



Digital Commons@

Loyola Marymount University
LMU Loyola Law School

Honors Thesis

Honors Program

5-6-2021

Markov Model Composition of Balinese Reyong Norot Improvisations

Taylor Flanagan
tflanag4@lion.lmu.edu

Robert Rovetti
Loyola Marymount University

Follow this and additional works at: <https://digitalcommons.lmu.edu/honors-thesis>



Part of the [Probability Commons](#)

Recommended Citation

Flanagan, Taylor and Rovetti, Robert, "Markov Model Composition of Balinese Reyong Norot Improvisations" (2021). *Honors Thesis*. 374.
<https://digitalcommons.lmu.edu/honors-thesis/374>

This Honors Thesis is brought to you for free and open access by the Honors Program at Digital Commons @ Loyola Marymount University and Loyola Law School. It has been accepted for inclusion in Honors Thesis by an authorized administrator of Digital Commons@Loyola Marymount University and Loyola Law School. For more information, please contact digitalcommons@lmu.edu.



Loyola Marymount University
University Honors
Program

Markov Model Composition of Balinese ***Reyong Norot* Improvisations**

A thesis submitted in partial satisfaction
of the requirements of the University Honors Program
of Loyola Marymount University

by

Taylor Flanagan

May 8th, 2021

MARKOV MODEL COMPOSITION OF BALINESE *REYONG*
NOROT IMPROVISATIONS

A thesis submitted to
Loyola Marymount University
The Mathematics Department
in partial fulfillment of the requirements
for Graduation with the Bachelor of Science Degree

by

Taylor Flanagan

May 2021

MARKOV MODEL COMPOSITION OF BALINESE *REYONG* *NOROT* IMPROVISATIONS

Senior Thesis by

Taylor Flanagan

Dr. Robert Rovetti, Thesis Director

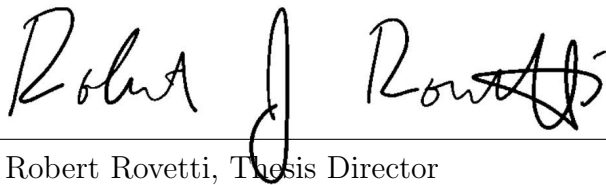
Abstract

Markov models are mathematical structures that model the transition between possible states based on the probability of moving from one state to any other. Thus, given a distribution of starting points, the model produces a chain of states that are visited in sequence. Such models have been used extensively to generate music based on probabilities, as sequences of states can represent sequences of notes and rhythms. While music generation is a common application of Markov models, most existing work attempts to reconstruct the musical style of classical Western composers. In this thesis, we produce a series of Markov chains that model the composition of Balinese *gamelan gong kebyar* improvisations on the *reyong*. This music features distinct rules and limitations. Each of the *reyong's* four players can play only some of the *gamelan's* five tones and must use specific patterns learned only by listening and playing. And yet, the music structure also provides room for ample creativity with improvisation. The model's probability values come from a combination of top-down and bottom-up techniques, making extensive use of Leslie Tilley's work on the grammar of *reyong norot* and example patterns from her concurrent study of musician Dewa Ketut Alit's improvisation. The model outputs MIDI files for audio playback of the constructed songs. Though the model's music lacks some of the improvisational creative quality that humans provide, we find that our model does produce musically interesting *reyong* elaborations that fit within the confines of Tilley's grammar.

Thesis written by

Taylor Flanagan

Approved by



06 May 2021

Dr. Robert Rovetti, Thesis Director

Date



05/06/2021

Dr. Patrick Shanahan, Mathematics Department Chair

Date

Contents

1	Background	1
1.1	Markov Models	1
1.2	Balinese <i>Gamelan</i>	2
1.3	The Grammar of <i>Reyong Norot</i>	3
2	Methods	7
2.1	Overview	7
2.2	Model Details	7
2.3	Probability Estimations	13
2.4	Other Details	13
3	Results	14
4	Discussion	16
5	Acknowledgements	17
	Glossary	18
	References	19
A	<i>Reyong</i> Sample 4 Score	20
B	<i>Reyong</i> Sample 5 Score	23
C	Codebase	28
C.1	Tone.py	28
C.2	Cell.py	29
C.3	MarkovModel.py	32
C.4	ModifiedMarkovModel.py	33
C.5	Song.py	36
C.6	Driver.py	39

1 Background

1.1 Markov Models

Markov chains model the transition between various states in a system that changes over time. The chain itself is made up of discrete states. At every time step, the model can transition from its current state to any other state with a pre-determined and traditionally unchanging probability. The Markov property of the model states that the probability of transitioning to the next state depends solely on the current state. All states prior to the current one are independent from the next state.

Thus, we can specify a Markov chain with three basic inputs: a list of possible states, a probability distribution that gives the likelihood of starting in each state, and a probability of transitioning from each state to every other state, including from the state back to itself. These probabilities can be conveniently written in matrix form, where the row indicates the current state and the column indicates the state to be transitioned to. This is called the transition matrix of the Markov chain.

For example, say we have a Markov model with three states: A, B, and C. The graph of the chain might look something like Fig. 1. Each node represents a discrete state of the model. The directed edges between nodes indicate the probability of moving from state to state. Thus, the probability of transitioning to state B given the current state is state A is 0.25.

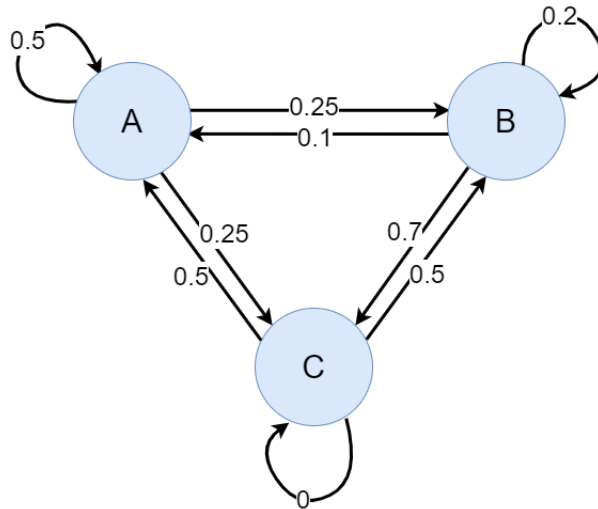


Figure 1: Example Markov model with 3 states: A, B, and C. The probability of transitioning from one state to another is indicated on the directed edge between the nodes.

The corresponding transition matrix, P , can then be defined:

$$P = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{ccc} A & B & C \\ \left[\begin{array}{ccc} 0.5 & 0.25 & 0.25 \\ 0.2 & 0.1 & 0.7 \\ 0.0 & 0.5 & 0.5 \end{array} \right] \end{array}.$$

The values in this matrix give the probability of transitioning from the state given by the row to the state given by the column. Thus, the probability of transitioning to state B from state A is found in the second column of the first row.

Markov chains have been used extensively to model scenarios in which states change over time. They can therefore be used to model music generation. By treating notes or groups of notes as individual states and calculating the probability of moving from one note to the next, we are able to specify a Markov chain that generates sequences of notes. Thus, the model effectively composes its own music.

Generating music based on probabilities is not a new idea. Much work has been done to recreate the musical style of many composers. In general, this process looks like the following. Individual notes or rhythms make up the states of the model. The transition probabilities are taken from a sample of music from one or more composers. On the most basic level, each state represents one note. The probability of transitioning between the notes can then be estimated. If the current state is C and we want the probability of transitioning to a G, then we can count the number of times a G follows a C in the sample music and then divide by the total number of times a C is present. For more complex models, the states might be sequences of notes themselves and incorporate various rhythms, with probabilities calculated similarly. Using the probabilities, the models are transitioned to build sequences of notes that sound similar to the composers upon which they are based. For a brief overview of the history of these attempts, see [2].

Recently, Markov models have been combined with more advanced methods of generating sequences, such as neural networks, as in [4], or a genetic algorithm, as in [1]. To varying degrees of success, these mathematicians have generated music that sounds as though it could have been produced by humans.

However, like much of the rest of music theory, this field has been dominated by Western classical composers. Such composers certainly differ in style, but tend to follow similar rules of Western music theory. Structures to imitate the music of other cultures are few and far between. See, for example, the work in [9] attempting to recreate Chinese folk music.

This work explores the possibility of modeling music that wildly differs from previous examples, that of Balinese *gamelan*.

1.2 Balinese *Gamelan*

A *gamelan* is an Indonesian orchestra made up of percussive instruments, each with a unique and well-defined role in the ensemble. The *reyong* is a set of twelve

gongs arranged in a line from low to high. Four players strike and dampen the gongs with wooden sticks called *panggul*. See Fig. 2



Figure 2: A *reyong*, shown here, is a set of twelve gongs played by four players. Image courtesy of [8].

Much like Western music, *gamelan* music can be classified into different genres. This study specifically looks at the *reyong's* role in the *gamelan gong kebyar* genre. In this context, the *reyong* parts are improvisational. We further focus on one particular style of improvisation called *norot* which is defined by the baseline pattern of notes in a typical improvisation.

Because the *reyong* is learned by rote, the rules of the improvisations had never been written out in the way a Western percussive part might be. Ethnomusicologist Dr. Leslie Tilley has constructed and transcribed a grammar for *reyong norot* improvisations, found in [6]. Tilley studied the improvisations of musician Dewa Ketut Alit. Although Alit is an expert on what is and is not good practice, these rules cannot easily be expressed in words. Instead, Tilley transcribed Alit's playing and worked to classify different improvisations. See Section 1.3 below for an abbreviated summary of these rules.

