OPTION PF2 - Automne 2016
Jean-Paul Roy
L2-Info&Maths, Faculté des Sciences

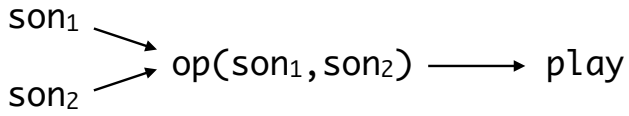
http://deptinfo.unice.fr/~roy

Cours 4

Les signaux audio et la fabrication du son

(avec RSound)

- Dans le cours précédent nous avons vu comment **combiner des sons déjà existants** (par ex. issus de fichiers .wav), et comment les écouter.

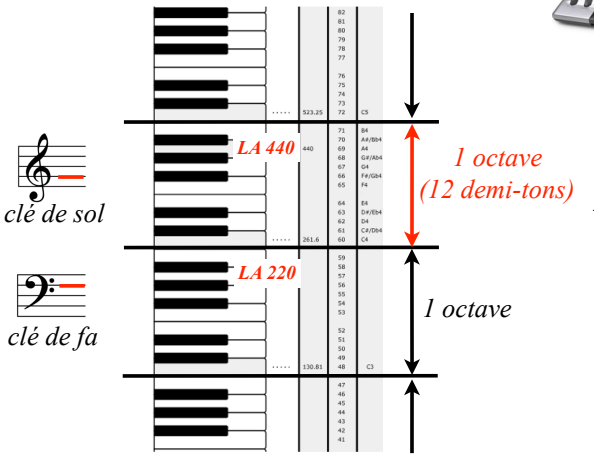


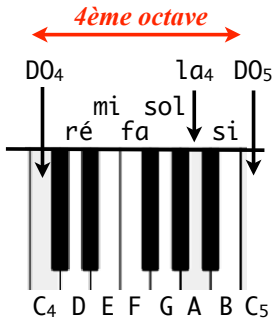
- Dans ce cours, l'accent sera mis sur la **génération du son à partir d'ondes sinusoïdales**. Nous serons amenés à introduire des **signaux** (flux audio) se propageant via les noeuds d'un réseau logiciel : générateurs d'ondes, filtres, etc. *Thématique : **signaux & systèmes**.*

- Le style de programmation reste fonctionnel mais se double d'un style dit **dataflow** pour exprimer la construction d'un signal audio qui coule à l'intérieur d'un réseau . Ce signal pourra être écouté ou converti en son.

- Nous débuterons par quelques compléments sur les notes de musique et la notation MIDI.

L'échelle musicale occidentale





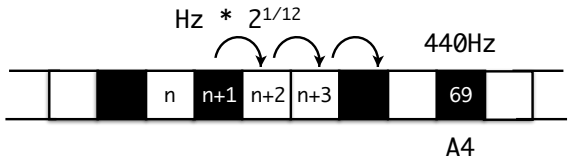
- L'**échelle musicale occidentale** divise un octave en **12 intervalles** égaux correspondant à 12 notes au clavier . Le rapport entre deux intervalles consécutifs est donc $\sqrt[12]{2} = 2^{1/12}$

Le système de notation musicale MIDI (notes de 0 à 127)

FR	DO3	DO#	RE	MI ♭	MI	FA	FA#	SOL	SOL#	LA3	SI ♭	SI
USA	C4	C#	D	D#	E	F	F#	G	G#	A4	A#	B
Hz	261.6	277.2	293.7	311.1	329.6	349.2	370	392	412.5	440	466.2	493.9
MIDI	60	61	62	63	64	65	66	67	68	69	70	71

- Un clavier MIDI envoie à l'ordinateur une suite d'informations MIDI (le numéro de la note, la pression sur la touche, etc). Pour transiter entre numéros de note MIDI et fréquences :

```
(midi-note-num->pitch 69) -> 440    (Hz)
(pitch->midi-note-num 440) -> 69    (MIDI)
```



Comment jouer des notes de piano ?

```
> (require rsound/piano-tones)

> (play (piano-tone 60))

> (play (assemble                          ; Beethoven 5th
          `((,(piano-tone 67) 0)           ; G   ==   sol
            (,(piano-tone 67) 10000)       ; G   ==   sol
            (,(piano-tone 67) 20000)       ; G   ==   sol
            (,(piano-tone 63) 30000))))    ; Eb  ==   mi-bémol

> (play (assemble                          ; accord DO majeur
          `((,(piano-tone 60) 0)           ; C == DO
            (,(piano-tone 64) 0)           ; E == MI
            (,(piano-tone 67) 0))))        ; G == SOL
```

• Trouver des algorithmes qui génèrent des suites de notes agréables à l'oreille n'a rien d'évident. Il s'agit de **composition algorithmique** (ex: utilisation de *chaînes de Markov*)... L'aléatoire est-il beau ?

