Fuzzy Music in Lisp

The file fuzzy_music.lisp contains functions that build on those in fuzzy.lisp to apply fuzzy procedures to musical problems.

Basics

The use of fuzzy logic for music depends on using sets to represent familiar music objects. This is not the usual music "set theory"-- when I need that kind of set, I'll call it a pitch list. The set I mean is an ordered set on the universe of pitch class. It's a list of 12 0s and 1, where a 1 indicates the inclusion of a pitch in the set.

Thus {1 0 0 0 0 0 0 0 0 0 0 0} is a single note, C.
{1 0 0 0 1 0 0 1 0 0 0 0} is a C major traid.
{1 0 1 0 1 1 0 1 0 1 0 1} is a C major scale.

These sets are transposed by rotation to the right. If I rotate a set right by 2, the last two elements of the set are pasted in front.
Thus {1 0 0 0 1 0 0 1 0 0 0 0} becomes {0 0 1 0 0 0 1 0 0 1 0 0}, which is a D major triad.

This gets fuzzy when values other than one are used in the list. For instance, the set {0 0 0 1 0.9 0 0 0 0 0 0 0} represents an interval of a third up. There are two non zero members because there are two kinds of third. To find a note a third above D, that set is rotated two steps and intersected with the active scale:

```
{0 0 0 1 0.9 0 0 0 0 0 0 0}  ;; start with a third above
{0 0 0 0 0 1 0.9 0 0 0 0 0}  ;; rotate to D
{1 0 1 0 1 1 0 1 0 1 0 1}      ;; intersect (fuzzy intersection is the minimum values)
{0 0 0 0 0 1 0 0 0 0 0 0}      ;; this is an F
```

Likewise {0 0 0 0 0 0 0.9 1 0.8 0 0 0} represents a fifth above. The relative weights of the tritone and augmented fifth represent a preference for one over the other. In use, these values may be adjusted to fit the situation. To find a fifth above a D:

```
{0 0 0 0 0 0 0.9 1 0.8 0 0 0}  ;; a fifth above
{0 0 0 0 0 0 0 0 0.9 1 0.8 0}  ;; rotate to D
{1 0 1 0 1 1 0 1 0    1 0    1}      ;; intersect  (spaced to match the line above)
{0 0 0 0 0 0 0 0 0    1 0    0}      ;; this is an A
```

To build a chord, we take the union of the note, the third above the note and the fifth above the note.

Functions for Fuzzy Music

Constants

The following fuzzy sets for musical structures are defined in fuzzy_music.lisp:

```
(defConstant *FZ-NOTE* '(1 0 0 0 0 0 0 0 0 0 0 0))

; the scales
(defConstant *MAJ-SCALE* '(1 0 1 0 1 1 0 1 0 1 0 1))
(defConstant *NMIN-SCALE* '(1 0 1 1 0 1 0 1 1 0 1 0))
(defConstant *HMIN-SCALE* '(1 0 1 1 0 1 0 1 1 0 0 1))

;; update these to change key
(defVar *CURRENT-SCALE* *maj-scale*)
(defVar *TONIC* 0)

; intervals up
(defConstant *FZ-2* '(0 0.9 1 0 0 0 0 0 0 0 0 0))
(defConstant *FZ-3* '(0 0 0 1 0.9 0 0 0 0 0 0 0))
(defConstant *FZ-4* '(0 0 0 0 0 1 0.5 0 0 0 0 0))
(defConstant *FZ-5* '(0 0 0 0 0 0 0.9 1 0.5 0 0 0))
(defConstant *FZ-6* '(0 0 0 0 0 0 0 0 1 0.9 0 0))
(defConstant *FZ-7* '(0 0 0 0 0 0 0 0 0 0 0.9 1))

;intervals down
(defConstant *FZ-2B* '(0 0 0 0 0 0 0 0 0 0 1 0.9))
(defConstant *FZ-3B* '(0 0 0 0 0 0 0 0 0 0.9 1 0 0))
(defConstant *FZ-4B* '(0 0 0 0 0 0 0 0.5 1 0 0 0 0))
(defConstant *FZ-5B* '(0 0 0 0 0 0.5 1 0.9 0 0 0 0 0))
(defConstant *FZ-6B* '(0 0 0.9 1 0 0 0 0 0 0 0 0))
(defConstant *FZ-7B* '(0 0.9 1 0 0 0 0 0 0 0 0 0))

; pitch classes
(defConstant PC-C 0)
(defConstant PC-DFLAT 1)
(defConstant PC-D 2)
(defConstant PC-EFLAT 3)
(defConstant PC-E 4)
(defConstant PC-F 5)
(defConstant PC-GFLAT 6)
(defConstant PC-G 7)
(defConstant PC-AFLAT 8)
(defConstant PC-A 9)
```

```
(defConstant PC-BFLAT 10)
(defConstant PC-B 11)
```

