

Travaux pratiques d'informatique musicale, TP1 : Plugin synthétiseur

Licence 2, de physique et musicologie (version : 08/02/2024)

Frédéric Faure

Université Grenoble Alpes, France
frederic.faure@univ-grenoble-alpes.fr



Abstract

Dans ces **travaux pratiques** on utilise la librairie **JUCE** en C++ pour fabriquer des applications et plugins audio et midi. Dans ce TP1, on propose de créer un plugin synthétiseur (Midi→Audio) basé sur un **modèle physique** d'oscillateur non linéaire comme le modèle de **Van der Pol** ou l'attracteur étrange du **modèle de Rossler**. Ce projet s'adresse à des étudiants ayant des bases en musique, en mathématiques et en physique mais pouvant être débutant en programmation informatique.

Contents

1	Notion préalables de cours et TD	2
1.1	Signaux audio en informatique	2
1.2	Signaux Midi en informatique	2
1.3	Logiciel DAW	2
1.4	Logiciel Plugin (ou Greffon ou Module)	3
1.5	Représentation des nombres en informatique, binaire, hexadécimal, complément à deux.	3
1.6	Informatique, langage C++ et JUCE	4
2	Plugin synthétiseur	4
2.1	Résumé du projet	4
2.2	Installation de logiciels	4
2.3	On part d'un projet 'template'	5
2.4	Synthétiseur monophonique avec un son de sinus	7
2.5	Synthétiseur polyphonique avec un son sinus	8
2.6	Synthétiseur par intégration de l'oscillateur de Van der Pol	9
2.7	Synthétiseur basé sur le modèle chaotique de Rossler avec un attracteur étrange	13
A	Solutions aux exercices	15

B Installations de logiciels	15
B.1 Installation d'un compilateur pour C++	15
B.2 Installation de JUCE	16

Remark 0.1. Dans la version électronique de ce document (pdf), vous accédez aux pages de cours et autres documents en **cliquant sur les liens de couleur qui entourent le texte**.

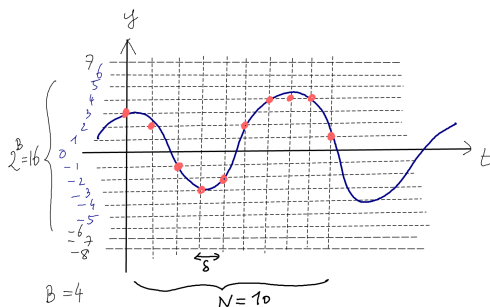
Le travail à faire en TP est indiqué sous la forme “Exercice (TP)”. En préalable à la séance, on conseille de lire attentivement ce document, de faire les “Exercices (TD)”, et d’installer les logiciels requis sur son ordinateur personnel (Appendice B). Ne pas hésiter à demander de l’aide technique à [ChatGpt](#) par exemple.

1 Notion préalables de cours et TD

Dans cette première partie on résume des notion importantes à connaître pour la suite du TP.

1.1 Signaux audio en informatique

- Un **signal audio** est un signal sonore (une fonction $t \in \mathbb{R} \rightarrow s(t) \in \mathbb{R}$) qui a été transformée en une suite de nombre à l’aide d’une carte son (on parle de **conversion analogique numérique**).
- Pour cela on sélectionne des valeurs discrettes du temps t espacés par un petit intervalle $\delta = \frac{1}{N}$, avec par exemple $N = 44100$ Hz intervalles par seconde, appelée **fréquence d’échantillonnage** ou “rate frequency”.
- A chacun de ces instants t , l’amplitude du signal $s(t)$ est approximée par un entier dans l’intervalle $(-2^{B-1}, 2^{B-1} - 1)$ où l’entier B est appelée “bit sampling”, par exemple $B = 4$ sur la figure ci-dessous, mais habituellement on prend $B = 32$ pour avoir une bonne précision. Cette approximation est parfois appelée **quantification du signal**



1.2 Signaux Midi en informatique

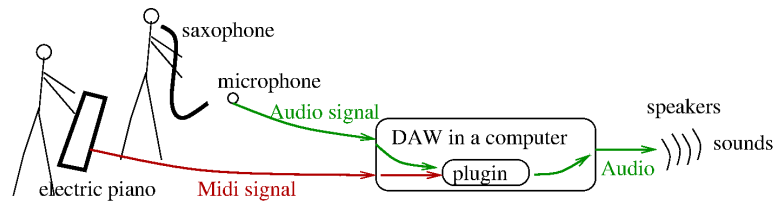
- Un **signal MIDI** est une suite de nombres qui représentent des instructions entre appareils musicaux (ex: claviers, **surfaces de controle**, **expandeur**, ordinateurs etc) en musique électronique.

Exemple 1.1.

- l’instruction de “jouer la note C de l’octave 5 avec un volume maximal” est représentée par la suite de nombres $\{144, 60, 127\}$ (car 144 signifie ‘note on’, 60 signifie ‘C5’ c’est à dire ‘C’ de l’octave 5, et 127 est le volume).
- l’instruction “éteindre la note C de l’octave 5” est représentée par la suite de nombres $\{128, 60, 127\}$ (car 128 signifie ‘note off’, 60 signifie ‘C’ de l’octave 5 et 127 signifie coupure nette du son).
- La liste des instructions MIDI est définie par la **MIDI association** et un **résumé des messages importants** que l’on étudiera plus tard.

1.3 Logiciel DAW

- Un logiciel **DAW** (signifie Digital Audio Workstation) est un logiciel qui permet **d’enregistrer des signaux audio et midi sur des pistes** (tracks) et de **rejouer ces pistes**, puis envoyer le son sur des enceintes.
- On proposera d’utiliser **Ardour**, qui est un logiciel DAW libre et performant. Dans le monde professionnel les ingénieurs du son utilisent fréquemment **Protools** sur windows PC ou **Logic-Pro** sur MacOS (mais ce sont des logiciels propriétaires).



1.4 Logiciel Plugin (ou Greffon ou Module)

- Un **audio plugin** est un logiciel qui peut être utilisé dans un DAW pour transformer des signaux audio et/ou MIDI.



- Voici des exemples de plugins où on précise la nature des signaux Entrée→Sortie:
 - Audio → Audio, sont des plugins appelés “AudioFX” (**effets audio**). Par exemple une “**réverbération**”, un filtre fréquentiel.
 - Midi→Audio, sont des plugins appelés “instruments”. Par exemple un **synthétiseur**.
 - Moins courants sont:
 - Midi→Midi, sont des plugins appelés “MidiFX” (Effet MIDI).
 - Audio,Midi→Audio, appelés “MusicEffect”.
 - Audio→Midi, par exemple pour détecter des notes d’instruments acoustique et les envoyer vers un synthétiseur (voir par exemple **Video on audio->midi**).

