5_cac

July 13, 2025

Index

0.1 Software installation

Download and install:

- sc_kernel (if you want to execute supercollider code in this notebook)
- SuperCollider
- Lilypond
- Fosc (a Supercollider API for generating musical notation in lilypond)

0.1.1 Score

Configure Fosc:

1. Download the zip file
2. Rename the 'fosc-master' folder to 'fosc'Rename
3. (Select SuperCollider (base) Kernel in this notebook)
4. Find the your SuperCollider user extension hidden folder path:

[ ]:
```
Platform.userExtensionDir
```

5. Move the 'fosc' folder to your SuperCollider User Extensions folder.
6. Search in SuperCollider Qarks 'wslib' and installi it.
7. Recompile Class library in SuperCollider or Refresh SuperCollider Kernel in Notebook.
8. Add code to allow Fosc to communicate with LilyPond to your sclang startup file.

Mac users (run this cell):

[ ]:
```
s.boot;

~lilypath="/Applications/lilypond-2.24.4/bin"; // Lilypond path
                                    // Filename without extension
~filepath=Platform.userHomeDir++"/Desktop/GHub/EMC/7_cac/score/test";

Fosc.lilypondPath = ~lilypath++"/lilypond";
Fosc.lilypondVersion;

~print = {arg pitch, dur=[1/4], vel=[nil], tsig=[4,4], bpm=60,
          filepath=Platform.userHomeDir++"/Desktop/GHub/EMC/7_cac/score/test";
          var staff, mus, ts, seq, dn;
```

```
Routine{
        dur = dur.collect{arg i;
                    if(i.isSequenceableCollection){i[0]*i[1].normalizeSum}{i}
                          }.flat;
        mus = FoscLeafMaker().(pitches:pitch, durations:dur);
        dn = #['!','ppppp','pppp','ppp','pp','p','mp','mf',
               'f','ff','fff','ffff','fffff'];
        mus.selectLeaves().do({arg item, id;
                               item.attach(FoscDynamic(
                                                   i = vel.wrapAt(id);
                                                   case
                                                   {i==nil}{dn[0]}
                                                   {i>=1 && i<=9}{dn[1]}
                                                   {i>=10 && i<=19}{dn[2]}
                                                   {i>=20 && i<=29}{dn[3]}
                                                   {i>=30 && i<=39}{dn[4]}
                                                   {i>=40 && i<=49}{dn[5]}
                                                   {i>=50 && i<=59}{dn[6]}
                                                   {i>=60 && i<=69}{dn[7]}
                                                   {i>=70 && i<=79}{dn[8]}
                                                   {i>=80 && i<=89}{dn[9]}
                                                   {i>=90 && i<=99}{dn[10]}
                                                   {i>=100 && i<=109}{dn[11]}
                                                   {i>=110}{dn[12]}
                                                   ))
                                            });
          staff = FoscStaff(mus);
          ts    = FoscTimeSignature(tsig);
          staff[0].attach(ts);
          staff.writePDF(filepath++".ly");

          seq = Pbind(
                          \midinote, Pseq(pitch.replace([nil],\rest)
                                          if(dur.size > pitch.size){inf}{1}),
                          \dur,      Pseq(dur * tsig[1],
                                          if(dur.size>pitch.size){1}{inf}),
                          \amp,      Pseq((vel.replace([nil],100)/127)**2,inf),
                          );
          ~midi = SimpleMIDIFile.new( filepath++'.mid' );
          ~midi.init;
          ~midi.init0(bpm,
                           tsig[0].asString++"/"++tsig[1].asString
                          );
          ~midi.timeMode = \ticks;
          ~midi.fromPattern(seq);
          ~midi.checkWrite(filepath++'.mid', true);
          1.wait;
```

```
            ~midi.play(TempoClock(bpm/60));
          }.play;
        };
```

```
-> a Function
```

10. Test it - generates three files:
    - test.ly
    - test.pdf
    - test.mid

    We should also hear an audio preview with default intrumental timbre.

We can load the midifile in an external midi sequencer or notation software as Musescore, Finale, Sibelius, etc.

