INTRODUCTION TO COMPUTER MUSIC

SPECTRAL CENTROID

An estimate of brightness

Roger B. Dannenberg
Professor of Computer Science, Art, and Music

Project 3

- Goal: Use spectral centroid to control FM synthesis parameters
- What's a spectral centroid?
- Example code

Discrete Fourier Transform

$$R_k = \sum_{i=0}^{N-1} x_i \cos(2\pi ki / N)$$

$$X_k = -\sum_{i=0}^{N-1} x_i \sin(2\pi ki / N)$$

How to Interpret a Discrete Spectrum

- These points X_k and R_k are evenly (linearly) spaced in frequency.
- Point $R_{N/2}$ is at SR / 2.
- Points X_k and R_k are at (k / (N/2)) * (SR / 2) = k * SR / N Hz.
- Frequency spacing (width of "bins") is SR / N Hz – the "bin width"
- Example: SR=44100 Hz, FFT size = 1024 points, bin size = 44100/1024 = 43.0664 Hz
- FFT takes in N samples and outputs N *values*
- This must be because FFT and Inverse FFT preserve information: N-dimensions in, N-dimensions out
- The output values are:
  - R_0 – the "DC" component
  - X_0 – always zero, not in output
  - R_1, X_1, R_2, X_2, … $R_{N/2-1}$, $X_{N/2-1}$
  - $R_{N/2}$ – the "Nyquist" component
  - $X_{N/2}$ – always zero, not in output
- Note there are N points as expected

Discrete Magnitude (or Amplitude) Spectrum

- Magnitude A_k = sqrt($R_k^2 + X_k^2$)
- The magnitude spectrum is:
  - A_0 , A_1 , … , $A_{N/2}$
- Note there are $N/2+1$ points.
- How can this be? There are only $N/2-1$ non-zero phases, so we still have N total dimensions.

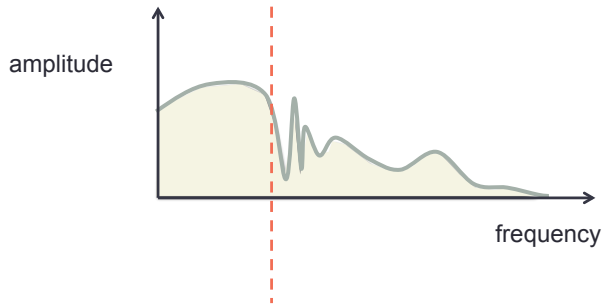
Spectral Centroid

- Weighted average of the magnitude (amplitude) spectrum:

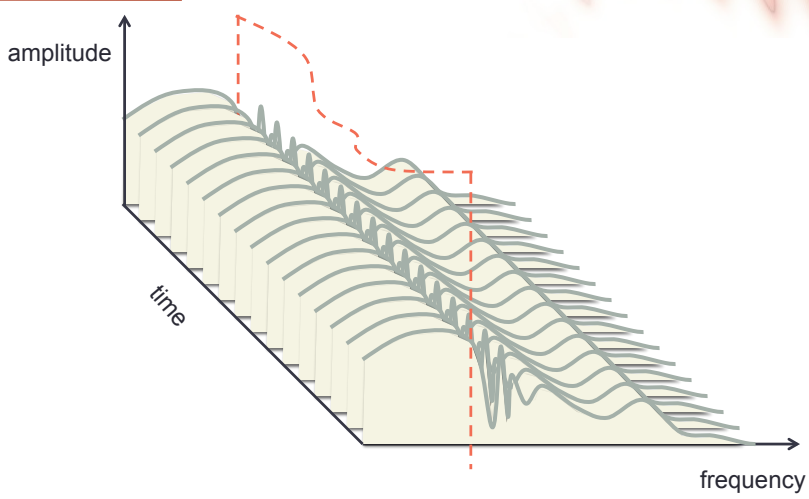
$$spectral\ centroid = \frac{\sum_{i=0}^{N} i \cdot w \cdot A_i}{\sum_{i=0}^{N} A_i}$$