L'exemple de Clements à RacketCon 2014

original 18 sec

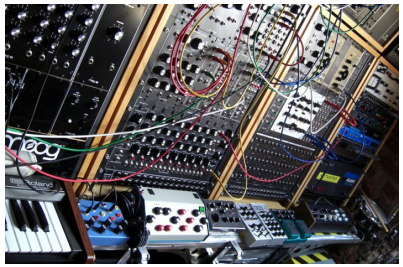
clipped 12 sec

```
#lang racket
;;; Notes de piano avec chevauchement, tous les 1/10 sec.
;;; Echantillonné à 44.1kz, 1 sec. de son = 44100 frames
(require rsound rsound/piano-tones rsound/envelope)
(define s1 (assemble (for/list ([i 100])
                       (list (piano-tone (+ 25 (random 30)))
                             (*  i 4410)))))    ; 1/10 sec
(define s2 (clip s1 0 540000))
(define s3 (rs-mult s2 ((adsr 1 1 1 1 44100) 540000)))
(rs-write s3 "racketcon-clipped.wav")
```


L'organisation d'un Réseau Musical (network)

• Les joueurs de musique électronique utilisent des appareils branchés en réseau : générateurs de sons, filtres, synthétiseurs, clavier, guitare, ADC, DAC, HP. Ils parlent souvent d'un ***patch***.

• Nous allons assembler des composants musicaux logiciels sous la forme d'un graphe formé de **noeuds**. Certains noeuds seront des générateurs, d'autres des transformateurs.



• Dans un tel réseau (**network**) , un noeud (**node**) peut comporter des entrées et une sortie. S'il n'a pas d'entrée, c'est un **signal**. S'il n'a qu'une seule entrée, c'est un **filtre**. Le dernier noeud est la sortie **out**.


```
(network (in ...)                                                    syntax
         network-clause
         ...)

            in = identifier

 network-clause = [node-label = expression]
                | [node-label <= network expression ...]
                | [(node-label ...) = expression]
                | [(node-label ...) <= network expression ...]

     node-label = identifier
```

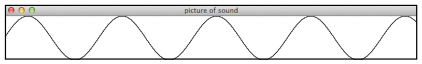
*Produces a **network**. The **in** names specify input arguments to the network. Each network clause describes a **node**. Each node must have a label, which may be used later to refer to the value that is the result of that node. Multiple labels are used for clauses that produce multiple values.*
There are two kinds of clause. A clause that uses = simply gives the name to the result of evaluating the right-hand-side expression. A clause that uses <= evaluates the input expressions, and uses them as inputs to the given network.

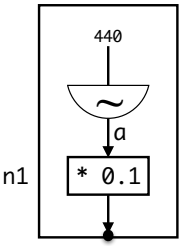
Les signaux élémentaires : sine, sawtooth, square, pulse

• L'**onde sinusoïdale** **de fréquence f** (en Hz) est pure et réduite à un son fondamental. Elle est décrite par (sine-wave f) qui est un signal. Attention, sine-wave n'est pas une fonction Scheme mais un réseau. Elle n'est utilisable que pour décrire un noeud à l'intérieur d'un réseau.

```
(define n1 (network ()                ; un signal
             (a <= sine-wave 440)     ; LA440 (A4)
             (out = (* 0.1 a))))      ; attention aux oreilles !
```

```
> (list (signal? n1) (signal? sine-wave) (network/s? sine-wave))
(#t #f #t)
> (define s1 (signal->rsound 44100 n1)) ; 1 sec
> (play s1)
> (rs-draw (signal->rsound 441 n1))     ; 0.01 sec
```

