1-1-2018

Algorithmic Music Generation for Pedagogy of Sight Reading

Ryan Stephen Davis
University of Denver

Algorithmic Music Generation for Pedagogy of Sight Reading

Abstract

Autodeus is the name of the program that has been developed and was designed to aid guitar students in the attainment and betterment of musical notation sight reading skills. Its primary goal is to provide a very flexible tool that has the ability to generate virtually endless types of sight reading exercises at many various skill levels.

A complimentary 2 year-long comprehensive guitar sight-reading course syllabus can be implemented via Autodeus as it is capable of generating all the necessary exercises. It is able to generate these exercises quickly and efficiently through the use of a back tracking algorithm that utilizes modular rules. In addition to these modular rules, it allows users many input options: 9 different generation methods, the option of which strings and frets will be utilized, the ability to input types of musical intervals and their probabilities, the ability to input types of note durations and their probabilities, and the ability to input the probability of rests, among others. It also has the ability to generate two voice species counterpoint that abides by classical music rules.

A subset of Autodeus' capabilities are now available on a public URL for free. The genesis of this type of online application is highly beneficial to any guitarist looking to expand and improve their sight-reading capabilities as there is currently no system available that provides all of the developed features. It is highly encouraged by any reader to go to this website and experiment with the various available inputs and generation types.

Document Type
Thesis

Degree Name
M.S.

Department
Computer Science

First Advisor
Mario A. Lopez, Ph.D.

Second Advisor
Ricardo Iznaola, Ph.D.

Third Advisor
Ramakrishna Thurimella, Ph.D.

Keywords
Algorithmic, Generation, Guitar, Music, Sight reading

Subject Categories
Computer Sciences | Music Pedagogy | Programming Languages and Compilers | Software Engineering

Algorithmic Music Generation for Pedagogy of Sight Reading

A Thesis

Presented to

the Faculty of the Daniel Ritchie School of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

by

Ryan S. Davis

November 2018

Advisor: Mario A. Lopez

Author: Ryan S. Davis
Title: Algorithmic Music Generation for Pedagogy of Sight Reading
Advisor: Mario A. Lopez
Degree Date: November 2018

Abstract

Autodeus is the name of the program that has been developed and was designed to aid guitar students in the attainment and betterment of musical notation sight reading skills. Its primary goal is to provide a very flexible tool that has the ability to generate virtually endless types of sight reading exercises at many various skill levels.

A complimentary 2 year-long comprehensive guitar sight-reading course syllabus can be implemented via Autodeus as it is capable of generating all the necessary exercises. It is able to generate these exercises quickly and efficiently through the use of a back tracking algorithm that utilizes modular rules. In addition to these modular rules, it allows users many input options: 9 different generation methods, the option of which strings and frets will be utilized, the ability to input types of musical intervals and their probabilities, the ability to input types of note durations and their probabilities, and the ability to input the probability of rests, among others. It also has the ability to generate two voice species counterpoint that abides by classical music rules.

A subset of Autodeus' capabilities are now available on a public URL for free. The genesis of this type of online application is highly beneficial to any guitarist looking to expand and improve their sight-reading capabilities as there is currently no system available that provides all of the developed features. It is highly encouraged by any reader to go to this website and experiment with the various available inputs and generation types.

Acknowledgements

The two most important people involved in this project are Dr. Mario Lopez (professor of Computer Science) and Dr. Ricardo Iznaola (professor and chair of Lamonts Guitar and Harp Department). It is humbling and inspiring to work with men of such accomplishment and brilliance. Having meetings and discussing the project provided limitless possibilities and never-ending good ideas. It has been one of the biggest opportunities of my life to work with them and I've felt insanely lucky to be in a position where I get to help create a bridge between the worlds of music and computer science. Without these awesome human beings, none of this would've been possible.

Kyle Nesslein was an undergraduate student in the computer science department who initially started this project. I would like to thank him very much for his continued work and communication with us about the project even long after graduation. It is very apparent to me how much work and time was put into creating a great foundation for the software. His research into many open-source projects and libraries definitely helped the whole project take off. The architecture and design patterns laid down by him in the first stages of the code have proven to be highly re-usable and flexible. Thanks!

Andrew Hannum is a friend and a graduate student in the computer science department that helped with the HTML/CSS/Javascript portion of the frontend. This provided a huge leap forward in progress as my lack of these types of skills causes me to get lost pretty quickly in that realm of software. Without him, there may very likely be no slick user interface or a clunky one at best. Thanks!

Lastly, I'd like to thank Dr. Ramakrishna Thurimella, professor and chair of the Computer Science Department. In late summer of 2011, I embarked on a road trip by myself with no destinations. I had been thinking about the possibility of graduate schools and so stopped in Denver to meet with Ramki. That meeting profoundly changed the direction of my life and it really astonishes me how fast time has gone since then. I've learned much valuable knowledge and have met and worked with countless great people as a result. I could not possibly thank him enough.

Contents

List of Figures

Chapter 1

Introduction

Statement of Purpose

Autodeus' primary goal is to serve as an extremely flexible pedagogical tool that outputs virtually endless guitar sight reading exercises for a 2 year-long sight-reading course syllabus developed by Ricardo Iznaola. A copy of this syllabus can be found in the sample syllabus chapter and serves as comprehensive guide for which parameters students should use as input to Autodeus. The coupling of the web application and syllabus fills an empty niche that is currently unavailable in any form for guitarists seeking to improve their sight-reading abilities.

Traditionally, a music student learning guitar will sight read exercises from a highly finite set of music resources such as books and paper sheet music. Sans the inefficiencies of maintaining such a collection, inevitably the student will encounter three issues: 1) The current piece of music they've set in front of themselves is too difficult, 2)The current piece is too easy, or 3) They have previously practiced the piece. Incurring issue number one will frustrate and deter a student from practicing. The second, will not challenge the student sufficiently, thus they will not improve. Lastly, the third issue, is simply not sight reading. Focused on creating many pitch, rhythmic, and harmonic variations, Autodeus is designed to assure the student will always be seeing and hearing new exercises at a skill level within a certain Goldilocks zone ... not too hard, not too easy, but just right, allowing a student to continually progress.

Cope defines creativity as "The initialization of connections between two or more multifaceted things, ideas, or phenomena hitherto not otherwise considered actively connected." [1] His studies use many types of algorithmic generation strategies to create music that attempts to pass what could be thought of as a musical Turing test. Autodeus makes no attempt to approach this depth and breadth. Unfortunately, the music it produces would not sound creative and be bland at best to the musically trained ears. On the other hand, it does follow suit with much of Cope's work with respect to the narrowing of parameters involved in the various elements of musicality: "pitch, rhythm, timbre, articulation, phrasing, dynamics, and so on, to name but a few" [1] This allows Autodeus and it users to focus on only pitch, rhythm, and harmony.

Other Sight Reading Software

With the proliferation of the Internet and smart phone apps, many various software packages have been developed to help students learn to sight read music. There's a wealth of sight reading apps that do not focus on the guitar, but instead intend to produce exercises in general for all instruments and these miss the point completely of what Autodeus achieves. Ricardo Iznaola's syllabus is designed for focused work on specific differing areas of the guitar fretboard as a student progresses, until they've mastered the entirety.

The following are some of the most popular guitar sight-reading web apps (according to Google search results) that are currently available and reasons why they fail to achieve the functionality of Autodeus.

- Sight Reading Factory[2]

  This website has a very nice interface and a ton of options when generating exercises for the guitar, but falls short of Autodeus due to the fact that one cannot input strings and frets.

- Practice Sight Reading[3]

  This falls short also due to the lack of being able to input strings and frets.

- http://thefluentguitarist.comThe Fluent Guitarist[4]

  This has some great guitar sight-reading exercises in a searchable database, but does not generate anything on the fly with sets of input parameters. This would surely present students with the type of situation where they find themselves outside of the Goldilocks zone.

- SmartChord

  This is a nice little app that offers a ton of features to help students learn the guitar, including input of frets and strings to show scales/chords/arpeggios, but does not generate sight-reading exercises.

Musical and Mathematical Connections

Pitch and Harmony

The music harmonic theory utilized by Autodeus is based on classical and Western musical styles. In these, the most common tuning system for the past few hundred years has been twelve-tone equal temperament, which divides frequencies within an octave (frequencies double or half of each other) into 12 parts, all of which are equal on a logarithmic scale. These 12 notes then form what is called a chromatic scale which names all the notes using the letters A through G with combinations of

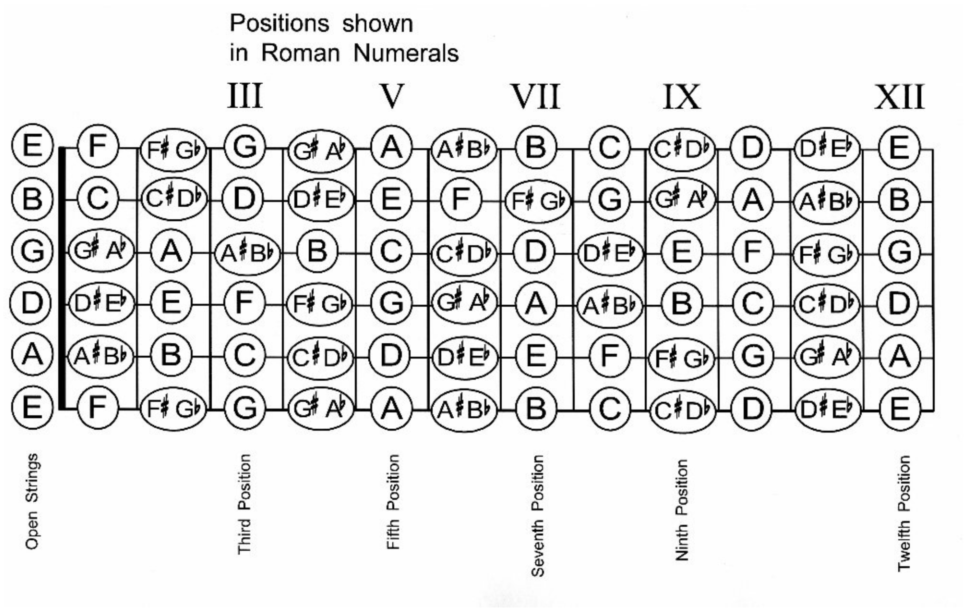


Figure 1.1: Notes on Guitar Fretboard in Standard Tuning

the symbols ♯ and ♭. The naming scheme is arbitrary, but their widespread common acceptance is based on a long history of musical composition. A guitar fretboard (in standard tuning) can be seen in Figure 1.1 and has all of the notes of the chromatic scale and their corresponding locations labeled.

Almost all of the musical theory in this document and the code to implement Autodeus relies heavily on the notion of a musical interval. This is simply a distance between 2 notes where (in a vast majority of western music) the smallest atomic distance is called a half-step. The interval of a half-step translates visually to the fretboard of a guitar in an elegant manner as each fret up or down is a half step from the previous note as can be seen in Figure 1.2. In musical notation a ♯ indicates a note half-step up the chromatic scale and a ♭ indicates half-step down. All other intervals can be defined as a sum of half steps where each has one or more verbose names and a shorthand notation in the form of a possible ♭ or ♯ concatenated with a digit. Some of the most common are the following with their corresponding number of half-steps listed:

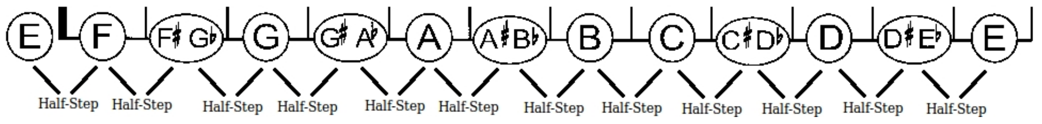


Figure 1.2: E String on Fretboard with Half-Steps

- Half Step, Minor 2nd or ♭2 - 1 half step is the smallest interval
- Whole Step, Major 2nd or 2 - 2 half steps
- Minor 3rd or ♭3 - 3 half steps
- Major 3rd or 3 - 4 half steps
- Perfect 4th or 4 - 5 half steps
- Augmented 4th or ♯4 - 6 half steps
- Diminished 5th or ♭5 - 6 half steps
- Perfect 5th or 5 - 7 half steps
- Minor 6th or ♭6 - 8 half steps
- Major 6th or 6 - 9 half steps
- Minor 7th or ♭7 - 10 half steps
- Major 7th or 7 - 11 half steps
- Octave - 12 half steps, Both notes have the same name, but are double or half each other's frequencies

Note that there are intervals larger than an octave, but for the purposes of Autodeus, intervallic considerations were made only for those enumerated here.

There have been vast amounts of connections made between music and mathematics, in particular graph theory and abstract algebra.[5] Some of these connections

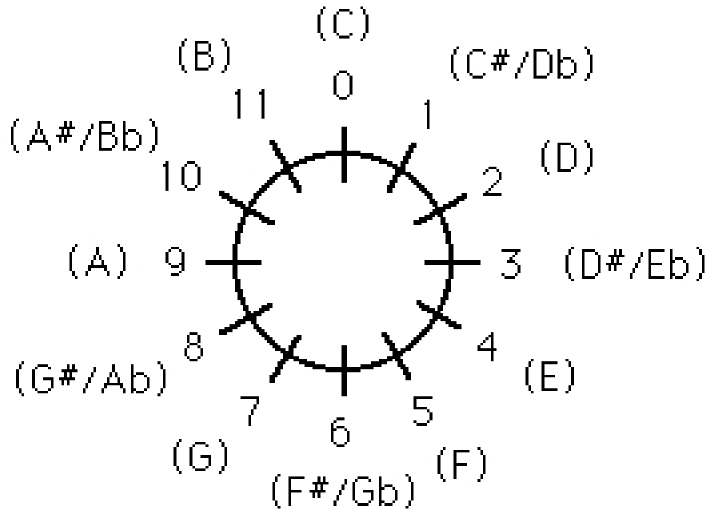


Figure 1.3: Chromatic Scale with C as Root Note in Modular 12 Arithmetic

were used to an advantage when programming Autodeus. A mapping of note names to integers was extremely useful in manipulation of notes and in understanding the relationship between them. The lowest note Autodeus is capable of producing is an A♭0 (also named G♯0) which is a half-step lower than the lowest note (A0) on a standard 88 key piano. Thus, A♯0/G♭0 is mapped to an integer value of 0, A0 is mapped to 1, A♯/B♭ mapped to 2, and so on and so fourth up to the maximum positive integer value which can describe notes far past the human hearing frequency range. Each half-step can be thought of as an increment of +/-1 and any other intervallic relationship between any given two notes is easily found using modular 12 arithmetic. Consider Figure 1.3 where moving around the circle clockwise is ascending the note frequencies and visa versa. For example, we can say that C to G ascending is an interval of a perfect 5th (7 half-steps) and descending from C to G is an interval of a perfect 4th (5 half-steps).

A scale is a set of notes starting from an initial note called the root which ascends and/or descends to an octave above or below. Using any combination of the aforementioned intervals relative to a root note, one can define a scale. This is akin to starting on 0 in Figure 1.3 and picking notes around the circle until we arrive back to the top and the circle can be thought of as a guitar string where each integer value is a fret. Obviously, there are many permutations that can be generated in this manner. There are resources widely available which enumerate many guitar scales and their patterns. [6] [7]

7-note scales in Western music are often studied and can be defined in 2 ways: a series of half-steps (difference of 1) notated as "H" and whole-steps (difference of 2) notated as "W" that separate each adjacent note starting from a root note, or by declaring each interval present relative to the starting root note (R notates root note). One the most common scales in Western music is the Major scale which can be described as in steps as W-W-H-W-W-W-H or intervals present as R-2-3-4-5-6-7-R. If we apply this sequence to our modular 12 note circle starting with C and moving clockwise, we see attain the notes C-D-E-F-G-A-B-C which are conveniently all of the white keys on a piano. See Figures 1.4 and 1.5 for a visual representation of these derivations.

From a major scale, 6 other scales can be derived that are known as modes of the major scale. This is done by selecting a note in the major scale, considering it a new root note, going around the modular 12 circle clockwise, and noting the sequence of steps and intervals produced using the same initial notes. For example, a mode called Dorian starts on the second note of the major scale and goes around producing the steps W-H-W-W-W-H-W and intervals R-2-♭3-4-5-6-♭7-R. Notice how Dorian mode now includes 2 of the minor intervals, ♭3 and ♭7. See Figure 1.6 for a visual representation of the D Dorian mode.

