0].accentFirst;
        };
    }

    //Method to return the maximum speed in this KonaTime, works recursively
    //  on KonaTimes.
    speed {
        var ret = this[0].speed;
        this.do { |item, i|
            if(item.speed < ret) {
                ret = item.speed;
            };
        };
        ^ret;
    }

    //Getter method, returns the gati of the first object.
    gati {
        ^this[0].gati
    }

    //Method to return the largest word in number of jatis (works recursively
    //  for KonaTimes)
    greatestJatis {
        var ret = 0;
        var val;
        this.do { |item, i|
            if(item.class==KonaWord) {
                val = item.jatis;
            } {
                val = item.greatestJatis;
            };

            if(val>ret) {ret = val};
        };

        ^ret;
    }

    //Overridden reverse method to include passing the tani into the resulting
    //  reversed KonaTime
    reverse {
        ^this.class.newFrom(this.asArray.reverse, tani)
    }
}
```

```
/* ============================================================================ */
/* = KonaTani Class - Class to contain and playback whole pieces of music    = */
/* ============================================================================ */

KonaTani {
    classvar allTalas;  //  A set of the common talas

    var <laya;          //  Tempo
    var <tala;          //  Beats in cycle
    var <talaSym;       //  The tala in symbols
    var <gati;          //  'Default' sub-divisions per beats
    var <otherGatis;    //  Sub-divisions to change to
    var <gen;           //  Material generator
    var <store;         //  Whole solo stored in a KonaTime

    //Playback variables
    var <s;             //  Server
    var <konaSynth;     //  Synth to use
    var <clock;         //  Playback Tempo Clock
    var <fftBuffArray;  //  Array of buffers for PV_PlayBuf FFT
    var <fftRout;       //  Routine to cycle through FFT buffers
    var <fftBuff;       //  Next FFT buffer to use
    var <syls;          //  Array of syllables for opening analysis file/directory
    var <buffers;       //  Buffers for scpv files
    var <talaRout;      //  Routine for tala clapping
    var <pRout;         //  Routine for playback;
    var <mOut;          //  MIDIOut


    *initClass {
        allTalas = [
                        ["I4","O","O"],                 //  Adi Tala
                        ["U","O"],                      //  Rupaka Tala
                        ["U1", "R", "W", "W", "R"],     //  Khanda Capu
                        ["W", "W", "R", "U", "U"]       //  Misra Capu
                    ];
    }

    *new {|argLaya=60, argTala=#["I4","O","O"], argGati=4, argOtherGatis=#[3],
argSynth=\konaHit|

        ^super.new.konaTaniInit(argLaya, argTala, argGati, argOtherGatis,
argSynth);
    }

    *rand {|argLaya, argTala, argGati, argOtherGatis, argSynth=\konaHit|

        var laya = argLaya ?? {rrand(60,140)};
        var tala = argTala ?? {allTalas.choose};
        var gati = argGati ?? {[4,3,5,7,9].choose};
        var otherGatis = argOtherGatis ?? {[4,3,5,7,9].select({|item, i|
item!=gati })};

        ^super.new.konaTaniInit(laya, tala, gati, otherGatis, argSynth);

    }

    konaTaniInit {|argLaya, argTala, argGati, argOtherGatis, argSynth|
        var oneCycleDur;
        laya = argLaya;
        talaSym = argTala;
        tala = this.setTala(talaSym);
        gati = argGati;
        otherGatis = argOtherGatis;

        gen = KonaGenerator.new(this, laya, tala, gati);

        oneCycleDur = tala.sum*(60/laya);           //Duration of one cycle

        store = KonaTime.new(this);

        //Playback Variables
        konaSynth = argSynth;
        this.setPlayback;

        //Setup tala and clapping Routine
        this.makeTalaRout();

        //Load SynthDefs
        this.setSynthDefs;
    }

    //Convert the tala from symbols to numbers
    setTala {|aTala|
        var ret = List[];

        if(aTala.every { |item, i| item.class==Integer}) {
            ^aTala;
        } {
            aTala.do { |item, i|
                switch (item[0].asSymbol)
                    {'I'}   {ret.add(item[1].digit)}
                    {'O'}   {ret.add(2)}
                    {'U'}   {
                                switch ((item[1]!=nil).and({item[1].digit}) )
                                    {1}     {ret.add(0.5)}
                                    {false} {ret.add(1)};
                            }
                    {'W'}   {ret.add(0.5)}
                    {'R'}   {ret.add(0.5)};
            };
        };

        ^ret.asArray;

    }

    //Setup playback variables
    setPlayback {
        s = Server.default;
        clock = TempoClock.new(laya/60);
        //Array of buffers for FFT
        fftBuffArray = Array.fill(10, {Buffer.alloc(s, 1024)});

        //Syllables Array
        syls = ['Tam', 'Ta', 'Ka', 'Ki', 'Tah', 'Di', 'Mi', 'Da', 'Gi', 'Na',
                'Dom',  '-', 'Ju', 'Lan', 'Gu', 'Tom', 'Nam', 'Ri', 'Du', 'Din'];

        //Buffers for PV analysis files
```

```
        buffers=Array.newClear(syls.size);
        buffers.size.do({|i| buffers[i] = Buffer.read(s, "sounds/Solkattu
/"++syls[i]++".wav.scpv")});

        //FFT Buffers and Routine
        fftBuff=fftBuffArray[0];
        fftRout = Routine.new({
            inf.do({|i|
                fftBuff = fftBuffArray.wrapAt(i);
                0.yield;
            })
        });

        //MIDI
        MIDIClient.init(1,1);
        mOut = MIDIOut.newByName("IAC Driver", "IAC Bus 1");

    }

    //Setup SynthDefs
    setSynthDefs {
        //Default SynthDef
        SynthDef(\konaHit, { arg out=0, bufnum=0, recBuf=1, rate=1, amp=0.8;
            var chain, signal;
            chain = PV_PlayBuf(bufnum, recBuf, rate, 0, 0);
            signal = IFFT(chain, 1)*amp;
            DetectSilence.ar(signal, doneAction:2);
            Out.ar(out, signal.dup);
        }).load(s);

        //  Clapping SynthDef by Thor Magnusson
        SynthDef(\clapping, {arg t_trig=1, amp=0.5, filterfreq=100, rq=0.1;
            var env, signal, attack,  noise, hpf1, hpf2;
            noise = WhiteNoise.ar(1)+SinOsc.ar([filterfreq/2,filterfreq/2+4 ],
pi*0.5, XLine.kr(1,0.01,4));
            hpf1 = RLPF.ar(noise, filterfreq, rq);
            hpf2 = RHPF.ar(noise, filterfreq/2, rq/4);
            env = EnvGen.kr(Env.perc(0.003, 0.00035));
            signal = (hpf1+hpf2) * env;
            signal = CombC.ar(signal, 0.5, 0.03, 0.031)+CombC.ar(signal, 0.5,
0.03016, 0.06);
            //signal = FreeVerb.ar(signal, 0.23, 0.15, 0.2);
            signal = Limiter.ar(signal, 0.7, 0.01);
            Out.ar(0, Pan2.ar(signal*amp, 0));
            DetectSilence.ar(signal, doneAction:2);
        }).load(s)
    }

    //Playback the contained piece of music with the tala cycle
    play {
            talaRout.reset;
            pRout.reset;
            talaRout.play(clock);
        {
            ((60/laya)*(tala.sum)).wait;
            pRout.play(clock);
        }.fork
    }

    //Stop routine playback
    stop {
        //this.clock.stop;
        talaRout.stop;
        pRout.stop;

    }

    //Generate the clapping routine for the Tala
    makeTalaRout {
        var func = {|amp1, amp2, freq1, freq2, rq| s.bind {Synth(\clapping,
[\amp, rrand(amp1, amp2), \filterfreq, rrand(freq1, freq2), \rq, rq.rand]) }};

        talaRout = Routine {
            inf.do {
                talaSym.do { |item, i|
                    switch (item[0].asSymbol)
                        {'I'}   {
                                    //Clap
                                    func.(0.4, 0.5, 2000, 2500, 0.9);
                                    1.wait;
                                    //Finger Taps
                                    (item[1].digit-1).do { |j|
                                    func.(0.01, 0.05, 6000, 7000, 0.9);
                                    1.wait;
                                };
                                }
                        {'O'}   {
                                    //Clap
                                    func.(0.4, 0.5, 2000, 2500, 0.9);
                                    1.wait;
                                    //Back of hand / wave
                                    func.(0.01, 0.03, 400, 600, 0.9);
                                    1.wait;
                                }
                        {'U'}   {
                                    //Clap
                                    func.(0.4, 0.5, 2000, 2500, 0.9);
                                    switch ((item[1]!=nil).and({item[1].digit}) )
                                        {1}     {0.5.wait}
                                        {false} {1.wait};
                                }
                        {'W'}   {
                                    //Wave
                                    func.(0.3, 0.4, 400, 600, 0.9);
                                    0.5.wait;
                                }
                        {'R'}   {
                                    //Rest
                                    0.5.wait;
                                };
                };
            };
        };

    }

    //Getter for the music routine
    rout {
        ^store.rout;
    }
```

```
    //Add an item to this instance's KonaTime
    add {|aKonaItem|
        store.add(aKonaItem);
        pRout = store.rout;
    }

    //Add a collection of items to this instance's KonaTime
    addAll { |argCollection|
        store.addAll(argCollection);
        pRout = store.rout;
    }

    //Clear this instance's KonaTime
    clear {
        store = KonaTime.new(this);
        pRout = store.rout;
    }

    //Setter method for the gati
    gati_{|aGati|
        gati = aGati;
        gen.gati = aGati;
    }
}
```

```
/* ================================================================== */
/* = KonaGenerator Class - Class to generate and manipulate rhythms = */
/* ================================================================== */

KonaGenerator {
    var tani;           //  The Tani that this KonaGenerator belongs to
    var laya;           //  The Laya (tempo) of the tani
    var <>gati;         //  The current Gati (beat subdivision) of the Tani
    var <gatiPowers;    //  The series of powers belonging to the gati;
    var <tala;          //  The Tala of this Tani
    var <>pMax; //Maximum perceptual time for a motif. May make this method
specific

    *new {|argTani, argLaya, argTala, argGati=4|
        ^super.new.konaGeneratorInit(argTani, argLaya, argTala, argGati);
    }

    konaGeneratorInit {|argTani, argLaya, argTala, argGati|
        tani = argTani;
        laya = argLaya;
        tala = argTala;
        gati = argGati;

        gatiPowers = this.setGatiPowers(argGati);

        pMax = 5;       //Rough perceptual present in seconds
    }

    //  setGatiPowers
    //  Method to create the series of values to which the gati belongs
    //
    //  e.g. Series for gati 3 = [3, 6, 12, 24, 48, 96, 192]
    //  Special circumstances for gati 4 to include 2 = [2, 4, 8, 16, 32, 64, 128]
    setGatiPowers {|aGati|
        var powers;

        if(aGati==4,
            {powers = List[(aGati/2).asInteger]},
            {powers = List[(aGati).asInteger]}
        );
        10.do { |i|
            powers.add( (powers[powers.size-1]*2).asInteger)
        };

        ^powers.asArray;

    }

    /* ======================================================= */
    /* =             Generation Methods                      = */
    /* ======================================================= */

                    /* = Maths Methods = */

    //  allPartitions
    //  @n      Total number of beats to partition
    //  @min    Minimum part size
    //  @max    Maximum part size
    //
    //  -Method to generate all partitions and permutations of an integer
(duration)
    //  -Uses ZS2 Algorithm from "Fast Algorithms For Generating Integer
Partitions"
    //      by Zoghbi and Stojmenovic
    //  -Doubly restricted by default to 2 and 9, but the mininum
    //      restriction is available as an argument
    //  -This is because of the restriction on Konakkol word size

    allPartitions { |aNum, aMin=2, aMax, aUParts|
        var tmax;   //  The maxmimum size a part of a partition may be
        var x;      //  An array to store each new partition in;
        var h;      //  The index of the last part of partition that is > 1
        var m;      //  The number of parts in a partition
        var j;      //  The index of the next part to be increased
        var r;      //  A variable used to calculate the next m
        var partition;  //  A freshly baked partition
        var add;        //  Boolean; whether this partition should be added.
        var ret;        //  The array to be returned.
        var readFunc;   //  Function to read partitions from a file.
        var n, min, max;    //vars for aNum aMin and aMax.