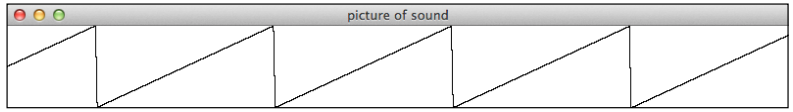




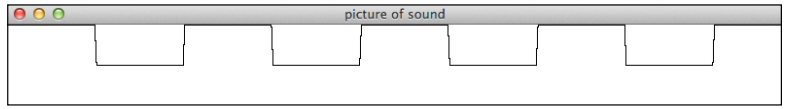
• La **scie** **de fréquence f** (en Hz) produit un son plus riche. La **fréquence fondamentale** est ici f_0 = 440 Hz, mais on y trouve aussi à degrés moindres toutes les fréquences multiples de 440 : 880, 1320…

```
(define n2
  (network ()
    (a <= sawtooth-wave 440)
    (out = (* 0.1 a))))
```

sine sawtooth

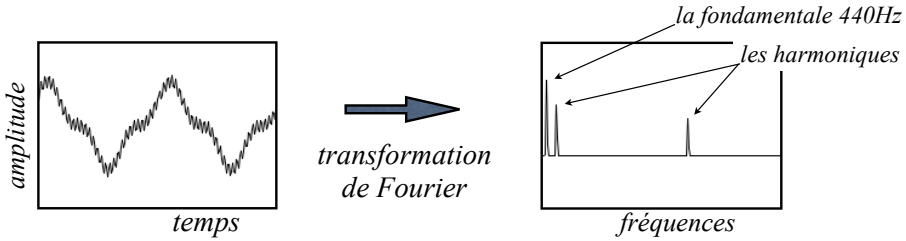


• Le **signal carré** (square-wave f) **de fréquence f** (en Hz) alterne entre 0 et 1. Si la **fréquence fondamentale** est f_0, on y trouve aussi à degrés moindres toutes les fréquences multiples impaires $3f_0$, $5f_0$, etc.



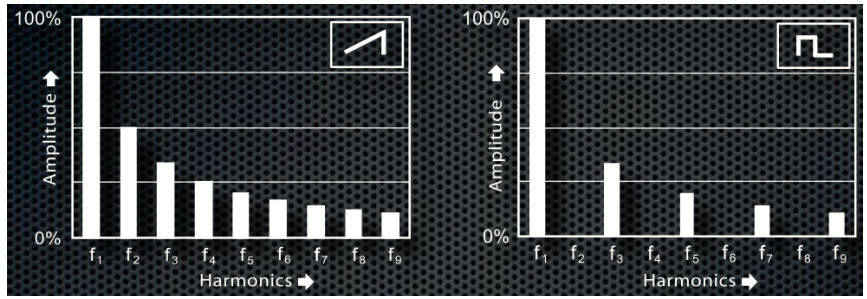
• Le **signal rectangulaire** ou **impulsionnel** (pulse-wave dc f) **de fréquence f** (en Hz) généralise le signal carré. L'argument dc ∈ [0,1] est la fraction du temps pendant lequel le signal vaut 1 (*duty cycle*).

• **Le son fondamental est le son sinusoïdal**. A partir de lui, le mathématicien Joseph **Fourier** a pu décomposer (1807) tout signal périodique en une somme (infinie ?) de signaux sinusoïdaux dont les fréquences se trouvent parmi les multiples d'une fréquence fondamentale f_0.



Le domaine temporel Le domaine fréquentiel

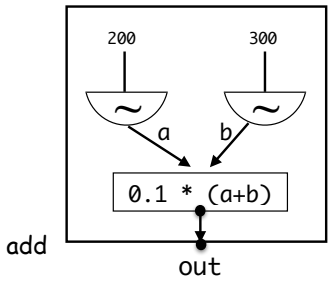
Les harmoniques des signaux sawtooth et square



http://pages.uoregon.edu/emi/11.php

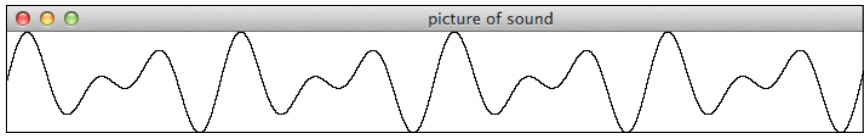
La synthèse additive

• La somme de deux signaux sonores périodiques est un signal périodique *(si le rapport des périodes est rationnel)*, mais pas un signal sinusoïdal pur en général.




```
(define add
  (network ()
    (a <= sine-wave 200)
    (b <= sine-wave 300)
    (out = (* 0.2 (+ a b)))))
```