- w is the width of each spectral bin in Hz
- w = sample rate / size of the FFT in samples

Spectral Centroid



Time-Varying Spectral Centroid



Review Project 3 Code Examples

ALGORITHMIC COMPOSITION

Introducing the score-gen construct

Programs and Data

- We've seen:
  - Ordinary programs:
    - pwl(…) * osc(…)
  - Score-like programs:
    - sim(note(…) ~ 2 @ 0, note(…) ~ 3 @ 1, …)
  - Scores:
    - ```
      {{0 2 {note …}}
       {1 3 {note …}}
       …}
      ```


Lots of Choices

- Data and programs have different properties
- Data and programs can work together:
  - Programs create data (scores)
  - **`timed-seq`** interprets scores to invoke functions
  - Programs can even create (Lisp) programs
- No right/wrong answers
- Today, we look at programs creating scores


      ```

# The score-gen Macro: Introduction

- The problem:
  - Create a score of notes
  - Specify attribute values with SAL expressions (evaluated for each note)
  - Flexible expression of start time, inter-onset time, or duration
- The solution: score-gen
- Alternative: build scores with list primitives
  - (been there, done that in Project 2 – was it fun?)

# score-gen

```
score-gen(attribute: expression,
          attribute: expression,
          attribute: expression,
          …)
score-gen(score-len: 2, pitch: 60,
          vel: 100, ioi: 0.7,
          name: quote(note))
```

score-gen Loop Variables

- sg:start – starting time for current note
- sg:ioi – current inter-onset interval
- sg:dur – current duration
- sg:count – how many notes computed so far
- Example:
  - `score-gen(score-len: 10, ioi: 0.2, pitch: c4 + sg:count)`

PATTERN GENERATORS

Designing sequences of parameters

Introduction to Pattern Generating Objects

- Score-gen is convenient, but …
- … hard problem is generating attribute values
- Pattern Objects encapsulate many selection/ sequencing algorithms for data
- Some are composable – very flexible, powerful
- Examples:
  - sequential selection from list,
  - random selection from a set,
  - output 3 copies of every set of 5 input values

Using Pattern Objects

```
begin
  with pat = make-cycle({60 62 64 65})
  exec score-gen(save: quote(simple),
                 score-len: 20,
                 ioi: 0.3,
                 pitch: next(pat))
end

exec play-score(simple)
```

MULTIPLE PATTERNS

Another score-gen example

make-heap

- `make-heap(`*`list-or-pattern`*`,`
  `for:` *`number-or-pattern`*`,`
  `max:` *`number-or-pattern`*`)`
- Output elements of input list in *random order*
- Length of period given by `for:`
  - Default is the length of the input list
- If max: is 1, do not repeat elements in output
- Every period:
  - Update any pattern inputs

Two Pattern Objects

```
begin
  with pitch-pat = make-heap(
          list(c4, cs5, e4, f4, a4, bf4), max: 1),
       rhythm-pat = make-cycle(
          list(s, s, i, s, i, s))
  exec score-gen(save: quote(test1),
                 score-len: 50,
                 ioi: next(rhythm-pat),
                 pitch: next(pitch-pat))
end
```

MORE PATTERNS

make-palindrome

- `make-palindrome(`*`list-or-pattern`*`,`
  `for:` *`number-or-pattern`*`,`
  `elide:` *`keyword`*`)`
- Output elements of input list forward then reverse order
- Length of period given by `for:`
  - Default is full forward-backward traversal
- Elide:
  - :first – A,B,C becomes A,B,C,C,B,A,B,C,C,B,... (elide the final A)
  - :last – A,B,C becomes A,B,C,B,A,A,B,C,B,A,...
    (elide the duplicate of C)
  - #t – A,B,C becomes A,B,C,B,A,B,C,B,... (elide first and last)
  - #f – A,B,C becomes A,B,C,C,B,A,A,B,C,C,B,A,... (no elision)
- Every period:
  - Update any pattern inputs