        //Ensure against floats.
        n = aNum.asInteger;
        min = aMin.asInteger;
        readFunc = {|val|
Object.readArchive(Platform.userExtensionDir++"/FYPClasses/partitions/"++val.asString)};

        case
            //There are no partitions of 1
            {n==1}              {^[[1]]}
            //If n==2 and min==2 there are no partitions of
            {n==2 && (min==2)}  {^[[2]]}
            {n>=40}             {
                if(aUParts==nil) {
                    ^readFunc.(n);
                } {
                    ^this.removeGreaterThan(readFunc.(n), aUParts);
                };
            };

        ret = List[];
        //Fill the array with n 1s
        x = Array.fill(n, 1);
        //Add the array as it forms the first partition
        if(min==1) {
            ret.add(x[0..n]);
        };
        //Alter x; set the second element ([1]) to 2 and add the subarray x[1..n]
        x[1] = 2;
        if(min==1) {
            ret.add(x[1..n]);
        };

        h = 1;
        m = n-1;

        //If the max argument is not set, use n if below 9, else use 9
        if( aMax == nil ) {
            if(n>9) {
                tmax = 9;
```

```
        } {
            tmax = n;
        };
    } {
        //Else if the argument is n-1 or n, use the argument as given
        if(aMax >= (n-1)) {
            tmax = aMax.asInteger;
        } {
            //Else add 1 to the argument.
            //This is to ensure that the maximum argument works correctly
            //If used as given, a max arg of 3 would return results up to size 3

            tmax = aMax.asInteger+1;
        };

    };

    //Generate further partitions
    while({x[1] != tmax},
        {
            if( (m-h) > 1) {
                h = h+1;
                x[h] = 2;
                m = (m-1);
            } {
                j = (m-2);

                while({x[j] == x[m-1]},
                    {
                        x[j] = 1;
                        j = (j-1);
                    }
                );
                h = (j+1);
                x[h] = (x[m-1] +1);
                r = (x[m] + ((x[m-1])*(m-h-1)));

                x[m] = 1;

                if( (m-h) > 1 ) {
                    x[m-1] = 1
                };
                m = (h + (r-1));

            };
            partition = x[1..m];
            //If a maximum number of unique parts has been set
            if(aUParts!=nil) {
                //If the number of unique parts is acceptable.
                if(partition.asSet.size<=aUParts) {
                    add = true
                } {
                    add = false
                };
            } {
                add = true;
            };

            if(partition.minItem >= min && (add==true)) {
                ret.add(partition);
                        };

            });

            ^ret.asArray;

        }

        //  randomPartition
        //  Method to choose a random partition
        //
        //  @duration   number of beats for the partition
        //  @min        minimum part size
        //  @max        maximum part size
        //  @notSize    boolean, true means partition of 1 part equal to size will
        //                  not be returned. E.g. duration 4 can't return partition [4]
        //  @seed       seed for random selection

        randomPartition { |duration, min=2, aMax, notSize=false, seed|
            var allPartitions;
            var max;

            if(seed!=nil) {
                thisThread.randSeed=seed
            };

            max = aMax ?? {if(duration>9) {9} {duration}};

            allPartitions = this.allPartitions(duration, min, max);

            if(notSize && (allPartitions.size>1)) {

                allPartitions.do { |item, i|
                    if(item[0]==duration) {
                        allPartitions.removeAt(i)
                    };
                };
            };

            ^allPartitions.choose;
        }

        //  allPerms
        //  Method to generate all permutations of a partition
        //
        //  @aCollection    The partition array to generate permutations of

        allPerms {|aCollection|
            var col;    //Collection to permute
            var ret;    //Array to return permutations
            var perm;   //Temp variable for storing permutation
            col = aCollection;
            ret = List[];

            //If the partition is not just made up of 1 unique number (e.g. [2,2,2])
            if(col.occurrencesOf(col[0]) != col.size) {
                //Loop to create all permutations
                col.size.factorial.asInteger.do { |i|
                    perm = col.permute(i);
```

```
            //If ret doesn't already contain the new permutation
            if(not(ret.any { |item, i| item.asArray == perm })) {
                //Add it
                ret.add(col.permute(i));
            };
        };
    } {
        //Else (if the partition IS made up of just 1 unique number)
        ret.add(col);
    };

    //Return all partitions
    ^ret.asArray;
}

//  randomPerm
//  Method to choose a random permutation
//
//  @partition  Partition to generate permutations from
//  @seed       seed for random selection
randomPerm { |partition, seed|
    var permutation;
    if(seed!=nil) {
        thisThread.randSeed=seed
    };

    permutation = (partition.size+1).factorial.asInteger.rand;

    ^partition.permute(permutation);
}

//  removeGreaterThan
//  Method to remove all partitions from a collection
//      that have more than a given number of unique parts
//
//  @aCollection    Collection to remove partitions from
//  @val            Maximum number of unique partitions
removeGreaterThan {|aCollection, val, weight=0.97|
    var col;        //  Instance collection
    var temp;       //  Temporary list for checking partitions

    col = aCollection;
    temp = List[];
    col.do { |item, i|
        if(item.asSet.size>val) {
            temp.add(i)
        };
    };
    col = col.removeAtIndexes(temp);
    ^col;
}

//  removeThoseContaining
//  Method to remove all partitions from a collection
//      that contain certain values
//  Individual weights can be passed for probabalistic results
//
//  @aCollection    Collection to remove paritions from
//  @valCol         Collection of taboo values
//  @weightCol      Collection of weights for taboo values
removeThoseContaining {|aCollection, valCol, weightCol|

    var col;        //  Partitions
    var vCol;       //  Values
    var wCol;       //  Weights
    var inds;       //  Indexes of partitions to remove
    var saveIndex;

    col = aCollection;

    vCol = valCol;

    //If no weights are supplied, remove is guaranteed
    wCol = weightCol ?? {Array.fill(vCol.size, 1)};

    inds = List[];
    //For each forbidden value
    vCol.size.do { |i|
        //Check for partitions that include the value
        col.do { |jtem, j|
            if(jtem.includes(vCol[i])) {
                //Store the index depending on given weight
                if(wCol[i].coin) {
                    inds.add(j);
                };
            };
        };
    };
    inds = inds.asSet.asArray.sort;

    //If all partitions are to be removed, select one at random to keep
    if(inds.size==col.size) {
        //Store the index of value least likely to be removed
        saveIndex = wCol.indexOf(wCol.minItem);
        //Select an index from those partitions that include
        //  the least likely value
        saveIndex = inds.select({|item, i|
            col[i].includes(vCol[saveIndex]);
        }).choose;

        inds.removeAt(saveIndex);
    };
    col.removeAtIndexes(inds.asArray)

    //Return updated collection
    ^col
}

//  partsToWords
//  Method to turn a partition array into KonaWords
//
//  @aPartitionArray    Partition Array
//  @aOne               Boolean; if KonaWords can be 1 syllable
//  @aMult              Boolean; if Konawords can have syllables == part size
partsToWords {|aPartitionArray, aKarve, aOne=true, aMult=true|
    var partitionArray;
    var one, mult;
    var ret;
    var chance;
    var jatis, karve;
```

```
        partitionArray = aPartitionArray;
        one = aOne;
        mult = aMult;
        case
            {aOne==true && (aMult==true)}       {chance = 0.5}
            {aOne==true && (aMult==false)}  {chance = 1}
            {aOne==false && (aMult==true)}  {chance = 0}
            {aOne==false && (aMult==false)} {chance = 0.5};

        ret = KonaTime.new(tani);

        aPartitionArray.size.do { |i|
            if(chance.coin) {
                jatis = 1;
                karve = aPartitionArray[i];
            } {
                jatis = aPartitionArray[i];
                karve = 1;
            };
            ret.add(KonaWord.new(jatis, gati, karve*aKarve, tani))
        };
        ^ret
    }

                                            /* ================ */
                                            /* = Music Methods = */
                                            /* ================ */


    //  vSarvaPhraseLength
    //  Method to determine the phrase length (in beats)
    //      for sarvalaghu patterns for the Vilamba Kala section
    //  Uses a tweaked perceptual present model,
    //      currently uses a window of 5 seconds.
    vSarvaPhraseLength {
        var phraseLength;
        var oneBeat;
        var maxBeats;
        var val;
        //Time in seconds for one beat
        oneBeat = 60/laya;
        Post << "oneBeat: " <<  oneBeat << "\n";

        //The number of beats that can fit into the maximum perceptual time
        //With a maximum number of beats of 5. Even if perceptual time is 3 seconds,
        //  phrases are not usually longer than this
        maxBeats = (pMax/oneBeat).min(5);
        Post << "maxBeats: " <<  maxBeats << "\n";

        //  This algorithm attempts to find that largest phrase length that
        //  fits neatly into a full cycle.
        //  At the moment this only works in terms of half/quarter/eigth cycles etc
        //  Could be adapted to find other durations of phrase that
        //  can fit neatly into a cycle
        //  E.g. a 9 beat tala could be made up of 3 * 3 beat phrases
        //  Not a huge amount of material to support this theory.

        phraseLength = 0;
        val = tala.sum;

        while({phraseLength==0},
            {
                phraseLength = maxBeats-(maxBeats%val);
                val = val/2;
            }
        );

        Post << "phraseLength: " <<  phraseLength << "\n";

        ^phraseLength
    }

    //  vSarvaPhraseAuto
    //  Automation of vSarvaPhrase
    vSarvaPhraseAuto {

        "vSarvaPhraseLength*gati: ".post; (this.vSarvaPhraseLength*gati).postln;

        ^this.vSarvaPhrase(this.vSarvaPhraseLength*gati);
    }

    //  vSarvaPhrase
    //  Method to generate a phrase for the Vilamba section sarvalaghu
    vSarvaPhrase {|phraseDur, aMin=2|
            var phraseMatras;
            var jatiParts;
            var min;
            var max;
            var partsArray, weights, maxW, maxW1, maxW2, maxW1MI;
            var muteChance;
            var ret;

            if(phraseDur%1!=0) {
                ret = KonaTime.new(tani);
                ret.add(this.vSarvaPhrase(phraseDur.floor, aMin));
                ret.add(KonaWord.new(1, gati, (phraseDur-phraseDur.floor)));
                ^ret;
            };

            phraseMatras = phraseDur;

            if(phraseDur<aMin) {
                min = phraseDur;
            } {
                min = aMin;
            };

            if(2*gati<=phraseDur) {
                max = gati;
            } {
                max = phraseDur;
            };