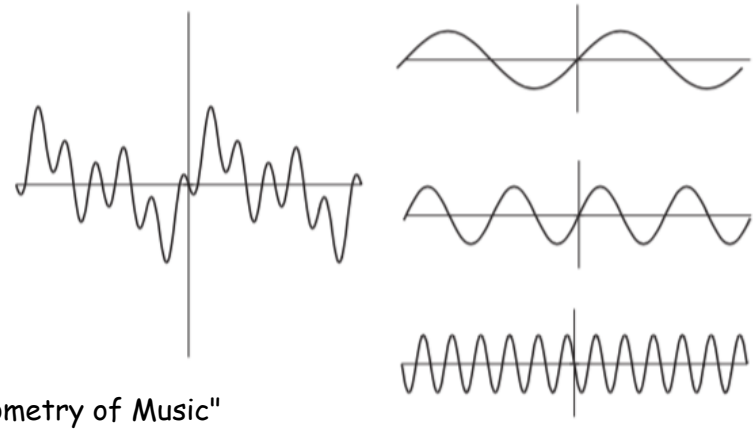
```
(rs-draw (signal->rsound 2000 add))
```



• La période résultat est ici 1/100 s. Vous voyez pourquoi ?...

Vers Fourier : la synthèse additive inverse

Figure 2.10.1 The periodic sound on the left has frequency *f*. (Two repetitions of the waveform are shown.) This sound can be analyzed as the sum of the sine waves on the right, with periods *f*, 2*f*, and 6*f*.



"A Geometry of Music"
D. Tymoczko
Oxford Univ. Press, 2011

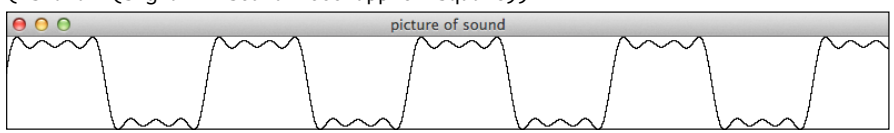
Vers Fourier : approximation du signal carré

• Le **signal carré** *square(t)* qui est périodique, doit suivant la théorie de Fourier, être décomposable en somme d'oscillateurs. On peut montrer en cours de math que :

$$\text{square}(t) = \frac{4}{\pi} \sum_{n=1,3,5,\ldots} \frac{1}{n} \sin(n f_0 t)$$

```
(define approx-square
  (network ()
    (a <= sine-wave 440)          ; la fondamentale f0
    (b <= sine-wave (* 3 440))    ; et seulement
    (c <= sine-wave (* 5 440))    ; trois
    (d <= sine-wave (* 7 440))    ; harmoniques.
    (out = (* 0.1 (+ a (* 1/3 b) (* 1/5 c) (* 1/7 d))))))
```

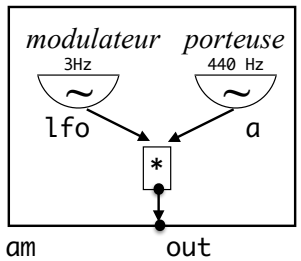
```
(rs-draw (signal->rsound 2000 approx-square))
```



La modulation d'un signal (AM : modulation d'amplitude)

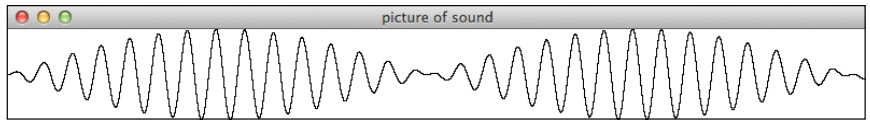
• Une **modulation** consiste à modifier un signal (la *porteuse* ou *source*) à l'aide d'un autre signal (le *modulateur*). Ici on s'intéresse à l'**amplitude** (**AM**), mais ce pourrait être la fréquence (FM).

• Prenons comme porteuse un sinus à 440 Hz et comme modulateur un autre sinus **à basse fréquence** (LFO à 3 Hz), ou tout traitement mathématique variant dans [-1,+1].



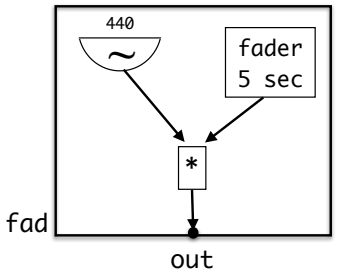

```
(define am (network ()
             (a <= sine-wave 440)
             (lfo <= sine-wave 3)
             (out = (* a lfo))))
```

am



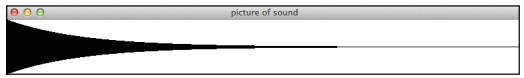
L'affaiblissement d'un signal (fading)

• La fonction (fader n) retournant un signal qui décroît exponentiellement vers 0.001 après n échantillons.



