SMFFTI User Manual

Applies to v0.45 - December 5, 2023

Table of Contents

Introduction

SMFFTI is short for: *Simple MIDI Files From Text Input*. It is a program for creating basic MIDI files containing chord progressions that have been specified in plain text files. These MIDI files can then be dropped straight into Ableton Live. (The created MIDI should work with other DAWs, but can only be guaranteed to work in Ableton Live, because that's all I've got.)

Editing notes in the Ableton Live piano roll can get a bit tedious, especially for things like velocity. This program tries to make life slightly easier by allowing you to simply specify the chords and a few other basic parameters, such that a MIDI file can quickly be created.

The MIDI files themselves are single-channel (0) and single-track, just as if you had created it inside an Ableton Live channel and exported the clip.

Warning: It's a Windows-only console application, and works only for 4/4 time. Sorry about that, people! Thing is, I created for me, but decided to release it for any interested party.

Nevertheless, it offers some handy features, like randomized velocity, randomized note start/end, downward note transposition (inversion), arpeggiation and randomized chord progressions, rhythm patterns and melody lines.

SMFFTI.exe does not need special installation. Just place it in a convenient folder. To use it, open a Command Window (type "cmd" in the Windows taskbar search field). In the Command Window, locate yourself in the folder where SMFFTI.exe lives. For example, if you put SMFFTI.exe in your Documents folder (eg. *C:\Users\Fred\Documents),* then in the Command Window type:

```
pushd C:\Users\Fred\Documents
```

It's convenient now to also create your SMFFTI command files in the same folder. For this, use your text editor of choice. Notepad will do.

> *Disclaimer: This program is free to use for personal and commercial use, but no responsibility is taken by the developer(s) for errors or failures as a result of using it. It is a purely experimental product and may well contain a number of bugs. If used as expected it should perform adequately, ie. don't specify meaningless or extreme parameter values. Use at your own discretion.*

Basic Usage

NB. To determine which version of SMFFTI you are using, enter this command:

```
SMFFTI.exe -v
```

Creation of MIDI files, and various other SMFFTI operations, is controlled using plain text files that specify various directives and parameters. You can use any text editor you like to create and amend such SMFFTI command files. For a comprehensive example of a SMFFTI command file see section *Full Example Command File*.

Here is the simplest example of a SMFFTI command file that defines a chord progression:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+###############+####### ########+###############+###############
F, Am, G, C
```

> NB. If you want to generate a basic SMFFTI command file that includes most of the necessary parameters, you can use a wizard which asks a few simple questions and then creates a custom file for you. See section *SMFFTI Command File Wizard*.

The first line is a ruler. Each dollar sign indicates the start of a bar. The vertical bars are aligned on 1/4 notes. The dots are aligned on 1/16th notes. (Thus, the smallest note length possible is 1/32nd notes.) The ruler is just a guide for you to see where to place your notes on the second line. Though the example shows two bars, you can specify between one and four bars per line.

(NB. From version 0.4 the above ruler style replaces the old ruler style, which was:

```
[......|.......|.......|.......]
```

The new style ruler is considered clearer to use. However, you can still use the old style ruler.

The second line indicates where the notes play in the bar(s), using sequences of characters consisting of either (a) a single plus sign followed by zero or more hash signs or (b) one or more hash signs. These sequences of characters are analogous to the horizontal notes you draw in the piano roll in your DAW. The plus signs indicate the start of a chord (corresponding to the chord names in the third line) and the subsequent hashes indicate the length of the notes. A series of hashes without an initial plus sign means: Play the same chord again. The number of plus signs MUST correspond to the number of chords specified on the third line.

The third line specifies the chords you want playing. It is simply a series of comma-separated chord names. The number of chords must correspond to the number of plus signs in the second line.

All three lines are required to specify a sequence of chords.

So, in this example, we play F (major) for half a bar; followed by Am for almost a quarter note and Am again for a quarter note; then G (major) for half a bar, and finally C (major) for half a bar. Notice the space at the end of the third quarter note of the first bar: This ends the playing of Am, and then Am is played again for a quarter note, but because it is the same chord, we don't need a plus sign.

We have four plus signs, corresponding to four chords; but we actually have five chords playing.

To convert this text into a MIDI (.mid) file, put the text into a file called, say, mymidi.txt, and run SMFFTI.exe in a Command Window, eg.

```
SMFFTI.exe mymidi.txt test.mid [-o]
```

In this example, SMFFTI.exe and mymidi.txt are in the same folder.

The -o switch is optional and means overwrite the output MIDI file if it exists. If the output file already exists and -o is not specified, your output file will not be created. You should now be able to drag test.mid straight into an Ableton Live MIDI channel.

Here's another two-bar example:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +###### ####### +############## +###############
C(3), Am
```

Notice the number 3 in parentheses. This is a shorthand way of indicating a chord should be played multiple times. This is the equivalent of:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +###### ####### +############## +##############
C, C, C, Am
```

By default the chord root notes are placed in the C3 - B3 octave; thus the lowest note will be C3. This can be changed using a parameter, which is described later on.

There is a limited range of chord types that can be specified, and these are:

- Major (eg. C)
- Dominant 7th (C7)
- Major 7th (Cmaj7)
- Dominant 9th (C9)
- Major 9th (Cmaj9)
- Add 9 (Cadd9)
- Minor (Cm)
- Minor 7th (Cm7)
- Minor 9th (Cm9)
- Minor Add 9 (Cmadd9)
- Sus 2 (Csus2)
- 7 Sus 2 (C7sus2)
- Sus 4 (Csus4)
- 7 sus 4 (C7sus4)
- 5th aka Power Chord (C5)
- Diminished (Cdim)
- Diminished 7th (Cdim7)
- Half-diminished (Cm7b5)

The examples in parentheses here show how you must specify the chord types, eg.

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +###### ####### +############## +##############
C7, Bbadd9, F#maj7, Gsus2
```

Note here the use of a lowercase b to indicate a flat chord, and a hash sign for a sharp chord.

That's it, really, for bashing out quick chord progressions. You just need one or more sets of three lines (1. ruler; 2. note positions; and 3. chord names). You can space them out, ie. insert blank lines wherever you like to make the text file more readable. You can also include comment lines by starting a line with a hash sign. For example:

```
# This is a chord progression I've been noodling
# with on my guitar.
```

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +###### ####### +############## +###############
C(3), Am

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +###### ####### +############## +###############
C, Am, F, G
```

You can also comment out a series of lines by surrounding them with `(#` and `#)`, eg.:

```
# Disable the first two bars

(#
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +###### ####### +############## +###############
C(3), Am
#)

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +###### ####### +############## +###############
C, Am, F, G
```

So far, so good - you now have a general idea of how SMFFTI operates. But the program includes additional features that add some sophistication to your MIDI files. These features attempt to *save you time* that would be spent laboriously editing the MIDI directly inside Ableton Live, as well as automatically generate *creative* MIDI file content that gives you a good starting point for developing your track's rhythms and melodies.

To achieve this, a SMFFTI file may contain, not only chord progression data, but also *parameters*. Parameters are values which modify the behaviour of SMFFTI when it runs. Parameters are indicated with a plus (+) sign at the beginning, followed by a identifier, an equals (=) sign and a value. For example:

```
+Velocity = 80
```

Note that all parameters must be defined *before* the chord progression data. In other words, the chord progression data must be at the *end* of the file.

The rest of this document describes the various operations and features of SMFFTI, and makes references to a multitude of parameters. Be sure to consult the *Command File Parameter Reference* section for information about all the parameters that can be included in SMFFTI command files.

Automatically-Generated Rhythm Patterns

The basic operation of SMFFTI, as you have seen, requires specifying a sequence of *note position strings* - the "`+############## +####### +#######`" strings. So it's up to you to define your rhythm, which may be fine for you. However, SMFFTI allows you to inject a randomized rhythm pattern into the command files, such that "`+############## +####### #######`" is modified to, for example "`+### +### +##### +### +# +## +##`". This feature is called *Auto-Rhythm*.

Auto-Rhythm can be applied to chord sequences, melodies and basslines for groovier, syncopated rhythms.

Auto-Rhythm works by creating a modified version of your SMFFTI command file, and is almost identical except that the original note position strings are replaced with ones that represent an enhanced rhythmic pattern. To invoke *Auto-Rhythm*, execute SMFFTI in this form:

