Académie de Montpellier Année scolaire

Lycée Nevers Montpellier 2018 - 2019



# Introduction

# Cahier des charges

debut realiser une guitare virtuelle pour representer des accords sur le manche d'une guitare et jouer le son correspondant à ces accords sur l'ordinateur.

Pour jouer le son nous avons rapidement choisi d'utiliser la sortie midi de notre ordinateur qui comprend des instructions de la forme « jouer la note 60 (Do4) sur le canal 0 avec un volume de 130 » ou « mettre l'instrument 0 (piano accoustique) sur le canal 0 » ce qui est joue en midi est moins joli mais utiliser de vrais sons neccessiterais d'avoir un fichier sond par note et par instrument ce qui n'est pas possible pour nous.

Nous avons commencé a utiliser midi pour créer la musique mais écrire des listes d'accords à jouer était un travail énorme de copie et calcul alors qu'on peut trouver toutes les chansons qu'on veut dans des fichiers midi (fichier binaire qui codent des séquences d'instructions midi).

Aussi, au niveau de l'instrument virtuel nous avons vite abandonné l'idee de la guitare qui posait de grosses dificulté dans la représentation des accords sur le manche d'une guitare (par exemple la note 60 (Do4) se trouve à plusieurs endroits sur le manche et il y a plusieurs facons de jouer l'accord majeur de DO4 (les notes 60, 64, 67) . On ne peut pas contruire un accord de guitare en connaissant que les notes par exemple, pour l'accord C4 (Do) composé de C4, E4, G4 mais l'accord le p.

On veut jouer accord maj de C4 cad [C4E4G4] on pourrait calculer des positions differentes de guitare qui font exactement ces notes par exemple accord 1=[X,X,0,2,3,X] ou accord 2=[X,X,0,2,X,8] mais dans ces solutions il y en a une qui est impossible accord2 à jouer physiquement. Et accord 1 n'est pas joué comme ca on joue accord 1bis =[01023X] mais cet accord rajoute les notes E5 et C5 ca ne gène c'est meme plus joli mais comment le savoir. Autre facon accord3=[8,8,9,10,10,8] qui rajouttent E5 C5 et C6 mais enlevent E4.

On aurait pu avec beuacoup de travail renter toutes les bibliotheque d'accords sur python, mais il serait resté un probleme quand on veut jouer un accord, quel accord de la bibliotheque uyiliser ca depend en fait de l'accord joué avant pour faciliter le deplacement des doigts. On ne savait pas comment faire

Nous avons donc finalement décider de représeter un piano virtuel qui a l'avantage d'avoir une touche associé a chaque note et nous permet ainsi d'eviter les problemes evoqué ci dessus.

Au bout d'un ou deux mois d'essais et de test nous avons pu decider du cahier des charges suivant :

Fichier midi

Décodage d'un fichier midi

recuperation et traduction de toutes les infos codées en binaire dans le ficher midi

stockage de ces infos traduites dans un objet python que nous appelons un album

Lecture d'un album

comprehention des données contenues dans l'album

envois de ces donnée sur une sortie midi pour jouer la musique

Interface de lecture

interface graphique Tkinter permettant de conroler la lecture

ouverture fichier, play, pause, accelerer

PIANO

interface graphique pour les touches d'un piano

liaison avec sortie midi pour entendre ce que l'on joue

liaison avec la lecture d'un album pour voir sur le piano les notes jouées

# Rapport technique

## 3.1 Généralités

#### organisation du code

Au fur et a mesure qu'on avancait dans le projet le fichier.py dans lequel on codait devebait illisible et on arrivait pus à corriger les erreurs et a corriger le code nous avons donc decidé d'organiser le code de la facon suivante :

