SMFFTI User Manual

Applies to v0.53 - April 1, 2025

Table of Contents

1. Introduction....4
2. Basic Usage....5
3. The SMFFTI Workflow Principle....8
4. Automatically-Generated Chord Progressions....9
    4.1 Modal Interchange....10
    4.2 Auto-Chords Customization....10
        4.2.1 Major/Minor/Diminished Chord Bias....10
        4.2.2 Chord Type Variations....11
        4.2.3 Note Length....13
5. The AutoFill Function....15
6. Random Chord Position Offset....16
7. Random Chord Replacement....18
    7.1 Using Random Chord Replacement and Random Chord Position Offset Together....19
8. MIDI-to-SMFFTI Conversion....20
9. Automatically-Generated Rhythm Patterns....22
    9.1 Note Length Bias....22
    9.2 Gap Length Bias....23
    9.3 Consecutive Note Chance....23
10. Automatically-Generated Melodies....24
    10.1 Melody Line Description....25
11. Ostinato Modes....27
    11.1 Ostinato Rhythm....27
    11.2 Ostinato Melody....28
    11.3 Controlling The Length of Ostinato Notes....29
12. Random Funk-Guitar Grooves....30
13. Creating Arpeggios....32
14. Percussion Patterns....33
15. Command File Modification Commands....34
    15.1 Set Parameter Command....34
    15.2 Disable/Enable Parameter Command....34
    15.3 Delete Parameter Command....35
    15.4 Enable Chords for Random Chord Replacement....35
    15.5 Write Protect Command....35
    15.6 Lock Command....35
    15.7 Unlock Command....36
16. Command Parameter Precedence....37
17. SMFFTI Wizard....38
    17.1 Create SMFFTI Command File....38
    17.2 Create Complete Set Of Command Files....39
18. SMFFTI Workflow Discussion....40
19. Summary Of Commands....44
20. SmfftiWin....45
    20.1 General Commands....47
        20.1.1 Quit (q)....47

20.1.2 Load Command File (cf)....47
20.1.3 Initialize (init)....47
20.1.4 Copy To File (copy)....47
20.1.5 Write Protect (wp)....47
20.1.6 Lock (lock)....47
20.1.7 Unlock (unlock)....48
20.1.8 Execute Script File (@)....48
20.1.9 Undo Command (undo)....48
20.1.10 Redo Command (redo)....48
20.1.11 SplashScreen Command (splash)....49
20.2 Chord Progression Commands....49
20.2.1 Display Chord Progression Page (cp)....49
20.2.2 Chord Add/Change (c)....49
20.2.3 Delete Chord (dc)....50
20.2.4 Add Measure (addm)....50
20.2.5 Insert Measure (insm)....50
20.2.6 Delete Measure (delm)....50
20.2.7 Duplicate Measure (dupm)....51
20.2.8 Initialize Measure (im)....51
20.2.9 Set Gaps Between Chords (gaps)....51
20.2.10 Enable Chords For RCR (ecfr)....51
20.3 Parameter Commands....51
20.3.1 Show Parameter Page (sp)....51
20.3.2 Change Parameter (p)....51
20.3.3 Reset Parameter (rp)....52
20.4 MIDI Operations....53
20.4.1 Create MIDI File (midi)....53
20.4.2 Create MIDI File With Root Notes Only (midiro)....53
20.4.3 Create MIDI File With Root And 5th Notes Only (midir5)....53
20.4.4 Auto-Chords (ac)....53
20.4.5 Auto-Rhythm (ar)....54
20.4.6 Auto-Melody (am)....54
20.4.7 Ostinato Rhythm (or)....54
20.4.8 Ostinato Melody (om)....54
20.4.9 Auto-Rhythm Bassline (arb)....55
20.4.10 Auto-Melody Bassline (amb)....55
20.4.11 Ostinato Rhythm Bassline (orb)....55
20.4.12 Ostinato Melody Bassline (omb)....55
20.4.13 Random Funk-Guitar Groove (rfgg)....55
20.4.14 Percussion Patterns (pp and ppm)....55
20.4.15 More Percussion Patterns (mpc, dfc)....56
20.4.16 MIDI-To-SMFFTI Conversion (m2s)....57
20.5 Example Script File....58
21. File Parameter Reference....59
21.1 +Arpeggiator....59
21.2 +ArpGatePercent....59
21.3 +ArpOctaveSteps....59
21.4 +ArpTime....60
21.5 +AutoChords7thChordsOnly....60
21.6 +AutoChordsAllowedChordTypes....60
21.7 +AutoChordsChunkSize....61
21.8 +AutoChordsConvertMajorToMinor....61

21.9 +AutoChords_CTV_<type>....61
21.10 +AutoChordsMajorChordBias....62
21.11 +AutoChordsMinorChordBias....62
21.12 +AutoChordsNoteLenBias....62
21.13 +AutoChordsNumBars....63
21.14 +AutoChordsWholeNotesOnly....63
21.15 +AutoFill....63
21.16 +AutoRhythmConsecutiveNoteChancePercentage....63
21.17 +AutoRhythmGapLenBias....63
21.18 +AutoRhythmNoteLenBias....63
21.19 +BassNote....64
21.20 +FunkStrum....64
21.21 +FunkStrumUpStrokeAttenuation....64
21.22 +FunkStrumVelDeclineIncrement....64
21.23 +ModalInterchangeChancePercentage....65
21.24 +NoteStagger....65
21.25 +OctaveRegister....65
21.26 +OstinatoGapLenBias....66
21.27 +OstinatoLenBars....66
21.28 +OstinatoNoteLenBias....66
21.29 +RandNoteEndOffset....66
21.30 +RandNoteOffsetTrim....66
21.31 +RandNoteStartOffset....67
21.32 +RandVelVariation....67
21.33 +RCPO_ChancePercentage....67
21.34 +RCPO_MaxEighths....67
21.35 +RCREnabled....67
21.36 +RCRKey....67
21.37 +RFGGNoteLenBias....67
21.38 +RootAnd5thOnly....68
21.39 +TrackName....68
21.40 +TransposeThreshold....68
21.41 +Velocity....68
21.42 +WriteProtected....68
22. System Parameter Reference....69
22.1 +SYS_RCRHistoryCount....69
23. Full Example Command File....70

1. 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*.

*Development of SMFFTI was done in conjunction with Ableton Live. The created MIDI should work with other DAWs, but cannot be guaranteed.

Editing notes in the Ableton Live piano roll can be 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 text file containing the chord progression data and parameters can be described as a SMFFTI Command File.

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

SMFFTI 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.

Be aware that SMFFTI is a *Windows*-only console application, and works only for 4/4 time.

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 the developer(s) are not responsible 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.

2. 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 a chord progression and various parameters, ie. a SMFFTI command file. You can use any text editor you like to create and amend such 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
```

In this example, we have a two-bar progression that plays four chords: F, Am, G and C. Each chord is half a bar (minim) long.

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.

The second line is the *note-position* line and indicates where the notes play in the bar(s), using sequences of characters consisting of a single plus sign followed by zero 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. 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 note-position line.

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

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.

> 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*.

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
```

(NB. In the above two examples, notice, at the last quarter of the first bar, the series of hash-signs that do *not* begin with a plus-sign. This simply means that the previous chord will be played again, in this case C.)

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 23 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`)
- Augmented (`C+`)
- Augmented 7th (`C+7`)
- Augmented Major 7th (`C+maj7`)
- Augmented 9th (`C+9`)
- Augmented Major 9th (`C+maj9`)

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 can have up to four sets, or measures, of three lines (1. ruler; 2. note positions; and 3. chord names) in a SMFFTI command file. 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 can save time that would be spent laboriously editing the MIDI directly inside Ableton Live, as well as offer the ability to 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.

3. The SMFFTI Workflow Principle

Using SMFFTI to create a MIDI file requires, as you have seen, a command file which comprises a chord sequence and various control parameters. However, that is just the start. A complete set of MIDI clips, consisting of harmonies, melodies and basslines, can be developed, using the main command file as the primary source.

What this means, in practice, is that the main command file is automatically cloned, when required, to create temporary, modified versions for the SMFFTI operations that generate the MIDI clips containing harmonies, melodies, etc.

Thus the main command file lies at the heart of the development of a given musical segment. This being the case, the SMFFTI Workflow Principle is this:

1. Tinker with your SMFFTI command file until you decide on a chord progression that you are totally happy with.

2. Lock the chord progression (and control parameters). This is a step that write-protects the command file, ensuring that the chord progression cannot be altered, accidentally or otherwise.

3. Use this now read-only command file to generate the main MIDI clip and its harmony/melody derivatives.

It should be noted that, until the command file is locked, certain SMFFTI operations (eg. *Auto-Rhythm*, *Auto-Melody*) cannot be executed. The idea behind this is to prevent your derivative MIDI clips being rendered invalid as a result of changing the chords (or their note positions) inadvertently or otherwise.

The *Lock* (and *Unlock*) commands is described in the *Command File Modification Commands* section.

A more detailed look at workflow is contained in the *SMFFTI Workflow Discussion* section.

4. Automatically-Generated Chord Progressions

If you are feeling rather uninspired and struggling to begin a chord progression, there is the *Auto-Chords* feature. Based on a few parameters specified in the input 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 typically 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.

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

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

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 a rhythm pattern of (by default) varying note lengths.

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

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

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

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

Note that it is almost certain that the chord progression and rhythm pattern will require amendment. *Auto-Chords* 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 *Auto-Chords* multiple times to discover a progression/rhythm that you can work with.

> **Tip:** One method of changing the chords in the progression that *Auto-Chords* generated is the *Random Chord Replacement* (RCR) mechanism. To easily facilitate this, *Auto-Chords* automatically prefixes all chord names with a question-mark("?") to make them candidates for change using RCR. (You can quickly remove these "?" prefixes using the *Enable Chords For Random Chord Replacement* command (`-ecfr` switch).)

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

```
+AutoChordsNumBars = 8
```

4.1 Modal Interchange

Modal interchange defined here means chords from the corresponding (or 'opposite') minor/major key. For example, if your main key is G Minor, then *Auto-Chords* may use chords from G Major. Similarly, if your main key is Eb Major, then chords from Eb Minor may be used. 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 you want the possibility of modal interchange, you specify parameter `+ModalInterchangeChancePercentage`, giving it a value between 0 and 100. 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.`

If any modal interchange chords are included in the generated chord progression a comment line is inserted in a comment block in the output file, eg.:

```
(#------------------------------------------------------------
20240403 14:36:13 Modal Interchange Chords (2):
Emadd9, D7
------------------------------------------------------------#)
```

NB. Modal interchange can also be applied to the *Random Chord Replacement* operation, using parameter `+ModalInterchangeChancePercentage`.

4.2 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. This section describes 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.

These are all discussed below.

4.2.1 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.

4.2.2 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 22 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
- Augmented
- Augmented 7th
- Augmented Major 7th
- Augmented 9th
- Augmented Major 9th

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).

These chord variations may be used in the generated chord progression, instead of the basic triad, according to parameters described below.

Since augmented chords are quite similar to major chords - they differ only by having the fifth a semitone higher - they are considered major chord variations. Similarly, suspended chords, which are not associated *per se* with major or minor chords, are nevertheless considered variations of major chords.

With regard to diminished chords, there are two variations: Diminished 7th and Half-diminished.

Thus, SMFFTI provides 22 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 (plus suspended and augmented)
+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
+AutoChords_CTV_aug       =   10
+AutoChords_CTV_aug7      =    0
+AutoChords_CTV_aug_maj7  =    0
+AutoChords_CTV_aug9      =    0
+AutoChords_CTV_aug_maj9  =    0

# 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.

A minimum requirement is that both `+AutoChords_CTV_maj` and `+AutoChords_CTV_min` must be non-zero.

Additional Parameters For Controlling Chord Type Variations

- Quick Chord Type Variation Subsets. You can specify a subset of the 22 chord type variations using the `+AutoChordsAllowedChordTypes` parameter. For example:

  ```
  +AutoChordsAllowedChordTypes = maj, min, add9, m7
  ```

  This will result in *Auto-Chords* in choosing only *Major*, *Minor*, *Add 9th* or *Minor 7th* chords.

- Seventh Chords Only. To cause *Auto-Chords* to use only certain 7th chords, use parameter `+AutoChords7thChordsOnly`. For example:

  ```
  +AutoChords7thChordsOnly = 1
  ```

  This is just an arbitrary parameter which is intended to more easily create jazz/funk-style chord progressions (which comes in handy if you use the *Random Funk-Guitar Groove* operation). To include other 7th chord types, use `+AutoChordsAllowedChordTypes`.

- Convert Major Chords To Minor. To direct *Auto-Chords* to convert any *Major* chords selected to *Minor*, use the `+AutoChordsConvertMajorToMinor` parameter. For example:

  ```
  +AutoChordsConvertMajorToMinor = 1
  ```

  Again, this is a convenience parameter which facilitates musical experimentation and variety.

For more information about these parameters, consult the *Command File Parameter Reference* section.

4.2.3 Note Length

This is all about the *rhythm pattern* of the chord progression, which *Auto-Chords* outputs by

generating a series of notes of (by default) varying length.

