**Introduction**

Many features of modern music are derived from the influence of ragtime, a genre that introduced traits such as the extensive use of syncopation to Western literature around the turn of the 20th century. This paper presents and compares multiple methods to translate music to ragtime, based on the analysis of approximately 11,000 ragtime songs. It additionally presents a novel format to represent melodic and harmonic data of a song in a text based format designed for simple parsing. We generate the bassline for ragtime translations using the algorithm proposed by Scott Bradley in *Ragtimify.* We warp the original rhythm of a song to a more syncopated rhythm using multiple methods. First, we shift notes in measures based on a stochastic analysis of pattern frequencies in the RAG-collection. We also implement the deterministic solution to syncopation described in “Syncopalooza: Manipulating the Syncopation in Rhythmic Performances.” Finally, we perform our own analysis of the RAG-collection to gather transitional data between rhythmic patterns to eventually use in a new stochastic rhythm shifting algorithm.

**The xmk format**

The xmk format is a simple format created to standardize musical input with monophonic melodies to our algorithm. It includes simple musical data consisting of the time signature of the song, the beats per minute, measure boundaries and the durations, notes, and chords for each period. It is designed so that it can be easily read into a list of float arrays.

The header is written as follows:

[time\_signature\_numerator][time\_signature\_denominator][beats\_per\_minute]

Measure boundaries are marked as follows:

=measure\_number

Each onset in the song is written as follows:

duration\_numerator/duration\_denominator midi\_note chord\_root\_midi\_note[alteration]

This format is ideal for storing monophonic melody due to its simplicity and ease to read. We have been working on algorithms capable of turning other music formats, namely midi and humdrum into xmk. However, these could use some improvement.

**Chord generation algorithm**

The chord generation algorithm generates ragtime style chord notes. The xmk input file provides a chord for every onset in the music. These chords can be turned into ragtime format chord progressions based on the rules prescribed in Ragtimify by Scott Bradley, a long-time ragtime player and composer. There are three steps to the algorithm. Firstly, we must read each chord and store them in a list of arrays in term of the notes that make them up. Secondly, we must create a 4-beat pattern in the style of ragtime. Finally, we stochastically choose some second or fourth beats to change to passing tones, tones between its adjacent first and third beats. Each of these steps will be explained in further detail.

**First Step**

To achieve the first step, we need to read in the root note for each chord in the xmk, denoted by the chord name, and figure out the third and fifths of it. We then proceed to add any alterations that come with it, e.g. if it has a 7th, we add a 7th to our chord array. A chord is stored as an array of floats, in which each float is the midi code of a note. The notes in the array will be in order, meaning that the root would lead the array, the third would follow (for a major chord) and so on. After step one, we will have a list of float arrays containing each chord in the song.

**Second Step**

In this step, we create a 4-beat pattern, giving form to the ragtime harmony. Each beat has the following format with slight chances of later modification: first beat, low root note with a chance of adding a mid-root note, creating an octave; second beat, full chord inversion as close to middle C as possible; third beat, low fifth note with a chance of adding a mid-fifth note; fourth beat, full chord inversion as close to middle C as possible. The inversions are produced by translating each note of the chord to its closest counterpart to middle C. The product of this step consists of a list of float arrays in which each array contains three floats: the onset, the offset, and the midi code for the note to be played.