Palindrome Example 1

```
begin
  with pitch-pat = make-palindrome(
                       list(c4, f4, bf4, ef5, af5))
  exec score-gen(save: quote(palindrome-1),
                 score-len: 20,
                 ioi: 0.3,
                 pitch: next(pitch-pat))
end
```

Palindrome Example 2

```
begin
  with pitch-pat = make-palindrome(
                       list(c4, f4, bf4, ef5, af5),
                       elide: #t)
  exec score-gen(save: quote(palindrome-2),
                 score-len: 25,
                 ioi: 0.3,
                 pitch: next(pitch-pat))
end
```


More Simple Patterns

- `make-random(`*`list-or-pattern`*`, for:` *`number-or-pattern`*`)`
  - Select items from list at random
  - Fancy list elements: { value weight: 5 min: 3 max: 5 }
- `make-line(`*`list-or-pattern`*`, for:` *`number-or-pattern`*`)`
  - Output elements of list, repeating last element forever
- `make-accumulation(`*`list-or-pattern`*`, for:` *`number-or-pattern`*`)`
  - Output initial substrings
  - {a b c} → a a b a b c, a a b a b c, …
- `make-markov` – maybe later or see manual




Pattern Periods

- Pattern object output is structured into *periods*
- next(*pattern*) returns one element
- next(*pattern,* #t) returns list of one full period
- next(make-cycle({1 2 3}), #t) → {1 2 3}
- Why periods?
  - Sometimes patterns do something *every period.*

make-cycle

- `make-cycle(`*`list-or-pattern`*`,`
  `for:` *`number-or-pattern`*`)`
- Output elements of input list in sequence
- Length of period given by `for:`
  - Default is the length of the input list
- Every period:
  - Update list-or-pattern to next period
  - Update number-or-pattern to next value

Patterns of Patterns - 1

- `make-accumulate(`*`pattern`*`,`
  `max:` *`expr`*`, min:` *`expr`*`,`
  `for:` *`number-or-pattern`*`)`
  - Sum successive elements from input pattern
- `make-copier(`*`pat`*`, repeat:` *`expr-or-pat`*`,`
  `merge:` *`boolean`*`,`
  `for:` *`number-or-pattern`*`)`
  - Copy each period *repeat* times,
  - merge to one period if *merge* is true

Accumulate Example

```
begin
  with pitch-incr-pat = make-random(
           list(-3, -2, -1, +1, +2, +3)),
       pitch-pat = make-accumulate(pitch-incr-pat)
  exec score-gen(save: quote(accumulate-1),
                 score-len: 25,
                 ioi: 0.2,
                 pitch: 60 + next(pitch-pat))
end
```

Copier Example

```
begin
  with pitch-heap-pat = make-heap(
           list(c4, cs5, e4, f4, a4, bf4)),
       pitch-pat = make-copier(pitch-heap-pat,
                               repeat: 4)
  exec score-gen(save: quote(copier-1),
                 score-len: 4 * 6 * 3,
                 ioi: 0.15,
                 pitch: next(pitch-pat))
end
```

Patterns of Patterns - 2

- `make-length(`*`list-or-pattern, number-or-pattern`*`)`
  - Regroup input sequence to specified period lengths
- `make-window(`*`pattern, window-size, window-skip`*`)`
  - Output *window-size* elements,
  - then advance *window-skip*
  - make-cycle({a b c d}), 3, 1 → a b c b c d c d a d a b a b c …

Window Example

```
begin
  with pitch-line-pat = make-line(
           list(c4, cs4, e4, f4, a4, bf4)),
       pitch-pat = make-window(pitch-line-pat, 3, 1)
  exec score-gen(save: quote(window-1),
                 score-len: 17,
                 ioi: 0.2,
                 pitch: next(pitch-pat))
end
```

```
make-window(pitch-pat, 9, 3)
```