Note length can randomly be any of a whole-, half-, quarter-, eighth- or sixteenth-note, and refers, in actual fact, to *the distance between the start of one note and the one following it*, because notes are always truncated to leave a 1/16th-note gap between the *end* of one note and the *start* of the next (except when there are occurrences of 1/16th-notes, in which case the subsequent gap will be 1/32nd).

The chances of the different note lengths appearing in the rhythm pattern is controlled by parameter `+AutoChordsNoteLenBias`, the supplied value to which must be a comma-separated list of five numbers, eg.:

```
+AutoChordsNoteLenBias = 1, 2, 6, 2, 0
```

Each of the numbers indicates, relative to the others, the chances of a given note length occurring. Reading from left to right, the numbers refer to whole-notes, half-notes, quarter-notes, eighth-notes and sixteenth-notes. Thus, in the above example, the chances of quarter-notes (6) occurring is three times more likely than half- or eighth-notes (both 2), and six times more likely than whole-notes (1); there is also no possibility of sixteenth-notes.

It will be seen, therefore, that you can control note lengths quite considerably using this weighting mechanism. The maximum possible value of each of the numbers is 100.

If, for example, you only wanted half- and quarter-notes, you could specify:

```
+AutoChordsNoteLenBias = 0, 1, 1, 0, 0
```

To create a sequence that is exactly one-chord-per-bar, you set parameter `+AutoChordsNoteLenBias` accordingly (ie. `"1, 0, 0, 0, 0"`). Alternatively, you can achieve the same thing with another convenient parameter:

```
+AutoChordsWholeNotesOnly = 1
```

When set to 1, `+AutoChordsWholeNotesOnly` overrides the values of `+AutoChordsNoteLenBias`.

If not specified, the default value of `+AutoChordsNoteLenBias` is:

```
+AutoChordsNoteLenBias = 4, 1, 1, 0, 0
```

By default, chord notes will not traverse bar boundaries. That is, each bar will begin with a new note. A measure (two or four bars) is thus considered as a series of one-bar "chunks". However you can change the chunk size. For example, you can have two-bar chunks, or the whole measure (eg. four bars) can be a chunk. It simply means that notes will be contained inside one chunk and not cross chunk boundaries. A chunk size that is larger than one bar will permit notes to cross bar boundaries within a chunk. Measure chunk size is set using using parameter `+AutoChordsChunkSize`. Valid values are 2, 4, 8, 16, 32, 64 and 128. These numbers are the possible chunk sizes expressed as 1/32nds. So, for example, 32 specifies a chunk length of one bar.

5. The AutoFill Function

AutoFill is SMFFTI's ability to automatically remove the gaps between chords in a measure. This applies when the MIDI file is being created. By default, in the note-position lines, you specify the precise length of chords (in 1/32nds) using the "+########" notation as described in the *Basic Usage* section. Thus you manually set the size of gaps between chords.

However, depending on what you are trying to achieve, musically-speaking, you may prefer to have no gaps between chords, effectively making the chords contiguous. *AutoFill* will take care of this for you, such that you can dispense with hash-signs in the note-position lines and use only plus-signs to indicate the start position of chords, eg.:

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

This saves somewhat tedious typing if you are experimenting with moving your chord positions around.

The parameter to control *AutoFill* is, naturally enough, `+AutoFill`. Its value is a number of 1/32nds. It can be set as a number in the range zero to 127, or "off". For example:

```
+AutoFill = 4
```

This means a gap of an 1/8th note (4 x 1/32nds) between chords. Zero means no gap at all. "off" means don't do anything. If the specified gap size is larger than can be accommodated because of the close proximity of two chords, the gap will be reduced and the length of the first of the two chords will be only 1/32nd.

After creation of the MIDI file, the effect of *AutoFill* is reflected in the note position lines in the command file.

Autofill works *only* when SMFFTI is creating a MIDI file. It is ignored for all other operations, eg. *Auto-Rhythm*.

6. Random Chord Position Offset

Random Chord Position Offset (RCPO) is a method of adding rhythmic variation to a chord progression. This is simply a randomized shifting of chord start positions from their original notional positions. For example, if you have four chords across a four-bar measure, rather than each chord starting at the beginning of each bar, RCPO can change these start positions. Musically, this offsetting of chord positions can be quite effective.

Whereas *Auto-Rhythm* chops up the original chord note positions by inserting gaps to provide a localized rhythmic pattern for each chord, RCPO will simply shift the chord start position. *Auto-Rhythm* never changes the start position or length of a chord; RCPO, on the other hand, moves the chord start position (either left or right) and, consequently, changes the length of the chord.

Auto-Rhythm does not change the original SMFFTI command file, but creates a modified version that contains the rhythmic chord notes. RCPO, however, directly modifies the input SMFFTI command file.

Consider this simple example snippet of a chord progression from a SMFFTI command file:

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

RCPO can randomly change this to look like:

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

or, possibly:

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

Note the shift in the start position of the second chord, either left of right, from its original position. Note there is always a 1/32nd note gap between chords (except when *AutoFill* is also active, in which case it is *AutoFill* which determines the length of the gap).

There are two parameters to control the operation of RCPO: `+RCPO_ChancePercentage` and `+RCPO_MaxEighths`. For example:

```
+RCPO_ChancePercentage = 50
+RCPO_MaxEighths = 2
```

`+RCPO_ChancePercentage` dictates the likelihood of chords being offset. A value of zero (the default if not specified) means no offset will occur. A value of 100 means offset will occur for every chord (except the first chord - see below) in the progression.

`+RCPO_MaxEighths` controls the amount of offset. Valid range: 1 - 4. Default is 1. `+RCPO_MaxEighths` will do nothing if `+RCPO_ChancePercentage` is set to zero. Each offset can be *n* 1/8th notes, where *n* is in the range 1 to `+RCPO_MaxEighths`. In the above example, each offset could be an 1/8th or 1/4 note. The maximum possible offset is a half-note (by setting `+RCPO_MaxEighths` to 4).

The way RCPO works is that, when it parses the SMFFTI command file, it considers only the *number* of chords in the progression, and completely disregards the original note position line. It does this because it initially considers all chords in the progression to have the same length and spreads them, as it were, across the whole measure. So, conceptually, if you had three chords in a two-bar progression, RCPO initially sees it like this, for example:

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

Note that chord positions are quantized to 1/8th notes. From this original perspective RCPO will then randomly apply offsets, eg.:

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

Note that RCPO will always position the first chord at the start.

In averaging out the chord lengths across the measure, RCPO will not operate if it finds that the average length of a chord is less than a 1/4 note. So, in a two-bar measure, you could not have more than eight chords for RCPO to work. If you did, SMFFTI will issue an error message.

If you only have one chord specified, SMFFTI will set it to span the entire measure (leaving a 1/32nd gap at the end).

Also note that RCPO works *only* when SMFFTI is creating a MIDI file. It is ignored for all other operations, eg. *Auto-Rhythm*.

7. Random Chord Replacement

Random Chord Replacement (RCR) 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 was originally implemented to augment the use of *Auto-Chords* (although it can be used quite independently 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 parameter `+RCREnabled`, which can be set to 0 or 1. `+RCRKey` determines the key from which replacement chords will be selected, eg.:

```
+RCREnabled = 1
+RCRKey = Gm
```

If `+RCRKey` is not specified, the default key is Gm.

With RCR enabled, you now indicate chords in your progression which will be replaced with randomly-chosen chords from the key specified by the `+RCRKey` 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 `+RCRKey` parameter and removing the question marks (you might also tidy up the command file by removing the history of unsuccessful chord progressions.)

If RCR is not active (`+RCREnabled = 0`), 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, ie. `+AutoChordsMinorChordBias`, `+AutoChordsMajorChordBias` and the `+AutoChords_CTV_*` set of parameters.

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.

RCR works *only* when SMFFTI is creating a MIDI file. It is ignored for all other operations, eg. *Auto-Rhythm*.

> **Tip:** You can quickly add or remove the "?" prefix for all chords in the progression using the *Enable Chords For Random Chord Replacement* command.

7.1 Using Random Chord Replacement and Random Chord Position Offset Together

This technique is an alternative to *Auto-Chords*, and is perhaps more controlled, since it does not change the *number* of chords nor introduce additional gaps between the chords.

All you need to do is enable *Random Chord Replacment* (RCR) and *Random Chord Position Offset* (RCPO), eg.:

```
+RCREnabled = 1
+RCRKey = Gm
+RCPO_ChancePercentage = 50
+RCPO_MaxEighths = 2
```

and then prefix all your chords with a question mark, eg.:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############### +############### +############### +###############
?Gm, ?Dm, ?Eb, ?Cm
```

(The example here is four chords over two bars (for clarity on the page) but it possibly works better if you have eight chords over four bars.)

Now just run SMFFTI as many times as needed to arrive at a nice chord progression and rhythm.

8. 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*, as per the *SMFFTI Workflow Principle*.

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 added 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 can be an existing SMFFTI command file, in which case its current chord progression data will be replaced. If the specified output file is text file that is NOT a valid SMFFTI command file, the chord progression data will simply be appended to it. If the specified output file does not exist, a new SMFFTI command file will be created, containing some useful parameters and the chord progression data.

The MIDI file should not be longer than sixteen bars. If it is, only data for the first sixteen bars will be output. If the chord progression is longer than four bars, the output will be formatted in four-bar measures. If any of the chords in the MIDI file extend from the end of one four-bar measure into the next one, that chord will be split into two chords, since SMFFTI requires that all chords must be contained in a single measure of four bars..

If the output file is a SMFFTI command file, *Random Chord Replacement*, *Random Chord Position Offset* and *AutoFill* will all be disabled in that file. This is to prevent the risk of undesirable changes to the chord and/or note-position data in the command file, which would also be reflected in the MIDI file subsequently created from it. It's to protect the MIDI notes that you painstakingly edited inside Ableton Live (or other DAW).

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!

9. 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*.

> **NB.** *Auto-Rhythm* can be invoked only when your SMFFTI command file has been "locked", meaning that the chord progression has been completely decided upon and will no longer change. Use the `-lock` operation to accomplish this.

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, which is essentially similar except that the original note position strings are replaced with ones that represent an enhanced rhythmic pattern. The modified copy file is also condensed to remove unnecessary content, ie. comments and disabled parameters. To invoke *Auto-Rhythm*, execute SMFFTI in this form:

```
SMFFTI.exe -ar mymidi.txt mymidi_ar.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_ar.txt mymidi_ar.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 to bias note length, the gap between notes and the likelihood of consecutive notes. This is describe in the following sections.

9.1 Note Length Bias

This is controlled with parameter `+AutoRhythmNoteLenBias.` It 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 cannot 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 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 the opinion of the SMFFTI development team ;-)

9.2 Gap Length Bias

This is controlled with parameter `+AutoRhythmGapLenBias`. It works 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.

9.3 Consecutive Note Chance

This is controlled with parameter `+AutoRhythmConsecutiveNoteChancePercentage`. It specifies the chances of *consecutive notes*, that is, no gaps between notes (within a bar*). 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 be used for 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 later) where the notes of the generated melody conform exactly to the note lengths of the original chord progression.)

*Consecutive notes can appear, and are likely to, across bars. That is, there may not be a gap between a note at the end of a bar and the note at the beginning of the next bar. This is a limitation of *Consecutive Note Chance*.

10. Automatically-Generated Melodies

The default operation of SMFFTI is to create MIDI files containing chords. But it is also possible to make SMFFTI output *single notes* instead of chords, using the *Auto-Melody* operation. *Auto-Melody* generates randomized sequences of "notes", representing melody lines, and inserts them into a SMFFTI command file. The existence of such melody lines in the SMFFTI command file is detected and acted upon when subsequently creating a MIDI file.

> **NB.** *Auto-Melody* can be invoked only when your SMFFTI command file has been "locked", meaning that the chord progression has been completely decided upon and will no longer change. Use the `-lock` operation to accomplish this.