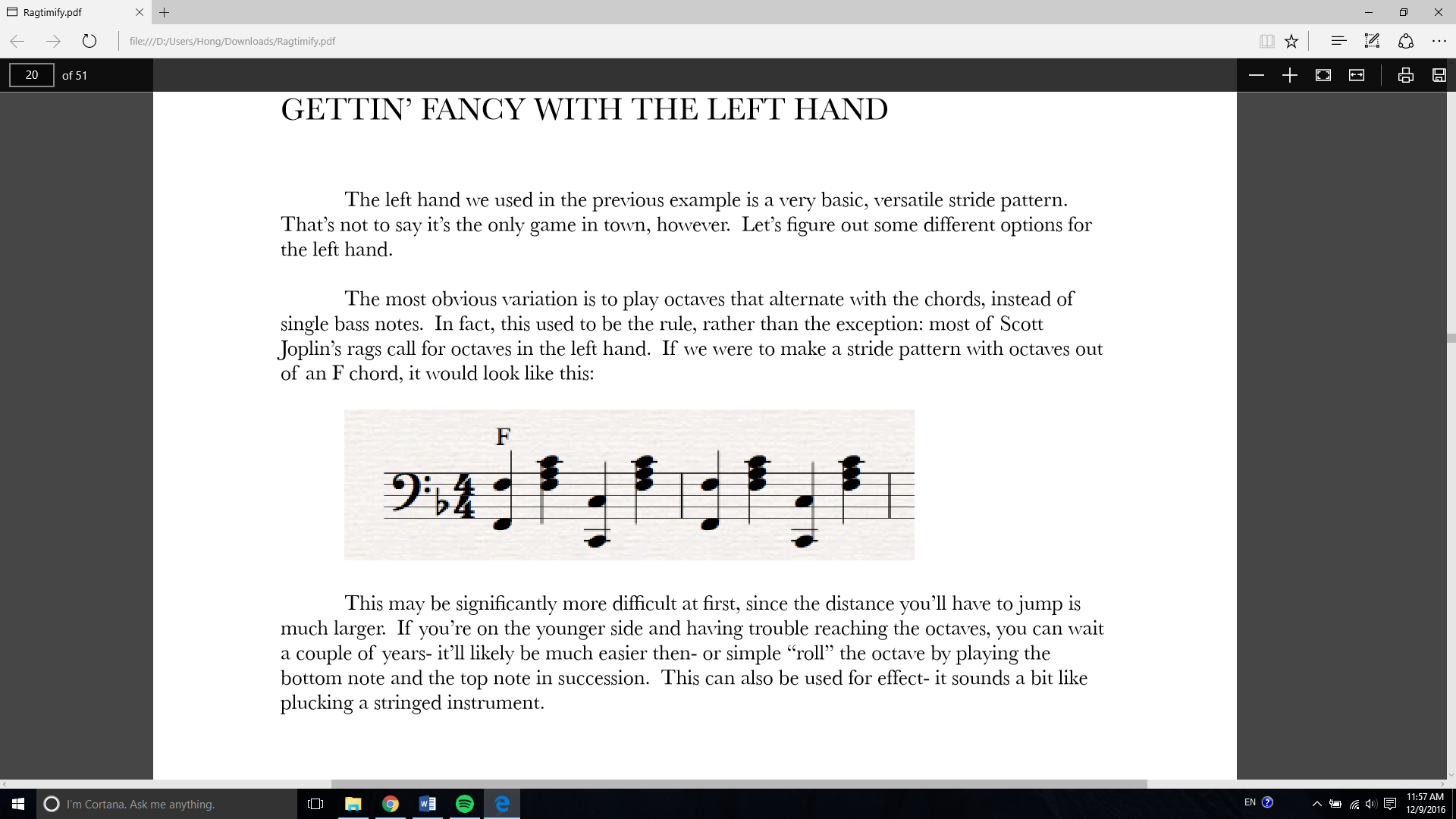


Figure x.0 Graphic representation of the results after step 2

**Third step**

This step selects some measures to add passing tones to. To do so, we must edit either the second or the fourth beat of our 4-beat progressions. Each of these beats, which initially have inverted chords, will be replaced by the arithmetic mean of the midi values of the adjacent first and third beats. The beats to which passing tones is given by a probability p, which seems quite successful when set to .25.