[713]:
```
~bpm   = 60;
~tsig  = [3,4];
~pitch = [60,64,67,nil,67,64,72,[67,64],nil,64,60,[64,67,72]];
~durs  = [1/8];
~vels  = [nil];

~print.value(~pitch, dur:~durs, vel:~vels, tsig:~tsig, bpm:~bpm);
```

```
-> a Routine
```

Windows users (run this cell):

[ ]:
```
s.boot;

~lilypath=Platform.userHomeDir++"/Desktop/lilypond-2.24.4/bin";// Lilypond path
~filepath=Platform.userHomeDir++"/Desktop/test";   // Filename without extension

Fosc.lilypondPath = ~lilypath++"/lilypond.exe";
Fosc.lilypondVersion;

~print = {arg pitch, dur=[1/4], vel=[nil], tsig=[4,4], bpm=60,
              filepath=Platform.userHomeDir++"/Desktop/test";
          var staff, mus, ts, seq, dn;
          Routine{
            dur = dur.collect{arg i;
                              if(i.isSequenceableCollection){i[0]*i[1].
 ↪normalizeSum}{i}
                              }.flat;
            mus = FoscLeafMaker().(pitches:pitch, durations:dur);
            dn = #['!','ppppp','pppp','ppp','pp','p','mp','mf',
                   'f','ff','fff','ffff','fffff'];
            mus.selectLeaves(pitched:true).do({arg item, id;
                              item.attach(FoscDynamic(
                                                  i = vel.wrapAt(id);
```

```
                                                    case
                                                    {i==nil}{dn[0]}
                                                    {i==0}{dn[0]}
                                                    {i>=1 && i<=9}{dn[1]}
                                                    {i>=10 && i<=19}{dn[2]}
                                                    {i>=20 && i<=29}{dn[3]}
                                                    {i>=30 && i<=39}{dn[4]}
                                                    {i>=40 && i<=49}{dn[5]}
                                                    {i>=50 && i<=59}{dn[6]}
                                                    {i>=60 && i<=69}{dn[7]}
                                                    {i>=70 && i<=79}{dn[8]}
                                                    {i>=80 && i<=89}{dn[9]}
                                                    {i>=90 && i<=99}{dn[10]}
                                                    {i>=100 && i<=109}{dn[11]}
                                                    {i>=110}{dn[12]}
                                                    ))
                                                });
            staff = FoscStaff(mus);
            ts    = FoscTimeSignature(tsig);
            staff[0].attach(ts);

            staff.writeLY(filepath++".ly");
("cd" + ~lilypath + "&& lilypond -o" + filepath + filepath ++ ".ly").unixCmd;

            seq = Pbind(
                            \midinote, Pseq(pitch.replace([nil],\rest),
                                                    if(dur.size>pitch.
 ↪size){inf}{1}),
                            \dur,      Pseq(dur * tsig[1],
                                                    if(dur.size>pitch.
 ↪size){1}{inf}),
                        \amp,      Pseq((vel.replace([nil],100)/127)**2,inf),
                        );
            ~midi = SimpleMIDIFile.new( filepath++'.mid' );
            ~midi.init;
            ~midi.init0(bpm,
                             tsig[0].asString++"/"++tsig[1].asString
                            );
            ~midi.timeMode = \ticks;
            ~midi.fromPattern(seq);
            ~midi.checkWrite(filepath++'.mid', true);
            1.wait;
            ~midi.play(TempoClock(bpm/60));
          }.play;
        };
```

```
~bpm   = 60;
~tsig  = [3,4];
~pitch = [60,64,67,nil,67,64,72,[67,64],nil,64,60,[64,67,72]];
~durs  = [1/8];
~vels  = [nil];

~print.value(~pitch, dur:~durs, vel:~vels, tsig:~tsig, bpm:~bpm);
```

0.2 Symbolic musical representation and numbers

The purpose of this computer music practice is to help the composer generate and manipulate musical elements according to the syntactic rules of their chosen musical language.

One of the goals of computer-aided composition is to translate numerical values into symbols specific to musical notation and vice versa.

The final result typically consists of a musical score that can be interpreted by human performers.