* Utilisation de modules : tout notre code est contennu dans un repertoire pyanote qui contient des fichiers nom.py appelés modules chaque module contient un ensemble de fonctions servant a resoudre un probleme particulier par exeple le module pistes.py contient les fonctions necessaires à la construction d'une liste d'évènements en lisant une piste dans un fichier midi. Quand on veut utiliser une fonction d'un de nos modules depuis un autre module, il suffit de faire un import pyanote.nom\_module éventuellement avec un as pour raccourcir le nom d'appel de la fonction. Nous avons eu des problèmes avec from pyanote.nom\_fichier import \* quand des fonctions de differents modules avaient le meme nom. Pour pouvoir faire ceci il a fallu rajouter un fichier vide\_\_init\_\_.py dans le répertoire pyanote (voir <https://python-guide-pt-br.readthedocs.io/fr/latest/writing/structure.html>)
* test du code : chaque module a souvent du etre testé le problème est que si on laisse le test dans le module, quand on importe le module dans le programe final, le test est exécuté (on ne veut pas lancer les tests de tous les modules dans notre programme final). Cependant enlever/commenter les tests demande à réécrire ou decommenter les tests des qu'on modifie le code du module. La solution choisie est de mettre tous les tests d'un moduledans la patie (if \_\_name\_\_ == ''\_\_main\_\_'' :) ainsi le test s'exécule que si on execute directement le module et pas dans un fichier qui l'importe. Tous nos modules contiennent ces tests
* commentaires : quand on travaille à plusieurs. Il est necore plus important de commenter le code tous les modules commencent par un commentaire qui décrit ce que fait le module et les fonctions contiennent elles aussi des commentaires indiquant ce qu'elles font nous avons suivi pour ca les conventions PEP227 de <https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python/235263-de-bonnes-pratiques> ceci perment dans certains editeurs (visualstudio code) de voir les comentaires de la fonction quand on tape so nom dans un autre module.

### Outils python que nous avons découvert

Au cours de notre projet nous avons découvert des fonctionnalités de python qui nous ont été bien utiles.

* Les dictionnaires : semblables aux listes mais plus pratiques pour certaines utilisations. Par exmple pour nous un evenement est une liste E=[temps,numéro,message] on accède au temps en utilisant E[0]. C'est pratique et rapide tant qu'on a pas des listes trop grandes. Nous avons par exemple un objet controleur qui contient plus de 30 éléments differents. Il est impossible de programmer en se souvenant des indexs de chaquns de ces éléments. C'est dans ces cas la que nous avons choisi d'utilister des dictionnaires. Par exemple, si on a un controleur C={''index\_chanson'':0,''index\_evenement'':1299,''vitesse'':1.2,…} en tapant C[''vitesse''] on récupere 1.2 et en tapant C[''vitesse'']=1.5 on change la vitesse dans le controleur C. Ceci nous a été très utile et on l'a utilisé un peu partout. <https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python/232273-utilisez-des-dictionnaires>
* les erreures : nous avons déjà vu en cours l'existance d'erreures par exeple quand on tape 1/0 python affiche une erreure dont le type est ZeroDivisionError nous pouvons créeer nos propres erreures. Par exeple au début de la lecture d'un fichier midi notre code est if fichier.read(4) != b'Mthd' : raise TypeError (''ce n'est pas un fichier midi'') ainsi notre programme s’arrête si il y a une erreure. Cependant les erreures peuvent aussi nous poser des problèmes par exemple otre interface grafique va ouvrir des fichiers midi mais si on ouvre un fichier midi qui n'est pas un fichier midi ou un fichier défectueux (ce qui nous ait arrivé) on ne veut pas que l'interface s'arrete. Le mécanisme que nous avons utilisé est la recuperation des erreures. En tapant le code :

def **changer\_fichier**():

fichier = askopenfilename()

try:

album = creer\_album(fichier)

except ValueError:

return

jouer\_album(album)

lorsque on veut changer de fichier, on essaie (try) de créer un album. Si ca marche on ne regarde pas ce qu'il y a dans la partie except et on continue la programme qui joue l'album. Si ca ne marche pas, l'erreur est interceptée par le except et n'arrête pas le programme on execute le code qui est dans la partie except (ici on fait un return pour sortir de la fonction qui ne fera donc rien).

* Les fonctions anonymes : Elles ont été necessaires lors de l'utilissation de Tkinter en effet certaines widgets de Tkinter lorsqu'ils sont modifiés, appellent une fonction qui n'as pas d'arguments

### Lecture du son

Pour ceci nous avons importé le module pygame.midi (<https://github.com/pygame>). On aurait voulu utiliser le module Mido (<https://mido.readthedocs.io/en/latest/index.html>) qui avait l'avantage de faire la lecture d'un fichier midi (ce qui nous aurait économisé énormément de travail) le problème est que Mido utilise une librairie que nous n'avons jamais réussi à l'installer.

