

Guide d'utilisation de SuperCollider

Avant-propos

Avant toute chose, SuperCollider (SC) est un outil puissant et dont les possibilités créatrices chatouillent l'infini. **Il nécessite cependant des bases solides en programmation et en acoustique** pour être utilisé à son plein potentiel. Même si ce guide s'adresse au débutants, il ne reprend pas ou peu les bases de ces deux sujets.

Il vous est donc conseillé en parallèle de vous documenter sur ces deux sujets si besoin.

Ce document présente SuperCollider de manière graduelle, en présentant peu à peu le cadre de travail que celui-ci propose. Si une notion mentionnée lors d'un exemple vous échappe, il peut donc être nécessaire de se renseigner sur cette notion avant de poursuivre la lecture.

Pour les lecteurs totalement débutants en programmation, il peut être utile de commencer par découvrir le langage [Sonic-Pi](#), qui est une version simplifiée de SuperCollider, destinée à l'apprentissage de la programmation, et donc une excellente porte d'entrée à SuperCollider.

Installation

Si vous êtes d'attaque, il faut d'abord commencer par l'installer : vous pourrez trouver la version adaptée à votre système d'exploitation sur [le site officiel](#), à la section 'Downloads'.

Alternativement, pour les utilisateurs de Linux, vous pouvez le trouver depuis le gestionnaire de paquets :

```
sudo apt-get install scide
```

Introduction au logiciel

Avant de détailler les différents espaces qui s'affichent lorsque l'on lance le logiciel, un point important. **SuperCollider n'est pas un logiciel construit d'un seul bloc.** Il contient en fait trois composants :

- **scsynth** : le **serveur sonore**, c'est lui qui est responsable de créer le son qui sort des enceintes.
- **sclang** : un **langage de programmation** spécialement conçu pour manipuler facilement le serveur.
- **scide** : un **éditeur de code** développé spécialement pour manipuler le langage *sclang*. Dans ce guide, il sera dénommé **IDE** (Integrated Development Editor).

En tant qu'artistes, c'est d'abord le serveur *scsynth* qui nous intéresse, car il nous permet de produire des sons complexes 'assez facilement'. Le serveur est conçu pour fonctionner de manière autonome, et peut être piloté par n'importe quel langage de programmation. Cependant, il faut de très bonnes connaissances pour le faire.

De fait, les développeurs ont créé le langage *sclang* pour piloter le serveur bien plus facilement. Ce n'est pas obligatoire, mais fortement conseillé.

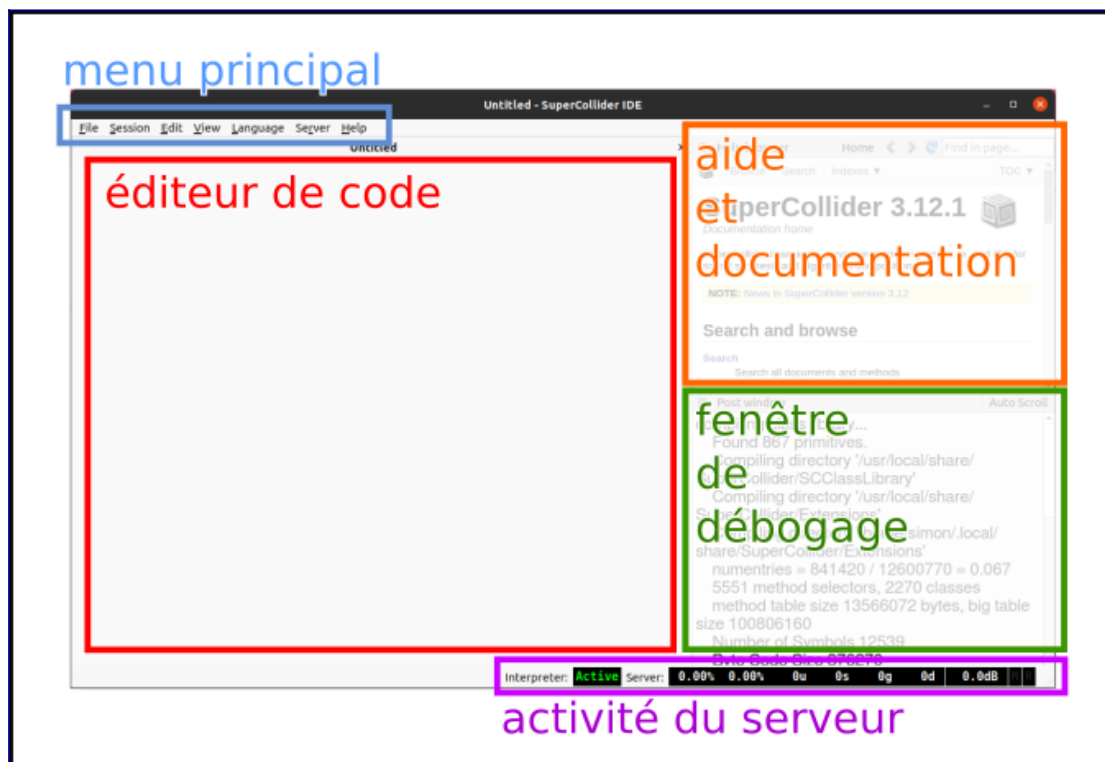
De la même manière, pour écrire en *sclang*, on peut utiliser n'importe quel éditeur de code, mais l'éditeur intégré par défaut, *scide*, a été développé spécifiquement pour ce faire. Encore une fois, ce n'est pas obligatoire, mais fortement conseillé.