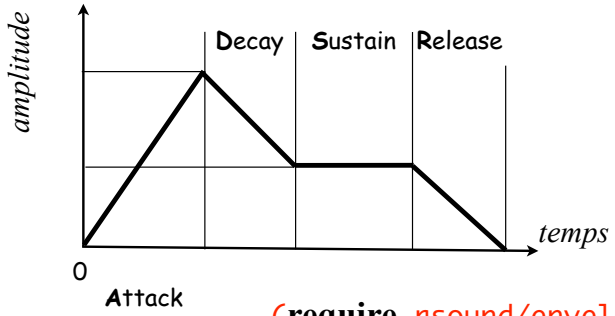

```
> (signal? fad)
#t
```

```
(define (sec n)
  (inexact->exact (round (* n 44100))))    ; seconds → frames
(define sin440 (network () (out <= sine-wave 440)))
(define fad (signal-* sin440 (fader (sec 5))))
(rs-draw (signal->rsound (sec 5) fad))
(signal-play fad)
```



Enveloppe d'une note : Attack/Decay/Sustain/Release

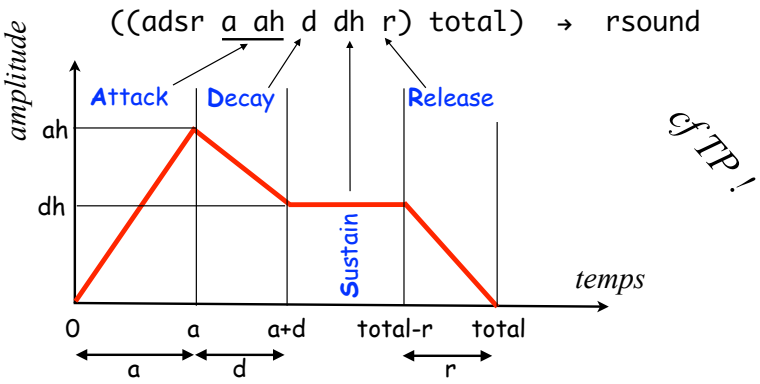
• Le LA-440Hz pur est… trop pur ! On va essayer de monter vivement vers une note (**attaque**/*attack*), la faire chuter un peu (**déclin**/*decay*), la maintenir un certain temps (**soutien**/*sustain*), pour la faire mourir en douceur (**relâchement**/*release*). C'est un cas particulier d'**enveloppe sonore**. En jouant sur ces 4 paramètres A/D/S/R, on obtient des effets sonores différents.



(require rsound/envelope)

• On peut en effet affiner un son avec une enveloppe adsr. Soit s un son déjà fabriqué. On peut l'envelopper en le *multipliant* avec une fonction ADSR variant entre 0 et 1 pour forcer une montée initiale depuis 0 et une descente finale vers 0.

• On peut **multiplier deux sons** avec rs-mult, et deux signaux avec signal-*, convertir entre sons et signaux, et construire une enveloppe sonore avec la fonction adsr :

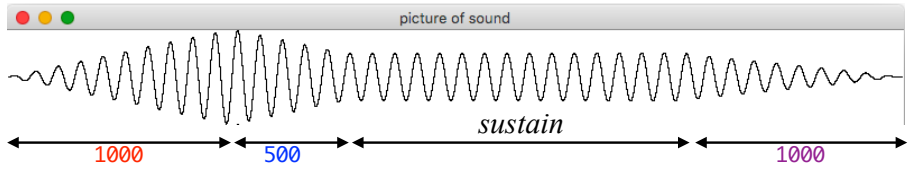


cf TP !

• Exemple : je pars d'un signal SIG d'où je déduis un son SND, je lui construis une enveloppe sous la forme d'un son ENV (inaudible), je multiplie les deux pour appliquer l'enveloppe, et j'obtiens le son enveloppé SND-ENV. *J'ai* ***sculpté*** *le son !*

```
#lang racket
(require rsound rsound/draw rsound/envelope)

(define SIG (network () (out <= sine-wave 440)))
(define SND (signal->rsound 4000 SIG))
(define ENV ((adsr 1000 1 500 0.5 1000) (rs-frames SND)))
(define SND-ENV (rs-mult SND ENV))
(play SND-ENV)
(rs-draw SND-ENV)
```



Synthèse soustractive - les Filtres

• Au lieu de *réunir* plusieurs sons, on choisit d'**élaguer un son**. en atténuant des fréquences :

- basses (filtre HP : ***high-pass***)
- hautes (filtre LP : ***low-pass***)
- en-dehors d'une bande (filtre BP : ***band-pass***)

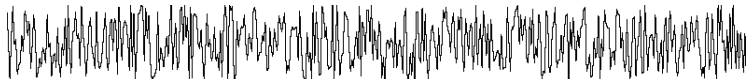
• Le filtre passe-bande BP est utilisé en téléphonie : pour le son de la conversation humaine, on peut filtrer entre 300 et 3000Hz.

