Random Binary Tree Subdivision

A Method for Rhythmic Improvisation

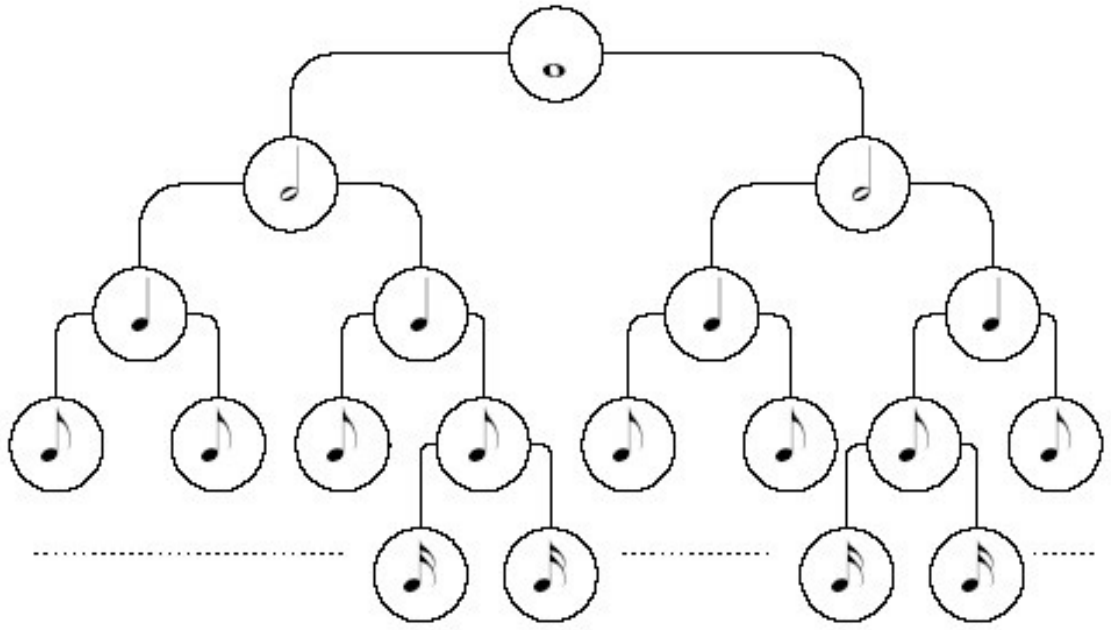
Adam Smith

MUSC 80L

March 17, 2005

Introduction:

In this project, I present a method for rhythmic improvisation based on the idea of modeling rhythms as binary trees. The diagram below shows how a rhythm can be represented as a binary tree. With the restriction that no rests be used and that all notes start and stop on powers-of-two boundaries, we can represent any rhythm in this way. I thought of applying random subdivisions to these trees in order to create a more interesting rhythm that still retained all of the original rhythm and density characteristics as well. Also, if each leaf node stores a pitch, we can use this method for simple melodic improvisation as well.



Algorithms:

I use several algorithms to build the finally improvised binary pitch tree. The most important of which are *subdivide*, *renoodle*, and *eventify*.

Subdivide takes a binary pitch tree represented as a list and recursively locates all of the leaf nodes of the tree. Once at a leaf node it uses a stochastic process involving the depth in the tree to decide if a leaf should be subdivided. If it is chosen for subdivision, the node is replaced with two child nodes, each played for half of the duration of the node they are replacing. The pitch of the old leaf is copied to the new nodes and the algorithm is applied recursively on them. In this manner, even the simplest (single node) tree can be expanded into an intricate rhythm while preserving the intended pitch.

Renoodle takes a binary pitch tree of arbitrary complexity and selects more interesting pitches to play for each node in the tree. It does this by comparing the leaves in order of time. If the node stores the same node as the immediate previous note, it is safe to replace it with a more interesting note, otherwise the note is left the same (to preserve original melodic patterns). To pick a new note, a randomly perturbed offset value is stored, and is added to the old note to create a noodling effect around the

originally intended note. The noodling then restricted to scale notes to avoid too much dissonance.

Finally, *eventify*, works simply by converting a binary pitch tree into cope events with a total duration given by the user. The algorithm works recursively, building sub-sequences of events from the subtrees of the given tree.

Although it is not directly related to the subdivision idea, the velocity for each event is selected using the depth in the tree. This works on the assumption that very quick notes will always be played softer (which is not true in practice). The velocity is *127*e^((- d/4.5)^2)* where *d* is the depth in the tree.

Results:

I tested the program by giving it simple seedling trees to improvise from. I found the program was most effective when it only made minor improvisations to pitched inputs. When told to produce many new improvised notes it resulted in a mumbling effect where the notes were played very fast (and as a result, quietly) mostly rolling around the original pitches but not sounding very satisfying. As a further experiment I tried having the program improvise a long composition from a single node tree. When played with a slightly pitched percussion instrument (the log drum in my case) in a pentatonic scale it produced pleasing results.