```
SMFFTI.exe -ar mymidi.txt mymidi_autorhythm.txt [-o]
```

Note the `-ar` switch. In this mode a .MID file is not produced, but rather, another SMFFTI command file which, as already mentioned, is almost identical to the input command file, except for modified note position strings.

(Note, again, the -o option to automatically overwrite an existing version of the output file.)

Next, you use the generated command file to create a .MID file, eg:

```
SMFFTI.exe mymidi_autorhythm.txt mymidi_autorhythm.mid [-o]
```

Then you can load the .MID file into Ableton Live and see how it sounds. You can then either tweak the MIDI *in situ*, or run SMFFTI again in *Auto-Rhythm* mode to get another randomized rhythm pattern.

How is the randomized rhythm controlled? By using an algorithm to determine the frequency and length of both the notes and gaps between the notes. Can it be tweaked? Yes: The default operation of *Auto-Rhythm* can be adjusted by including any or all of these three command file parameters in the SMFFTI command file:

```
+AutoRhythmNoteLenBias=<value>
+AutoRhythmGapLenBias=<value>
+AutoRhythmConsecutiveNoteChancePercentage=<value>
```

Command File Parameters are items that you can include to invoke special modes and operations. They are introduced as we describe SMFFTI features, but please consult the *Command File Parameter Reference* section for the full list.

+AutoRhythmNoteLenBias

`+AutoRhythmNoteLenBias` defines a weighting that specifies the chances of notes being a given length. First, you should be aware that, with *Auto-Rhythm*, notes will not traverse a single bar; that is, all notes, regardless of length, will be confined to the single bar they play in.

The length of a note can be a whole note, half note, quarter note, eighth note, sixteenth or thirty-second. The relative chances of which of these lengths to randomly choose is defined in a comma-separated list, eg:

```
      +AutoRhythmNoteLenBias=0, 0, 8, 16, 16, 4
```

Reading from left-to-right, values are specified for whole, 1/2, 1/4, 8th, 16th and 32nd notes. This example means that no whole or half notes are possible, and that it is twice as likely for 8th and 16th notes to be played as 1/4 notes. It is also twice as likely for 1/4 notes to be played as 32nd notes.

Since, as we mentioned, notes are not allowed to carry over into the next bar, you can have only one whole note in a bar, or two half notes, or four 1/4 notes, etc. Therefore, as SMFFTI proceeds through each bar of your original note position template, it will be able to choose only note lengths that fit into the remaining space in the bar. It can thus be seen that, as space in the bar reduces, the chances of fitting in smaller notes increases, overriding the normal randomized selection.

So as you can appreciate, if you want a tighter groove for perhaps an up-tempo track, you will likely give greater priority to shorter notes. It's really a suck-it-and-see process to finally arrive at something you like.

By the way, for the sake of not entirely ruining the rhythm generated, no notes will *ever* begin on even-numbered 32nd notes - it just doesn't ever sound right, in this author's opinion ;-)

+AutoRhythmGapLenBias

+AutoRhythmGapLenBias is similar to `+AutoRhythmNoteLenBias` but defines a weighting that specifies the chances of the *gaps between notes* being a given length. Example:

```
      +AutoRhythmGapLenBias=0, 0, 0, 4, 8, 1
```

This example means that there will never be any whole, half or 1/4 note gaps; and that there is twice as much chance of 16th note gaps as 8ths. Four times the possiblity of 8ths as 32nds.

Technical note: Whereas it's possible (and likely) to have consecutive notes, it is not possible for consecutive gaps - the program applies this to avoid the likelihood of huge gaps occurring between notes. But you shouldn't have to worry about this.

+AutoRhythmConsecutiveNoteChancePercentage

`+AutoRhythmConsecutiveNoteChancePercentage` specifies the chances of *consecutive notes*, that is, no gaps between notes. Percentage value, ie. range 0 - 100. The higher the value, the less gaps between notes. If you are aiming for a stacatto style for a chord progression, you will likely want less chance of consecutive notes, since they will run into each other. Whereas, if your rhythmic pattern is intended to accompany a *melody* instead of a chord progression (see the following sections) then you might well prefer a greater chance of consecutive notes.

Note that *Auto-Rhythm* will always generate a rhythm pattern across the entire bar length; it is not dictated by the note lengths specified in the original chord progression. (This is different from the behaviour of *Automatically-Generated Melodies* (described below) where the notes of the generated melody conform exactly to the note lengths of the original chord progression.)

Automatically-Generated Melodies

The basic operation of SMFFTI creates MIDI files containing chords. But it is also possible to make SMFFTI output *single notes* instead of chords, using the *Auto-Melody* mode. *Auto-Melody* uses randomization to choose which notes to output, based on the underlying chord specified in your source SMFFTI command file.

For extra musicality, in the case of major and minor chords (<u>not</u> suspended or diminished chords), as well as the notes of the underlying chord, by default SMFFTI will also output a few occurrences of notes from the respective *pentatonic* scale. This feature can be disabled by setting parameter +AutoMelodyDontUsePentatonic = 1.

The randomized selection of notes for the melody is biased firstly toward the root and fifth notes, then the third note; and then, if applicable, notes from the chord's underlying pentatonic scale.

To invoke *Auto-Melody*, add this line to your SMFFTI command file:

```
+AutoMelody=1
```

Run SMFFTI as normal, eg:

```
SMFFTI.exe mymidi.txt mymidi.mid [-o]
```

The presence of the `+AutoMelody=1` parameter causes two things:

1. The created .MID file contains a randomized melody line instead of chords.

2. The creation of a text file that echoes the generated melody line. The name of the file includes a timestamp, eg. `mymidi_220728125019.txt`, that makes it unique

every time you generate a new iteration. Here is an example of the contents of this text file:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+#######  +#    +####+# +#  +#  +#+#+ +#  + +#  +#  +#+#  +#  +#
Bb, Bb, Am, Am, Am, Gm, Gm, Gm, Gm, Gm, Gm, Gm, Gm, Am, Am, Am, Am
M: 4, 2, 3, 10, 3, 7, 0, 0, 7, 7, 5, 3, 0, 10, 3, 5, 10

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+###+###+###  +#+#  +#  +###+###    +#  +###+###+#  + +#  +#  +#
Bb, Bb, Bb, Bb, Am, Am, Am, Gm, Gm, Gm, Gm, Gm, Gm, Gm, Gm, Gm
M: 0, 4, 4, 9, 3, 3, 7, 5, 7, 0, 7, 3, 0, 5, 10, 7

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+###+###  +#+###+###  +#    +###+###  +#    + + +###+###    +#+#
Bb, Bb, Bb, Bb, Am, Am, Am, Gm, Gm, Gm, Gm, Gm, Am, Am, Am
M: 2, 2, 4, 7, 7, 10, 7, 0, 10, 3, 7, 10, 0, 5, 0

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+###+#+#  + +###+#+#    + +#+#  +#######  +#    +###  +#  + +# #
Bb, Bb, Bb, Bb, Bb, Ab, Ab, Ab, Ab, F, F, F, F, F, F, F
M: 0, 0, 4, 0, 9, 0, 9, 9, 0, 0, 4, 4, 7, 9, 7, 9, 4
```

This file contains chord progression data from the original file (in this case with a complex rhythm) but also a sequence of notes indicated by the "M:" lines. The values are semitone intervals: Zero meaning the root note.

The melody saved here can be used in any SMFFTI command file by porting across these "M:" lines - more about this in the next section *Specifying a Fixed Melody*.