One can repeat this process for the other notes of the major scale and derive Figure 1.7 which is a table that enumerates all of the names and possible modes of the major scale and their intervals. It should be noted that Ionian is the modal

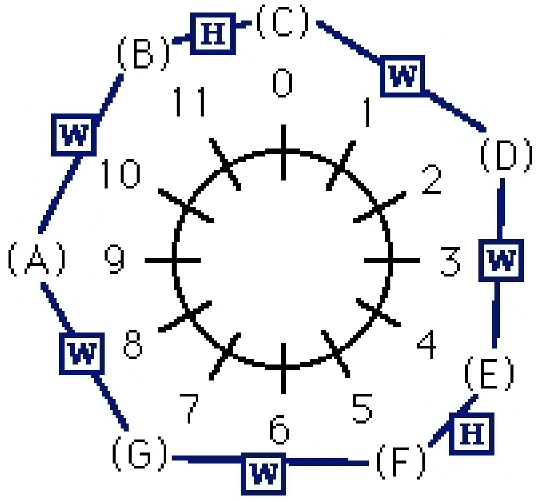


Figure 1.4: C major scale with Half and Whole Steps in Modular 12 Arithmetic

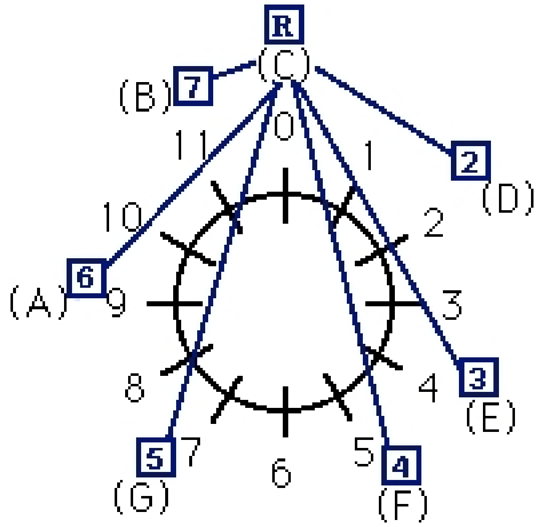


Figure 1.5: C major scale with Intervals in Modular 12 Arithmetic

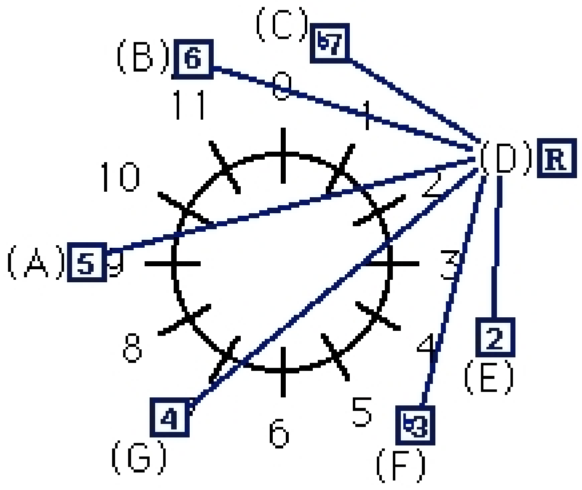


Figure 1.6: D Dorian Mode with Intervals in Modular 12 Arithmetic

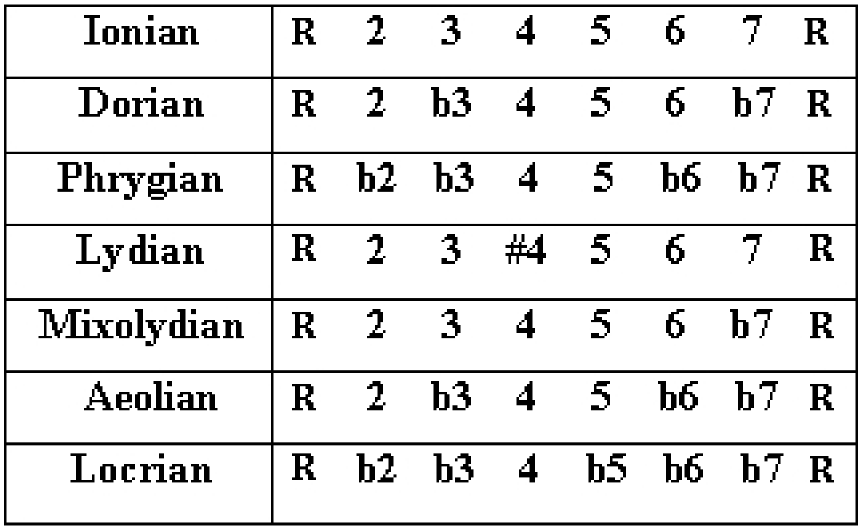
name for the major scale. Modes can be thought of as "flavors" where each has characteristic sounds. Autodeus is able to generate exercises in any mode starting from any root note as input parameters.

Rhythm

Musical rhythm notation has many connections with mathematics as it denotes the amount of time to play (or not play) notes and tells us how a set amount of time is sub-divided. The following are common musical rhythm notation terms and their definitions which should be understood in order to comprehend how Autodeus generates rhythm. (Please note that this section is meant to define these concepts within the context of this thesis and Autodeus' code as there are likely many other definitions and ways to explain them.)

1. Tempo - Defines the number of pulses (or beats) per minute used to count the speed of music. This is in units of Beats Per Minute (BPM).

2. Measure - Segment of time comprised of a specific number of beats whose contents is contained within 2 vertical lines in musical notation See Figure 1.9



Ionian	R	2	3	4	5	6	7	R
Dorian	R	2	b3	4	5	6	b7	R
Phrygian	R	b2	b3	4	5	b6	b7	R
Lydian	R	2	3	#4	5	6	7	R
Mixolydian	R	2	3	4	5	6	b7	R
Aeolian	R	2	b3	4	5	b6	b7	R
Locrian	R	b2	b3	4	b5	b6	b7	R

Figure 1.7: Modes and their Intervals

3. Whole Note - Notation that means play a note for 4 beats

4. Half Note - play a note for 1/2 of a Whole Note's value or 2 beats

5. Quarter Note - play a note for 1/4 of a Whole Note's or 1 beat

6. Eighth Note - play a note for 1/8th of a Whole Note's or 1/2 of a beat

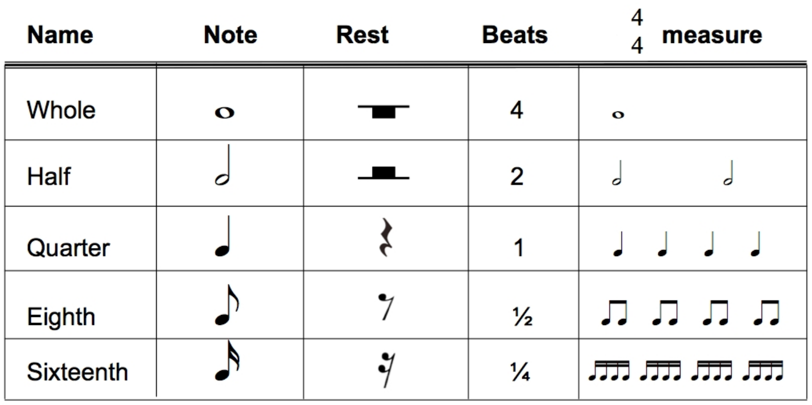
7. 16th Note - play a note for 1/16th of a Whole Note's or 1/4th of a beat

8. 32nd Note - play a note for 1/32th of a Whole Note's or 1/8th of a beat

9. 64th Note - play a note for 1/64th of a Whole Note's or 1/16th of a beat

10. Rest - A type of notation used to indicate that silence or nothing should be played for a certain amount of time. All of these have a direct correlation to the aforementioned Note values. See Figure 1.8

11. Time Signature - A musical notation comprised of 2 numbers that look like a numerator and a denominator in a fraction. The numerator defines how many beats there are per measure and the denominator defines which of the aforementioned note values is used for each of those beats.



Name	Note	Rest	Beats	$\frac{4}{4}$ measure
Whole	𝅝	𝄻	4	𝅝
Half	𝅗𝅥	𝄼	2	𝅗𝅥 𝅗𝅥
Quarter	♩	𝄽	1	♩ ♩ ♩ ♩
Eighth	♪	𝄾	½	♫ ♫ ♫ ♫
Sixteenth	𝅘𝅥𝅯	𝄿	¼	♬♬ ♬♬ ♬♬ ♬♬

Figure 1.8: Notation of Note Durations and Rests

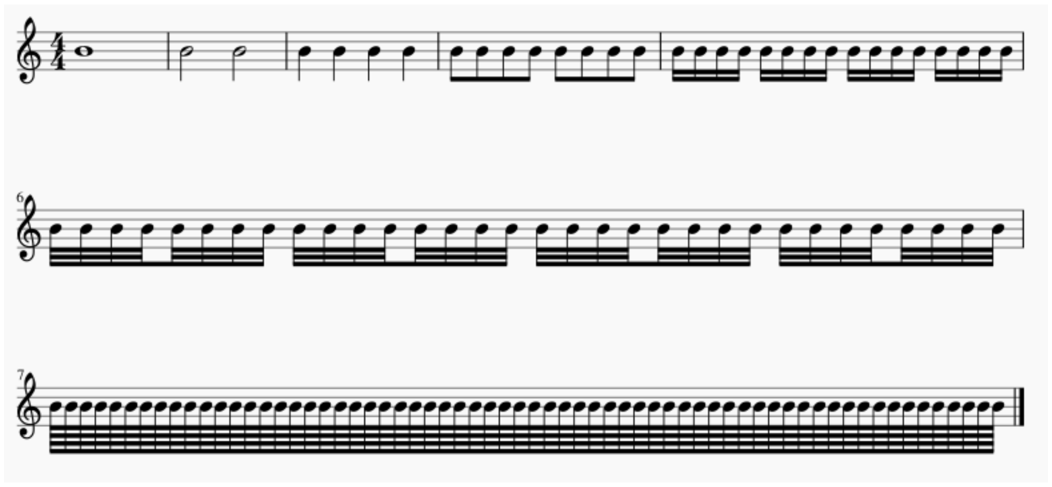


Figure 1.9: Note Durations in 4/4 Measures

As an example, consider one of the most common time signatures, 4/4. The numerator tells us that there are 4 beats per measure and the denominator tells us to use quarter notes. In other words, 1 measure = 4 * (1/4) notes. Figure 1.9 Another common time signature is 6/8 which indicates 1 measure = 6 * (1/8) notes or 6 eighth notes per measure. See Figure 1.10

Another set of musical notation commonly used is the dot notation. A dot appended to a note or rest indicates to add half its value to the amount of time to play the note. For example, a quarter note (1 beat) with a dot indicates to play

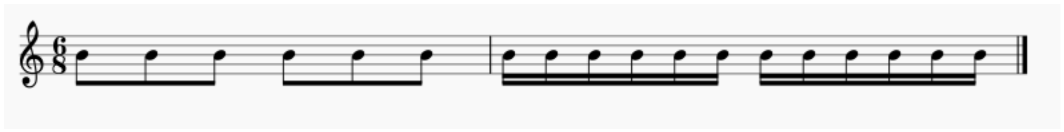


Figure 1.10: Measures in 6/8

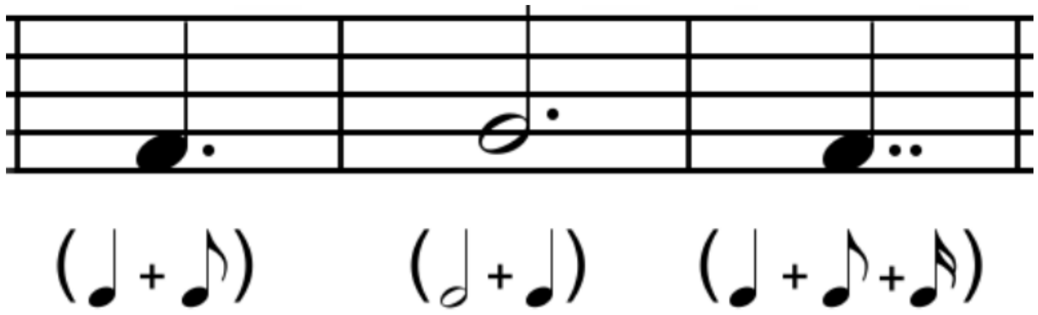


Figure 1.11: Dot Notation for Notes and Rests

the note for 1 and a half beats (a quarter note plus an eighth note). Although less commonly used, if 2 dots are appended to the note, this indicates to add half the value of the first dot to the overall time. Adding more dots can be repeated recursively, but this would be very rare due to its difficulty to interpret by human sight readers. See 1.11 for visual examples.

Lastly, another common musical rhythm notation is a tie. A tie is used to denote that note values should be added together. For example (see Figure 1.12), a half note (2 beats) tied with a quarter note (1 beat) denotes that the note should be played for a total of 3 beats. This can also be used to denote that a note should be played over a measure bar. (See Figure 1.13) for an example of this.

Autodeus is capable of generating and notating any time signature where the denominator is 1, 2, 4, 8, 16, 32, or 64 and this time signature is used as an input parameter. It is also capable of notation using dot notation and ties.

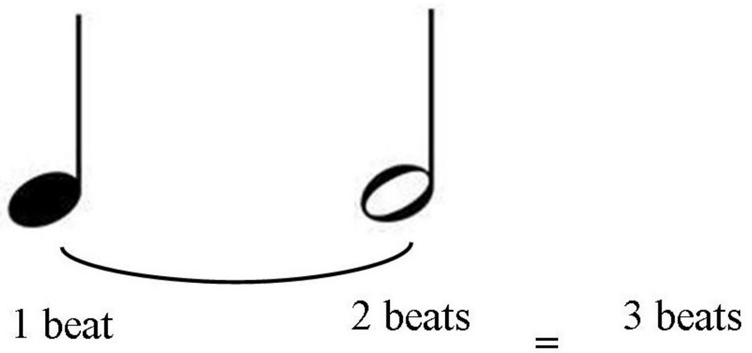


Figure 1.12: Tied Notes Examples

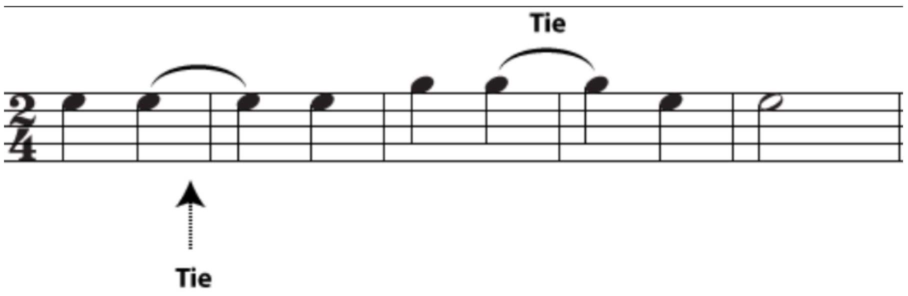


Figure 1.13: Tied notes over Measure Bar

Chapter 2

System Architecture

Overview

Autodeus has several layers of software that allow it to run with the interface at its public URL. This chapter describes how all of these layers interact to create the whole and how one can setup the system to run a server on a local computer or on a headless web server. Figure 2.1 visually summarizes the entire software stack used.

Code Repository

All of the code necessary to get an instance of Autodeus running is available on a private Github[8] repository at the following URL (Email me at ryandavisbusiness@gmail.com to request access):

https://github.com/ryansdavis/GenLessonRefactored.git

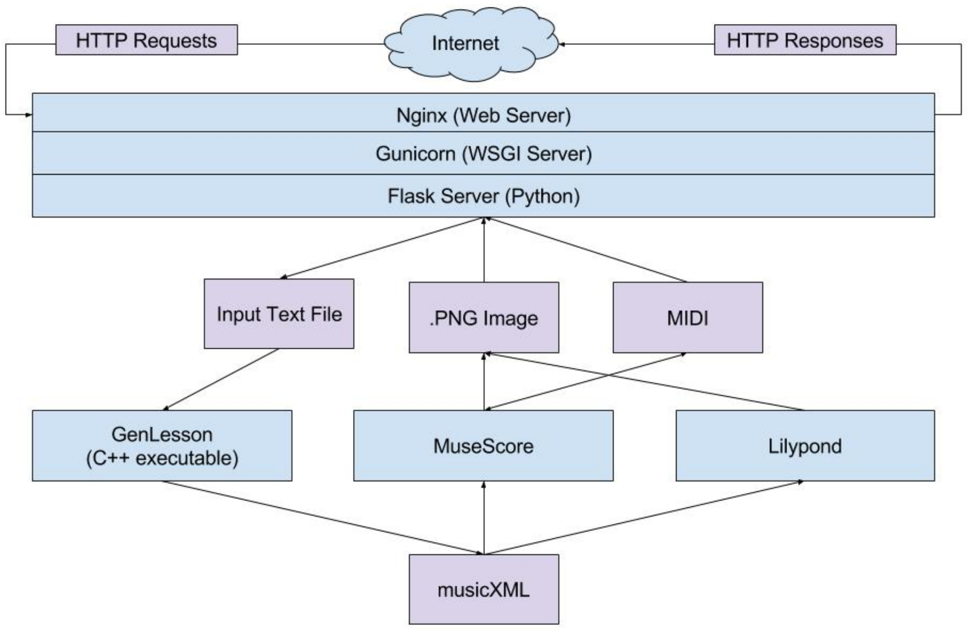


Figure 2.1: Overall System Architecture

Clone the repository to a desired location locally to view the repository's contents which are the following directories:

- GenLesson Contains all of the source code, scripts, and other text files written for Autodeus. Contains the following sub-directories:
  - trunk/

    This directory holds all of the C++ source code for the executable "GenLesson".
  - build/

    Location for scripts and build files when compiling "GenLesson"
  - GenerateLessonFiles/

    Location containing scripts for testing/running the "GenLesson" executable and a directory full of example input text files.