1.4.1 Latence et buffer audio

La perception humaine de l’audio accepte des **retards, appelé latence, inférieur à $L = 10\text{ms} = 0.01\text{s}$. mais pas supérieurs**. Cela est du au **temps de réponse des neurones** dans notre cerveau.

Ainsi un plugin Audio comme celui que nous réalisons nécessite de gérer des événements MIDI et Audio et faire des calculs sur une durée assez courte, inférieure à cette latence de 10ms. On appelle cela du **calcul en temps réel**. Pour cela, le logiciel DAW envoie (et reçoit en retour) au programme interne plugin les signaux MIDI et AUDIO de façon périodique par petits paquets appelés **buffer**.

Par exemple, nous avons vu en section 1.1 que la fréquence d’échantillonnage est $N = 44100\text{Hz}$ échantillons par seconde. Donc dans la durée de latence acceptable $L = 10^{-2}\text{s}$. il y a

$$n = N \times L = 441 \text{ échantillons.}$$

Ainsi on demandera au DAW d’envoyer et recevoir des buffers audio de taille $n = 441$ à chaque période $L = 10^{-2}\text{s}$. ou quelque chose de comparable.

1.5 Représentation des nombres en informatique, binaire, hexadécimal, complément à deux.

Ce sont des notions simples mais importantes sur la représentation des nombres, utilisés en informatique en général et plus particulièrement utile en informatique musicale. Voir l’annexe B4 de ce **cours d’acoustique musicale**. Pour rappel, voici une liste des premiers entiers codés sur 4 bits, très utile, qu’il faut savoir retrouver:

base 16	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
base 2	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
base 10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Exercice 1.2. Ecrire les messages midi de l’exemple 1.1 en hexadécimal et en binaire.

Solution 1.3.

Signification	Note on C5, vel=127	Note off C5, vel=127
décimal	144, 60, 127	128, 60, 127
hexadécimal	90, 3c, 7f	80, 3c, 7f
binaire	1001 0000	1000 0000
	0011 1100	0011 1100
	0111 1111	0111 1111

1.6 Informatique, langage C++ et JUCE

Ce projet nécessite des concepts assez évolués d’informatique comme la programmation temps réel, multi thread, multi plateformes et programmation objet en C++. Cependant le but est de réaliser un projet intéressant d’un point de vue physique, mathématique et musical en s’initiant à la programmation. Pour cette raison, **on ne demande pas de maîtriser à priori les concepts d’informatiques, on présentera leur logique au fur et à mesure**. Il faut savoir que ces concepts ne sont pas compliqués d’un point de vue logique mais sont souvent compliqués en apparence car leur syntaxe suit des conventions précises (l’informatique actuelle ne supporte pas la moindre erreur de syntaxe sinon le programme **bug**).

- Pour une initiation à C++ on propose le **tutorial** ou celui ci: **tutorial C++ orienté physique numérique**
- Pour approfondir on donnera des liens à la documentation des **classes C++ de JUCE** utilisées.

2 Plugin synthétiseur

Dans cette section on propose de **créer un plugin synthétiseur** (Midi→Audio) basé sur un **modèle physique d’oscillateur non linéaire** en commençant par le **modèle de Van der Pol** puis le **modèle de Rossler** qui a un attracteur étrange.

2.1 Résumé du projet

On rappelle les faits suivants qui motivent ce projet:

- La perception auditive humaine “apprécie” les **signaux périodiques** en temps dans les intervalles de fréquence [100Hz, 3000Hz], cela étant probablement du au fait que la voix humaine émet des signaux périodiques générés par la vibration des cordes vocales (voir [2, chap.3]).
- Par conséquent, les sons utilisés en musique sont de façon préférentielle périodiques (sauf pour les effets de percussions) et les instruments de musique sont conçus pour générer un son périodique (donc musical) selon l’un ou l’autre des mécanismes suivants (voir [2, chap.4]):
 1. “type système dynamique”: un système dynamique dissipatif non linéaire et possédant un attracteur qui est un **cycle limite**. Le mouvement de l’objet est donc périodique sur le cycle et par contact avec l’air cela produit un signal sonore périodique donc musical. C’est le cas du violon avec archet, trompette, flûte, voix, etc
 2. “type onde 1D”: Une **équation d’onde à une dimension**. L’onde se propage donc de façon périodique. C’est le cas de la guitare, piano, etc

L’idée du projet est de simuler un instrument de “type système dynamique”, c’est à dire un système dynamique dissipatif non linéaire ayant un cycle limite comme attracteur et de l’utiliser pour générer un signal périodique audio en sortie du plugin. En entrée du plugin, ce système dynamique sera déclenché par un message MIDI ‘note_on’ où le numéro de touche (ex: ‘C5’) déterminera sa fréquence et la force (la vélocité) déterminera la non linéarité et donc le timbre du son.

Pour commencer on va utiliser le modèle simple et très connu qui est le **modèle de Van der Pol** (ou plus généralement appelé un système d’**oscillations par relaxation**, ou **cycle limite**).

Objectif: L’objectif est de créer un plugin AU (pour MacOS) ou VST3 (pour Windows et Linux) qui réalise ce synthétiseur, qui soit pleinement utilisable dans un DAW, avec une interface qui montre en temps réel la trajectoire sur l’attracteur du système dynamique, et quelques **widgets** pour contrôler les paramètres. Pour simplifier, dans ce premier TP on utilisera notre programme de façon autonome (en mode appelé “StandAlone”) sans DAW.

Remark 2.1. Voici des liens sur le sujet de “**modèles physiques de synthétiseurs**”, exemples de sons à partir de l’attracteur de Lorenz: **youtube**, produits commerciaux **Sur youtube**, système chaotique en paramètres: **youtube**.

2.2 Installation de logiciels

Exercice 2.2. (TP)

1. Installer un compilateur C++ selon la section **B.1**

2. Installer JUCE en suivant les instructions de la section [B.2](#).

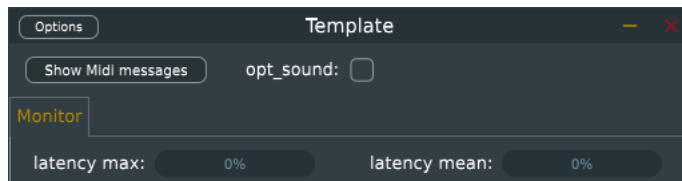
2.3 On part d'un projet 'template'

2.3.1 On télécharge le projet Template

“Template” signifie modèle. On va télécharger un projet de base qui est un plugin mais qui ne fait rien d'autre que de montrer les messages midi entrant et de jouer un son sinus fixe, mais que l'on va enrichir petit à petit dans les étapes suivantes.