Note that if you have set `+AutoMelody=1` in the command file it will be ignored if you specify `-ar` (*AutoRhythm*) in the command to SMFFTI.

Note that *Auto-Melody* will always generate a melody line that conforms exactly to the note lengths of the original chord progression. (This is different from the behaviour of *Auto-Rhythm* where the generated rhythm pattern extends to the entire bar length, regardless of the note lengths of the chord progression.)

Specifying a Fixed Melody

The *Automatically-Generated Melodies* section describes one way of creating randomized melody lines (there is actually another), which automatically outputs the generated melody to the .MID file. However, if you want to *fix* the melody, so to speak, so that whenever you run SMFFTI it will always output that melody to the .MID file, what you must do is insert those "M:" melody lines into your SMFFTI command file. They must be placed under the chord list line, eg:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
```

```
+#######  +#    +###+#  +#  +#  +#+#+ +#  + +#  +#  +#+#  +#  +#
Bb, Bb, Am, Am, Am, Gm, Gm, Gm, Gm, Gm, Gm, Gm, Gm, Am, Am, Am, Am
M: 4, 2, 3, 10, 3, 7, 0, 0, 7, 7, 5, 3, 0, 10, 3, 5, 10
```

Their presence will direct SMFFTI to output single notes to the .MID file, as per the melody line, instead of chords.

SMFFTI automatically transposes notes in these melody lines to match the underlying chord. That is, if you copy in a melody line that contains notes, for example, from a minor (pentatonic) scale, but which are aligned with major chords, SMFFTI will correct the relevant notes. The same applies to a major (pentatonic) melody aligned with minor chords.

This brings us to a second method of generating random melody lines: The `-grm` switch, eg:

```
SMFFTI.exe -grm smffti_rand_melodies.txt [-o]
```

This will simply output a whole bunch of "M:"-format melody lines - a thousand, in fact! - to the specified output file. The contents, in part, look something like this:

```
# Generic Randomized Melody lines. Generated by SMFFTI (-grm) at 220725130801

#Melody 0
M: 7, 9, 7, 0, 2, 2, 4, 7, 4, 2, 9, 9, 2, 2, 7, 4, 7, 0, 4, 9, 0, 9, 2, 2, 4, 0,
2, 4, 0, 2, 2, 2, 4, 0, 2, 2, 9, 2, 7, 0, 9, 0, 9, 9, 7, 4, 2, 4, 4, 0, 9, 0, 9,
0, 0, 0, 4, 7, 2, 0, 0, 2, 9, 7

#Melody 1
M: 9, 7, 4, 9, 4, 0, 0, 2, 9, 0, 0, 0, 9, 4, 4, 7, 9, 7, 2, 9, 4, 4, 2, 4, 0, 7,
9, 2, 2, 0, 4, 7, 9, 2, 7, 0, 7, 0, 0, 7, 4, 0, 4, 4, 4, 0, 2, 0, 9, 9, 0, 7, 9,
4, 4, 0, 0, 4, 0, 2, 7, 4, 2, 7

#Melody 2
M: 2, 2, 2, 4, 7, 0, 0, 9, 2, 4, 4, 7, 9, 2, 7, 7, 4, 0, 2, 4, 0, 7, 7, 9, 2, 4,
2, 7, 0, 0, 9, 2, 0, 0, 2, 4, 4, 9, 4, 9, 4, 2, 0, 4, 0, 0, 4, 7, 7, 9, 7, 7, 2,
7, 2, 0, 0, 2, 2, 0, 2, 2, 7, 9
```

Why a thousand melody lines? Well, why not! You can now try any of these out by simply copying-and-pasting to your SMFFTI command file.

Some things to note:

1. The melody lines all use the major pentatonic scale but, as mentioned above, SMFFTI will auto-correct when they are aligned with minor chords.

2. Each melody lines consists of 64 notes, which is likely enough for two bars at a time. (Although SMFFTI allows for four-bar sequences on a single line, in terms of text file editing, two bars is more manageable.) But you do not have to worry about the extra notes in excess of the number of chords in your source SMFFTI command file: They will be ignored. So, If your command file specified 12 chords in a sequence, the last 52 notes of an inserted melody line will be ignored.

3. If you have *Auto-Melody* active in your command file (`+AutoMelody=1`), but you also

have the presence of "M:" melody lines, then those melody lines are *fixed* and will be output to the .MID file, as well as the timestamped copy file that saves the melody lines. This applies on a line-by-line basis; for example:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+#######  +#    +####+# +#  +#  +#+#+ +#  + +#  +#  +#+#  +#  +#
Bb(2), Am(3), Gm(6), Gm(2), Am(4)
M: 7, 9, 7, 0, 2, 2, 4, 7, 4, 2, 9, 9, 2, 2, 7, 4, 7, 0, 4, 9, 0, 9, 2, 2,
4, 0, 2, 4, 0, 2, 2, 2, 4, 0, 2, 2, 9, 2, 7, 0, 9, 0, 9, 9, 7, 4, 2, 4, 4,
0, 9, 0, 9, 0, 0, 0, 4, 7, 2, 0, 0, 2, 9, 7

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+###+###+###  +#+#  +#  +###+###    +#  +###+###+#  + +#  +#  +#
Bb(4), Am(3), Gm(3), Gm(2), Gm(4)

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+###+###  +#+###+###  +#    +###+###  +#    + + +###+###    +#+#
Bb(4), Am(3), Gm(2), Gm(3), Am(3)

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+###+#+#  + +###+#+#    + +#+#  +#######  +#    +###  +#  + +# #
Bb(5), Ab(4), F(7)
```

The first two bars here include a fixed melody line, so *Auto-Melody* will not generate another random melody in that case, but will, of course, do so for the other three sets of two bars sequences.

For such fixed melody lines, mismatched minor/major notes will be corrected by SMFFTI for the corresponding chords when output to the timestamped copy file.

At any time, of course, you can disable an inserted melody line by commenting out the "M:" line, using a # character.

NOTE: The -grm method produces melody lines with no biasing towards chord notes and which are also likely to contain a few bum notes when aligned with suspended/diminshed chords (since the choice of notes could be taken from the pentatonic scale). Thus, the Auto-Melody way of generating randomized melodies is the preferred method, but we're keeping the -grm feature for now.

Random Funk-Guitar Grooves

A slightly more specialized type of rhythmic pattern, tailored for the rapid strumming style of funk guitar, is outlined here. It's operation is designed for use when `+FunkStrum` is enabled. SMFFTI offers two methods for generating randomized funk-guitar-style grooves.

Method One

This method creates a random groove on the fly and inserts the generated result directly into your output MIDI file. Simply replace the note position line with the keyword "RandomGroove". So, instead of this:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +###### ####### +############## +###############
Em7(3), Am7
```

you have this:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
RandomGroove
Em7
```

Note also you should specify ONLY ONE chord name. Once you drop the MIDI file into Ableton Live it's relatively simple to move the notes vertically to introduce other chords.

As with normal note position lines, you can specify up to four contiguous rulers on the same line.

Method Two

This method generates a complete SMFFTI command file, with all the required parameters for funk strumming, along with a 8-bar randomized groove and pre-determined chords (essentially in the key of Em). Thus you will have a "hard copy" of the groove in case you want to copy or edit it.

This method also outputs multiple chords to demonstrate the groove more musically, which you can edit manually in the generated command file.

To create such a command file, execute SMFFTI using the -rfg (*Random Funk Groove)* switch, eg:

```
SMFFTI.exe -rfg rfgmidi.txt [-o]
```

This example creates a command file called rfgmidi.txt.

Again, the -o switch is optional but is necessary if you want to overwrite an already-existing output file.

Every invocation of either method will generate a unique groove, but you might want to finesse the results by stitching together bits of multiple versions in order to create the most funkadelic groove ever!