This work aims to build a series of Markov models that generate improvisations for the *reyong* based on Tilley's work. A successful model will output unique and random improvisations that are all within the confines of the established grammar. Unlike prior music generation approaches, our model operates outside the established theory of Western music, instead relying on a uniquely constricting set of rules found in no other genre or instrument.

1.3 The Grammar of *Reyong Norot*

The instruments in a Balinese *gamelan* are tuned to the *pélog* scale containing five notes. These notes are not necessarily consistent between different *gamelans*. In

Tilley’s study, the notes used were roughly equivalent to C#, D, E, G#, and A and are referred to by the names ding, dong, deng, dung, and dang respectively (or, to be concise, i, o, e, u, and a). Each of the four players of the *reyong* controls two to four individual gongs. You can see which players control which notes in Fig. 3.¹

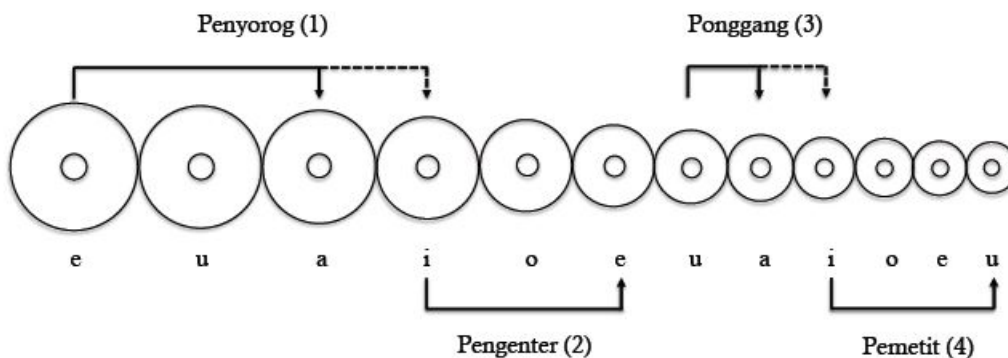


Figure 3: This diagram shows the twelve gongs on a *reyong*. The four players are called *penyorog*, *pengenter*, *ponggang*, and *pemetit*. Diagram courtesy of [5].

Gamelan music features melodic tones played on the *calung* called *pokok* tones. These *pokok* tones are not improvised and are the same for all four players. They occur every two beats and define the improvisation structure of the *reyong*. Because *gamelan* music tends to be cyclic, the same sequence of *pokok* tones repeats throughout the song. The improvisations on the *reyong*, however, differ every cycle.

In between every two *pokok* tones, each *reyong* player plays eight notes or rests. This grouping is called a “cell”. The surrounding *pokok* tones define what can be played in a cell. Thus, when talking about an individual cell, we will describe it as the shift between the two *pokok* tones. For example, a cell that moves from ding to dong is notated as an i-o shift.

Unlike in most Western music, *gamelan* music emphasizes the last note of a cell, not the first. Thus, an i-o shift will start on the first note *after* a ding and end on a dong on the eighth note. The next shift then necessarily shifts from dong to another tone. To better see this, when writing a single cell, we write the previous *pokok* tone in parenthesis before starting the eight notes.

For generality, Tilley uses numbers instead of note names when describing cell patterns. Each cell is defined by two numbers. The second number, representing the second *pokok* tone, is always zero, and the first number is the number of steps the first *pokok* tone is above the second. So an i-o shift can be described as a 4-0 shift, as ding is four steps above dong. If the next cell is then o-a, then it is represented as 2-0. Note that dong in the first cell was 0 but in the second cell became 2.

¹The extended ranges of the *penyorog* and *ponggang*, shown with the dotted lines in Fig. 3, are sometimes used in specific cases when improvising. For the sake of simplicity, we ignore these cases here.

For $n \neq 0$, the template pattern for an n -0 cell is

$$(n)//n + 1, n, n + 1, n/0, 0, 1, 0// \pmod{5}.$$

Because there are only five notes that repeat, the addition is modulo 5 (e.g. if $n = 4$, then $n + 1 = 0$). When the *pokok* tone does not change, $n = 0$ and there is a 0-0 shift. In such a case, the template pattern is

$$(0)//1, 0, 1, 0/1, 0, 1, 0//.$$

For example, an o-a shift (2-0) is written

$$(2)//3, 2, 3, 2/0, 0, 1, 0//$$

or, using note names,

$$(o)//e, o, e, o/a, a, i, a//.$$

Fig. 4 shows some examples of these template patterns.

Figure 4: Some example template patterns. On the left, an e-o (1-0) shift. Center, an i-u (2-0) shift. On the right, a u-u (0-0) shift. Images from [6].

Because of the limited ranges of each position (as shown in Fig. 3), a player often cannot play the full template pattern described above. In these cases, players rest or play alternatives to this baseline instead. In addition to playing the regular template notes, players can play harmonies, called *kempyung*. Each note has a high and low *kempyung*, which are three steps above and below a note, respectively. For example, the high *kempyung* of ding is dung and the low *kempyung* is deng. At any point in the regular pattern, a player can play the high or low *kempyung* in place of the regular template note. Usually this is due to the restricted ranges of players. In other words, players are more likely to play a *kempyung* tone if they do not have the template notes. Thus, for every note in a cell's pattern, there are three possible tones that can be played. Players also often rest on a beat instead of playing a note at all.

Using these rules and no others, we can create a finite but large set of template patterns that can be played. For example, consider a u-e (1-0) shift for player 1, who has access to notes e, u, and a. They could play the template pattern,

$$(u)//a, u, a, u/e, e, u, e//$$

(1)//2, 1, 2, 1/0, 0, 1, 0//
 (t)//t, t, t, t/t, t, t, t//,

where "t" refers to a template note. Or, taking "h" to represent the high *kempyung* and "l" the low,

(u)//e, u, e, u/a, a, u, e//
 (1)//0, 1, 0, 1/2, 2, 1, 0//
 (t)//h, t, h, t/l, l, t, t//.

Another possibility is

(u)//a, u, e, u/e, a, u, e//
 (1)//2, 1, 0, 1/0, 2, 1, 0//
 (t)//t, t, h, t/t, l, t, t, //.

Or, using "-" and "r" to mean rest,

(u)//-, u, a, -/a, -, u, e//
 (1)//-, 1, 2, -/2, -, 1, 0//
 (t)//r, t, t, r/l, r, t, t//.

All of these possible patterns are considered to adhere to a broader "template," as Tilley puts it (although some musicians would describe it more as a baseline). Deviating from this template is what allows the players to improvise. Tilley refers to these as cell variations because they vary from the template.² Many of the variations simply shift the indices of certain notes in the template pattern. For example, rather than a 1-0 pattern played originally as

(1)//2, 1, 2, 1//0, 0, 1, 0//,

this cell might be played as

(-)//1, 1, 2, 1//0, 0, 1, 0//.

Note that the first note, which should be a 2 (or, as a *kempyung*, a 0 or 4), is now played as a 1. This is a very common variation that Tilley calls a "delayed *pokok* tone unison," where the *reyong*'s tone echoes the *pokok* tone by one count.

The bulk of Tilley's paper works through a sample of improvisations and attempts to classify each into one or more of several types of variations, each with a unique rule for changing the template pattern of notes. Tilley also describes how often and under what conditions these different variations are played. For a complete list of these variations, see her work in [6].

²In prior work, Tilley referred to cells containing variations as "deviant" cells. She has since stopped using that language.

2 Methods

2.1 Overview

Our generation model is constructed in three layers, described in detail below. The first layer determines the structure of the song by selecting the sequence of *pokok* tones. The second determines which variations, if any, are present in each cell. The third and final layer selects which notes each player will play of those within their range. Because of the lack of a large data set of *reyong* improvisations, the probabilities for each layer had to be estimated slightly differently, but they all attempt to remain faithful to the analysis done by Tilley and described above. See Section 2.3 below for details of the probability calculations.

The completed model runs from a single Python script. It takes in user input to determine the length of the song and then generates a sequence of notes and rests for each of four *reyong* players. With the help of Python’s music21 package [7], the sequences are stored in a MIDI file that can then be played or viewed by the user. In order to remain authentic to the musical tradition of *gamelan*, the music uses a MIDI voice containing *reyong* samples. This is vital to fully represent the music and culture that was studied.

2.2 Model Details

Let T be the set of all possible *reyong* tones, or $T = \{a, e, i, o, u, rest\}$. Let $\hat{T} = T \setminus \{rest\}$.

The first layer of the model determines the pattern of *pokok* tones from which the players will improvise. This is a simple Markov model featuring five states, one for each element of \hat{T} . See the model diagram in Fig. 5.