  - frontend/

    This directory holds all of the Python/JavaScript/HTML/CSS source code for running the front end web app along with deployment scripts and configuration.

- libmusicxml-3.00-ubuntu-x86_64 Contains all the source files, shared object library, and other documentation for the musicXML C++ library

- Writeup Contains all of the documentation for this project

Open Source Dependencies

Nginx/Gunicorn/Flask(Python3)

The stack of Nginx/Gunicorn/Flask(Python) serves as the front-end to Autodeus. This layer of software is quite common on modern-day production systems for serving up static and dynamic content on websites.[9] Flask itself is a Python micro-framework that can handle only one HTTP request at a time. Gunicorn is used to spawn several identical Flask worker processes and Nginx forwards all HTTP requests on port 80 to these workers. This stack is then run on an Amazon Web Service's EC2 instance in the cloud, providing access to the public URL.

libmusicXML

Music XML is a widely accepted standard for sharing and editing sheet music online and is compatible with almost all popular music software suites.[10] Autodeus uses an open source musicXML library in the backend which is dynamically linked to an executable called "GenLesson" written in C++.[11] "GenLesson" accepts an input text file that defines all of the input parameters and then outputs a text file in musicXML format.

An open source library is utilized called “libmusicxml” and is dynamically linked during the the build process and during execution of Autodeus[12][13] It must be placed in the directory ‘/usr/local/lib/’ and have a link to it which likely will require super user permissions. There is a script at ‘GenLessonRefactored/libmusicxml-3.00-ubuntu-x86_64/setup_library.sh’ that performs this action. After successful completion of this script, ‘/usr/local/lib’ should contain ‘libmusicxml2.so.3.00’ and have a link to it called ‘libmusicxml2.so’

Lilypond

Lilypond is a project which is based upon a markup language for creating sheet music.[14] Autodeus uses Lilypond to create nicely formatted .PNG images of the musicXML for exercises where the rhythm is strictly pre-defined and codified in the musicXML output. Figure 2.1 shows a block converting musicXML straight to a .PNG image, but this is somewhat of an over-simplification. Autodeus actually converts the musicXML to a ‘.ly’ file first (Lilypond’s markup language) and then calls the “lilypond” binary with arguments to convert the Lilypond markup to a .PNG image. This can be seen in Figure 2.2

MuseScore

MuseScore is a project popular on Linux distributions for creating, reading, and playing sheet music.[15] Autodeus uses MuseScore as a command line utility (sans the GUI) to convert musicXML to MIDI and to .PNG files.

MuseScore has proven very useful in producing properly notated rhythms which simplifies much of the musicXML creation. When an exercise is produced in a manner where rhythm is not pre-defined, MuseScore takes the musicXML, converts it to MIDI and then converts the MIDI to the .PNG image as in Figure 2.3. This

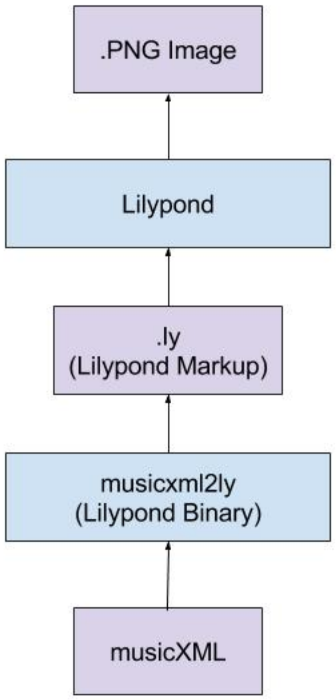


Figure 2.2: Lilypond Format Conversions

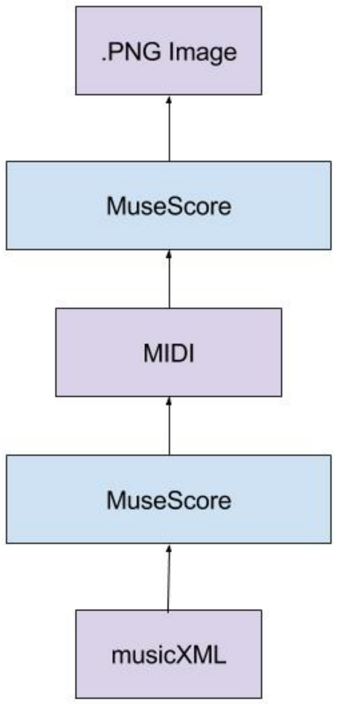


Figure 2.3: MuseScore Format Conversions

allows MuseScore to interpret and produce the rhythms in a more readable manner when outputting the image.

xvfb

Musescore requires an output X windowing system server in order to display its contents. Autodeus, running on an Amazon EC2 instance, has no graphical output and thus no X server. ”xvfb” (X Virtual Frame Buffer) serves as a way for Musescore to run without failure in this environment due to its dependency on the C++ Qt library and graphical output.

Mido

Mido is a Python module (used within the Flask server) for working with MIDI files.[16] This library was used to append a metronomic click track to the beginning of a generated MIDI file. Initially, a user had no indication of when the exercise playback was going to begin after pressing the web app's button that started playing the MIDI. Adding a number of beats according to the time signature before playback improved the user's experience greatly.

Frontend

HTML/CSS/Javascript

The frontend consists of a combination of HTML, CSS, and Javascript. This allows users to access a clean interface to create, play, and listen to guitar sight reading exercises with ease on any device that has a web browser. It works as a Facade design pattern [17], vastly reducing the number of input options available to a user and therefore greatly simplifying the interface. Most of the pages in the web app are static content, except when generating a sight reading exercise. The interface consists of several drop-down menu options and a GENERATE button. Pressing the GENERATE button causes the Javascript to produce an HTTP POST with a JSON argument containing all of the parameters from user input which gets passed to the Python Flask server.

Flask (Python3) Server

When the Flask server receives a JSON POST request from the front end it goes through the following process:

- Validates the JSON parameters and uses them to create an exercise in XML (more on this later)
- Converts the XML to MIDI and adds a click track to the beginning so that the MIDI player on the front end can play the exercise audibly.
- Creates a ".png" image according to the XML produced and passes it back to the front end for the web page to display.

How to Run on a Local Server

For development and debugging purposes, it is nice to have a server running locally on a computer. This sections describes how to get a Flask server running locally which removes the dependencies on Nginx and Gunicorn. The following are steps to setting up a local server:

- Clone the Git repository locally to a development computer. See the Code Repository section if this has not already been done.
- Put the music XML library in place. See section libmusicXML and follow instructions there if the libmusicXML library has not been put in place at this point.
- Run the script at the following to get all necessary open source dependencies installed: 'GenLessonRefactored/GenLesson/frontend/get_local_server_dependencies.sh'

How to Setup and Run on an Amazon Web Server

Autodeus is currently running a headless server setup which provides a public-facing IP address and URL. This public server setup necessitates a couple extra layers of complexity in the setup. The web application uses an Amazon EC2 server with an Ubuntu 16.04 operating system. In order to setup on an Amazon Web

Server, one must first have a running EC2 instance configured to connect with an SSH key pair and make sure its security group allows incoming traffic on ports 22 and 80 (SSH and HTTP, respectively)[18]

- SSH into the EC2 instance
- Clone the Git repository to a preferred location. See the section "Code Repository" on how to do this.
- Put the music XML library in place with reference to the section libmusicXML
- Navigate to 'GenLessonRefactored/GenLesson/frontend/server_setup_and_config/' and with sudo privileges, perform the following:
  - Run the script "get_headless_server_dependencies.sh" with the sudo's "-H" flag and answer affirmative to all prompts. This installs all of the required packages that Autodeus needs.
  - Run the script "config_server.sh". This configures Nginx, Gunicorn, and the Flask app so that the EC2 instance is able to serve the website on a public URL.
  - Run the script "restart_server.sh". This restarts the Nginx and Gunicorn services so they use the most recently made configuration changes.

If everything was successful, the public URL/IP address for the created EC2 instance should display the Autodeus web app in a web browser.

Backend

The engine that runs Autodeus is contained in the executable called "GenLesson". This executable, written in C++, has the responsibility of reading and parsing through an input file containing all of the desired parameters and then generating music XML on the cout output stream after utilizing various algorithms to generate an exercise.

Building the C++ Source Code

Building the executable for Autodeus requires a 64-bit Linux Operating System and following the steps below:

- Use the following command to install all of the required packages:

  sudo apt-get install git cmake build-essential

- Navigate on a command line to where the code repository is to be checked out and clone the Git repository.

- Put the music XML library in place. See section libmusicXML and follow instructions there if the libmusicXML library has not been put in place at this point.

- Run the shell script in the repository, 'GenLessonRefactored/GenLesson/build/rebuild.sh', which compiles and links everything with cmake and make.

- If all of the steps above succeeded, there should now be an executable in the directory 'GenLessonRefactored/GenLesson/build/src/' called "GenLesson"

A working up-to-date executable has been committed into the Git repository. If new changes are made in the code and a successful build is created, this makes it easy to push the locally created "GenLesson" executable to Git, then SSH into the Amazon web server and pull it from the Git repository. "GenLesson" is not terribly large so this is much more convenient than performing the build on an Amazon server.

Running GenLesson

For development debugging and testing, running the executable “GenLesson” by itself is extremely useful. Doing so also exemplifies the full capabilities of Autodeus when the facade front end is not present (The frontend intentionally limits input parameters to users). In the directory ’GenLessonRefactored/GenLesson/GenerateLessonFiles’ there is a script and a directory.

- doAllThatStuff.sh

  Runs “GenLesson” with a given input file and uses Lilypond and MuseScore to create a series of files from the created music XML file that was output from “GenLesson”. This script accepts 2 positional arguments: 1st argument is the path to an input “.fmt” file and the 2nd argument is a string name that will be used for all of the files that are created as a result of running the script (extensions ‘.xml’, ‘.ly’, ‘.mid’, and ‘.png’ files).

- input_files

  Directory with a variety of example input ‘.fmt’ text files. These files are used to specify all of the parameters/constraints that a generator will use when generating an exercise. See the subsection “GenLesson Input Parameters” for full definitions of the key/value pairs in these files.

All logging information is written to stdout throughout the process of the program running and this provides useful insight as to the inner-workings of the code. Many internal boolean flags in the source code may be set to disable/enable logging features upon re-compilation.

Chapter 3

Generating Music XML

This chapter describes the true power and flexibility of Autodeus as it goes into detail of how the executable "GenLesson" operates.

GenLesson Input Parameters

The GenLesson executable built with instructions from the previous subsection "Building the C++ Source Code" accepts 2 positional arguments: the 1st specifies the path to an input '.fmt' file and the 2nd specifies the name of the '.xml' file to be output. While the Flask frontend limits the options available to users to avoid an over-complicated interface and remove many error-prone combinations of inputs, an input '.fmt' file has none of these limitations. To see an example of every single possible input param in a '.fmt' file, refer to the file in the Git repository 'GenLesson/GenerateLessonFiles/input_files/master_input.fmt'. This file has many comments throughout (denoted via a starting '#' character) that explain what all of the possible inputs are and how to format them. The very first key is 'gen_method' which defines which type of generation will be used. All possible generation types and their unique parameters are enumerated in the subsection of this chapter, section

"Generators", but they all have some parameters in common which are defined as follows here:

- title

  Metadata for the musicXML file

- part_name

  Metadata for the musicXML file

- instrument_name

  Metadata for the musicXML file

- composer

  Metadata for the musicXML file

- beats

  Number of beats per measure. In other words, the "numerator" of the time signature

- beat_type

  Beat type for time signature. In other words, the "denominator" of the time signature

- divisions

  A unit-less value which defines the length of a quarter (1 beat) note. This is set to 64 currently as a 64th note is the smallest time division possible.

- key_fifths

  This defines the number of sharps or flats in the key signature to be used. Positive numbers from 0 to 7 add that number of sharps and Negative numbers from -1 to -7 add that many (absolute value) flats

- key_mode

  Mode to be used for the given key signature where the following mapping applies: 0 Ionian, 1 Dorian, 2 Phrygian, 3 Lydian, 4 Mixolydian, 5 Aeolian, and 6 Locrian

- clef_sign

  The clef sign to be used for the exercise. Autodeus always assumes a treble clef at this juncture

- clef_line

  Vertical alignment of the clef sign.

- avail_frets

  A space separated list of fret numbers to be used in creating an exercise. 0 denotes an open string, 1 is first fret, etc... These should be in ascending order and have unique fret numbers, but not all fret integers need to be adjacent. For example, '0 3 5 7 10 12' is valid input.

- avail_strings

  A space separated list of string number to be used in creating an exercise. The string tuning and their numbers are defined with the next set of key/value pairs. Once again, these should be in ascending order and have unique numbers, but not all defined string numbers need to be present.

- string_*

  These parameters define the string numbers and their tuning. This allows Autodeus to create exercises with not only a standard tuned 6 string guitar, but exercises for other stringed and fretted instruments with various tunings and numbers of strings. These must start with 'string_1', 'string_2', ...etc. and the integers must be in ascending order. The two integers values set for these keys denote pitch and octave information, respectively. Pitch starts with 0-C, 1-C#/Db, 2-D, 3-D#/Eb, 4-E,...etc. and octave 3 is the octave of a low E string in standard tuning, 5 is octave of high E string.

- min_num_bars

  The minimum number of measures to generate.

- max_num_bars

  The maximum number of measures to generate. Note that this must be at least 1 greater than min_num_bars or infinite loops will happen.

Object Oriented Design and Data Structures

All of the GenLesson executable was written with C++11 and uses data structures from the standard template library, namely, vector and map.[11] The source code uses a class hierarchy to organize its code which is depicted visually in Figure 3.1 where arrows represent the relationship "depends on". For example, Part depends on Voice as it has Voice objects within it. The following subsections describe these classes from the bottom up. (Please note that the UML diagrams are meant to be an overview of the most important features of each class and are not meant to be comprehensive, that's what the source code is for...)

Note Class

The Note class is the lowest level abstraction in GenLesson as it contains all pertinent information for a single note. This class also has many static functions that were created as helper functions that make comparing notes and understanding their intervallic relationships much more convenient. Figure 3.2 is a UML diagram with its most relevant member variables.