The attached CD has the following example tracks:
TRACK 1 – simple improvisations on two 4 node seedlings in C major scale
TRACK 2 – simple improvisations on two more 4 node seedlings in C minor scale
TRACK 3 – extended improvisations on a two single node seedlings in a random pentatonic scale
TRACK 4 – another extended improvisation on two single node seedlings in a random pentatonic scale
TRACK 5 – extremely extended improvisation on two 4 node seedlings in C major scale (a good example of the mumbling effect)

```
;;;; Binary Tree Subdivision-Improvisor
;;;;  by Adam Smith (adam@adamsmith.as)
;;;;
;;;; Generates melodic lines by probabilistically subdividing a binary
;;;; pitch tree in half and noodling over the pitches it describes.
;;;;
;;;; Definition of Binary Pitch Trees:
;;;;  * a single pitch value, played for entire duration
;;;;  * a list of two BPTs, each played for half of the duration
;;;;
;;;; Examples:
;;;;  60              ; play pitch 60 for entire duration
;;;;  (60 62)          ; play 60 for half duration and 62 for other half
;;;;  (60 (65 67)      ; play 60 for half, then 65 and 67 for quarter
;;;;  ((60 62)(60 62)  ; play each pitch for same duration
;;;;
;;;; Important functions: subdivide, renoodle, eventify
;;;;
;;;; 60            --subdivide--> (60 ((60 60) 60))
;;;; (60 (65 67)  --subdivide--> (((60 60) 60) (65 (67 (67 67))))
;;;;
;;;; 60                --renoodle-->  (60)
;;;; ((60 60)(60 60)  --renoodle--> ((60 62)(65 64))
;;;;
;;;; 60            --eventify--> ((0 60 1000 1 127))
;;;; (60 (65 67)  --eventify--> ((0 60 500 1 127)
;;;;                             (500 65 250 100)
;;;;                             (750 65 250 100))
;;;;
;;;; Conclusion:  Computers are too damn casual and creative to play
;;;;              anything but piano.  Only humans can create silly
;;;;              repetitive techno melodies -- the original goal of
;;;;              this program.

;; http://www.peltonium.net/aclmidi/
(load "D:/aclmidi/cormanmidi.lisp") ; load your own here

;; Test if a something is a pitch.
;;  (same behaviour numberp for now)
(defun pitchp (something)
  (numberp something))


;; Calculate a velocity value for a note at given tree depth.
;;  (decreasing, monotonic)
;;
;;  returns: 127*e^((- d/4.5)^2)
(defun velocity-from-depth (d)
  (round (* 127 (exp (- (expt (/ d 4.5) 2))))))


;; Convert a tree of pitches into cope-events.
;;
;;  root: root of pitch tree
;;  duration: length of whole pattern to be played
;;  ontime: initial ontime of the first event
;;  depth: an effective tree depth (used to compute velocity)
;;
;;  returns: list of cope-events
(defun eventify (root duration &optional (ontime 0) (depth 0))
  (if (pitchp root)
    ;; construct an event list for this leaf node
    (list (list (round ontime)
          root
          (round duration)
          1
          (velocity-from-depth depth)))

    ;; sequence the children of this node
    (let ((half-dur (/ duration 2)))
      (append  (eventify  (first root)
```

```
                        half-dur ontime
                        (+ depth 1))
                (eventify (second root)
                        half-dur
                        (+ ontime half-dur)
                        (+ depth 1))))))


;; Subdivide a pitch tree to create an interesting rhythm.
;;  (copies pitches to new subtrees)
;;
;;  root: root of the pitch tree
;;  depth: effective tree depth (influences subdivision decision)
;;
;;  returns: new pitch tree with slick rhythm
(defun subdivide (root &optional (depth 0))
  (if (> depth (random 6)) ; dont be too deterministic

    ;; we chose to leave this node as is
    root

    ;; otherwise we chose to to make it interesting
    (if (pitchp root)

      ;; split a leaf node into new subtrees
      (list  (subdivide root (+ depth 1))
             (subdivide root (+ depth 1)))

      ;; recurse on internal nodes
      (list (subdivide (first root) (+ depth 1))
            (subdivide (second root) (+ depth 1))))))


;; Get the effective octave number of a tree
;;  corresponding to the octave of its first pitch.
;;  (octave 60) => 3
(defun octave (root)
  (if (pitchp root)
    (- (floor (/ root 12)) 2)
    (octave (car root))))


;; Get pitches in the scale close to the given pitch
;;  (by shifting the scale up or down by multiples of 12).
;;
;;  pitch: single pitch in question
;;  scale: list of pitches in the chosen scale
;;
;;  returns: shifted copy of scale
(defun local-scale (pitch scale)