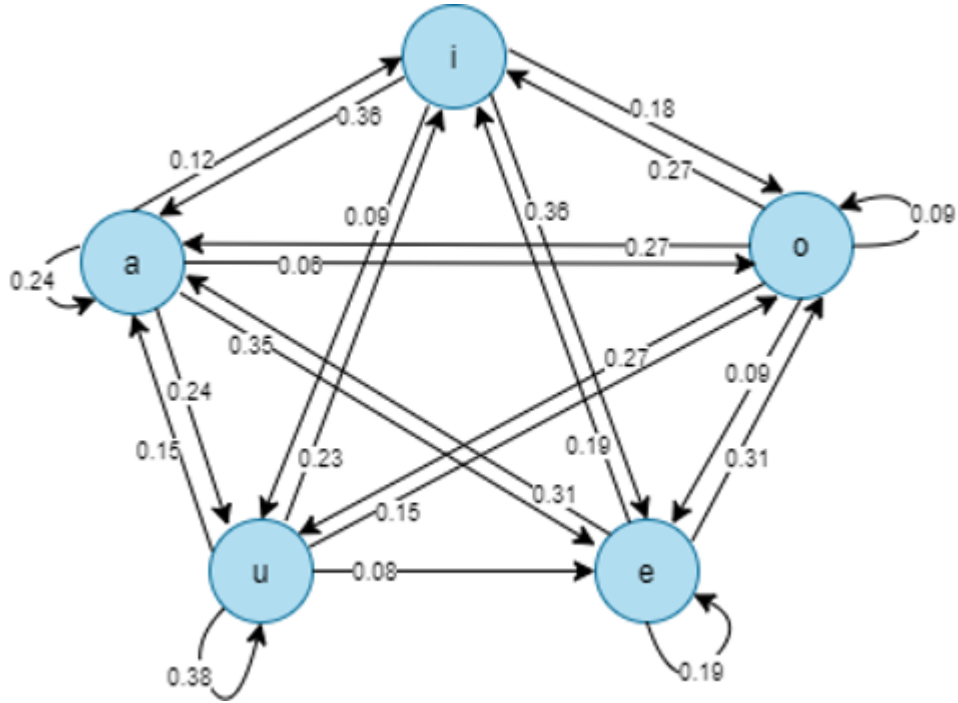


Figure 5: Graph representing the Markov model in the first layer. Each state is an element of \hat{T} and represents a *pokok* tone.

The corresponding transition matrix is

$$P_1 = \begin{matrix} & \begin{matrix} i & o & e & u & a \end{matrix} \\ \begin{matrix} i \\ o \\ e \\ u \\ a \end{matrix} & \begin{bmatrix} 0.00 & 0.18 & 0.36 & 0.09 & 0.36 \\ 0.27 & 0.09 & 0.09 & 0.27 & 0.27 \\ 0.19 & 0.31 & 0.19 & 0.00 & 0.31 \\ 0.23 & 0.15 & 0.08 & 0.38 & 0.15 \\ 0.12 & 0.06 & 0.35 & 0.24 & 0.24 \end{bmatrix} \end{matrix}$$

with initial probabilities $[0.2, 0.2, 0.2, 0.4, 0.0]$.

To incorporate the cyclic nature of *gamelan* music, the user can specify the number of cells in one cycle, m , and the number of cycles in the song, p . Then the song will contain a total of $n = m * p$ cells.

The first layer model visits m states and then repeats this sequence p times. This first layer will thus produce a sequence of n *pokok* tones, x_1, x_2, \dots, x_n , where each $x_i \in \hat{T}$. To finish the last cycle, a final tone, x_{n+1} is added, where $x_{n+1} = x_1$.

While the first layer contains a single model for the whole *reyong*, the second layer of the model is run four times, once for each player. This layer is a Markov model that selects the structure/variation of each cell in a player's improvisations. It is completely independent from the first layer.

Recall that each cell is based on a template, $//n+1,n,n+1,n/0,0,1,0//$, that is then modified by different variations. Different states in this model are represented

as binary arrays that encode possible variations. Thus, the model can transition from, say, a template cell, to a delayed *pokok* unison cell. Combinations of different variations are also permitted, although they are not utilized to the fullest extent here for simplicity. This model currently features ten cell structures which are defined in Table 1.

State #	Variation Name
1	No variations (template)
2	Delayed <i>pokok</i> unison
3	Advanced <i>pokok</i> unison
4	Both 2 and 3
5	Anticipation
6	Suspension
7	<i>Ngubeng-majalan</i> switch
8	<i>Kendang</i> pattern substitution
9	Reverse <i>norot</i> (first half)
10	Reverse <i>norot</i> (second half)

Table 1: A brief description of each of the states featured in Layer 2’s Markov model.

The corresponding transition matrix is

$$P_2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{matrix} & \left[\begin{array}{cccccccccc} 0.15 & 0.20 & 0.15 & 0.04 & 0.15 & 0.15 & 0.04 & 0.04 & 0.04 & 0.04 \\ 0.08 & 0.20 & 0.08 & 0.20 & 0.08 & 0.16 & 0.05 & 0.05 & 0.05 & 0.05 \\ 0.28 & 0.00 & 0.28 & 0.00 & 0.16 & 0.08 & 0.05 & 0.05 & 0.05 & 0.05 \\ 0.28 & 0.00 & 0.28 & 0.00 & 0.12 & 0.12 & 0.05 & 0.05 & 0.05 & 0.05 \\ 0.08 & 0.08 & 0.16 & 0.16 & 0.24 & 0.08 & 0.05 & 0.05 & 0.05 & 0.05 \\ 0.08 & 0.16 & 0.08 & 0.16 & 0.08 & 0.24 & 0.05 & 0.05 & 0.05 & 0.05 \\ 0.10 & 0.05 & 0.15 & 0.10 & 0.05 & 0.05 & 0.15 & 0.15 & 0.05 & 0.15 \\ 0.10 & 0.05 & 0.15 & 0.10 & 0.05 & 0.05 & 0.15 & 0.15 & 0.05 & 0.15 \\ 0.10 & 0.15 & 0.05 & 0.10 & 0.15 & 0.15 & 0.05 & 0.05 & 0.10 & 0.10 \\ 0.10 & 0.05 & 0.15 & 0.05 & 0.05 & 0.05 & 0.15 & 0.15 & 0.10 & 0.15 \end{array} \right] \end{matrix}.$$

To start things off simply, the starting state is set to state 1, a basic template cell.

For each player, this second-layer model visits n cell structure states, resulting in a sequence of cell structures $\bar{y}_1, \bar{y}_2, \dots, \bar{y}_n$, where each \bar{y}_i is a binary vector of length 8 that indicates a state of the model that was visited. The vector \bar{y}_i encodes the type(s) of variations that exist in cell i . For example, if in cell j there was no variation, then $\bar{y}_j = [0, 0, 0, 0, 0, 0, 0, 0]$. If, however, in cell k there is an advanced *pokok* unison variation, then that is represented by $\bar{y}_k = [0, 1, 0, 0, 0, 0, 0, 0]$.

We now have a sequence of *pokok* tones x_1, x_2, \dots, x_{n+1} as well as a sequence of binary arrays representing cell structures $\bar{y}_1, \bar{y}_2, \dots, \bar{y}_n$ for each player. The next step in the composition combines these two sequences.

First, however, a quick definition:

Definition 1. Let $x \in \hat{T}$ and $a \in \mathbb{Z}$. Then $x + a = a + x = y$, where $y \in \hat{T}$. If $a \geq 0$, then y is a steps above x . If $a < 0$, then y is a steps below x . This addition is modulo 5, so adding 3 to a note is the same as adding -2.

For cell $i \in [0, n]$ we define function $f(x_i, x_{i+1}, \bar{y}_i) = \bar{z}_i$, where \bar{z}_i is a vector of notes in cell i . See the diagram in Fig. 6 for an overview of the function f .

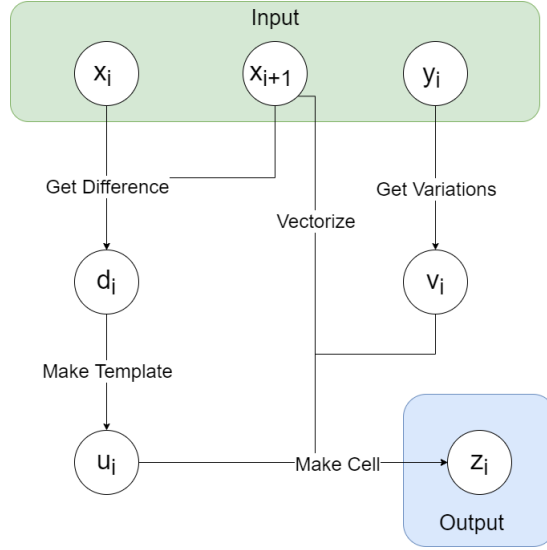


Figure 6: Function f takes inputs of x_i , x_{i+1} , and \bar{y}_i and returns \bar{z}_i , a vector of notes in cell i . The steps are described in detail below.

To begin, we get the difference between x_i and x_{i+1} . Let d_i equal the number of steps that x_i is above x_{i+1} . From this difference, we can make the template cell pattern, \bar{u}_i as follows:

$$\bar{u}_i = \begin{cases} [d_i + 1, d_i, d_i + 1, d_i, 0, 0, 1, 0] & d_i \neq 0 \\ [1, 0, 1, 0, 1, 0, 1, 0] & d_i = 0 \end{cases}$$

This represents the template pattern, in which no variations have occurred.

Also, we vectorize x_{i+1} to define \bar{x}_{i+1} as a vector of length 8 with every term equal to x_{i+1} .

We then get the variations from \bar{y}_i . The exact definition of this step is a large conditional based on the values in the vector \bar{y}_i . For ease of reading, we will say that these variations can be stored in a vector \bar{v}_i and give a few simple examples. This vector of integers represents the steps to be added or subtracted from the template pattern, \bar{u}_i .

For instance, if cell j is a cell with no variations, $\bar{y}_j = [0, 0, 0, 0, 0, 0, 0, 0]$. Then $\bar{v}_j = [0, 0, 0, 0, 0, 0, 0, 0]$, meaning there is zero variation from the template for each note.

If, however, cell k is a cell with an advanced *pokok* unison variation, then $\bar{y}_k = [0, 1, 0, 0, 0, 0, 0, 0]$ and $\bar{v}_k = [0, 0, 0, 0, 0, 1, -1, 0]$. In this case, the sixth note moves up by one step and the seventh note moves down one step.