Ce guide présente donc la manière d'utiliser *scide* pour écrire en *sclang* afin de manipuler le serveur *scsynth*.

La fenêtre principale

Voyons en premier ce qui s'affiche à l'écran lorsque que vous lancez SuperCollider : il s'agit de *scide*, l'IDE, l'éditeur de code intégré.

Voici les différents espaces qui le composent :



- le **menu principal** vous permet les opérations communes : manipulation de fichiers, paramétrages, infos...
- l'**éditeur de code** vous permet... de taper du code !
- la **fenêtre d'aide et de documentation** intègre la documentation de SuperCollider.

- la **fenêtre de débogage** est un terminal qui affiche les messages internes, comme les erreurs, et les informations que vous souhaitez afficher.
- la **fenêtre d'activité du serveur** indique un certain nombre d'information à propos du serveur.

Nous allons maintenant **cheminer tranquillement à travers les opérations basiques que vous pouvez effectuer avec l'IDE**. La plupart d'entre elles sont liées à **des raccourcis claviers**, que vous devrez sans doute mémoriser tant vous aurez à vous en servir.

La fenêtre de débogage

Premièrement, vous serez souvent amené à **vous servir de la fenêtre de débogage**, qui vous permettra d'obtenir des informations intéressantes sur ce qui se passe, notamment sur ce qui se passe mal...

Il y a déjà des informations qui y sont affichées : **lorsque l'IDE démarre, il vous donne des informations sur la manière dont il a chargé SuperCollider**. Par exemple la dernière ligne :

```
SCDoc: Indexed 1925 documents in 1.21 seconds
```

m'indique qu'il a trouvé 1925 pages de documentation, et qu'il les a chargées en 1 secondes 21.

Pour **remettre à zéro la fenêtre de débogage**, on utilise la commande **CTRL + MAJ + P** . C'est pratique pour éviter que les informations s'accumulent !

Écrire dans la fenêtre de débogage

Voyons maintenant **comment afficher des informations dans la fenêtre de débogage**.

Pour ce faire, il vous faudra vous servir de l'*éditeur de code*. Commençons par écrire un programme 'Hello World', dont le but est simplement d'afficher "Hello World" dans la fenêtre de débogage.

Voici la commande qui le permet, qu'il faudra taper dans l'*éditeur de code* :

```
"Hello World".postln;
```

Ensuite, pour *évaluer* le code, c'est-à-dire demander à SuperCollider de l'exécuter, il faudra utiliser la commande **CTRL + RETOUR** . « Retour » est la grande touche du clavier avec le symbole ↵, souvent nommée « Entrée ». En fait, il s'agit du retour chariot des anciennes machines à écrire.

Si vous avez eu de la chance, le texte s'est mis temporairement en surbrillance, et "Hello World" s'est affiché deux fois dans la *fenêtre de débogage*. Sinon...

Deux possibilités :

- vous avez mal tapé la ligne, et celle-ci contient une erreur. Dans ce cas, un grand message indigeste s'est affiché dans la *fenêtre de débogage*.

- Vous êtes revenu à la ligne après avoir tapé le code.

Contrairement à d'autres langages de programmation, **SuperCollider ne va pas évaluer l'ensemble du code écrit dans l'éditeur** lorsque que vous utilisez **CTRL + RETOUR**.

Considérons le code suivant :

```
"Je suis la première ligne".postln;  
"Je suis la deuxième ligne".postln;
```

On pourrait s'attendre qu'en appuyant sur **CTRL + RETOUR** s'affichent successivement "Je suis la première ligne", puis "Je suis la deuxième ligne".

Il n'en est rien. Par défaut, **CTRL + RETOUR** n'évalue que la ligne de code sur laquelle votre curseur de texte est placée. Dans cet exemple, elle n'évaluera donc qu'une seule de ces deux lignes.

Quel intérêt me direz-vous ? Eh bien dans le contexte musical, c'est assez pratique ma foi ! **SuperCollider est pensé pour la pratique du *livecoding***, c'est-à-dire la programmation en temps réel. Nous pouvons dans ce contexte avoir une ligne par instrument et les faire jouer indépendamment les uns des autres, ce qui est moins restrictif que d'avoir à exécuter *toutes* les instructions du programme quand on veut simplement modifier un paramètre. Mais nous y reviendrons.

Les régions de code

Comment faire pour exécuter plusieurs lignes à la fois ?

Deux solutions sont possibles :

Sélectionner les deux lignes et appuyer sur **CTRL + RETOUR**, ce qui implique qu'elles soient adjacentes.

Autrement, et dans la plupart des cas, **nous utilisons des parenthèses pour regrouper du code en un seul bloc**. Dans ce cas là, **CTRL + RETOUR** exécutera l'ensemble du code entre parenthèse si le curseur de texte est à l'intérieur :

```
(  
"Je suis la première ligne".postln;  
"Je suis la deuxième ligne".postln;  
)
```

Une suite de lignes de code regroupées par des parenthèses s'appelle une *région*.

Pour évaluer une seule ligne de code à l'intérieur d'une région, on utilise le raccourci **CTRL + MAJ + RETOUR**.