In computer music as well as in Western musical tradition the following sound parameters are typically generated and/or manipulated:

- Pitch (frequency)
- Rhythm (time)
- Dynamic (amplitude)
- Expression (timbre)

0.2.1 Pitch

Some symbolic representations of frequency sorted by level of abstraction:

- Note
  - diatonic or chromatic representation by letters (notenames).
  - fixed octave.
  - relative to syntactically defined rules (western music tones, semitones).
  - typically strings or chars data types.

```
a = ['do', 're', 'mi', 'fa', 'sol', 'la', 'si', 'do']; // Note name
a = ['C3', 'D3', 'E3', 'F3',  'G3', 'A3', 'B3', 'C4']; // Octave
```

- Degree
  - determine the pitch in degrees relative to a root note → 0.
  - no fixed octave
  - they may or may not be related to defined syntactic rules (scale degrees or other).
  - typically int (positive and negative).

```
a = [ 0, 2, 4, 5, 7, 9, 11, 12]; // Major scale model
a = [ -3, 5, 6, -4, 0, 2];       // A sequence
```

- MIDI note
  - determines pitch as a fractional MIDI note.
  - they are not related to syntactically defined rules.
  - root note → 60 = C4 = middle C.

  - divided in semitones (int).

[ ]:
```
a = [ 60, 62, 64, 65, 67, 69, 71, 72];
```

- Frequency
  - determines the pitch as a frequency in Hertz (or cps).
  - absolute values not relate in any way to syntactically defined rules.
  - int or float

[ ]:
```
a = [ 262, 294, 330, 349, 392, 440, 494, 523];
```

In computer music there are further forms of pitch representation but they can easily be traced back to those just mentioned.

In SuperCollider we can convert from a form to another:

[ ]:
```
~degree    = [0, 2, 4, 5, 7];   // Degree
~root      = 60;
~midinote  = root + a;          // Midi note
~freqency  = ~midinote.midicps; // Frequency
~midinote1 = ~freqency.cpsmidi;
~degree1   = ~midinote1 - 60;

~midinote.postln;               // change
```

Chords → 2D arrays.

[109]:
```
a = [[60,64,67,72],62,[60,65,69,72],64,[62,65,67,71],62,[60,64,67,72]];
b = 1/4!6 ++ [1/2];

~print.value(a, b);
```

```
-> a Routine
```

Rest → nil instead notemidi number.

[110]:
```
a = [60,62,64,nil,65,67,nil,74,73,74,75,nil,76,72,nil,64];
b = [1/16];

~print.value(a, b);
```

```
-> a Routine
```

0.2.2 Rhythm

We can define any kind of musical event in time choosing at least two of these parameters:

- Onsets → absolute time of a sound event relative to the beginning (as in DAW software).
- Delta times → time between two consecutive sound events.
- Durations → duration of a sound event.

All these parameters can be specified as:

- absolute measurement units (typicalli seconds or millisecond).
- relative to a bpm.
  - 1 → beat, reference unit (could be quarter, eight, whole, etc).
  - 1/4 → fractional notation.
  - 0.5 → multiply factor (result from fractional notation).
  - 3:2 → Rhythmic proportions (ratios of one single beat or its subdivisions).

We can convert the values by simple mathematical operations.

[ ]:
```
~bpm = 82;

~sec = 60/~bpm;  // bpm     --> seconds
~bpm = 60/~sec;  // seconds --> bpm

~sec;     // change
```

In this notebook we adopt the fractional notation as input (but inside it's converted to multiply facors of unit).

[112]:
```
a = [72,76];
b = [1/16, 1/16, 1/8, 1/16, 1/16, 1/16, 1/16, 1/4,1/4];

~print.value(a, b);
```

```
-> a Routine
```

If we want to define dotted or irregular rhythms we can use thi syntax:

[ ]:
```
//    [unity, [subdivisions] ]
a = [ [1/4,   [3, 1        ] ] ];
```

We ca mix regular and irregular/dotted rhythms.

[113]:
```
a = [60,        nil,62,          63,64,nil,[62,66],67,    68,  [64,72]];
b = [1/4, [1/4,[3,   1]], [1/4, [1,  1,  1, 1, 1]], 1/8, 1/8];

~print.value(a, b);
```

```
-> a Routine
```

0.2.3 Dynamic

Tipically we can define amplitude with three different measurament units:

- 1.0 → linear amplitude (floating point form 0.0 to 1.0).
- 127 → midi velocity (integer from 0 to 127).
- -6 → decibels (from -inf to 0.0).