Automatically-Generated Chord Progressions

For the lazy, or those who struggle to come up with nice chord progressions, there is the *Auto-Chords* feature. Based on a few parameters specified in the SMFFTI command file, *Auto-Chords* will create a chord progression for you. The progression will contain randomly-chosen chords set to a rhythm pattern, which is also randomly-generated.

Your SMFFTI command file only needs to specify a single bar and a note that represents the key for the chord progression. For example:

```
$ . . . | . . . | . . . | . . .
+################################
Gm
```

This says that you want a chord progression in the key of G Minor.

If you want the possibility of *modal interchange*, you can specify parameter +ModalInterchangeChancePercentage, giving it a value between 0 and 100. Modal interchange defined here means chords from the corresponding (or 'opposite') minor/major key. For example, if your main key is G Minor, then modal interchange will use chords from G Major. Similarly, if your main key is Eb Major, modal interchange will use chords from Eb Minor. Thus, instead of a choice of seven primary chords for your generated chord progression, SMFFTI will now select from a possible fourteen primary chords. If +ModalInterchangeChancePercentage is set to 0 then Modal Interchange will not occur; if it is set to 100, then *all* the chords will be from the 'opposite' key (which is a bit pointless!). You probably only want occasional modal interchange, so perhaps set a value of 10 for +ModalInterchangeChancePercentage.

Auto-Chords is invoked by using the -ac switch in your SMFFTI command. For example:

```
SMFFTI.exe -ac mymidi.txt mymidi_ac.txt [-o]
```

Note the `-ac` switch. In this mode a .MID file is not produced, but rather, another SMFFTI command file which is very similar to the input command file, except it contains the generated chord progression and rhythm pattern.

(Note, again, the -o option to automatically overwrite an existing version of the output file.)

Next, you use the generated command file to create a .MID file, eg:

```
SMFFTI.exe mymidi_ac.txt mymidi_ac.mid [-o]
```

Then load the .MID file into Ableton Live and see how it sounds.

Note that it is almost certain that the chord progression and rhythmn pattern will need customizing. The *Auto-Chords* algorithm is limited and able only to *suggest* something that can be used as a starting point for further development. However, it can save you time and help your creativity. Nevertheless you might need to run SMFFTI several times to discover a progression/rhythm that works for you.

By default you will get a four-bar chord progression. However, you can specify a two-, four-, eight- or sixteen-bar progression by including the `+AutoChordsNumBars` parameter in your SMFFTI command file, eg:

```
+AutoChordsNumBars=16
```

Auto-Chords Customization

Okay, so the default settings in SMFFTI attempt to make *Auto-Chords* work without user-intervention, but these settings can all be overridden. So let's talk about the *Auto-Chords* algorithm a bit more so you better understand how to control it. Regarding the *randomness* of the generated chord progression there are three aspects to consider:

1. Major/Minor/Diminished Chord Bias.
2. Chord Type Variations.
3. Note Length and Alignment.

These are all discussed below.

Major/Minor/Diminished Chord Bias

Your chosen key will, of course, consist of seven chords: three major, three minor and one diminished. It is thus a question of how you would like these seven chords to appear in your chord progression to get a pleasing balance. (For a start, it is unlikely that you will want too many instances - if any - of the diminished chord because it usually sounds awful!)

For a more cheery sound, you will prefer more major chords; for melancholy, more instances of minor chords. SMFFTI provides parameters that allow you to balance occurrences of major, minor and diminished chords. For example, you can, if you wish, exclude the possibility of *any* minor chords. Or major chords; or diminished. This is all achieved using the `+AutoChordsMinorChordBias` and `+AutoChordsMajorChordBias` parameters, and is best explained with an example:

```
+AutoChordsMinorChordBias=22, 44, 32
+AutoChordsMajorChordBias=32, 32, 32
```

Now, pay attention...

If your chosen key is a *minor* key, then `+AutoChordsMinorChordBias` is relevant (otherwise, for a *major* key, `+AutoChordsMajorChordBias` applies).

For a *minor* key (where parameter `+AutoChordsMinorChordBias` applies): The three numerical parameters specify a percentage value - ie. a maximum combined total of 100 - that specify a bias toward, respectively, (1) the root chord, (2) the other two *minor* chords in the key and, (3) the three *major* chords. If the total of these values is less than 100, then the remainder percentage will be allotted to the diminished chord.

So, in this example, if you have specified a minor chord as the key (eg. Gm) there is a 22% chance that the chord progression will contain the root chord (Gm). There is also a 44% chance of the other two minor chords (Cm, Dm) occurring; and, a 32% chance of the major chords (Bb, Eb and F) occurring.

Since 22% + 44% + 32% = 98%, it means we have 2% chance of the diminished chord (A dim, or its variants) appearing. That is, any percentage left over is allotted to the diminished chord.

The same principle applies if you have specified a *major* chord as your key: Parameter `+AutoChordsMajorChordBias` will apply. The three numerical parameters specify a percentage value that specify a bias toward, respectively, (1) the root chord, (2) the other two *major* chords in the key and, (3) the three *minor* chords. Again, if the total of these values is less than 100, then the remaining percentage will be allotted to the diminished chord. In the example parameter above, there will be an equal chance (32%) of the root chord (eg. Bb), the other two major chords (Eb, F) and the three minor chords (Gm, Cm, Dm) occurring. 32% + 32% + 32% = 96%, so there is a 4% chance of the diminished chord occurring.

NB. All the above is with respect to the basic chord triads, but *Auto-Chords* can enhance these (by default or with parameters) by adding additional notes - see *Chord Type Variations* below.

Chord Type Variations

By this we mean all the basic triads, plus the chords that *add* extra notes, or *shift* existing notes, to enhance the basic triads that constitute the major/minor/diminished chords. With respect to *Auto-Chords*, there are seventeen chord type variations:

- Major
- Dominant 7th
- Major 7th
- Dominant 9th
- Major 9th
- Add 9
- Sus 2
- 7 Sus 2
- Sus 4
- 7 Sus4
- Minor

- Minor 7th
- Minor 9th
- Minor Add 9
- Diminished
- Diminished 7th
- Half-diminished

So, for example, a basic C *major* triad *could* be randomly rendered as a *Dominant 7th*, *Major 7th*, *Dominant 9th*, *Major 9th* or *Add 9* (five chord variations).

Similarly, a basic C *minor* triad *could* be randomly rendered as a *Minor 7th*, *Minor 9th* or *Minor Add 9* (three chord variations).

Regarding suspended chords - which are not, it seems, associated *per se* with major or minor chords - SMFFTI arbitrarily works (in accordance with the default or user-specified parameters that influence random selection) by converting *major* chords to suspended chords. Suspended chords, when used sparingly, sound great when inserted as musical punctuation.

Even more sparingly used are the diminished chords. The plain *diminished* chord sounds pretty awful at the best of times but, interestingly enough, the *diminished 7th* and the *half-diminished* can sound quite nice. So, for *Auto-Chords*, SMFFTI will never, by default, select a basic *diminished* and, unless specified by the user, will always randomly choose equally between a *diminished 7th* and a *half-diminished*.

Thus, bearing in mind all the above-mentioned, SMFFTI provides a total of seventeen parameters that control the chances of each of the above-listed chord types being chosen for the generated chord progression. These parameters (shown with example values) are:

```
# Major chords (and suspended)
+AutoChords_CTV_maj       = 1000
+AutoChords_CTV_7         =  100
+AutoChords_CTV_maj7      =  150
+AutoChords_CTV_9         =   50
+AutoChords_CTV_maj9      =   50
+AutoChords_CTV_add9      =  200
+AutoChords_CTV_sus2      =   15
+AutoChords_CTV_7sus2     =   15
+AutoChords_CTV_sus4      =   10
+AutoChords_CTV_7sus4     =   10

# Minor chords
+AutoChords_CTV_min       = 1000
+AutoChords_CTV_m7        =  250
+AutoChords_CTV_m9        =   50
+AutoChords_CTV_madd9     =  150

# Diminished chords
+AutoChords_CTV_dim       =    0
+AutoChords_CTV_dim7      =    1
```

```
+AutoChords_CTV_m7b5      =    1
```

(NB. CTV = Chord Type Variation)

The values you supply are relative and allow you to set a proportional bias across the chord type variations. The maximum value that can be specified is 100,000. You do not have to specify any of these values - SMFFTI uses the above values as defaults.