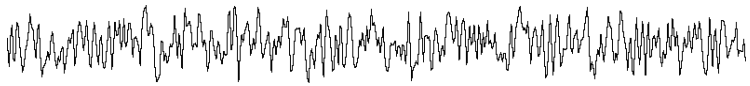
• Les techniques de constructions de filtres sont difficiles et utilisent la notion de ligne retardée (**delay**). Disons qu'un signal est une suite infinie d'échantillons x[n], transformée en y[n] par un filtre.

• La manière la plus naïve d'élaguer les hautes fréquences (filtre ***low-pass***) consiste à prendre la moyenne de deux échantillons successifs :

y[n] = 0.5 * (x[n] + x[n-1])

```
(define (low-pass-filter snd)
  (define n (rs-frames snd))
  (build-sound n
    (lambda (i)
      (* 0.5 (+ (rs-ith/left snd (modulo (- i 1) n))
                (rs-ith/left snd i))))))
```

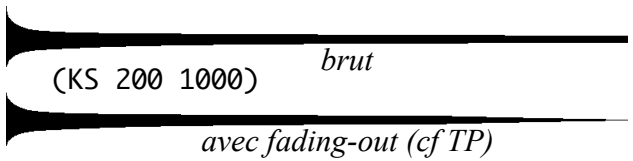
avant : 

après : 

L'algorithme de Karplus-Strong

• L'algorithme de **Karplus-Strong** (1983) en itérant ce filtre LP sur un **bruit blanc** court permet de synthétiser le pincement d'une corde de guitare, ou d'autres percussions (tambour, etc). Implémentation *naïve* :

```
(define (KS n k)        ; n = longueur de la ligne de délai
  (define d (build-sound n (lambda (i) (- (random) 0.5))))
  (define (iter i d L)    ; L est une liste de sons
    (if (= i k)
        (rs-append* (reverse L))
        (let ((new-d (low-pass-filter d)))
          (iter (+ i 1) new-d (cons new-d L)))))
  (iter 0 d '()))
```



Lecture en boucle d'un son

• Soit un son à lire en boucle :

```
(define SONG (rs-read "5th.wav"))
(define SONGLEN (rs-frames SONG))
```

• La première stratégie consistait à utiliser une *pstream* avec un `pstream-queue-callback` qui relance la lecture à la fin.

• La seconde s'inscrit plus dans une optique dataflow (*networks*). On commence par construire un noeud `i` débitant un signal contenant les entiers naturels (à partir de 0 par incrément de 1) avec `simple-ctr`.

```
(define looper
  (network ()
           [i <= (simple-ctr 0 1)]
           ...))
```

• On imagine les entiers `i` coulant le long d'un fil…

• On réduit ensuite i modulo SONGLEN pour revenir automatiquement à 0 au dernier échantillon :

```
(define looper
  (network ()
           [i <= (simple-ctr 0 1)]
           [j = (modulo i SONGLEN)]
           ...))
```

• Il suffit alors d'envoyer dans la sortie audio l'échantillon numéro j du morceau SONG :

```
(define looper
  (network ()
           [i <= (simple-ctr 0 1)]
           [j = (modulo i SONGLEN)]
           [out = (rs-ith/left SONG j)]))

(signal-play looper)
```


Jouer la gamme chromatique

• Il s'agit de faire jouer les 12 notes de la **gamme chromatique**, soit **toutes** les notes (blanches et noires) de numéros MIDI 60 à 71. Mais avec des oscillateurs sinusoïdaux, pas des notes de piano…