Conversions:

[ ]:
```
~lin  = 64  / 127;  // velocity to linear
~vel  = 0.5 * 127;  // linear to velocity
```

```
~lin1 = -6.dbamp;   // dB to linear
~db   = 0.5.ampdb;  // linear to dB
~cub  = 0.5**4;     // linear to quartic (best for human perception)

~db;                // change
```

0.2.4 Collections

As we saw we can describe sets of values as collections.

In SuperCollider arrays and lists.

Collections are a musically neutral data type as they can contain any data type (string, int, float, etc.).

In algorithmic composition we can think it musically in two ways:

1. Out of time (pre compositional materials)
    - rules paradigm.
    - item positions on x axis are indexes without time references (scale degree or other).

[115]:
```
//   - Scale
//          0   1   2   3   4   5   6   7        // Scale degrees
~dorico = [60, 62, 64, 65, 67, 69, 71, 72];      // Absolute pitches
~chrom  = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]; // Intervals

//   - Series

~serie  = [1, 10, 6, 4, 8, 2, 9, 7, 0, 5, 11, 3]; // Dodecaphony
~reich  = [64, 66, 71, 72, 73, 66];               // Minimalism

//   - Harmonic fields (Array or Array 2D)

~mayor  = [0, 4, 7, 12];
~harm   = [[60, 64, 67, 72], [62, 65, 67, 71], [60, 65, 69,72], [60, 64,␣
 ↪69,72]];
~harmd  = [[0, 4, 7, 12],    [2, 5, 7, 11],    [0, 5, 9,12],    [0, 4, 9,12]];

~durs  = [1/8];

~print.value(~dorico, ~durs);
```

```
-> a Routine
```

2. Timeline
    - sequence paradigm.
    - items are placed on a timeline (x axis).

[117]:
```
//   - Sequence of index from defined scale

```

```
//            0   1   2   3   4   5   6   7
~scale  = [60, 62, 64, 65, 67, 69, 71, 72];
~melody = 10.collect({rrand(0,7)});
~seq    = ~melody.collect({arg i; ~scale[i]});

~durs   = ([1/16,1/8,1/16]!3++[1/4]).flat;

~print.value(~seq,~durs);
```

```
-> a Routine
```

0.2.5 Live

We can also use live algorithmic composition techniques by sending the numerical data via MIDI to external sound generators (Reaktor, GarageBand, Finale, Logic) without print it in a musical score.

[ ]:
```
"open -a /Applications/GarageBand.app/".unixCmd;
```

[ ]:
```
// 1. Checks which available MIDI sources and destinations there are, and
//    opens as many connections as desired

MIDIClient.init;

// 2. Choice the destination where you want send your MIDI data (id from 0)

c = MIDIOut.new(0); // 0 = "Driver IAC"
c.latency_(0);      // Set latency to 0 sec

// 3. Open an external MIDI sound bank (Reaktor, GarageBand, Finale, Logic)
// 4. Send some MIDI data..

(
~bpm   = 60;
~note  = 200;
~tsig  = [4,4];
~pitch = ~note.collect({rrand(60,75)});
~durs  = ~note.collect({[1/32, 1/32,1/32, 1/32,1/32,1/8, 1/16].choose});
~vels  = ~note.collect({[100, 40, 60].choose});

p = Pbind(
        \type, \midi,
        \midicmd, \noteOn,
        \midiout, c,
        \chan, 0,
        \midinote, Pseq(~pitch.replace([nil],\rest)),
        \sustain,  Pseq(~durs * ~tsig[1],inf),    // Duration
        \dur,  Pseq(~durs * ~tsig[1],inf),        // Delta time
```

```
        \amp, Pseq(~vels/127,inf),                // 0.0 to 1.0
        ).play;
)
```

0.3 Generative techniques and processes

In Western musical tradition, we have only two ways to conceive of the superposition of two or more voices: * counterpoint (horizontal dimension - static voice allocation)

- harmony (vertical dimension - dynamic voice allocation).

They are two musical approaches to which different musical languages refer.

We are going to explore compositional practices about them through three case studies:

- Modes and scales (counterpoint).
- Harmonic fields (harmony).
- XIX sec. music (both).