Note that the chord types are categorised by Major, Minor and Diminished, and the relative values specified for one category do not affect the bias in the other two categories. For example, if you were to set `+AutoChords_CTV_m7 = 100000` it would mean only that, when a minor is output it will most likely be a Minor 7th; it would not affect the bias of major or diminished chords. That is why, for diminished chords, we need only specify very small values to accomplish the bias toward Diminished 7th and Half-diminished.

Note Length and Alignment

This is all about the *rhythm* of the chord progression: Where the notes start and end and the gaps in between.

Note length can randomly be between an 1/8th and a whole note (ie. range 4 to 32 1/32nd notes). Notes that are half-note or smaller are considered by SMFFTI as *short* notes; any notes larger are considered *long* notes. You can set a bias toward shorter or longer notes using the +AutoChordsShortNoteBiasPercentage parameter. For example:

```
+AutoChordsShortNoteBiasPercent = 50
```

will tell SMFFTI to use a 50/50 chance of outputting short notes. If +AutoChordsShortNoteBiasPercentage is set to 100, all notes output will be half-notes or shorter. Thus, you can bias note length for more rhythmic patterns (shorter notes) or pad/harmony patterns (longer notes).

If you do not specify a value for +AutoChordsShortNoteBiasPercent, SMFFTI uses a default value of 35, meaning about a one-in-three chance of shorter notes.

Regarding the length of the gaps between the notes, this is determined by the fact that SMFFTI will always align the notes - randomly - on 1/8th, 1/4 or half-notes. It uses a fixed bias slightly in favour of 1/4-note alignment over 1/8th note alignment; with only occasional half-note alignment. This aspect of the algorithm is fixed and, at this time, cannot be modified by the user.

Random Chord Replacement

The *Random Chord Replacement* (RCR) facility allows you to selectively replace individual chords in a progression in a SMFFTI command file. Chords that are marked for replacement will be substituted with randomly chosen chords. This feature thus allows you to audition different chords - randomly chosen - as part of a progression you are developing.

RCR has been implemented to augment the use of *Auto-Chords*. For example, if you have used *Auto-Chords* to create a progression, it's likely that some of the chords will not sound harmonious, and the idea is that RCR will provide a simple means of substituting the "bad" chords with better ones.

RCR is enabled by using command parameter +RandomChordReplacementKey, which requires a chord that specifies a key, eg.

```
+RandomChordReplacementKey = Gm
```

NB. If +RandomChordReplacementKey is specified, it will be ignored if you are executing SMFFTI in Auto-chords (-ac) mode.

With RCR enabled, you now indicate chords in your progression which will be replaced with a randomly-chosen chords from the key specified by the +RandomChordReplacementKey parameter (in this example G Minor). You do this by prefixing chord names with a question mark. For example:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +############## +############## +###############
Cm, ?Am, F, G
```

The A Minor chord will be changed to another randomly-selected chord. The replacement chord is what is output to your MIDI file but, but SMFFTI keeps a record of the original and replacement chords, by updating the SMFFTI command file. After running SMFFTI for the above example, the command file will be changed to something like this:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +############## +############## +###############
Cm, ?Eb, F, G
#Cm, ?Am, F, G
```

In this example, A Minor has been replaced by Eb Major. Note that the original chord progression line is commented out and a new chord progression line inserted, which is what is reflected in the MIDI output file. In other words, a history of the chord progression iterations is kept (in reverse order, ie. the latest is listed first). Note, too, that the replacement chord is still prefixed with the question mark; this is so that, if you don't like replacement chord, you simply run SMFFTI again and yet another chord progression line will be inserted, with the previous one being commented out, eg.:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
```

```
+############## +############## +############## +###############
Cm, ?Dm, F, G
#Cm, ?Eb, F, G
#Cm, ?Am, F, G
```

This log of the attempted chord progressions is handy for reference - you can see what worked and what didn't. When you are happy with a result, you simply disable RCR (by commenting out - or deleting - the +RandomChordReplacementKey parameter and removing the question marks (you might also tidy up the command file by removing the list of unsuccessful chord progressions.)

If +RandomChordReplacementKey is not enabled, all question mark prefixes are ignored.

The selection of chords for replacement is based on the same algorithm as *Auto-Chords*. That is, you use the same command parameters as you would for *Auto-Chords* that set a bias for choice of chord and chord type variations, eg. +AutoChordsMinorChordBias, +AutoChordsMajorChordBias.

Similar to *Auto-Chords*, you can enable *modal interchange*, so that the selection of substitute chords can also include chords from the corresponding (or 'opposite') major/minor key. You do this by setting parameter +ModalInterchangeChancePercentage.

RCR checks any history lines to avoid selecting chords that have already been tried before (and rejected, hence they exist in the history). This speeds up your workflow since it avoids needless repetition of effort. Only when all possible chords and their allowed variations have been tried will RCR begin selecting them again.

NB. Regarding *Auto-Rhythm*, if your source file is one that has RCR enabled (`+RandomChordReplacementKey` is active and chord names are prefixed with '?'), be aware that the output file will contain modified chords. This is likely not what you want. Therefore, you should probably ensure that RCR is disabled before you use *Auto-Rhythm*.

MIDI-to-SMFFTI Conversion

Using this feature you can convert a chord progression in a MIDI file into SMFFTI text format. The purpose of this facility is to allow you to perform manual customization of chords in Ableton Live, and then preserve the result in SMFFTI format. This customized chord progression might be required, for example, as the basis for mulitple iterations of *Auto-Rhythm* or *Auto-Melody*.

To emphasize this, suppose you used a combination of *Auto-Chords* and *Random Chord Replacement* to create a working chord progression in a MIDI file. Inside Ableton Live, you are likely going to chop and change the chord positions and length to come up with a nice composition. Once this creative process is complete, you can consider the finished chord progression to be the backbone, as it were, of a musical section; and from which you might use SMFFTI to further generate additional rhythm patterns and melodies. Therefore, you will need to save this 'master' chord progression in SMFFTI format.

The *MIDI-to-SMFFTI* operation is achieved using the -m switch. For example:

```
SMFFTI.exe -m mymidi.mid chords.txt [-o]
```

If the output file already exists, the operation will not be allowed unless you specify the override (-o) option. The override option, in this case, will cause the output to be *appended* to the existing file, rather than completely overwrite it.

The text output will consist of the essential SMFFTI chord progression data, ie. ruler, note positions and chord names, eg.:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############### +############### +############### +###############
F, G, Am, C
```

The specified output file is expected, of course, to be a text file, but it might contain *any* text - SMFFTI does not care. The chord progression data (interpreted from the MIDI file) is always appended to the existing file. However, significantly, if the file is a SMFFTI command file, the *MIDI-to-SMFFTI* operation will *replace all existing chord progression data with the new data*. This allows you to quickly update your original SMFFTI command file with modified chord progression data without need to perform manual editing. This only happens if the specified file is perceived to be a SMFFTI command file.

Your source MIDI file must be a one-channel, one-track file - the type of file that is created when you export from Ableton Live. It must also be meaningful, in the sense that it contains actual chords, ie. typically, at least three notes. Valid chords are those eighteen chord types that are listed in the *Basic Usage* section of this document.

The notes of the chords do not have to be quantized, or precisely aligned; but the length of each chord will be determined by the *length* of the *first-played* note of the chord.

All chord positions will be quantized to 1/32nd notes for the sake of the SMFFTI text format.