Syntaxe de séparation des instructions

Comme vous l'aurez sûrement remarqué, **on doit indiquer la fin des lignes de codes par un « ; » pour signaler à SuperCollider la fin d'une commande.** Un oubli résultera probablement en une erreur.

Note : il est théoriquement possible d'omettre le « ; » de la dernière instruction d'une région. Cependant, cela est lié à une convention syntaxique qui permet d'indiquer à un éventuel lecteur que la région renvoie le résultat de la dernière instruction. Si cela ne fait pas sens pour vous, vous devriez simplement terminer chaque instruction par « ; ».

Objets et Messages

Le langage de SuperCollider, *sclang*, suit un paradigme appelé *Orienté Objet*. Il n'est pas nécessaire de connaître cette notion en profondeur pour l'utiliser, d'autant plus que *sclang* prend certaines libertés avec ce paradigme, contrairement à d'autres langages plus 'rigides'.

Cependant, la notion d'*Objet* et de *Message* permet de mieux comprendre comment se servir du langage.

Pour schématiser, la plupart du temps, **on commence par créer un *Objet*, puis on le manipule à l'aide de *Messages*.**

C'est ce que nous avons fait dans notre exemple précédent, bien que cela soit un peu dissimulé.

```
"Hello World".postln;
```

est une forme simplifiée du code suivant :

```
(  
// Création de l'objet :  
var string = "Hello World";  
// Manipulation via un message :  
string.postln;  
)
```

Notez les deux lignes précédées de « // » : ce sont **des commentaires, c'est-à-dire des lignes qui seront ignorées par SuperCollider** lorsqu'il exécutera le programme. **Elle permettent au programmeur de laisser des indications à un lecteur** (y compris lui-même) à propos du code.

Dans cet exemple, nous créons d'abord l'*Objet* "Hello World", puis nous le manipulons à l'aide du *Message* `postln`, qui l'affiche dans la fenêtre de débogage. On dit que l'*Objet* "Hello World" répond au *Message* `postln`.

Deux syntaxes permettent d'envoyer des Messages aux Objets dans SuperCollider :

```
objet.message();  
message(objet);
```

Autrement dit, nous aurions également pu écrire

```
postln("Hello World");
```

Messages et arguments

La première syntaxe est généralement préférable, car **il arrive que des Messages prennent des arguments, des paramètres qui modifient le comportement du Message**, indiqué(s) entre parenthèses.

Par exemple, un Objet représentant un nombre à virgule peut être tronqué selon une certaine décimale :

```
3.14957.round(0.01);
```

Ici, l'Objet à tronquer et le paramètre sont assez clairs.

L'autre syntaxe les mélange :

```
round(3.14159, 0.01);
```

Message scramble

À titre d'exemple, voici un second Message auquel "Hello World" répond :

```
"Hello World".scramble;
```

scramble permet de mélanger les lettres d'une suite de caractères.

Veuillez-noter que si le résultat s'affiche dans la *fenêtre de débogage*, c'est parce que **la dernière instruction évaluée dans SC s'affiche toujours dans la fenêtre de débogage**.

Pour mélanger la suite de caractère, puis demander explicitement de l'afficher, nous pourrions faire comme-ça :

```
(  
var string = "Hello World";  
string = string.scramble;  
string.postln;  
)
```

Cependant, il est aussi possible de **‘chaîner’ les messages**. Ainsi, on peut avoir le même résultat avec la commande suivante :

```
"Hello World".scramble.postln;
```

L'Objet **"Hello World"** est un cas particulier, qui nous a permis d'aborder la question des Messages sans trop se compliquer. Mais certaines notions restent à expliquer avant que vous puissiez créer et manipuler des Objets par vous-même. Nous les expliquerons d'ici peu.

Avant de faire du son : sécuriser son audition



Les plus impatients, tout comme moi, en ont ras-le-bol des explications et aimeraient enfin faire du son. Mais avant ça, il est important d'aborder **les questions de sécurité relatives à l'audition**.

Comme tous les logiciels audio, l'utilisation de SuperCollider peut être dangereuse.

Certains logiciels audio intègrent des sécurités, ou ne permettent pas de faire des choses trop dangereuses. Ce n'est pas le cas de SuperCollider. Celui-ci est pensé implicitement pour des artistes ayant des notions en traitement du signal sonore, et conscience de ces dangers. Le parti pris est de laisser la plus grande liberté aux artistes en terme de design sonore, ce qui inclut la possibilité de se tromper et de produire un son démesurément fort, par exemple.

Parmi les dangers :

- **un son trop fort peut abîmer** votre matériel sonore, et surtout, **vos tympans**.
- **une écoute prolongée de son répétitifs**, surtout dans les aigus, **peut dégrader votre audition, même à faible volume**.

Quelques conseils de sécurité :

- en ayant coupé le son du PC, utiliser **CTRL + M** pour afficher les niveaux de sortie de SuperCollider. Cela permet d'**avoir un indice visuel de la puissance sonore, qui permet de vérifier si le son est trop fort via la vue, sans risquer de s'abîmer les oreilles.**
- tant que vous n'êtes pas à l'aise avec les outils de production de son de SuperCollider : **si vous n'êtes pas sûr du son qui va sortir de vos enceintes ou de votre casque : baissez le volume au maximum (de SuperCollider, du PC, ou du diffuseur), c'est-à-dire jusqu'au silence. Lancez votre son. Puis augmentez graduellement le volume. Répétez pour chaque nouveau son dont vous n'êtes pas sûr.**
- **si vous utilisez à la fois un micro et un haut-parleur qui diffuse le son issu du micro : attention au larsen**, qui pourrait advenir lorsque le son issu du haut-parleur est capté par le micro, provoquant une boucle de rétroaction.
- **si l'une de vos oreilles, ou les deux, commencent à être douloureuses, il est conseillé de s'arrêter immédiatement** et de passer à une activité dans le silence pour les laisser se reposer.