In general if we want formalize our musical ideas we should:

1. define a set of parametric structures that we want to use as basic material according to musical rules (scale, modes, serie, chords, rhythmic or dynamic patterns, etc.).
2. define a set of possible modifications always starting from musical needs and rules (variations, filters, interpolations, augmentations, diminutions, etc.).
3. organize it over time by defining a formal structure choosing from two strategies:
    - top to bottom (from a fixed general structure to particular).
    - bottom to top (from particular to a general formal structure).

As we know, in carrying out these three steps, we have only two possible approaches:

- nondeterministic

- deterministic

They are often mixed toghether.

0.3.1 Modes and scales

Both are based on the concept of 'mode' (gregorian and nedieval music) or 'scale' (baroque and classical music).

It define the interval patterns through which composer organize the pitches of a piece.

They are usually some kind of subdivision of the octave interval.

We can define it in interval or grades.

[ ]:
```
~major = [0,2,4,5,7,9,11,12];  // Try to change intervals
```

Then we have to define a root note in midi values which will assume the function of the first degree (0)

- in modal rules is called 'finalis'.
- in tonal (harmonic) rules is the 'tonality' (D major, E-flat minor, etc.).

[118]:
```
~major = [0,2,4,5,7,9,11,12];  // Try to change intervals
~root  = 60;
~pitch = ~root + ~major;
~dur   = [1/8];

~print.value(~pitch, ~dur, filepath:~filepath);
```

```
-> a Routine
```

We can use historical models or invent our own.

In SuperCollider there is a Class that represent a repository of different scales from different coltures.

[ ]:
```
Scale.directory;
```

We can choose a scale from the available scales and then recall the interval model.

[121]:
```
~model = Scale.lydian;    // Try to change scale

~degree = ~model.degrees;
~root   = 60;
~pitch  = ~root + ~degree ++[72];
~dur = [1/8];

~degree.postln;
~pitch.postln;

~print.value(~pitch, ~dur, filepath:~filepath);
```

```
[ 0, 2, 4, 6, 7, 9, 11 ]
[ 60, 62, 64, 66, 67, 69, 71, 72 ]
-> a Routine
```

Once the interval model has been established, we can create a melody by recalling the different degrees of the model according to deterministic or indeterministic principles.

Let's start from define a function that generate a melodic cell (inciso) according to few simply rules:

1. define the number of notes in the inciso (random from 5 to 7).
2. define a percentage of random rest in the inciso (from 0.0 to 1.0).

[24]:
```
~model  = Scale.lydian.degrees;

~inciso  = {arg degrees=a, root=60, rrest=0.0;
            var notes, rests, pitch;
                notes = rrand(3,6);
                rests = [degrees,[nil]];      // [[degrees],[nil]]
                pitch = notes.collect{var i;
                                      i = rests.wchoose([1-rrest,rrest]).choose;
                                      if(i.notNil){i = i + root}
                                     };
```

```
                pitch
            };

a = ~inciso.value(~model,rrest:0.2);
a.postln;
~print.value(a,[1/8])
```

```
[ 69, nil, 60, nil, 64 ]
-> a Routine
```

3. define a percentage of random velocities to apply only on notes.
   - veld → array from wich velocities are chosen.
   - rvels → percentage of vels (0.0-1.0). N.B. The dynamic change only the note, if not specified is 'mezzoforte' di default.

[33]:
```
~model  = Scale.lydian.degrees;

~inciso  = {arg degrees=a, root=60, rrest=0.0, vels=[nil], rvels=0.0;
            var notes, rests, pitch, vel;
                notes = rrand(3,6);
                rests = [degrees,[nil]];      // [[degrees],[nil]]
                pitch = notes.collect{var i;
                                      i = rests.wchoose([1-rrest,rrest]).choose;
                                      if(i.notNil){i = i + root}
                                     };
                vel = pitch.collect{arg i;
                                    if(i.notNil){[vels,[nil]].
 ↪wchoose([rvels,1-rvels]).choose}
                                    }.flat;
                [pitch, vel]
            };

a = ~inciso.value(~model,rrest:0.2,vels:[30,100],rvels:0.7 );
a[0].postln;
a[1].postln;
~print.value(a[0],vel:a[1])
```

```
[ 60, 69, 66, 71, 62, nil ]
[ 100, nil, nil, 30, 30, nil ]
-> a Routine
```

4. define a database of rhythmic patterns from which to extract one according to the number of pitches. Each pattern should be of the same lenght.