To invoke *Auto-Melody*, you use the `-am` switch. For example:

```
SMFFTI.exe -am mymidi_ar.txt mymidi_am.txt [-o]
```

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

Auto-Melody works by creating a modified version of your SMFFTI command file, which is essentially the same except that all measures in the chord progression will have a melody line inserted, indicated by the prefix "M:". For example:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+###+###  +#  +#+###  +#    +#  +###  + +#######  + +###  + +#+#
Bb(7), Gm7(8)
M: 3, 5, 3, 1, 3, 5, 3, 9, 5, 3, 3, 5, 9, 5, 7
```

(The modified copy file is also condensed to remove unnecessary content, ie. comments and disabled parameters.)

The sequence of numbers in the "M:" melody line corresponds to the number of notes in the note position line. In this example, there are fifteen notes played (ie. fifteen plus-signs), hence fifteen numbers in the melody line sequence. (The meaning of the numbers in the melody line is described below, in section *Melody Line Description*.)

Typically, *Auto-Melody* works in tandem with *Auto-Rhythm*. In order to create a useful melody line, the note position lines in the input file should, ideally, reflect a rhythmic pattern containing shorter notes, as this example illustrates. To achieve this easily, it is suggested that you invoke *Auto-Rhythm* first, and use its output file as input for *Auto-Melody*.

You must now run SMFFTI again to create a MIDI file that contains the melody line, using this generated command file, eg:

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

It is precisely the presence of "M:" lines that causes the melody line to be output to the final MIDI file, rather than chords.

NB. Rather than use *Auto-Melody*, you can, of course, create these "M:" melody lines manually, if you wish.

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

10.1 Melody Line Description

The numbers in the "M:" melody lines can be 1, 3, 5, 7 or 9. They indicate, notionally, notes from a chord, where:

1 = Root
3 = Major 3rd
5 = Perfect 5th
7 = Major 7th
9 = Major 9th

It should be understood, however, that a melody note sequence is *abstract*, in the sense that it is not directly tied to the underlying chord sequence. Another way of describing these notes would be to call them *placeholder* notes. Only at the point when the MIDI file is created will these notes be *resolved* to correspond directly to the underlying chords. So it can be seen that melody lines, or fragments of melody line, can be ported around to other chord progressions without any changes required to the number sequence, since they will be resolved at MIDI file creation-time.

This table defines how SMFFTI, when creating the MIDI file, resolves the *abstract* notes in the melody sequence, converting them to *real* semitone values, with respect to the underlying chord:

Chord Type	Major 3rd	Perfect 5th	Major 7th	Major 9th
maj	4	7	0	4
7	4	7	10	0
maj7	4	7	11	0
9	4	7	10	14
maj9	**4**	**7**	**11**	**14**
add9	4	7	0	14
m	3	7	0	3
m7	3	7	10	0
m9	3	7	10	14
madd9	3	7	0	14
sus2	2	7	0	2
7sus2	2	7	10	0
sus4	5	7	0	5
7sus4	5	7	10	0
5	7	7	0	7
dim	3	6	0	3
dim7	3	6	9	0
m7b5	3	6	10	0

+	4	8	0	4
+7	4	8	10	0
+maj7	4	8	11	0
+9	4	8	10	14
+maj9	**4**	**8**	**11**	**14**

(Obviously, root notes are never altered, so are not mentioned in this table.)

Astute readers will see that it is only in the case of *Major 9th* and *Augmented Major 9th* chords that all five of the notes - *Root, 3rd, 5th, 7th* and *9th* - retain their intrinsic values, since the *Major 9th* chord contains all those notes. In all other cases, one or more of these notes loses its notional meaning. For example, in the case of minor chords, the *Major 3rd* becomes a *Minor 3rd*. Another example is the diminished chords, where the *Perfect 5th* is turned into a *Flat 5th*.

11. Ostinato Modes

An ostinato is a short musical phrase, so to speak, which is repeated in a track or song. As far as SMFFTI is concerned, the are two elements to an ostinato: the rhythm and the melody. SMFFTI provides the ability to randomly generate both these elements and combine them. The Ostinato Modes are somewhat similar to *Auto-Rhythm* and *Auto-Melody* in their operation, except they create only short phrases of rhythm/melody - one or two bars - that are repeated throughout the chord progression.

Ostinato Rhythm will generate a short randomized rhythmic pattern that will be applied to the entire chord progression. The result can be used as-is, or it can be used as input for *Ostinato Melody*, which generates a randomized melody line to match the short rhythmic pattern, and which is also applied to the entire chord progression.

The ostinato modes are only suitable when the chords in a progression are quantized to whole bars. So don't try to apply them to progressions which have the chord positions offset, eg. as a result of the *Random Chord Position Offset* operation.

> **NB.** *Ostinato Rhythm/Melody* can be invoked only when your SMFFTI command file has been "locked", meaning that the chord progression has been completely decided upon and will no longer change. Use the `-lock` operation to accomplish this.

11.1 Ostinato Rhythm

To create an ostinato rhythm you use the `-or` switch, eg.:

```
SMFFTI.exe -or chords.txt chords_or.txt [-o]
```

The `-o` switch is optional but is required if you want to overwrite an already-existing output file.

This command will cause SMFFTI to create a modified copy of the input file (`chords.txt`), calling it `chords_or.txt`, with the note position lines replaced with an ostinato rhythm pattern. The modified copy file is also condensed to remove unnecessary content, ie. comments and disabled parameters. For example, if the input file's chord progression is this:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+                             +
Bb, Gm7
```

The output file's chord progression data might contain something like this:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+ ##    ##  #   ##  ####  ####  + ##    ##  #   ##  ####  ####
Bb, Gm7
```

The hash-signs in the original note position lines are ignored, it only considers the position of the plus-sign. So, if you wish, you can just place single plus-signs at whole- or half-bar positions, depending on the number of chords in the measure.

In this example, the ostinato is one bar long. Note that it repeats for the second bar. If you had four bars in the measure, or multiple measures (maximum of four), the ostinato would be repeated

throughout.

Note also, in this example, the first instance of the ostinato will play chord Bb, and the second instance will play chord Gm7. The chords in the chord list are thus spread equally across the measure, so to speak. If we thus consider that the chords have a *notional length* equal to `number of bars / number of chords` (in this case one bar), the ostinato rhythm length can be longer or shorter than that notional length, depending on the number of chords you specify.

Ostinato Rhythm has some constraints regarding the chord progression data lines:

- All measures in original chord progression data must (a) be one, two or four bars long, (ie. *not* three), (b) be the same length and (c) all have the same number of chords.
- The minimum chord length is half a bar so, for example, in four-bar measures, you cannot specify more than eight chords.
- The number of chords in a measure must be one, two, four or eight (subject to the number of bars in the measure).
- The note start positions (ie. the "+" signs), must be equally spread across the measure, indicating that the notional length of all the chords is the same. They must thus be quantized at bar or half-bar positions, depending on the number of chords in the measure.

You can use parameter `+OstinatoLenBars` to specify how long the ostinato rhythm pattern will be: Either one or two bars, eg.:

```
+OstinatoLenBars = 2
```

If `+OstinatoLenBars` is not specified the length of the ostinato rhythm will be one bar.

11.2 Ostinato Melody

The *Ostinato Melody* operation creates a short, randomized melody line that is repeated throughout the chord progression. The input file is expected to contain a repeating rhythmic pattern that would have been, typically, created using the *Ostinato Rhythm* operation.

To create an ostinato melody use the `-om` switch, eg.:

```
SMFFTI.exe -om chords_or.txt chords_om.txt [-o]
```

The `-o` switch is optional but is required if you want to overwrite an already-existing output file.

This command will cause SMFFTI to create a modified copy of the input file (`chords_or.txt`), calling it `chords_om.txt`, which is almost identical except that all measures in the chord progression will have a melody line inserted, indicated by the prefix "M:". For example:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+ ##    ##  #   ##  ####  ######+ ##    ##  #   ##  ####  ######
Bb, Gm7
M: 5, 9, 7, 1, 1, 1, 1, 5, 9, 7, 1, 1, 1, 1
```

The modified copy file is also condensed to remove unnecessary content, ie. comments and disabled parameters.

The presence of "M:" lines causes this melody line to be output to the final MIDI file, rather than

chords. The format and meaning of the "M:" lines is described in section *Automatically-Generated Melodies*.

11.3 Controlling The Length of Ostinato Notes

The possible note lengths in a SMFFTI ostinato are 1/4 notes, 8ths, 16ths and 32nds. To control the chances of these note lengths occurring, we use parameter `+OstinatoNoteLenBias`. For example:

```
+OstinatoNoteLenBias = 10, 40, 30
```

The three numbers are percentage values, and specify the chance of the ostinato note length being a 1/4 note, an 8th note or a 16th note. So, in this example, there is a 10% chance of notes being a 1/4 note, a 40% chance of notes being an 8th, and a 30% chance of notes being a 16th. If the total of the three numbers is less than 100, the implied remaining percentage specifies the chance of 32nd notes. In this example, the total percentage is 80%, so there is a 20% chance of 32nd notes.

Similarly, the gaps between notes can also be 1/4 notes, 8ths, 16ths or 32nds. We can control the gaps between the notes using parameter `+OstinatoGapLenBias`. For example:

```
+OstinatoGapLenBias = 10, 20, 30
```

If you do not specify these parameters, the default values are:

```
+OstinatoNoteLenBias = 10, 30, 30
+OstinatoGapLenBias = 10, 30, 30
```

You should be aware that a generated ostinato pattern may not always *exactly* conform to the values specifed by the `+OstinatoNoteLenBias` and `+OstinatoGapLenBias` parameters, for various reasons. These parameters are, as their name implies, more of a *biasing* control; ostinato patterns may contain what might be termed *artifacts* - unexpected note/gap lengths. A mitigating argument here is that the generation of ostinatos is random and ephemeral: They are expected to be, for want of a better term, organic! ;-)

12. Random Funk-Guitar Grooves

A slightly more specialized type of rhythmic pattern, tailored for the rapid strumming style of funk-guitar, is outlined here.

> NB. The MIDI files that contain rhythm patterns generated by *Random Funk-Guitar Groove* are intended to be used with a software instrument that emulates an electric guitar. For example, *Ableton Live* has some guitar presets that uses its *Tension* instrument. With the right effects, a passable funk-guitar sound can be created, albeit not as good as audio samples from an actual guitar.

The *Random Funk-Guitar Groove* (RFGG) operation operates in a similar way to *Auto-Rhythm*, in that it creates fragmented rhythm patterns, but RFGG uses a different algorithm to create the patterns.

RFGG is invoked using the `-rfgg` switch. For example:

```
SMFFTI.exe -rfgg chords.txt rfgg.txt [-o]
```

This command essentially copies the content of the input file (`chords.txt`) to the output file (`rfgg.txt`), except that the note position lines of the chord progression in the output file will contain a different note pattern; a complex rhythmic one. The output file is also condensed to remove unnecessary content, ie. comments and disabled parameters.

(Use the `-o` option to automatically overwrite an existing version of the output file.)

So, for example, if `chords.txt` contained this:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############################## +################################
Gm7, Fm7
```

then output file `chords_rfgg.txt` might contain this:

```
$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+###  +#+#+#  +#+#  +#    +#  +#    +#  +#####+#+###  +#+#+#
Gm7(9), Fm7(7)
```

Additionally, the RFGG operation will also set a fixed value for these parameters in the output file (eg. `rfgg.txt`):

```
+FunkStrum = 1
+FunkStrumUpStrokeAttenuation = 0.7
+FunkStrumVelDeclineIncrement = 5
```

(thus overriding the command file values for these parameters.) These parameters are important when generating the MIDI file: They arrange the MIDI notes to give them an articulation that attempts to emulate the rapid strumming of an electric guitar. See the *Command File Parameter Reference* section for more information about these parameters.

The rhythm patterns can consist of notes which can be a length of 1/16th, 1/8th, 3/16ths or a 1/4-note; and gaps which can be a 1/16th or 1/8th. The chances of the different note lengths and gaps occurring is dictated by parameter `+RFGGNoteLenBias`, which takes a comma-separated list of five

numbers, eg.:

```
+RFGGNoteLenBias = 4, 4, 4, 1, 1
```

The numbers specified for `+RFGGNoteLenBias` define a relative chance bias for the gaps and different possible note lengths. Consult the *Command File Parameter Reference* section for more information about `+RFGGNoteLenBias`.

Note that a generated pattern will never contain:

- More than three consecutive 1/16th notes.
- Gaps larger than an 1/8th note.
- Two consecutive 3/16th notes.
- Two consecutive 1/4 notes.
- More than two consecutive 1/8th notes.

13. Creating Arpeggios

When generating the MIDI file from the SMFFTI command file, you can output an arpeggio, rather than chords. This is achieved by simply setting the `+Arpeggiator` parameter to a value in the range 1 to 13, and this specifies the type of arpeggiation.

You can use three other parameters also, to vary the arpeggio's note length (`+ArpGatePercent`), its beat rate (`+ArpTime`), and the number of times it is repeated using octave transpositon (`+ArpOctaveSteps`).

Please refer to the *Command File Parameter Reference* section for more information setting these parameters.

The default behaviour is for the arpeggio pattern to last for the length of the chord specified in the note position lines in the chord progression. But if you want the pattern to continue across any gaps that may be present between the chords, use the `+AutoFill` parameter.

When `+Arpeggiator` is set *Random Chord Replacement* and *Random Chord Position Offset* are disabled.

14. Percussion Patterns

A SMFFTI "Percussion Pattern" is a one-bar MIDI clip that contains randomly-generated "notes", and which is intended to be used with a percussion instrument that contains a group of percussive sounds (for example, an Ableton Live Drum Rack). The purpose of Percussion Patterns are to augment the core drum pattern of your track.

To create a Percussion Pattern MIDI clip, use the `-pp` command:

```
SMFFTI.exe -pp <note range> <base note> <MIDI output file> [-o]
```

where:

`<note range>` is the number of percussion sounds to choose notes from.

`<base note>` is the lowest note that can be used.

`<MIDI output file>` is the name of the output file.

`-o` is required if you wish to overwrite an already existing MIDI file.

For example:

```
SMFFTI.exe -pp 16 C1 pp.mid -o
```

This tells SMFFTI to use a possible 16 notes, using notes in the range C1 - D#2. There will only ever be a single note playing at a time, ie. no "chords".

The maximum number of notes that can be specified is 84 (7 octaves).

The MIDI file will contain between 4 and 17 notes, depending on the specified note range - the larger the note range, the likliehood of more notes. All notes are placed on 16th notes, except for occasional instances of a triplet note, which is placed on a 32nd note, in between two adjacent 16th notes. There will only ever be a single triplet in a MIDI file.

It is important to note that, because Percussion Patterns are essentially intended to create "top-loops", *no notes will be placed on the "beat" notes*; that is, there won't ever be notes appearing at the start of each quarter note. This is to avoid clashing with kick drums existing elsewhere in the track. However, if you explicitly wish to have such beat notes, there is "Metronome Mode", which is a variation of the -pp command: -ppm. For example:

```
SMFFTI.exe -ppm 16 C1 pp.mid -o
```

Metronome Mode will *always* place a note at the start of each quarter; and that note will *always* be the note indicated by `<base note>` (eg. C1 in this example). Typically, therefore, your percussion instrument (or drum rack) would implement a kick drum in the first location (ie. lowest note).

> NB. There are two other commands for creating percussive patterns (`mpc`, `dfc`) which can be used only in *SmfftiWin*. Refer to the *SmfftiWin* section, and see subsection *More Percussion Patterns*.

15. Command File Modification Commands

The most basic way of modifying a command file is by directly editing the file. But it is also possible to use SMFFTI commands to perform certain modifications. The purpose of these commands is to facilitate the use of *Windows* command files (.cmd). This provides the ability to 'program' SMFFTI to execute a sequence of commands without manual user intervention. These modification commands are described below.

15.1 Set Parameter Command

The *Set Parameter* operation is invoked using the `-sp` switch, followed by a valid parameter definition. For example:

```
SMFFTI.exe -sp "+Velocity = 80" chords.txt
```

This command tells SMFFTI to update the `chords.txt` file with the `+Velocity` parameter, setting it to 80. (You *must* enclose the parameter definition in double quotes.)

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.

You can use the *Set Parameter* command inside a *Windows* command file (.cmd). For example, a command file might contain the following:

```
copy /y chords.txt bass.txt
smffti.exe -sp "+OctaveRegister = 2" bass.txt
smffti.exe -sp "+RootAnd5thOnly = ro" bass.txt
smffti.exe bass.txt bass.mid -o
```

This command file copies `chords.txt` to `bass.txt`, and then sets the values of `+OctaveRegister` and `+RootAnd5thOnly` in file `bass.txt`. This allows the remaining commands to proceed without you needing to manually edit `bass.txt` first. This example illustrates how we are able to use one SMFFTI command file (`chords.txt`) to create a second, modified command file (`bass.txt`).

15.2 Disable/Enable Parameter Command

The *Disable Parameter* command will deactivate a parameter by prefixing it with a hash-sign (#), thus turning it into a comment. You execute the *Disable Parameter* command by using the `-disable` switch. For example:

```
SMFFTI.exe -disable RandVelVariation chords.txt
```

This command tells SMFFTI to disable parameter `+RandVelVariation` (if it is present) in file `chords.txt`. (You do not need to specify a plus-sign (+) in the parameter name.)

Conversely, the *Enable Parameter* will do the opposite: It will remove hash-signs from the beginning of a comment line which otherwise contains a valid parameter, to make it active. You execute the *Enable Parameter* command by using the `-enable` switch. For example:

```
SMFFTI.exe -enable RandVelVariation chords.txt
```

15.3 Delete Parameter Command

The Delete Parameter command will remove a parameter from the specified SMFFTI command file. For example:

```
SMFFTI.exe -dp Velocity chords.txt
```

This command tells SMFFTI to remove the `+Velocity` parameter from `chords.txt`.

Parameters are removed even if they are disabled, ie. commented-out.

15.4 Enable Chords for Random Chord Replacement

The *Enable Chords For Random Chord Replacement* command is a convenient way of adding or removing the question-mark ("?") prefix to all of the chord names in the chord progression.

Adding the "?" prefix prepares the chords in anticipation of invoking the *Random Chord Replacement* (RCR) operation. Note that the *Enable Chords For Random Chord Replacement* command does not activate RCR - that is done as a separate step (by setting `+RCREnabled = 1`).

The format of the *Enable Chords For Random Chord Replacement* command is:

```
SMFFTI.exe -ecfr [ y | n ] <file>
```

where `<file>` is the name of a SMFFTI command file. You can specify `y` to add the "?" prefix to all chords, or `n` to remove it from all chords. For example:

```
SMFFTI.exe -ecfr y chords.txt
```

This command will prefix all chord names in the chord progression of the file `chords.txt` with "?".

15.5 Write Protect Command

The *Write Protect* command will add (or remove) the `+WriteProtected` parameter. The presence of the `+WriteProtected` parameter in a SMFFTI command file will prevent that file from being modified by a SMFFTI operation.

The format of the *Write Protect* command is:

```
SMFFTI.exe -wp [ 0 | 1 | 2 ] <file>
```

where `<file>` is the name of a SMFFTI command file.

Specifying 0 will remove the `+WriteProtect` parameter. A value of 1 will set `+WriteProtect` = 1, and this means that nothing in the file can be modified by a SMFFTI operation. Whereas, a value of 2 will set `+WriteProtect = 2`, which means that only the chord progression data is protected; parameter values can still be amended.

15.6 Lock Command

Use the *Lock* command when you have completed development of your main command file and have decided on the chord sequence. Example:

```
SMFFTI.exe -lock chords.txt
```

This command will set `chords.txt` to read-only mode (by setting parameter `+WriteProtected = 1`).

While this is very similar to the *Write Protect* command, it differs in that is also disables *Random Chord Replacement* (RCR) and *Random Chord Position Offset* (RCPO), to prevent unwanted changes to the chord progression data output to the MIDI file. Therefore, using the *Lock* command is a tad more secure than just using the *Write Protect* command.

15.7 Unlock Command

Use the Unlock command to allow a command file to be changed, eg.:

```
SMFFTI.exe -unlock chords.txt
```

16. Command Parameter Precedence

To avoid ambiguity when creating a MIDI file, a precedence is given to certain parameters, meaning that when those parameters are set, others will be overridden.

- When `+NoteStagger` is set, these parameters are disabled:
  `+RandNoteStartOffset`
  `+RandNoteEndOffset`
  `+RandVelVariation`

- When `+FunkStrum` is, these parameters are disabled:
  `+Arpeggiator`
  `+RandNoteStartOffset`
  `+RandNoteEndOffset`
  `+RandVelVariation`

- When `+Arpeggiator` is set, these parameters are disabled:
  `+NoteStagger`
  `+RandNoteStartOffset`
  `+RandNoteEndOffset`
  `+RandVelVariation`

Thus it can be seen that `+FunkStrum` has the highest priority, followed by `+Arpeggiator`.

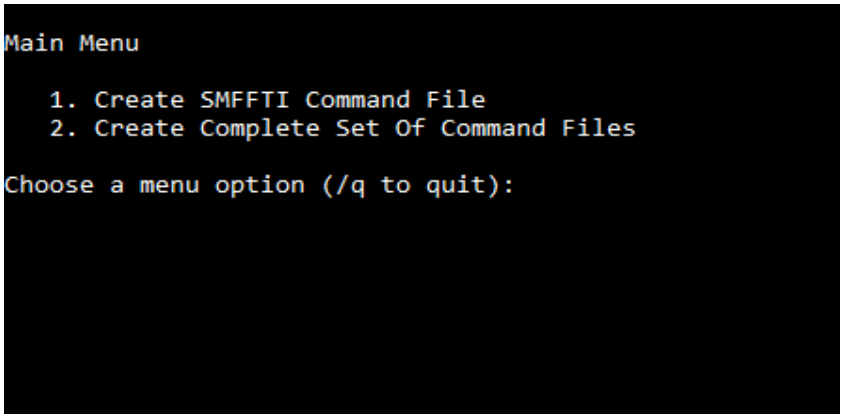
17. SMFFTI Wizard

The *SMFFTI Wizard* is a simple, text-based user interface for performing certain tasks. The wizard presents dialogs, to which you input responses to a series of prompts.

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 simple menu:



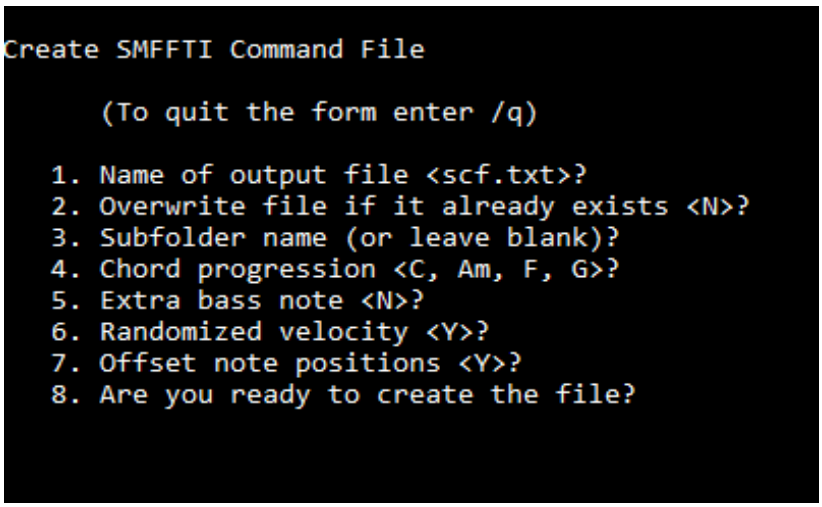
Select a menu option to open a dialog. You can use these commands to navigate a dialog:

`/q`	Quit the dialog.
`/b`	Go back to the previous question.
`/<n>`	Go to the question indicated, where `<n>` is a number, 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 `/1`.

After completing all the questions it will indicate that the task is being actioned; or, report any errors that may occur.

17.1 Create SMFFTI Command File

The Create *SMFFTI Command File* option is an easy way of allowing you to create a basic command file.



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.

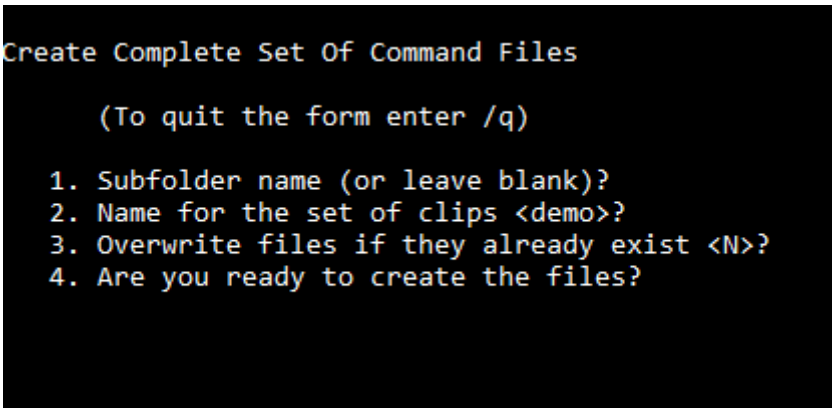
Randomized velocity means the individual notes of the chords will have slightly different velocities.

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.

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.

17.2 Create Complete Set Of Command Files

This option will create a set of files: Two SMFFTI command files and several *Windows* command files. The command files have a file extension of .cmd and are intended to speed up your workflow by performing a pre-defined sequence of SMFFTI operations.



See section *SMFFTI Workflow Discussion* for information about how to use the generated files.

18. SMFFTI Workflow Discussion

This is a discussion of one possible methodology for generating MIDI files for a section of music, using pre-defined sequences of SMFFTI commands. These commands are contained in a group of *Windows* command files, which can be generated using a *SMFFTI Wizard* dialog.

Manual usage of SMFFTI can be a little complex, since there are a lot of commands and parameters, and it's not always easy to remember the order of steps you must perform to create your final MIDI files. The command files attempt to simplify the process for you.

The general workflow is this:

1. Develop a chord progression across a number of bars.

2. Develop randomized sub-rhythms from the chord progression, by fragmenting the lengths of the chord notes of the progression. Each sub-rhythm generated can be used for a different instrument. A sub-rhythm is also used as the basis for melody lines.

3. Develop melodies by using a sub-rhythm (described above) and single-note sequences derived from the underlying chord. Each melody line generated can be used for different instruments in the track.

4. Develop various automatically-generated basslines.

5. Develop arpeggios if required.

Here is a list of the files created by the wizard:

```
chords.txt
scf.txt
smffti_init.cmd
chords.cmd
randomchords.cmd
autochords.cmd
midiToSmffti.cmd
autorhythm.cmd
automelody.cmd
bass.cmd
octavebass.cmd
arpbass.cmd
arp.cmd
genmusic.cmd
ostinato.cmd
rfgg.cmd
```

The philosopy here is to work on a *section* of music, which might be, typically, four or eight bars long. First you need to determine a chord progression, and from this you create your MIDI clips that contain the harmonies, melodies and basslines, etc. You should inspect the `.cmd` files to see what they do and, indeed, edit as necessary, then experiment and develop your own workflow.

Here is the outline workflow for using these files:

1. Double-click file `smffti_init.cmd` to open a command window. It will initialize *environment*

variable `%smffti%`, which you can use to run `SMFFTI.exe`, eg.:

```
%smffti% -v
```

2. Now, if you wish to specify your own chord progression, edit `chords.txt` accordingly to replace the default four-bar progression (Gm - Dm - Eb - Cm). Then run `chords.cmd`, which looks something like this:

```
@echo off
%smffti% chords.txt chords.mid -o
```

The output file is `chords.mid`, which contains your chord progression and can be dropped into Ableton Live. Since you are starting with a known chord progression, steps 4 and 5 do not apply - move on to step 6.

3. If you do not have a chord progression in mind, you can invoke *Auto-Chords* at this point, by running `autochords.cmd`, which looks something like this:

```
@echo off
::exit /b
copy /y scf.txt temp.txt

:: NB. Any parameter changes should be applied to temp.txt, and
:: not chords.txt. Otherwise, the Autochords command (-ac) won't work.