Premier son

Maintenant que les règles de sécurité sont posées, voici donc enfin **la première ligne de code qui va nous permettre de faire du son !**

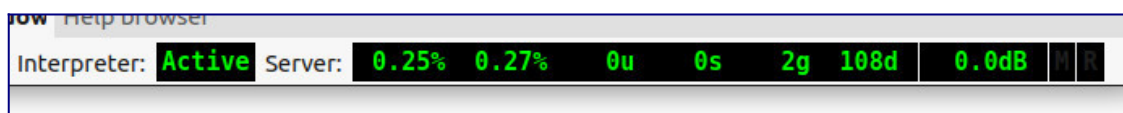
Enfin presque : l'exécuter maintenant n'aurait pas d'effet. **Lorsqu'on lance l'IDE, le serveur sonore de SuperCollider (*scsynth*) est éteint.** Il ne peut donc pas produire de son.

Pour l'allumer, on utilise en général le raccourci **CTRL + B** (ou **clic droit > Boot Server** sur la *fenêtre d'activité du serveur*).

Alternativement, on peut évaluer la commande :

```
s.boot;
```

Lorsque le serveur est allumé, c'est visible dans la *fenêtre d'activité du serveur*, qui affichera ses informations en vert :



Voici donc la ligne de code la plus simple pour jouer du son dans SC :


```
{ SinOsc.ar(440, mul: 0.1); }.play;
```

C'est en quelque sorte le 'Hello World' auditif de SuperCollider.

Avant de le décortiquer, voici **la commande qui permet d'arrêter tous les synthés du serveur** : **CTRL + MAJ + .** . Il paraît que le silence après du SuperCollider, c'est encore du Mozart.

Arguments : identifiant et valeur par défaut

Commençons par diviser pour mieux régner, et regardons seulement la commande à l'intérieur des accolades :

```
SinOsc.ar(440, mul: 0.1);
```

C'est une forme que nous avons déjà abordé : l'Objet `SinOsc` répond au Message `ar`, qui prend plusieurs arguments en entrée.

Pour être précis, `SinOsc` n'est pas exactement un Objet, même si on peut lui envoyer des Messages. Cela importe peu pour l'instant, nous aborderons cette question un peu plus tard, restons concentrés sur la question du son.

`SinOsc.ar()` est en fait **une commande qui crée un Objet**, et dans ce cas précis, un Objet sonore : **une sinusoïde, avec les paramètres associés aux arguments**. Ici, 440 indique la fréquence (un *la*, donc), et 0.1 le volume sonore. Je l'ai mis volontairement très bas pour éviter de vous abîmer les oreilles.

Dans certains cas, les arguments de certains Messages peuvent posséder une valeur par défaut, ainsi que des noms. C'est le cas du Message `ar` quand il est appliqué à `SinOsc` : les arguments se nomment `freq`, `phase`, `mul` et `add`, leurs valeurs par défaut sont 440.0, 0.0, 1.0 et 0.0.

Si un argument n'est pas spécifié, il prend sa valeur par défaut. Dans notre cas, `phase` est donc égale à 0.0, `add` à 0.0, et notre argument `freq` (le premier) aurait pu être omis car nous spécifions la valeur par défaut dans tous les cas. Seule la valeur de `mul` est réellement modifiée, car 1.0 est, à mon sens, beaucoup trop élevée. Pour rappel, tous les arguments n'ont pas forcément une valeur par défaut.

Si nous n'indiquons pas le nom des arguments, ceux-ci sont assignés implicitement de gauche à droite :

```
SinOsc.ar(600, 0.5, 0.2);
```

Ici, une sinusoïde avec une `freq` de 600, une `phase` de 0.5, 'puis' une `mul` de 0.2.

Dans notre cas, nous spécifions le nom de `mul` pour modifier cet argument précisément, plutôt que `phase`, qui est normalement le second argument, que nous souhaitons laisser à 0.0 :

```
SinOsc.ar(freq: 440, mul: 0.1);
```

freq étant le premier argument, indiquer son identifiant importe peu, même si cela rend quand même le code plus facile à comprendre.

Des fois, il est préférable de rendre le code plus lisible en revenant à la ligne entre chaque argument :

```
SinOsc.ar(  
    freq: 440,  
    phase: 0.0,  
    mul: 0.1,  
    add: 0.0  
);
```

SinOsc : la composante ‘fondamentale’ du son

Ceux qui connaissent le nom de Joseph Fourier (cocorico) savent peut-être que **tout signal sonore** (tout signal en général) **peut être décomposé en une somme de signaux sinusoïdaux**. Cela fait de SinOsc un des composants les plus utiles de SC, puisqu’en théorie, il permet de recréer n’importe quel son (en pratique, c’est galère).