[14]:
```
~model  = Scale.lydian.degrees;

~inciso =  {arg degrees=a, root=60, rrest=0.0, vels=[nil], rvels=0.0;
            var notes, rests, pitch, vel, rhytm;
                notes = rrand(3,6);
```

```
                rests = [degrees,[nil]];      // [[degrees],[nil]]
                pitch = notes.collect{var i;
                                      i = rests.wchoose([1-rrest,rrest]).choose;
                                      if(i.notNil){i = i + root}
                                     };
                vel = pitch.collect{arg i;
                                     if(i.notNil){[vels,[nil]].
↪wchoose([rvels,1-rvels]).choose}
                                     }.flat;
                switch(notes,
                    3, {rhytm = [
                        [1/8,1/16,1/16],   // Rhythmic patterns all in 1/4 or␣
↪multiply
                        [1/16,1/8,1/16],       // 1 beats
                        [1/16,1/16,1/8],       // 1 beats
                        [1/4,[1/4,[3,1]]],     // 2 beats
                        [1/8, [1/8,[1,2]]]     // 1 beats
                                ].choose},
                    4, {rhytm = [
                        [1/8,1/16,1/16,1/4],   // 2 beats
                        [1/4,1/16,1/8,1/16],   // 2 beats
                        [1/16, 1/4, 1/16,1/8], // 2 beats
                        [1/4,[1/4,[3, 1, 2]]], // 2 beats
                        [1/8,[1/8,[1, 1, 1]]]  // 1 beats
                                ].choose},
                    5, {rhytm = [
                        [1/8,1/16,1/16,1/8,1/8],    // 2 beats
                        [1/8,1/16,1/8,1/16,1/8],    // 2 beats
                        [1/32, 1/4, 1/32,1/8,1/16], // 2 beats
                        [1/4,[1/4,[1, 2, 1, 1]]],   // 2 beats
                        [1/8,1/4, [1/8,[1, 1, 1]]]  // 2 beats
                                ].choose},
                    6, {rhytm = [
                        [1/8,1/16,1/16,1/16,1/8,1/16],    // 2 beats
                        [1/8,1/32,1/8,1/32,1/8,1/16],     // 2 beats
                        [1/32, 1/8, 1/32, 1/8, 1/16,1/8], // 2 beats
                        [1/8,[1/4,[1, 2, 1, 1]], 1/8],    // 2 beats
                        [1/8,1/4, [1/8,[1, 1, 1]],1/8]    // 2 beats
                                ].choose};
                        );
            [pitch, rhytm, vel]
            };

a = ~inciso.value(~model, rrest:0.2, vels:[20,100], rvels:0.2);
a[0].postln;
a[1].postln;
a[2].postln;
```

```
~print.value(a[0], a[1],a[2])
```

```
[ 66, 67, nil, 66, 71 ]
[ 0.125, 0.0625, 0.125, 0.0625, 0.125 ]
[ nil, nil, nil, 20, nil ]
-> a Routine
```

We can now construct several incises and concatenate them into a sequence:

[38]:
```
a = ~inciso.value(~model,rrest:0.2,vels:[20,100],rvels:0.3);
b = ~inciso.value(~model,rrest:0.3,vels:[20,100],rvels:0.3);

~pseq = (a[0] ++ b[0] ++ a[0]).flat;
~rseq = (a[1] ++ b[1] ++ a[1]).flatten(-1);
~vseq = (a[2] ++ b[2] ++ a[2]).flat;

~pseq.postln;
~vseq.postln;
~rseq.postln;

~print.value(~pseq, ~rseq, vel:~vseq);
```

```
-> a Routine
```

[ ]:
```
a = ~pseq ++ ~pseq ++ ~pseq;
b = ~rseq ++ ~rseq ++ ~rseq;
c = ~vseq ++ ~vseq ++ ~vseq;

~print.value(a, b, vel:c);
```

```
[ 67, 67, 69, 67, nil, 66, 67, 67, 69 ]
[ nil, nil, nil, 100, nil, nil, nil, nil, nil ]
[ 0.125, 0.0625, 0.0625, 0.125, 0.0625, 0.0625, 0.125, 0.0625, 0.0625 ]
-> a Routine
```

0.4 Composition sketches proposal (SC and Lilypond)