NB. A note about Diminished 7th chords. Because these chords use four notes that are each three semitones apart, they are somewhat ambiguous. For example, Adim7 (A-C-Eb-Gb) uses the same notes as Cdim7 (C-Eb-Gb-A) - they are just in a different order. Since SMFFTI does not concern itself with precise music theory as such, it interprets such chords as simple transpositions. It will be observed, therefore, that a MIDI file containing an Adim7 chord will, when written to SMFFTI format using this function, will be labelled as Cdim7. Do not be alarmed - it's just the simple-mindedness of SMFFTI!

Set Parameter Operation

It is possible to set or change the value of a SMFFTI file parameter using the *Set Parameter* command. The *Set Parameter* operation is invoked using -p switch, followed by a valid parameter definition. For example:

```
SMFFTI.exe -p "+AutoMelody = 1" chords.txt
```

This command tells SMFFTI to update the `chords.txt` file with the +AutoMelody parameter, setting it to 1.

If `chords.txt` already has the specified parameter defined, it will update the value to the one specified in the command. Otherwise, it will add the parameter as a new definition, placing it at the end of all existing paramater data, and just before the first chord progression ruler line.

The purpose of *Set Parameter* is to facilitate command batch (*Windows* .bat) files. For example, a command file might contain the following:

```
copy /y chords.txt melody.txt
smffti.exe -p "+AutoMelody = 1" melody.txt
smffti.exe -ar melody.txt melody_r.txt -o
smffti.exe melody_r.txt melody_r.mid -o
```

This command file copies `chords.txt` to `melody.txt`, and then sets the value of +AutoMelody in file `melody.txt`. This allows the remaining commands to proceed without you needing to manually edit `melody.txt` first. Using the original `chords.txt` file, this command sequence results in the creation of the MIDI file `melody_r.mid`, which contains a randomly-generated melody line - all accomplished in a single step, as it were.

This use of *Set Parameter* enhances the possibility of 'programming' SMFFTI to execute a series of commands without manual user intervention, allowing rapid regeneration of output files.

SMFFTI Command File Wizard

The *SMFFTI Command File Wizard* is an easy way of allowing you to create a basic command file, by asking you a few simple questions. To invoke the wizard (using a *Windows* command file window, of course) enter this command:

```
SMFFTI.exe -w
```

Note the -w switch. SMFFTI will then present a very simple menu, with a single option:

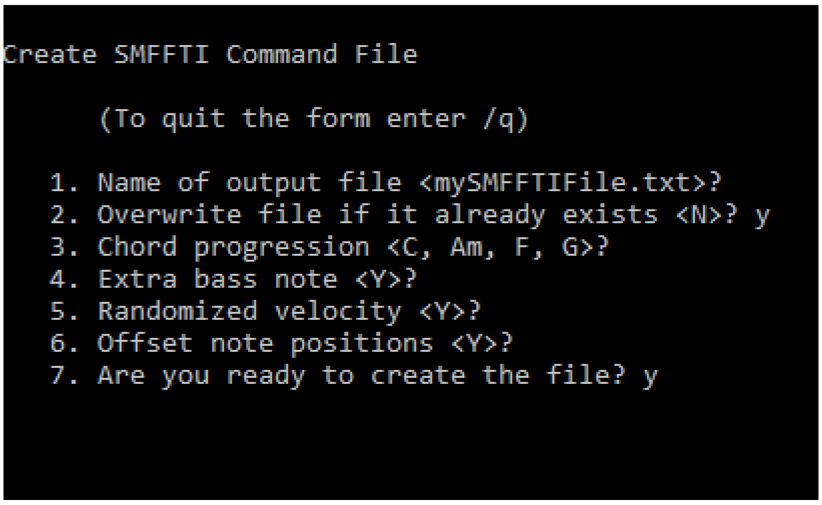
```
1. Create SMFFTI Command File
```

Choose option 1 and it will present you with the first question:

```
1. Name of output file <mySMFFTIFile.txt>?
```

Enter your answer and press RETURN, or accept the default answer shown in angled brackets by simply pressing RETURN. The next question will then be presented.

Here's the full dialog:



```
Create SMFFTI Command File

      (To quit the form enter /q)

   1. Name of output file <mySMFFTIFile.txt>?
   2. Overwrite file if it already exists <N>? y
   3. Chord progression <C, Am, F, G>?
   4. Extra bass note <Y>?
   5. Randomized velocity <Y>?
   6. Offset note positions <Y>?
   7. Are you ready to create the file? y
```

A few points about this wizard dialog...

When entering the chord progression, use a comma-separated list, and make sure the chord names follow the naming convention shown in the *Basic Usage* section of this document (eg. C, Cm7, Cdim7). You can enter any number of chords - within reason; perhaps no more than sixteen. It works on the basis of one bar per chord specified, so four chords = a four-bar progression.

Randomized velocity means the individual notes of the chords will have slightly different velocities.

Offset note positions means that the individual notes of the chords will each start and end at different points, very slightly offset from their notional position.

You can use these commands to navigate the dialog:

/q	Quit the dialog.
/b	Go back to the previous question.
/<n>	Go to the question indicated by the number specified, eg. /3.
/l	Go to the latest question requiring input. For example, you may be on question 5, and decide to go back to question 2 by entering /2. After that, you can return to question 5 by entering /l.

After completing all the questions it will indicate that the file is being created; or, reporting any errors that may occur.

Having created the SMFFTI command file you can then inspect/customize it using your text editor, prior to using it as input for one of SMFFTI's main operations.

Command File Parameter Reference

Command file parameters are indicated using a plus sign followed by the parameter name, an equals sign, and the parameter value. For example:

```
+TrackName=My Little Tune

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############## +###### ####### +############## +###############
C, Am, F, G
```

None of the parameters are mandatory, since defaults are applied.

Following is a reference guide to all the parameters, in alphabetical order.

+AllMelodyNotes

Valid values: 0, 1. When set to 1, SMFFTI will output to the MIDI file all possible notes that could be used as a melody. The output thus appears as a "chord", consisting of the underlying chord notes plus, if applicable, some possible extra notes from the pentatonic scale. The purpose of this feature is to allow you to see, in the MIDI file, the full choice of notes suitable for a melody. Typically, you will disable all these notes in your DAW (eg. in Ableton Live, by selecting them and pressing 0 (zero)), and then enable individual notes to manually create a melody. In other words, the full set of notes, expressed as a chord, provides you with an easy-to-see template.

When `+AllMelodyNotes` is set, the following parameters are ignored:

```
+AutoMelody
+RandNoteStartOffset
+RandNoteEndOffset
```

+Arpeggiator

Number in range 0 - 13. If set to zero (or parameter is omitted) no arpeggiation is applied. Values in the range 1 - 13 will arpeggiate the output MIDI. The 13 arpeggiator types mirror those available in the Ableton Live Arpeggiator MIDI device, according to the following:

1. Up
2. Down
3. Up Down
4. Down Up
5. Up & Down
6. Down & Up
7. Converge
8. Diverge
9. Converge & Diverge

10. Pinky Up
11. Pinky UpDown
12. Thumb Up
13. Thumb UpDown

NB. Not all of the Ableton Live Arpeggiator types have been implemented, eg. there is no *Play Order* or *Random* arpeggiator types.

If arpeggiation is enabled, the following are disabled: `+RandNoteStartOffset` `+RandNoteEndOffset` and `+NoteStagger`.

+ArpGatePercent

Range 1 - 200. Default 50. This is a percentage value. 50% means half the arpeggiated note length (similar to the Ableton Live Arpeggiator device).

+ArpOctaveSteps

Range -6 to 6. Sets the number of times the pattern is transposed. The pattern will play once at its original transposition and then in successively higher or lower octaves according to the value specified. (Similar to the *Steps* value in the Ableton Live Arpeggiator device.)

+ArpTime

Valid values: 1, 2, 4, 8, 16, 32. Default (if omitted) is 8. The value is applied as a fraction, eg. `+ArpTime=8` means 1/8th.

