# Gradus ad Fuzzem

One test of musical application of any methodology is to see if it works with simple counterpoint. This is easily testable and usually well understood by the reader. Gradus ad Fuzzem is an application of fuzzy logic to species 1.

## The Rules

### Rules for Harmonic intervals

The interval requirements of counterpoint vary with each section of the Cantus Firmus.

The starting interval must be the unison,  fifth or octave.

In the mid-game Fux (via Mann) states four rules based on states of consonance:

- Motion from perfect consonance to perfect consonance must be contrary or oblique.
- Motion from imperfect consonance to perfect consonance must be contrary or oblique.
- Motion from perfect consonance to imperfect consonance may be contrary, oblique or direct.
- Motion from imperfect consonance to imperfect consonance may be contrary, oblique or direct.

In species I only consonant intervals are allowed ( unison, thirds, fifths, sixths and octaves).

The penultimate interval must be a major sixth if the CF is the lower voice, or a minor third if the CF is upper. (A proper CF will end on stepwise motion to the final; down if the CF is in the bass, up if the CF is in the tenor.)

The ending interval must be the unison, fifth or octave.

The Interval rules can be restated as a set of legal targets for each CF motion. The target is some interval from the next note of the CF:

- The set of imperfect intervals is always allowed.
- If cf repeats the same pitch, any perfect interval is allowed
- If cf movement is up , perfect intervals are allowed if the counterpoint moves down. (Lower perfects)
- If cf movement is down, perfect intervals are allowed if the counterpoint moves up. (Higher perfects)

## Motion rules

The counterpoint has some constraints on its own, primarily to make it singable:

- Notes are limited to the tessitura
- No tritone steps.
- Favor small steps.

These rules help avoid writing the counterpoint into a corner.

- Do not repeat pitch.
- If future of cf is up, go up

Stating the rules with sets:
Membership in the range of notes is defined:
*tessitura-d* '(1 0.5 1 0 1 1 0 1 0 1 0 1) (To 3 octaves)

(The fuzzy value of 0.5 on PC 1 and its octave allow the raised 7th in Dorian.) Tessitura for other modes are constructed in a similar manner.

The interval types are defined in fz_gradus.lisp

```
*perfs*       (1 0 0 0 0 0 0 1 0 0 0 0) (To 3 octaves)
*imperfs*     (0 0 0 1 1 0 0 0 1 1 0 0)
```

These sets are rotated (>>) to the pitch class under consideration. Thus

{*perfs*>> D } is (0 0 1 0 0 0 0 0 0 1 0 0 )

The set lower-perfects is an intersection (∩) of the *perfs* set and a lower-than function, where:

(lower-than n) is constructed as M where for i < n = 1 $m_i$ = 1; for i >= n = 1 $m_i$ = 0
(higher-than n) is constructed as M where for i <= n = 1 $m_i$ = 0; for i > n = 1 $m_i$ = 1

Higher than 3 would be (0 0 0 0 1 1 1 1 1 1 1 1)

Thus target sets are constructed by
If cf motion is the same (zerop):
{{*consonant*>> cf } ∩ {{lower-than last-pc} ∪ {higher-than last-pc}}}
∩ {tessitura}

If cf motion is up (plusp)
{{{*perfs*>>cf } ∩ {lower-than last-pc ∪ last-pc}} ∪ { *imperfs*>>cf}}
∩ {tessitura}

If cf motion is down (plusp)
{{{*perfs*>> cf} ∩ {higher-than last-pc ∪ last-pc}} ∪{ *imperfs*>>cf}}
∩ { tessitura }

**The Lisp Version**

My code for this is in fz_gradus.lisp.
This is dependent on loading (in order)
pqe-note-events.lisp
fuzzy.lisp
fuzzy_music.lisp
MGC-lite and a saveit routine are also expected.