NESTED PATTERNS

Nested Patterns

- When a pattern generator accesses a list or other parameter and gets a pattern object, it uses the next value generated by the pattern object.

```
begin
  with pitch-pat = make-heap(list(c4, cs5, e4, f4, a4, bf4),
                             max: 1),
       phrase-pat = make-heap(
           list(make-cycle(list(s, s, s, s, q)),
                make-cycle(list(q, s, s, s, s)),
                make-cycle(list(q, q))))
  exec score-gen(save: quote(test2),
                 score-len: 50,
                 ioi: next(phrase-pat),
                 pitch: next(pitch-pat))
end
```

Summary

- SCORE-GEN is a special loop structure that evaluates expressions and inserts values into scores.
- Pattern objects are like unit generators, but they generate streams of numbers rather than audio.
- Pattern objects can serve as parameters to other pattern objects, enabling very complex behaviors on multiple time scales.

MORE ALGORITHMIC COMPOSITION

Probability Distributions, Random Walks, Grids, Masks, and more

Types of Machine-Aided Composition

- Completely directed by composer
  - Notation packages
  - Cut and Paste
  - Editing macros
- Algorithmic Compositions
  - Procedures + random numbers
- Artificial Intelligence
  - Music models
  - Models of composition
  - Machine learning
  - Search

Types of Machine-Aided Composition

- Completely directed by composer
  - Notation packages
  - Cut and Paste
  - Editing macros
- Algorithmic Compositions
  - Procedures + random numbers
- Artificial Intelligence
  - Music models
  - Models of composition
  - Machine learning
  - Search

Greatest "High" Culture Impact

Types of Machine-Aided Composition

- Completely directed by composer
  - Notation packages
  - Cut and Paste
  - Editing macros
- Algorithmic Compositions
  - Procedures + random numbers
- Artificial Intelligence
  - Music models
  - Models of composition
  - Machine learning
  - Search

Greatest "High" Culture Impact

Increasing "Pop" Culture Impact

Techniques – Tricks of the Trade

- Rhythm using Negative Exponential distribution
- Melody using random walk
- Markov algorithm
- Rhythmic pattern generation
- Melodic transformations & serialism
- Fractals
- Grammars
- Pitch and Rhythm grids
- Tendency masks

Scores and Score Manipulation

- SCORE-BEGIN-END is not synthesized:
  (score-begin-end
  <start-time>
  <end-time>)
- Pitch lists are expanded as chords

```
set myscore =
 {{0 1 {score-begin-end 0 2}}
  {0 1 {tpt :pitch {64 67 72} :vel 100}}
  {1 1 {tbn :pitch 48 :vel 80}}}

function tpt(pitch: 60, vel: 100)
   return trumpet(pitch, vel)

eval score-play(myscore)
```


Scores and Score Manipulation (2)

score-shift(score, offset)
score-stretch(score, factor)
score-transpose(score, keyword, amount)
score-scale(score, keyword, amount)
score-sustain(score, factor)
score-voice(score, replacement-list)
score-merge(score1, score2, …)
score-adjacent-events(score, function)
score-apply(score, function)
score-stretch-to-length(score, length)

score-append(score1, score2, …)
score-select(score, predicate)
score-filter-length(score, cutoff)
score-repeat(score, n)
score-filter-overlap(score)
score-print(score)
score-play(score)
score-last-index-of(score, function)
score-randomize-start(score amt)
score-sort(score, [copy-flag])

Scores and Score Manipulation (3)

- All score functions take some optional keyword parameters:
  - :from-index *i*
  - :to-index *i*
  - :from-time *seconds*
  - :to-time *seconds*
- Score functions construct new scores
- Standard MIDI File I/O:
  - score-read-smf(*filename*)
  - score-write-smf(*score, filename*)

Workspaces

- How do you save score data?

```
set my-score = {… score data …}
set new-score = score-transpose(my-score,
                                :pitch 3)
exec add-to-workspace(quote(my-score))
exec describe(my-score, "original data")
exec add-to-workspace(quote(new-score))
exec describe(new-score, "transposed version")
exec save-workspace()
```

- Later, you can just load workspace.lsp to restore everything. The variable names are in `*workspace*`.