            //Possible part sizes
            partsArray = (min..max);
            Post << "partsArray: " <<  partsArray << "\n";
```

```
        //Parts 2 to gati. Given heaviest weightings
        weights = Array.fill(partsArray.size, 0);

        weights.size.do { |i|

            if(this.gatiPowers.includes(partsArray[i]),
                {weights[i] = 1.5},
                {weights[i] = 0.4}
            );

            if(partsArray[i]<gati) {
              weights[i] = weights[i] + 0.25;
            } {
                if(partsArray[i]!=gati) {
                    weights[i] = weights[i] - 0.4
                };
            };

            if(gati==5 || (gati==7)) {
              if((partsArray[i]== 3) || (partsArray[i]== 2)) {
                    weights[i] = weights[i] + 0.25;
                } {
                    weights[i] = weights[i] - 0.25
                };
            };

            if(weights[i]<0,
                {weights[i] = 0}
            );

        };
        //Scale and invert values.
        weights = (weights/weights.maxItem-1).round(0.01).abs;

        jatiParts = this.allPartitions(phraseMatras.asInteger, aMax: max);

        jatiParts = this.removeGreaterThan(jatiParts, 4);

        jatiParts = this.removeThoseContaining(jatiParts, partsArray,
weights);

        jatiParts = jatiParts.choose;

        jatiParts = this.randomPerm(jatiParts);

        ret = KonaTime.new(tani);
        jatiParts.size.do { |i|
            if((jatiParts[i].even && (jatiParts[i]>2)) && 0.75.coin ) {
                ret.add(KonaWord.new(jatiParts[i]/2, gati, 2, tani))
            } {
                ret.add(KonaWord.new(1, gati, jatiParts[i], tani))
            };
        };
        "Conversion to KonaWords: ".postln;ret.postWord(true, true, false);

        ret = this.combineSimilar(ret, 2, 4, 0.9);
        "combineSimilar: ".postln; ret.postWord(true, true, false);

        muteChance = 0.75;
        3.rand.do { |i|
            if(muteChance.coin) {
                ret = this.randomMuteJati(ret);
            };
            muteChance = muteChance/2;
        };
        "randomMuteJati: ".postln; ret.postWord(true, true, false);

        ^ret;

    }

    //  mutatePhrase
    //  Method to mutate a given phrase using many possible
    //      combinations of automated manipulation methods
    //
    //  @aKonaItem Item to manipulate;
    mutatePhrase {|aKonaItem, aChance, aRec=0.75, aNum|
        var col;        //  Input collection
        var ret;        //  Output collection
        var change;     //  The chance an item will be mutated;
        var min;        //  Minimum value for alteration
        var max;        //  Maximum value for alteration
        var val;        //  Variable used to calculate density possibilties
        var count;      //  Variable used when calculating density possibilities
        var index;      //  Index of element to mutate
        var store;      //  Array to store indexes to be removed (atDensity)
        var num;        //  Index of process to use;

        if(aKonaItem.class==KonaTime) {
            col = aKonaItem;
        } {
            col = KonaTime.newFrom([aKonaItem], tani);
        };

        change = aChance ?? {(1/col.size)*1.5};
        if(aNum==nil) {
            num = {5.rand}
        } {
            num = {aNum}
        };
        ret = KonaTime.new(tani);
        col.size.do { |i|
            if(change.coin,
                {
                    if(col[i].class==KonaWord) {
                    switch (num.())
                        {0} {
                                ret.add(this.randomAtDensity(col[i]));
                            }
                        {1} {
                                ret.add(this.randomExtendJati(col[i]));
                            }
                        {2} {
                                ret.add(this.randomMuteJati(col[i]));
                            }
                        {3} {
                                ret.add(this.randomDensityJati(col[i]));
                            }
                        {4} {
                                ret.add(this.partitionWord(col[i]));
```

```
                        };
                } {
                    ret.add(this.mutatePhrase(col[i], aRec=aRec/4))
                };

            },
            {
                ret.add(col[i])
            }
        );
    };

    //Possible recursion for more mutation.
    if(aRec.coin) {
        ^this.mutatePhrase(ret, aRec:aRec/2, aNum:aNum)
    } {
        ^ret
    };
}

//  vSarvaStat
//  A method for generating Sarva Laghu material based on
//      a statistical analysis of a performance by Trichy Sankaran;
//  Currently only works with an n of 1, no context.
vSarvaStat {
    var stats;
    var ret;

    ret = KonaTime.new(tani);

    stats = [
        100, 37.5, 87.5, 68.75,
        93.75, 12.5, 100, 25,

        100, 80, 100, 13,
        100, 13, 100, 6,

        100, 0, 73, 53,
        100, 0, 100, 22,

        89, 66, 100, 22,
        100, 30, 80, 30
    ];

    stats.size.do { |i|
        if((stats[i]/100).coin) {
            ret.add(KonaWord.new(1,4,1,tani))
        } {
            ret.add(KonaWord.new(0,4,1,tani))
        };
    };

    ^ret
}

////////////////////////   Moras   ////////////////////////

//  createSimpleMora
//  Builds a mora structure from a given statement
//      with optional gap and offset.
//
//  @statement  KonaObject for Statement
//  @gap        KonaObject for Gap
//  @offset     KonaObject for Offset
createSimpleMora {|statement, gap, offset|

    var mora = KonaTime.new(tani);

    if(offset!=nil,
        {mora.add(offset)}
    );

    2.do {
        mora.add(statement);
        if(gap!=nil,
            {mora.add(gap)}
        );
    };
    mora.add(statement);

    ^mora
}

//  moraStatement
//  Method to generate a mora statement
//
//  @statePulses    Statement jatis
//  @aGati      Gati of the statement
//  @aKarve     Karve to use
moraStatement {|aStateMatras, aGati, aKarve|
    var statePulses;
    var statement;
    var ret;
    var temp;


    statePulses = aStateMatras*(1/aKarve);

    //Turn statements into KonaItems
    //If the statement duration can be a single word
    if(statePulses<=9) {
            statement = KonaWord.new(statePulses, aGati, aKarve, tani);
    } {
        //If a statement duration requires more than a single word
        //Generate a partition
        statement = this.randomPartition(statePulses.asInteger);
        //Choose a permutation
        statement = this.randomPerm(statement);

        //Convert to KonaTime
        ret = KonaTime.new(tani);

        statement.size.do {|i|
            //New word jatis equal to part duration
            temp = KonaWord.new(statement[i], aGati, aKarve, tani);
            ret.add(temp);
        };
        statement = ret;
    };
    //statement = this.partitionWord(statement);
```

```
        if(0.5.coin) {
            statement = this.randomDensityJati(statement);
        };

        ^statement;
    }

    //  moraGap
    //  Method to generate a mora gap.
    //
    //  @gapPulses      Statement jatis
    //  @aGati      Gati of the statement
    //  @aKarve     Karve to use
    moraGap {|aGapMatras, aGati, aKarve|
        var gapPulses;
        var gap;
        var temp;

        gapPulses = aGapMatras*(1/aKarve);

        if(gapPulses==0) {
            gap=nil;
        } {
            if(gapPulses>4 && 0.95.coin) {
                gap = this.randomPartition(gapPulses.asInteger, notSize:true);
                gap = this.randomPerm(gap);

                temp = KonaTime.new(tani);
                gap.size.do { |i|
                    if(i==0) {
                        temp.add(KonaWord.new(1, aGati, gap[i]*aKarve, tani))
                    } {
                        temp.add(KonaWord.new(gap[i], aGati, aKarve, tani));
                    };
                };
                //gap = this.mutatePhrase(temp);
                gap = temp;

            } {
                //Generate single jati gap with gapPulses duration
                if(aKarve>=0.25 && (0.5.coin)) {
                    gap = KonaWord.new(0, aGati, gapPulses*aKarve, tani)
                } {
                    gap = KonaWord.new(1, aGati, gapPulses*aKarve, tani)
                };
            };
        };

        ^gap
    }

    //  moraOffset
    //  Method to generate a mora offset
    //
    //  @offsetPulses   Offset jatis
    //  @aGati      Gati of the statement
    //  @aKarve     Karve to use
    moraOffset {|aOffsetMatras, aGati, aKarve|
        var offsetPulses;
        var offset = nil;
        var phraseMin;      //Minimum part size if the offset is to be a phrase.


        offsetPulses = aOffsetMatras*(1/aKarve);

        if(offsetPulses!=0) {
            case
                //If the offset is greater than 2 beats, use a phrase
                {offsetPulses>(aGati*2)}    {
                    if(offsetPulses>20) {
                        phraseMin = 4
                    } {
                        phraseMin = 2;
                    };
                    offset = this.vSarvaPhrase(aOffsetMatras, aMin:phraseMin);

                }
                //If the offset is less than 2 beats, has a 0.05 chance articulation.
                {0.05.coin} {
                    offset = KonaWord.new(offsetPulses, aGati, aKarve, tani)
                }
                //Else a single syllable word is used.
                {true}  {
                    offset = KonaWord.new(1, aGati, aOffsetMatras, tani)
                };

        } {
            offset = nil;
        };

        ^offset
    }

    //  randomMoraValues
    //  Calculation of mora values (statement, gap, offset durations).
    //
    //  @aMatras    The duration of the mora in matras
    //  @aGati      The gati of the mora elements
    //  @aGati      The karve of the mora elements
    //  @aGap       Boolean, gaps or not, overridden for certain durations.
    //  @aOffset    Boolean, offset or not
    randomMoraValues {|aMatras, aGati, aKarve, aGap=true, aOffset=true|
        var pulses;
        var stateMin, gapMin;
        var stateMax, gapMax;
        var gapArray, gapWeights;
        var stateMatras, gapMatras, offsetMatras;
        var totalStateMatras, totalGapMatras;

        pulses = aMatras*(1/aKarve);

        //  Nelson 2008 p 23
        //  'It is a practical fact of Karnatak rhythmic behaviour that if a mora
        //  statement is shorter than five pulses, its gap will nearly always be
        //  at least two pulses'.
        //  This is impossible if a duration of less than 7 is used.
        //  In this instance a mora with the same duration,
        //  but using double the jatis and half the karve is returned
        //  Moras under 2 whole beats are also given a chance of being altered.
```

```
        if(pulses<7 || (aMatras/aGati<=2 && 0.25.coin && (aKarve>0.5))) {
            ^this.randomMoraValues(aMatras, aGati, aKarve/2, aGap, aOffset);
        };

        //Any duration under 15 will result in statements less than 5 pulses
        //  so requires a minimum gap of 2
        if(pulses<15) {
            gapMin = 2;
        } {
            gapMin = 0;
        };

        //  Calculate the mininum matras for the statements.
        //  If might be no gap, use a minimum size of 1/4 of the total mora duration
        if(gapMin==0) {
            stateMin = (pulses/(3.00, 3.05..4.00).choose).asInteger
        } {
            stateMin = (pulses/5).asInteger
        };

        //Calculate the maximum possible statement size
        stateMax = (pulses-(gapMin*2)/3).asInteger;

        //Select a statement duration
        stateMatras = (stateMin..stateMax).choose;
        totalStateMatras = stateMatras*3;

        if(aGap) {
            //Calculate the maximum possible gap size.
            gapMax = (pulses-totalStateMatras)/2;

            gapArray = (gapMin..gapMax);