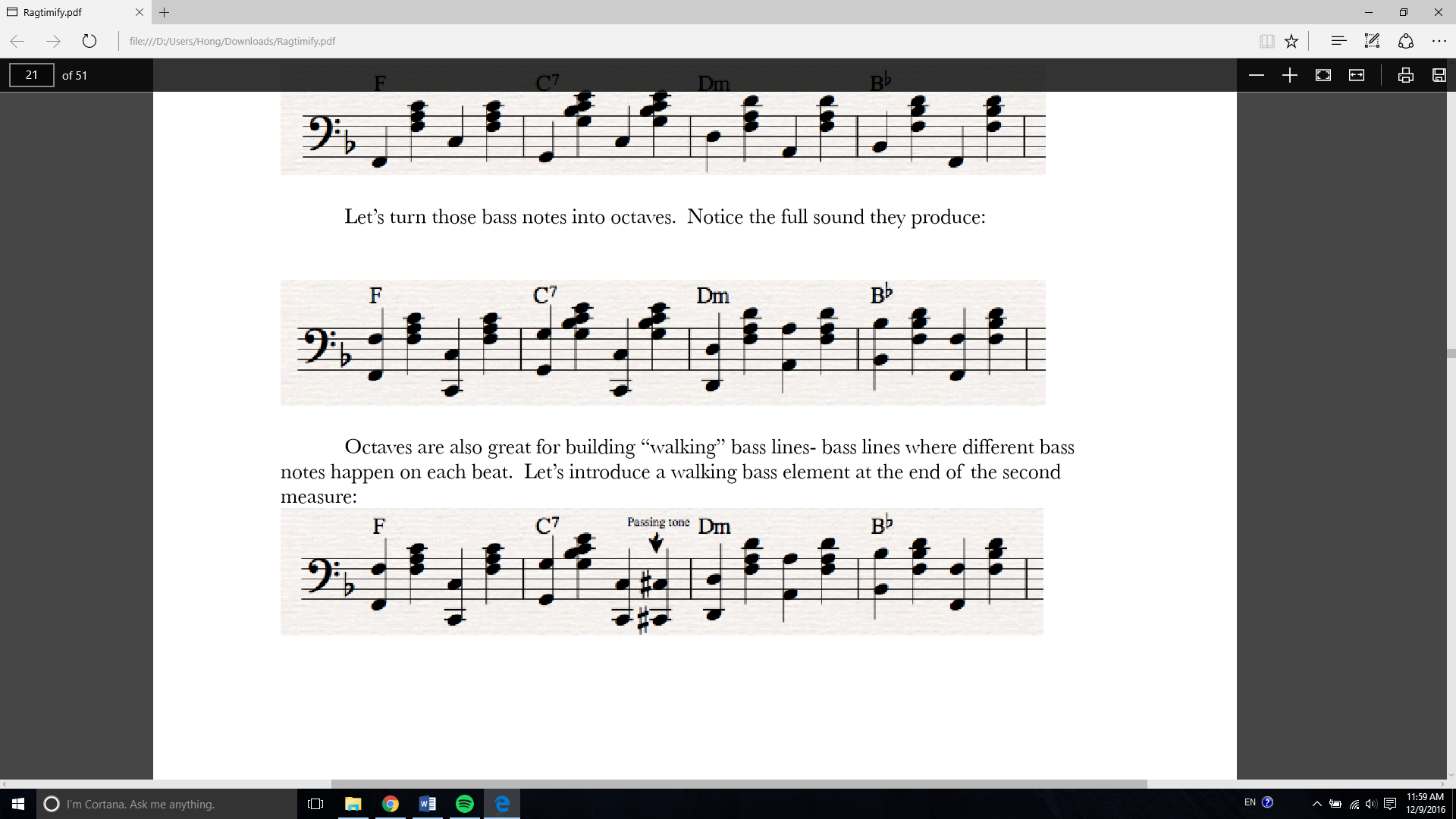


Figure x.0 The addition of a passing tone to the fourth beat of the second measure

**Rhythm Changing Algorithm**

**Version 1:**

**Ragtime Data**

Version 1 of the rhythm changing algorithm generates ragtime rhythms from other rhythmic input. Each half measure is represented by a binary onset pattern of length 16, in which an “I” signifies at least one onset occurring within that time frame and an “O” signifies there being no onsets. For this stage in our research, transitions between measures are not taken into account so only the first 16 of 24 columns given for each binary onset pattern were used.

We use the ragpat dataset to find the number of occurrences of each binary onset pattern, grouped by number of onsets. We are then able to assign a frequency number to each binary onset pattern using this data and the total amount of onset patterns for each number of onsets. Let us refer to these patterns and associated frequencies as the BOP dataset.

**Shifting Rhythm**

We apply the Rhythm Changing Algorithm to each half measure of onsets in a song, input via the xmk format. In order to change the rhythm of a song consistently, we generate a set of rules by which to warp each binary onset pattern in the song. A list of each unique binary onset pattern in the original song is generated. Because the song to be changed is input in the xmk format, assigning binary onset patterns to each half measure is trivial. For each item in this list, a warping rule is selected, where the first item is the binary onset pattern in the original song and the second item represents the pattern that it will be changed to. The second pattern is randomly selected from the items in the BOP dataset with the same number of onsets as the first pattern, weighted by their frequency numbers. Because un-syncopated rhythms (e.g. “IOOOIOOOIOOOIOOO”) are so common in the dataset, rules that would make no changes are overwritten with new selections until they would make changes. Additionally, rules that would shift onsets by a total of 10 or more positions are rewritten to prevent changes too drastic such as a note at the beginning of a half measure being shifted to the end.

The temporal location of each note in the song is then shifted according to the generated rule for the half measure in which that note falls. When multiple notes fall within the time period of one onset in the binary onset pattern, each note is shifted by the same amount to preserve rapid ornaments or embellishments.

**Version 2:**

For the next version of our algorithm, we use transitional data between binary onset patterns in the RAG-collection to improve the quality of the output. We are performing an analysis similar to that of Koops et al., but we intend to record the probability that each binary onset pattern will transition to another. We use an adapted version of Temperley’s Streamer program to isolate streams of notes. The skyline algorithm selects the melody from these streams using the average pitch of all the notes in a given stream as its height. An adapted version of the beat induction algorithm from “Beat Tracking with Musical Knowledge” by Simon Dixon and Emilios Cambouropoulos is used to find the amount of time in one half measure. A slightly inaccurate ticks/beat number generates rhythmic patterns which become shifted throughout the song, resulting in data with wildly varying onset positions. The original algorithm does not produce output accurate enough for our purposes, so several changes were made. Instead of simply taking the most frequent beat length induced by the algorithm, we test the validity of every beat length within 12 ticks of each of the algorithm’s outputs to the tenth of a tick. Since every potential candidate is examined, we remove the merging step for the clusters. A test consists of generating a probability distribution of the 16 onset times for the entire song and finding its standard deviation. A high standard deviation correlates with onsets frequently being in the same metric positions, indicating a success. Candidates that produced the highest standard deviation are then used to generate binary offset patterns for the song, the frequencies and transitional frequencies of which are recorded in our dataset. We ignore songs with standard deviations less than 0.04.

**Syncopalooza:**

For comparison to an algorithmic approach to the syncopation of melodies, we implemented the de-syncopation and re-syncopation algorithms described in “Syncopalooza: Manipulating the Syncopation in Rhythmic Performances.”