Le module pygame.midi nous permet d'ouvrir une sortie son de la manier suivante :

pygame.midi.init()

identificateur = pygame.midi.get\_default\_output\_id()

sortie\_son = pygame.midi.Output(identificateur)

Une fois qu'on a la sortie son, on peut lui envoyer un message, par exemple :

sortie\_son.write\_short(0xC3, 27, 0)

sortie\_son.write\_short(0x93, 60, 240)

time.sleep(2)

sortie\_son.write\_short(0x83, 60, 240)

chaque message envoie un statut (un entier qu'on comprend mieux sous forme hexadecimale) et deux arguments. Dans le premier message, le statut 0xC3 veut dire changer d'instrument (C) sur le canal 3, le nouvel instrument est 27 (guitare jazz). Le changement d'instrument ne necessite qu'un seul argument donc le dernier est à zero. Le deuxième message a pour statut 0x93 et veut dire jouer une note (9 = note on) sur le canal 3. cette note est un Do4 (60) d'un volume 240 (max = 255). Apres avoir dormi deux secondes le programme éteindre la note (8 = note off).

Remarques importantes :

* toute note commencée par un note on doit être éteinte par un note off sinon elle continue à jouer (même si on ne l'entend pas). Une exception, les notes jouées sur le canal réservé à la batterie (on n'a pas besoin de les éteindre).
* La sortie midi n'a que 16 canaux différents numérotés de 0 à 15. Le canal batterie est le numéro 9.
* Nous avons décrits ici des messages de contrôle que nous expliquerons mieux dans la partie fichier midi. Mais on peut aussi envoyer des messages systèmes qui comportent uniquement une chaine binaire. Ces messages systèmes sont définis par les constructeurs de materiel midi. Par exemple, si Yamaha construit un clavier midi ce clavier va envoyer à la sortie midi de Yamaha des messages qu'il va pouvoir comprendre. Si on a connecté notre programme à la sortie midi de microsoft, il ne comprendra pas le message système de yamaha et va l'ignorer sans faire d'erreurs dans pyanote, nous avons décider d'envoyer à la sortie midi tous les messages systhème sans les comprendre car STANDARE. Pour les messages contrôle on aurait pu envoyer les messagees a la sortie midi sans les comprendrent mais nous avons besoin d'en comprendre certains par exemple pour changer la couleir d'une touche du piano quand la note est jouée

Dans le module pyanote.son.py nous avons juste écrit de petites fonctions qui permettent d'utiliser de facon plus simple les fonctions de pygame.midi. Ce fichir comprend aussi des tests pour comprendre les messages.

### Notes et accords :

Une note en midi est représentée par un entier augmenter l'entier de un revient à modifier la note d'un demi ton (vers les aigüs) dasn le module pyanote/note.py nous avons écrit des fonctions qui permettent de transformer un numéro de note midi en une de chaine caractère (par exemple 60 donnera Do4 et inversemment de transformer une chaine de caractère représentant une note en entier une note en midi*VOIR COMMENT*)

**print**(nombre\_vers\_note(82))

**print**(nombre\_vers\_note(82, "b"))

**print**(nombre\_vers\_note(82, "#", FR))

ce programme affichera A#5, Bb5 puis La#5, en effet, la note 82 est un la# (A# en anglais) à l'octave 5. Mais, dans la notation tempérée qu'on utilise aujourd'hui en musique, le la# et le sib (A# et Bb) sont la même note dans le premier appel on a traduit le nombre 82 avec les paramètres par défaut qui sont # (préféré) et langage = anglais (EN). Dans le deuxième appel on a indiqué qu'on préférait bémol et la langue par défault restait l'anglais dans le troisième appel on a indiqué que le préféré était # et la langue le français.

Pour la fonction inverse note\_vers\_nombre il n'y a qu'un paramètre optionnel la langue.

Dans tout notre projet nous ne manipulons les notes que sous la formes d'entiers le seul moment ou nous utilisons ce module est pour afficher le nom des notes sur les touches d'un piano quand la note est jouée.