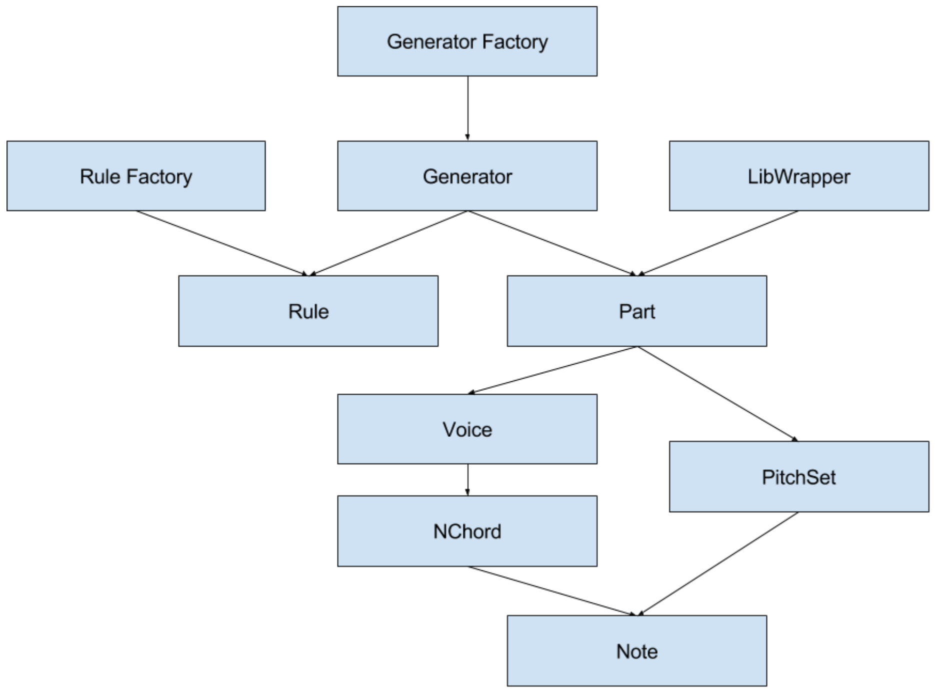


Figure 3.1: Overview of Class Dependencies

- char step - the musical note name,
- int octave - octave number according to scientific pitch notation
- int alter- indicates accidental where a positive number indicates number of sharps and a negative number indicates number of flats
- int string - string where note is located
- int fret - fret where note is located
- bool tieStart - indicates whether or not the note is the start of a rhythmic tie
- bool tieEnd - indicates whether or not the note is the end of a rhtyhmic tie
- int numOfDots - indicates the number of dots to append to note for rhythmic notation

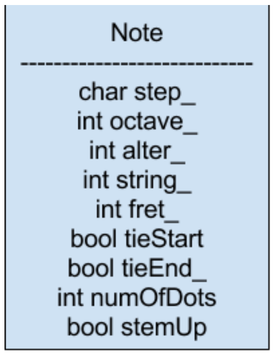


Figure 3.2: Note Class UML

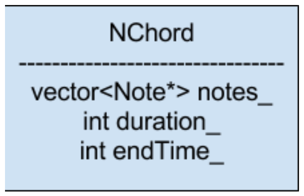


Figure 3.3: NChord Class UML

- bool stemUp - indicates if the stem of the note in the musical notation should go up or down

NChord Class

The NChord has proven to be a useful data structure for the purposes of generation. It has a vector of Notes so that it can hold none or many Notes. If this vector is empty, then it is interpreted as a musical rest when the XML is output. NChord also has 2 integer values: one for its duration and another for its end time. Figure 3.3 is UML diagram showing some of this classes main features.

Voice Class

A voice in music terminology is a set of notes meant to be played by one instrument. This class is a representation of that, containing a vector of NChords. Figure 3.4 is a UML diagram of this class.

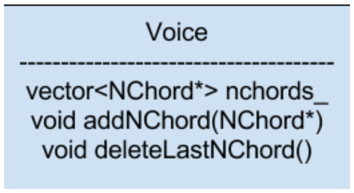


Figure 3.4: Voice Class UML

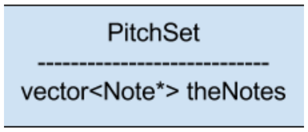


Figure 3.5: PitchSet Class UML

PitchSet

A PitchSet is essentially a wrapper for a vector of notes and its UML diagram can be seen in Figure 3.5.

Part Class

A Part object's responsibility is to act as a data structure that holds all the information for a sight reading exercise. A Part has a vector of Voice objects and, thus, is a composition of Voices, NChords, and Notes. There is a direct correspondence between this data structure's layout and the music XML schema so that it can be parsed by a LibWrapper object into an XML file.

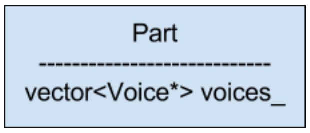


Figure 3.6: Part Class UML

Generator Object

A Generator's responsibility is to take a Part object and fill the data structure with an exercise that abides by the constraints of the input parameters. In the directory 'GenLesson/trunk/src/Generators/', there is an abstract base class which has a pure virtual method 'bool generate(Part* aPart)' that performs this duty.

A Generator's constructor takes input in the form of a text file and upon reading the "input.fmt" file, the Lesson object fills 4 PitchSets class members with Note objects according to the "input.fmt" format file's constraints (among setting many other states and preparing other data depending on the generation type). These PitchSets are as follows:

- masterPitchSet

  Set of notes including all chromatic notes on available strings and frets

- keyPitchNames

  Set of 7 notes in scale, octaves are all -1000

- keyPitchSet

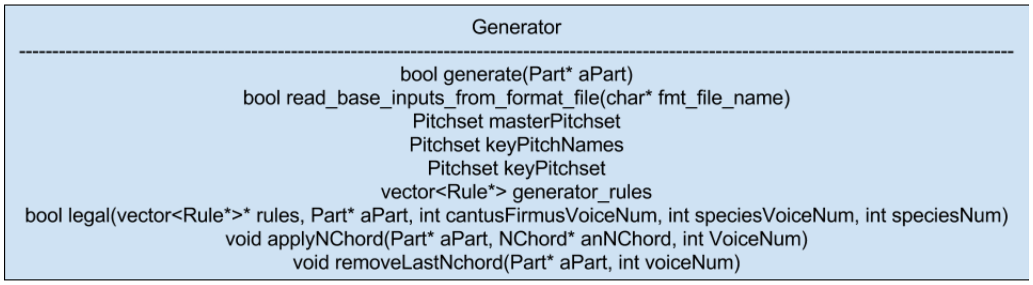
  Set of all notes in a given scale and on available strings and frets

- keyPitchSetM6M7

  A superset of keyPitchset that includes a major 6th for the Aeolian scale mode and a major 7th for Aeolian, Dorian, and Mixolydian scale modes for the purpose of cadential endings in 2 voice counterpoint exercises

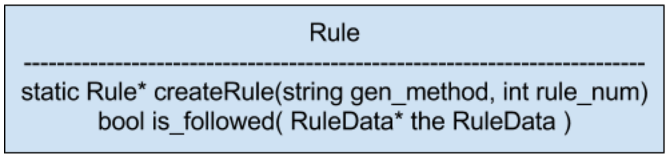
Generator Factory

A Generator Factory's responsibility is to take an input ".fmt" file and return an instantiated Generator object.


Generator

bool generate(Part* aPart)
bool read_base_inputs_from_format_file(char* fmt_file_name)
Pitchset masterPitchset
Pitchset keyPitchNames
Pitchset keyPitchset
vector<Rule*> generator_rules
bool legal(vector<Rule*>* rules, Part* aPart, int cantusFirmusVoiceNum, int speciesVoiceNum, int speciesNum)
void applyNChord(Part* aPart, NChord* anNChord, int VoiceNum)
void removeLastNchord(Part* aPart, int voiceNum)

Figure 3.7: Generator Class UML


Rule

static Rule* createRule(string gen_method, int rule_num)
bool is_followed(RuleData* the RuleData)

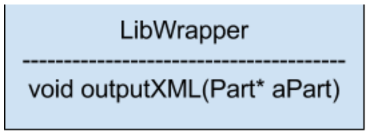
Figure 3.8: Rule Class UML

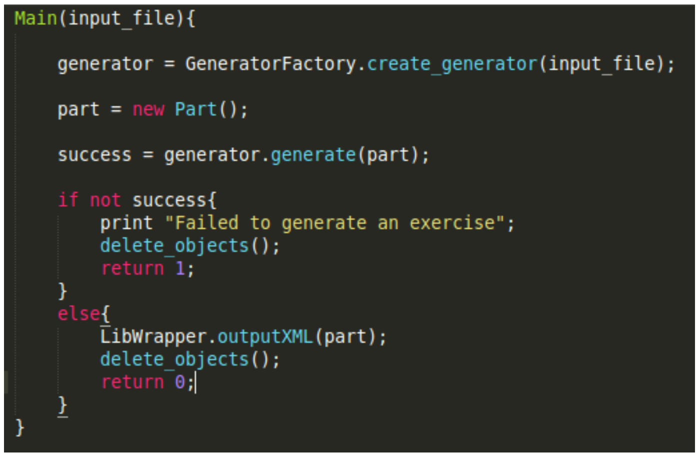
Rule Object

Rule objects are a very important part of Autodeus' ability to be extensible. The abstract rule base class has many static helper functions and, most importantly, a pure virtual function called "bool is_followed(RuleData* the RuleData)". The Generator base class has a vector called "generator_rules" that it iterates through when generating a lesson, calling "is_followed()" on each to validate that all constraints are being kept, thus allowing any rule to be programmed for experimentation and creation of a variety of musical exercises.

LibWrapper Object

This class acts as a wrapper for the libmusicxml library and its responsibility is to parse through a Part data structure and output XML based on the contents of said Part.





```
Main(input_file){

    generator = GeneratorFactory.create_generator(input_file);

    part = new Part();

    success = generator.generate(part);

    if not success{
        print "Failed to generate an exercise";
        delete_objects();
        return 1;
    }
    else{
        LibWrapper.outputXML(part);
        delete_objects();
        return 0;
    }
}
```

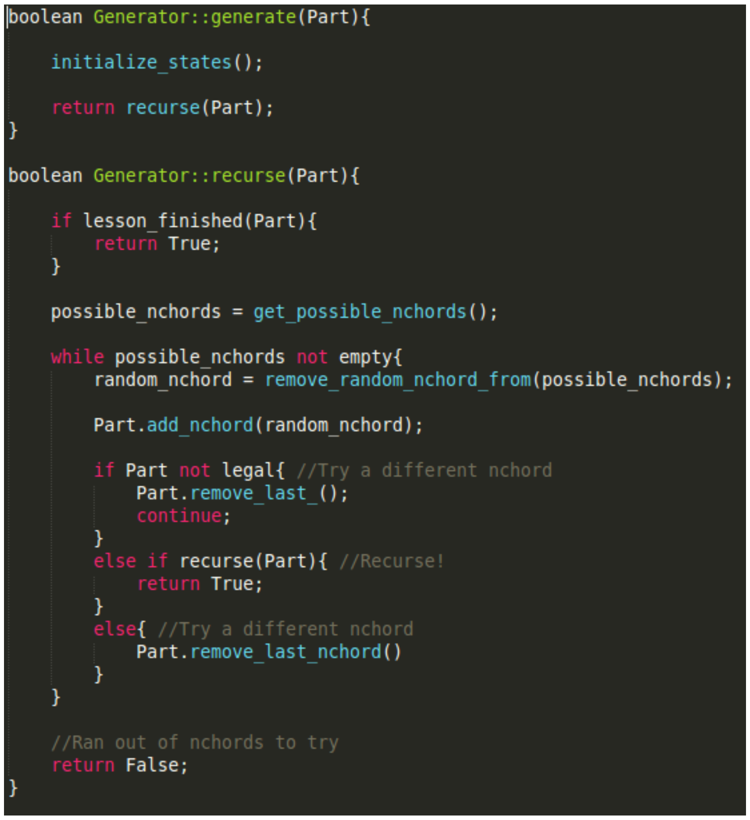
Figure 3.9: Pseudocode for GenLesson Main Function

GenLesson Overview

The highest level pseudocode of how GenLesson works can be seen in Figure 3.9 A Generator instance is created using a GeneratorFactory with a input configuration file as a parameter. A Part instance is then created and passed to the generator's generate method. If this generation is successful, the generated Part instance is used to output a musicXML file.

The Primary Generation Algorithm

The most prominent algorithm used to generate exercises is a recursive back tracking algorithm. The recursion causes the program to take a random walk down a tree of note choices while maintaining that all of the rules are followed. The pseudocode of how this algorithm works can be seen in Figure 3.10 Eventually, the



```
boolean Generator::generate(Part){

    initialize_states();

    return recurse(Part);
}

boolean Generator::recurse(Part){

    if lesson_finished(Part){
        return True;
    }

    possible_nchords = get_possible_nchords();

    while possible_nchords not empty{
        random_nchord = remove_random_nchord_from(possible_nchords);

        Part.add_nchord(random_nchord);

        if Part not legal{ //Try a different nchord
            Part.remove_last_();
            continue;
        }
        else if recurse(Part){ //Recurse!
            return True;
        }
        else{ //Try a different nchord
            Part.remove_last_nchord()
        }
    }

    //Ran out of nchords to try
    return False;
}
```

Figure 3.10: Pseudocode for Recursive Backtracking Algorithm

algorithm will find a solution and return True or it will exhaust all possibilities and return False, that it could not find a solution for the given random seed and rules.

This algorithm was chosen to allow flexibility for all of the various input parameters and lends itself well to the creation of modular melodic rules. Figure 3.11 is an example where our possible notes are always A, B, or C and the only rule is that notes cannot be repeated. This is an example of the rules being broken and the algorithm backtracking to try a different note. Figure 3.12 is an example where the possible notes are again always A, B, or C and the only rule is that the exercise must end with a B then a C. This serves to exemplify how the algorithm can enter a node where there are no possible legal notes, try all of them, and then backtrack when the list of possible notes is empty.

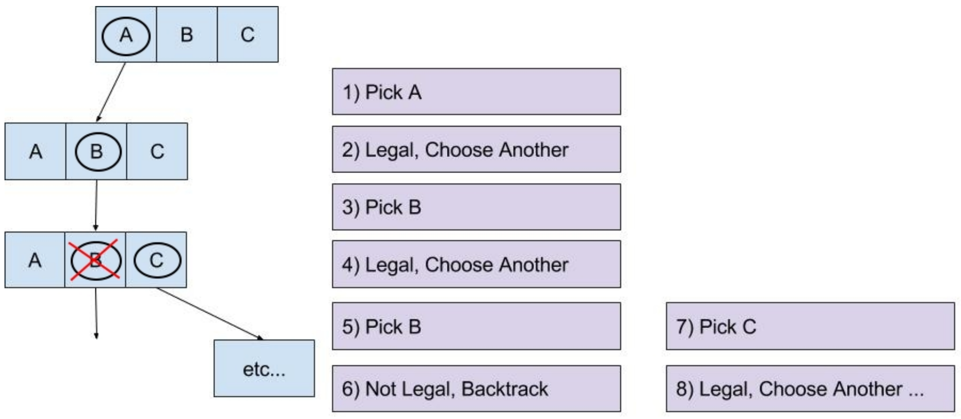


Figure 3.11: Tree Walk Example, Possible notes are A, B, C and rules are no repeated notes

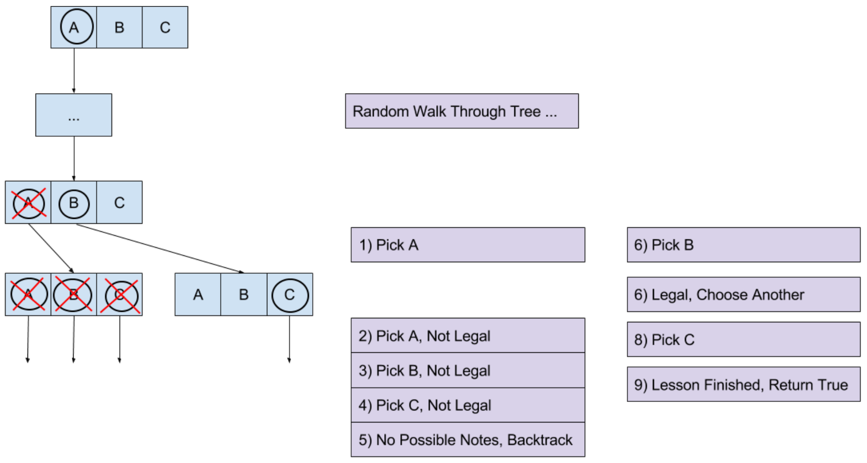


Figure 3.12: Tree Walk Example, Possible notes are A, B, C and Rule is must end with B then C

The Primary Algorithm's Wrapper

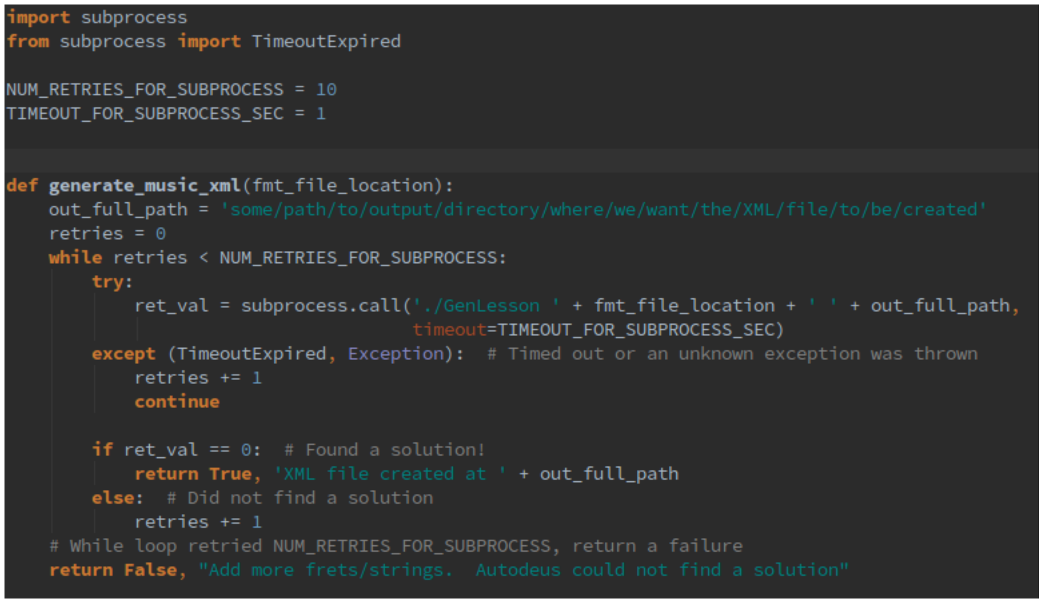
The biggest downfall of this algorithm is its random nature. For an arbitrary set of rules, random seed, and input parameters, various outcomes are possible:

- The algorithm finds a solution quickly on the order of milliseconds.
- The algorithm finds a solution slowly on the order of seconds.
- The algorithm gets unlucky because a combination of rules and a random seed don't find a solution even though one exists. This is unlikely but possible.
- The algorithm cannot find a solution because the search space is over-constrained and it's impossible.