            //Calculate weights for gap matras selection, with bias for smaller gaps.

            gapWeights = (gapArray.size..1).normalizeSum;

            //Choose a gap duration
            gapMatras = gapArray.wchoose(gapWeights);
        } {
            gapMatras = 0;
        };
        totalGapMatras = gapMatras*2;

        //If there should be an offset, calculate the duration
        if(aOffset) {
            offsetMatras = pulses - totalStateMatras - totalGapMatras;
        } {
            offsetMatras = 0;
        };


        ^[stateMatras, gapMatras, offsetMatras, aKarve];
    }

    //  randomMora
    //  Generation of a mora from given parameters;
    //
```



```
//  @aMatras    The duration of the mora in matras
//  @aGati      The gati of the mora elements
//  @aGap       Boolean, whether there should be gaps or not
//  @aOffset    Boolean, whether there should be an offset or not
randomMora {|aMatras, aGati, aKarve, aGap=true, aOffset=true|
    var values;
    var statement, gap, offset;

    values = this.randomMoraValues(aMatras, aGati, aKarve, aGap, aOffset);


    //Convert statements/gaps/offset into KonaItems

    statement = this.moraStatement(values[0]*values[3], aGati, values[3]);


    gap = this.moraGap(values[1]*values[3], aGati, values[3]);
    if(gap!=nil) {
    };


    offset = this.moraOffset(values[2]*values[3], aGati, values[3]);
    if(offset!=nil) {
    };


    ^this.createSimpleMora(statement, gap, offset);
}

//  Generative method to create a mora from a given statement,
//      with optional maximum mora size, gap and offset.
//  Differs from createSimpleMora in that gaps and offsets will
//      be calculated and generated if possible
//
//  @aStatement     Kona object to use for statement
//  @aMoraMatras    Total maximum number of matras, overidden if less
//                      than sum of aStatement, aGap, aOffset matras
//  @aGap           Kona object to use for gap
//  @aOffset        Kona object to use for offset
moraFrom {|aStatement, aMoraMatras, aGap, aOffset|
    var statement, gap, offset;
    var objArray;
    var objMatras, moraMatras, gapMatras, offsetMatras;

    statement = aStatement;
    gap = aGap;
    offset = aOffset;

    objMatras = 0;
    objArray = [statement,gap,offset];
    //Calculate total matras of mora sections passed as arguments
    objArray.do { |item, i|
        var val;
        if(item!=nil && (item!=false)) {
            switch (item)
                {statement} {val = 3}
                {gap}       {val = 2}
                {offset}    {val = 1};

            objMatras = objMatras + (item.matras*val);
```

```
            };
        };

        //If no maximum duration has been given
        if(aMoraMatras==nil || (aMoraMatras ? 0 <objMatras) ) {

            //Calculate the new maximum duration
            moraMatras = objMatras;
            if(gap==nil || (gap==false)) {
                gapMatras=0;
            } {
                gapMatras=gap.matras
            };
            if(offset==nil) {
                offsetMatras=0;
            } {
                offsetMatras = offset.matras;
            };
        } {
            //If a maximum duration has been given
            moraMatras = aMoraMatras;

            //Calculate the lengths of the various sections.
            //Various cases of passed in gaps and offset.
            case
                {gap==nil && (offset==nil)} {
                    gapMatras = (0..(moraMatras-objMatras)/2).choose;
                    objMatras = objMatras + (gapMatras*2);
                    offsetMatras = moraMatras - objMatras;
                    objMatras = objMatras + offsetMatras;
                }
                {gap==nil && (offset!=nil)} {
                    offsetMatras = offset.matras;
                    gapMatras = (0..(moraMatras-objMatras)/2);
                    objMatras = objMatras + gapMatras*2;
                }
                {gap!=nil && (gap!=false) && (offset==nil)} {
                    if(gap!=false) {
                        gapMatras = gap.matras;
                    };
                    offsetMatras = moraMatras - objMatras;
                    objMatras = objMatras + offsetMatras;
                }
                {gap!=nil && (gap!=false) && (offset!=nil)} {
                    gapMatras = gap.matras;
                    offsetMatras = offset.matras;
                };
        };

        //If no gap has been set, create one.
        gap = gap ?? {this.moraGap(gapMatras, statement.gati, statement.karve)};
        //If there should be no gap, set it to nil
        if(gap==false) {
            gap = nil;
        };

        //If no offset has been calculate the duration and create one
        if(offsetMatras==nil) {
            if(offset==nil) {
                offsetMatras = moraMatras - objMatras;
            } {
                offsetMatras = offset.matras;

            };
        };
        offset =  offset ?? {this.moraOffset(offsetMatras, statement.gati, statement.karve)};

        //Construct and return the mora
        ^this.createSimpleMora(statement, gap, offset);
    }

    //  randomSamaCompoundMora
    //  Generate a random compound mora with the 'sama' shape
    //
    //  @aMatras    Duration in matras
    //  @aGati      The gati to use
    //  @aKarve     The karve to use
    randomSamaCompoundMora {|aMatras, aGati, aKarve|
        var values;
        var stateDur, gapDur, offsetDur;
        var statement, gap, offset;

        values = this.randomMoraValues(aMatras, aGati, aKarve, true, true);

        stateDur = values[0];

        gapDur = values[1];

        statement = this.randomMora(stateDur*values[3], aGati, values[3], true, false);


        if(gapDur==0) {
            gap = false;
        } {
            gap = this.moraGap(gapDur*values[3], aGati, values[3]);
        };

        ^this.moraFrom(statement, aMatras, gap);
    }

    //  basicStructure
    //  A method to generate a basic structure based on the tala.
    //  The purpose is to get a feel for the system's components in a context
    //  Always follows the same structure:
    //  First Cycle:        Basic phrase and a development with suffix
    //  Second Cycle:       More developments of the basic phrase
    //  Third Cycle:        Basic phrases with a half cycle mora
    //  Fourth Cycle:       Developed phrases
    //  Fifth/Sixth Cycle:  Compound Mora
    //  Remaining Cycles:   Play previous six cycles in a new gati,
    //                          with a final mora to fill any unfinished cycles.

    basicStructure {
        var basicPhrase, developedPhrase;
        var phraseMatras, cycleMatras, moraMatras;
        var phraseMult;
        var newGati, newKarve, gatiChange, gatiChangeMatras, changeRemainder;
        var preFinalFiller, finalMora;
        var ret;
```

```
		ret = KonaTime.new(tani);
		cycleMatras = tani.tala.sum*tani.gati;
		newGati = [3,5,7,9].wchoose([1,0.5,0.25,0.125].normalizeSum);
		switch (newGati)
			{3} {newKarve = [0.5, 1].choose}
			{5} {newKarve = 2}
			{7} {newKarve = 4}
			{9} {newKarve = 4};

		basicPhrase = this.vSarvaPhraseAuto;
		phraseMatras = basicPhrase.matras;
		phraseMult = cycleMatras/phraseMatras;
		developedPhrase = this.mutatePhrase(basicPhrase);


		//Fill first cycle
		ret.addAll([basicPhrase,developedPhrase]);
		(phraseMult-2).do { |i|
			ret.add(this.mutatePhrase(basicPhrase));
		};
		ret[ret.size-1] = this.addSuffix(ret[ret.size-1])[0];

		Post << "1st ret.dur: " <<  ret.dur << "\n";

		//Fill second cycle
		ret.add(this.makePostMora(developedPhrase));
		(phraseMult-1).do { |i|
			ret.add(this.randomDensityJati(developedPhrase));
		};
		Post << "2nd ret.dur: " <<  ret.dur << "\n";

		//Fill third cycle
		moraMatras = (cycleMatras/2).ceil;
		if(cycleMatras-moraMatras!=0) {
			ret.add(this.vSarvaPhrase(cycleMatras-moraMatras));
		};
		ret.add(this.randomMora(moraMatras, tani.gati, 1));
		Post << "3rd ret.dur: " <<  ret.dur << "\n";

		//Fill fourth cycle
		ret.add(this.makePostMora(developedPhrase));
		(phraseMult-1).do { |i|
			ret.add(this.mutatePhrase(developedPhrase));
		};
		Post << "4th ret.dur: " <<  ret.dur << "\n";

		//Fill fifth and sixth cycles
		ret.add(this.randomSamaCompoundMora(cycleMatras*2, tani.gati, 1));

		Post << "5th 6th ret.dur: " <<  ret.dur << "\n";
		//Gati Change
		gatiChange = this.phraseAtGati(ret, newGati, newKarve);
		gatiChangeMatras = (gatiChange.matras/newGati)*tani.gati;
		Post << "newGati: " <<  newGati << "\n";

		Post << "gatiChangeMatras: " <<  gatiChangeMatras << "\n";

		if(gatiChangeMatras%1!=0) {
			preFinalFiller = (1-(gatiChangeMatras%1))*newGati;
			ret.add(KonaWord.new(1, newGati, preFinalFiller, tani))
		};
		changeRemainder = gatiChangeMatras%cycleMatras;
		ret.add(gatiChange);
		Post << " gati change  ret.dur: " <<  ret.dur << "\n";

		Post << "changeRemainder: " <<  changeRemainder << "\n";

		//Fill remainder
		if(changeRemainder!=0) {
			if(changeRemainder>(cycleMatras/2)) {
				finalMora = changeRemainder-(cycleMatras/2);
				ret.add(this.vSarvaPhrase(changeRemainder-(cycleMatras/2)));
			} {
				finalMora = changeRemainder;
			};
			finalMora = this.randomMora(finalMora, tani.gati, [1, 0.5].choose);
		};

		ret.add(KonaWord.new(1, tani.gati, tani.gati, tani));
		Post << "ret.dur: " <<  ret.dur << "\n";

		^ret;
	}


	/* ================================================================================ */
	/* =                         Manipulation Methods                         = */
	/* ================================================================================ */




	//  wordAtGati
	//  @argWord        Word to manipulate
	//  @argGati        New Gati
	//  @argKarve       Number of gati divisions each jati should occupy
	//
	//Method to return a word  at a new Gati, including double tempo etc
	//The gati and karve arguments are taken absolutely
	//For 4-->G3E1 A word with Gati 4, Karve 2, [0.125, 0.125], would be [0.33, 0.33]

	wordAtGati { |argWord, argGati, argKarve|
		^KonaWord.new(argWord.jatis, argGati, argKarve, tani)
	}

	//  phraseAtGati
	//  @argObj                 KonaTime (or word) to manipulate
	//  @argGati                New Gati
	//  @argGatiExpansion       Karve multiple.
	//
	//  Method to return a phrase at a new Gati, including double tempo etc
	//  The expansion value is relative to the input objects expansion,
	//      so that phrases maintain their relative values
```

```
    phraseAtGati {|argObj, argGati, argGatiExp|
        var temp = KonaTime.new(tani);

        if(argObj.class==KonaWord) {
            ^KonaWord.new(argObj.jatis, argGati, argObj.karve*argGatiExp, tani)
        } {
            argObj.do{ |item, i|
                temp.add(this.phraseAtGati(item, argGati, argGatiExp));
            }
            ^temp;
        };
    }