In the final step, we make the completed cell. Here, we can specify f as

$$f(x_i, x_{i+1}, \bar{y}_i) = \bar{x}_{i+1} + \bar{u}_i + \bar{v}_i = \bar{z}_i.$$

Thus, the function returns a vector of eight notes as modified by any variations. We now have a set of n cells \bar{z}_i each containing eight notes that vary based on the cell structure.

The third and final layer of the model no longer depends on the cell structure or a note's position within the cell. Thus, for ease of reading, we will modify our notation here. Rather than continuing on with a set of n cells \bar{z}_i , we will simply use the sequence of notes, w_1, w_2, \dots, w_{8n} , where each w_k is simply a note in the total sequence. With eight notes in n cells, this sequence is now of length $8n$.

The third layer takes the template notes in w_k and selects the final output notes. It contains four states that represent the possible notes that can be played: the template note t , the high *kempyung* h , the low *kempyung* l , or a rest r . Call this set of states S . The basic transition matrix is

$$P_3 = \begin{matrix} & \begin{matrix} t & h & l & r \end{matrix} \\ \begin{matrix} t \\ h \\ l \\ r \end{matrix} & \begin{bmatrix} 0.40 & 0.20 & 0.20 & 0.20 \\ 0.40 & 0.20 & 0.20 & 0.20 \\ 0.40 & 0.20 & 0.20 & 0.20 \\ 0.50 & 0.25 & 0.25 & 0.00 \end{bmatrix} \end{matrix}$$

with initial probabilities

$$[0.5, 0.25, 0.25, 0].$$

The model is based on a traditional Markov model but is modified due to the changing limitations of the player's range. For example, in one cell, the template note may fall within the range, but in the next, it may not. Thus, at every time step, the model must check to see which states are permitted. The probability of transitioning to out-of-range states is then temporarily set to zero in the transition matrix and the remaining probabilities are normalized before selecting the next state. Due to this extra step, this level is not a traditional Markov model. This process is visualized in Fig. 7 and described below.

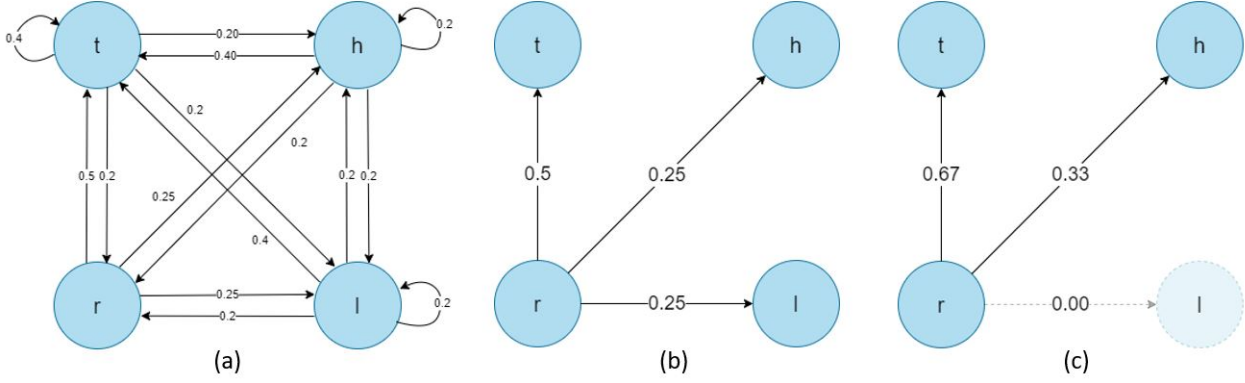


Figure 7: (a) Markov chain for the third layer, containing states for template, higher and lower *kempyung*, and rests. If the model is currently in state r, (b) shows the probabilities of transitioning to other states. However, if the lower *kempyung* tone is out of range for a player, its probability gets zeroed out and the other probabilities are normalized, as shown in (c).

Let R be the set of all notes in a player's range. Define function g as

$$g(R, s_{k-1}, w_k) = (s_k, t_k),$$

where s_k is the state in S of the k^{th} note and t_k is the corresponding output note in T .³ We begin with a row in transition matrix P_3 that gives the likelihood of transitioning to each possible next state,

$$[p_{s_{k-1}, t}, p_{s_{k-1}, h}, p_{s_{k-1}, l}, p_{s_{k-1}, r}].$$

We modify this vector as follows:

$$\text{If } w_k \notin R, \text{ then } p_{s_{k-1}, t} = 0.$$

$$\text{If } w_k + 3 \notin R, \text{ then } p_{s_{k-1}, h} = 0.$$

$$\text{If } w_k - 3 \notin R, \text{ then } p_{s_{k-1}, l} = 0.$$

We then normalize the row vector so that it sums to 1. From there, we can use this modified distribution of probabilities to randomly select the next state s_k . Finally, we obtain t_k , the output note in T , with the following equation:

$$t_k = \begin{cases} w_k, & s_k = t \\ w_k + 3, & s_k = h \\ w_k - 3, & s_k = l \\ rest, & s_k = r. \end{cases}$$

³Note that this definition is defined recursively. To obtain the first state, s_1 , we follow much the same process, modifying the vector of initial probabilities rather than the transition matrix.

This returns t_k , the k^{th} note to be played in the sequence.

Repeating layers two and three for all eight notes in all n cells for each of the four players will return a sequence of output notes and rests for each player. These lists are added to a song and output as a MIDI file.

2.3 Probability Estimations

The transition matrices and probabilities therein shown above were calculated in a variety of ways. The first layer is the only to feature probabilities that were calculated bottom-up, or from actual data. Tilley’s study includes an analysis of five songs. The likelihood of transitioning from one *pokok* tone to another was estimated using the *pokok* tones from those songs via a maximum likelihood estimate. To find the probability of transitioning from tone x to tone y , the number of times x was followed by y in the data was counted, and this total was divided by the number of times x appeared in the data at all.

For example, let’s look at the probability of transitioning from *pokok* tone ding to *pokok* tone dong, shown in Fig. 5 to be about 0.18. To find this, we counted the number of times ding appeared as a *pokok* tone in Tilley’s songs, which occurred eleven times. We then counted the number of times a ding was followed by dong, which only happened twice. Thus, to estimate the probability that the next state is dong given that the current state is ding, we simply divide $2/11 \approx 0.18$.

Unfortunately, there is no existing large data set from which the probability of transitioning between two cell structures could be estimated. Instead, the transition probabilities were estimated top-down using information from Tilley’s work. Tilley describes certain variations as more common than others, so the probability of transitioning to those structures was set higher. Additionally, similar variations tend to follow after each other. The resulting transition matrix is thus rudimentary, but would require expert opinion or a large, labeled data set to improve.

The transition probabilities for the third layer are again calculated top-down. Tilley notes that playing the template note is more common than either *kempyung*, if available. Thus, the probability of transitioning to the template note is consistently quite higher than the other states. Additionally, *reyong* players never rest for two consecutive beats in order to keep their hands moving. To model this, the probability of transitioning from a rest back to another rest is always zero. Tilley gives no indication of either the low or high *kempyung* being preferred over the other, and indeed they appear to be used the same number of times. Thus, we keep the probabilities of moving to either *kempyung* state equal.

2.4 Other Details

The code base contains six Python classes and a driver script that runs the entire process. The Markov model class makes use of the numpy package’s `random.choice()` method to randomly select the next state in sequence based on the transition probabilities.

The user has the option to add *pokok* tones and *kempli* beats to the resulting MIDI file. Since the *pokok* tones are generated regardless and the *kempli* beat is a single note repeated, this adds no computation to the model. One key benefit of adding these additional parts is that they give context to the *reyong*. On its own, it is difficult to pick out the structure and harmony that is present in the *reyong* improvisation.

The structure of the code can be seen in the flowchart in figure 8. To see the full code base, see Appendix C.

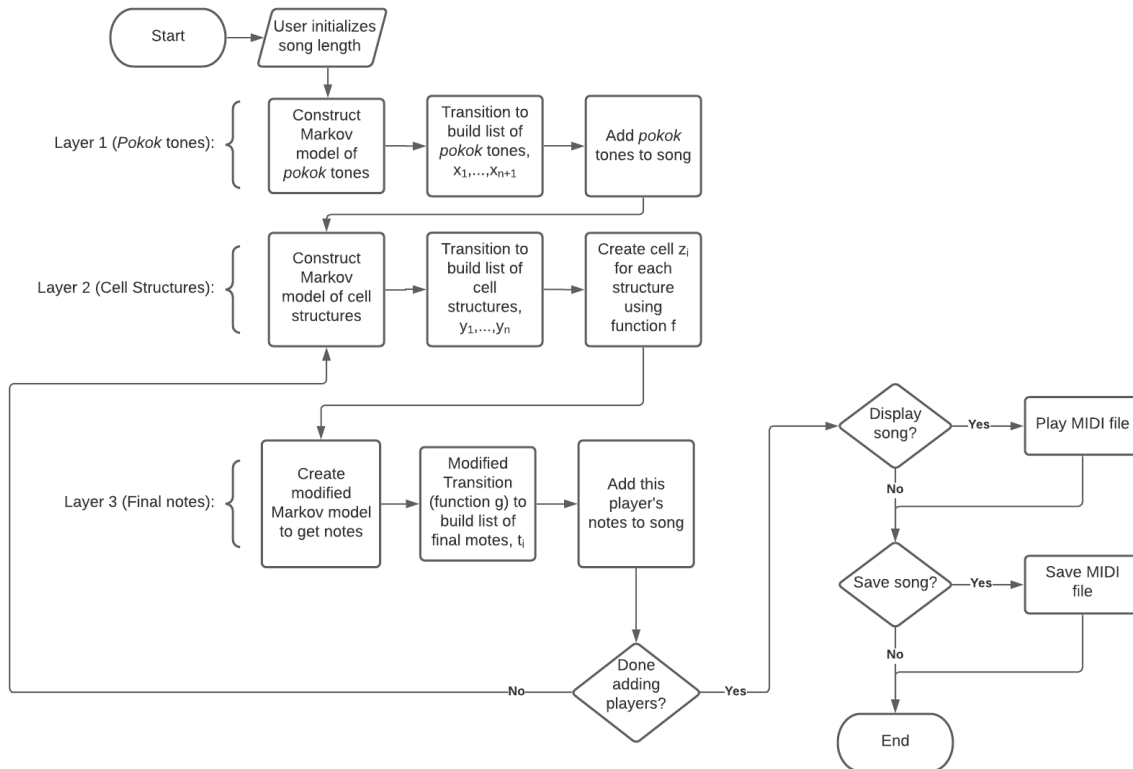


Figure 8: Flowchart showing the general processes contained within the model.