Exercice 2.3. (TP)

1. Sur le site [juce_template.git](https://github.com/juce/template),
 - cliquer sur `Code/Download source code in zip` (par exemple)
 - sur votre ordinateur, avec le bouton droit, 'extraire l'archive' dans un répertoire de travail, par exemple 'TP'
2. Lancer le logiciel '**ProJucer**' (voir section [B.2](#)),
 - faire Menu/File/Open et choisir le fichier Template.jucer dans le répertoire qui vient d'être téléchargé.
 - faire Menu/File/Save Project and Open in IDE pour lancer le logiciel VisualStudio
3. Dans **Visual Studio** (qui vient d'être lancé)
 - Pour **compiler**, faire Menu/Build/Build Solution
 - quand c'est terminé, faire Menu/Debug/Start without debugging pour **lancer le programme** appelé “StandAlone” (cela signifie que c'est un programme indépendant de tout DAW. Cela sert à tester le projet). On voit la fenêtre suivante appelée **GUI** ('Graphical User Interface'):



- **Tester ce programme**, en observant les messages midi entrant (que l'on peut produire en branchant un clavier et cliquer Options/Audio_midi_settings/Active_Midi_inputs) et cliquer sur `opt_sound` pour entendre le son fixe produit, qui est un sinus de fréquence 440Hz.
- **Remarque:** les deux Jauges du bas (latency) montrent le pourcentage occupé par les calculs par cycle du processor. Ils doivent rester inférieur à 30%. On verra cela plus loin.

2.3.2 Observation des messages midi entrant

Brancher un [clavier externe](#). Avec ce plugin “Template”, vous pouvez observer les messages midi entrant. Il sont écrit en hexadécimal (voir section [1.5](#)).

Voici tout d'abord des précisions sur quelques messages midi importants pour nous (on utilise la notation hexadécimale):

Note-on: c'est le message constitué de 3 nombres

$$\{90 + c, k, v\}$$

où

- $c \in \{0, \dots, f\}$ est le numéro du canal midi utilisé par le clavier (attention sur le clavier l'affichage du canal est décalée de 1 et parcourt donc les nombres $1, \dots, 16$ en décimal). En général le clavier est configuré sur le premier canal $c = 0$.
- $k \in \{0, \dots, 7f\}$ est le numéro de touche. Par convention $k = 0$ est un 'C' de l'octave 0, très grave, que l'on notera C_0 , et ensuite on augmente k de 1 pour chaque demi-ton. Ainsi $D\flat_0$ est la touche $k = 1$, D_0 est la touche $k = 2$, etc, C_1 est la touche $k = 12$ en décimal, C_5 est la touche $k = 5 \times 12 = 60$ en décimal et $3c$ en hexadécimal. Le La du diapason A_5 de fréquence $f_{A_5} = 440\text{Hz}$ est la touche $k = 60 + 9 = 69$ en décimal et 45 en hexadécimal.
- $v \in \{0, \dots, 7f\}$ est la vélocité de la touche. Cela correspondra à l'intensité du son.

Note-off: c'est le message

$$\{80 + c, k, v\}$$

avec c, k, v comme ci-dessus, mais ici la vitesse v correspondra à un temps d'extinction du son.

Exercice 2.4. (TD et TP) Quels seront les messages midi successifs si on joue l'accord de Do Majeur? vérifier en TP avec le plugin "Template". **Solution:** A.1.

Remark 2.5. "Sur la numérotation des octaves". Attention, on a appelé ici A_5 de l'octave 5 le La du diapason de fréquence $f = 440\text{Hz}$, pour la simple raison que sa touche MIDI est $k = 5 \times 12 + 9$. En solfège il est conventionnel de l'appeler A_3 , i.e. il y a un décalage de 2 octaves avec la convention MIDI.

2.3.3 Présentation rapide du code C++

Dans la suite nous allons lire et éventuellement modifier le code C++ du programme. Pour cela, nous allons utiliser le logiciel VisualStudio (ou XCode sous MacOS).

Remark 2.6. Attention: avant de modifier le code C++, il est important de **fermer le logiciel Projucer de JUCE au préalable, sinon il risque d'annuler vos modifications.**

Nous avons expliqué en section 1.4.1 que le plugin doit gérer les événements audio et MIDI en temps réel avec une latence inférieure à 10ms (ou durée comparable). Or certains calculs peuvent prendre plus de temps, comme des dessins sophistiqués. Pour cette raison le plugin contient deux programmes qui tournent en parallèle, appelés **processus**:

- Un processus qui s'occupe des tâches rapides 'temps réel' qui nécessite une petite latence, comme la gestion des événements audio et midi. On appellera ce processus le '**processor**'.
- Un processus qui s'occupe des autres tâches qui peuvent prendre plus de temps, comme la réalisation d'un dessin ou la communication avec l'utilisateur via des boutons et **widjets** du **GUI** ('Graphical User Interface'). On appellera ce processus l'**editor**.

Le code C++ du programme se trouve dans le répertoire **Source**. Vous y trouvez:

- Le sous répertoire **editor** qui contient les programmes pour le processus editor.
- Le sous répertoire **processor** qui contient les programmes pour le processus processor en temps réel.

La fonction processBlock: C'est la fonction principale du plugin, où le travail est fait. (Avec VisualStudio, Menu/File/Open/File) ouvrir le fichier **Source/processor/PluginProcessor.cc** et y repérer la ligne suivante qui est le début d'une **fonction c++** appelée processBlock:

```
void Processor::processBlock(AudioBuffer<float>& buffer, MidiBuffer& midiMessages)
{
    //.. ici il y a du code c++
}
```

Cette fonction du processus processor du plugin est appelée périodiquement par le DAW. Elle échange avec le DAW en entrée et sortie les objets suivants.

- **buffer** : est un tableau de n nombres réels à valeur dans l'intervalle $[-1, 1]$. Ce sont les données audio (par exemple venant d'un microphone). Ici on ne s'en occupe pas car notre plugin ne prend pas d'audio en entrée. Par contre en sortie de la fonction on y mettra le signal audio de notre synthétiseur que l'on souhaite envoyer aux haut parleurs.
- **midiMessages** : est un tableau contenant éventuellement des messages midi en entrée. C'est ce qui nous intéresse dans cette première étape du projet.