    //  combine
    //  @aCollection    A collection of KonaItems with a combined desired jati number
    //
    //  Method to create a new KonaWord/KonaTime from the number of
    //      syllables of the given items
    //  Only used for KonaWords of the same karve

    combine {|aCollection|
        var dur;        //Desired jatis for output
        var karve;      //Karve of the input (determines output Karve)
        var ret;        //KonaItems to return
        var allRest;    //Boolean; whether the collection is silent syllables
        var oneSyl;     //Boolean; if the collection is one syllable.

        dur = 0;
        ret = KonaTime.new(tani);
        oneSyl = false;
        allRest = aCollection.every { |item, i| item.word == ['-']};

        if(aCollection.size>0) {
            karve = aCollection[0].karve;
        } {
          karve = nil;
        };

        aCollection.size.do { |i|
            dur = dur + aCollection[i].jatis;
            if(i==0) {
                if(aCollection[i].word==['Ta']) {
                    oneSyl = true;
                };
            } {
                if(aCollection[i].word!=['-']) {
                    oneSyl = false;
                };
            };

        };


        if(oneSyl) {
            ret = KonaWord.new(1, gati, aCollection.matras, tani);
        } {
            while({dur!=0},
                {
                    if(dur<=9) {
                        if(allRest) {
                            ret.add(KonaWord.new(0, gati, dur, tani));
                        } {
                            ret.add(KonaWord.new(dur, gati, karve, tani));
                        };

                        dur = dur - dur;
                    } {
                        if(allRest) {
                            ret.add(KonaWord.new(0, gati, dur, tani));
                        } {
                            ret.add(KonaWord.new(9, gati, karve, tani));
                        };
                        dur = dur - 9;
                    };
                }
            );
        };

        if(ret.size==1) {
            ^ret[0];
        } {
            ^ret;
        };
    }

    //  combineSimilar
    //  @aCol       A KonaTime containing KonaWords.
    //  @alMax      Maximum number of items to combine
    //  @avMax      The maximum size for a combination
    //  @prob       Probability of combination
    //
    //  Method to combine identical adjacent KonaWords within a KonaTime
    combineSimilar {|aCol, alMax, avMax=9, aProb=1|

        var col;                    //  Collecion to modify
        var lMax;                   //  Maximum string length to combine
        var vMax;                   //  Maximum value of a combination
        var start, middle, newMiddle, end;      //  Temporary storage
        var n;                      //  Item lookahead number
        var i;                      //  Iterator variable
        var y;                      //  Next Iterator variable value
        var func;                   //  Function to do most of the work
        var prob;                   //  Probability the function will occur

        col = aCol;
        prob = aProb;

        n=1;
        i = 0;

        // use argument length if provided
        lMax = alMax ?? {col.size};
        // set combo max
        vMax = avMax;

        func = {

            //Reduce n to the index of the last matching value
```

```
            n = n-1;
            //If this is not the first item/string to be evaluated
            if(i>0) {
                //Store the sub-array that proceeds the string/items
                start = KonaTime.newFrom(col[0..i-1], tani);

                //Store the next index to evaluate.
                y = start.size+1;
            } {
                //Else, this this is the first item/string to be evaluated
                //There are no values before the first item
                start = KonaTime.new(tani);
                //Store next index to use: 1
                y = 1;
            };

            //The string of matching values
            middle = KonaTime.newFrom(col[i..i+n], tani);

            if(middle.size>4) {
                prob=1
            };

            if(prob.coin) {
                newMiddle = this.combine(middle);
            } {
                newMiddle = middle;
            };

            //If all elements have been evaluated or combined
            if(middle.includes(col[col.size-1]).not) {
                //Use all elements after the middle
                end = KonaTime.newFrom(col[i+n+1..col.size-1], tani);
            } {
                //Else. There are no values after the last element
                end = KonaTime.new(tani);

            };

            //Combine three sections, summing the middle items
            col = start ++ newMiddle ++ end;
            //Reset n
            n = 1;
            //Set the next index to start as;
            i = y;
        };


        //Evaluate the whole collection
        col.size.do {
            if(col[i].class==KonaTime) {
                col[i] = this.combineSimilar(col[i], prob:0.85);
            } {

                //If there are trailing elements
                if(col[i+n]!=nil) {
                //If a value is followed by an identical value,
                //  and current string length is within bounds;
                    if( ( (col[i].val == col[i+n].val) || (col[i].word==['Ta'] &&
col[i+n].word==['-']) )  && (n<lMax) && (col[i..i+n].jatis <= vMax)) {
                        if(col[i-1]!=nil) {
                            if(col[i].word==['Ta'] && (col[i-1].word==['-']) ) {
                                func.();
                            } {
                                //Extend string length
                                n = n+1;
                            };
                        } {
                            func.();
                        };
                    } {

                        //Else combine all identical adjacent items.
                        func.();
                    }
                } {
                    //If there are no more trailing items, combine those stored.
                    func.();
                };
            };
        };

        ^col;
    }

    //  atDensity
    //  @item       Item to alter density of
    //  @density    Density multiplier
    //
    //  Method to return the word at a new density (same duration, more notes).
    //  E.g. Ta - Ka - Di - Mi -  @ density 2 becomes TaKaDiMiTaKaJuNa

    atDensity { |aKonaItem, density|
        var item;
        var newJatis;
        var newKarve;
        var newWords;
        var newPhraseFunc;
        var ret;

        item = this.fillOut(aKonaItem, true);
        Post << "density: " <<  density << "\n";

        newPhraseFunc = {
            this.phraseAtGati(item, item.gati, 1/density)
        };

        if(item.class==KonaWord) {

            newJatis = (item.jatis*density).max(1);
            newKarve = (item.karve*(1/density)).min(item.matras);
            //If the result can be a single word
            if(newJatis<=9) {
                //If the adjustment results in a non Integer
                if(newJatis%1!=0) {
                    //A 1 syllable word with a matching duration is returned
                    ^KonaWord.new(1, item.gati, item.jatis, tani)
                } {
                    //Otherwise a new word with adjusted density is returned
                    ^KonaWord.new(newJatis, item.gati, newKarve, tani)
```

```
                };
            } {
                //If the results require multiple words, create them
                newWords = newJatis/item.jatis;
                ^KonaTime.fill(newWords, {newPhraseFunc.()}, tani)
            };
        } {

            //If the item is a collection of words, call this method on each of them
            ret = KonaTime.new(tani);

            item.size.do { |i|
                ret.add(this.atDensity(item[i], density))
            };

            ^ret
        };
    }

    //  randomAtDensity
    //  automation of the atDensity method
    //
    //  @aKonaItem  Item to be manipulaed
    randomAtDensity {|aKonaItem|
        var store;          //Storage for invalid density multipliers
        var min, max;       //Min and Max multiplier values
        var val;            //Array of multipliers from min to max
        var wVal;           //Array of weights for multipliers;
        var count;          //Counter for the value of greatest reduction.
        var xGatiFunc;      //Function to calculate the xGati, for gati=4 exception.
        var xGati;          //Gati of object

        store = List[];
        val = List[];
        count = 1;
        xGatiFunc = {
            xGati = aKonaItem.gati;
            if(xGati==4) {xGati=2}; //Exception for caturasra gati
        };
        xGatiFunc.();
        //Find largest numbers of jatis in the object
        if(aKonaItem.class==KonaWord) {
            min = aKonaItem.jatis;
        } {
            min = aKonaItem.greatestJatis;
        };

        //Calculate the smallest possible multiplier
        while({count*min>1}, {
            count = 1;
            count = count/xGati;
            xGati = xGati*2;
        });
        //Reset the xGati;
        xGatiFunc.();
        min = count;

        //Calculate the greatest possible multiplier
        max = aKonaItem.speed/(0.5/aKonaItem.gati);

        //Calculate all values between max and min.
        val.add(max);
        while({(max/xGati)>min}, {
            val.add(max/xGati);
            xGati = xGati*2;
        });
        val.add(min);
        val = val.asArray.reverse;
        //Convert whole numbers into Integers
        val.size.do { |i|
            if(val[i]==val[i].asInteger) {
                val[i] = val[i].asInteger
            }
        };
        //Remove multiplier of 1 if others are possible
        if(val.size>1) {val.remove(1)};
        //Create weights from multiplier values.
        //  Positive multipliers are given the heaviest weight,
        //  with greatest weight given to those
        // >=2. Multipliers below 0.5 are given the lowest weightings.
        wVal = Array.newClear(val.size);
        val.size.do { |i|
            if(val[i]<1) {
                wVal[i]=0.75;
            } {
                wVal[i]=1
            };
            if(val[i]<0.5 || (val[i]>2)) {
                wVal[i] = wVal[i] - 0.25;
            };
        };

        val = val.wchoose(wVal.normalizeSum);
        //Return the altered item
        ^this.atDensity(aKonaItem, val.asInteger);
    }

    //  extendJati
    //  Method to extend the jati of a given Kona Word.
    //  e.g. TaKaDiMi with index [1] extended by 1 becomes TaTa-Ta
    //
    //  @aKonaWord  Word to manipulate
    //  @aIndex     Index of jati to ext
    //  @aExt       Number of jatis to extend the given jati by

    extendJati {|aKonaWord, aIndex, aExt|
        var ret;    //  Variable to return output;
        var start;  //  Jatis before the extended jati
        var middle; //  Extended Jati
        var end;    //  Jatis after extended jati

        //If there is 1 jati, extension is impossible so return the input.
        if(aKonaWord.jatis==1) {
            ^aKonaWord
        };

        //If overextending...
        if( (aIndex+aExt+1)>aKonaWord.jatis,
```

```
                {^"Trying to extend tooo much..."}
            );

            //If the first jatis is being extended, there are no prior jatis to store
            if(aIndex==0) {
                start = nil;
            } {
                //Else use all jatis before the extended jati
                start = KonaWord.new((0..aIndex-1).size, aKonaWord.gati, aKonaWord.karve, aKonaWord.tani);
            };

            //Jati to extend
            middle = KonaWord.new(1, aKonaWord.gati, 1+aExt*aKonaWord.karve, aKonaWord.tani);

            //If the extension will take up all of the word duration
            if((aIndex+aExt)==(aKonaWord.jatis-1)) {
                //Have no following jatis
                end = nil;
            } {
                //Else use the jatis following the extension
                end = KonaWord.new((aIndex+aExt+1..aKonaWord.jatis-1).size, aKonaWord.gati, aKonaWord.karve, aKonaWord.tani);
            };

            //Create a KonaTime, add the new KonaWords and return it
            ret = KonaTime.new(tani);
            [start,middle,end].do { |item, i|
                if(item!=nil, {ret.add(item)});
            };
            ^ret;

        }

        //  randomExtendJati
        //  Automation of the extendJati Method
        //
        //  @aKonaItem  Object to manipulate

        randomExtendJati{|aKonaItem|
            var itemSize;
            var index;
            var max;
            var count;
            var ret;
            var chance;
            var success;

            if(aKonaItem.class==KonaTime) {
                ret = KonaTime.newClear(aKonaItem.size, tani);
                success = Array.newClear(aKonaItem.size);
                chance = 1/aKonaItem.size;
                aKonaItem.size.do {|i|
                    if(chance.coin) {
                        ret[i] = this.randomExtendJati(aKonaItem[i]);
                        chance = chance/2;
                        success[i]=true;
                    } {
                        ret[i] = aKonaItem[i];
                        success[i]=false;
                    }
                };
                if(success.every { |item, i| item.not }) {
                    success = aKonaItem.size.rand;
                    ret[success] = this.randomExtendJati(ret[success])
                };
                ^ret;
            } {
                index = (aKonaItem.jatis-1).rand;
                max = (aKonaItem.jatis-1 - index);
                count = (1..max).choose;