Répartition du travail :

* Lisa Baget et Matthieu Durand algorithme de base
* Matthieu Durand ajout des différentes langues

ACCORDS

Dans le module pyanote/accords.py, nous avons écrit une bibliothèque d'accords sous la forme d'un dictionnaire. Par exemple

ACCORDS = {

"aucun": [0],

"majeur": [0, 4, 7],

"mineur": [0, 3, 7],

Dans cet exemple on voit qu'un accord est une liste de nombres qui comprend toujours le nombre 0 et éventuellement d'autres nombres. La fonction construire accord prendra comme paramètre une note et le nom d'un accord et construira la liste de notes qui forme cet accord pour cette note. Par exemple si on appelle construire\_accord(82) avec ''majeur'' ça nous donnera la liste [82+0 ,82+4,82+7] c'est à dire [A#5, D6, F6] qui est l'accord de A#majeur.

Répartition du travail :

* Lisa Baget et Matthieu Durand

## lecture fichier

Pouvoir déchiffrer un fichier midi à été un énorme travail rendu necessaire car nous n'arrivions pas à utiliser Mido. Ce travail a pourtant été instructif car il nous a obligé à nous plonger dans les spécifications du format midi (bien expliqué dans <https://github.com/colxi/midi-parser-js/wiki/MIDI-File-Format-Specifications>) et à utiliser toute la partie de notre cours d'ISN consacré au codage binaire. Le resultat est contenu dans quatres modules de pyanote :

* pyanote/utils.py contient toutes les fonctions pour lire des infos élémentaires dans un fichier binaires par exemple lire\_entier permet de lire un entier sur un nombre connu d'octet et la fonction lire\_entier\_variable nous permet de lire un entier codé sur un nombre inconnu d'octets.
* Pyanote/resume.py permet de construire un dictionnaire que nous avons appelé résumé à partir d'un fichier midi ce résumé contient des informations essentielles par exemple le nombre de piste du fichier midi, à quel octet commence et fini chaque pistes. La construction du résumé ets extrèmement rapide puisqu'on ne lit pas toutes les informations contenues dans le fichier (on ignore tous les événements midi en utilisant la fonction seek(nombre d'octets).
* pyanote/piste.py un fichier midi contient une ou plusieurs pistes. Ce module permet de construire une liste de touts les événements contenus dans une piste. Nous avons choisi de représenter chaque événement par une liste : [delta temps, numéro piste, message].
* Pyanote/album.py enrichi un résumé en ajoutant la liste des chansons. En effet les pistes ne sont pas interprétées de la même manière suivant le format d'un fichier midi :
  + 0 (monopiste), le fichier ne contient qu'une chanson dont ses événements sont ceux de la piste 0.
  + 1 (multipistes simultanées), le fichier ne contient qu'une seule chanson qui est obtenue en fusionnant toutes les pistes du fichier.
  + 2(multipistes successives), le fichier contient plusieurs chansons une pour chacune des pistes du fichier. Ce cas est prévu dans notre code mais nous n'avons jamais pu le tester car nous n'avons pas trouvé d'exemple de fichier midi de format 2.

Dans cette partie nous allons d'abord expliquer le format midi puis nous citerons quelques parties de notre code python qui nous au posé le plus de problèmes. Comme tout le code et commenté nous pensons que les autres fonctions ne devraient pas poser de problèmes de comprehention.

### Le format MIDI :

Ici explications sur le format midi

### pyanote.utils : lire\_entier\_variable OCTET A REVOIR

Le format MIDI définit une facon de coder un entier sur un nombre variable d'octets. C'est ce qui est appelé VARINT dans la spécification. Ces entiers variables sont principalement utilisés pour représenter les intervales de temps entre deux événements MIDI. La plupart du temps, ces durées seront courtes et il n'y aura besoin que d'un octet pour les stocker. C'est dans les rares cas où la durée est longue qu'ils seront stockés sur plusieurs octets ainsi l'utilisation de VARINT économisera de la pl ace et des octets.