%smffti% -sp "+TrackName=demo_autochords" temp.txt

:: Useful parameters - enable/change if required.
::%smffti% -sp "+AutoChordsWholeNotesOnly = 1" temp.txt
::%smffti% -sp "+AutoChordsNumBars = 8" temp.txt
::%smffti% -sp "+AutoChordsNoteLenBias = 1, 1, 1, 0, 0" temp.txt
::%smffti% -sp "+AutoChords7thChordsOnly = 1" temp.txt
::%smffti% -sp "+ModalInterchangeChancePercentage = 50" temp.txt

%smffti% -ac temp.txt chords.txt -o
%smffti% chords.txt chords.mid -o
```

The `scf.txt` file is just a template file (scf stands for "SMFFTI Command File"). Note here that `scf.txt` is copied to `temp.txt`, which is the file actually used as input for *Auto-Chords*. `scf.txt` contains just a single chord (Gm*) and anticipates the use of *Auto-Chords*. Once again, `chords.mid` is your MIDI clip that can be dropped into Ableton Live. That single chord also specifies the key from which the chords are chosen. Change it to whatever key you prefer.

* To use a different key, edit scf.txt first: Change the last line in the file, where it specifies "Gm".

If you want slightly more jazzy/funky chords, enable the command on line 4 (ie. remove the double-colon).

Now, of course, *Auto-Chords* is unlikely to present you with a perfect chord progression straight-off-the-bat, so you can simply re-run `autochords.cmd` as often as you like until you get a decent start point for further customization.

4. An alternate strategy to *Auto-Chords* is to use `randomchords.cmd`, which enables *Random*

Chord Replacement (RCR). Unlike *Auto-Chords,* RCR does not change the note positions - it only replaces those chords which are prefixed with "?" - so this is a slightly more conservative method of random chord selection. (All the chords in `chords.txt` are already prefixed with "?".)

5. If you end up changing chords and chord positions *manually* inside Ableton Live, you will need to export the clip to `temp.mid` and run `midiToSmffti.cmd`, which looks something like this:

```
@echo off
%smffti% -m temp.mid chords.txt -o
%smffti% -sp "+TrackName=demo_chords" chords.txt
%smffti% chords.txt chords.mid -o
```

6. At this point we assume you have now decided on your final chord progression for the section of music you are working on. This is a cut-off point, and you no longer execute any of the preceding steps. **The critical file is `chords.txt`, which is not expected to change for the remaining part of the workflow.** *The remaining steps are to create rhythms and melodies based on this chord progression, and thus depend on the underlying chord progression not changing.*

   WARNING: To avoid overwriting your chosen chord progression saved in `chords.txt`, make sure you do not run `autochords.cmd` or `randomchords.cmd` again. To protect your file, you should set parameter `+WriteProtected`. This will stop SMFFTI from being able to change the content of the file (even when the -o swtich is used).

   It is also important that *Random Chord Replacement* (RCR) and *Random Chord Position Offset* (RCPO) be disabled in `chords.txt`, because these features may be inadvertently invoked by the operations performed in the command files used in the followinig steps.

7. Run `autorhythm.cmd`. This produces `ar.mid`, which contains the chord progression set to a randomly-generated rhythm pattern. You can re-run `autorhythm.cmd` to create new iterations of `ar.mid` for different instrument in your track. (Note that each execution of `autorhythm.cmd` overwrites `ar.mid`, but that need not be an issue, since each iteration can be imported into your DAW *before* running `autorhythm.cmd` again. In other words, the DAW project itself becomes a database for the various MIDI clips created.)

8. Run `automelody.cmd`, one or more times for different instruments in your track. The melody lines can freely be edited inside your DAW to improve on the timing and notes selected by SMFFTI. By default, the MIDI file produced by `automelody.cmd` is `am.mid`.

9. Basslines. We have four different ways of producing a bassline:

   1. Melodic (`bass.cmd`)
   2. Chord root notes set to a random rhythm (`simplebass.cmd`)
   3. Bouncing octave bass (`octavebass.cmd`)
   4. Arpeggiated bass (`arpbass.cmd`)

10. Arpeggios. Use `arp.cmd`, or copies of it, for different arpeggio styles.

11. Random Funk-Guitar Grooves. Have a look at `rfgg.cmd`. Remember, you need a decent sounding guitar synth.

As a convenience, there is also command file `genmusic.cmd`, which executes `autorhythm.cmd`, `automelody.cmd, bass.cmd` and `arp.cmd` in one easy step. `genmusic.cmd`, like the other command files, is just a template: Copy and customize them all to your heart's content.

You might also find `ostinato.cmd` useful for creating one-bar rhythmic and melodic phrases. However, your chord progression will need to be of an appropriate structure, eg. one, two or four chords which are all the same length - see section *Ostinato Modes* for more information. `ostinato.cmd` produces four MIDI files:

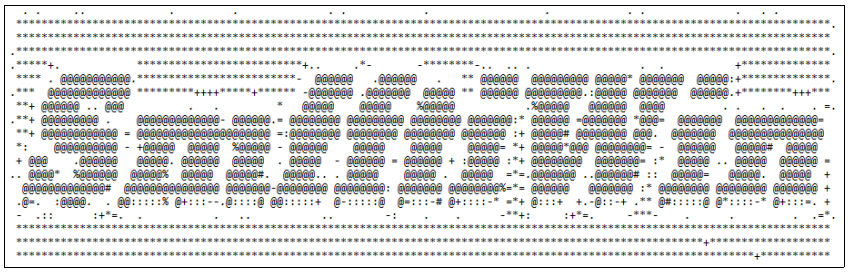
1. A rhythm phrase (`or.mid`)
2. A melodic phrase (`om.mid`)
3. A bass melodic phrase (`bass_ost1.mid`)
4. A bass rhythm phrase using only chord root notes (`bass_ost2.mid`)

19. Summary Of Commands

Key		
SCF	-	SMFFTI Command File
RCR	-	*Random Chord Replacement*
`<scf_in>`	-	Input SCF
`<scf_out>`	-	Output SCF
`<midi_in>`	-	Input MIDI file
`<midi_out>`	-	Output MIDI file
`<text_out>`	-	Output text file
`<pname>`	-	SMFFTI parameter name
`<pval>`	-	SMFFTI parameter value
`<scf_amd>`	-	SCF to be amended
`<nr>`	-	Note Range
`<bn>`	-	Base Note

Create MIDI File	`SMFFTI.exe <scf_in> <midi_out> [-o]`
Auto-Chords	`SMFFTI.exe -ac <scf_in> <scf_out> [-o]`
Auto-Rhythm	`SMFFTI.exe -ar <scf_in> <scf_out> [-o]`
Auto-Melody	`SMFFTI.exe -am <scf_in> <scf_out> [-o]`
Random Funk-Guitar Groove	`SMFFTI.exe -rfgg <scf_in> <scf_out> [-o]`
Ostinato-Rhythm	`SMFFTI.exe -or <scf_in> <scf_out> [-o]`
Ostinato-Melody	`SMFFTI.exe -om <scf_in> <scf_out> [-o]`
MIDI-To-SMFFTI Conversion	`SMFFTI.exe -m <midi_in> <scf_out> [-o]`
Generate Set Of Random Melody Lines	`SMFFTI.exe -grm <text_out> [-o]`
Create Percussion Pattern	`SMFFTI.exe -pp[m] <nr> <bn> <midi_out> [-o]`
Set SMFFTI Parameter	`SMFFTI.exe -sp "<pname>=<pvalue>" <scf_amd>`
Enable SMFFTI Parameter	`SMFFTI.exe -enable <pname> <scf_amd>`
Disable SMFFTI Parameter	`SMFFTI.exe -disable <pname> <scf_amd>`
Delete Parameter	`SMFFTI.exe -dp <pname> <scf_amd>`
Enable Chords For RCR	`SMFFTI.exe -ecfr [y|n] <scf_amd>`
Write Protect	`SMFFTI.exe -wp [0|1|2] <scf_amd>`
Display the version of SMFFTI	`SMFFTI.exe -v`
SMFFTI Wizard	`SMFFTI.exe -w`

20. SmfftiWin



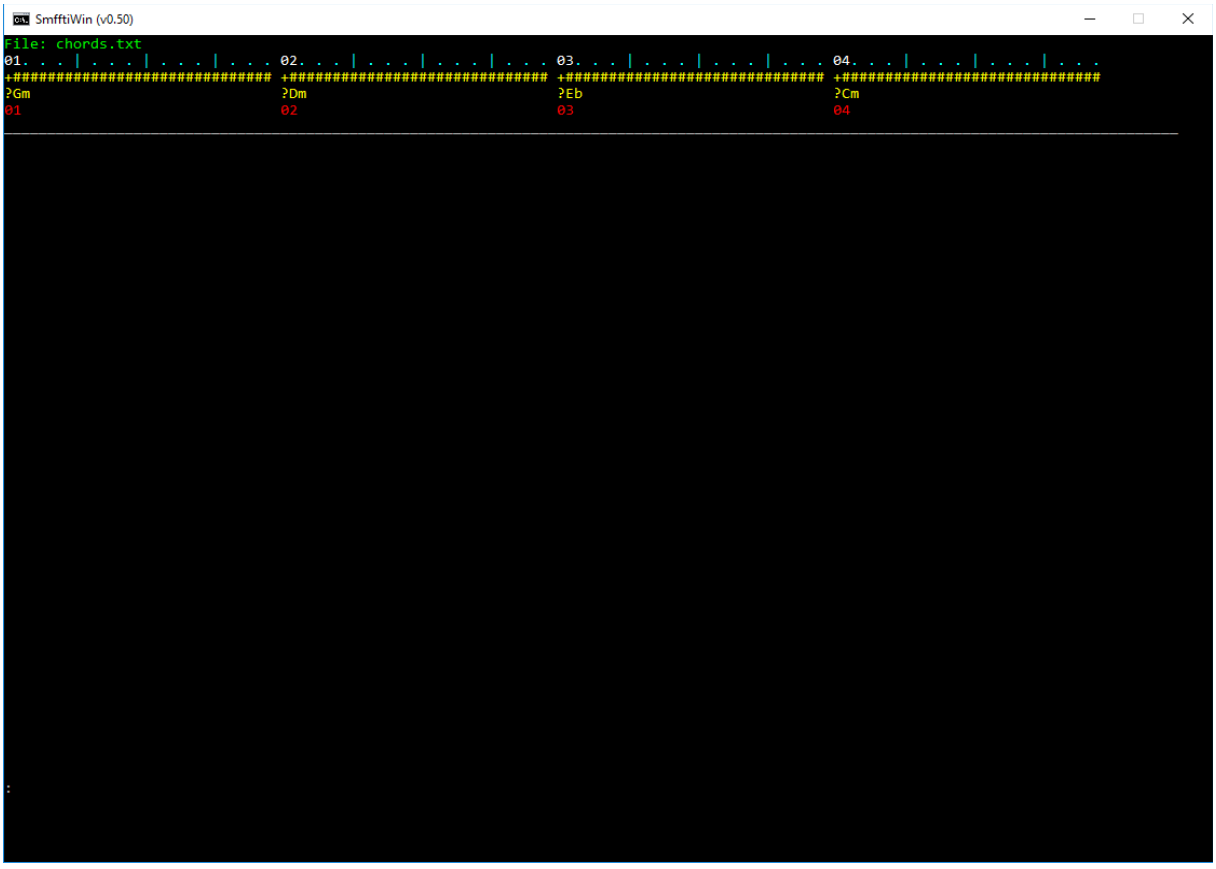
SmfftiWin is an enhancement of the core SMFFTI functionality, by wrapping it a basic "window" application.

All of the operations described in this manual up to this point are performed as single executions of SMFFTI (even if a series of such operations are executed within a Windows command file (`.cmd`)). We might call this "immediate mode", and it's a very free-form method of using SMFFTI.

SmfftiWin, however, provides a more controlled environment. Immediate-mode operation typically involves direct editing of the command file, whereas *SmfftiWin* handles that for you, and gives you a simple interface to the chord progression and control parameters. *SmfftiWin* means you can largely forget about referring directly to the text command file.

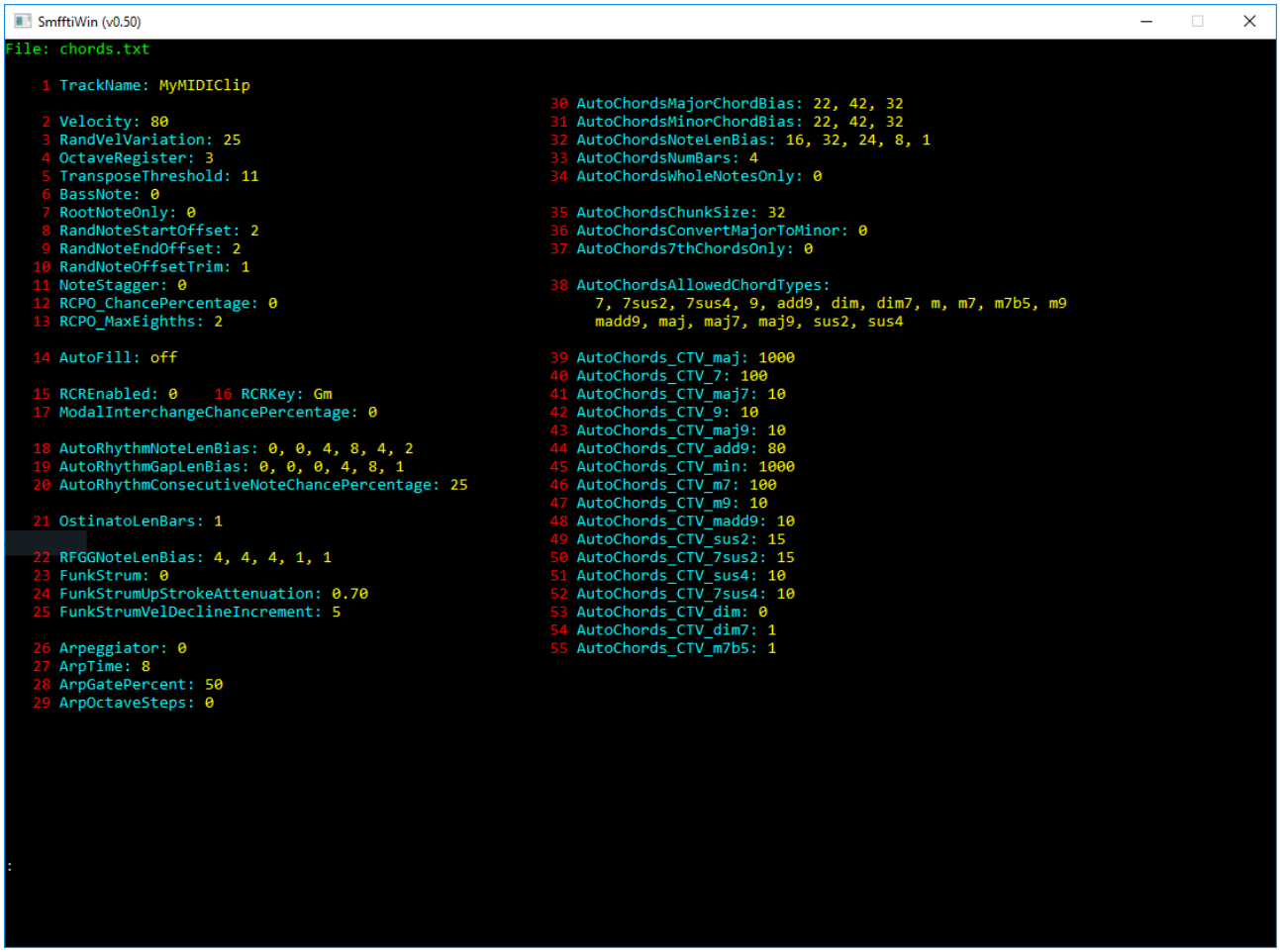
You launch *SmfftiWin* by executing `SMFFTI.exe` without specifying any arguments. You can do this from a Windows command window, or by double-clicking `SMFFTI.exe` in Windows Explorer.

What you first see is a chord progression, eg.:



SmfftiWin attempts to load data from `chords.txt` present in the directory where `SMFFTI.exe` resides. If `chords.txt` does not exist, *SmftiWin* creates a template version of it, complete with a four-bar measure and four chords: Gm-Dm-Eb-Cm.

To view the control parameters, press F3 (or enter `sp` at the prompt).



To return to the chord progression page, press F3 (or enter `cp` at the prompt).

Thus, in two page displays, you have everything at your fingertips. *SmfftiWin* provides a set of commands for easy manipulation of the chord progression and control parameters, as well as all the operations previously described: *Auto-Chords*, *Auto-Rhythm*, *Auto-Melody*, creating MIDI files, etc.

SmfftiWin has been designed to keep things simple. Short commands, often with default values assumed. Unlike immediate-mode, there is less emphasis on the various temporary, intermediate command files that are generated in the workflow process. *SmfftiWin*, by default, assumes `chords.txt` to be at the heart of its operations. `chords.txt` is expected to be your central chord progression, from which all other files are generated.

When you make changes to the chord progression or control parameters using *SmfftiWin*, the changes are automatically saved to the file on disk.

SmfftiWin works with four-bar measures. You can have between one and four of them, ie. a maximum of sixteen bars.

NOTE: Typically, the *SmfftiWin* window will be large enough to display even a quite complex sixteen-bar chord progression. However, in unusual situations the chord progression data *might* extend beyond the the bottom of the display. Unfortunately, at this time, this is a limitation of *SmfftiWin*, which is admittedly a fairly primitive GUI!

The remainder of this section describes the various commands that *SmfftiWin* provides. Note that many of these commands leverage the SMFFTI operations explained in the previous sections of this manual.

20.1 General Commands

20.1.1 Quit (q)

To exit the program, enter `q` or `quit` at the prompt.

20.1.2 Load Command File (cf)

The *Load Command File* will load a SMFFTI command file, replacing whatever file you currently have loaded, eg.