                ^this.extendJati(aKonaItem, index, count);
            };
        }

        //  muteJati
        //  Method to mute a jati of a given Kona Word.
        //  e.g. TaKaDiMi with index [1] muted becomes Ta-Taka
        //
        //  @aKonaWord  Word to manipulate
        //  @aIndex     Index of jati to mute
        muteJati {|aKonaWord, aIndex|
            var start;
            var middle;
            var end;
            var ret;

            if(aIndex>=aKonaWord.jatis,
                {^"overstretched yourself a bit..."}
            );
            //If the first jatis is being extended, there are no prior jatis to store
            if(aIndex==0) {
                start = nil;
            } {
                //Else use all jatis before the extended jati
                start = KonaWord.new((0..aIndex-1).size, aKonaWord.gati, aKonaWord.karve, aKonaWord.tani);
            };

            //Muted Jati
            middle = KonaWord.new(0, aKonaWord.gati, aKonaWord.karve, aKonaWord.tani);

            //Following Jatis
            if(aIndex==(aKonaWord.jatis-1)) {
                end = nil;
            } {
                end = KonaWord.new((aIndex+1..(aKonaWord.jatis-1)).size, aKonaWord.gati, aKonaWord.karve, aKonaWord.tani)
            };

            //Combine Jatis into a new KonaTime
            ret = KonaTime.new(tani);
            [start,middle,end].do { |item, i|
                if(item!=nil, {ret.add(item)});
            };
            ^ret;
        }
```

```
    //  randomMuteJati
    //  Automation of the muteJati Method
    //
    //  @aKonaItem  Object to manipulate
    randomMuteJati {|aKonaItem|
        var index;
        var ret;
        var iter;
        var chance;
        var success;

        if(aKonaItem.class==KonaTime) {
            ret = KonaTime.new(tani);
            success = Array.newClear(aKonaItem.size);
            chance = 1/aKonaItem.size;
            iter = aKonaItem.size-1;
            aKonaItem.size.do { |i|
                if(chance.coin) {
                    ret.add(this.randomMuteJati(aKonaItem[iter]));
                    chance = chance/5;
                    success[i] = true;
                } {
                    ret.add(aKonaItem[iter]);
                    success[i] = false;
                };
                iter = iter - 1;
            };
            if(success.every { |item, i| item.not }) {
                success = aKonaItem.size.rand;
                ret[success] = this.randomMuteJati(ret[success]);
            };
            ret = ret.reverse;
            ^ret;
        } {
            index = aKonaItem.jatis.rand;
            ^this.muteJati(aKonaItem, index);
        };
    }

    //  densityJati
    //  Method to alter the density of a jati in a given Kona Word.
    //  e.g. Ta Ka Di Mi with index [1], density 2 becomes Ta TaKa Ta Ka
    //
    //  @aKonaWord  Word to manipulate
    //  @aIndex     Index of jati to mute
    //  @aDensity   Density to alter by
    densityJati {|aKonaWord, aIndex, aDensity|
        var start;
        var middle;
        var end;
        var ret;

        if(aIndex>=aKonaWord.jatis,
            {^"overstretched yourself a bit..."}
        );

        //If the first jatis is being altered, there are no prior jatis to store
        if(aIndex==0) {
            start = nil;
        } {
            //Else use all jatis before the extended jati
            start = KonaWord.new((0..aIndex-1).size, aKonaWord.gati, aKonaWord.karve, aKonaWord.tani);
        };

        //Density altered jati;
        middle = this.atDensity(KonaWord.new(1, aKonaWord.gati, aKonaWord.karve, aKonaWord.tani), aDensity.max(1));

        //Following Jatis
        if(aIndex==(aKonaWord.jatis-1)) {
            end = nil;
        } {
            end = KonaWord.new((aIndex+1..(aKonaWord.jatis-1)).size, aKonaWord.gati, aKonaWord.karve, aKonaWord.tani)
        };

        //Combine Jatis into a new KonaTime
        ret = KonaTime.new(tani);
        [start,middle,end].do { |item, i|
            if(item!=nil, {ret.add(item)});
        };
        ^ret;

    }

    //  randomDensityJati
    //  Automation of the densityJati Method
    //
    //  @aKonaItem  Item to manipulate
    //  @aRec       Chance of recursion
    randomDensityJati {|aKonaItem, aRec=0.5|
        var store;
        var index;
        var val;
        var wVal;
        var max;
        var ret;
        var chance;
        var success;

        if(aKonaItem.class==KonaTime) {
            ret = KonaTime.new(tani);
            success = Array.newClear(aKonaItem.size);
            chance = 1.5/aKonaItem.size;

            aKonaItem.size.do { |i|
                if(chance.coin) {
                    ret.add(this.randomDensityJati(aKonaItem[i], 0));
                    chance = chance/2;
                    success[i] = true;
                } {
                    ret.add(aKonaItem[i]);
                    success[i] = false;
                };
            };

            if(success.every { |item, i| item.not }) {
                success = aKonaItem.size.rand;
                ret[success] = this.randomDensityJati(ret[success], 0);
```

```
            };

        } {
            store = List[];

            index = aKonaItem.jatis.rand;

            max = aKonaItem.speed/(0.5/aKonaItem.gati);
            val = (2..max);

            //Remove unacceptable densities (for this gati)
            val.do { |item, i|
                if((aKonaItem.speed*(1/val[i])%
(0.5/aKonaItem.gati)).round(0.001)!=0,
                    {store.add(i)}
                );
            };
            val.removeAtIndexes(store);
            //If the item is not alterable
            if(val.size==0) {
                ^aKonaItem
            };
            //Select an expansion
            wVal = Array.newClear(val.size);
            val.size.do { |i|
                if(val[i]>2)
                    { wVal[i] = 0.5 }
                    { wVal[i] = 1};
            };

            wVal = wVal.normalizeSum;
            val = val.wchoose(wVal);
            Post << "val: " <<  val << "\n";


            ret = this.densityJati(aKonaItem, index, val);
        };

        if(aRec.coin) {
            ^this.randomDensityJati(ret, (aRec/2));
        } {
            ^ret
        };

    }

    //  permutePhrase
    //  Method to return a permutation of a KonaTime phrase
    //
    //  @aKonaTime      KonaTime to be permuted
    //  @permutation    Optional permutation specificiation
    //  @seed           Optional seed for random selection;
    permutePhrase { |aKonaTime, permutation, seed|
        var phrase;
        var partition;
        var permuteNum;

        if(seed!=nil) {
            thisThread.randSeed=seed
        };

        if(aKonaTime.class==KonaTime) {
            phrase = aKonaTime;
        } {
            phrase = KonaTime.newFrom([aKonaTime], tani)
        };

        //Permutations 0 and size.factorial return the input, so they are ignored.
        permuteNum = permutation ?? {if(phrase.size==1) {1}
{(1..phrase.size.factorial-1).choose}};

        ^phrase.permute(permuteNum.asInteger);
    }

    //  partitionWord
    //  Method to partition and permute a KonaWord
    //  E.g. KonaWord(5,4,1) could be returned
    //      as KonaTime[KonaWord(2,4,1),KonaWord(3,4,1)];
    //
    //  @aKonaWord  Word to partition
    //  @seed       Optional seed value for randomness
    partitionWord {|aKonaWord, min=2, aMax, seed|
        var partition;
        var int;
        var ret;
        var max;

        if(seed!=nil) {
            thisThread.randSeed=seed
        };

        int = aKonaWord.jatis;

        max = aMax ?? {if(int>9) {9} {int}};

        partition = this.randomPartition(int, min, max, true, seed);

        partition = this.randomPerm(partition, seed:seed);

        ret = this.partsToWords(partition, aKonaWord.karve, false);

        ^ret
    }


    //  randomPartitionMutate
    //  Method to (possibly) partition a word and (definitely) mutate it
    //
    //  @aKonaWord  Word to part/mutate
    //  @aChance    Chance that partitioning will occur
    //  @seed       Random Thread seed
    randomPartitionMutate {|aKonaWord, aChance=0.5, seed|
        var ret;
        var chance;

        chance = aChance;
        if(seed!=nil) {
            thisThread.randSeed=seed
        };
```

```
        //Decision: Partition word or not
        if(chance.coin) {
            ret = this.partitionWord(aKonaWord, seed:seed);

        } {
            ret = KonaTime.newFrom([aKonaWord], tani);
        };


        //Mutate the phrase for added interest
        ret = this.mutatePhrase(ret);
        ^ret
    }

    //  addSuffix
    //  Method to add a densly articulated suffix to the end of a phrase
    //
    //  @aPhrase    Phrase to alter
    addSuffix {|aPhrase|
        var phrase;         //Phrase to be altered and returned
        var iter;           //Iterator
        var suffixMatras;   //Number of matras in the suffix
        var suffixTemp;     //Temporary storage for items to be included in the
suffix
        var densMax;        //The maximum possible density;
        var densities;      //Density multipliers for suffix parts
        var temp;           //Temporary storage for mutations.

        phrase = aPhrase.deepCopy;
        densMax = {|item| item.speed/(0.5/item.gati); };

        if(phrase.class!=KonaTime) {
            phrase = this.partitionWord(phrase, aMax:(phrase.matras/4));
        } {
            if(phrase.size==1) {
                phrase = this.partitionWord(phrase, aMax:(phrase.matras/4));
            };
        };

        iter = phrase.size-1;
        suffixMatras = 0;

        //Add up item matras from the end until a sufficient number is found
        while({suffixMatras < (phrase.matras/4)}, {
            suffixMatras = suffixMatras + phrase[iter].matras;
            iter = iter - 1;
        });

        //Store those KonaItems to be used for the suffix.
        suffixTemp = phrase.select({|item, i| (iter+1..phrase.size-
1).includes(phrase.indexOf(item)) });
        densities = Array.newClear(suffixTemp.size);

        //Store suitable densities
        if(suffixTemp.size==1) {
            densities[0] = 4
        } {

            suffixTemp.do { |item, i|

                case
                    {i==0}                          {densities[i] = 2}
                    {i==(suffixTemp.size-1)}        {densities[i] = [2,4].choose}
                    {true}                          {densities[i] = [2,4].choose}
            };
        };


        //Alter densitiy of items
        densities.size.do { |i|
            //Protect against going too fast.
            if(suffixTemp[i].karve/densities[i] < densMax.(suffixTemp[i])) {
                densities[i] = densMax.(suffixTemp[i]);
            };

            temp = this.atDensity(suffixTemp[i], densities[i]);
            temp = this.randomMuteJati(temp);

            if(i==(suffixTemp.size-1)) {
                "hi".postln;
                temp = this.randomDensityJati(temp);
            };
            phrase[iter+1+i] = temp;

        };

        ^[phrase, iter+1];
    }

    //  fillOut
    //  Method to 'fill in' konaWords/Times
    //      e.g. a KonaWord 1,4,3 (0.75) gets turned into 3,4,1.
    //
    //  @aKonaItem      Item to be altered
    //  @aOnlyUneven    Boolean, if only uneven parts should be filled in.
    fillOut {|aKonaItem, aOnlyUneven=false|
        var ret;
        var mult;
        var jatis;
        var action;
        var i;
        var temp;

        if(aKonaItem.class==KonaWord) {
            mult = aKonaItem.speed/(1/aKonaItem.gati);

            block {|break|
                while({mult.round(0.0001 )%1!=0}, {
                    mult = (mult*2);
                    if (i==100) { break.value(999)};
                });
            };

            if(mult.asInteger.odd && (mult!=1)) {
                action = true;
            } {
                if(aOnlyUneven) {
                    action = false;
                } {
                    action = true
```