Reformatons un peu notre exemple précédent, et ajoutons un petit quelque chose : si vous avez écouté l’exemple précédent sur un diffuseur stéréo, vous avez dû vous rendre compte que seul le canal gauche produisait du son. Voici comment régler ce problème :

```
(  
{  
    SinOsc.ar(  
        freq: [440, 440],  
        mul: 0.1  
    );  
}.play;  
)
```

Au lieu de spécifier un simple nombre comme paramètre de la fréquence, **nous utilisons un Objet nommé Array, qui est une sorte conteneur dans lequel on peut insérer plusieurs Objets**. On indique une Array en spécifiant les Objets contenus entre crochets ([]) (dans notre cas, deux nombres).

Lorsqu’une Array est utilisée comme paramètre d’un Objet sonore, elle duplique cet Objet autant de fois qu’elle ne contient d’Objets, en utilisant la valeur de chaque Objet pour la nouvelle copie (voir exemple suivant). Les copies créées sont ‘adjacentes’, et comme le premier Objet sonore est créé par défaut dans le canal audio gauche, la copie suivante est créée dans le canal audio droit.

Si l'on avait huit haut-parleurs, on aurait donc simplement à spécifier une `Array` contenant 8 valeurs pour diffuser le son sur l'ensemble du système. C'est plutôt pratique. On appelle cela l'**Expansion Multi-Canaux** (*Multichannel Expansion*).

Pour créer une `Array` contenant la même valeur un certain nombre de fois, il existe une syntaxe particulière :

```
440!2
```

équivalent à

```
[440, 440]
```

Je le mentionne ici car cette syntaxe est omniprésente dans les exemples de code que l'on peut trouver en ligne.

À partir de ça, voici **comment jouer sur la stéréo dans SuperCollider en introduisant une onde de battement**, grâce à un léger décalage de fréquence entre les deux canaux :

```
(
{
    SinOsc.ar(
        freq: [440, 441],
        mul: 0.1
    );
}.play;
)
```

Notez que *tous* les arguments peuvent être modifiés de la sorte :

```
(
{
    SinOsc.ar(
        freq: 440,
        mul: [0.1, 0.05]
    );
}.play;
)
```

Math un peu ça

De manière surprenante, et surtout, élégante, **beaucoup d'Objets de SuperCollider répondent aux opérations arithmétiques**.

Par exemple, **multiplier une `Array` multiplie chacun de ses éléments individuellement**. Ainsi, pour obtenir la quinte de notre *la*, *mi*, nous pouvons simplement multiplier la fréquence de base par 1.5 :

```
(
{
    SinOsc.ar(
        freq: [440, 440] * 1.5,
        mul: 0.1
    );
}.play;
)
```

Là où l'opération devient franchement plus rigolote, c'est que **les Objets sonores peuvent eux-mêmes être multipliés entre eux** :

```
(
{
    SinOsc.ar(
        freq: [440, 440],
        mul: 0.1
    ) *
    SinOsc.ar(
        freq: 1,
        mul: 1
    );
}.play;
)
```

Les opérations arithmétiques sur un Objet sonore modifient son volume. C'est pourquoi cet exemple équivaut à un LFO appliqué au volume du synthé.

Des synthés modulaires

Mais cette interchangeabilité entre les nombres et les Objets sonores ne s'arrête pas là : **au lieu de spécifier des nombres en tant que paramètres d'un Objet sonore, on peut spécifier... un autre Objet sonore !**

L'exemple précédent peut donc être obtenu également comme ceci :

```
(
{
    SinOsc.ar(
        freq: [440, 440],
        mul: 0.1 * SinOsc.ar(1)
    );
}.play;
)
```

Il est donc également **possible de modifier la fréquence via un Objet sonore** :

```
(  
{  
  SinOsc.ar(  
    freq: [440, 440] + (220 + (220 * SinOsc.ar(2))),  
    mul: 0.1  
  );  
}.play;  
)
```

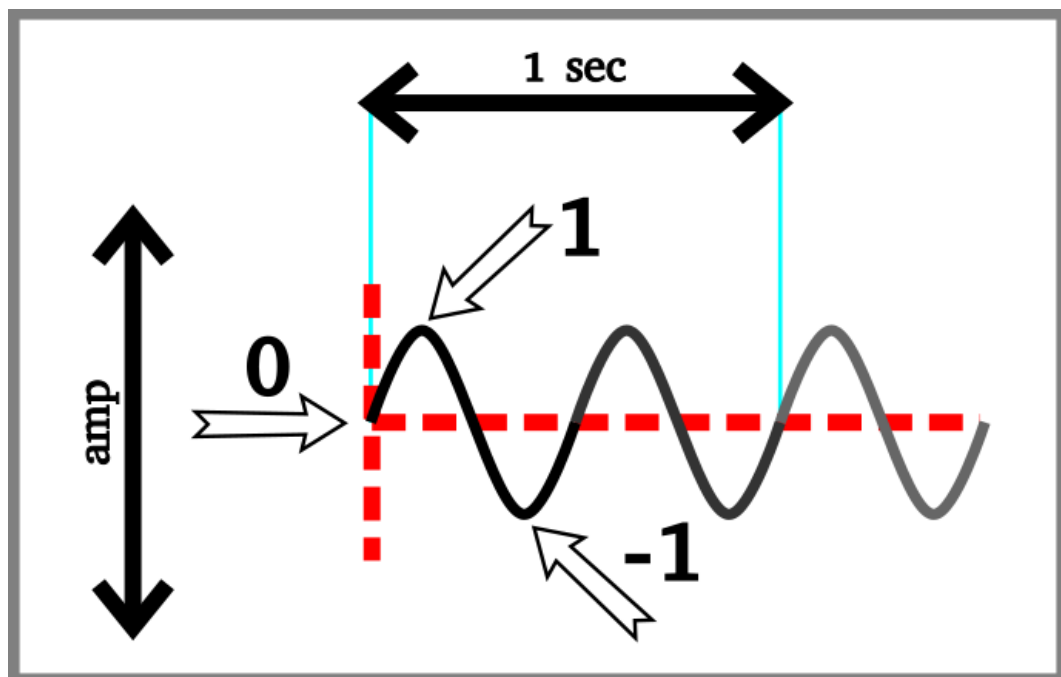
Regardons ce qui se passe de plus près :