+AutoChords_CTV_<type>

Applies only when *Auto-Chords* mode (-ac) is used. For setting the chance of *Chord Type Variations* (CTV) occurring, eg. how often a Cm chord might be converted to Cm7.

This is a *group* of seventeen parameters, and <type> can be one of the following (with its chord type shown in parentheses):

- maj (Major)
- 7 (Dominant 7th)
- maj7 (Major 7th)
- 9 (Dominant 9th)
- maj9 (Major 9th)
- add9 (Add 9)
- sus2 (Suspended 2nd)
- 7sus2 (7th Suspended 2nd)
- sus4 (Suspended 4th)
- 7sus4 (7th Suspended 4th)

- min (Minor)
- m7 (Minor 7th)
- m9 (Minor 9th)
- madd9 (Minor Add 9)

For example settings, see the *Full Example Command File* section at the end of this document. The values shown there are also the default values if these parameters are not specified.

Each of these parameters takes a numeric value (range 0 - 100000). The collective values of all the parameters defines the bias of the chances of the various chord type variations being used.

Theses parameters are categorised by Major (10, including the suspended chords), Minor (4) and Diminished (3), and the relative values specified for one category do *not* affect the bias in the other two categories, so the relative magnitude of the values can be different for each of the three categories.

+AutoChordsMajorChordBias

Applies only when *Auto-Chords* mode (-ac) is used. For chord progressions in a *major key*, it sets a bias, respectively, toward (1) the root chord, (2) the other two major chords in the key and (3) the three minor chords in the key. The values are expressed as a percentage, so the total value of the three values must not exceed 100. If the total value is less than 100, the remaining percentage is allotted to the diminished chord of the key. Example:

```
+AutoChordsMajorChordBias=22, 42, 32
```

Here, if the key of the chord progression is Bb, there is a 22% chance of the *Bb* chord being used, a 42% chance of *Eb* or *F* being used, and a 32% chance of *Gm*, *Cm* or *Dm* being used. The total percentage value is 96 (22 + 42 + 32), so there will a 4% chance of *A dim* being used.

If not specifed, the default value is 22, 42, 32.

+AutoChordsMinorChordBias

Applies only when *Auto-Chords* mode (-ac) is used. For chord progressions in a *minor key*, it sets a bias, respectively, toward (1) the root chord, (2) the other two minor chords in the key and (3) the three major chords in the key. The values are expressed as a percentage, so the total value of the three values must not exceed 100. If the total value is less than 100, the remaining percentage is allotted to the diminished chord of the key. Example:

```
+AutoChordsMajorChordBias=22, 42, 32
```

Here, if the key of the chord progression is Gm, there is a 22% chance of the *Gm* chord being used, a 42% chance of *Cm* or *Dm* being used, and a 32% chance of *Bb*, *Eb* or *F* being used. The total percentage value is 98 (22 + 42 + 32), so there will a 4% chance of *A dim* being used.

If not specifed, the default value is 22, 42, 32.

+AutoChordsNumBars

Applies only when *Auto-Chords* mode (-ac) is used. Determines the length, in bars, of the output chord progression. Valid values: 2, 4, 8 or 16. Default: 4.

+AutoChordsShortNoteBiasPercent

Applies only when *Auto-Chords* mode (-ac) is used. Range 0 - 100. Determines the chance of shorter notes (1/8th notes up to half-note) being output. If set to zero, only longer notes (17/32nds up to a whole note) will be output. If set to 100, only shorter notes will be output. Default: 35.

+AutoMelody

When enabled (`+AutoMelody=1`) the .MID file created will contain a randomized melody and the chord and generated melody data is saved to a timestamped copy of the original command file so that the melody lines can be replicated elsewhere. The melody data is saved in text format as lines beginning with "M:", underneath the chord list line.

+AutoMelodyDontUsePentatonic

Applies to *Auto-Melody*. When enabled, ie. `+AutoMelodyDontUsePentatonic=1`, *Auto-Melody* will not use any notes from the underlying chord's pentatonic scale. (Pentatonic scale notes will only ever be output for major and minor chords, and not suspended or diminished chords.)

+AutoRhythmConsecutiveNoteChancePercentage

Applicable to *Auto-Rhythm* mode (`-ar` switch). Percentage chance of consecutive notes. Valid range 0 - 100. Example:

```
+AutoRhythmConsecutiveNoteChancePercentage=35
```

A zero value means no consecutive notes - there is always a gap between notes.

A value of 100 means virtually no gaps - it will be almost a thick glut of notes. We say *virtually* because, as already-mentioned, no notes can commence on even-numbered 32nds, so 32nd-note gaps are quite likely.

The default value is 25, ie. modest likelihood of consecutive notes.

+AutoRhythmGapLenBias

Applicable to *Auto-Rhythm* mode (`-ar` switch). Sets a bias of the chances of each possible gap length between notes. Similar parameter structure and operation to `+AutoRhythmNoteLenBias`. Default values: 0, 0, 0, 4, 8, 1.

+AutoRhythmNoteLenBias

Applicable to *Auto-Rhythm* mode (`-ar` switch). Sets a bias of the chances of each possible note length. Possible note lengths: whole note, half note, 1/4 notes, 8th notes, 16th notes, 32nd notes. Best illustrated by example:

```
+AutoRhythmNoteLenBias=1, 2, 3, 4, 5, 6
```

From left-to-right, longest note first: The "1" value refers to whole notes, the "6" refers to 32nd notes. In this example, SMFFTI is six times more likely to output 32nd notes over whole notes; and four times more likely to output 8th notes over whole notes.

Comma-separated list of six values, each value range 0 - 1000. Default values: 0, 0, 4, 8, 4, 2.

+BassNote

A value of 1 causes the program a output an additional root note an octave lower. Omit if bass note not required, or specify `+BassNote=0`.

+FunkStrum

Range 0 - 6. This is an attempt to facilitate a kind of funk strumming dynamic. It kind of works, as long as you've got a decent-sounding guitar instrument.

What this does is apply note stagger such that, for "downstrokes" the notes are staggered from low to high (to simulate a micro-delay between the strings as they are strummed downward); and the opposite for "upstrokes", notes staggered from high to low. The definition of downstroke and upstroke is simply alternate 1/16th notes: Odd 1/16ths are downstrokes, and even 1/16ths are upstrokes. Small values are appropriate: `+FunkStrum=2` seems about the best.

NB. To avoid these somewhat tight notes running into each other, the note length is shortened slightly.

If `+FunkStrum` is enabled it disables the following: `+Arpeggiator`, `+RandVelVariation`, `+RandNoteStartOffset`, `+RandNoteEndOffset`.

NOTE: Funk guitar is all about the strumming groove, but creating a groove manually - either in a DAW or by editing a SMFFTI text command file - is quite tedious. So check out the section Random Funk Grooves below to see a feature of SMFFTI that allows you to geneate randomized funk strumming grooves.

+FunkStrumUpStrokeAttenuation

Applicable only if `+FunkStrum` active. Range 0.1 to 1.0. Default if omitted is 1.0. This allows you to attenuate the velocity of "upstrokes" (even 1/16th notes), the idea being that funk guitar technique often means much reduced volume - frequently muted - for upstrokes. Unfortunately, there's no way we can magic up muted strums, so this will have to do.

A value of 1.0 does not attenuate, while 0.1 drops the velocity to a tenth of the nominal velocity.

+FunkStrumVelDeclineIncrement

Applicable only for `+FunkStrum`. Range 0 - 20. Default if omitted is 5. `+FunkStrum` applies note stagger, and what this does is reduce the velocity of each successive note by the amount specified. It attempts to emulate the likelihood that, when a guitar is strummed, each successive string will be struck with less velocity. It's my unproven theory, nothing more.

For example, if the base velocity is 80 and `+FunkStrumVelDeclineIncrement=10`, the first note velocity will be 80, the second note will be 70, the third will be 60, etc.