  (let ((difference (- (octave pitch) (octave scale))))
    (mapcar
      #'(lambda(pitch) (+ (* 12 difference) pitch))
      scale)))


;; Find the closes pitch in the choices to the given pitch
;;  (in case of a tie pick the lower pitch in the choices).
;;  Use with local-scale to snap a pitch into a given scale.
;;  Use with several concatenated scales to keep a pitch in
;;  the range playable by an instrument.
;;
;;  pitch: single pitch in question
;;  choices: list of pitches to be chosen from
;;
;;  returns: one of the pitches in choices
(defun closest (pitch choices &optional (contender 0) )
  (if choices
    (if (= pitch (car choices))
      pitch
      (if (<  (abs (- pitch (car choices)))
```

```
          (abs (- pitch contender)))
        (closest pitch (rest choices) (car choices))
        (closest pitch (rest choices) contender)))
    contender))


;; Get a list of notes in the major scale for a given root note.
;;  If not specified give c-major
;;
;;  returns: list of pitches starting with root
(defun scale-maj (&optional (root 60))
  (mapcar
    #'(lambda (offset) (+ root offset))
    '(0 2 4 5 7 9 11)))

;; Get a list of notes in some minor scale for a given root note.
;;  If not specified give c-minor something.
;;
;;  returns: list of pitches starting with root
(defun scale-min (&optional (root 60))
  (mapcar
    #'(lambda (offset) (+ root offset))
    '(0 2 3 5 7 8 10)))


;; Find a more interesting pitch to play than the given one.
;;  Keeps improvising as long as you pass it to play the same pitch.
;;
;;  pitch: the pitch to be reinterpreted
;;  scale: a scale to snap to (allows any number transpositions)
;;  range: maximum noodling interval from pitch
;;
;;  returns: a slicker pitch
(let ((offset 0)(last 0))
  (defun nood (pitch scale range)
    ;; wiggle offset within +/- range by up to +/- 2 each step
    (setf offset
      (min  (max  (+ offset (- (random 5) 2))
                  (- range))
            range))

    ;; snap to original pitch when its different than the last one
    (when  (/= last pitch)
          (setf last pitch)
          (setf offset 0))

    ;; return the closes note in the scale to the one selected
    (closest  (+ pitch offset)
              (local-scale (+ pitch offset) scale))))


;; Reinterpret a given pitch tree by improvising over each pitch.
;;  Rebuilds the tree using the return from nood as the new pitch.
;;
;;  root: root of pitch tree
;;  scale: passed to nood for snapping
;;  range: passed to nood for clamping
;;
;;  returns: pitch tree with same rhythm and slicker pitches
(defun renoodle (root scale &optional (range 8))
  (if (pitchp root)
    (nood root scale range)
    (list  (renoodle (first root) scale range)
          (renoodle (second root) scale range))))


;;;;
;;;; An example composition using BPT subdivision-improvisation.
;;;;

;; define some basic pitch trees
```

```
;; cooresponding to a decent chord progression
;; (negative depths encourage more subdividing)
(setf *meltree* (subdivide '((60 62) (64 65)) -3))
(setf *hartree* (subdivide '((50 60) (55 53)) -4))

;; pick a nice scale to snap to (c-major)
(setf s (scale-maj))

;; save this composition in case its really cool
(setf *lastsong*
  (transpose 24 ; make it sound nice on a piano
    (change-tempo 8 ; slow down the notes
      (append
        ;; unwavering bass notes
        (transpose -24 (eventify (renoodle *hartree* s 5) 4000))
        ;; noodling bass notes (to make some random chords)
        (transpose -36 (eventify (renoodle *hartree* s 0) 4000))

        ;; noodling melody using same rhythm for each part
        ;; repeated 4 times to give some structure
        (eventify (renoodle *meltree* s) 1000 0000)
        (eventify (renoodle *meltree* s) 1000 1000)
        (eventify (renoodle *meltree* s) 1000 2000)
        (eventify (renoodle *meltree* s) 1000 3000)
      )
    )
  )
)

;; a hack to stick the opening chord at the end (pretending to loop)
(setf *lastsong*
  (append
    *lastsong*
    (remove nil
      (mapcar
        #'(lambda (cope-event)
          (if (zerop (first cope-event))
            (list 32000  ; should be right after the end
                  ; of the composition
                  (second cope-event)
                  1000  ; really long
                  (fourth cope-event)
                  5) ; really soft
            nil))
        *lastsong*))))

;(play-dev-events 2 *lastsong*)  ; device 2 is Reason for me, uncomment for love

; vim: tabstop=2
```