The constant sets are actually defined for 3 octaves but these give the idea

```
;; the intervals
(defConstant *IMPERFS* '(0 0 0 1 1 0 0 0 1 1 0 0))
(defVar *PERFS* '(1 0 0 0 0 0 0 1 0 0 0 0 ))
```

*perfs* is a variable, because it will need to be changed to do counterpoint below the cantus firmus.

```
;; the allowed pitches
(defConstant *MAJ-TESSITURA* '(1 0 1 0 1 1 0 1 0 1 0 1))   ;phrygian
(defConstant *D-TESSITURA* '(1 0.5 1 0 1 1 0 1 0 1 0 1)) ;dorian
(defConstant *G-TESSITURA* '(1 0 1 0 1 1 0.5 1 0 1 0 1)) ;mixolydian
(defConstant *A-TESSITURA* '(1 0 1 0 1 1 0 1 0.5 1 0 1)) ;aeolian
(defVar *TESSITURA* *d-tessitura*)
```

I use *tessitura* to define both the range and the scale. The major tessitura works for Phrygian, Lydian and Ionian. The others have fuzzy members to give the proper leading tone when necessary.

```
;; note movement constraints
(defConstant *FZ-NOTE2* '(1 0 0 0 0 0 0 0 0 0 0 0 ))
(defConstant *NOT-TT* '(1 1 1 1 1 1 0 1 1 1 1 1))
(defConstant *PREFER-UP* '(1 0.9 0.9 0.8 0.8 0.6 0 0.5 0.4 0.4 0 0))
(defConstant *PREFER-DOWN*'(1 0 0 0.4 0.4 0.5 0 0.6 0.8 0.8 0.9 0.9))
(defConstant *STAY-CLOSE* '(1 0.9 0.9 0.8 0.8 0.6 0 0.5 0.4 0.40 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0  0.4 0.4 0.5 0 0.6 0.8 0.8 0.9 0.9))

(defConstant *CF-MOTION* '(0 0 0.2  0.4  0.6  0.8 1 1 1 1 1 1))
(defConstant *AVOID-P* '(0.2 1 1 1 1 1 1 1 1 1 1 1))
```

Avoid-p is used to discourage unison with the CF or repetitions of a pitch in the counterpoint.

The functions use these variables;
***cantus*** is list of pitches in the cantus firmus
Current pitch of the cantus is **cf**
Beginning pitch is **first-cf**
Previously considered cf is **last-cf**
Pitch following cf is **next-cf**
Final pitch in cantus is **final-cf**
**Penultimate** is one less than the length of the cantus

variable to hold computed pitch is **pc**
previously computed pitch is **last-pc**

List of counterpoint pitches is placed in ***counterpoint***

The underlying rules for this kind of counterpoint might be stated "Within the set of legal pitches, take a short step in the direction the cantus firmus is about to take."

Most of the code is about building a result set of pitch classes which will contain "votes" for each according to how well each possibility follows the rules.

We need to build sets for below the pitch and above the pitch on the fly:

```
(defun C-STYLE-< (x Y)
  "Comparison that returns 1 or 0"
  (if (< x y) 1 0))


(defun MAKE-LOWER-THAN (position &optional (howmany 12) (here 0))
"Returns a list with all members before position set to 1"
  (if (<= howmany 0)
    ()
    (cons (C-style-< here position)
         (MAKE-LOWER-THAN position (1- howmany)
                        (1+ here)))))

(defun C-STYLE-> (x Y)
  "Comparison that returns 1 or 0"
  (if (> x y) 1 0))

;(C-style-> 4 5)

(defun MAKE-HIGHER-THAN ( position &optional (howmany 12) (here 0))
"Returns a list with all members past position set to 1"
  (if (<= howmany 0)
    ()
    (cons (C-style-> here position)
         (MAKE-HIGHER-THAN position (1- howmany)  (1+ here)))))
```

Nothing complicated, these functions just cons up 0s and 1s.

The target interval sets are defined by:

```
(defun LEGAL-FOR-SAME-A (cf last-pc)
  (fz-intersection *tessitura*
              (ror-n (fz-union *imperfs* *perfs*) cf)
              (fz-union (make-higher-than last-pc 32)
                     (make-lower-than last-pc 32))))
```

```
(defun LEGAL-FOR-UP-A (cf last-pc)
  (fz-intersection
   *tessitura*
   (fz-union (ror-n *imperfs* cf)
          (fz-intersection (ror-n *perfs* cf)
               (fz-union (make-lower-than last-pc 32)
                      (ror-n *fz-note2* last-pc))))))

(defun LEGAL-FOR-DOWN-A (cf last-pc)
  (fz-intersection
   *tessitura*
   (fz-union (ror-n *imperfs* cf)
          (fz-intersection (ror-n *perfs* cf)
               (fz-union (make-higher-than last-pc 32)
                      (ror-n *fz-note2* last-pc))))))
```