3 Results

The model outputs the completed improvisations as a MIDI file. From there, a voice can be added to each player in order to make the music sound authentic to an actual *gamelan*. The MIDI sample was provided by [3]. A few sample songs can be heard at [this link](#). Note that for each song, there is a track with only the *reyong* and one that also includes the *pokok* tones and *kempli* beat, which help bring out the structure in the audio.

Let's take a look at the score for one example output, *Reyong* Sample 4. Fig. 9 shows an excerpt from the beginning of the song. The entire score can be found in Appendix A.

The image shows a musical score for a generated set of *reyong* parts. It consists of six staves. The first staff, labeled 'Pokok Tones', is in treble clef, G major, and 4/4 time, featuring a melodic line with notes G4, A4, B4, C5, B4, A4, G4. The second staff, labeled 'Kempli Beat', is in bass clef, G major, and 4/4 time, showing a rhythmic pattern of eighth notes with rests. The third staff, labeled 'Penyorog', is in treble clef, G major, and 4/4 time, showing a rhythmic pattern of eighth notes with rests. The fourth staff, labeled 'Pengenter', is in treble clef, G major, and 4/4 time, showing a rhythmic pattern of eighth notes with rests. The fifth staff, labeled 'Ponggang', is in treble clef, G major, and 4/4 time, showing a rhythmic pattern of eighth notes with rests. The sixth staff, labeled 'Pemetit', is in treble clef, G major, and 4/4 time, showing a rhythmic pattern of eighth notes with rests.

Figure 9: Sample score from a generated set of *reyong* parts. This excerpt contains four cells for each position on the *reyong*. The *pokok* tones and *kempli* beat are also shown on the first and second line respectively.

Note that while the *pokok* tones are visible, the sequence of states from the other two layers of the model are not very clear. The combination of rests and *kempyung* tones make it difficult to decipher what variations were present in the cell structure. In this sense, the model is similar to a hidden Markov model, in which the actual states of the model are never seen, but instead some noisy emission given the states themselves is observed.

If the model is run again, we can get an entirely different output. Fig. 10 shows an excerpt from a second song, *Reyong* Sample 5, demonstrating the variability that occurs with each unique run of the model. The complete score can be found in Appendix B.

The image shows a musical score for a *reyong* song, consisting of six staves. The top staff, 'Pokok Tones', is in treble clef with a key signature of one sharp (F#) and a 4/4 time signature. It contains a sequence of notes: a quarter note G4, a dotted quarter note A4, a half note B4, a quarter note C5, a dotted quarter note D5, and a quarter note E5. The second staff, 'Kempli Beat', is in bass clef with the same key signature and time signature, featuring a steady eighth-note rhythm. The remaining four staves ('Penyorong', 'Pengenter', 'Ponggang', and 'Pemetit') are in treble clef with the same key signature and time signature, each containing complex rhythmic patterns with many slurs and accents.

Figure 10: Another example *reyong* song. This song features rests in between each cycle, which can be seen in the full score in Appendix B.

Optionally, the output can add rests between cycles. When *reyong* is played by itself, rests may be interspersed between cycles or cells to provide further structure to the improvisation. In this model, the added rests help guide the listener through the produced rhythms and provide additional variation. See *Reyong* Samples 5, 6, and 7 for examples of songs with added rests.

4 Discussion

The produced model goes a long way towards recreating Tilley’s grammar, but it is not complete. After listening to several of the sample outputs, Dewa Alit stated that it sounds good and is a good start at generating improvisations. However, he was also able to easily pick out parts of the output that sounded wrong. For example, Alit said of *Reyong* Sample 5: “the *pokok* note (*calung*) is on ding, but [the] *reyong* is on dang. This does not work for *norot*. I do not consider this combination [to be] in harmony or “*ngempyung*.”

Exactly why inconsistencies like this appeared is difficult to pinpoint. Several simplifying assumptions have been made that cause the output to differ from actual *reyong* improvisations. First, the model treats each position as an independent model, generating sequences of cell structures and notes based solely on the *pokok* tones and the previous note/cell. In a real *gamelan*, the four players listen to each other and interact musically. They may hear ideas from each other and incorporate them into their own variations. Tilley attempted to address this by classifying certain variations that require collaboration between positions. Additionally, there are other variations that are restricted to certain cell shifts and/or positions. To keep the model uniform,

these are not included here. Furthermore, multiple variations can often occur in the same cell for *reyong* players. Such functionality is possible with this model, but has not been fully implemented due to little reference in Tilley's work about these stacked variations. Finally, Tilley herself was unable to classify every single cell that Alit played into one of her variation definitions. Some notes went unexplained, and are thus not included in our model based on her work.

However, like Alit said, this model is only a start that leaves open many opportunities for future work. First, to address some of the assumptions above, the collaborative nature of the four *reyong* players could be addressed through additional structure and states. Adding these additional variations would allow for more nuanced harmonies and structures to appear throughout the generated songs.

One of the main ways to improve the model would be to use a more data-driven approach to calculate the transition probabilities. For all three layers, this would rely on obtaining or creating large data sets of *reyong* improvisations. In the grand scheme of *gamelan* music, the small subset notated by Tilley is not enough to deduce accurate probabilities. We could further generalize the model based on input from more musicians and/or subgenres of Balinese *gamelan* music.

Given a larger data set of this type, other models could also be explored to better approximate reality. This could include employing machine learning or other artificial intelligence techniques, which would avoid the need for humans to define the rules as we have attempted here.

Finally, the ethics of this study should also be addressed. After all, by removing the human aspect of the improvisation, this model has taken out a key component of the music itself. To Alit and other musicians, the improvisations are felt rather than learned, and this model merely copies some of that feeling. To that end, the authors of this work are not Balinese and have never been to Bali or played in a *gamelan*. The project is based on the analysis by Tilley who is also neither Balinese nor a *gamelan* expert. Tilley's work, and by extension perhaps this work, does bring attention to the diversity of music around the globe by framing the theory of this music in a way familiar to Westerners. On the other hand, the grammar and the model are only approximations by outsiders who do not actually represent the people and culture of Bali. In all, this analysis can be for good, so long as it is acknowledged that it does not replace the actual musical background of this culture. The model, and ethnomusicology in general, are mere approximations to learn about the music, but are a long way away from fully conveying an understanding and appreciation of the music and musicians.

5 Acknowledgements

Thank you to thesis advisor Dr. Robert Rovetti and musical consultants Dr. Leslie Tilley, Dr. Paul Humphreys, Dr. Michael Tenzer, and Professor Aaron Smith. Funding for this project was provided by the Loyola Marymount University Honors Program.

Glossary

- calung*** the metallophone responsible for playing the *pokok* tones of the song. 4
- cell** a group of eight 16th notes in which the *reyong* improvises, beginning directly after a *pokok* tone and ending on the next *pokok* tone. 4
- gamelan*** an Indonesian orchestra consisting of mostly percussion instruments, including gongs, drums, and metallophones. 2
- gong kebyar*** a modern, secular genre of *gamelan* music featuring complex interlocking parts. 3
- kempli*** a small gong that is used to keep time in *gamelan gong kebyar* music by playing steady quarter notes on a single pitch. 14
- kempyung*** a note that is harmonious to a given note, located either three steps above or below the note in question. 5
- norot*** a style of *reyong* improvisation that is defined by the template pattern that alternates between the *pokok* tone and the tone one step above it. 3
- panggul*** a wooden mallet with which musicians strike a gong on the *reyong*. 3
- pemetit*** the highest position on the *reyong*, also called the fourth position. 4
- pengenter*** the second lowest position on the *reyong*, also called the second position.
4
- penyorog*** the lowest position on the *reyong*, also called the first position. 4
- pokok*** the melodic baseline of *gamelan* music that defines the cells for the *reyong* improvisations, played every two beats on the *calung*. 4
- ponggang*** the second highest position on the *reyong*, also called the third position.
4
- pélog*** the five-tone scale used in *gamelan gong kebyar* music, featuring the notes deng, dung, dang, ding, and dong. 3
- reyong*** a set of twelve gongs that are played by four players, featuring an improvisational role in a *gamelan*. 2
- template** the idea of a standard baseline cell that can then be modified by improvisations, term coined by Leslie Tilley. 5
- variations** changes in the standard template cell structure, sometimes referred to as “deviations”. 6

References

- [1] C. Bell, *Algorithmic Music Composition Using Dynamic Markov Chains and Genetic Algorithms*, Computing Sciences in Colleges. **27** (2011), no. 2, 99-107, <https://doi.org/10.5555/2038836.2038850>.
- [2] D. Herremans, C. Chuan, and E. Chew, *A Functional Taxonomy of Music Generation Systems*, ACM Comput. Surv. **50** (2017), no. 5, <https://doi.org/10.1145/3108242>.
- [3] D. Powell and J. Thompson, *Discovery Series Balinese Gamelan*, Native Instruments GmbH. (2009).
- [4] K. Verbeurgt, M. Fayer, and M. Dinolfo, *A Hybrid Neural-Markov Approach for Learning to Compose Music by Example*, LNCS. **3060** (2004), 480-484, https://doi.org/10.1007/978-3-540-24840-8_41.
- [5] L. Tilley, *Making It Up Together: The Art of Collective Improvisation in Balinese Music and Beyond*, University of Chicago Press, 2020.
- [6] L. Tilley, *Reyong Norot Figuration: An Exploration into the Inherent Musical Techniques of Bali*, University of British Columbia. (2000).
- [7] M. Cuthbert, *music21*, v.6.1.0, MIT Music and Theater Arts. (2020).
- [8] N. Walker, *Improvising Reyong Norot* (photo). (2003).
- [9] X. Zheng, D. Li, L. Wang, L. Shen, Y. Gao, and Y. Zhu, *An Automatic Composition Model of Chinese Folk Music*, AIP Conference Proceedings **1820** (2017), <https://doi.org/10.1063/1.4977359>.