`SinOsc.ar(2)`

Cette commande crée une sinusoïde de basse fréquence : deux fois par seconde, elle oscille autour de 0 : elle part de 0, elle monte jusqu'à 1, puis descend jusqu'à -1, avant de revenir à 0, selon la forme caractéristique d'une sinusoïde.

À noter : Cette variation de hauteur, de -1 à 1, est contrôlée par le paramètre `mul`. Des fois elle correspond au *volume* du son, mais dans notre cas ce n'est pas vrai, puisque la sinusoïde contrôle en fait la fréquence d'un autre son : cette sinusoïde n'est pas à proprement parler un son. **Le terme adéquat de ce paramètre est l'amplitude de l'onde, souvent abrégée en *amp*.**

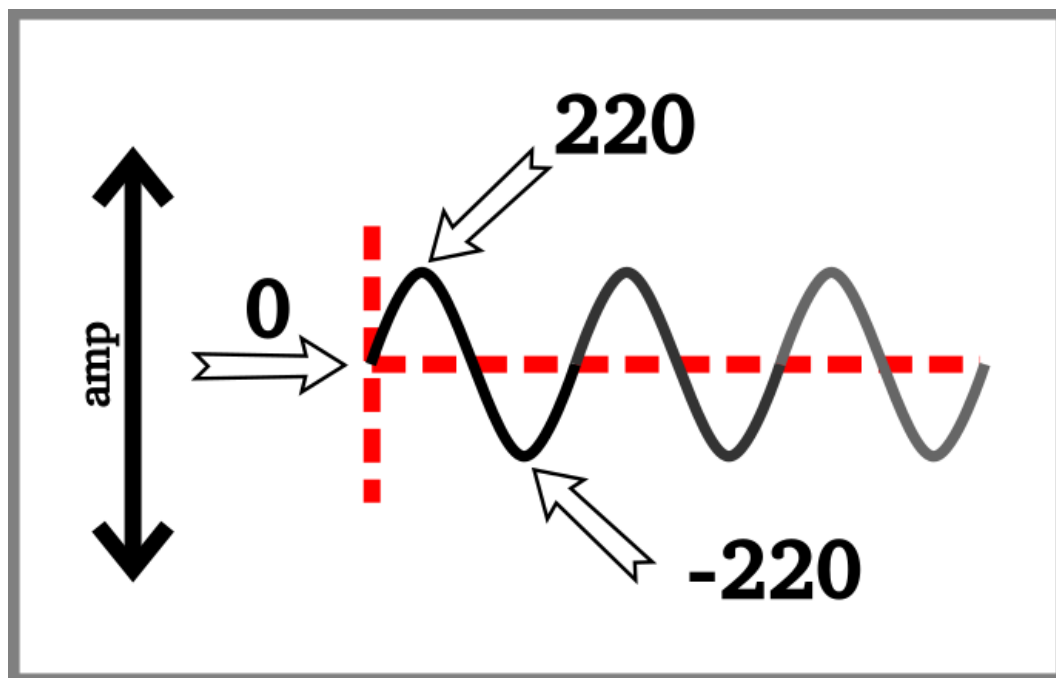
Voici une manière visuelle représenter de cette onde sonore :



Voyons maintenant la suite :

```
220 * SinOsc.ar(2)
```

Comme vu précédemment, **multiplier une onde sonore modifie son *amplitude***. Cette opération ne change pas la fréquence d'oscillation, mais au lieu d'osciller entre -1 et 1, l'onde oscille maintenant entre -220 et 220 :



En fait, nous aurions pu **directement utiliser le paramètre `mul`** pour obtenir ce résultat :

```
SinOsc.ar(2, mul: 220)
```

est équivalent à la commande précédente, mais peut-être un peu moins facile à comprendre au premier coup d'œil.