```
                };
            };

            if(action) {
                jatis = mult*aKonaItem.jatis;

                if(mult>9) {
                    ret = this.partsToWords(this.randomPartition(jatis, notSize:true), aKonaItem.karve/mult, false);
                } {
                    ret = KonaWord.new(jatis, aKonaItem.gati, aKonaItem.karve/mult, tani)
                };

            } {
                ret = aKonaItem.deepCopy;
            };
        } {
            ret = KonaTime.new(tani);
            aKonaItem.size.do { |i|
                temp = this.fillOut(aKonaItem[i], aOnlyUneven);
                ret.add(temp);
            };
        };
        ^ret
    }


    //  makePostMora
    //  Method to convert a phrase so that it's suitable after a mora,
    //      i.e. that it starts with a strong long beat
    //
    //  @aKonaItem      Phrase to be altered
    makePostMora {|aKonaItem|
        var phrase;             //Phrase being altered
        var sectionMatras;      //Matras in the section being overridden
        var i;                  //Iterator
        var index;              //Index used when calculating makeup Matras
        var hitMatras;          //Duration of the initial beat in matras
        var makeupMatras;       //Duration of the material being made up for
        var hit;                //KonaWord for the initial beat
        var makeup;             //KonaWord for the makeup
        var ret;                //KonaTime to return the phrase

        //If phrase is a KonaWord, convert it to a KonaTime
        if(aKonaItem.class==KonaTime) {
            phrase = aKonaItem.deepCopy;
        } {
            phrase = KonaTime.newFrom([aKonaItem], tani);
        };

        sectionMatras = 0;
        i = 0;
        //Create initial hit
        hitMatras = aKonaItem.gati;
        hit = KonaWord.new(1, phrase.gati, hitMatras, tani);

        //Count how many parts of the phrase will be overridden
        while({sectionMatras<hitMatras}, {
            sectionMatras =  sectionMatras + phrase[i].matras;
            i = i + 1;
        });
        //Calculate the duration of the overridden section that needs to be recreated
        makeupMatras = sectionMatras - hitMatras;
        //Generate makeup material.
        if(makeupMatras!=0) {

            if(phrase.size==1) {
                index = 0
            } {
                index = i-1;
            };
            makeup = this.vSarvaPhrase(makeupMatras)
        };
        //Add and return all items
        ret = KonaTime.newFrom([hit], tani);
        if(makeup!=nil) {ret.add(makeup)};
        if(phrase.size>1) {
            ret.addAll(phrase[i..phrase.size-1]);
        };

        ^ret;
    }
}
```

```
+ ArrayedCollection{

      removeAtIndexes{arg indexes;
                indexes.sort.reverse.do{arg index;
          this.removeAt(index);
      }
  }
}
```

Appendix E

Project Log

Updated Project Log, please refer to my Interim report for prior work.

12/1/09

Re-wrote SynthDef to use PV_Playbuf for time stretching, also found very useful DetectSilence UGen.
Solved playback from FFT analysis files.
Had to separate the file writing into a different routine from the SoundFile creating and buffer allocation, then had to 'call' each buffer (that had read from an FFT file) to get it to work properly.

Changed Konakkol Class to send messages to the server via a NetAddr object instead of creating Synth instances. This allows playback through any Synth sound, or sending messages via MIDI by making changes to the OSCresponder. Will be fun to try the system through drums, drumkits, beatbox instruments, melodic instruments etc...

13/1/09

Gutted Konakkol class of methods that return different speeds and Gopucca Method. Implementing arguments when creating instances so that Konakkol class instances have a defined form, which is returned as a Routine. This allows concatenation of Routines.

Will implement methods to return the instance in another form so that variations can be created. e.g. y = x.atDoubleSpeed.
Implemented Method atSpeed which returns a new Konakkol object with the same phrase, but with a specified speed.

Realised that I don't need a Gati argument for Konakkol class as Tisra Gati (3 per beat) is just a speed of 1/3. But may be more intuitive to have one that just alters the speed.

14/01/09

Started writing KonaPhrase class, (re-named Konakkol class KonaWord).
Takes an indefinite number of arguments, will be KonaWords.
Will be used when a KonaWord (e.g. Ta ka di mi) is altered and requiring new syllables (e.g. Ta - Ta Ka).
Will also be used as motivic cell.

15/01/09

Made KonaPhrase a subclass of List, to ease adding items and creationg. Tried Array but had problems using instance variables.
Overwrote add method to only accept KonaWord objects
Overwrote ++ concatenation method to return a new KonaPhrase

16/01/09

Installed wslib for AutoBackup, SC Crashing at the moment.

Added dur and rout methods to KonaPhrase.
Added an at method to KonaWord for [index] access to jatis in the word

19/01/09

Added phrase method to konaphrase which returns all the words in the phrase in an array of strings.

Read into sarvalaghu patterns with gati != chatusra (4). Wrote a method return all possible combinations to fill a given time period with values with a minimum size of 2. E.g. input 6 returns [6, [2, 4,], [3,3], [4,2]]

developed ideas on structure of whole program

Met with Nick Collins. Discussed structure of program
Discussed class design
Decided not to use OSC messages because of latency
should use bundles for rock solid timing
auto call init method rather than set up in new method

Task is to complete structure classes
Nick suggested using a generic 'duration of time' class rather than specific ta etc
I feel that the naming will still be useful, especially when printing out compositions, esp for calculations
Considering generic class with subclasses just for name.

22/01/09

Working on algorithm to return all possible combinations of groupings for a given grouping.

E.g. 5 would return [5, [3,2], [2,3]]

Hard choices about whether or not to include 1.
There are phrases that are more like 1 1 rather than 2
e.g. |na na din - | |din - jo no|
As well as phrases such as
  |na tom tom ta |tom tom tom ta|

Including 1s makes the algorithm very tricky, phrases of 1 1 1 1 1 1 1 are also unlikely.

Thoughts
-Phrases could be altered later, represented as 2, but sound like 1 1
-Or Altered to be 1s, will have to change the sounds anyway.

23/01/09

Continuing work on partition algorithm. Tried implementing zs1 algorithm, works in C but not sc....

Going back to own method of diving into parts of two.
Almost there, however, returning an array with all answers + an element with all answers

24/01/09

Successfully Implemented ZS1 algorithm, much faster. However with larger numbers it still runs noticeably slower in SC than C.
Not completely sure how it works, makes modification difficult. Instead of only generating partitions with certain values (between 2 and 9)
-I am simply removing them afterwards
-Also generates all permutations of partitions.

27/01/09

Decided not to have a distinct tala class, but to have a time period class which can return duration
in beats and talas.
KonaTime takes the tala as an argument, a literal array of the beats; Adi Tala #[4,2,2]

Considered using KonaTime instead of KonaPhrase, but decided that KonaPhrase is not a time duration class, it is a musical block class with musical permutations- simply a class for concats of KonaWords

30/01/09

Started KonaTani class.
Has following variables

```
var <laya; // Tempo
var <tala; // Beats in cycle
var <gati; // 'Default' sub-divisions per beats
var <otherGatis; // Sub-divisions to change to
var <eduppu; // Starting/strong beat
var <gen; // Material generator

var <duration; // Duration in minutes
var <totalTalas; // Total number of cycles
```

A rough duration can be passed in by the user, a more exact duration is calculated by fitting maximum number of talas into the given duration, and multiplying them by the duration of each tala.

02/02/09

Upgraded KonaTime class to replace KonaPhrase;

Moved synth generation from an OSC responder to the KonaWord class.
Would still like to be able to set up generic output, but maybe just have Konakol synth, kanjira synth, and MIDI out.
Tried to have synth buffers etc as classvars set up with InitClass, but couldn't
get it to work,had to use instance variables, not optimal.

Analysing Trichy Solo from Nelson DVDs.

03/02/09

Coding re-think.

Moving all alteration methods (e.g. atDensity etc) from KonaWord to KonaTime to avoid duplication, allows manipulation of KonaTime objects rather than just words, or call downs.

Adapting toGati method (change a word/time to a new gati) proved troublesome.
No problem for a single KonaWord, but a KonaTime provided a new problem. How to generate phrases in the new gati with the same relative values. e.g. 0.25, 0.25, 0.125, 0.125 in triplets should be 0.33, 0.33, 0.166, 0.166
Changed KonaWord from having Gati of any value to being limited to 4,3,5,7,9, but then using expansions (multipliers); E.g. 0.125 is gati 4, expansion 2

Added a postWord method to KonaWord which prints in the following format

```
[ Ta, Ka, Di, Mi ]
[ 0.33, 0.33, 0.33, 0.33 ]
[ [ 1, 12 ], [ 1, 12 ], [ 1, 12 ], [ 1, 12 ] ]
```

04/02/09

New priority: Get a working version, then improve it rather than going for gold

Reading material for Sarvalaghu Generation.
Hulzen and Prakash for non-Adi Tala material + adapting
Nelson + Transcriptions for Adi-Tala

Will have to adjust playback method. a word could be all rests, or all resonant.
  e.g. ta-ka played rest rest, or ta din

Page 7 of Analysis 3, shows that Mora design is open for development....

05/02/09

Completed basic analysis of last of 5 solos.
Beginning Vilamba Sarvalaghu generation..

Looking at the 5 solos I have, it is clear that approaches vary but there are some things in common.

Some solos (mani, raghu) have very obvious grooves/motifs etc from the start, while others (sankaran, murthy) play more generic grooves for the tala, there is repetition of ideas, but it's less straightforward.

Some solos have constant phrase durations (sankaran)

Thoughts:

Micro level
they are all very similar : focus on 2 pulses per beat.
Macro level,
they always feature some small variations
suffixes are common

Had to fix problems with accessing routines from KonaWord and KonaTime.

Was overriding getter methods using the variable name within the method
e.g. getter rout { rout = 4+5, ^rout} was messing things up, didn't declare
new variables. Testing statistical generation of Vilamba Sarvalaghu,
beats 1-4 of adi tala (2 kalai, so 8 beats) Data used :
Adi Tala 2 kallai, first 4 beats, each set of 4 represents 1 beat