+ModalInterchangeChancePercentage

To enable *modal interchange* for *Auto-Chords* (-ac mode) or *Random Chord Replacement*. Valid range 0 to 100, representing a percentage chance that chord choice will come from the corresponding (or 'opposite') major/minor key. A value of zero means *never* use modal interchange; a value of 100 means *always* use chords from the 'opposite' key.

For example, if your chosen key for randomized chords is G Minor, modal interchange will also enable chords to be selected from G Major. Similarly, if your specified key is Eb Major, modal interchange will also use chords from the key of Eb Minor.

For occasional invocation of modal interchange, perhaps set +ModalInterchangeChancePercentage to a value of 10. Default if not specified: 0.

+NoteStagger

Range -32 to 32. This will result in the start position of successive notes of the chord being staggered by the amount specified. A positive value starts from the lowest note, while a negative value starts from the highest note. Either way, the first note is not staggered. A zero value results in zero stagger. Typically, values will be small, say, 3, to simulate perhaps a guitar strum. End positions of notes are unchanged. A non-zero +NoteStagger value disables `+RandNoteStartOffset` and `+RandNoteEndOffset`.

+OctaveRegister

Number in range 0 to 7. Default 3. Determines the general register of where the notes of the chords will be placed. When no downward transposition occurs, it effectively specifies the lowest octave for all notes in the chord progression (except the optional bass note).

+RandNoteEndOffset

Adds some randomization to the end position of each note. Range 0 - 32.

+RandNoteOffsetTrim

By default, if you have applied `+RandNoteStartOffset` and/or `+RandNoteEndOffset`, the MIDI output file will contain an additional bar at the beginning to accommodate notes which start *before* their nominal start position; or an extra bar at the end to accommodate notes which end *after* their nominal end position. By setting this parameter, ie. `+RandNoteOffsetTrim=1`, the program will not append these additional bars, but will ensure that, for the first bar, `+RandNoteStartOffset` will not be applied; and for the last bar, `+RandNoteEndOffset` will not be applied.

+RandNoteStartOffset

Adds some randomization to the start position of each note. Range 0 - 32. This can result in the note starting either before or after its nominal position. Typically this will be quite small, eg. `+RandNoteStartOffset=3`. Useful for "humanizing" piano chords, perhaps.

+RandomChordReplacementKey

Enables *Random Chord Replacement* (RCR), and also defines the key from which replacement chords are randomly selected. For example:

```
+RandomChordReplacementKey = Gm
```

+RandVelVariation

Use this to add some randomization to the velocity of each note. For example, if the base velocity is 80, and you specify `+RandVelVariation=20`, the velocity of each note will randomly be in the range 70 - 90.

+RootNoteOnly

Possible values: 0 or 1. A value of 1 causes SMFFTI to output *only* the root notes of the chord. This can be handy for auditioning auto-generated rhythms for, say, a bassline. Starting with a such root-note-only MIDI file, you can then, relatively easily, transpose notes to create a melody.

+TrackName

Expects some text. When specified, this name is what the clip will be labelled with when you drag it into Ableton Live. If omitted, clip is named as "Made by SMFFTI".

+TransposeThreshold

Number in range 0 to 48. Default 48. This dictates how notes in a chord (including the root note) will be downward transposed. It specifies a number of semitones *above* the note C for OctaveRegister. Notes beyond this will be transposed downward. For example, if `+OctaveRegister=3` and `+TransposeThreshold=11`, it means that any notes in the chord progression that are C4 or above will be transposed downward to below C4. This example demonstrates how to keep all chord progression notes within a one octave register, ie. C3 to B3*. Such transposition is also applied for *Auto-melody*, to keep melody notes within a given register.

*Using the Ableton Live designation of octave numbers.

+WriteOldRuler

In the case of SMFFTI modes that create a modified version of the command file, eg. *Auto-Rhythm*, the style of ruler that is written to the output file may be specified. From version 0.4 of SMFFTI, the default is the new style ruler, ie.:

```
$ . . . | . . . | . . . | . . .
```

However, if you prefer the old style of ruler to be output, ie.:

```
[......|.......|.......|.......]
```

you can use the command file parameter +WriteOldRuler and set its value to one, ie:

```
+WriteOldRuler = 1
```

This is purely a cosmetic feature to assist you should you wish to further customise the modified file: You can choose the style of ruler that best helps you to position notes.

+Velocity

Use this to set the velocity of all notes. Range 1 - 127. If omitted, velocity defaults to 80.

System Parameter Reference

Command files may also contain system parameters, which may be written or updated by SMFFTI to maintain certain status values between executions of SMFFTI. Such parameters are not intended to be amended by the user. System parameters are indicated as such with the prefix SYS_.

+SYS_RCRHistoryCount

> Used by SMFFTI to track the *Random Chord Replacement* (RCR) history lines that appear (as comment lines) underneath the chord name list in the chord progression data section. It refers to the number of history lines that RCR will check in order to avoid choosing chords that have already been previously used. This parameter is updated whenever RCR is invoked.

Full Example Command File

It may be useful to include all the parameters in your input file, even if you don't enable them. It helps to see what options are available. Comments are also encouraged to annotate your chord progression. Comment blocks - using `(#` and `#)` - are handy for temporarily disabling chord sections.

Here's a complete sample file:

```
# SMFFTI Input File: mymidi.txt

+TrackName=My Test MIDI Sequence

+BassNote=1
+RootNoteOnly=0

+Velocity=70
+RandVelVariation=25

+OctaveRegister=3
+TransposeThreshold=11

+RandNoteStartOffset=2
+RandNoteEndOffset=2
+RandNoteOffsetTrim=1

+NoteStagger=0

+Arpeggiator=0
+ArpTime=8
+ArpGatePercent=50
+ArpOctaveSteps=0

#+AutoMelody=1
#+AutoRhythmNoteLenBias=0, 0, 8, 16, 16, 4
#+AutoRhythmGapLenBias=0, 0, 0, 4, 8, 1
#+AutoRhythmConsecutiveNoteChancePercentage=25

+FunkStrum=0
+FunkStrumUpStrokeAttenuation=0.5
+FunkStrumVelDeclineIncrement=8

# Parameters relating to Auto-Chords (-ac) feature.
#
+AutoChordsNumBars=4;
+AutoChordsMinorChordBias=22, 42, 32
+AutoChordsMajorChordBias=22, 42, 32
+AutoChordsShortNoteBiasPercent=35
#
# Chance of Chord Type Variations (CTV) for Major chords.
# Note that you can specify a chance of suspended chords
# replacing a major chord.
#
+AutoChords_CTV_maj     = 1000
+AutoChords_CTV_7       = 100
```

```
+AutoChords_CTV_maj7    = 10
+AutoChords_CTV_9       = 10
+AutoChords_CTV_maj9    = 10
+AutoChords_CTV_add9    = 80
+AutoChords_CTV_sus2    = 15
+AutoChords_CTV_7sus2   = 15
+AutoChords_CTV_sus4    = 10
+AutoChords_CTV_7sus4   = 10
#
# Chance of Chord Type Variations (CTV) for Minor chords
+AutoChords_CTV_min     = 1000
+AutoChords_CTV_m7      = 100
+AutoChords_CTV_m9      = 10
+AutoChords_CTV_madd9   = 10
#
# Chance of Chord Type Variations (CTV) for Diminished chords.
# NB. Zero chance of plain old dim, theyt generally rubbish;
# instead, equal chance of Diminihed 7th or Half-diminished.
+AutoChords_CTV_dim     = 0
+AutoChords_CTV_dim7    = 1
+AutoChords_CTV_m7b5    = 1

+WriteOldRuler = 0

#+RandomChordReplacementKey = Gm
#+ModalInterchangeChancePercentage=10


$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############################### +###############################
Bb, F

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############################### +###############################
Dm, C

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############################### +###############################
G, Bb

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############################### +###############################
F, D
```