```
cf fred.txt
```

20.1.3 Initialize (init)

The *Initialize* command will reset the chord progression, by initializing it with a template. You begin again with a single four-bar measure and a chord progression of Gm-Dm-Eb-Cm. The parameters remain unchanged.

20.1.4 Copy To File (copy)

The *Copy To File* command will write the content of the currently loaded file (eg. `chords.txt`) to the specified file, eg.:

```
copy bass.txt
```

This will write the currently loaded file content to `bass.txt`. This is useful for manipulating copies of, say, `chords.txt`, to avoid making changes to an original source command file. Having created a copy file, you can load it using the *Load Command File* (`cf`) command.

20.1.5 Write Protect (wp)

The *Write Protect* command allows you to prevent the data in the currently-loaded file from being changed. You can protect the entire content, or just the chord progression (you can still change parameters).

```
wp [ off | all | chords ]
```

20.1.6 Lock (lock)

For finalizing a command file after completion of its chord progression data:

```
lock
```

The *Lock* command is similar to the *Write Protect* command, but it also disables *Random Chord Replacement* (RCR) and *Random Chord Position Offset* (RCPO), to prevent unwanted changes to chord progression data output to the MIDI file. Therefore, using the *Lock* command is a more comprehensive safety mechanism.

20.1.7 Unlock (unlock)

The *Unlock* command allows the command file to be changed:

```
unlock
```

20.1.8 Execute Script File (@)

A script file is a text file that contains a series of valid *SmfftiWin* commands. You can use the at-sign ('@') to run the script, eg.:

```
@bass.txt
```

This will read the file bass.txt and execute all the *SmfftiWin* commands contained therein. Script files offer a convenient way to "program" *SmfftWin*.

`bass.txt` might, for example, contain something like this:

```
# Copy current file content
copy bass.txt
# Load the copy file
cf bass.txt

p OctaveRegister 2
p TransposeThreshold 6
am bass.mid

# Restore original file
cf chords.txt
```

Note that you can also include comments and blank lines in script file.

20.1.9 Undo Command (undo)

The *Undo* command undoes the effect of the previous command, eg.:

```
undo
```

Note that the `undo` command simply restores the previous state of the command file. It does not restore the contents of any files that the command had created, eg. `chords.mid`. So, in such cases, after an `undo`, you might need to, say, use the `midi` command to create a MIDI file that reflects the current state of the commanf file.

20.1.10 Redo Command (redo)

The *Redo* command redoes the effect of a command that was undone, eg.:

```
redo
```

Note that the `redo` command simply changes the state of the command file. See comment in `undo` for ramifications.

20.1.11 SplashScreen Command (splash)

```
splash [ y | n ]
```

Sets whether the splashscreen is displayed at startup.

If you don't specify y or n, it displays the splashscreen, and you must press Enter to continue after each page is displayed.

20.2 Chord Progression Commands

20.2.1 Display Chord Progression Page (cp)

The *Display Chord Progression Page* command, er, displays the chord progression page, eg.:

```
cp
```

20.2.2 Chord Add/Change (c)

The *Chord Add/Change* command allows you to add and change chords.

Format for adding a chord:

```
c <bar>.<beat>.<32nd> <chord name> <length in 32nds>
```

For example:

```
c 3.1.1 Gm7 30
```

This says: On bar three, in the first beat (quarter-note), at the first 32nd of the beat, add a Gm7 chord, and make it thirty 32nds long. It will overwrite any existing chord data that may be present in the specified location.

Format for changing a chord:

```
c <chord number> <chord name> <length in 32nds>
```

When changing a chord, it's more convenient to specify the chord number, rather than the location (bar/beat/32nd). Chord numbers are shown in red below the chord names. For example:

```
c 2 Cm 14
```

This says: Change the chord marked as chord #2 to Cm, and make it fourteen 32nds long. The length is optional, so you can change the chord and keep it the same length.

If you just want to change the length:

```
c 2 * 14
```

Note the asterisk: It means the chord is not being changed.

Whatever length you specify, it will be reduced if it would overlap a following chord.

Chords cannot traverse measures. Thus, the maximum length of a chord is one hundred and twenty eight 32nds - the length of a measure.

You can prefix a chord name with "?", to make it a candidate for *Random Chord Replacement* (RCR).

20.2.3 Delete Chord (dc)

The *Delete Chord* command expects either a chord number, or chord position (bar-beat-32nd), eg.:

```
dc 2
dc 3.1.1
```

The first command says: Delete the second chord. The second command says: Delete the chord at the first 32nd of the first beat of the third bar.

You cannot delete a chord if it is the only chord in a measure.

20.2.4 Add Measure (addm)

The *Add Measure* command will append a new four-bar measure. You can have up to four measures. You must specify one or more chords (maximum of 16). Example:

```
addm Gm, Cm, F, Dm
```

This will add a measure to the end and populate it with chord sequence Gm-Cm-F-Dm. The chords will be spread evenly across the four bars of the measure. (By default a 1/16th gap is left between chords.)

20.2.5 Insert Measure (insm)

The *Insert Measure* command is similar to the *Add Measure* command, but it allows you to insert a new measure before one of the others. You need to indicate which measure to insert before, and enter one or more chords (max. 16). Example:

```
insm 2 Gm, Cm, F, Dm
```

This will insert a new measure before the 2nd one (assuming it exists) and populate it with the chord sequence Gm-Cm-F-Dm.

20.2.6 Delete Measure (delm)

The *Delete Measure* command expects a measure number, eg.:

```
delm 2
```

If there is only one measure, it cannot be deleted.

20.2.7 Duplicate Measure (dupm)

The *Duplicate Measure* command requires a measure number, eg.:

```
dupm 2
```

This will duplicate the 2nd measure and insert it after the 2nd measure.

20.2.8 Initialize Measure (im)

The *Initialize Measure* command will populate a measure with a specified chord sequence, eg.:

```
im 1 Gm, Dm, Eb, Cm
```

This will put the chord progression Gm-Dm-Eb-Cm into the 1st measure. You can specify up to 16 chords. The chords will be spread evenly across the four bars of the measure.

20.2.9 Set Gaps Between Chords (gaps)

This will insert gaps of the specified length, in 32nds, between all chords in the progression. For example:

```
gaps 4
```

This will insert a gap of four 32nds (1/8th) between all chords. Where chords currently have a short length, this may have the effect of extending the chord length. Note that these gaps may be replaced by the effect of the *AutoFill* parameter if it is enabled. *AutoFill* only takes effect when a MIDI file is created. (There is a certain redundancy here; it just so happens that, with regard to setting gaps, you have a choice ;-)).

20.2.10 Enable Chords For RCR (ecfr)

This command will make all chords in the progression candidates for being replaced by the *Random Chord Replacement* (RCR) operation (which takes effect when a MIDI file is created). This is indicated by each chord being prefixed with a question-mark. You use the same command to turn off such candidacy for all chords. Examples:

```
ecfr y
ecfr n
```

20.3 Parameter Commands

20.3.1 Show Parameter Page (sp)

This command will display the page that lists all the control parameters.

20.3.2 Change Parameter (p)

This command allows you to change the value of a parameter listed on the parameter page. The command format is:

```
p <parameter number or name> [<parameter value>]
```

For example:

```
p 4 3
p OctaveRegister 3
```

This changes the value of parameter `OctaveRegister` to 3. If you omit the value from the command, *SmfftiWin* automatically recalls the current value and appends it to the end of the command for you to edit. For example:

```
p 30
```

This will auto-recall the value for parameter #30 (`AutoChordsMajorChordBias`) and present back to you something like this:

```
p 30 "22, 42, 32"
```

and you can edit this and submit.

> NB. If the parameter value remains unchanged after entering the *Change Parameter* command, reset the parameter (`rp` command - see below) and try again. This will happen if the parameter has been manually disabled (ie. commented-out with a #) in the command file - the value of the parameter will have been changed by the *Change Parameter* command, but it won't be reflected in *SmfftiWin* because the parameter is disabled and *SmfftiWin* will therefore reflect the parameter's default value.

20.3.3 Reset Parameter (rp)

This will reset the specified parameter back to its default value. You can specify a parameter number or name, eg.:

```
rp 2
rp Velocity
```

This will reset the value of the `+Velocity` parameter back to its default value. To restore default values for all parameters:

```
rp all
```

20.4 MIDI Operations

20.4.1 Create MIDI File (midi)

This will create a MIDI file using the content of the currently loaded command file, eg. `chords.txt`.

Example 1:

```
midi
```

If the currently loaded file is `chords.txt`, this will create `chords.mid`.

Example 2:

```
midi fred.mid
```

Using the content of the currently loaded command file, this will create a MIDI file called `fred.mid`.

20.4.2 Create MIDI File With Root Notes Only (midiro)

Similar to the `midi` command, but the MIDI file will contain only the root notes of the chords.

Format:

```
midiro [output-MIDI-file]
```

`output-MIDI-file` is optional. If not specified, the output file is `ro.mid`.

20.4.3 Create MIDI File With Root And 5th Notes Only (midir5)

Similar to the `midiro` command, but the MIDI file will contain both the root and 5th notes of the chords.

Format:

```
midir5 [output-MIDI-file]
```

`output-MIDI-file` is optional. If not specified, the output file is `r5.mid`.

Depending on the chords the 5th may not a perfect 5th; it might be a flat or augmented 5th.

20.4.4 Auto-Chords (ac)

The *Auto-Chords* command will create a randomly-generated chord progression, overwriting the existing chord progression. You must specify a key. For example:

```
ac Gm
```

This tells *SmfftiWin* to generate a random chord progression using chords from the key of Gm. A MIDI file is automatically created with a filename corresponding to the name of the currently loaded command file, eg. `chords.mid`.

Please consult the *Auto-Chords* section for information about parameters that can be used to control the *Auto-Chords* algorithm.

20.4.5 Auto-Rhythm (ar)

The *Auto-Rhythm* command will create a MIDI file containing a randomly-generated rhythm pattern based on the current chord progression. By default the MIDI file will be called `ar.mid`, but you can specify a filename if you wish. Examples:

```
ar
ar fred.mid
```

The *Auto-Rhythm* command cannot be used until the chord progression is locked - see *Lock* command.

Please consult the *Auto-Rhythm* section for information about parameters that can be used to control the *Auto-Rhythm* algorithm.

20.4.6 Auto-Melody (am)

The *Auto-Melody* command will create a MIDI file containing a randomly-generated melody (also incorporating a randomly-generated rhythm) based on the current chord progression. By default the MIDI file will be called `am.mid`, but you can specify a filename if so desired. Examples:

```
am
am bill.mid
```

The *Auto-Melody* command cannot be used until the chord progression is locked - see *Lock* command.

20.4.7 Ostinato Rhythm (or)

The *Ostinato Rhythm* command will create a MIDI file containing a randomly-generated ostinato rhythm pattern based on the current chord progression. By default the MIDI file will be called `or.mid`, but you can specify a filename if required. Examples:

```
or
or chas.mid
```

The *Ostinato Rhythm* command cannot be used until the chord progression is locked - see *Lock* command.

20.4.8 Ostinato Melody (om)

The *Ostinato Melody* command will create a MIDI file containing a randomly-generated ostinato melody (also incorporating a randomly-generated ostinato rhythm pattern) based on the current chord progression. By default the MIDI file will be called `om.mid`, but you can specify a filename if you wish. Examples:

```
om
om dave.mid
```

The *Ostinato Melody* command cannot be used until the chord progression is locked - see *Lock*

command.

20.4.9 Auto-Rhythm Bassline (arb)

Similar to *Auto-Rhythm* (`ar`), except the MIDI clip will contain a rhythm pattern that uses only the root notes of the chords, and will be in a lower octave register in order to provide a simple bassline. The default MIDI file is `arb.mid`. Examples:

```
arb
arb bass.mid
```

20.4.10 Auto-Melody Bassline (amb)

Similar to *Auto-Melody* (`am`), except the MIDI clip will be in a lower octave register. The default MIDI file is `amb.mid`. Examples:

```
amb
amb bass.mid
```

20.4.11 Ostinato Rhythm Bassline (orb)

Similar to *Ostinato Rhythm* (`or`), except the MIDI clip will contain an ostinato rhythm pattern that uses only the root notes of the chords, and will be in a lower octave register. The default MIDI file is `orb.mid`. Examples:

```
orb
orb bass.mid
```

20.4.12 Ostinato Melody Bassline (omb)

Similar to *Ostinato Melody* (`om`), except the MIDI clip will be in a lower octave register. The default MIDI file is `omb.mid`. Examples:

```
omb
omb bass.mid
```

20.4.13 Random Funk-Guitar Groove (rfgg)

The *Random Funk-Guitar Groove* command will create a MIDI file containing a randomly-generated rhythm pattern that attempts to articulate the rapid, syncopated strumming style of funk-guitar. By default the MIDI file will be called `rfgg.mid`, but you can specify an alternative name. Examples:

```
rfgg
rfgg chic.mid
```

The *Random Funk-Guitar Groove* command cannot be used until the chord progression is locked - see *Lock* command.

20.4.14 Percussion Patterns (pp and ppm)

The *Percussion Pattern* command will create a one-bar MIDI file that contains a randomly-generated sequence of notes, intended to be used with a percussion instrument. Example:

```
pp 8 C1
```

This will output single notes (ie. no chords) randomly-chosen from C1 - G1 (range of 8 notes). The output file defaults to `pp.mid`. (WARNING: It will automatically overwrite any existing file `pp.mid`.) To output to, say, `ppfred.mid`:

```
pp 8 C1 ppfred.mid
```

Another example, Metronome Mode:

```
ppm 16 C1
```

This ensures that a C1 note will be *placed at the start of every quarter* (so you would, typically, have a kick-drum on C1). The MIDI file will contain randomly-chosen notes from C1 - D#2 (range of 16 notes).

See section *Percussion Patterns* for more information.

20.4.15 More Percussion Patterns (mpc, dfc)

The *Monophonic Percussive Clip* (`mpc`) and *Drum Fill Clip* (`dfc`) commands both serve a different purpose to the `pp/ppm` commands (see above), so are described separately in this section.

However, similar to `pp/ppm`, both commands produce MIDI files containing randomized notes, and are ideal for use with samplers and drum racks that contain percussive sounds.

Both commands output MIDI files that contain, in practice, a contiguous series of notes; that is, with no gaps. All notes in the output MIDI files will be 8th (quaver) or 16th (semiquaver) notes only, which are the best note lengths for most percussive patterns.

The *Monophonic Percussive Clip* command is intended for use with samples that typically contain one or more instruments playing together, eg. a sample from a multi-instrument recording. It is unlikely you would want to play more than one clip at a time from such samples; rather, you would likely play just single snippets, hence it is described as *monophonic*. As a simple example, this works well for randomly chopping up the "Amen Break" sample loaded into a sampler.

The *Drum Fill Clip* command is intended for use with samplers and drum racks that contain classic, individual drum or percussion hits. This result is polyphonic: For each note there can be, randomly, between one and four notes played together. The idea is to emulate a "real" drummer, and for this reason the notes produced are slightly randomly offset and given a random variation in velocity, to provide "humanization".

Both commands have the same input structure, which is:

```
mpc <num note> <root note> <num bars> [<8th-note bias %age> <MIDI file>]
dfc <num note> <root note> <num bars> [<8th-note bias %age> <MIDI file>]
```

where:

`<num notes>` is the number of notes that can be used from the instrument. Range 1 - 60.

`<root note>` is the lowest, starting-point note. Range C0 - B7.

`<num bars>` is the required clip length, in bars. Range 1 - 16.

`<8th-note bias %age>` indicates a bias toward the choice of either 8th or 16th notes, defined as a percentage. For example, if you specify 75, it means: A 75% chance the notes will be 8ths; and a 25% chance of 16ths. If you want only 8ths, specify 100 (or omit this parameter, since it is optional, and the default is 100). If you want only 16ths, specify 0.

`<MIDI file>` is an optional parameter for specifying a MIDI output filename that is different from the default when not specified, which is either `mpc.mid` or `dfc.mid`.

Example commands:

```
mpc 36 C1 4
```

Create a four-bar *Monophonic Percussive Clip* that chooses from 36 notes starting from C1. All the notes will the 8ths. The MIDI output will be `mpc.mid`.

```
dfc 24 C2 8 70
```

Create an eight-bar *Drum Fill Clip* that chooses from 24 notes starting from C2. Most of the notes (70%) will likely be 8ths, the others will be 16ths. The MIDI output file will be `dfc.mid`.

```
mpc 32 C1 4 amen.mid
```

Create a four-bar *Monophonic Percussive Clip* that randoms chooses from a possible 32 notes starting from C1. All the notes will the 8ths. The MIDI output file will be `amen.mid`.

```
dfc 16 C1 2 60 fill.mid
```

Create a two-bar *Drum Fill Clip* that chooses from 16 notes starting from C1. Most of the notes (60%) will likely be quavers, the others will be semiquavers. The MIDI output file will be `fill.mid`.

20.4.16 MIDI-To-SMFFTI Conversion (m2s)

This command will take a MIDI file (that has been exported from Ableton Live*) and convert it to SMFFTI command file format. It will overwrite the content of the currently loaded command file. For example:

```
m2s temp.mid
```

*MIDI-To-SMFFTI conversion is guaranteed to work only with clips exported from Ableton Live.

20.5 Example Script File

Using script files to perform a sequence of *SmfftiWin* commands can be useful. Script files are invoked using the at-sign ('@'). By way of example, the following demonstrates a script that will output multiple MIDI files intended for use a basslines.

```
# Create four standard bassline MIDI clips.

# First, copy current file content
# (which is expected to be chords.txt)
copy bass.txt
# Load the copy file
cf bass.txt

#-----------------------------------
# Melodic bass
p OctaveRegister 2
p TransposeThreshold 6
am bass.mid

#-----------------------------------
# Simple bass
p RootAnd5thOnly ro
ar bass_simple.mid

#-----------------------------------
# Octave bass
p OctaveRegister 1
p Arpeggiator 1
p ArpTime 8
p ArpGatePercent 100
p ArpOctaveSteps 1
midi bass_octave.mid

#-----------------------------------
# Arp bass
p RootAnd5thOnly off
p OctaveRegister 2
p Arpeggiator 9
p ArpTime 8
p ArpGatePercent 100
p ArpOctaveSteps 0
midi bass_arp.mid

# Reload chords.txt
cf chords.txt
```

21. 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.

Parameters can also be disabled by prefixing them with a hash-sign, turning them into comments.

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

21.1 +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`.

21.2 +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).

21.3 +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.)

21.4 +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.

21.5 +AutoChords7thChordsOnly

Causes *Auto-Chords* to use only certain 7th chords. For example:

```
+AutoChords7thChordsOnly = 1
```

This results in *Auto-Chords* choosing from these chord types:

- Dominant 7th (`7`)
- Minor 7th (`m7`)
- 7th Suspended 2nd (`7sus2`)
- 7th Suspended 4th (`7sus4`)
- Half-Diminished (`m7b5`)
- Augmented 7th (`+7`)
- Augmented Major 7th (`+maj7`)

and is the equivalent of:

```
+AutoChordsAllowedChordTypes = 7, m7, 7sus2, 7sus4, m7b5, +7, +maj7
```

Note that not *all* of the 7th chord types are included, eg. *Major 7th*.

`+AutoChords7thChordsOnly` cannot be used together with `+AutoChordsAllowedChordTypes`.

21.6 +AutoChordsAllowedChordTypes

Specifies a subset of the 22 chord type variations. It requires a comma-separated list of the chord types that *Auto-Chords* can choose, subject to what the `+AutoChords_CTV_*` parameters permit. This parameter avoids editing the `+AutoChords_CTV_*` parameters, and potentially losing the original values if you want to restore them. For example:

```
+AutoChordsAllowedChordTypes = maj, min, add9, m7
```

This will cause *Auto-Chords* to choose only *Major*, *Minor*, *Add 9th* or *Minor 7th* chords. You must use the same abbreviations as you would for the chord names in the chord progression; except you must specify `maj` to indicate major chord triads, and you can use either `m` or `min` to specify minor chord triads.

Any chord type variation specified in `+AutoChordsAllowedChordTypes` must have a non-zero value for the corresponding `+AutoChords_CTV_*` parameter.

`+AutoChordsAllowedChordTypes` cannot be used together with `+AutoChords7thChordsOnly`.

21.7 +AutoChordsChunkSize

Applies only to *Auto-Chords* (`-ac`). Specifies the length, within a measure, of a chunk, such that a measure is considered a series of chunks of equal size. A chunk is a section in which notes will be self-contained, ie. notes cannot traverse chunk boundaries. Valid values: 2, 4, 8, 16, 32, 64, 128. These numbers are the possible chunk sizes expressed as 1/32nds. If `+AutoChordsChunkSize` is not specified the default value is 32, ie. the chunk length is one bar.

21.8 +AutoChordsConvertMajorToMinor

Directs *Auto-Chords* and *Random Chord Replacement* to convert any *Major* chords selected to *Minor*. For example:

```
+AutoChordsConvertMajorToMinor = 1
```

21.9 +AutoChords_CTV_<type>

Applies only when *Auto-Chords* mode (`-ac`), or *Random Chord Replacement* 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 22 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)
- dim (Diminished)
- dim7 (Diminished 7th)
- m7b5 (Half-diminished)
- aug (Augmented)
- aug7 (Augmented 7th)
- aug_maj7 (Augmented Major 7th)
- aug9 (Augmented 9th)
- aug_maj9 (Augmented Major 9th)

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.

21.10 +AutoChordsMajorChordBias

Applies only when *Auto-Chords* mode (`-ac`), or *Random Chord Replacement* 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.

21.11 +AutoChordsMinorChordBias

Applies only when *Auto-Chords* mode (`-ac`), or *Random Chord Replacement* 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:

```
+AutoChordsMinorChordBias=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.