Remark 2.7. Pour plus d'informations, la classe Processor est définie dans le fichier **Source/processor/PluginProcessor.h**. Elle dérive de la classe **juce::AudioProcessor**. Le buffer audio est de type **AudioBuffer**. Le buffer MIDI est de type **MidiBuffer**.

2.3.4 Modification du nom du projet

Dans la suite on va réaliser le projet avec plusieurs étapes. Il sera utile de conserver des étapes intermédiaires. Voici comment 'dupliquer' le projet actuel.

Exercice 2.8. (TP) Considérons ce projet qui est dans le répertoire 'juce_template-main'.

- Faire copier/coller ce répertoire et appeler le nouveau répertoire 'vanderpol_1'.
- Renommer le fichier vanderpol_1/Template.jucer en vanderpol_1/vanderpol_1.jucer
- Avec le logiciel 'Projucer', ouvrir le projet vanderpol_1.jucer
 - dans Menu/View/Show_project_settings, dans la ligne 'Project Name' mettre 'vanderpol_1' et de même plus bas dans la ligne 'Plugin Name'. Dans la ligne 'Plugin Manufacturer' mettre votre nom sans accent.
 - Sur la gauche dans Exporters/Visual_Studio/Debug et Release, dans la ligne 'Binary Name' mettre 'vanderpol_1'
 - faire Menu/File/Save_Project_and_Open_in_IDE pour lancer le logiciel VisualStudio
- Reprendre comme dans la partie 3 ci-dessus afin de compiler et executer ce projet.

2.4 Synthétiseur monophonique avec un son de sinus

Objectif: On va partir du projet 'vanderpol_1' comme expliqué dans la section 2.3.4. Dans cette première étape on va fabriquer un synthétiseur qui peut émettre une seule note à la fois (monophonique) en réponse à la touche du clavier actionnée. La travail va donc consister à adapter la fréquence et l'amplitude du signal audio produit en fonction des messages midi entrant.

Exercice 2.9. (TD) "Fréquence d'une note". A partir des informations données en section 2.3.2, trouver la formule mathématique qui donne la fréquence $f \in \mathbb{R}$ à partir du numéro de touche $k \in \mathbb{N}$. On rappelle que la note A_5 est la touche $k = 69$ et a la fréquence $f_{A_5} = 440\text{Hz}$ et que changer d'octave augmente k de +12 et multiplie la fréquence par $\times 2$. **Solution:** A.2.

Exercice 2.10. (TD) "Synthèse additive, Harmoniques"

1. Donner la formule qui exprime le signal sinusoïdal $s(t) \in \mathbb{R}$ de fréquence $f \in \mathbb{R}$ et amplitude $A > 0$ en fonction du temps $t \in \mathbb{R}$.
2. (Option) ajouter quelques termes "d'harmoniques", i.e. fréquences $2f, 3f$, etc. (Cela correspond à faire de la **Synthèse additive.**)

Solution: A.3.

Exercice 2.11. (TP)

- Rappel important: quand vous modifiez les fichiers depuis Visual-Studio (avec Ctrl-O), il faut que **Projucer soit au préalable fermé**, sinon il va annuler vos modifications.
- Dans le fichier **Source/processor/PluginProcessor.cc**, juste avant la fonction processBlock(), ajouter la fonction c++ suivante, et modifier la formule par la formule de l'exercice 2.9, donnant la fréquence f à partir du numéro de touche k . Aide: 2^x s'écrit en c++: **pow**(2., x)

```
//=====
// input: k key MIDI value
// output: f frequency
double key_to_f(int k)
{
    double f = 440; // change this formula
    return f;
}
```

- Dans la fonction processBlock, à la place de la partie existante intitulée "// - genere un son sinus" ajouter les lignes de code suivantes

```
//----- Loop over input MIDI messages
for (const MidiMessageMetadata &metadata : midi_buffer)
{
    MidiMessage message = metadata.getMessage();

    if (message.isNoteOn())
    {
        int c = message.getChannel() - 1; // shift channel
        int k = message.getNoteNumber();
        int v = message.getVelocity();

        f = key_to_f(k);
    }
}
```



```

        A = 0.1 * v/127.;
        t = 0;
    }
    else if (message.isNoteOff())
    {
        int c = message.getChannel() - 1; // shift channel
        int k = message.getNoteNumber();
        int v = message.getVelocity();
        A = 0;
    }
}

//--- Audio parameters
int N = getSampleRate(); // nombre echantillons / sec.
int n = audio_buffer.getNumSamples(); // nombre echantillons dans le buffer
int nchan = audio_buffer.getNumChannels(); // 1: mono, 2: stereo

//.....fill audio buffer .....
int ch = 0; // channel, mono
float* bufferPtr = audio_buffer.getWritePointer(ch); // pointeur du buffer,

for (int i = 0; i < n; i++) // audio samples
{
    bufferPtr[i] = 0; // put here the formula s(t)
    t = t + 1./N; // increment time
} // for i

```

- Modifier la ligne ci-dessus: `bufferPtr[i] = 0;` avec la formule de l'exercice 2.10.
- Dans le fichier **Source/processor/PluginProcessor.h**, ajouter la déclaration des variables f, A, t après le mot 'public': (attention la ligne double `t = 0;` est déjà présente, ne pas l'écrire deux fois).

```

public:
    double f = 440.; // frequency
    double A = 0.1; // amplitude
    double t = 0; // time

```

- Compiler et exécuter ce programme. Vérifier qu'il marche bien.

2.4.1 Solutions

Le projet demandé dans cette section se trouve ici: [solution vanderpol_1](#).

2.5 Synthétiseur polyphonique avec un son sinus

Il nous faut gérer une liste des notes actives (qui sonnent). Après un message 'note_on' le programme doit ajouter une note à cette liste et après un message 'note_off' il doit enlever la note si elle est présente dans la liste. Pour chaque note active, le programme doit mémoriser:

- Le numéro de touche k et la vélocité v de la note qui sont des entiers.
- La date t depuis le début de la note et sa fréquence f qui sont des nombres réels.

Au niveau programmation c++ on va utiliser la classe `vector` qui permet de gérer des listes de nombres (ou autre).