FR	DO3	DO#	RE	MI ♭	MI	FA	FA#	SOL	SOL#	LA3	SI ♭	SI
USA	C4	C#	D	D#	E	F	F#	G	G#	A4	A#	B
Hz	261.6	277.2	293.7	311.1	329.6	349.2	370	392	412.5	440	466.2	493.9
MIDI	60	61	62	63	64	65	66	67	68	69	70	71

```
(define (chromatic n1 n2)     ; gamme chromatique 1 seconde/note
  (network ()
           [frame <= (simple-ctr 0 1)]              ; frames écoulées
           [sec = (floor (/ frame #i44100))]        ; secondes écoulées
           [note = (+ n1 sec)]                      ; note MIDI à jouer
           [freq = (if (> note n2) 0 (midi-note-num->pitch note))]
           [signal <= sine-wave freq]               ; signal en sortie
           [out = (* signal 0.1)]))

(signal-play (chromatic 60 71))
```


Construire ses propres percussions

• Rsound propose des percussions toutes prêtes : ding, crash-cymbal, etc. Mais on peut en télécharger d'autres voire en construire soi-même à partir des ondes élémentaires (sine, square, sawtooth, etc) et une ***enveloppe percussive*** bien choisies. Exemple :

```
; je définis un son pur F6# (MIDI 90)
(define SND
  (signal->rsound 11025        ; 0.25 sec à 1480 Hz
    (network () (out <= sine-wave 1480))))
; je lui construis une enveloppe percussive
(define ENV ((adsr 100 1 100 0 200) (rs-frames SND)))
(define PERC (rs-mult SND ENV)) ; et j'obtiens un son enveloppé
(define PERC-LEN (rs-frames PERC))
(define ps (make-pstream))
(define CYMBAL (rs-scale 0.2 crash-cymbal))
(for ([i (in-range 100)])
  (pstream-queue ps
    (if (> (modulo i 8) 5) CYMBAL PERC)
    (+ (pstream-current-frame ps) (* i PERC-LEN))))
```

Pour approfondir (l'été prochain ?)

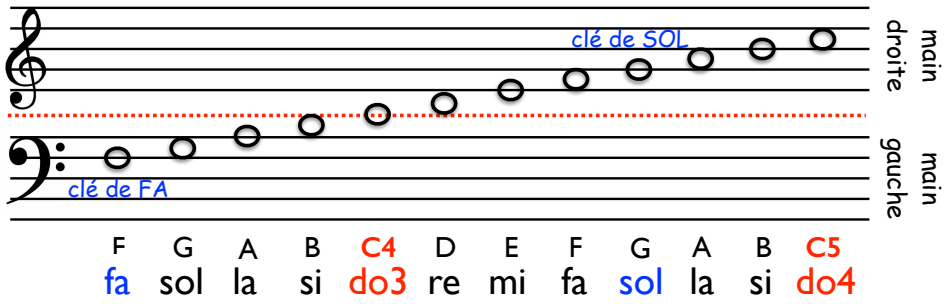
http://music.columbia.edu/cmc/MusicAndComputers/

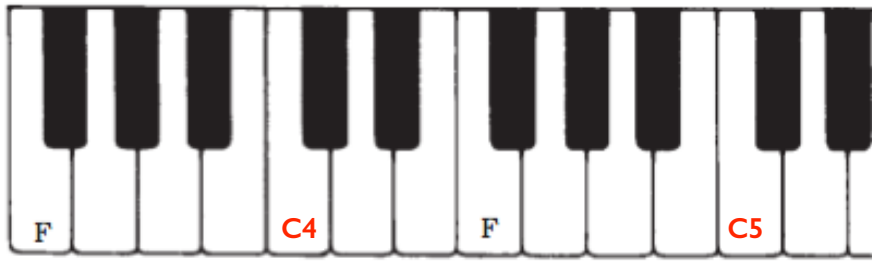
Bon panorama

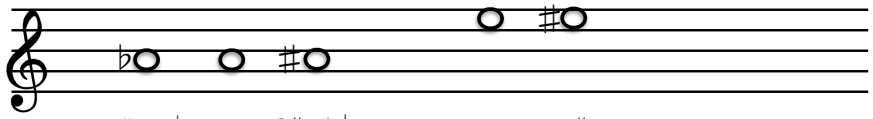
MOOC

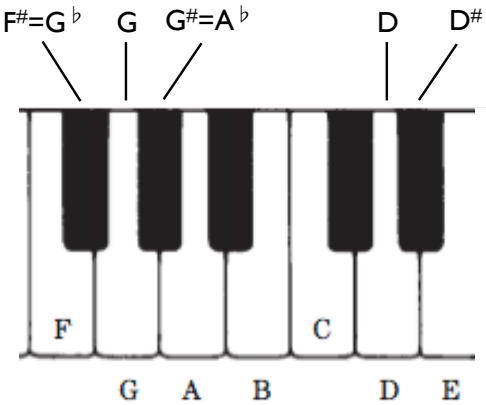
https://www.coursera.org/course/digitalsounddesign