The Negative Exponential Distribution

- "Random" is interesting(!)
- What does it mean to be random in time?
  - Uniform random interval between events?
  - Gaussian?
  - Some other distribution?
- Examples from real world:
  - Atomic decay
  - Sequence of uncorrelated events (yellow cars driving by)

Negative Exponential Distribution (2)

- The inter-arrival time has a negative exponential distribution: longer and longer intervals are less and less likely
- Equivalently: in each very small interval of time, generate an event with some small probability P = density * interval duration
- Equivalently: generate events at times that are uniformly random across total duration.

```
score-gen(score-len: 50,
          time: real-random(0, 12.5),
          dur: 1,
          name: quote(s-pop-kwp),
          lo: 800, hi: 1200)
```


Probability distributions in Nyquist

- load “distributions”
- See Distributions, probability in Nyquist index

```
begin
  with ne-score
  loop
    for i below 30
    for now = 0 then now + exponential-dist(1.0)
    set ne-score @= list(now, 1, {s-pop 800 1200})
  end
  return reverse(ne-score)
end
```

score is in reverse order, so rather than sorting, we can just reverse it

RANDOM WALK, MARKOV, AND RHYTHMIC PATTERNS

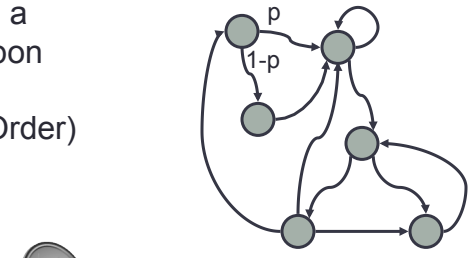
Melody Using Random Walk

- What kinds of pitches create interesting melody?
- Uniform random pitch has too many large intervals.
- Lots of small intervals is more typical.
- Melodies are said to have fractal properties.

```
begin
  with small-pat = make-random(
        {0 0 -1 -1 -1 -1 -2 -2 -2 -2 -3 -3 -3 -4 -4
         -5 -5 -6 1 1 1 1 2 2 2 2 3 3 3 4 4 5 5 6}),
       pitch-pat = make-accumulate(
        make-line(list(57, small-pat)))
end
```

Markov Algorithm

- Generate sequence of "states"
- Probability of being in a state depends only upon probability of being in previous state (First Order)
- Or previous 2 states (Second Order)
- Etc.
- See example code



Rhythmic Pattern Generation

- Of course there are many techniques; here's one:
  - Generate a sequence length from some probability distribution or just by your choice
  - Generate a random number with that number of bits, e.g. length N → (random 2^N)
  - Translate 0 to rest, 1 to event

Rhythmic Pattern Example

```
function rhythm-pattern-demo(n, freq)
  begin
    with pat = make-cycle(list(random(2), random(2),
                  random(2), random(2), random(2),
                  random(2), random(2), random(2)))
    return score-gen(score-len: n,
                ioi: 0.15,
                name: quote(s-pop-kwp),
                hi: freq, lo: freq,
                pitch: #?(next(pat) = 1, c4, nil))
  end
```

Three instances of this function, each with different resonance frequency:

SERIALISM, FRACTALS, GRAMMARS AND GRIDS

Algorithmic Composition

Serialism

- Arnold Schoenberg and *Serialism*
- Chromatic scale – 12 notes/octave with equal ratios between (half)steps
- Pitch – an element of the chromatic scale
- Pitch class – pitch mod 12, e.g. "C-sharp" without regard to octave
- Tone row – permutation of the 12 pitch classes
- Music based on tone rows can be *atonal*

Melodic/Tone Row Transformation

- Original: p[i]
- Transposition: T(p[i],c) = (p[i] + c) mod 12
- Inversion: I(p[i]) = (-p[i]) mod 12
- Retrograde: R(p[i]) = p[12 – i]
- Also: (p[i]*5) mod 12 = I(p[i]*7)

Why serialism?