2.5.1 Programmation

Exercice 2.12. (TP)

1. Dans le fichier **Source/processor/PluginProcessor.h**, enlever la déclaration des variables f, A, t qui ne servent plus et ajouter à la place les déclarations de listes qui contiendront les valeurs de k, v, t, f :

```

public:
    vector<int> L_k, L_v;
    vector<double> L_t, L_f;

```

2. Dans le fichier **Source/processor/PluginProcessor.cc**, avant la fonction `processBlock`, ajouter la fonction c++ suivante qui permet de savoir si une valeur k est présente ou non dans une liste L et à quel endroit.


```
//=====
// input: k key MIDI value
// output: index of the value k in vector L if found,
//         -1 if not found.
int Is_note_in_the_list(int k, vector<int>& L)
{
    auto it = std::find(L.begin(), L.end(), k);
    return (it != L.end()) ? std::distance(L.begin(), it) : -1;
}
```

3. Dans la fonction processBlock() si on détecte un évènement 'note_on', ajouter les lignes de code suivantes à l'endroit adéquate

```
if(Is_note_in_the_list(k, L_k) == -1) // note is not in the list
{
    L_k.push_back(k); // add k to the list L_k
    L_v.push_back(v);
    L_t.push_back(0);
    L_f.push_back(key_to_f(k));
}
```

4. De même si on détecte un évènement 'note_off', ajouter les lignes de code suivantes à l'endroit adéquate

```
int pos = Is_note_in_the_list(k, L_k);
if(pos >= 0) // note is in the list
{
    L_k.erase(L_k.begin() + pos); //remove element at position pos in the list L_k
    L_v.erase(L_v.begin() + pos);
    L_t.erase(L_t.begin() + pos);
    L_f.erase(L_f.begin() + pos);
}
```

5. Un peu plus loin dans le code c++, au moment de remplir la valeur bufferPtr[i] du buffer audio, écrire les lignes suivantes

```
bufferPtr[i] = 0; // init buffer
for(int ik = 0; ik < L_k.size(); ik++) // loop on active notes
{
    double f = L_f[ik];
    double A = 0.1 * L_v[ik] / 127.;
    double t = L_t[ik]; // time from start of the note
    bufferPtr[i] += 0; // put here the formula s(t)
    L_t[ik] = t + 1./N; // increment time
} // for ik
```

en remplaçant la ligne `bufferPtr[i] += 0` par la formule de l'exercice 2.10.

6. Compiler et exécuter ce programme. Vérifier qu'il marche bien.

2.5.2 Solutions

Le projet demandé dans cette section se trouve ici: [solution vanderpol_2](#).

2.5.3 Variantes pour aller plus loin:

1. Ajouter quelques termes "d'harmoniques", i.e. fréquences $2f$, $3f$, etc. (Cela correspond à faire de la [Synthèse additive](#).)
2. Ajouter un [vibrato](#) et un [tremolo](#) aux notes (de fréquence 5 Hz et amplitude relative 0.2).
3. Ajouter une attaque des notes plus douce, et une fin de notes plus douce aussi.

2.6 Synthétiseur par intégration de l'oscillateur de Van der Pol

Dans cette section nous allons modifier le programme précédent et remplacer la formule explicite du signal sinusoïdal $s(t)$ par la solution d'un modèle physique d'oscillations. On va commencer avec le modèle de Van der Pol. Il va falloir que le programme résolve (i.e. intègre) numériquement le champs de vecteur du modèle de Van der Pol et transforme la trajectoire en signal audio.

2.6.1 Champ de vecteur de Van der Pol

Pour un paramètre $\mu \in \mathbb{R}$ fixé, c'est un champ de vecteur dans le plan $\vec{x} = (x_1, x_2) \in \mathbb{R}^2 \rightarrow \vec{V} = (V_1, V_2) \in \mathbb{R}^2$ dont voici l'expression des composantes:

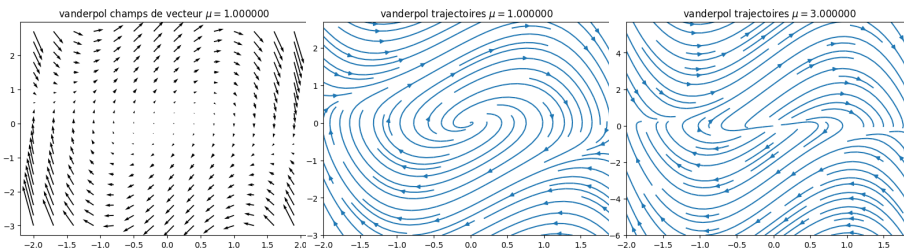
$$\begin{aligned} V_1(x_1, x_2) &= x_2 \\ V_2(x_1, x_2) &= \mu(1 - x_1^2)x_2 - x_1 \end{aligned} \quad (2.1)$$

Ce champs de vecteur détermine des **trajectoires** (ou flot) $\tau \in \mathbb{R} \rightarrow (x_1(\tau), x_2(\tau)) \in \mathbb{R}^2$ partant de conditions initiales quelconques $(x_1(0), x_2(0)) \in \mathbb{R}^2$ et d'après les équations de mouvement

$$\frac{dx_j}{d\tau} = V_j(x_1, x_2), \quad j = 1, 2. \quad (2.2)$$

Un tel modèle est appelé “**système dynamique différentiel** ou **flot**, défini par un champs de vecteur”.

Remark 2.13. Voici le champs de vecteur et des portions de trajectoires dessinées dans le plan (x_1, x_2) par ce **programme python**, par exemple sur **ce site en ligne**, pour $\mu = 1$ et $\mu = 3$.



On observe que toutes les trajectoires sont attirés vers une trajectoire périodique, appelée **cycle limite**, qui se déforme si μ augmente. Plus tard on verra que sa période $T_\mu \xrightarrow{\mu \rightarrow 0} 2\pi$. Pour avoir un effet sonore intéressant, on décide de fixer le paramètre μ à la vitesse v de la note par la formule (par exemple)

$$\mu = 10(v/127) + 0.03. \quad (2.3)$$

2.6.2 Résolution numérique par la méthode de Euler

Numériquement on va résoudre les équations de mouvement (2.2) par la **méthode de Euler**. Pour cela on fixe un petit intervalle de temps $\delta_\tau = 10^{-2} \ll 1$, et on approxime

$$V_j \stackrel{(2.2)}{=} \frac{dx_j}{d\tau} \approx \frac{x_j(\tau + \delta_\tau) - x_j(\tau)}{\delta_\tau}, \quad j = 1, 2. \Leftrightarrow x_j(\tau + \delta_\tau) = x_j(\tau) + \delta_\tau \times V_j \quad (2.4)$$

Donnant l'algorithme suivant.