The tendency of the counterpoint to anticipate the motion of the cantus is based on how far the cantus is about to jump. This uses fuzzy logic inference. The result is just added to a preference for small steps.

```
(defun LIST-ADD (alist blist)
  "Returns a list of the sums of corresponding members of two lists"
    (if(or (null alist) (null blist)) ()
          (cons (+ (first alist) (first blist))
              (list-add (rest alist)(rest blist)))))

(defun GOING-UP (motion last-pc)
  (list-add (fz-rule
          (abs motion) *cf-motion*
          (ror-n *prefer-up* last-pc))
         (ror-n *stay-close* last-pc)))

(defun GOING-DOWN (motion last-pc)
  (list-add (fz-rule
          (abs motion) *cf-motion*
          (ror-n *prefer-down* last-pc))
         (ror-n *stay-close* last-pc)))
```

These are called in one grand function:

```
(defun SPEC-1-ABOVE (cantus last-pc penultimate &optional (index 1))
  (if (= index penultimate) ()
    (let* ((cf (second cantus))
            (motion (- cf (first cantus)))
            (future (- (third cantus) cf))
            (pc)
            (legal-pcs (cond
                      ((zerop motion) (legal-for-same cf last-pc))
                      ((plusp motion) (legal-for-up cf last-pc))
                      (t (legal-for-down cf last-pc)))))
       (cons (setq pc (LTOP (fz-intersection
                           (if (minusp future)
                             (going-down future last-pc)
                             (going-up future last-pc))
                           (ror-n *avoid-p* cf)
                                                (ror-n *avoid-p* last-pc)
                           legal-pcs)))
             (spec-1-above (rest cantus)
                         pc penultimate (1+ index))))))
```

Let's take this apart. This function will terminate with 2 notes to go, since those are special cases. The easy way to do this is use an index for termination.

```
(if (= index penultimate) ()
```

A lot happens in the let*

```
(let* ((cf (second cantus))
            (motion (- cf (first cantus)))
            (future (- (third cantus) cf))
            (pc)
            (legal-pcs (cond
                      ((zerop motion) (legal-for-same cf last-pc))
                      ((plusp motion) (legal-for-up cf last-pc))
                      (t (legal-for-down cf last-pc)))))
```

Note that we need to know the preceding and following pitches of the cantus to calculate motion (how we got here) and future (where the cf will go next). This means that while the initial call to this function includes the entire cantus, the first note is harmonized elsewhere.

The definition of legal-pcs has tripped up college freshmen forever. Fuzzy logic reduces it to three lines of code.

The rest of the function contains the rules that actually shape the piece.

```
(fz-intersection
 (if (minusp future)
         (going-down future last-pc)
         (going-up future last-pc))    ; prefer up or down according to cf
 (ror-n *avoid-p* cf)                             ; but not unison
 (ror-n *avoid-p* last-pc)             ; don't repeat
 legal-pcs)                        ; and of course don't break any rules
```

The result set will be a set of possible pitches (across three octaves) with membership function that describes how each conforms to the rules. To illustrate, here is a short cantus with ltop traced to display the result sets.

```
(setq *cantus*  '(2 4 5 7 5 4 2))

Calling (ltop (0.4 0.4 0 0 0.2 0 0 0.9 0 0 0 0 0.8 0.5 0 0 0 0 0 0.0 0 0 0 0 0.0 0.0 0
0 0 0 0 0.0 0 0 0 0))
 ltop returned 7
 Calling (ltop (0.5 0.0 0.6 0 0 0.2 0 0 0 1 0 0 0 0.0 0.6 0 0 0 0 0 0 0.0 0 0 0 0.0 0.0
0 0 0 0 0 0 0.0 0 0 0))
 ltop returned 9
 Calling (ltop (0 0 0.6 0 0.72 0 0 0.2 0 0 0 0.9 0 0 0 0 0.5 0 0 0 0 0 0 0.0 0 0 0 0 0
0.0 0 0 0 0 0 0 0 0.0))
 ltop returned 11
 Calling (ltop (0 0.0 0.4 0 0 0 0 0 0 0.9 0 0 0.9 0.5 0.8 0 0 0.0 0 0 0 0.0 0 0 0 0.0 0.0
0.0 0 0 0 0.0 0 0 0 0 0.0 0 0))
 ltop returned 9
```

The finished counterpoint was (9 7 9 11 9 13 14). Remember, the first pitch and last two pitches are not handled by the spec-1-above function, so all that was calculated here was the best choices for 4 5 7 5, given 9 as a starting pitch. You will notice that in the final case, there was a tossup with 9 and 12, both having a membership of 0.9. Ltop has a built-in bias to pick the first in case of a tie. This is only a minor problem, as the program will doubtlessly be refined with more rules.