Le principe de VARINT est simple : tant que le premier bit (le bit de poids fort) est à 1, il faut lire l'octet suivant. Lisons un premier octet o1= 129. Le premier bit de cet octet est à 1 car 129 >128

(en effet si le premier bit est à 1 le nombre est superieur à 1000 0000(2) = 27 (10) = 128(10)). Aussi, il faut lire l'octet suivant o2=145 comme 145>=128 nous lisons également l'octet o3 = 25 et comme 25<128 on s’arrête et on ne lis plus rien. Le VARINT que nous cherchons est codé sur les entiers o1,o2,o3. Cependant le premier bit de o1 et o2 ne sert poas à code l'entier mais sert à dire de continuer la lecture. Il faut donc faire comme si il n'etait pas la. Pour enlever ce bit il suffit d'enlever 128 au nombre. On a o'1 = o1-128 ,o'2 = o2-128 et o'3 = o3 (puisque son bit de poids fort est déjà à 0). les nombres o'1, o'2 et o'3 ne sont plus vraiment des octets car ils sont réélement codés sur 7 bits. Chacun ne peut coder que 27 = 128 nombres possibles et le nombre VARINT est donc o'1\*128\*\*2 + o'2\*128+o'3 c'est ce qui est calculé par notre fonction vlire\_entier \_variable.

def **lire\_entier\_variable**(fichier):

entier = 0

octet = **ord**(fichier.read(1))

while octet >= 128:

entier = entier \* 128 + octet - 128

octet = **ord**(fichier.read(1))

return entier \* 128 + octet

remarque : ord(fichier.read(1)) lit une chaine binaire de longueur 1 et la transforme en un entier. La fonction ord ne peut marcher que sur 1 octet.

### pyanote.pistes:lire\_message\_controle

Une des plus grosses difficultées que nous avons rencontré est lié à ce que midi appelle le running status l'explication du format midi que nous utilisions principalement n'en parlait pas et donc nous n'avions pas géré ce running satus dans notre code. La lecture de piste marchait sur certains fichiers et pas sur d'autres. C'est en lisant d'autres explications du format midi comme celle de : <http://www.gweep.net/~prefect/eng/reference/protocol/midispec.html> que nous avons appris l'existance de ce running status

A chaque fois qu'on lit un message de contrôle dans le fichier, il faut enregister le statut de ce messsage (on n'enregistre que le dernier) au prochain message de contôle si le statut qu'on lit ne correspond pas à celui d'un message de contôle, alors il faut comprendre que le vrais statut de ce message est celui qu'on avit sauvegardé, et que ce qu'on vient de lire est en fait le premier argument du message.

Nous présentons ci dessous une version implifiée des fonctions de piste.py qui montrent comment nous avons géré ce running status

def **lire\_message\_controle**(fichier, sauvegarde):

statut = **ord**(fichier.read(1))

*#l'instruction est codée par les 4 premiers bits du statut*

instruction = statut // 16

if instruction in **range**(8, 15): *# pas de running status*

sauvegarde[0] = statut *#mide a jour de la sauvegarde*

arg1 = **ord**(fichier.read(1))

else: *#besoin du running statut*

arg1 = statut

statut = sauvegarde[0]

*#ici il faut tester qu'il faut un 2eme argument*

arg2 = **ord**(fichier.read(1))

return [statut, arg1, arg2]

Cette fonction est appellée dans notre boucle de lecture de tous les événements (elle s'appelle creer\_\_piste) où nous initialisons la variable sauvegarde par sauvegarde = [None]. En effet au début il n'a a pas de sauvegarde et nous mettons cette information dans une liste pour quelle puisse être changée dans une autre fonction(ce qui ne serait pas possible si l'on avait essayé avec un entier). Une chose n'est pas montrée dans ce code simplifié : quand on lit un message système, il faut anuler la sauvegarde (on fait sauvegarde[0] = None) mais quand on lit un message meta ce n'est pas clair de savoir si il faut ou pas annuler la sauvegarde (les personnes qui explique,t le format ne sont pas toujours d'accord). Nous avons décidé& d'annuler la sauvegarde dans ce cas et pour l'instant aucune erreure n'a été trouvée.

### pyanote.album : fusionner\_pistes

Nous avons vu que quand on a un fichier MIDI de format 1 ce fichier correspond à une unique chanson obtenue en fusionnant toutes les fiches du ficchier. La fusion de ces pistes sera une unique liste d'événemenyts qui aura la même signification que si l'on jouait toutes les pistes en même temps.DONNER UN EXEMPLE POWER POINT.Imaginons que la fonction creer\_piste nous ait donné deux pistes :

P0 = [ [0,0,A], [0,0,B], [20,0,E], [20,0,H] ]

P1 = [ [0,1,C], [10,1,D], [10,1,F], [10,1,G], [10,1,I] ]

Dans ces pistes, nous avons remplacé les messages par des lettres pour le pas encombrer l'exemple avec des messages qui n'influencent pas la fusion.