Optionnel : mini-mémento sur les partitions (clavier)



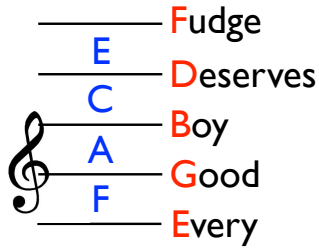


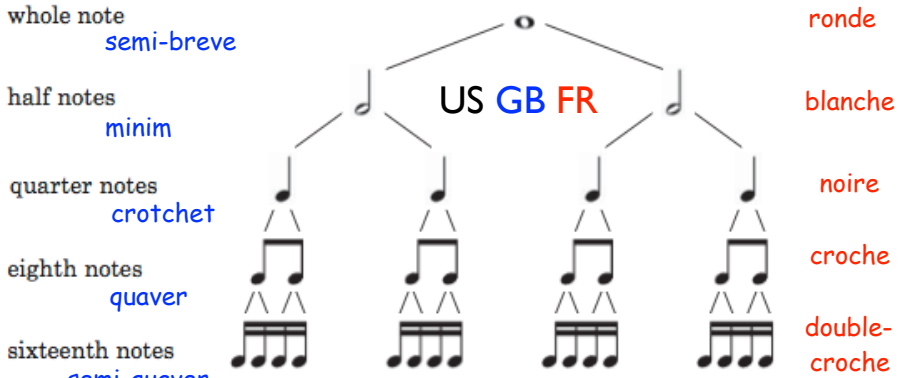


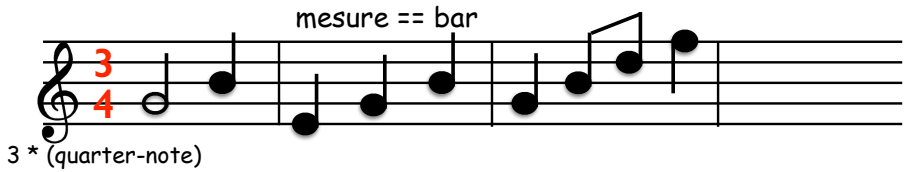


F# : F sharp, FA dièse

G♭ : G flat, SOL bémol

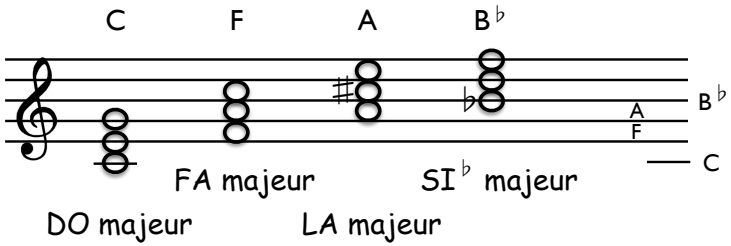






• Le **tempo** (nombre **bpm** de battements par minute) permet d'avoir la durée d'une croche (quarter note). Avec 60 battements/minute : ♩ = 60

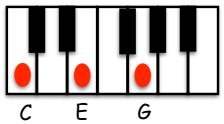
- Quelques accords majeurs…



p. 174
(cliquez)

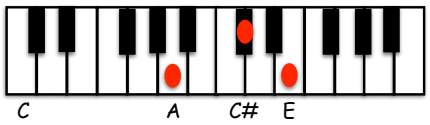
accord de DO majeur :

DO (C) —4 demi-tons→ MI (E) —3 demi-tons→ SOL (G)



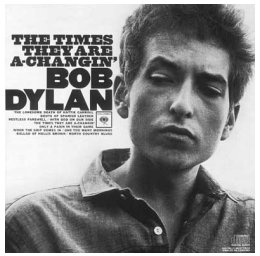
accord de LA majeur :

LA (A) —4 demi-tons→ DO# (C#) —3 demi-tons→ MI (E)



Voir p. 6 avec assemble…

The Times They Are A-Changin' (1964)



Come writers and critics
Who prophesize with your pen
And keep your eyes wide
The chance won't come again
And don't speak too soon
For the wheel's still in spin
And there's no tellin' who
That it's namin'.
For the loser now
Will be later to win
For the times they are a-changin'.