A *Reyong* Sample 4 Score

The image displays a musical score for a piece titled "A *Reyong* Sample 4 Score". The score is written for six parts: Pokok Tones, Kempli Beat, Penyorong, Pengenter, Ponggang, and Pemetit. The music is in a 4/4 time signature and the key signature has two sharps (F# and C#). The score is divided into two systems. The first system contains the first four measures of music. The second system begins at measure 3, as indicated by a "3" above the first staff. The parts are arranged vertically, with Pokok Tones at the top and Pemetit at the bottom. The Pokok Tones part consists of a few notes, while the other parts feature more complex rhythmic patterns, including eighth and sixteenth notes.

5

Pokok

Kempli.

Penyrog

Pengenter

Ponggang

Pemetit

7

Pokok

Kempli.

Penyrog

Pengenter

Ponggang

Pemetit

9

Pokok

Kempli.

Penyorog

Pengenter

Ponggang

Pemetit

B *Reyong* Sample 5 Score

The image displays a musical score for a piece titled "Reyong Sample 5". The score is organized into two systems, each containing six staves. The parts are labeled on the left as follows:

- Pokok Tones:** The top staff of each system, written in treble clef with a key signature of two sharps (F# and C#) and a 4/4 time signature. It features a melodic line with a few notes and rests.
- Kempli Beat:** The second staff, written in bass clef with the same key signature and time signature. It provides a rhythmic accompaniment with a series of eighth notes and rests.
- Penyorong:** The third staff, written in treble clef with the same key signature and time signature. It contains a complex rhythmic pattern of eighth notes.
- Pengenter:** The fourth staff, written in treble clef with the same key signature and time signature. It features a rhythmic pattern of eighth notes, often with beamed pairs.
- Ponggang:** The fifth staff, written in treble clef with the same key signature and time signature. It contains a rhythmic pattern of eighth notes, similar to the Pengenter part.
- Pemetit:** The sixth staff, written in treble clef with the same key signature and time signature. It features a rhythmic pattern of eighth notes, similar to the Pengenter part.

The first system concludes with a measure containing a fermata over a note. The second system begins with a measure containing a fermata over a note, followed by a measure with a triplet of eighth notes indicated by a "3" above the staff. The score continues with several measures of music for each part, maintaining the same key signature and time signature throughout.

2

5

Pokok Tones

Kempli Beat

Penyorog

Pengenter

Ponggang

Pemetit

7

Pokok Tones

Kempli Beat

Penyorog

Pengenter

Ponggang

Pemetit

9

Pokok Tones

Kempli Beat

Penyorog

Pengenter

Ponggang

Pemetit

11

Pokok Tones

Kempli Beat

Penyorog

Pengenter

Ponggang

Pemetit

13

Pokok Tones

Kempli Beat

Penyorog

Pengenter

Ponggang

Pemetit

15

Pokok Tones

Kempli Beat

Penyorog

Pengenter

Ponggang

Pemetit

17

Pokok Tones

Kempli Beat

Penyorog

Pengenter

Ponggang

Pemetit

19

Pokok Tones

Kempli Beat

Penyorog

Pengenter

Ponggang

Pemetit

C Codebase

C.1 Tone.py

Tone.py contains a Python class representing a tone, used both for *pokok* tones as well as notes on the *reyong*. It provides methods for converting between notes in the *pélog* scale and notes in Western music, as well as methods for finding the *kempyung* tones corresponding to a tone.

```
class Tone:
    tones = ["i", "o", "e", "u", "a"]

    # Constructor for Tone class. Takes in a toneName argument,
    # which should be a
    # string of value "a", "e", "i", "o", "u", or "r"
    def __init__(self, toneName):
        self.toneName = toneName
        self.noteName = self.matchNote()
        if self.noteName == None:
            raise ValueError("Incorrect input for Tone: '" + str(
                toneName) + "'\nPlease
                choose from 'a', 'e',
                'i', 'o', 'u', 'r")

    # Chooses the corresponding 'western' notename to go with the
    # tone, or rest if toneName = r

    def matchNote(self):
        switcher = {
            "i": "C#",
            "o": "D",
            "e": "E",
            "u": "G#",
            "a": "A",
            "r": "rest"
        }
        return switcher.get(self.toneName, None)

    # returns the toneName
    def getToneName(self):
        return self.toneName

    # returns noteName
    def getNoteName(self):
        return self.noteName

    # Creates a new tone object for the high kempyung tone
    def getHighKempyung(self):
        currentIndex = self.tones.index(self.getToneName())
        kempyungIndex = (currentIndex + 3) % 5
        highKempyung = Tone(self.tones[kempyungIndex])
        return highKempyung

    # Creates a new tone object for the low kempyung tone
```

```

def getLowKempyung(self):
    currentIndex = self.tones.index(self.getToneName())
    kempyungIndex = (currentIndex - 3) % 5
    lowKempyung = Tone(self.tones[kempyungIndex])
    return lowKempyung

# When converted to string, want to display toneName.
def __str__(self):
    return self.toneName

```

C.2 Cell.py

Cell.py contains two classes. One, CellStructure, is essentially a wrapper class around an array representing the cell structure. The other, Cell, essentially performs function f to combine layers 1 and 2 of the model. It takes in two *pokok* tones and the cell structure and creates the list of notes in that cell.

```

import numpy as np
import Tone as tn

# Class for a cell of 8 tones
class Cell:
    tones = ["i", "o", "e", "u", "a"]

    # Cell constructor. Takes in starting and ending Tone objects
    # for the surrounding Pokok
    # Tones,
    # as well as a cellStructure array, which defaults to a template
    # cell (all 0s)
    def __init__(self, startTone, endTone, cellStructure=[0,0,0,0,0,
0,0,0,0,0]):
        self.cellStructure = cellStructure
        self.startTone = startTone.getToneName()
        self.endTone = endTone.getToneName()
        self.setDiff()
        self.setNumPattern()
        self.setTonePattern()

    # Given the start tone and end tone, set the difference between
    # the two (with wraparound)
    # Is used to get the numbers of the cell (e.g. and a to i is a 4
    # -0 shift, so the difference is
    # 4)
    def setDiff(self):
        start = None
        end = None
        for i in range(len(self.tones)):
            if self.tones[i] == self.startTone:
                start = i
            if self.tones[i] == self.endTone:
                end = i
            self.endIdx = i

```

```

difference = start - end
if difference < 0:
    difference = difference + 5
self.difference = difference

# Once a difference is set, uses the cellStructure array to
                                construct a list of numbers
# of the tones in the template.
def setNumPattern(self):
    n = self.difference
    if n > 4:
        raise ValueError("Starting tone number must be < 5")

    if n != 0:
        self.numList = [n+1, n, n+1, n, 0, 0, 1, 0]
    else:
        self.numList = [1, 0, 1, 0, 1, 0, 1, 0]

# Check for delayed pokok (0th index):
if self.cellStructure[0]:
    self.numList[0] = n

#Check for advanced pokok (1st index):
if self.cellStructure[1]:
    self.numList[5:7] = [1,0]

# Anticipations (2nd index):
if self.cellStructure[2]:
    index = np.random.randint(1,4)
    self.numList[index] = self.numList[index+1]

# Suspensions (3rd index):
if self.cellStructure[3]:
    index = np.random.randint(1,4)
    self.numList[index] = self.numList[index-1]

# Second-half flexibility
# Ngubeng-majalan switch (4th index)
if self.cellStructure[4]:
    if n!=0:
        self.numList[4] = 1
    else:
        self.numList[4] = 0

# Kendang Pattern Substitution (5th index)
if self.cellStructure[5]:
    self.numList[4:6] = [0,1]

# reverse norot first half: (6th index)
if self.cellStructure[6]:
    self.numList[0:4] = [n-1, n, n-1, n]

# second half: (7th index)
if self.cellStructure[7]:

```

```

        if n!=0:
            self.numList[4:8] = [0, 0, 4, 0]
        else:
            self.numList[4:8] = [4, 0, 4, 0]

# And anything else not covered here

self.reduce()

# Reduces the numlist mod 5
def reduce(self):
    for i in range(8):
        self.numList[i] = self.numList[i]%5

# Given the numList and the starting and ending tones, sets the
# toneList (aka the sequence
# of Tone objects that form the template of the cell.)
def setTonePattern(self):
    self.toneList = []
    for num in self.numList:
        idx = (num + self.endIdx) % 5
        self.toneList.append(tn.Tone(self.tones[idx]))

# Want the string version of a cell object to return the tone
# names in the toneList
def __str__(self):
    string = '['
    for tone in self.toneList:
        string = string + ' ' + tone.getToneName()
    string = string + ']'
    return string

# Class for cell structure (basically array wrapper for numpy
# purposes)
# Will be fed into Cell class to determine what the structure of the
# cell should be.
class CellStructure:

    def __init__(self, array):
        self.structure=array

    def getList(self):
        return self.structure

    def __str__(self):
        return str(self.structure)