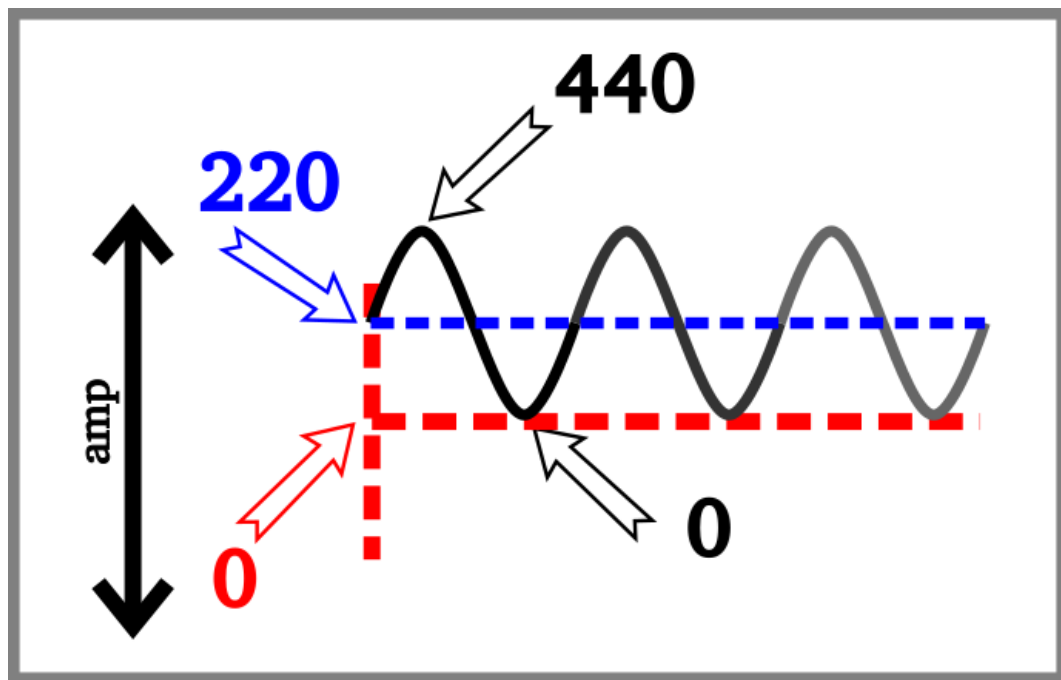
C'est ici qu'il faut être vigilant. Nous avons créé un Objet sonore avec un volume démesuré. Comme celui-ci n'est pas audible directement, puisqu'il contrôle un paramètre, pas de risque. Mais **si vous demandiez à SuperCollider de jouer ce son, vous vous retrouveriez avec un son dont le volume est 220 fois supérieur à la norme maximale**, ce qui est extrêmement dangereux pour vos oreilles.

La suite de la commande est une addition :

```
220 + SinOsc.ar(2, mul: 220)
```

L'addition ne modifie pas l'amplitude de la sinusoïde : elle oscille toujours entre -220 et 220.

Cependant, **elle modifie le pivot autour duquel elle oscille** : elle le fait maintenant autour de 220, ce qui amène l'amplitude à ne plus varier entre -220 et 220, mais entre 0 et 440 :



Là encore, **il existe un paramètre intégré pour modifier ce pivot** : `add` .

L'exemple précédent peut-être réécrit comme suit :

```
SinOsc.ar(2, mul: 220, add: 220)
```

Nous pouvons donc réécrire notre exemple de départ comme ceci :

```
(  
{  
  SinOsc.ar(  
    freq: [440, 440] + SinOsc.ar(2, mul: 220, add: 220),  
    mul: 0.1  
  );  
}.play;  
)
```

Lorsque l'on est à l'aise avec *sclang*, la lecture de ce code se traduit par :

‘Créer une sinusoïde dont la fréquence oscille deux fois par seconde entre 440HZ et 880HZ selon une forme sinusoïdale.’

Vers l'infini et au-delà

Comme les petits malins l'auront noté, il devient possible de modifier un Objet sonore par un Objet sonore lui-même modifié par un Objet sonore :

```
(
{
  SinOsc.ar(
    freq: [440, 440]
    + SinOsc.ar(
      freq: 2 + SinOsc.ar(0.333, mul: 2, add: 2),
      mul: 220,
      add: 220),
    mul: 0.1
  );
}.play;
)
```

De nouveaux jouets pour la récréation

Après une longue introduction (qui n'est malheureusement pas finie...), c'est le bon moment pour arrêter de lire et expérimenter un peu avec ces connaissances.

Mais avant ceci, trois nouveaux Objets sonore simples qui peuvent se substituer à *SinOsc* (ils possèdent les mêmes paramètres *freq*, *mul* et *add*) :

L'onde en dent de scie :

```
{ Saw.ar([440, 440], mul: 0.1); }.play;
```

L'onde triangulaire :

```
{ LFTri.ar([440, 440], mul: 0.1); }.play;
```

L'onde carrée :

```
{ LFPulse.ar([440, 440], mul: 0.1); }.play;
```


Simuler un Thérémine

Avant de poursuivre, un exemple emprunté à Nick Collins dans son ouvrage le *Petit Traité de SuperCollider* (*A Gentle Introduction To SuperCollider*).

Les objets `MouseX` et `MouseY` permettent de contrôler les paramètres d'un synthé selon la position du curseur de la souris, au sein des bornes qui leurs sont passées en arguments.

Cela permet d'**implémenter de manière très simple un thérémine** dans SuperCollider :

```
(
{
  SinOsc.ar(
    freq: MouseX.kr(220, 1760),
    mul: MouseY.kr(0, 0.25)
  );
}.play;
)
```

Encore une fois, n'hésitez pas à expérimenter en modifiant le code à l'aide des exemples que nous avons vu précédemment.

Fin de `{ }.play;`

Nous avons pris beaucoup de temps pour expliquer `SinOsc` et ses équivalents, sans même aborder le reste du programme que nous avons manipulé, à savoir le code qui enclot notre Objet sonore :

```
{ }.play;
```

Il s'agit d'un cas particulier de l'utilisation d'un Objet nommé `Function` (Fonction), que nous n'allons pas encore aborder, car cet exemple se prête assez mal à des explications.