For the reasons of an exponential explosion of possible inputs and for a good user experience, the front-end wraps the primary algorithm with retries and a timeout. Doing so gives a user 2 outcomes: a quickly revealed exercise or a message that tells them to loosen the constraints. The pseudocode is shown in Figure 3.13. The overall algorithm is then a "Las Vegas Type" algorithm [19] meaning that it always gives a correct solution or indicates that it cannot find one.

Generators

The directory located at 'GenLesson/trunk/src/Generators' contains all the source code for the Generator base class, GeneratorFactory, and all the various classes that inherit from the Generator base class. Figure 3.14 shows how each of these classes relate to each other inheritance-wise. The rest of this section enumerates each generator class, explaining the variations that they accomplish.



```
import subprocess
from subprocess import TimeoutExpired

NUM_RETRIES_FOR_SUBPROCESS = 10
TIMEOUT_FOR_SUBPROCESS_SEC = 1


def generate_music_xml(fmt_file_location):
    out_full_path = 'some/path/to/output/directory/where/we/want/the/XML/file/to/be/created'
    retries = 0
    while retries < NUM_RETRIES_FOR_SUBPROCESS:
        try:
            ret_val = subprocess.call('./GenLesson ' + fmt_file_location + ' ' + out_full_path,
                                      timeout=TIMEOUT_FOR_SUBPROCESS_SEC)
        except (TimeoutExpired, Exception):  # Timed out or an unknown exception was thrown
            retries += 1
            continue

        if ret_val == 0:  # Found a solution!
            return True, 'XML file created at ' + out_full_path
        else:  # Did not find a solution
            retries += 1
    # While loop retried NUM_RETRIES_FOR_SUBPROCESS, return a failure
    return False, "Add more frets/strings.  Autodeus could not find a solution"
```

Figure 3.13: Pseudocode for Primary Algorithm's Wrapper

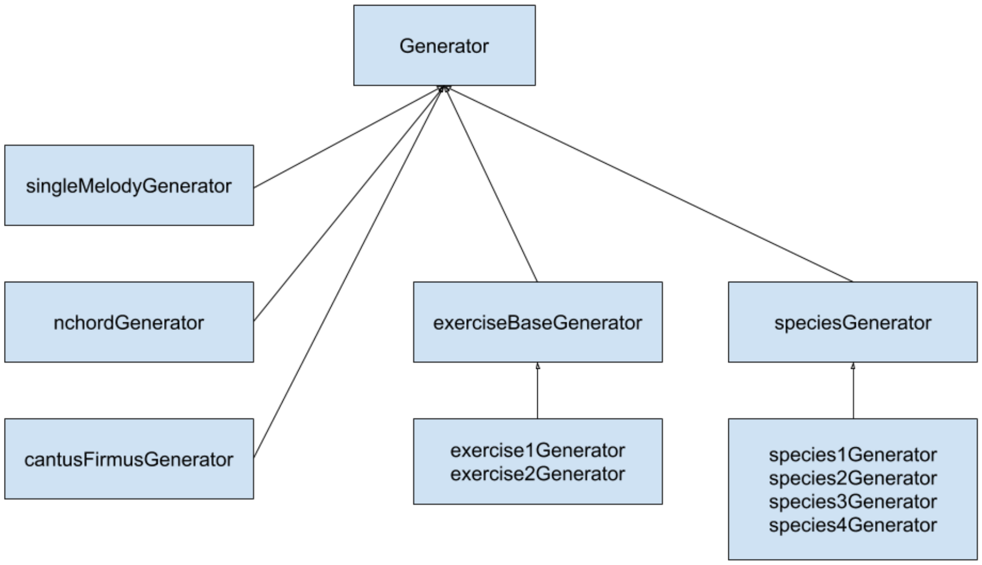


Figure 3.14: Generators Class Inheritance

Single Note Melody

This is one of the simplest forms of generation. It uses notes from the "keyPitchSet" Pitchset in the Generator class to generate an exercise with one voice (no rests) and corresponds to Generation Type "Melody" on the web app. This type of generation utilizes some special input parameters in the "input.fmt" file. These input keys are described as follows:

- note_durations

  Relative to the "divisions" key's integer value being set as a quarter note, these are space separated integers that define which rhythmic durations are allowed in the generated musicXML.

- note_duration_p

  A space separated list of integers defining the probabilistic weights assigned to each of the integers in "note_durations". This must be the same length as "note_durations" so that each weight corresponds to each duration.

- mel_intervals

  A space separated list of the musical intervals that are allowed in the generated musicXML in terms of half steps.

- mel_interval_probs

  A space separated list of integers defining the probabilistic weights assigned to each of the integers in "mel_intervals". This must be the same length as "mel_intervals" so that each weight corresponds to each interval.

- mel_rules

  A space separated list of integers (including only 1, 2, 3, 4, or 5) that indicate which rules to abide by when searching for a solution. For a full description of these 5 rules, see the next chapter.

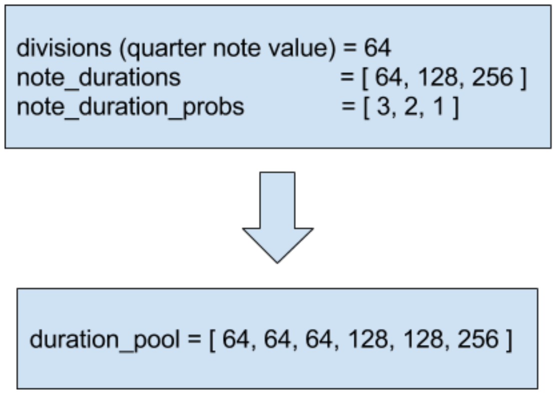
Before starting generation, Single Note Melody generation pre-computes 2 data structures within its class:

- vector<int>duration_pool

  This is created using 'note_durations' and 'note_durations_p'. It is a multiset of all the durations in 'note_durations' where each duration has multiplicity of its corresponding weight in 'mel_interval_probs'. Getting a rhythmic duration at random (with the proper weights) is then quick and easy by picking a random duration from this "duration_pool". See visual example of this in Figure 3.15

- map<Note*, Pitchset*>noteMap

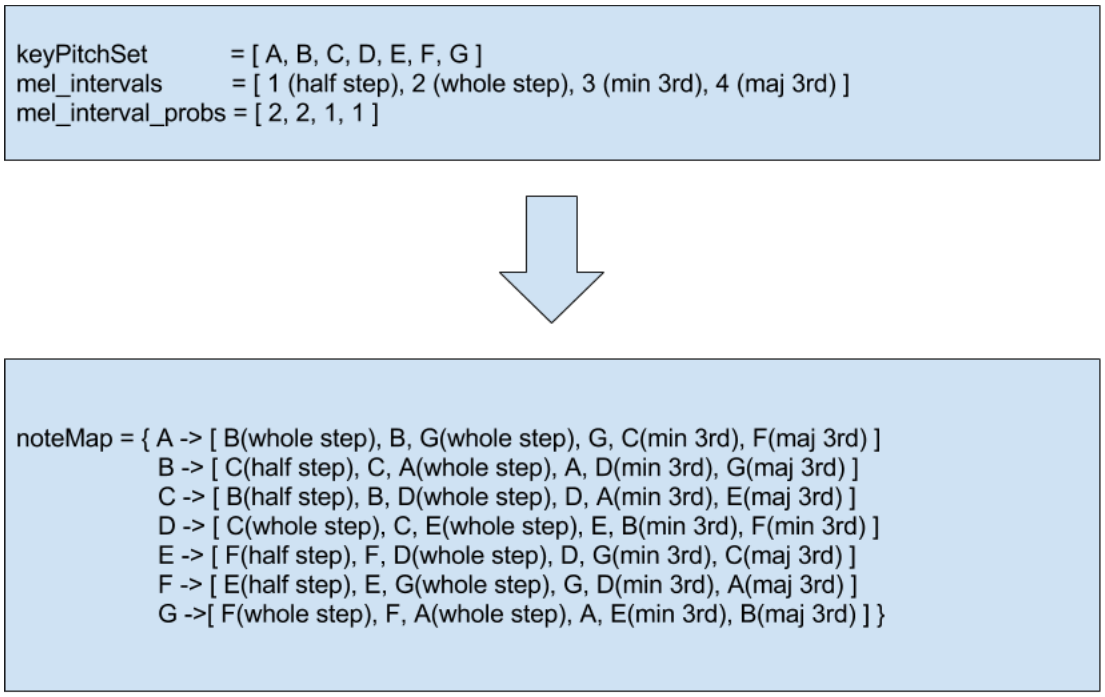
  This is a little more involved than the "duration_pool", but serves as a great optimization for when generation is occurring and a node needs to find a new note to add to the generated exercise that abides by the intervallic probabilities prescribed by 'mel_intervals' and 'mel_interval_probs'. It serves as a hash from a Note to a Pitchset of possible notes to go to next. Somewhat like the "duration_pool" generation, this Pitchset is a multiset of notes where the multiplicity of each note corresponds to the weight of the interval set in 'mel_interval_probs'. To generate this, each note is considered in "keyPitchSet" individually as every other note is iterated through and if the distance between those notes have a matching interval, then that note is added to the Pitchset (which is mapped to from the note under consideration) a number of times equal to the weight in 'mel_interval_probs'. See a visual example of this process in Figure 3.16.



```
divisions (quarter note value) = 64
note_durations                 = [ 64, 128, 256 ]
note_duration_probs            = [ 3, 2, 1 ]
```

```
duration_pool = [ 64, 64, 64, 128, 128, 256 ]
```

Figure 3.15: Example of Duration Pool Pre-Computation



```
keyPitchSet        = [ A, B, C, D, E, F, G ]
mel_intervals      = [ 1 (half step), 2 (whole step), 3 (min 3rd), 4 (maj 3rd) ]
mel_interval_probs = [ 2, 2, 1, 1 ]
```

```
noteMap = { A -> [ B(whole step), B, G(whole step), G, C(min 3rd), F(maj 3rd) ]
            B -> [ C(half step), C, A(whole step), A, D(min 3rd), G(maj 3rd) ]
            C -> [ B(half step), B, D(whole step), D, A(min 3rd), E(maj 3rd) ]
            D -> [ C(whole step), C, E(whole step), E, B(min 3rd), F(min 3rd) ]
            E -> [ F(half step), F, D(whole step), D, G(min 3rd), C(maj 3rd) ]
            F -> [ E(half step), E, G(whole step), G, D(min 3rd), A(maj 3rd) ]
            G ->[ F(whole step), F, A(whole step), A, E(min 3rd), B(maj 3rd) ] }
```

Figure 3.16: Example of Note Map Pre-Computation

The aforementioned special input parameters for the Single Note Melody generator are all obfuscated from the user by the frontend and are hard-coded depending upon the difficulty level and time signatures that are selected. The idea is that easier exercises have larger durations and smaller intervals and the harder exercises visa versa. There are two types of mappings from difficulty to how the intervals and rhythmic that are defined as follows:

- DEF_DIFFICULTY_SETTINGS

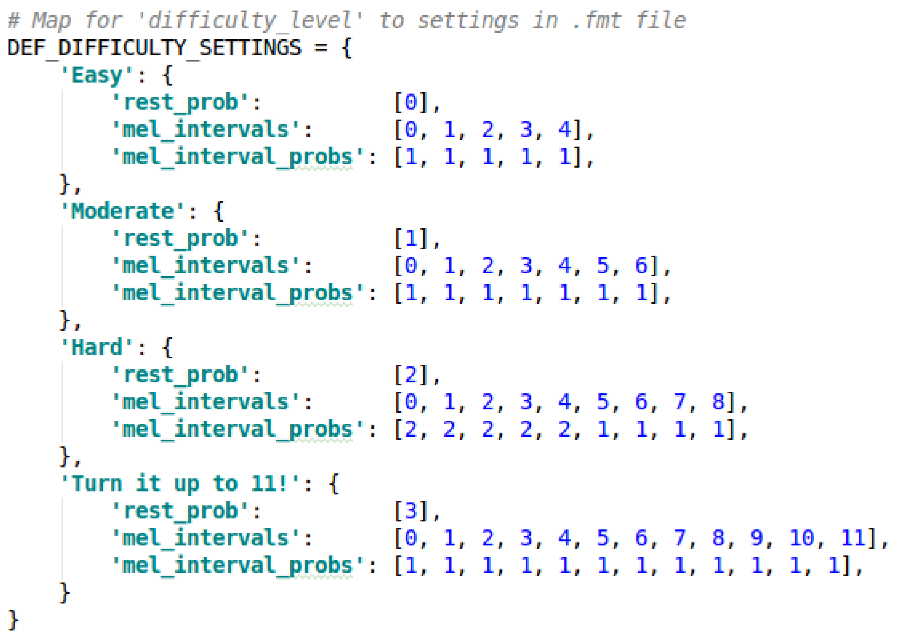
  This is a python dictionary which maps difficulty setting to 'mel_intervals' and 'mel_interval_probs' (also to 'rest_prob' which is NOT used by this generator but becomes relevant in other types of generation). The hard coded values used by the frontend are shown in Figure 3.17

- DEF_TIME_SIG_DIFFICULTY_DURATION_SETTINGS

  This is another python dictionary which maps a time signature setting to difficulty level to the already known 'note_durations_p' and a new float list of 'note_durations_mult'. These float values indicate how to create the 'note_durations' used in the exercise by defining coefficients to be multiplied times a quarter note ('divisions' setting). For example, a setting of '3/8' time and 'Moderate' difficulty uses multipliers of [1.5, 1, 0.5] which would indicate to use, respectively, dotted quarter notes, quarter notes, and eighth notes with equal probability weight when generating the exercise. These hard coded values can be seen in Figure 3.18

NChord Generation

The name for this type of generation may be somewhat misleading as all types of generation use the NChord data structure, but this type of generation refers to the creation of musical exercises that (within 1 Voice) may include from 1 to 6



```
# Map for 'difficulty_level' to settings in .fmt file
DEF_DIFFICULTY_SETTINGS = {
    'Easy': {
        'rest_prob':          [0],
        'mel_intervals':      [0, 1, 2, 3, 4],
        'mel_interval_probs': [1, 1, 1, 1, 1],
    },
    'Moderate': {
        'rest_prob':          [1],
        'mel_intervals':      [0, 1, 2, 3, 4, 5, 6],
        'mel_interval_probs': [1, 1, 1, 1, 1, 1, 1],
    },
    'Hard': {
        'rest_prob':          [2],
        'mel_intervals':      [0, 1, 2, 3, 4, 5, 6, 7, 8],
        'mel_interval_probs': [2, 2, 2, 2, 2, 1, 1, 1, 1],
    },
    'Turn it up to 11!': {
        'rest_prob':          [3],
        'mel_intervals':      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11],
        'mel_interval_probs': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    }
}
```