```

C.3 MarkovModel.py

MarkovModel.py contains a class used to create and perform typical Markov model operations, including transitioning to the next state. This class also contains the ability to modify the transitions, as in the third level of our model.

```
import numpy as np
import copy

# Class for a generic Markov Model
class MarkovModel:

    # Constructor takes in a list of states, a list of lists
    # transitionMatrix, and an
    # optional list of
    # initial probabilities (if empty, assume all equally likely)
    def __init__(self, states, transitionMatrix, initialProb=[]):
        self.states = copy.deepcopy(states)
        self.transitionMatrix = copy.deepcopy(transitionMatrix)
        if len(initialProb) == 0:
            self.initialProb = [1/len(self.states)] * len(self.
                states)
        else:
            self.initialProb = copy.deepcopy(initialProb)
        # will need to add some input validation. For now, assume
        # all correct

        self.setStartingValue()

    # Sets the current state based on the initial probabilities
    def setStartingValue(self):
        self.currentState = np.random.choice(self.states, replace=
            True, p=self.initialProb)

    # Method to transition the model
    # For normal transitions, leave modifiedProbs empty and specify
    # the number of transitions as n
    # For modified transitions, enter the list of the row of the
    # current state with the new
    # probabilities
    # which only apply to this step. For modified transitions, must
    # occur one at a time.
    def transition(self, modifiedProbs=[], n=1):
        if not modifiedProbs:
            return self.normalTransition(n)
        else:
            return self.modTransition(modifiedProbs)

    # A typical markov transition. Returns a list starting at the
    # current state and ending at
    # the most recent state.

    # Then transitions one last time
    # Default n=1.
    def normalTransition(self, n):
        memory=[self.currentState]
```

```

for i in range(n):
    currentIndex = self.states.index(self.currentState)
    self.currentState = np.random.choice(self.states,
                                        replace=True, p=self.
                                        transitionMatrix[
                                        currentIndex])

    memory.append(self.currentState)
self.currentState = np.random.choice(self.states, replace=
                                     True, p=self.
                                     transitionMatrix[
                                     currentIndex])

return memory

# modifiedProbs is a vector of transition probabilities for the
# current state's row.
# Does not alter the transition matrix and thus does not change
# future transitions.
def modTransition(self, modifiedProbs):
    self.currentState = np.random.choice(self.states, replace=
                                         True, p=modifiedProbs)

    return self.currentState

# Return the current state of the model
def getCurrentState(self):
    return self.currentState

# Return the transition matrix
def getTransitionMatrix(self):
    return self.transitionMatrix

# Return the index at which the currentState is located in the
# states list.
def getCurrentIndex(self):
    return self.states.index(self.currentState)

# Return the list of states
def getStates(self):
    return self.states

```

C.4 ModifiedMarkovModel.py

ModifiedMarkovModel.py performs most of the work in level three of our model: creating a Markov model, checking the ranges, adjusting the probability vector, and transitioning.

```

import Tone as tn
import MarkovModel as mv
import copy

# Class to get each position's model for actual notes being played
# Possibly consider renaming
class ModifiedMarkov:

```

```

rangeDict = {1: ["e", "u", "a"],
             2: ["i", "o", "e"],
             3: ["u", "a"],
             4: ["i", "o", "e", "u"]}

# Constructor takes in a playerNum from 1 to 4 as well as a
# cellList (list of cells in the
# song)
def __init__(self, playerNum, cellList):
    self.playerNum = playerNum
    self.templateList = []
    for cell in cellList:
        for tone in cell.toneList:
            self.templateList.append(tone)
    self.playerRange = self.rangeDict.get(self.playerNum)
    self.model = self.getModel()

# returns initial probability list based on which notes are
# actually in range
# (and does not allow for starting on a rest... for now)
def getInitialProbs(self):
    availableStates = self.checkRange(self.templateList[0])
    newProbabilities = self.replaceVector(availableStates, [0.50
                                                         , 0.25, 0.25, 0.00])
    return newProbabilities

# Transitions the model one step from index i.
# Looks to see what states are available and zeros out others
def transition(self, index):
    currentStateIdx = self.model.getCurrentIndex()
    oldVector = self.model.getTransitionMatrix()[currentStateIdx
                                                ]
    availableStates = self.checkRange(self.templateList[index +
                                                         1])
    newVector = self.replaceVector(availableStates, oldVector)
    newState = self.model.transition(modifiedProbs=newVector)
    return self.convertToTone(newState, self.templateList[index
                                                            + 1])

# Method to return a tone's template, high, low, or rest based
# on input state
def convertToTone(self, state, tone):
    if state == "t":
        return tone
    elif state == "h":
        return tone.getHighKempyung()
    elif state == "l":
        return tone.getLowKempyung()
    else:
        rest = tn.Tone("r")
        return rest

# Main function, transitions through all indices and returns a
# tuple of the toneList (list

```



```

# of tones to be played) as well as a letterList (list of states
                    't', 'h', 'l', and 'r')
# Technically can only be called once... need to fix this so
                    that it works if you call it
                    again...
def getFullList(self, verbose=False):
    toneList = []
    letterList = []
    toneList.append(self.convertToTone(self.model.
                                        getCurrentState(), self.
                                        templateList[0]))
    letterList.append(self.model.getCurrentState())
    for i in range(len(self.templateList) - 1):
        toneList.append(self.transition(i))
        letterList.append(self.model.getCurrentState())
    if verbose:
        print("Player " + str(self.playerNum))
        print("Template Notes:  ", end = '')
        print(*self.templateList)
        print("Letter Names  :  ", end = '')
        print(*letterList)
        print("Actual Notes  :  ", end = '')
        print(*toneList)
    return (toneList, letterList)

# Method to check which states are within the player's range.
                    Returns a list
# of states within range.
# Note that rest 'r' will always be in range, as will at least
                    one other state,
# so always returns a list of length 2 or 3.
def checkRange(self, tone):
    availableStates = ["r"]
    if tone.getToneName() in self.playerRange:
        availableStates.append("t")
    if tone.getHighKempyung().getToneName() in self.playerRange:
        availableStates.append("h")
    if tone.getLowKempyung().getToneName() in self.playerRange:
        availableStates.append("l")
    return availableStates

# Takes in a list of available states as well as a probability
                    vector
# and returns a new vector with unavailable states zeroed out
                    and
# the available states normalized.
def replaceVector(self, availableStates, vector):
    newVector = copy.deepcopy(vector)
    if "t" not in availableStates:
        newVector[0] = 0
    if "h" not in availableStates:
        newVector[1] = 0
    if "l" not in availableStates:
        newVector[2] = 0

```

```

total = sum(newVector)
for i in range(len(vector)):
    newVector[i] = newVector[i] / total
return newVector

# Creates a markov model of states "t", "h", "l", and "r"
# Note the transition probabilities - when template notes "t"
# are available,
# they are more likely to be played regardless of what comes
# before.
# Also note that rests never follow other rests/
def getModel(self):
    states = ["t", "h", "l", "r"]
    matrix = [ [0.40, 0.20, 0.20, 0.20],
               [0.40, 0.20, 0.20, 0.20],
               [0.40, 0.20, 0.20, 0.20],
               [0.50, 0.25, 0.25, 0.00]
             ]
    initial = self.getInitialProbs()
    model = mv.MarkovModel(states, matrix, initialProb=initial)
    return model

```

C.5 Song.py

The Song class makes extensive use of the music21 package to transform our lists of notes into MIDI objects. It contains methods for adding the *pokok*, *kempli*, and *reyong* notes to the song, as well as methods for playing and saving the song as a MIDI file.

```

import Tone as tn
import MarkovModel as mv
import music21 as m21
import matplotlib as plt

# Class for a song object
class Song:

    # Constructor creates a score and parts for pokok tones and all
    # players

    def __init__(self, numCells, numCycles, addRests=False):
        self.numCells = numCells
        self.numCycles = numCycles
        self.addRests = addRests

        self.fullScore = m21.stream.Score()
        self.pokokPart = m21.stream.Part()
        self.kempli = m21.stream.Part()
        self.player1 = m21.stream.Part()
        self.player2 = m21.stream.Part()
        self.player3 = m21.stream.Part()
        self.player4 = m21.stream.Part()

```

```

self.fullScore.append(self.pokokPart)
self.fullScore.append(self.kempli)
self.fullScore.append(self.player1)
self.fullScore.append(self.player2)
self.fullScore.append(self.player3)
self.fullScore.append(self.player4)
# Add the initial rest so we start after the first pokok
initialRest = m21.note.Rest()
initialRest.duration.quarterLength = 0.25
self.player1.append(initialRest)
self.player2.append(initialRest)
self.player3.append(initialRest)
self.player4.append(initialRest)

# Add a single tone (pokok) to the pokok part
def addPokok(self, pokok):
    # Currently putting pokoks at octave 4, but will need to
    # check this
    noteToAdd = m21.note.Note(pokok.noteName + "4")
    # Pokok tones last for two beats
    noteToAdd.duration.quarterLength = 2.0
    self.pokokPart.append(noteToAdd)
    # if addKempli:
    #     self.addKempli()
    #     self.addKempli()

# Add a single kempli tone to the kempli part
def addKempli(self):
    kempliToAdd = m21.note.Note("B2")
    kempliToAdd.duration.quarterLength = 0.25
    restToAdd = m21.note.Rest()
    restToAdd.duration.quarterLength = 0.75
    self.kempli.append(kempliToAdd)
    self.kempli.append(restToAdd)

# Add a list of tones (pokoks) to to the pokoko part
def addPokoks(self, pokoks, addKempli=True):
    #for tone in pokoks:
    for i in range(len(pokoks)):
        self.addPokok(pokoks[i])

        if (i % self.numCells == 0) and (i != 0) and self.addRests:
            # Add a rest!
            restToAdd = m21.note.Rest()
            restToAdd.duration.quarterLength = 2.0
            self.pokokPart.append(restToAdd)
            # if addKempli:
            #     restToAdd = m21.note.Rest()
            #     restToAdd.duration.quarterLength = 2.0
            #     self.kempli.append(restToAdd)