Les premiers éléments de chaque événement sont des temps relatifs, se lisent de la manière suivante : [20,0,E] veut dire que E devra être traité 20 ticks (unité de mesure du temps chez lees événements MIDI) après le message de l'événement précédent (B). Le problème est que ces delta temps n'ont de sens que dans une piste et ne peuvent être utilisés pour comparer des evenements dans des pistes differentes.

Une première étape de la fusion va donc être de transformer ces temps relatifs en temps absolus qui pourront etre comparés car ils seront dans le même référentiel temporel.

def **transformer\_temps\_absolu**(piste):

temps = 0

for evenement in piste:

evenement[0] = evenement[0] + temps

temps = evenement[0]

Après avoir appliqué cette fonction les temps dans P0 et P1 seront des temps absolus.

P0 = [ [0,0,A], [0,0,B], [20,0,E], [40,0,H] ]

P1 = [ [0,1,C], [10,1,D], [20,1,F], [30,1,G], [40,1,I] ]

Nous avons longuement travaillé pour écrire une fonction de fusion qui fusionnerait ces deux pistes de façon efficace. Nous avons fait un premier essais en rajoutant la première piste à la première puis en appelant une fonction de tri. Mais la fusion etait tres longue… En cherchant sur internet, nous avons trouvé une méthode plus efficace pour le faire (<https://www.geeksforgeeks.org/merge-two-sorted-arrays-python-using-heapq>) en utilisant la fonction merge de ce module heapq nous obtenons presque la liste fusionnée que nous voulions.

P = [ [0,0,A], [0,0,B], [0,1,C], [10,1,D], [20,0,E], [20,1,F], [30,1,G], [40,0,H], [40,1,I] ]

Pourtant, dasn cette liste, les éléments ne sont pas fdes événements car ils expriment des temps absolus et pas des temps relatifs (qui sont prévu dans le format MIDI) nous avons donc du écrire une fonction similaire à la précédente qui remplace les temps absolus par des temps relatifs.

Toutes ces fonctions sont dans le module pyanote.album qui permet de crééer un album qui est en fait un dictionnaire. Les clés de ce dictionnaire sont toutes celles d'un résumé plus une clée chanson qui contient une liste de toutes les fonctions du fichier.

## Jouer un fichier MIDI

### Lecture basique et gestion du temps

Grâce à nos modules pyanote.son et pyanote.album nous pensions aboir aucun problème avec la lecture d'un fichier midi. La fonction de lecture (voir version détaillée dans archives/l1\_lecteur\_basique.py) devait ressembler à ce qui suit :

def **jouer\_album**(album, sortie\_midi):

for chanson in album["chansons"]:

for evenement in chanson:

temps = calculer\_temps(album, evenement[0])

time.sleep(temps)

message = evenement[2]

if est\_un\_message\_système\_ou\_controle(message):

jouer\_message(sortie\_midi, message)

elif est\_un\_message\_meta\_changeant\_tempo(message):

mettre\_a\_jour\_tempo(album, message)

La difficulté ici est de bien gerer le temps. Les delta temps dans les événements sont en ticks. Dans le header du fichier midi on a lu une valeur de tempo H en ticks/beat (1 beat est la durée d'une noire). Les messages nous indiquent des changements de tempo M en microseconde/beat. Quand on connaît ces deux valeurs si on divise M/H on obtient une mesure T en micro/ticks. Quand on multiplie un delta temps en ticks par T on obtient une durée en microseconde. En divisant cette durée par 10\*\*6 on obtient bien le temps t en seconde qu'il faut donner comme parametre de time.sleep(t). Ainsi quand on doit mettre à jour le tempo, on recalcule T à partir du H (qu'on a récupéré dans notre dictionnaire à la création du résumé avec la clé ''ticks/noire'' )

et a partir du M qui est le deuxieme argument du message et on stocke ce nouveau T dans la clé ''micros/tick'' de notre dictionnaire album.