- On pose $X_1 = x_1(0), X_2 = x_2(0)$: conditions initiales
- On itère N_τ fois les étapes suivantes
 1. On calcule V_1, V_2 d'après (2.1)
 2. On déduit les nouvelles valeurs d'après (2.4):

$$\begin{aligned} X_1 &= X_1 + \delta_\tau V_1 \\ X_2 &= X_2 + \delta_\tau V_2 \end{aligned}$$

2.6.3 Relation entre le temps τ du modèle et le temps t du signal

Le paramètre d'évolution du modèle noté τ est différent du temps t utilisé pour le signal audio. On va relier τ et t par $t = c\tau$ avec une constante $c > 0$ qui va dépendre de la fréquence f souhaitée de la note. On note T_τ la période du cycle limite et $T = \frac{1}{f}$ la période de la note. On souhaite donc $T = cT_\tau$ donc $c = \frac{T}{T_\tau} = \frac{1}{fT_\tau}$.

Rappelons que le temps entre deux échantillons audio est noté

$$\delta = 1/N$$

avec $N = 44100\text{Hz}$. Pour l'intégration numérique, on va **diviser cette durée δ en $M \in \mathbb{N}^*$ parties**.

Exercice 2.14. (TD) “Intégration, méthode de Euler”

1. Dédire l'expression de δ_τ à partir de f, T_τ, M, N .
2. On imposera que δ_τ soit le plus grand avec la contrainte $\delta_\tau < \frac{1}{N_{\text{Euler}}}$ avec un entier $N_{\text{Euler}} = 100$ fixé. Dédire l'expression de M .

Solution: [A.4](#)

Dans un premier temps, on ne connaît pas la période T_τ et on va donc mettre la valeur arbitraire $T_\tau = 2\pi$. Plus tard on la mesurera précisément.

2.6.4 Transformation de la trajectoire en signal audio

A la fin, on transforme la trajectoire obtenue $\vec{x}(t) = (x_1(t), x_2(t))$ en signal audio en prenant la première composante par exemple $s(t) = Ax_1(t)$, avec une constante A correspondant au volume (on pourrait aussi considérer une formule $\vec{x}(t) \rightarrow s(t)$ plus compliquée ou au choix de l'utilisateur).

2.6.5 Programmation

Exercice 2.15. (TP)

1. Pour **dessiner la trajectoire du système dynamique** sur le GUI, il faut au préalable faire les étapes suivantes
 - Télécharger les fichiers [graphics.cc](#) et [graphics.h](#) et les sauvegarder dans le répertoire “Source/editor/”
 - Télécharger les fichiers [com.cc](#) et [com.h](#) et les sauvegarder dans le répertoire “Source” (en remplaçant les fichiers existants).
 - Avec le logiciel Projucer, ouvrir le fichier *.jucer et dans la colonne de gauche “File_Explorer” avec le bouton droit sur “Source” cliquer sur “Add Existing Files” et ajouter ces deux fichiers [graphics.cc](#) et [graphics.h](#).
 - Au début du fichier Source/editor/manager.h sous l'instruction #pragma once, ajouter la ligne:
#include "Source/editor/graphics.h"
2. Dans le fichier **Source/processor/PluginProcessor.h**, ajouter les déclarations suivantes

```
//... for VanDerPol
double mu = 4; // parameter of the model
int N_Euler = 100; // number of samples in a period for Euler integration method
vector<double> L_x1, L_x2; // position of the point for each active note

//.... for copy of the trajectory to the editor
int N_tau = 1000; // number of iterations
vector<double> L_x1_copy, L_x2_copy, L_x3_copy; // accessed with atomic variable changes_Lx_copy
int i_copy; // index
atomic<bool> changes_Lx_copy = false; // 1: ask to refresh display
```

3. Dans le fichier **Source/processor/PluginProcessor.cc**, dans la fonction processBlock, à l'endroit où il y a détection d'une nouvelle note ('note_on'), ajouter les lignes suivantes:

```
L_x1.push_back(0.1);
L_x2.push_back(0);
i_copy = 0;
```

et si il y a un message 'note_off', ajouter les lignes suivantes:

```
L_x1.erase(L_x1.begin() + pos);
L_x2.erase(L_x2.begin() + pos);
```

Ajouter aussi dans la fonction processBlock,

```
//-----for editor -----
if(L_x1_copy.size() != N_tau)
{
    L_x1_copy.resize(N_tau);
    L_x2_copy.resize(N_tau);
}
```

Un peu plus loin, dans la boucle sur les notes actives décrite en [5](#), écrire à la place:

```

for(int ik = 0; ik < L_k.size(); ik++) // loop on active notes
{
    double f = L_f[ik];
    double T_tau = 2*M_PI; // approximate period of the cycle

    double mu = (L_v[ik] / 127.) * 10. + 0.03; // map v \in [0,127] to mu \in [0,10]
    int M = floor(N_Euler * f * T_tau / N); // subdivision of time step
    if(M<1)
        M=1;

    double x1 = L_x1[ik], x2 = L_x2[ik];
    double delta_tau = f*T_tau/(M*N);

    for(int im=0; im<M; im++) // subloop
    {
        //... formula for the vector field
        double V1 = x2;
        double V2 = mu * (1 - x1*x1) * x2 -x1;
        double x2_old = x2;
        x1 = x1 + delta_tau * V1;
        x2 = x2 + delta_tau * V2;

        //... copy to the editor
        if(ik == (L_k.size()-1)) // consider last note only
        {
            if(changes_Lx_copy.load() == false) // editor needs a new copy
            {
                L_x1_copy[i_copy]= x1;
                L_x2_copy[i_copy]= x2;
                i_copy++;
            }

            if(i_copy > N_tau) // end of the copy
            {
                i_copy = 0;
                changes_Lx_copy.store(true); // copy is done
            }
        }
    } // for im

    L_x1[ik] = x1; // memorizes
    L_x2[ik] = x2;

    double A = 0.02 * (L_v[ik] / 127.);
    bufferPtr[i] += A* L_x1[ik];
} // for ik

```

4. Dans le fichier **Source/editor/manager.h**, ajouter les déclarations suivantes

```

//-----
void Dessin(juce::Graphics& g); // make_gui = nl Window2(ZC, "Phase space", 500, 400) help="Show :
TCanvas c; // window associated to Dessin()

```

5. Dans le fichier **Source/editor/manager.cc**, ajouter les lignes suivantes

```

//=====
void Manager::Dessin(juce::Graphics& g)
{
    c.cx = p_com->Manager_Dessin->getX();
    c.cy = p_com->Manager_Dessin->getY();
    c.wx = p_com->Manager_Dessin->getWidth();
    c.wy = p_com->Manager_Dessin->getHeight();
    g.fillAll(Colour((uint)kBlack));
    c.Range(-2, -10, 2, 10); // x1,y1,x2,y2

    //-----Draw the trajectory-----
    for(int j=0; j< processor->L_x1_copy.size(); j++)
    {
        TMarker m(processor->L_x1_copy[j], processor->L_x2_copy[j], 3);
        m.SetMarkerColor(kYellow);
        m.Draw(c, g);
    }
}

```

6. **Compiler et exécuter ce programme.** Observer que les **notes ne sont pas justes** et si on appuie plus fort sur une touche, la fréquence diminue. Cela est dû au fait que le paramètre μ augmente d'après la remarque 2.13, et donc la période T_τ de la trajectoire augmente d'après la remarque 2.13. La période $T = 1/f$ du signal augmente donc et sa fréquence $f = 1/T$ diminue.