21.12 +AutoChordsNoteLenBias

Applies only to *Auto-Chords* mode (`-ac`). Sets a bias for the chances of each possible note length. Possible note lengths: whole-, half-, quarter-, eighth- and sixteenth-notes. For example:

```
+AutoChordsNoteLenBias=1, 2, 4, 2, 1
```

Reading from left-to-right: The "1" value refers to whole-notes, the "2" refers to half-notes, the "4" refers to quarter-notes, etc. In this example, *Auto-Chords* is four times more likely to output quarter-notes than whole- or sixteenth-notes; and twice as likely than half- or eighth-notes.

Comma-separated list of five values, each value range 0 - 100. Default values: 16, 32, 24, 8, 1.

21.13 +AutoChordsNumBars

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

21.14 +AutoChordsWholeNotesOnly

Applies only to *Auto-Chords* mode (`-ac`). Can be 0 or 1 (default is 0). When enabled it will cause *Auto-Chords* to output one chord per bar, overriding the values of parameter `+AutoChordsNoteLenBias`. All chords will be, in fact 15/16ths long, with a 16th-note gap between them.

21.15 +AutoFill

Automatically fills the gaps between chords, or insert small gaps (up to 1/4 note length), so you do not need to explicitly specify note lengths in the chord progression. Range 0 - 127, or "off". Default value: "off". See section *The AutoFill Function* for more information.

21.16 +AutoRhythmConsecutiveNoteChancePercentage

Applicable to *Auto-Rhythm* mode (`-ar`). 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.

21.17 +AutoRhythmGapLenBias

Applicable to *Auto-Rhythm* mode (`-ar`). 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.

21.18 +AutoRhythmNoteLenBias

Applicable to *Auto-Rhythm* mode (`-ar` switch). Sets a bias for the chances of each possible note length. Possible note lengths: whole-, half-, quarter-, 8th-, 16th- and 32nd-notes. Example:

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

Reading 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 than whole-notes; and four times more likely to output 8th-notes.

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

21.19 +BassNote

Valid values: 0 or 1. A value of 1 causes the program a output an additional root note an octave lower. Example:

```
+BassNote = 1
```

Omit if bass note not required, or specify `+BassNote = 0`.

21.20 +FunkStrum

> NB: 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-Guitar Grooves* to see a feature of SMFFTI that allows you to generate randomized funk strumming grooves.

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

*It's been labelled as "funk strum" but, in reality, the articulation applies to general guitar strumming.

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`.

In order to stagger the chord notes appropriately, `+FunkStrum` overrides the existing value `+NoteStagger` and, indeed, sets it to the same value as `+FunkStrum`.

21.21 +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.

21.22 +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.

21.23 +ModalInterchangeChancePercentage

To enable *modal interchange* for *Auto-Chords* (`-ac`) 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.

21.24 +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`, `+RandNoteEndOffset` and `+RandVelVariation`.

21.25 +OctaveRegister

Specifies the octave in which the chord root notes will be placed. By default this is C3 - B3. The format of the parameter value is `<note><number>`, where:

> `<note>` is a note in the chromatic scale, eg. `C, Eb, G#`.
> `<number>` is a number in the range 0 to 7. Default is 3.

`<note>` is optional, in which case it defaults to note `C`. In other words, you can specify `<note><number>` or just `<number>`. You cannot specify just `<note>`. Valid examples:

```
+OctaveRegister = 2
+OctaveRegister = C3
+OctaveRegister = Eb4
+OctaveRegister = G#2
```

This is all with respect to the chord *root* notes. By default, the additional notes in the chord might well extend beyond the uppermost note in the register, unless parameter `+TransposeThreshold` is set to a value which causes these notes to be transposed downward. If, forexample, `+TransposeThreshold = 11`, then all *non-root* notes higher than (the register C note + 11 semitones) will be transposed downward.

It is important to understand that `<number>` defines the register for the chord root notes: This is strictly `C<number>` to `B<number>`, eg. C3 - B3. As such, you generally only need to specify `<number>` only.

The purpose of `<note>` is to specify a note that represents, provisionally, the *lowest note in the register above which notes can be transposed downward*. This is best explained by an example.

Suppose that:

```
+OctaveRegister = Eb3
+TransposeThreshold = 11
```

This means:

- All *root* notes will sit between C3 - B3.
- All *non-root* notes higher than Eb3 plus 11 semitones will be transposed downward.

21.26 +OstinatoGapLenBias

Controls the length of the gaps between the notes in an ostinato pattern. Format:

```
+OstinatoGapLenBias = <pc4>, <pc8>, <pc16>
```

The format is the same as `+OstinatoNoteLenBias` (see below).

21.27 +OstinatoLenBars

Applies to *Ostinato Rhythm*. For specifying how long the ostinato rhythm pattern will be: Either one or two bars, eg.:

```
+OstinatoLenBars = 2
```

If `+OstinatoLenBars` is not specified the length of the ostinato rhythm will be one bar.

21.28 +OstinatoNoteLenBias

Controls the length of the notes in an ostinato pattern. Comma-separated list of three numbers that each represent a percentage. Format:

```
+OstinatoNoteLenBias = <pc4>, <pc8>, <pc16>
```

where:

`<pc4>` is the percentage chance of 1/4 notes.
`<pc8>` is the percentage chance of 8th notes.
`<pc16>` is the percentage chance of 16th notes.

Maximum total percentage 100%. If the total percentage specified is less than 100, the remaining percentage is the chance of 32nd notes. Default values: 10, 30, 30.

21.29 +RandNoteEndOffset

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

21.30 +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.

21.31 +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.

21.32 +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.

21.33 +RCPO_ChancePercentage

The likelihood of *Random Chord Position Offset* (RCPO) occurring, expressed as a percentage. Default is zero. Incompatible with *Auto-Chords*, *Auto-Rhythm*, *Auto-Melody, Ostinato Rhythm* and *Ostinato Melody*. For these operations RCPO is disabled, and `+RCPO_ChancePercentage` is set to zero in the output file.

21.34 +RCPO_MaxEighths

The maximum size of a chord position offset for the *Random Chord Position Offset* operation, in terms of 1/8th notes. Valid range: 1 - 4. Default is one (ie. one eighth note). Ignored if `+RCPO_ChancePercentage` is zero.

21.35 +RCREnabled

Activates *Random Chord Replacement* (RCR). Valid values: 0 (off) or 1 (on).

21.36 +RCRKey

Defines the key from which replacement chords are randomly selected. For example:

```
+RCRKey = Gm
```

If not specified, the default key is Gm.

21.37 +RFGGNoteLenBias

Applies to the *Random Funk-Guitar Groove* operation. Sets a bias for the chances of the occurrence of gaps and different note length in the generated rhythm patterns. It takes a comma-separated list of five numbers, eg.:

```
+RFGGNoteLenBias = 2, 2, 6, 1, 1
```

From left to right the numbers refer to the likelihood of (1) gaps, (2) 1/16th notes, (3) 1/8th notes, (4)

3/16th notes, and (5) 1/4 notes. Each number can be in the range 0 - 100, but you must specifiy at least one non-zero number, and you can't specify gaps only.

The numbers specify a bias toward the respective gap/note-lengths. In the example above, therefore, there is:

- An equal chance of 3/16th and 1/4 notes occurring.
- A six times greater likelihood of 1/8th notes occurring than 1/4 notes, and three times greater likelihood than gaps or 1/16th notes.
- An equal chance of gaps or 1/16th notes.
- Double the chances of gaps and 1/16th notes compared to 3/16th and 1/4 notes.

If you do not specify `+RFGGNoteLenBias` in your SMFFTI command file, the default settings is:

```
+RFGGNoteLenBias = 4, 4, 4, 1, 1
```

21.38 +RootAnd5thOnly

Possible values: `off`, `ro` or `r5`. A value of `ro` causes SMFFTI to output only the root note of the chords to the MIDI file. `r5` causes the only the root and the 5th note to be output.

21.39 +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 "SMFFTI".

21.40 +TransposeThreshold

Number in range 0 to 48. Default 48. This dictates how notes in a chord (including the root note) will be transposed downward. It specifies a number of semitones *above* the note C for `+0ctaveRegister`. Notes beyond this will be transposed downward. For example, if `+0ctaveRegister=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.

21.41 +Velocity

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

21.42 +WriteProtected

When enabled, this prevents the SMFFTI command file from being changed. This is provide protection against accidentally overwriting, for example, chord progression data or parameters. Valid: 0, 1 or 2. If zero (or not specified), the file is not write-protected. If set to 1, nothing in the file can be changed. If set to 2, only the chord progression data is write-protected; parameters can still be changed.

Obviously, such write-protection is only afforded when using SMFFTI. You are still at liberty to manually edit the file.

22. 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 on the same command file. Thus, the SMFFTI command file also acts as a self-contained database. Such parameters are not intended to be amended or referred to by the user. System parameters are indicated as such with the prefix SYS_.

22.1 +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.

When the first iteration of RCR occurs - that is, a chord has been replaced - this parameter will be saved to the SMFFTI command file. It consists of a comma-separated list of four numbers, each of which pertain to one of the four possible sets (measures) of chord progression data.

23. 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
+RootAnd5thOnly=off

+Velocity=70
+RandVelVariation=25

+OctaveRegister=3
+TransposeThreshold=11

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

+NoteStagger=0

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

+AutoRhythmNoteLenBias=0, 0, 2, 8, 4, 2
+AutoRhythmGapLenBias=0, 0, 0, 4, 8, 1
+AutoRhythmConsecutiveNoteChancePercentage=25

# Parameters relating to Auto-Chords (-ac)
# and Random Chord Replacement.
#
+AutoChordsNumBars=4;
+AutoChordsMinorChordBias=22, 42, 32
+AutoChordsMajorChordBias=22, 42, 32
+AutoChordsNoteLenBias=1, 1, 1, 1, 0
#
# Chance of Chord Type Variations (CTV) for Major chords.
# Note that suspended and augmented chords are considered
# variations of major chords.
#
+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
+AutoChords_CTV_aug      = 5
+AutoChords_CTV_aug7     = 0
+AutoChords_CTV_aug_maj7 = 0
+AutoChords_CTV_aug9     = 0
+AutoChords_CTV_aug_maj9 = 0
#
# 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, they're generally unpleasant;
# instead, equal chance of Diminished 7th or Half-diminished.
+AutoChords_CTV_dim     = 0
+AutoChords_CTV_dim7    = 1
+AutoChords_CTV_m7b5    = 1
#
+AutoChords7thChordsOnly = 0
+AutoChordsAllowedChordTypes = all
+AutoChordsChunkSize = 32
+AutoChordsConvertMajorToMinor = 0
+AutoChordsWholeNotesOnly = 0

+FunkStrum=0
+FunkStrumUpStrokeAttenuation=0.70
+FunkStrumVelDeclineIncrement=5

+RCREnabled = 0
+RCRKey = Gm
+ModalInterchangeChancePercentage=0

+RCPO_ChancePercentage = 0
+RCPO_MaxEighths = 2
+AutoFill = 0
+OstinatoLenBars = 1
+OstinatoGapLenBias = 10,30,30
+OstinatoNoteLenBias = 10,30,30
+RFGGNoteLenBias = 4, 4, 4, 1, 1

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############################### +###############################
Gm, Dm

$ . . . | . . . | . . . | . . . $ . . . | . . . | . . . | . . .
+############################### +###############################
Eb, Cm
```