```
y = [
100, 37.5, 87.5, 68.75,
93.75, 12.5, 100, 25,

100, 80, 100, 13,
100, 13, 100, 6,

100, 0, 73, 53,
100, 0, 100, 22,

89, 66, 100, 22,
100, 30, 80, 30
];
```

07/02/09

Read Sangeetha Akshara Hridaya, replaced gatiexpansion in KonaWord with karve,
which defines how many gati a jati is worth.

Transcribed grooves from Ultimate Guru DVD for examples of different Gatis,
and non-adi talas
Problems...
Only tisra gati example is played in groupings of two ([1,2],[3,1],[2,3],[1,2],[3,1],[2,3])
This is fine, and shows variety, but no example of 123 123 123 123

Considering just working with tanis that start in chatusra, can't find
many/any examples of those that don't.
Possibly limit to adi tala, would make life easier.

08/02/09

Big problems....
Generating 6+ KonaWords seemed to crash SCLang
Often get grey count errors

sometimes will generate but word will be
3.6351256066833e-273
or similar.
Once this happens, it happens to all new instances.

Trying to play a KonaTime routine would only play once.

09/02/09
Solved problem, was because KonaWord had [slot], thanks Jordanous for
pointing out I was doing something clever without realising it

Finally get to properly test Trichy Sankaran statistical model.
Results are ok, it feels that context wasn't properly taken into account.'
These results might be suitable for mid/late Vilamba, but not introduction,
as they are quite busy although, some people do start this busy,
while others are much more sparse

The flaw of repeated generation for many cycles is that there is little underlying theme, just statistical matching.
Going to try an approach of starting with a basic pattern and then mutating it.

12/02/09

Continuing work on Vilamba Sarvalaghu, alternating with work on moras.

Tried phrase length generation on a tala of 12 beats, gati 4, tempo 120, (partition 40) partitioning crashed SC...
Trying to implement ZS2 Algorithm in the hope that it will be faster.
Successfully implemented ZS2. Made modifications to allow maximum and minimum values to be set.

Now working on Moras

Improved random Mora method.
As I am unable to find instances where a gap duration is a fractional value of the statement (e.g. s = 5, g = 3.5), i.e they are generally made up of different numbers of jatis of the same duration. A .floor method is used to get a round number for the gap duration, and an offset that precedes the mora is generated if there are unused beats from the given duration. Mora generation now takes into account Nelsons (2008 p 23) observation that if a mora statement is shorter than 5 pulses the gap will nearly always be at least 2 pulses

13/02/09

Resuming work on VSL. Now testing ZS2 for generation of partitions.
Turns out that it's not the ZS algorithms running slowly in SC. It's the time needed to add the results to the lists.
Maybe calculating P(n) (number of partitions) and creating an array of that size rather than using a List would be more efficient.
Turns out it's NOT adding it to the list, but the permutations that takes ages.
Changed the way minimum values were taken care of. Instead of using two loops, items are only added if all parts are > min
Divided allParts into two methods, allParts and allPerms. allPerms takes care of permutations.

Worked on getting playback functional again. Re-recorded syllables, added more to the vocab and re-did scpv analysis.

16/02/09

PV Analysis problem solved, separated various actions into different loops and gave enough time to analysis synth. Now need to incorporate time stretching.
Time stretching achieved using 3.min(1/speed) for the rate

VSL.
One problem with current phrase length generation is that it cuts at absolute values, e.g. 119 bpm gati 4, gives 16 jati phrase while 120 bpm gives 32 jati phrase, maybe some kind of blend would be better....

18/02/09

migrated remaining manipulation methods from KonaWord to KonaGenerator
Adapted atDensity method to work for KonaTime and KonaWord

19/02/09

Wrote combine method which creates new KonaItems with the number of jatis of a
given collection (e.g. ta ki ta + ta ka) returns da di gi na dom)
Worked on algorithm to search for adjacent identical values in an array and combine them
(e.g. [2,2,2,3,3] becomes [6,6])
This will be used stochastically for sarvalaghu generation

27/02/09

Included combine method in SarvaPhrase generation
Updated postWord methods to print evenly e.g

```
[ Ta      , Ka      ]
[ 0.5     , 0.5     ]
[ [ 1, 8 ], [ 1, 8 ] ]
```

28/02/09

Working on vilamba sl...
Implemented new method removeThoseContaining that removes any partitions that contain
a given value/set of values. Weights can be supplied for the values.

Had a bug with combination method, if all elements were combined earlier than
(1 before the end) then extra items appeared, Fixed now

01/03/09

Fixed .play methods on KonaTime and KonaWord
Added play to KonaTani
Created a Tala clapping routine usng Thor's clapping synthdef.
Problems with trying to play Tala + Tani routine's in sync

02/03/09
Working on manipulation methods.
Fixed Density method, wasn't functioning properly
Working on extend jati method, to extend a particular jati in a word e.g. takadimi with
[1] extended becomes ta ta - ta
Need to have an exception for this example so that it becomes ta lan - ga
Extend Jati method complete

Working on Mute jati method
Completed mute jati method

Fixed bug in vSarvaPhrase method where 10% of the time it would bugger up.

Quickly implemented densityJati method to alter the density of a jati within a KonaWord

Working on a phraseMutation method that works through the elements of a given phrase and
potentially mutates them using the recently made methods.

04/03/09

Fixed routing issue thanks to nick.

Interesting problem encountered when working on automation of atDensity method.
Difficult to determine what the maximum speed should be.
Carnatic music talks of having 4 speeds (prakash), but obviously this is dependant on tempo. E.g. @ 160 bpm it's hard to play at 4th speed
This is also dependant on the player.
Human haptic rates can be used, but alternative techniques (chunking) can be used to bypass these.
Current formula is speed/(0.5/col[i].gati). E.g. a 4,4,1 KonaWord (Ta Ka Di Mi) can go up to 4,4,0.5. This seems to work reasonably well, although creates discrepencies. E.g. At the same tempo 9 can go up to 0.0555555555555556 while 4 can go to 0.125

05/03/09

Fixed all mutation methods to work with mutatePhrase automation
Could add recursion.. need to test on materials to judge output

06/03/09

Added a permutePhrase method which can return either a specified or random permuation of a KonaTime

Working on randomMora method, updating to handle KonaItems.
Solving the problem of minimum gap sizes.
Looking for maximum gap sizes.
Largest in a single mora is 4.
Found a 6 in a compound mora with statements of 22
Found a 8 with statements of 16. But an Arudi rather than mora.
Nelson gives examples (not quotations) with long gap durations
Might not give an upper boundary

Mora Gap minimum is always 0 except in cases where statement<5
However, a gap of 1 is rare/not found.
One example found in mani solo, but it's a gap of 1 caturasra gati, and the statement is exclusively 1/2 or 1/4 chatusra gatis So relatively it is 2.

Going to rule out gap of 1
In the instance that the statement has density doubled, the gap could technically be 1
e.g Ta Ka Di Mi Ta Ka Ju Na (0.125) Ta (0.25)
Going to accept that this will be missed out on (ish)
Will include a % chance to recalculate Mora with 2*duration 0.5*gati

Moras generated that are just 3 * konaword == DULL
Need to make mutation 100% for them...
Same for moras that have same statement and gap length
e.g. ta ta ta ta ta all 0.5

10/03/09

Fixing bugs in randomMora method.
Splitting into sub-methods, moraStatement etc etc
Might be able to make one moraMaterial method with probabilities passed in...
Split Mora generation into sub methods
Also split mutatePhrase methods into sub-methods, each acting as an automated version of

their corresponding methods.

16/04/09

Wrote moraFrom method, builds moras from a given statement, optionally give max duration, gap, offset etc

10/04/09
Added accents to KonaWord + KonaTime
KonaWord given .matras (jatis*karve);
KonaTime given .greatestJatis method, returns the item with the greatest number of jatis (recursive for KonaTimes)
Added speed and gati methods to KonaTime for KonaGenerator compatibility.
KonaGenerator.atDensity newKarve now given .min(aKonaItem.matras) to size increase;
KonaGenerator.randomAtDensity can now handle KonaWords; for minimum size uses .jatis for KonaWords and .greatestJatis for KonaTimes.
Also a big change to the way possible values are calculated. Gone is the complicated method, now the max and min are calculated and all possible values calculated from them, using multiples of the gati.

10/04/09
Fixed routines in KonaTani, now can use .add, .play, and .stop. Resets correctly

10/04/09
Changes to randomExtendJati. Now supports KonaTimes. Loops through objects with a 1/size chance of randomly extending their jati, chance is halved if successful. Returned in a KonaTime.
Changes to randomMuteJati. Now supports KonaTime. Same as randomextendjati. half the chance though.
Same deal with randomDensityJati. Except that recursion has been added, but not for KonaTime items.

11/04/09

Improved weights in mora statement generation.
Split vSarvaPhrase into vSarvaPhraseAuto, which calls vSarvaPhraseLength. Now vSarvaPhrase can be called with a manual phrase Dur.
Re-written moraOffset.
Now if dur > 2*gati, uses vSarvaPhrase, if >1*gati uses 1 syllable word. Otherwise same.
Added word to KonaWord.val, to distinguish ta 0.25 and - 0.25
Changed combine to handle combinations of rests (used to turn into articulated syllables) and one syllable + rests.

12/04/09
RandomMora: StateMin is now (1/5) of duration is there is a gap, and 1/2.85 if there isn't.
RandomPerm: Now simply chooses a number between 0 and partition.size.factorial and returns that permutation.
allPartitions:
Now reads from stored files for numbers > 40.
Also includes an option for max no. unique parts. More efficient than removeGreaterThan, but won't work for numbers > 40. \textsc{So} removeGreaterThan is kept.
randomMora: Efficiency issues now sorted :D
Created moraFrom method. Takes a statement + optional duration, gap, offset.
Fixing partitionWord, use of jatis*karve back to just jatis.
Fixed removeThoseContaining, removed possibility of returning an empty array (all items succesfully removed).

randomMora now split into randomMoraValues method for calculating values.

17/04/09

Fixes to morastatement, wasn't mutating for instances < 9.
fixes to samaCompound mora method, removing nils passed to morapart methods.
Changing moraStatement, will never be just one syllable.
Fixed bugs with randomSamaCompoundMora. Incorrect calculation of offset. Now working for large values, proceeding with testing
of lower values where density alteration takes place.
randomMoraValues alteration: density alteration won't occur if karve will drop below 0.5
randomMora and randomMoraValues now have boolean gap and offset args.
Changed mora methods (moraOffset, moraGap, moraStatement) to take duration in matras rather than number of jatis.
Fixed bug in randomSamaCompoundMora with density alterations: added a check to vSarvaPhrase,
if a float is passed in, the .floor
is created, and a single Ta with the remainder added.
Wrote addSuffix method. Given a phrase it will return a new phrase with the last quarter into a
suffix by increasing the density.
Added karve argument to partsToWords.
Fixed bug with partsToWords not taking karve into account.
Fixed bug with addSuffix not working with konaWords properly.
Wrote and bug fixed fillOut method. Will 'fill out' uneven words for density alteration, e.g. Ta 0.75 becomes ta ki ta 0.25
Wrote makePostMora. Adds a 1 beat hit to the beginning of a phrase, for phrases preceded by a mora
randomMuteJati made to work in reverse, so that first elements less likely to be muted
reverse method overridden in KonaTime
vSarvaPhrase max reduced to gati, including randomMuteJati
Transcribed some basic phrases
Worked on section generation, with basic structure.
Fixed bug in randomSamaCompoundMora, wrong karves being used.
Written basicStructure. a mini tani, heh.
Guarantees in automated methods (randomAtDensity etc) that if no elements are altered, one will be.
Fixed rounding error bug in fillOut
Discovered how to loop material infinitely, accompaniment style.
Transcribed skeleton phrases from real sources

20/04/09

Changed KonaTani:
No longer takes a duration.
Now takes Tala as an array of strings e.g. ["I4", "O", "O"]
Recorded a few examples into Logic via Jack + MIDI
Changed KonaTani talas. "U" can now be "U1" for a quaver clap
added "W" and "R" for wave and rest respectively.

21/04/09

Created computer examples with first runs of necessary methods and transcribed human examples. Played all as MIDI into Logic.