Pour corriger cela et stabiliser les fréquences de notes à la valeur souhaitée, il va falloir mesurer la période T_τ et compenser cet effet.

2.6.6 Mesure de la période T_τ du cycle limite

D'après les figures de la remarque 2.13, on observe que le cycle limite tourne dans le **sens indirect**. Pour mesurer la période du cycle limite, on va détecter les instants lorsque la trajectoire traverse le demi axe $x_2 = 0$ avec $x_1 > 0$.

Exercice 2.16. (TP)

1. Dans le fichier **Source/processor/PluginProcessor.h**, ajouter les déclarations suivantes


```
vector<double> L_tau, L_tau_cross_old, L_T_tau; // to measure the period
```
2. Dans le fichier **Source/processor/PluginProcessor.cc**, dans la fonction processBlock, à l'endroit où il y a détection d'une nouvelle note ('note_on'), ajouter les lignes suivantes:

```
L_tau.push_back(0); // time from start of note
L_T_tau.push_back(2*M_PI); // period of the cycle
L_tau_cross_old.push_back(0); // date of cross section
```

et si il y a un message 'note_off', ajouter les lignes suivantes:

```
L_tau.erase(L_tau.begin() + pos);
L_T_tau.erase(L_T_tau.begin() + pos);
L_tau_cross_old.erase(L_tau_cross_old.begin() + pos);
```

Juste après le calcul de la position (x_1, x_2) du point, ajouter les lignes suivantes:

```
//.. look if we cross the positive axis x1
if(x1 > 0 && (x2*x2_old < 0))
{
    double tau_cross = L_tau[ik] + (x2/(x2_old-x2))*delta_tau; // date of cross section
    double T_tau = tau_cross - L_tau_cross_old[ik]; // measured period
    if(T_tau < 0.1)
        T_tau = 2*M_PI;
    L_T_tau[ik] = T_tau; // memorize period
    L_tau_cross_old[ik] = tau_cross; // memorize date of cross section
} // if
L_tau[ik] += delta_tau; // increment effective time
```

et plus haut, remplacer la ligne

```
double T_tau = 2*M_PI; // approximate period of the cycle
```

par

```
double T_tau = L_T_tau[ik]; // measured period of the cycle
```

3. **Compiler et exécuter ce programme.** Observer que maintenant les **notes sont justes!** et que le timbre des notes dépend de μ et donc de la force de la frappe (vélocité).

2.6.7 Variantes pour aller plus loin:

- Modifier la formule $v \rightarrow \mu$ en (2.3) pour que le son soit plus doux et seulement très timbré si on appuie fort. Par exemple

$$\mu = 10(v/127)^E + 0.01.$$

avec un exposant $E > 0$. Avant on avait $E = 1$. Comme $(v/127) < 1$, on déduit que prendre $E > 1$ va rendre le son plus doux (par exemple $E = 3$), et $E < 1$ le rendre plus timbré.

2.6.8 Solutions

Le projet demandé dans cette section se trouve ici: [solution vanderpol_3](#).

2.7 Synthétiseur basé sur le modèle chaotique de Rossler avec un attracteur étrange

Le système dynamique précédent de Van Der Pol a un attracteur qui est un cycle limite. Cela fait que le son produit sera un signal parfaitement périodique et donc peut être un peu ennuyeux d'un point de vue musical. On peut améliorer cela en considérant un système dynamique où l'attracteur sera plus plus complexe comme un "attracteur étrange" mais avec un comportement "presque périodique dominant" afin de donner une impression de note musicale.

2.7.1 Champ de vecteur de modèle de Rossler

On propose d'utiliser le **modèle de Rossler** qui dépend de trois paramètres $a, b, c \in \mathbb{R}$. C'est un champ de vecteur dans l'espace $\vec{x} = (x_1, x_2, x_3) \in \mathbb{R}^3 \rightarrow \vec{V} = (V_1, V_2, V_3) \in \mathbb{R}^3$ dont voici l'expression des composantes:

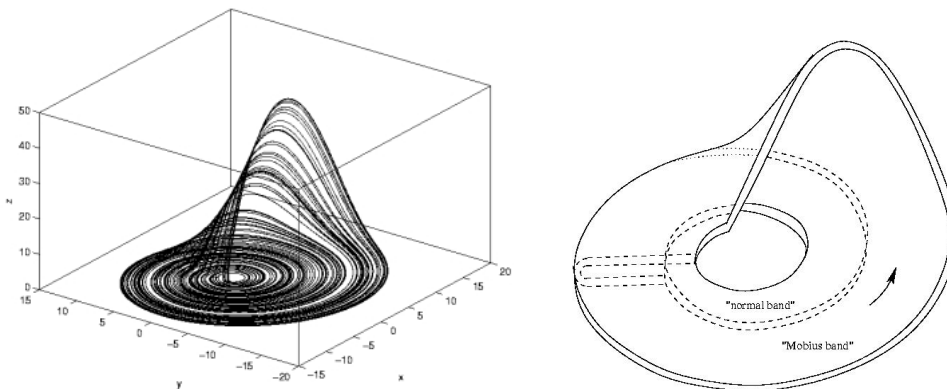
$$V_1(x_1, x_2, x_3) = -x_2 - x_3 \quad (2.5)$$

$$V_2(x_1, x_2, x_3) = x_1 + ax_2$$

$$V_3(x_1, x_2, x_3) = b + x_3(x_1 - c) \quad (2.6)$$

Comme expliqué dans la section 2.6.1, ce champs de vecteur génère des trajectoires $t \rightarrow \vec{x}(t)$ d'après les équations de mouvement $\frac{d\vec{x}}{dt} = \vec{V}(\vec{x}(t))$. Numériquement on intègre ces équations par la méthode de Euler.