Enclore des Objets Sonores dans `{ }.play;` est un outil spécial implémenté par les développeurs de *sclang* afin de pouvoir prototyper rapidement un signal sonore. Si l'on gagne en rapidité en utilisant cette méthode, on perd grandement en modularité.

En réalité, **pour produire du son dans SuperCollider, il faut respecter deux étapes distinctes :**

- **créer un modèle de synthé**, un peu comme si on inventait les plans d'un piano particulier
- effectivement **créer le synthé à partir du modèle**, ce qui lui permet d'émettre du son

`{ }.play;` , en arrière-plan, effectue ces deux étapes à notre place. Cependant, il ne nous permet plus ni d'avoir accès au 'plan', ni au synthé créé à partir de ce 'plan'.

Créer un modèle de synthé : la SynthDef

Créer un modèle de synthé et créer un synthé à partir de ce modèle sont des opérations effectuées par le serveur. Il faut donc que celui-ci soit allumé pour pouvoir ce faire.

Plutôt que de parler de modèle ou de plan, *sclang* parle de *définition* de synthé, et propose donc un Objet associé : la *SynthDef* .

La syntaxe de création est la suivante :

- un Objet de type `Symbol`, c'est-à-dire une suite de caractère précédé d'un anti-slash (`\`), qui permet d'**attribuer un nom unique** à la définition.
- un Objet de type `Function`, c'est-à-dire un bloc de code entre accolades (`{ }`), qui **définit les Objets sonores** qui composent notre définition.

En général, en plus de la créer, nous **rajoutons directement la définition au serveur pour que celui-ci la stocke et puisse l'utiliser**, en lui envoyant également le message `add` .

Ainsi, la *SynthDef* de notre exemple précédent devient celle-ci :

```
(
SynthDef(\helloWorld, {

    var sound = SinOsc.ar(
        freq: [440, 440],
        mul: 0.1
    );

    Out.ar(0, sound);

}).add;
)
```

Contrairement à la dernière fois, **il faut maintenant spécifier explicitement que nous voulons que le son sorte des enceintes**. C'est la commande :

```
Out.ar(0, sound);
```

qui s'en charge. L'argument `0` correspond au canal de sortie gauche, l'argument suivant correspond au son que nous voulons émettre, ici le contenu de la variable *sound*, à savoir une *SinOsc*. Comme l'Objet sonore *sound* est sujet à l'*Expansion Multi-Canaux*, puisque son paramètre de fréquence est une *Array*, le son sort également du canal droite (dont l'identifiant est `1`).

Créer un synthé à partir d'une SynthDef

Maintenant que le serveur sait à quoi devrait ressembler notre synthé, nous pouvons lui demander de le créer. Pour ce faire, un autre Objet est disponible : `Synth`. `Synth` prend le nom d'une `SynthDef` en argument et crée un synthé correspondant à celle-ci :

```
Synth(\helloWorld);
```

Comme celui-ci n'est pas référencé, il faut utiliser **CTRL + MAJ + .** pour l'arrêter. Réparons cette boulette **en le stockant dans une variable, qui permet de le manipuler *a posteriori***. Cependant, pour des raisons que nous verrons un peu plus tard, nous utiliserons pour l'instant des variables spéciales, composées d'une seule lettre (sauf la lettre `S`, qui est une exception), qu'il n'est pas besoin de déclarer :

```
x = Synth(\helloWorld);
```

Pour arrêter le synthé, on lui envoie le message `free` :

```
x.free;
```

SynthDef paramétrique

Notre `SynthDef` était beaucoup trop simple pour être maniable. Nous allons voir comment la rendre variable, en lui **rajoutant des paramètres**.

Il s'agit exactement de la même chose que les arguments que nous passons dans certains cas aux Messages. Pour les spécifier, on les ajoute juste après l'accolade ouvrante, enclos entre des barres verticales (`|`), en utilisant le caractère égal (`=`) pour leur assigner des valeurs par défaut :

```
(  
SynthDef(\helloWorld, { |freq = 440, amp = 0.1|  
  
    var sound = SinOsc.ar(  
        freq: [freq, freq],  
        mul: amp  
    );  
  
    Out.ar(0, sound);  
  
}).add;  
)
```

Ainsi, **les occurrences des chiffres qui spécifiaient nos paramètres** (comme 440 pour la fréquence), **sont remplacées par le nom de l'argument qui les modifie**.

Maintenant que ceci est fait, deux nouvelles possibilités s'offrent à nous.

D'abord, **modifier les paramètres dès la création du synthé** en indiquant les nouvelles valeurs, à côté de leur identifiant, au sein d'une `Array` lorsqu'on utilise `Synth` :

```
x = Synth(\helloWorld, [freq: 660, mul: 0.15]);  
x.free;
```

Mais il devient également possible de **réassigner les paramètres pendant que le synthé joue**. Attention, la syntaxe est un peu différente :

```
x = Synth(\helloWorld);  
x.set(\freq, 660);  
x.set(\freq, 880);  
x.free;
```

Cette nouvelle méthode permet également de faire cohabiter plusieurs synthés issus de la même définition côte à côte :

```
(  
x = Synth(\helloWorld, [freq: 440]);  
y = Synth(\helloWorld, [freq: 660]);  
)  
  
(  
x.free;  
y.free;  
)
```

Et de les manipuler indépendamment si besoin.

[EN CONSTRUCTION]