Il reste un problème : si on a pas recu de message de changement de tempo, quel est la valeur de M ? Le format MIDI nous dit que cette valeur correspond à 120bpm calculons cette valeur en microseconde/noire.

120noires/minute = 1/120 minutes/noire= 60/120 secondes/noire= 0,5\* 10\*\*6 microssecondes/noire

la valeure par défault de M est donc 500000.

Pour tester ce lecteur, vous pouvez lancer archive/l1\_lecteur\_basique.py.

### La lecture avec after :

Nous avons eu un problème en voulant appeler la lecture d'un fichier midi depuis l'interface graphique tKinter. En effet la fonction de lecture est une fonction très longue (elle dure la durée du morceau) et quand un événement tKinter lance cette lecture, aucun autre événement ne peut être traité : notre interface graphique est en freeze. C'est ce qui est montré dans le fichier archives/l2\_lecteur\_basique\_avec\_interface.py

Nous avons essayé une première solution en utilisant la fonction after de tKinter, en effet notre fonction de lecture entre deux évènements MIDI fait un time.sleep(t) dépendant de l'intervalle de temps t entre les deux événements MIDI. Nous avons essayé de remplacer ceci par la fonction after de Tkinter en appelant Tk().after(t, continuer\_lecture).

def **continuer\_lecture**(chanson, i , album):

evenement = chanson[i]

traiter\_evenement(evenement)

if i + 1 < **len**(chanson):

ev\_suivant = chanson[i+1]

temps = calculer\_temps(album, evenement[0])

Tk().after(temps, continuer\_lecture, chanson, i+1, album)

Avec cette solution notre fonction de lecture est découpée en pleins de petites fonctions continuer\_ lecture qui s'appellent les unes les autres par l'intermediaire de la fonction after. La derniere ligne du programme ci dessus veut dire que continuer\_lecture(chanson, i+1, album) sera appelée après un temps temps. Cette fonction semble faire la même chose que la version dans lecture basique mais il y a une difference importante. La fonction time.sleep utilisée dans la premiere version est bloquante : elle ne laisse pas au programme le temps de faire autre chose et provoque le freeze. LA fonction after de Tkinter est non bloquante et laisse à tKinter le temps de gerer se événements. Nous pensions avoir ici la bonne solution mais la fonction after est beaucoup moins précise que la fonction sleep (after(130) peut en fait créer un décalage de 150 secondes voire plus) en écoutant le morceau on ne le reconnaissait plus le rythme.

### Lecture dans un thread et utilisation d'un controleur :

On nous a conseillé une autre solution qui est le threading. Avec cette solution le programme principal contient notre interface graphique tkinter et la lecture du fichier midi s'exécute en parallèle dans un ''thread''. Dans la lecture, on fait bien les time.sleep qui sont précis et python distribue efficacement le temps entre le programme principal et le thread de lecture (voir archives/l3\_principe\_utilisation\_thread.py pour l’exemple qui nous a été fourni).

import threading

thread = threading.Thread(None, jouer\_album, None, [album, sortie\_midi])

thread.start()

**print**('le thread est vivant')

Dans ce programme, nous avons utilisé la fonction de lecture basique mais en appellant cette fonction dans un thread elle se déroule en parallele du reste du programme (le programme principal). En particulier, la chaine 'le thread est vivant' s'affiche de suite et n'attant pas la fin de l'exécution de la fonction lecture. Avec cette méthode, tKinter ne frise plus et comme la fonction de lecture utilise time.sleep, les intervalles de temps sont correctement respectés.

Communication du programme principale vers le thread :

Nous ne voulons pas seulement lancer la lecture dans un thread et attendre la fin de cette lecture. Nous voulons modifier cette lecture depuis l'interface graphique du programme principale (pour mettre la lecture en pause, arreter la lecture, stopper un piste, faire un retour en arrière). Pour ce faire, nous profitons que le programme principale est le thread ont la même mémoire. Si le programme principale et le thread partagent un dictionnaire que nous allons appeler le controleur, ce controleur va contenir toutes les variables qui déterminent l'exécution de la lecture. En modifiant une de ces variables depuis le programme principal on modifiera la lecture. Par exemple le dictionnaire contient une clé ''pause'' qui a normalement pour valeur False et une clé ''fin'' qui a aussi comme valeur False . Il faut écrire la fonction de lecture pour qu'elle utilise ces clés.