The value returned by Ltop is consed onto the list returned by the recursion.

Finally, a wrapper function deals with the first note and last two notes according to Fux. The first note of the counterpoint pc is either a fifth or

octave above the first note of the cantus firmus. (This is selectable as start in the calling function)
The penultimate note  is a major sixth above cf. (The penultimate note in the cantus must be leading tone or 2nd)
The final note of the counterpoint is an octave above the final note of the cantus firmus.

```
(defun SPECIES-1-ABOVE (cantus start)
  (let* ((penultimate (- (length cantus) 2))
       (first-pc (+ (first cantus) start)))
   (append (list first-pc)
         (spec-1-above cantus first-pc penultimate)
         (list (+ (nth penultimate cantus) 9))
         (list (+ (first(last cantus)) 12)))))
```

This gives a list of pitches that can be easily combined with *cantus* to play the results. Sometimes we want to change octaves first.

```
(defun ADD-OCTAVE (theList &optional (theOctave 4))
     (mapcar #'(lambda (x) (+ x (* theoctave 12)))
             theList))

(setq *events* (string-duet
                          (add-octave *cantus*)
                          (add-octave *counterpoint*) 750))

(saveit)
```

The string-duet function is found in pqe_note-events.lisp.

## Cantus on top

To create counterpoint below a cantus, the legal targets must be redefined:

```
(defun LEGAL-FOR-SAME-B (cf last-pc)
  (fz-intersection *tessitura*
             (ror-n (fz-union *imperfs* *perfs-b*) cf)
             (fz-union (make-higher-than last-pc 36)
                     (make-lower-than last-pc 36))))

(defun LEGAL-FOR-UP-B (cf last-pc)
  (fz-intersection
   *tessitura*
   (fz-union (ror-n *imperfs* cf)
           (fz-intersection (ror-n *perfs-b* cf)
                (fz-union (make-lower-than last-pc 36)
                       (ror-n *fz-note2* last-pc))))))

(defun LEGAL-FOR-DOWN-B (cf last-pc)
  (fz-intersection
   *tessitura*
   (fz-union (ror-n *imperfs* cf)
           (fz-intersection (ror-n *perfs-b* cf)
                (fz-union (make-higher-than last-pc 36)
                       (ror-n *fz-note2* last-pc))))))
```

The counterpoint function is modified to use these new versions:

```
(defun SPEC-1-BELOW (cantus last-pc penultimate &optional (index 1))
 (if (= index penultimate) ()
    (let* ((cf (second cantus))
          (motion (- cf (first cantus)))
          (future (- (third cantus) cf ))
          (pc)
          (legal-pcs (cond
                    ((zerop motion) (legal-for-same-b cf last-pc))
                    ((plusp motion) (legal-for-up-b cf last-pc))
                    (t (legal-for-down-b cf last-pc)))))  ; a lot happens in the let*
      ;(format t "cf ~A motion ~A future ~A~%~A~%" cf motion future legal-pcs)
;diagnostic
      (cons (setq pc (LTOP (fz-intersection
                      (preferred-motion last-pc motion future)
                      (ror-n *avoid-p* cf)
                      (make-lower-than cf 36)
                      (ror-n *avoid-p* last-pc)
                      legal-pcs)))
           (spec-1-below (rest cantus)
                     pc penultimate (1+ index))))))
```

And the wrapper function is only changed slightly.

```
(defun SPECIES-1-BELOW (cantus start)
 (let* ((penultimate (- (length cantus) 2))
      (first-pc (- (first cantus) start)))
  (append (list first-pc)
         (spec-1-below cantus first-pc penultimate)
         (list (- (nth penultimate cantus) 3))
```

```
        (last cantus))))
```