Voici une image de l'**attracteur** pour les paramètres $a = b = 0.2, c = 5.7$. Les trajectoires, tournent dans le sens direct. Observer leur devenir qui semble imprévisible (et qui l'est!). Malgré l'apparence simple des équations, le comportement des trajectoires est très subtile, "chaotique" et fait l'objet de recherches actuelles.



Si on change les paramètres a, b, c , la géométrie de l'attracteur change, il y a des phénomènes appelés "**bifurcations**". Voici **une description** de la géométrie de l'attracteur si $a = b = 0.1$ et c varie. D'après ces figures, pour avoir de la dynamique dans le son, on pourra relier la vitesse v de la note au paramètre c selon

$$c = 4 + 16 \left(\frac{v}{127} \right)^3. \quad (2.7)$$

Référence: voir les **animations** et le cours de système dynamique [1, chap. Lorenz]. Voici une **page interactive sur le flot de Lorenz**.

2.7.2 Programmation

Exercice 2.17. (TP)

1. Dans le fichier **Source/processor/PluginProcessor.h**, ajouter les déclarations suivantes
`double a=0.1, b=0.1, c=18; // parameters of the model Rossler`
2. Dans les fichiers **Source/processor/PluginProcessor.h**, et **Source/processor/PluginProcessor.cc**,
 - (a) ajouter tout ce qu'il faut pour travailler maintenant avec trois composantes (x_1, x_2, x_3) , (V_1, V_2, V_3) alors qu'avant on n'avait que deux composantes.
 - (b) Remplacer les expressions du champ de vecteur par (2.5).
 - (c) Ajuster le paramètre c à la vitesse selon (2.7).
3. Pour le dessin de l'attracteur,
 - (a) dans le fichier **Source/editor/manager.cc**, dans la fonction Dessin(), ajuster le cadre x_1, y_1, x_2, y_2 du dessin d'après les figures ci-dessus.
 - (b) dans **Source/editor/manager.h**, augmenter le nombre de points: `int N_tau = 10000;`
4. **Compiler et exécuter ce programme.**

2.7.3 Variantes pour aller plus loin

- Pour améliorer le son, on peut ajouter une **enveloppe**, une réponse au **pitch-bend**, au **breath controller**, et autres effets.
- On peut explorer l'espace des paramètres a, b, c et leur effets sur l'attracteur pour modifier le son de façon dynamique.

2.7.4 Solutions

Le projet demandé dans cette section se trouve ici: [solution vanderpol_4](#).

A Solutions aux exercices

Solution A.1. de l'exercice 2.4, "messages MIDI"

```
90,40,36,  
90,43,33,  
90,3c,37,  
80,40,40,  
80,43,40,  
80,3c,40,
```

Solution A.2. de l'exercice 2.9, "Fréquence d'une note"

$$f = 440 \times 2^{(k-69)/12.0}$$

```
double f = 440.0 * pow( 2.0, (k-69)/12.0 );
```

Solution A.3. de l'exercice 2.10, "Synthèse additive, Harmoniques"

1. C'est

$$s(t) = A \sin(2\pi ft)$$

2. Par exemple pour les trois premières harmoniques

$$s(t) = A \left(\sin(2\pi ft) + \frac{1}{2} \sin(2\pi (2f)t) + \frac{1}{3} \sin(2\pi (3f)t) \right)$$

Solution A.4. de l'exercice 2.14, "Intégration, méthode de Euler"

1. Le pas d'intégration δ_τ utilisé en (2.4) est

$$\delta_\tau = \frac{1}{c} \left(\frac{\delta}{M} \right) = \frac{fT_\tau}{MN}, \quad (\text{A.1})$$

2. On choisit $M \geq 1$ le plus petit entier de sorte que

$$\delta_\tau < \frac{1}{N_{\text{Euler}}} \Leftrightarrow \frac{fT_\tau}{MN} < \frac{1}{N_{\text{Euler}}} \Leftrightarrow M > N_{\text{Euler}} \frac{fT_\tau}{N}.$$

Donc

$$M = \max \left(1, \left[N_{\text{Euler}} \frac{fT_\tau}{N} \right] \right),$$

où $[.]$ signifie la partie entière.

B Installations de logiciels

B.1 Installation d'un compilateur pour C++

On aura besoin d'un environnement de programmation pour la **langage C++**. Selon votre ordinateur, suivre les instructions suivante.

B.1.1 Pour Windows

- Installer **Visual-Studio** option Community, avec le “Workloads Desktop development with C++”, qui est un environnement de programmation pour coder en C++.

B.1.2 Pour MacOS

B.1.3 Pour Linux

B.2 Installation de JUCE

JUCE est un ensemble de programmes en **C++** qui permettent de concevoir des plugins audio pour Linux, MacOS, Windows, Android, etc.

Pour approfondir on peut suivre des **tutoriaux de JUCE**.

B.2.1 Pour Windows, JUCE avec Projucer

- Sur le site de juce.com/download/, cliquer sur JUCE et Projucer pour windows, et extraire le fichier téléchargé. Cela crée le répertoire JUCE
- On peut déjà essayer JUCE/**DemoRunner** (cliquer sur “More Info/Run anyway” ou “informations complémentaires/Executer quand même”), onglet Browse Demos, et par exemple **Audio/PluckedStringsDemo** et cliquer sur les cordes de la harpe pour entendre le son. Prendre le temps d’essayer différents programmes de démonstration.
- Lancer JUCE/**Projucer** (cliquer sur informations complémentaires/Executer quand même)
 - La première fois, Cliquer dans Menu/File/Global_Paths, indiquer le chemin des répertoires de JUCE et JUCE/modules.
Remark B.1. Projucer est un logiciel proposé par JUCE qui permet de définir son projet de programmation.
- Création d’une application à partir d’un exemple existant:
 1. Dans Projucer
 - (a) choisir Menu/File/Open example/Audio/PluckedStringsDemo
 - (b) Menu/File/Save_Project_and_Open_in_IDE : sauvegarder ce projet dans un répertoire existant. Puis cela ouvre le projet dans le logiciel Visual Studio.
 2. Dans **Visual Studio** (qui est déjà lancé)
 - (a) Pour **compiler**, faire Menu/Build/Build_Solution et quand c’est terminé, faire Menu/Debug/Start_without_debugging pour **lancer le programme**.
 - (b) Remarque: le programme executable créé est dans le chemin Builds/VisualStudio/x64/Debug/App/P

B.2.2 Pour MacOS

B.2.3 Pour Linux

References

- [1] F. Faure. *Cours de systèmes dynamiques*. [link](#), 2016.
- [2] F. Faure. *Cours d’acoustique musicale. Niveau Licence 3*. [link](#), 2020.