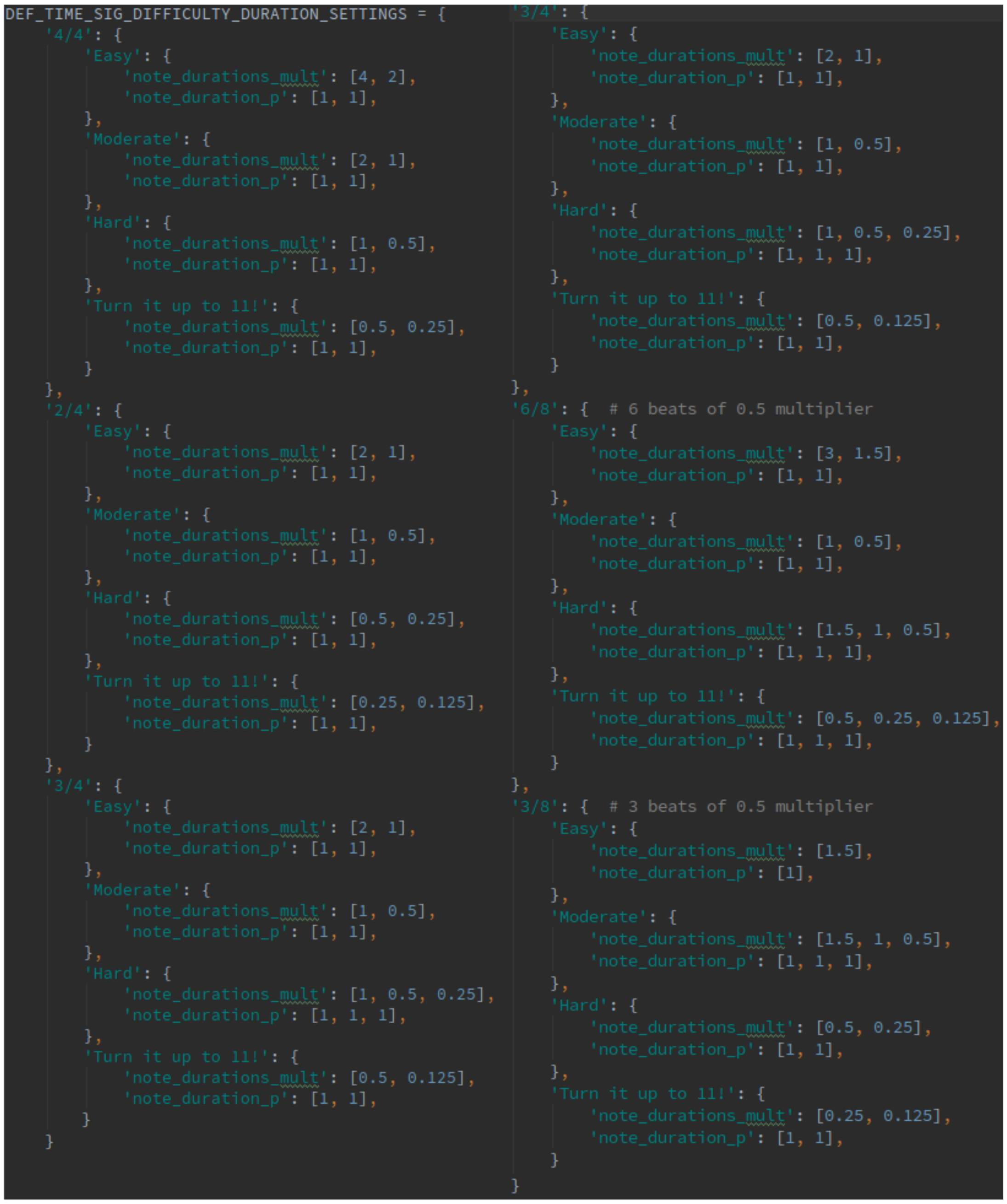
Figure 3.17: Difficulty to Melodic Interval Mapping

notes or a rest in a single NChord data structure. It also differs from all other of the generation types because it is the only type that does not use the "Primary Generation Algorithm" described in 3.4.

On the frontend, this type of generation can be utilized by selecting "Melody with up to 6 Note Chords" in the generation type drop-down menu. In the backend, this type of generation has the following special types of parameter keys that can be defined/modified in the 'input.fmt' file:

1. 'nchord_size_p'

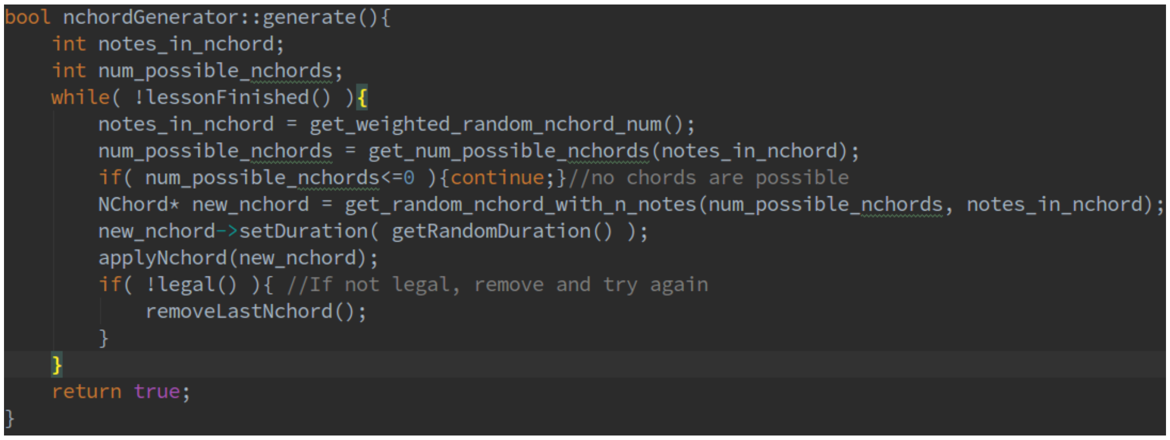
   This a list of 7 space separated integers that define the weights that are considered when generating an exercise for the size of NChord (how many notes) to be used. If labeling these 7 values with 0 indexing starting from the leftmost, each index refers to the number of notes to be used (0 notes in an NChord is a rest).



```
DEF_TIME_SIG_DIFFICULTY_DURATION_SETTINGS = {
    '4/4': {
        'Easy': {
            'note_durations_mult': [4, 2],
            'note_duration_p': [1, 1],
        },
        'Moderate': {
            'note_durations_mult': [2, 1],
            'note_duration_p': [1, 1],
        },
        'Hard': {
            'note_durations_mult': [1, 0.5],
            'note_duration_p': [1, 1],
        },
        'Turn it up to 11!': {
            'note_durations_mult': [0.5, 0.25],
            'note_duration_p': [1, 1],
        }
    },
    '2/4': {
        'Easy': {
            'note_durations_mult': [2, 1],
            'note_duration_p': [1, 1],
        },
        'Moderate': {
            'note_durations_mult': [1, 0.5],
            'note_duration_p': [1, 1],
        },
        'Hard': {
            'note_durations_mult': [0.5, 0.25],
            'note_duration_p': [1, 1],
        },
        'Turn it up to 11!': {
            'note_durations_mult': [0.25, 0.125],
            'note_duration_p': [1, 1],
        }
    },
    '3/4': {
        'Easy': {
            'note_durations_mult': [2, 1],
            'note_duration_p': [1, 1],
        },
        'Moderate': {
            'note_durations_mult': [1, 0.5],
            'note_duration_p': [1, 1],
        },
        'Hard': {
            'note_durations_mult': [1, 0.5, 0.25],
            'note_duration_p': [1, 1, 1],
        },
        'Turn it up to 11!': {
            'note_durations_mult': [0.5, 0.125],
            'note_duration_p': [1, 1],
        }
    }
```

```
'3/4': {
    'Easy': {
        'note_durations_mult': [2, 1],
        'note_duration_p': [1, 1],
    },
    'Moderate': {
        'note_durations_mult': [1, 0.5],
        'note_duration_p': [1, 1],
    },
    'Hard': {
        'note_durations_mult': [1, 0.5, 0.25],
        'note_duration_p': [1, 1, 1],
    },
    'Turn it up to 11!': {
        'note_durations_mult': [0.5, 0.125],
        'note_duration_p': [1, 1],
    }
},
'6/8': {  # 6 beats of 0.5 multiplier
    'Easy': {
        'note_durations_mult': [3, 1.5],
        'note_duration_p': [1, 1],
    },
    'Moderate': {
        'note_durations_mult': [1, 0.5],
        'note_duration_p': [1, 1],
    },
    'Hard': {
        'note_durations_mult': [1.5, 1, 0.5],
        'note_duration_p': [1, 1, 1],
    },
    'Turn it up to 11!': {
        'note_durations_mult': [0.5, 0.25, 0.125],
        'note_duration_p': [1, 1, 1],
    }
},
'3/8': {  # 3 beats of 0.5 multiplier
    'Easy': {
        'note_durations_mult': [1.5],
        'note_duration_p': [1],
    },
    'Moderate': {
        'note_durations_mult': [1.5, 1, 0.5],
        'note_duration_p': [1, 1, 1],
    },
    'Hard': {
        'note_durations_mult': [0.5, 0.25],
        'note_duration_p': [1, 1],
    },
    'Turn it up to 11!': {
        'note_durations_mult': [0.25, 0.125],
        'note_duration_p': [1, 1],
    }
}
```

Figure 3.18: Time Signature to Difficulty to Durations Mapping



```
bool nchordGenerator::generate(){
    int notes_in_nchord;
    int num_possible_nchords;
    while( !lessonFinished() ){
        notes_in_nchord = get_weighted_random_nchord_num();
        num_possible_nchords = get_num_possible_nchords(notes_in_nchord);
        if( num_possible_nchords<=0 ){continue;}//no chords are possible
        NChord* new_nchord = get_random_nchord_with_n_notes(num_possible_nchords, notes_in_nchord);
        new_nchord->setDuration( getRandomDuration() );
        applyNchord(new_nchord);
        if( !legal() ){ //If not legal, remove and try again
            removeLastNchord();
        }
    }
    return true;
}
```

Figure 3.19: Pseudocode for the NChord Generation Type

2. ‘note_durations’ and ‘note_duration_p’

   These defines are the same as in Single Note Melody generation

3. ‘nchord_rules’

   This is a space separated list of integers which defines which rules to include. Currently there is only 1 rule written for this type of generation and the infrastructure is present to stay consistent with design patterns and for extensibility.

This type of generation utilizes a while loop instead of the recursive style of generation and its pseudocode can be seen in Figure 3.19. This pseudocode has several function calls explained as follows:

- ‘get_weighted_random_nchord_num()’

  This method uses the weights from ‘nchord_size_p’ to return an integer from 0 to 6 with the correct probability. It does this using the “alias method” [20] which essentially is an algorithm, in a sense, that rolls a weighted dice.

- ‘get_num_possible_nchords(notes_in_nchord)’

  This method essentially calculates the number of possible permutations of notes in ‘KeyPitchset’ for a given integer. In other words it calculates the length of ‘KeyPitchset’ choose ‘notes_in_nchord’ (variable in pseudocode)

- ‘get_random_nchord_with_n_notes(num_possible_nchords, notes_in_nchord)’

  This function utilizes a rather interesting little algorithm called “Buckles Algorithm 515” [21] This algorithm allows Autodeus to randomly pick an index from 0 to (‘num_possible_nchords’ - 1), call it X, and use this to get a list of indices in ‘KeyPitchset’ that are the Xth permutation.

- ‘legal()’

  As stated earlier, this type of generation currently has only 1 rule. This rule is designed to make sure that the Nchord generated by ‘get_random_nchord_with_n_notes(...)’ is actually physically able to be played on a guitar.

Cantus Firmus

A Cantus Firmus can be loosely described as a pre-existing base (and bass) melody meant to be sung by a single voice (literally a vocal part and not our Voice data structure). It serves as a basis for many other more complex voices/melodies to be built around it and this process is comprehensively described in Joseph Fux’s Gradus Ad Parnassum [22]. While very subjective, within the context of this thesis, a Cantus Firmus is defined as a melody with all whole notes that starts and ends on a root note and is typically 12 or 16 measures long.

As the previously described ‘Single Note Melody’ generation does, the ‘Cantus Firmus’ generation uses 2 lists of space separated integers as its inputs to define allowable musical intervals and their corresponding weighted probabilities. These are named ‘cantus_firmus_intervals’ and ‘cantus_firmus_interval_probs’ and these are used to pre-compute a member variable of the ‘cantusFirmusGenerator’ class called ‘noteMap’ in the exact same way as the “Single Note Melody” generator. Unlike the ‘Single Note Melody’, there is no rhythmic duration settings as the Cantus Firmus generation uses all whole notes by definition. For a complete listing of all the rules that the Cantus Firmus generation uses, see the next chapter’s subsection 3.8.3

A Cantus Firmus generation also has the ability to generate exercises within a given mode. A mode parameter assigns a root note to the exercise that it should begin and end on. Musically this assures that the exercise “sounds” like it is in that mode.

Lastly, this type of generation has a special rule that creates a “climax” note. As music is subjective, one of the guidelines to creating a Cantus Firmus is that it should have a highest or lowest note “somewhere” in the middle”. This is accomplished with the ‘cantus_firmus_mid_radius’ setting in the ‘input.fmt’ file. This defines a number of measures from the middle-most measure of an exercise where this climax note should be located and a rule is used to ensure that a climax note is indeed present within this constraint.

Exercise Generators

The “Exercise” generators are an off-shoot variety of Cantus Firmus generation. As can be seen in Figure 3.14, they both have a base class that they inherit from. In a nutshell, these types of generation first produce a Cantus Firmus melody (The ‘exerciseBaseClass’ has a Cantus Firmus generator within it) and then modify the produced Cantus Firmus notes by adding rests and changing the rhythmic durations. This is done with a new input in the format file named ‘rest_prob’ which is an integer from 0 to 9 that defines the probability of rests out of 10. This is the other hard-coded values that are set in Figure 3.17. Before generation, an Exercise generator estimates how many Cantus Firmus whole notes will be needed based on the the inputs of ‘note_durations’, ‘note_durations_p’, and ‘rest_prob’.

Exercise 1

This generation method attempts to create a lesson that has a number of measures halfway between ‘min_num_of_bars’ and ‘max_num_of_bars’ by calculating the weighted average duration of each NChord from the ‘duration_pool’ and figures out how many notes are needed by calculating the following:

$$\frac{max\_num\_of\_bars + min\_num\_of\_bars}{2} * \frac{measure\_length}{average\_duration} \tag{3.1}$$

A Cantus Firmus with the same number of measures as notes needed is generated and then the ‘Exercise 1’ generator randomly permutes the duration of each NChord (which are all whole notes initially) and then continues on to use the ‘rest prob’ to make a correct ratio of those generated notes into rests (by removing all the notes in the NChord). This generation type is not currently hooked up to work with the frontend, but is still available to the backend ‘GenLesson’ executable via the ‘input.fmt’ file settings.

Exercise 2

This varies slightly from Exercise 1 generation because it figures out the number of notes needed AND rests before generation. It does this by using the same weighted average to estimate the number of notes needed but then subtracts a number so that when rests are added later, it will have the approximate proportion of rests to notes according to ‘rest_prob’. After generation of a Cantus Firmus is completed, it then randomly chooses indices and inserts rests throughout the exercise to satisfy ‘rest_prob’. This generation type can be used on the frontend app via the selection of “Melody with Rests” in the Generation type drop-down box.

Species Generators

Species generators are another whole type of subclass that can be seen in Figure 3.14. 2 voice species counterpoint are a step-by-step method of teaching music students to learn voice-leading rules in composition. A species exercise is comprised of an initially generated Cantus Firmus and then a second voice is generated with rules in relation to that initially generated voice to begin creating harmony. This creates 2 separate melodies that sound good by themselves and while played simultaneously.

Like the Cantus Firmus generation, mode is an input parameter and some modes have special characteristics that allow for notes outside the usual 'keyPitchSet' to be used. Thus, a 'speciesGenerator' class has an extra Pitchset called a 'keyPitchSetM6M7' which, for the modes of mixolydian, aeolian, and dorian, includes 2 extra intervals for the major 6th and major 7th. When generating any second voice in species generation, this is the Pitchset that is used and then rules are used to constrain when or if these notes should be included in the exercise being generated.

Every species two voice generation has a common set of input keys in 'input.fmt' file which are as follows:

1. general_counterpoint_melody_rules

   A space separated list of integers (only including 1, 2, 3, and 4) which defines which rules to use. These rules are used as constraints for all types of species 2-voice generation types.

2. repeated_note_odds

   This is a single integer which defines the probability out of 100 that a repeated note will be allowed. Subjective music rules dictate that a counterpoint melody rarely repeats a note so a "weighted dice" is rolled (in a manner of speaking) every time a repeated note is encountered to decide if the rule will allow it to happen or not.