def **bloucle\_lecture**(controleur):

while not controleur['fin']:

if controleur['pause']:

time.sleep(0.1)

else:

*#traiter prochain événement*

Avec cette version simple de la boucle de lecture qu'il faudra lancer dans un thread, il est possible de controler la lecture depuis le programme principal si 'fin' et 'pause' sont à Faulse la boucle de lecture traitera tous les événements ( et au dernier événement la boucle de lecture mettra fin à True). Par contre si le programme principale met 'pause' à True à chaque tour de boucle le thread fera un sleep de 0,1 seconde et vérifiera au prochain tour de boucle si la valeur n'a pas été changé. Le cintroleur que nous avons utilisé est plus riche que 'pause' et 'fin' il contient les clés 'index\_chanson' qui dit le numéro de la chanson qu'on joue, 'index\_évenement' qui donne le numéro de l’événement dans la chanson qu'on est en train de jouer, 'vitesse' qui est par défaut à 1 permet de ralentir/accélérer une chanson on divise le temps d'attente en deux événements par la vitesse. En particulier si controleur['vitesse'] = float('inf') (qui veut dire infini) Tous les temps d'attente sont à 0 et la boucle de lecture se termine casi immédiatement. Ceci est pratique pour faire des tests sur la lecture avant de lancer la 'vraie' lecture.

Au fur et à mesure qu'on avancait dans le code, la boucle de lecture devenait de plus en plus complexe, nous voulions toujours rajouter de nouvelle possibilité (compter le temps écoulé depuis le début, détecter les canaux midi inutilisés par le lecteur pour ne pas les utiliser avec notre interface de piano. Corriger des problèmes liés à la pause. En particulier les notes on qui continuaient à jouer car les notes off étaient après la pause, l'affichage des paroles d'une chanson (pour un karaoké). On ne se retrouvais plus dans notre code.

Les modificateurs de controleur

VOIR CA M

Chaque fois que nous voulions ajouter une fonctionnalité au controleur, nous devions écrire une fonction qui était appelée à des endroits précis de la boucle de lecture (quand on rentre en pause, calcul le temps,traite un message de control, etc). Nous avons eu l'idée de mettre la liste de toutes les fonction qui doivent etre appelés a un moment particulier de la boucle dans une clé spécifique du controleur. Par exemple notre controleur contient une clé 'mod\_message\_meta' qui devra contenir la liste de tous les noms de fonction qu'il faudra exécuter quand on traitera un message meta.

## L'interface graphique

Notre interface graphique contient pour l'instant 3 objets principaux tKinter, ce sont tous des canevas qui peuvent contenir plein de choses. Le clavier est un canevas qui contient plusieures touches de piano. En appuyant sur ces touches on fait un son. Le piano est un canevas qui contient une barre de contrôle et un clavier. La barre de contrôle contient des widgets qui modifient le comportement du piano (l'instrument joué, jouer un certain type d'accord quand on appuie sur une touche…). Enfin un autre objet est l'interface de lecture de fichier midi. Pour l'instant il ne permet que d'ouvrir et de lire des fichiers de les mettre en pause et de les arreters.

### Généralité sur la facon de coder

Dans notre cours d'ISN nous avons beaucoup utilisé les variable globales pour nos interfaces graphiques. Nous nous sommes vite appercus que ces variables nous posaient des problèmes.

* Si par exemple on utilise une variable global pour stocker quel instrument on joue dans un piano c'était pratique car toutes les fonctions appelées par l'interface permettent de modifier la variable global. Par contre ceci pose des problèmes d'évolution du code .Si on décide à un moment de lancer deux claviers, car on veut jouer des percusions sur l'un et de l'orgue sur l'autre ca ne marchera pas car les deux pianos utilisent la meme variable global. On devait aussi connaître beaucoup d'information sur les touches et leur nombre peut varier. Il faudrait donc changer le code si on change le parametre. Nous n'avons pas retenu cette solution.
* A me sure que le code grossissait nous ne savions plus dans quels fonctions des variables globales étaient modifiées et ca rendait la recherche d'erreurs de plus en plus dificile.

Nous parlons ici des facons que nous avons utilisé pour éviter l'utilisation de variables global

# Manuel d'utilisation

# Conclusion

# 