Transposition

Transposition of pitch class sets is done by circular rotation to the right. The last element is moved to the front as many times as required. This can be done by ror-n, which takes a list and the number of steps to rotate.

```
? (ror-n '(1 0 0 0 0 0 0 0 0 0 0 0) 3)
(0 0 0 1 0 0 0 0 0 0 0 0)
```

I seldom rotate to the left, but that is required for finding normal form in pitch list analysis.

```
? (rol-n '(0 4 7) 2)
(7 0 4)
```

Transposition of sets may occasionally be needed without rotation, in which case you wind up with a larger set. shift-n will do this, returning a two octave set. This can simplify voice leading. A negative (left or down) shift will return a set based one octave lower than the input.

```
? (shift-n '(1 0 0 0 0 0 0 0 0 0 0 0) 3)
(0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

Conversion of Sets to Pitch Class

At the end of the day, operations carried out as fuzzy sets will need to be converted to a pitch or group of pitches. There are two functions to do this: top-n-positions returns a list of indices of the highest memberships in the set, Ltop returns the index (not a list) of the highest member.

```
? (top-n-positions '(0 0 1 0 0 0 0.9 0 0 0.7 0 0) 3)
(2 6 9)

? (ltop '(0 0 0.9 0 0 0 1 0 0 0.7 0 0))
6
```

Building sets from pitch lists

Converting pitches to sets is available in make-set.

```
? (make-set '( 2 6 9))
(0 0 1 0 0 0 1 0 0 1 0 0)
```

Converting to playable form

The list returned by make-triad-list can be placed in proper order by ascend-list. and moved to a usable MIDI note range

```
? (ascend-list '(7 11 2))
(7 11 14)

? (add-oct '(7 11 14) 3)
(43 47 50)
```

The octave specified by the second argument to add-octave is octaves above 0, not the MIDI or traditional octave number. You may wish to modify this function to suit your preferred usage.

A chord list can be converted to an event with the function make-chord-event, which refers to global defaults for duration, channel and velocity, if these are not supplied.

```
(defVar *DEFAULT-DUR* 1000)
(defVar *DEFAULT-CH* 1)
(defVar *DEFAULT-VEL* 127)

;;;
(defun MAKE-CHORD-EVENT (pitch-list &optional (time 0)
                                (dur *default-dur*)
                                (ch *default-ch*)
                                (vel *default-vel*))
  "simple notelist to chord "

? (make-chord-event '(60 64 67) )
((0 60 1000 1 127) (0 64 1000 1 127) (0 67 1000 1 127))
```

Interval builders

These functions provide chord elements using the global *current-scale*. Similar functions are available intervals up to sevenths each way.

```
? (third-above 7)
11
? (fifth-above 7)
2
? (third-below 7)
4
? (fifth-below 7)
0
```

Triad Builders

Functions can use these to return chords with given pitch as specified element. These are defined for triads:

```
? (as-root 5)
(5 8 11)
? (as-third 5)
(1 5 8)
? (as-fifth 5)
(10 1 5)
```

They return a pitch list in root position.

Basic Harmonizer

To illustrate the use of these functions, here is a harmonizer of the type used by many keyboardists to accompany simple tunes.
Start with some global variables for solution sets and the like:

```
(defVar *SOL-SET* '(0 0 0))   ;places are (as-root as-third as-fifth)
(defVar *LAST-SOL* '(0 0 0))  ; the last solution set-- stores inversion
(defVar *LAST-SOL-WEIGHT* 0.6)  ; wieghting of last solution
(defVar *LAST-PC* 0)
(defVar *LAST-CHORD* '(0 4 7))
(defVar *LAST-ROOT* 7)
(defVar *THIS-ROOT* 0)
(defVar *OLD-CHORD-SET* '(0 0 1 0 0 0 0 1 0 0 0 1))
(defVar *AS-ROOT-SET* '(0 0 0 0 0 0 0 0 0 0 0 0))
(defVar *AS-THIRD-SET* '(0 0 0 0 0 0 0 0 0 0 0 0))
(defVar *AS-FIFTH-SET* '(0 0 0 0 0 0 0 0 0 0 0 0))
```

When discrete choices are required (as opposed to the "find a value on a sliding scale" operations usually encountered in fuzzy work) I use a process similar to voting. The solution set has a member for each option. The rules add to the membership for the options they favor. The amount added to the membership is determined by how true a rule is.

For this harmonizer, I will consider the three triads that can be constructed including a given pitch: pitch-as-root, pitch-as-third and pitch-as-fifth. The factors used will be the number of pitches each has in common with the previous chord, and how often a particular construction has been used. These are the rules:

Common tone rules

- if *as-root-set* has common tones with *old-chord-set* use as-root (1 0 0)
- if *as-third-set* has common tones with *old-chord-set* use as-third (0 1 0)
- if *as-fifth-set* has common tones with *old-chord-set* use as-fifth (0 0 1)

Last solution rules

- if last solution was as-root, use as-third or as-fifth (0 1 1)
- if last solution was as-third, use as-root or as-third (1 1 0)
- if last solution was as-fifth, use as-root (1 0 0)

The results of all of these rule will just be added together, which may look like this:

(5 3 2)

The Ltop function will then pick a winner.

Here are the complete functions that evaluate the common tone rules. These work through global variables so they can easily be called in conjunction with other functions.

This counts the tones in common, using sets:

```
(defun COMMON-TONES (set1 set4)
   (sumup (fz-intersect set1 set4)))
```

This lists the number of common tones in the appropriate slot of a solution set.

```
(defun COMMON-TONE-RULES (set1 set2 set3 master)
   (let ((result '(0 0 0)))
     (setf (first result) (common-tones set1 master))
     (setf (second result) (common-tones set2 master))
     (setf (third result) (common-tones set3 master))
     (fz-normalize-list result)))
```

This calls the test on the globals

```
(defun COMMON-TONES-TEST ()
   (common-tone-rules *as-root-set*
                      *as-third-set*
                      *as-fifth-set*
                      (fz-crisp-up *old-chord-set*)))
```

Here's a trial run:

```
? (setq *old-chord-set* (make-Set (as-root pc-g)))
(0 0 1 0 0 0 0 1 0 0 0 1)
(setq *as-root-set* (make-Set(as-root pc-c))
 (setq *as-third-set* (make-Set(as-third pc-c)))
 (setq *as-fifth-set* (make-Set(as-fifth pc-c)))

? (common-tones-test)
(1.0 0.0 0.0)
```

This shows a root position chord on C has more common tones with a G chord than the other options. A second run confirms the operation

```
? (setq *old-chord-set* (make-Set (as-root pc-e)))
(0 0 0 0 1 0 0 1 0 0 0 1)
? (common-tones-test)
(1.0 0.5 0.0)
```

This shows common tones between an Emin chord and chords with root as C or third as C, and that there are more common tones on the root position.

To break up parallel motion, I favor solutions that will encourage a change of position from chord to chord. To review:

- if last solution was as-root, use as-third or as-fifth
- if last solution was as-third, use as-root or as-third
- if last solution was as-fifth, use as-root

```
(defun LAST-SOL-TEST ()
   (let ((test (ltop *last-sol* )))
     (cond
      ((= 0 test) '(0 1 1))
      ((= 1 test) '(1 1 0))
      (t '(1 0 0)))))

? (last-sol-test)
(0 1 1)
```

Here is the function that executes the rules. Note that the rules are just added up and the highest vote-getter is chosen.

```
((defun PICK-CHORD (thenote)
  (let ((the-pc (rem thenote 12)) (solSet '(0 0 0))(result))
    (setq *as-root-set* (make-Set(as-root the-pc)))
    (setq *as-third-set* (make-Set(as-third the-pc)))
    (setq *as-fifth-set* (make-Set(as-fifth the-pc)))
    (setq solSet (add-lists solSet (common-tones-test)))
    (setq solSet (add-lists solSet (last-sol-test)))
    (setq result
         (case (ltop (setq *last-sol* solset))
           (2 (as-fifth the-pc))
           (1 (as-third the-pc))
           (otherwise (as-root the-pc))))
    (setq *old-chord-set* (make-set result))
    result))
```

Here is a short run of pick-chord on the pitches `0 7 5 4 2 7 0`

(9 0 4)
(4 7 11)
(5 9 0)
(9 0 4)
(2 5 9)
(7 11 2)
(0 4 7)

This is very dependent on previous state. If I evaluate a D, it will return (2 5 9), then the same pitches will yield:

(5 9 0)
(7 11 2)
(11 2 5)
(4 7 11)
(7 11 2)
(7 11 2)
(0 4 7)

The importance of the initial state should be kept in mind in serious applications.

After the harmonization is determined, I modify the voice leading. Here are some rules:

- Leave common tones under the same finger
- If too many root, then 6 or 6-4
- If too many 6 then root or 6-4
- Do not follow 6-4 with 6-4
- If motion is fifth, change inversion
- If tonic, favor root a bit

These requires more global variables to keep track of previous results:

```
(defvar *how-many-root* 0)
(defvar *too-many-root* '(0 0.2 0.6 1 1 1 1 1))
(defvar *how-many-6-4* 0)
(defvar *too-many-6-4* '( 0 1 1 1 1 1 1 1 1))
(defvar *how-many-6* 0)
(defvar *too-many-6* '(0 0.2 0.6 1 1 1 1 1 1))
```

To figure suspended tones, examine each possible inversion and compare it with the previously defined chord. We are looking for the same pitch in the same position. This will be done by count-common;

```
(defun COUNT-COMMON (alist blist )
```

```
 (if (or (null alist) (null blist)) 0
    (+ (count-common (cdr alist)(cdr blist))
      (if (equal (car alist)(car blist)) 1 0))))
```

```
? (count-common '(0 4 7) '(11 2 7))
1
```

We'll use this to initialize the solution set, called inv-set in the function. Inv-set is a three member set which will contain votes for root, 6-4, and 6 inversion. With that order, the Ltop function will return the number of steps to ror-n the root position chord. (Rotating pitch lists gives inversions of the chord.) The entire rule looks like this: (clist is the chord to be voiced)

```
(setq inv-set
        (fz-normalize-list
                (list (count-common *last-chord* clist)
            (count-common *last-chord* (ror-n clist 1))
                        (count-common *last-chord* (ror-n clist 2)))))
```

this can easily be expanded to consider open and closed positions. To count the number of times an inversion is used, we'll call a function to set the *how-many- variables once the result is calculated.

```
(defun UPDATE-COUNTS (inversion)
 (case inversion
  (0 (setq *how-many-root*(1+ *how-many-root*))
   (setq *how-many-6-4* 0)
   (setq *how-many-6* 0))
  (1 (setq *how-many-root* 0)
   (setq *how-many-6-4*(1+ *how-many-6-4*))
   (setq *how-many-6* 0))
  (2 (setq *how-many-root* 0)
   (setq *how-many-6-4* 0)
   (setq *how-many-6* (1+ *how-many-6* ))))
 inversion)
```

By returning the input variable like this, the function can be put in line with the result calculation.

These are used in the function by this type of code:

```
(setq inv-set
       (add-lists inv-set
              (fz-rule *how-many-root*
                     *too-many-root* '(0 0.8 1))))
```

To detect downward fifths there is this little function:

```
(defun CHECK-MOTION-FOR-FIFTH (arg1 arg2)
  (and (= 7 (- arg1 arg2))
       (/= *last-root* (car *last-chord*))))
```

The complete inversions function is this

```
(defun MAKE-INVERSION (clist)
  (let ((inv-set '(0 0 0)) (the-root (get-root clist)) )
    (setq inv-set
          (fz-normalize-list
           (list (count-common *last-chord* clist)
                 (count-common *last-chord* (ror-n clist 1))
                 (count-common *last-chord* (ror-n clist 2))))) ;rule 1
    (setq inv-set
          (add-lists inv-set
                     (fz-rule *how-many-root*
                              *too-many-root* '(0 0.8 1)))) ;rule2
    (setq inv-set
          (add-lists inv-set
                     (fz-rule *how-many-6-4*
                              *too-many-6-4* '(1 0 1)))) ;rule 3
    (setq inv-set
          (add-lists inv-set
                     (fz-rule *how-many-6*
                              *too-many-6* '(1 0.8 0)))) ; rule 4
    (if (check-motion-for-fifth *last-root* the-root)
      (setq inv-set (add-lists inv-set '(1 0 0))))    ;rule 5
    (if (= *tonic* the-root)
      (setq inv-set (add-lists inv-set '(0.2 0 0)))) ; rule 6
    ;(format T "~A  " inv-set)      ; for adjusting rules
    (setq *last-root* the-root)    ;remember for next time
    (setq *last-chord* (ror-n clist (update-counts (ltop inv-set))))))
```

Running a test set of (0 4 7)(5 9 0)(2 5 9)(7 11 2)(0 4 7) returns

```
(0 4 7)
(0 5 9)
(2 5 9)
(2 7 11)
(0 4 7)
```

A reasonable passage.

Putting it all together requires most of the functions covered here in a long string. This harmonizer expects both a pitch-list and duration-list and returns the tune with an accompaniment in a low octave. An example is provided in the file.

```
(defun Harmonize (plist dlist &optional (time 0) )
  (if (or (null plist)(null dlist)) ()
      (append
       (make-chord-event
        (cons (car plist)
              (add-oct
               (ascend-list
                (make-inversion
                 (pick-chord (car plist)))) 4)) time (car dlist))
       (Harmonize (cdr plist)(cdr dlist) (+ time (car dlist))))))
```