3. counterpoint_intervals

   A space separated list of integers like that used with other generation types that defines the allowable intervals to be used when generating the second voice.

4. counterpoint_interval_probs

   A space separated list of integers like that used with other generation types that defines the corresponding probabilistic weights of the above defined intervals.

5. counterpoint_mid_radius

   This is an integer value (like the Cantus Firmus generation's 'cantus_firmus_mid_radius') that defines the largest distance from the middle-most measure that a climax note in the counterpoint voice can be located. Once again, a rule is then used to force this constraint when generation is occurring.

Species 1

This generation type generates a second voice with the Cantus Firmus such that each note has a whole note rhythmic duration. This creates a one-note-per-cantus-firmus-note harmony. An example of this can be seen in Figure 3.20. On the frontend, this generation type can be used by setting 'Species 1' in the Generation type drop-down menu.

Species 2

This generation type generates a second voice with the Cantus Firmus such that each note in the Cantus Firmus has 2 notes that coincide with it (2 half notes per

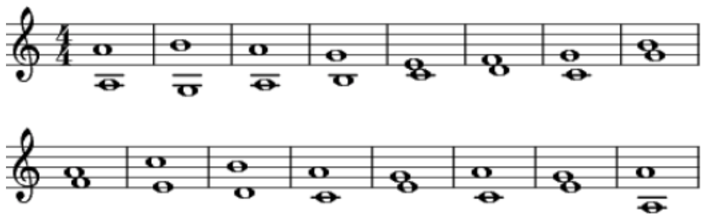


Figure 3.20: Species 1 Generated Example

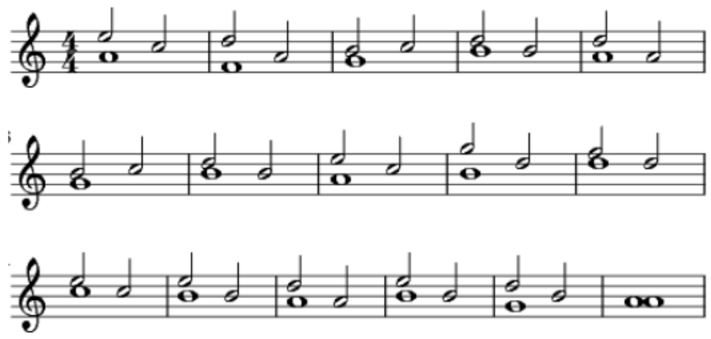


Figure 3.21: Species 2 Generated Example

Cantus Firmus whole note). This creates a 2-note-per-cantus-firmus-note harmony. An example of this can be seen in Figure 3.21. On the frontend, this generation type can be used by setting ‘Species 2’ in the Generation type drop-down menu.

Species 3

This generation type generates a second voice with the Cantus Firmus such that each note in the Cantus Firmus has 4 quarter notes that coincide with it. This creates a 4-note-per-cantus-firmus-note harmony. An example of this can be seen in Figure 3.22. On the frontend, this generation type can be used by setting ‘Species 3’ in the Generation type drop-down menu.

Species 4

This last species generation is somewhat different from the others rhythmically. The counterpoint voice starts with a half note, and then proceed to have “staggered”

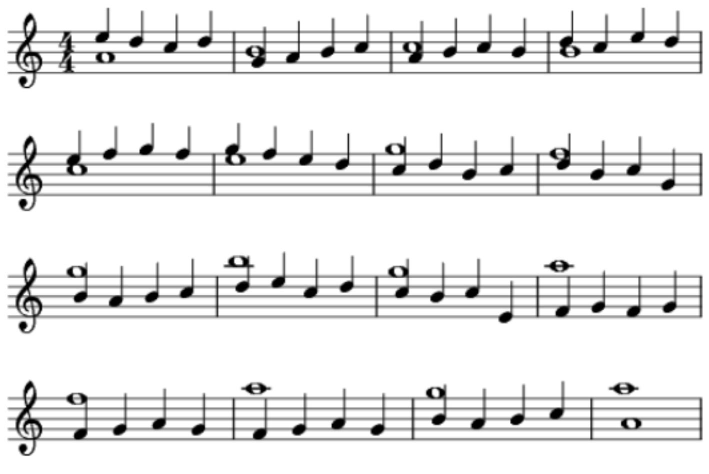


Figure 3.22: Species 3 Generated Example

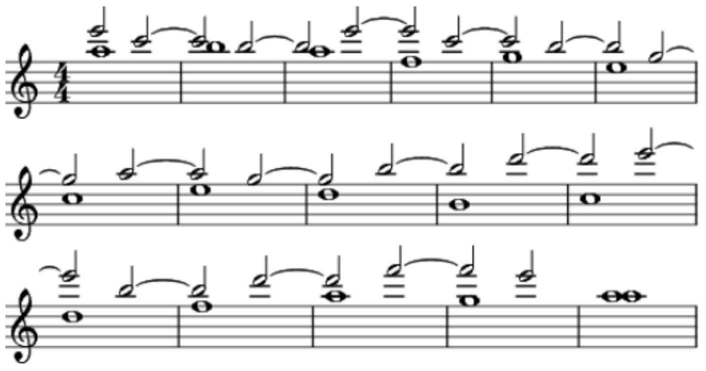


Figure 3.23: Species 4 Generated Example

whole notes that are tied over each bar in the exercise until the end of the exercise where the voice then ends with a penultimate half note then whole note. An example of this can be seen in Figure 3.23. On the frontend, this generation type can be used by setting 'Species 4' in the Generation type drop-down menu.

Optimizations

Many various optimizations were made to help the run time of the 'Primary Generation Algorithm'. In being a back tracking algorithm, the cost in run time is the number of nodes visited times the running time cost of each of those nodes. Reduction in both of these factors was aided by the following:

- The use of a hash to get a set of notes to choose from in a node given a current state as described in subsection 3.6.1 greatly reduces the cost for each node. The pre-computation of 'noteMap' avoids the need to consider parameterized musical intervals and their corresponding probabilistic weights in every node.
- Care was taken to avoid allocation/de-allocation of memory in any node while searching for a solution. The following describe ways this was achieved:
  - When 'keyPitchset' and 'keyPitchsetM6M7' are created, they allocate all the space needed for the legal notes before any search/generation begins and these sets of notes are then pointers to allocated memory for each note. This allows any NChord to simply uses pointers to those legal notes instead worrying about memory for them.
  - Before any generation is done, an estimate of the number of needed NChords is made and a pool of allocated empty NChord instances is then available. When entering a node during generation, an Nchord is fetched from this pool which is a quick operation relative to allocating a new Nchord instance in each new node. Conversely, if a node fails to find a solution, the NChord is returned to the pool for re-use by others. The only time during generation that an allocation for an NChord occurs is if the estimate was less than what was needed and the pool becomes empty.
- Pruning was introduced as a result of the set of notes to choose from in a given node being a multiset. If a randomly chosen note gets a value of false returned from its child, it is then obvious that choosing that note again will

also fail. Removal of any multiplicities from a failing note in the note set under consideration in a given node avoids choosing that same note again.

- When development first began, there were gigantic sets of "if" statements where each "if" checked if a rule was enabled or not and, if it was enabled, the rule would be checked to see if the current incomplete exercise was abiding the rule. This is inefficient because every single rule was checked every time (if it was enabled or note) and also created a mess of code. A factory pattern was implemented so that only the rules that are enabled get instantiated and added to the vector called 'generator_rules' in the 'Generator' base class. Checking for legality iterates over only the enabled rules and a ton of code looks nice and clean.

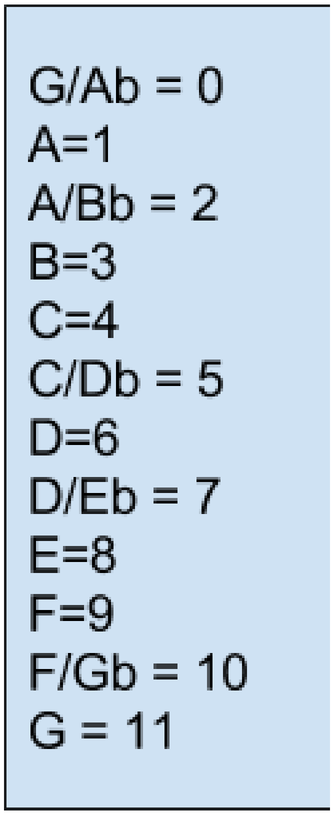
Other Algorithms

Outlining Major/Minor Chords

There are two note-worthy static functions that are included in the Note class. These are called 'outlineMajorChord' and 'outlineMinorChord'. These both have a return value of boolean and both accept three pointers to Note objects as their arguments. They both utilize a somewhat clever trick to recognize if the three notes are part of a major or minor chord.

If one imagines a single guitar string as a bitstring such that each bit represents a fretting position on that guitar string, then a value of 0 represents a fret thats not in a chord and a value of 1 represents a fret that is in a chord. Each type of chord then produces a signature pattern because of music's cyclical scale structure.

For example, a major chord's interval degrees are the root note, a major 3rd, and a 5th. The difference in half notes between each of these is: Root to major 3rd is 4 half steps, major 3rd to 5th is 3 half steps(a minor 3rd), a 5th to the next root



G/Ab = 0
A=1
A/Bb = 2
B=3
C=4
C/Db = 5
D=6
D/Eb = 7
E=8
F=9
F/Gb = 10
G = 11

Figure 3.24: Modular 12 Note to Integer Conversion

note is 5 half steps(a perfect fourth), and then the pattern repeats ad infinitum up and down throughout the octaves. Plotting this on a guitar string with two octaves worth of frets and assuming that the root starts on the 0th (most significant) bit, we get this bitstring:

1000 1001 0000 1000 1001 0000

The same can be done with a minor chord to produce:

1001 0001 0000 1001 0001 0000

When these two functions are called, they convert each note into an integer value and then they consider the modular 12 value. For every possible note, Figure 3.24 shows the result of this conversion.

Using these numbers as another bit string 12 bits long with the integer values set to 1, they can be right shifted 12 times and, at each iteration, compared to the chord pattern. If at any point during this process, if all 3 of the integers refer to indices in the chord pattern that are set to 1, then it is known that those notes outline the specific chord pattern. Keep in mind that if all the notes match at different octaves, are solely octaves and fifths, or are octaves and 3rds, this will still evaluate to true.

Climax Note Production

An example of an unforeseen conflict was that of the climax note rules sometimes choosing a root note as a climax note which then disallowed the counterpoint generation to end on a root note. When this happened, it took the program far too long to backup all the way to the middle from the end. This was fixed by making sure that a root note cannot be chosen as a climax note.

The "climax note production" rules do this, but take other actions also. One of the rules that is somewhat subjective in classical counterpoint production is the guideline that each single note melody should have a climax note somewhere in the middle. In order to objectify this rule, it was necessary to define some parameters and create some specialized rules that differed slightly in functionality from the others. "cantusFirmus_rule8" and "generalCounterpointMeldoy_rule5" return false if they are broken, but differ from the other rules quite substantially with respect to the fact that they are responsible for setting various booleans and variables that allow the program to keep track of if a climax note has been found or not and also for taking different actions in all of the various possible situations. Listed are the relevant variables and brief descriptions:

- 'counterpoint mid radius' and 'cantus firmus mid radius'

  These two integer variables define the number of measures that are in the middle. To calculate which measures are considered in the middle, the number of measures defined internally by the Lesson class as 'numMeasures' is divided by two (integer division) and then the upper and lower bounds of measures are the radii added to and subtracted to that number, respectively. For example, if the number of measures is 17 and the radius is 3, then the middle measures are any measure from (17/2) - 3 = 5th to the (17/2) + 3 = 11th measure (there is a 0th measure).

- 'counterpointFoundHighNoteInMiddle', 'counterpointFoundLowNoteInMiddle', 'cantusFirmusFoundHighNoteInMiddle', 'cantusFirmusFoundLowNoteInMiddle'

  These boolean variables are simply used to keep track of different situations that can occur as a result of the specialized rules. These could be thought of as state variables.

- 'counterpointClimaxNote' and 'cantusFirmusClimaxNote'

  These two variables are pointers to Note objects so that once a climax note is found, it can be quickly referenced.

- 'counterpointClimaxNChordIndex' and 'cantusFirmusClimaxNChordIndex'

  These integer variables are also used to quickly reference the climax notes if necessary.

One very important consequence of these 2 rules is that precautions must be taken in certain situations. The first special case occurs when a note in 'possibleNotes' is found to be a climax note in the middle, setting the internal variables, but then a later rule being checked is found to make that note illegal. All of the relevant variables must be reset to indicate that a climax note was not found so that when subsequent new notes are tried in 'possibleNotes', the program is not in the wrong state. A second special case occurs when a node is returning false and previously had a climax note. Once again, all the relevant variables must be reset properly to avoid a situation where the program could enter an incorrect state. All generation methods that use these rules were augmented within their main recursive function to avoid nasty problems as a result of these special cases.

Notation Fix Functions

There are several functions that are present whose sole purposes are to “fix up” the musicXML so that when it is converted to an actual visual representation, the notation looks nice and clean. As a necessity, the class variables ‘tieEnd’, ‘tieStart’, ‘numOfDots’, and ‘stemUp’ were added to the Note class. These additions are not terribly difficult from an algorithmic perspective, but are highly important in making the notation produced from the XML code look much better and also make the program much more flexible. Previous versions of the program did not have any tied notes, thus, notes had to be comprised of only doubles or halves of the number of divisions(the quarter note value) and could not traverse bar lines. The following are the “fix up” functions and brief descriptions of each:

- ‘mergeAdjacentRests()’

  This function simply walks through each voice in order to find rests that are next each other, removes them, and then adds a rest that has the same duration as the sum of the previously removed rests. For example, in music notation, it is not proper to have 2 quarter rests and a half rest in a measure where a single whole rest should be present.

- ‘dottedNotesFix()’

  This function finds any note that has an irregular duration length and then breaks it down in order to figure out how to add dots to the notes and/or add extra tied notes to properly notate the duration. For example, if our quarter note has 64 divisions, and an NChord somehow is generated that has a duration of 104=64+32+8, it should be notated as a dotted quarter note tied together with a 32nd note. Before this function was implemented, it wouldve been notated as only a quarter note which made the notation present in the produced pdf files from the music XML have incomplete looking measures.

- 'tiedNotesFix()'

  This function simply finds any places where a duration crosses a barline, removes that offending NChord, and then produces two tie NChords whose sum is the duration of the removed NChord.

- 'setStemsProperly()'

  This function goes through and sets a boolean in each note of each NChord so that the stems will be notated properly in two voice counterpoint. In other words, it makes sure that the cantus firmus stems are oriented downward and the counterpoint melodys stems are oriented upward. As a result of these additions, the function convertToXml(Part*) had some additional features programmed into it so that the extra notation would be output in the XML.

Rule Sets

Single Note Melody

These are the rules that Ricardo Iznaola suggested be implemented for experimentation in producing viable musical exercises. The source code for these can be found in 'GenLesson/trunk/src/Rules/' with the prefix 'singleMelody_rule*' and a summary of each is as follows:

Rule 1. Largest leap is a 10th (octave + major 3rd) in either direction

Rule 2. A note may be repeated a maximum of (repetitions do not include the first note)

(a) One time if note value is whole or half-note

(b) 2 times if note value is quarter-note

(c) 3 times if value is eighth-note

(d) 5 times if value is smaller than eighth-note

Rule 3. A two-note motion is allowed a maximum of(The first intervallic motion is not included in the repetition count):

(a) No repetition if both values are whole-note

(b) One repetition if both values are half-note

(c) 2 repetitions if both values are quarter-note

(d) 3 repetitions if both values are eighth note or smaller

(e) When values are mixed, the allowable repetitions are those of the smaller value

(f) Two-note motions may not return to either note more than 3 times in a row

Rule 4. All intervallic sequences are allowed, with the following limitations(The first intervallic sequence is not included in repetition count):

(a) 2 note intervallic sequences involving 3rds or bigger intervals can be repeated a maximum of two times unless the exercise is explicitly designed to train the interval

(b) Intervallic sequences involving 3 or more notes can be repeated only once, unless the exercise is explicitly designed to train the pattern

(c) Maximum intervallic distance allowed between first and last note of the pattern is a tenth.

Rule 5. Voice Leading rules

(a) Second note of whole step may jump in either direction within range of and up to a tenth.

(b) Second note of half step may jump in either direction within the range of and up to an octave

(c) A 3rd or a perfect fourth movement cannot continue with a jump bigger than a 5th from the second note. If the resulting interval between the first and the third note is a 7th or an octave, the next note has to move by step motion in the opposite direction.