```

```

def addAllKempli(self):
    for i in range(self.numCells*self.numCycles):
        self.addKempli()
        if (i % self.numCells == 0) and (i != 0) and self.
            addRests:
                restToAdd = m21.note.Rest()
                restToAdd.duration.quarterLength = 2.0
                self.kempli.append(restToAdd)
        self.addKempli()
    self.addKempli()

# Add a list of tones to the player1 part
# will need to modify so that tones can be added for all parts
# currently notes are added at octave 5 but need to check on
    that
# Notes are 16th notes and thus have length 0.25 ( a quarter of
    a quarter )
def addNotes(self, toneList, player_num):
    # get correct player:
    switcher = {
        1: self.player1,
        2: self.player2,
        3: self.player3,
        4: self.player4,
    }
    player = switcher.get(player_num, None)
    if player == None:
        raise ValueError("Incorrect player number: '" + str(
            player_num) + "'\n
            nPlease choose from 1,
            2, 3, or 4")

    for i in range(len(toneList)):

        if (i % (self.numCells*8) == 0) and (i != 0) and self.
            addRests:
                # Add a rest!
                restToAdd = m21.note.Rest()
                restToAdd.duration.quarterLength = 2.0
                player.append(restToAdd)

        noteNameToAdd = toneList[i].getNoteName()
        octave = self.getOctave(player_num, toneList[i].
            getToneName())

        if noteNameToAdd == "rest":
            noteToAdd = m21.note.Rest()
        else:
            noteToAdd = m21.note.Note(noteNameToAdd + str(octave
                ))
        noteToAdd.duration.quarterLength = 0.25
        player.append(noteToAdd)

```

```

# Given a player number and a tone name, returns the octave that
# tone should be in
def getOctave(self, player_num, toneName):
    if player_num == 1:
        return 4
    elif player_num == 4:
        return 6
    else:
        return 5

# Plays the song in the local midi player
def displaySong(self):
    self.fullScore.show("midi")

# Saves the song to the given filePath
def saveSong(self, filePath):
    self.fullScore.write("midi", filePath)

```

C.6 Driver.py

Driver.py is the file that runs the full model. It prompts the user for input, creates and transitions each layer of the model, and handles all output. To run the program, simply run “python driver.py”.

```

import Tone as tn
import Cell as cl
import MarkovModel as mv
import ModifiedMarkov as mm
import Song as Song
import numpy as np

# Returns the first layer Markov Model of the different pokok tones
def getFirstModel():
    tone1 = tn.Tone('i')
    tone2 = tn.Tone('o')
    tone3 = tn.Tone('e')
    tone4 = tn.Tone('u')
    tone5 = tn.Tone('a')
    states = [tone1, tone2, tone3, tone4, tone5]
    # Transition probabilities come from bottom-up analysis from
    # Tilley's study of Alit
    matrix = [
        [0/11, 2/11, 4/11, 1/11, 4/11],
        [3/11, 1/11, 1/11, 3/11, 3/11],
        [3/16, 5/16, 3/16, 0/16, 5/16],
        [3/13, 2/13, 1/13, 5/13, 2/13],
        [2/17, 1/17, 6/17, 4/17, 4/17]
    ]
    startingPitches = [1/5, 1/5, 1/5, 2/5, 0]
    model = mv.MarkovModel(states, matrix, initialProb=
        startingPitches)
    return model

```

```

# Returns the second layer markov model of cell structures.
# Currently only uses delayed pokok or advanced pokok structures.
# Probabilities are super vaguely estimated top-down
def getSecondModel():
    cell10 = cl.CellStructure([0,0,0,0,0,0,0,0,0])
    cell11 = cl.CellStructure([1,0,0,0,0,0,0,0,0])
    cell12 = cl.CellStructure([0,1,0,0,0,0,0,0,0])
    cell13 = cl.CellStructure([1,1,0,0,0,0,0,0,0])
    cell14 = cl.CellStructure([0,0,1,0,0,0,0,0,0])
    cell15 = cl.CellStructure([0,0,0,1,0,0,0,0,0])
    cell16 = cl.CellStructure([0,0,0,0,1,0,0,0,0])
    cell17 = cl.CellStructure([0,0,0,0,0,1,0,0,0])
    cell18 = cl.CellStructure([0,0,0,0,0,0,1,0,0])
    cell19 = cl.CellStructure([0,0,0,0,0,0,0,1,0])
    cell19 = cl.CellStructure([0,0,0,0,0,0,0,0,1])
# template delayed advanced both anticipation suspension second-
# half1 second-half2 reverse-first
# reverse-second
    states = [cell10, cell11, cell12, cell13, cell14, cell15, cell16, cell17,
              cell18, cell19]
    matrix = [ [0.15, 0.20, 0.15, 0.04, 0.15, 0.15, 0.04, 0.04, 0.04,
                0.04],
               [0.08, 0.20, 0.08, 0.20, 0.08, 0.16, 0.05, 0.05, 0.05,
                0.05],
               [0.28, 0.00, 0.28, 0.00, 0.16, 0.08, 0.05, 0.05, 0.05,
                0.05],
               [0.28, 0.00, 0.28, 0.00, 0.12, 0.12, 0.05, 0.05, 0.05,
                0.05],
               [0.08, 0.08, 0.16, 0.16, 0.24, 0.08, 0.05, 0.05, 0.05,
                0.05],
               [0.08, 0.16, 0.08, 0.16, 0.08, 0.24, 0.05, 0.05, 0.05,
                0.05],
               [0.10, 0.05, 0.15, 0.10, 0.05, 0.05, 0.15, 0.15, 0.05,
                0.15],
               [0.10, 0.05, 0.15, 0.10, 0.05, 0.05, 0.15, 0.15, 0.05,
                0.15],
               [0.10, 0.15, 0.05, 0.10, 0.15, 0.15, 0.05, 0.05, 0.10,
                0.10],
               [0.10, 0.05, 0.15, 0.05, 0.05, 0.05, 0.15, 0.15, 0.10,
                0.15]
             ]
    initial = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    model = mv.MarkovModel(states, matrix, initialProb=initial)
    return model

# Method that handles the second model to create list of cells and
# add them
# to song, called once per player
def addPlayer(playerNum, totalCells, pokokList, mySong):
    cellModel = getSecondModel()
    # Transition numCells-1 times to get n cell patterns
    cellStructureList = cellModel.transition(n=totalCells-1)

    # Loop through the CellStructure list to create the cell list of
    # template notes for the

```

```

                                pattern
cellList = []

for i in range(totalCells):
    cellToAdd = cl.Cell(pokokList[i], pokokList[i+1],
                       cellStructure=
                       cellStructureList[i].
                       getList())

    cellList.append(cellToAdd)

# Create a modified markov model using the cellList for each
                                player
player = mm.ModifiedMarkov(playerNum, cellList)

# transition the modified markov model to get the full toneList
                                and letterList, print them out
toneList, _ = player.getFullList(verbose=True)

# Add toneLists to the song
mySong.addNotes(toneList, playerNum)

# Main Driver for the project
if __name__ == "__main__":
    numCells = int(input("Enter number of cells per cycle: "))
    numCycles = int(input("Enter number of cycles: "))

    addRests = False
    rests = input("Rest after each cycle? (y/n): ")
    if rests == "y":
        addRests = True

    playPokoks = False
    pp = input("Play the Pokok tones? (y/n): ")
    if pp == "y":
        playPokoks = True

    playKempli = False
    pp = input("Play the Kempli beats? (y/n): ")
    if pp == "y":
        playKempli = True

    totalCells = numCells*numCycles
    mySong = Song.Song(numCells, numCycles, addRests=addRests)
    pokokModel = getFirstModel()

# Transition only numCells-1 times since last note has to be
                                same as first note.
pokokList = pokokModel.transition(n=numCells - 1)
# Repeat the pokoks for each cycle
for j in range(numCycles-1):
    for k in range(numCells):
        pokokList.append(pokokList[k])
# Add the final pokok tone as the starting tone

```

```

pokokList.append(pokokList[0])

# Print out the list of pokok tones
print("Pokok Tones   : ", end = '')
print(*pokokList, sep='          ')

# Add pokoks to song
if playPokoks:
    mySong.addPokoks(pokokList)

# Add Kempli to song
if playKempli:
    mySong.addAllKempli()

# Add all four players to song (see addPlayer helper method)
for i in range(4):
    addPlayer(i+1, totalCells, pokokList, mySong)

playSong = input("Play song? (y/n): ")
if playSong == "y":
    # Play the song in a midi player
    mySong.displaySong()

saveSong = input("Save song? (y/n): ")
if saveSong == "y":
    fileName = input("File name: ")
    # Save a song to a midi file
    mySong.saveSong("../..../music/%s.mid" % fileName)

```