- In general, listeners cannot hear retrograde and/ or inversion relationships
- Intervals are preserved
- Tone "row-ness" is preserved
- "Denial of tonality" produces new textures

Fractals and Nature

- Melodic contours are often fractal-like
- Composers often use fractal curves to generate music data
- Examples:
  - Austin, Canadian Coastline
  - Cage, Atlas Eclipticalis

Grammars

melody ::= intro middle ending
middle ::= phrase | middle phrase
phrase ::= sequence-a | sequence-b

```
function melody()
 return seq(intro(), middle(), ending())
function middle() return #?(random() < 0.5,
                   phrase(), seq(middle(), phrase()))
function phrase() return #?(random() < 0.3,
                   sequence-a(), sequence-b())
```


Pitch and Rhythm Grids

- Quantize random numbers to scales, grids

```
define function pitch-filter(p, f)
  return #?(member(p % 12, f),
            p, pitch-filter(p + 1, f))

define function grid-scale(grid-function)
  return score-gen(score-len: 50,
    ioi: 0.15,
    pitch: funcall(grid-function, 60 + random(12)),
    name: quote(pluck-kwp))

define function c-major(p)
  return pitch-filter(p, {0 2 4 5 7 9 11})
```


Quantizing to Rhythmic Grid

```
function on-beat(time, beat-len)
  begin with beats = round(time / float(beat-len))
        return beats * beat-len
  end

define function grid-rhythm(grid-function)
  return score-gen(score-len: 100,
     time: on-beat(real-random(0, 15), 0.15),
     pitch: funcall(grid-function, 60 + random(12)),
     dur: 1.0,
     name: quote(pluck-kwp))

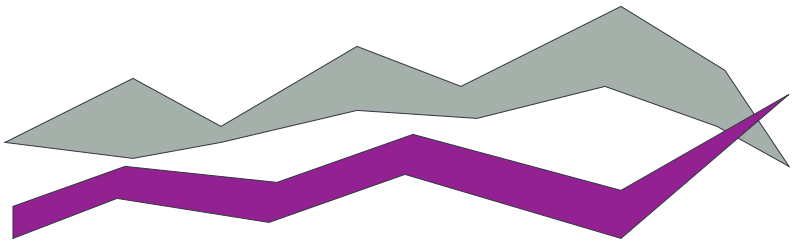
exec score-play(grid-rhythm(quote(c-major)))
```

TENDENCY MASKS

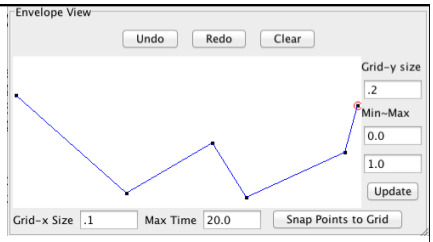
Directly controlling shape at the macro level

Tendency Masks

- A problem with algorithmic composition is that things can get static
- Manual parametric change allows composition of global trends:



Tendency Function Example




```
set dur-envelope = pwlv(…)

function follow-the-envelope()
  begin
    with pitch-pat = make-cycle(list(g4, a4))
    return score-gen(score-dur: 20,
              ; here is how to access a signal
              ; at some time:
              ioi: sref(dur-envelope, sg:start) +
                   real-random(-0.01, 0.01)
              pitch: next(pitch-pat))
  end

exec score-play(follow-the-envelope())
```

Summary

- Score manipulation functions
- Negative Exponential and Rhythm/Timing
- Probability Distributions
- Random Walk Melodies
- Repetition Creates Rhythm
- Serialism & 12-Tone Music & Atonal Music
- Quantization for scales and beats
- Tendency masks for long-term form