(d) The tritone must resolve by half-step motion

(e) Second note of the 5th may jump up or down within the range of and up to a 4th, except for cases described in c.

(f) Second note of a sixth may move by step or jump in the opposite direction within the range of and up to a 5th.

(g) The 7th resolves by step in the opposite direction.

(h) The second note of an octave moves either by step or (in opposite direction only) by jump no bigger than a fourth

(i) The ninth resolves by step in the opposite direction

(j) The second note of a 10th moves either by step or (in opposite direction only) by jump no bigger than a 4th. If the resulting interval between the first and third note is a 7th or a 9th, then the third note must move by step in either direction, or according to pre-established rule for the interval produced between second and 3rd notes.

NChord Generation

Rule 1. Make sure the notes are physically playable. This entails verifying that all the notes are on different strings and making sure that the notes are not spread out over too far a span of the frets.

Cantus Firmus

Rule 1. No two consecutive notes exactly the same

Rule 2. Must begin and end on root note

Rule 3. Second to last note must approach from step above/below root

Rule 4. Movement is mostly in steps

Rule 5. If interval is greater than a 5th (7 half steps) must be preceded and followed by movement in opposite directions

Rule 6. If a leap(greater than 2 half steps), must be followed by either:

(a) a. Movement in opposite direction

(b) b. Another leap so that all three notes outline maj/min chord. An exception to this rule is the beginning 3 notes which can begin with a leap of a 3rd followed by same motion of direction

Rule 7. Two successive leaps in the same direction must outline chord tones (root, 3rd, 5th) and then be proceeded by movement in the opposite direction

Rule 8. Must have a single low or high point in the middle somewhere

legalGeneralCounterpointMelody

Rule 1. Melody is diatonic with exceptions:

   (a) Always raise 7 if ascending to root before last note in Mixolydian, Aeolian, and Dorian modes

   (b) 5-M6-M7-R is allowed at ending cadence in Dorian or Aeolian

Rule 2. Notes are rarely repeated

Rule 3. Melodic motion is mostly in steps

Rule 4. Leaps acceptable with exceptions and conditions:

   (a) No augmented or diminished intervals allowed

   (b) Mostly not larger than 5th, but 8ve okay

   (c) Occasionally ascending m6 okay

   (d) If ascending/descending m6 or octave must be preceded and followed by motion in opposite direction

   (e) leap must be:

      i. Followed and preceded by movement in opposite directions

      ii. Double leap must outline min/maj chord and be followed and preceded by movement in opposite directions

      iii. Octave with intervals 5th + 4th

Rule 5. Species voice must have a climax point somewhere near middle

Rule 6. Last 3 to 4 notes of a passage in a single direction should not stress a tritone

Rule 7. Must approach last note step-wise

Rule 8. No leaps to a dissonance

Rule 9. No chromatic movement (notes next to each other with different accidentals)

legalGeneralTwoVoice

Rule 1. Climaxes in voices should be at different times

Rule 2. If other voice is higher than cantus firmus, begins with unison, 5th, or octave

Rule 3. If other voice is lower than cantus firmus, begins with unison or 8ve

Rule 4. Ends on unison or octave

Rule 5. Voices cannot overlap or cross (except for unisons at beggining or end)

legalSpecies1

Must follow legalGeneralTwoVoice and legalGeneralCounterpointMelody rules.

Rule 1. Only consonance of 5th, 8ve, m3, M3, m6, M6 or unison are allowed

Rule 2. Unison can only occur at beginning or end

Rule 3. Perfect intervals (unison, 5th, 8ve) approached only by contrary or oblique motion, not by parallel or similar Exception: horn fifths exception can move in similar motion if descending 3rd to 5th or ascending 6th to 5th(with top voice moving stepwise)

Rule 4. Parallel motions with unison, 5th, 8ve are always incorrect

Rule 6. No more than three parallel 3rds or 6ths in succession

legalSpecies2

Must follow legalGeneralTwoVoice and legalGeneralCounterpointMelody rules.

Rule 1. First half note in each measure must be consonant

Rule 2. If unstressed note is dissonant, it must be a passing tone

Rule 3. Unison allowed only on first note or last note or on unstressed notes

Rule 4. If an unstressed note is consonant, it should be part of a triad formed by previous note and cantus firmus

Rule 5. No parallel 5ths or 8ves between adjacent stressed beats

Rule 6. Voices cannot overlap nor cross except for an unstressed beat unison

Rule 7. No parallel 5ths or 8ves between an unstressed beat to a stressed beat

legalSpecies3

Must follow legalGeneralTwoVoice and legalGeneralCounterpointMelody rules. Used some extra rules found at [23] [24]

Rule 1. First beat must be consonant

Rule 2. Unison NOT allowed on stressed note

Rule 3. Dissonance may not occur on all of beats 2, 3, and 4 in a measure

Rule 4. Dissonances not on stressed note must be part of one of the following sequences:

(a) a passing tone

(b) a neighboring tone

(c) a double neighbor

(d) a nota cambiata

Rule 5. No Parallel 5ths or 8ves between a measure's stressed beats and the preceding downbeat

legalSpecies4

This species generation follows subsets of the legalGeneralTwoVoice rules and legalGeneralCounterpointMelody rules along with a set of its own species 4 rules which are enumerated below.

Use legalGeneralTwoVoice Rules 1, 4, and 5

Use legalGeneralCounterpointMelody Rules 2, 3, 4, 5, 6, 8, and 9

Rule 1. Weak beat of syncope must always be consonant

Rule 2. If strong beat of syncope is dissonant, must resolve downward by step

Rule 3. At most two octaves in a row on strong beats

Rule 4. Must end with 7-6 or 2-3 cadence

Rule 5. Ends on root (doesnt need to start on root or 5th)

Rule 7. Start with a consonant interval

Rule 8. Must be diatonic except for 3rd or 2nd to last notes which may be a maj6 or maj7 interval in Dorian, Mixolydian, or Aeolian modes

Rule 9. Parallel 5ths or 8ves with a dissonance in between not allowed

Rule 10. Weak beat species voice should not be the same note as previous Cantus Firmus note.

Rule 11. Use all these other rules

Chapter 4

Sample Syllabus

The Guitarist's Autodeus

Presented By:

The University of Denver's

Lamont School of Music and

College of Computer Science and Engineering

Autodeus is a web-based interactive sight-reading tutorial for guitarists offered free of charge to all users through the website. The following guide is a 2-year sequence of study in 2 levels. Level I is divided into 2 parts and each part may be subdivided further according to individual or programmatic needs. The gradually progressing materials are produced by the software in ever-changing random sequence, providing a virtually inexhaustible number of exercises for each level of study. Level II incorporates variable user-managed options for training individualized areas of sight-reading including specific time-signatures, rhythmic groupings, positions, and strings outside of the pre-established progressive sequence.

First Year Syllabus (Level I)

Part I

With each “Open String Exercise”, use the given time signatures, note durations, strings, and frets and practice each of the following permutations of different numbers of notes present in the chords:

- Single Notes
- Dyads
- 3-Note Chords
- 4-Note Chords
- Mixture of all of the above
- Mixture of all of the above including 5 and 6 note chords

With each “Roman Numeral Position” Section below, use the given time signature and note durations and practice the following with single note phrases comprised mostly of each of the given intervals and then mix in dyads of those intervals with the single note lines:

- Half steps and Whole steps
- 3rds
- 4ths
- 5ths
- 6ths

Open String Exercises in C Major

- 4/4 whole note durations

Open Strings and 12th fret in C Major

- 4/4 with whole and half note durations

Open Strings and 7th fret in C Major

- 4/4 with whole and half note durations
- 4/4 with whole, half, and quarter note durations and rests

Open Strings, 5th, 7th, and 12th frets in C Major

- 4/4 with whole, half, and quarter note durations and rests
- 2/4 with half, quarter, and eighth note durations and rests
- 3/4 with half, quarter, and eighth note durations and rests

Open Strings and 9th fret in C Major

- 4/4 with whole, half, quarter, and eighth note durations and rests

Open Strings, 3rd, 5th, 7th, 9th and 12th frets in C Major

- 2/4 with half, quarter, eighth, and sixteenth note durations and rests
- 4/4 with whole, half, quarter, eighth, and sixteenth note durations and rests
- 3/4 with half, dotted half, quarter, eighth, and sixteenth note durations with rests

VII Position Diatonic Exercises (Frets 7,8,9,10) in C and G major with their corresponding modes

1. 2/4 with half, quarter, dotted quarter, eighth, sixteenth notes and rests
2. 4/4 with whole, half, dotted half, quarter, dotted quarter, eighth, sixteenth notes and rests
3. 3/8 with quarter, dotted quarter, eighth, dotted eighth, sixteenth notes and rests
4. 6/8 with dotted half, quarter, dotted quarter, eighth, dotted eighth, sixteenth notes and rests

II Position Diatonic Exercises (Frets 2, 3, 4, 5) in G and D major with their corresponding modes

Use all of the same time signatures and note durations as the previous section

IX Position Diatonic Exercises (Frets 9, 10, 11, 12) in D and A major with their corresponding modes

Use all of the same time signatures and note durations as the previous section

IV Position Diatonic Exercises (Frets 4, 5, 6, 7) in A and E major with their corresponding modes

Use all of the same time signatures and note durations as the previous section

Part II

Single String Exercises (open to 16th fret)

For each separate string, use a variety of diatonic scales, then the chromatic scale with the following rhythmic variations:

1. 3/8 with quarter, dotted quarter, eighth, dotted eighth, sixteenth notes, and rests
2. 2/4 with half, quarter, dotted quarter, eighth, dotted eighth, sixteenth notes, and rests
3. 3/4 with half, dotted half, quarter, dotted quarter, eighth, dotted eighth, sixteenth notes, and rests
4. 6/8 with dotted half, quarter, dotted quarter, eighth, dotted eighths, sixteenth notes, and rests
5. 4/4 with whole, half, dotted half, quarter, dotted quarter, eighth, dotted eighth, sixteenth notes, and rests

Two String Exercises (open to 16th fret)

Repeat all exercises as in the previous section, but use 2 strings at a time with the following combinations:

1. Sixth (Low E) and Fifth (A)
2. Sixth (Low E) and Fourth (D)
3. Sixth (Low E) and Third (G)
4. Sixth (Low E) and Second (B)
5. Sixth (Low E) and First (High E)

6. Fifth (A) and Fourth (D)
7. Fifth (A) and Third (G)
8. Fifth (A) and Second (B)
9. Fifth (A) and First (High E)
10. Fourth (D) and Third (G)
11. Fourth (D) and Second (B)
12. Fourth (D) and First (E)
13. Third (G) and Second (B)
14. Third (G) and First (High E)
15. Second (B) and First (High E)

VI Position Diatonic Exercises (Frets 6,7,8,9) in B and F major with their corresponding modes

Same as section VII Position Diatonic Exercises

I Position Diatonic Exercises (Frets 1,2,3,4) in F and Db major with their corresponding modes)

Same as section VII Position Diatonic Exercises

VIII Position Diatonic Exercises (Frets 8,9,10,11) in Db and Ab major with their corresponding modes

Same as section VII Position Diatonic Exercises

III Position Diatonic Exercises (Frets 3,4,5,6) in Ab major, Eb major with their corresponding modes

Same as section VII Position Diatonic Exercises

X Position Diatonic Exercises (Scales of Eb major, Bb major with their corresponding modes

Same as section VII Position Diatonic Exercises

V Position Diatonic Exercises (Scales of Bb major, F major with their corresponding modes

Same as section VII Position Diatonic Exercises

Extended Positions

These exercises are designed to force the student to use what are know as extended positions. Each position starts on the roman numeral indicated fret with the first finger (pointer) and goes up 3 frets so that each finger on the left hand sits on a fret. An extended position is when the left hand's first or fourth finger reaches outside this 4 fret, 4 finger box to 1 fret above with the 4th finger (pinky) or 1 fret below with the 1st finger (pointer) VII Position Diatonic Exercises

1. X - D, F, G, and A♭ major and their corresponding modes
2. IX - D♭, E, F, and G major and their corresponding modes
3. VIII - Scales of C, E♭, F, F major and their corresponding modes
4. VII - Scales of B, D, E, and F major and their corresponding modes
5. VI - Scales of B♭, Db, E♭, and E major and their corresponding modes

6. V - Scales of A, C, D, and E♭ major and their corresponding modes
7. IV - Scales of A♭, B, Db and D major and their corresponding modes
8. III - Scales of G, B♭, C, and Db major and their corresponding modes
9. II - Scales of G♭, A, B, and C major and their corresponding modes
10. I - Scales of F, A♭, B♭, and B major and their corresponding modes

Second Year Syllabus (Level II)

User-Defined exercises

User selects options for each parameter:

1. Pitch Set (determined by area of fingerboard: strings, frets, positions(4 frets), trisects)
2. Note Values
3. Rhythmic profile (irregular groupings, syncopations)
4. Meter
5. N-Chords and/or N-Voice Set
6. Key/Mode

Transformations of User-Defined exercises

Transformations of User-Defined exercises

1. Transposition
2. Augmentation/Diminution
3. Meter Change

Chapter 5

Future Work

There are countless improvements and further research opportunities that could be done in addition to this thesis. Some of those are the following:

- Modify MuseScore's source code so it always uses treble clef. Currently, when an exercise is generated that has mostly lower notes, Musescore decides to interpret the musicXML into an output which uses the bass clef. This is an unknown with respect to fact that it's difficult to estimate how much work this would take.

- Modify MuseScore's source code so that it does not require an X server (only uses command line arguments). Every time an exercise is produced (except for species generation), Musescore is executed and this takes a long time. This would surely speed up generation.

- Experiment with adding triplets into generation. Autodeus does not currently have support for rhythmic musical phrases which sub-divide by 3. For example, randomly adding a 1/3 of a beat in the mix with other established rhythms would most certainly produce very badly formatted rhythmic notation. Lastly, due to the random nature of the program, the inclusion of triplet based rhythms would need to be constrained somehow to actually produce rhythms that are more likely to be created by humans.

- Improvements to the Frontend:
  - Allow users to track progress through the syllabus designed by Dr. Ricardo Iznaola.
  - Ability to save exercises in order to transpose to different keys and/or download them
  - Gathering of user statistics and feedback to get a better idea of how to vary exercises based on selected difficulty such as:
    - Usefulness of exercise
    - Was the exercise too hard or too easy?
    - How pleasing was the melody?
- While the frontend greatly reduces the likelihood of bad input, the actual executable 'GenLesson' could benefit greatly if it performed more input validation when scanning the 'input.fmt' file.
- Using machine learning, it would be massively interesting to use the user-generated statistics to train neural networks which then generate music.
- Study the efficacy of learning to sight-read with Autodeus as opposed to traditional means. This could also offer insight on how to better improve the difficulty settings and their relation to the input parameters.
- It would be great to get some unit test framework setup to test all the various functions and classes, but this could very likely more than double the size of the code base. Validating that each rule produces what is intended and testing edge cases for pre-computation steps would erase a lot of uncertainty.

- Lastly, the frontend could really benefit from some graphic design and extra features. It would be great to optimize the Javascript MIDI playback library downloading and play/stop experience and also have a feature which allows the user to change the playback tempo.

Bibliography

[1] D. Cope, *Computer Models of Musical Creativity*. 2005.

[2] *https://www.sightreadingfactory.com/*.

[3] *http://practicesightreading.com/*.

[4] *http://thefluentguitarist.com/*.

[5] A. Duncan, *http://andrewduncan.net/cmt/*. 1991.

[6] J. Abersold, *http://www.jazzBOOKs.com/mm5/merchant.mvc?Screen=FQBK*. 2010.

[7] C. Fischer, *The Guitar Grimoire*. 1995.

[8] *https://github.com/*.

[9] *https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-gunicorn-and-nginx-on-ubuntu-16-04*.

[10] *http://www.musicxml.com/*.

[11] *http://www.cplusplus.com/reference*.

[12] *http://libmusicxml.sourceforge.net/*.

[13] *https://code.google.com/p/libmusicxml/*.

[14] *http://www.lilypond.org/*.

[15] *https://musescore.org/*.

[16] *https://mido.readthedocs.io/en/latest/*.

[17] Gamma, Helm, Johnson, and Vlissides, *Design Patterns*. 2009.

[18] *http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html*.

[19] *goo.gl/RHThz1*.

[20] *goo.gl/ergKU7*.

[21] *https://msdn.microsoft.com/en-us/library/aa289166(v=vs.71).aspx*.

[22] A. Mann, *The Study of Counterpoint*. 1971.

[23] *http://openmusictheory.com/thirdSpecies.html*.

[24] *goo.gl/M99erul*.