2018 - 2019

Lycée Nevers
Associé à l'Etat par contrat

Projet Informatique et Sciences du Numérique

LISA BAGET ET MATTHIEU DURAND

Table des matières

[Remerciements]

Nous remercions monsieur Magne pour nous avoir fait découvrir Python et la programmation.

Nous tenons également à remercier toutes les personnes qui postent des tutoriaux sur le net, qui répondent à des questions sur les forums de discussion et qui laissent leur code à disposition des autres. Sans toutes ces personnes, notre travail aurait été impossible.

[Introduction]

Le projet *py@note* est un projet de programmation écrit en Python dans le cadre du projet « Informatique et Sciences du Numérique » de la terminale scientifique du lycée Nevers à Montpellier. Ce projet est disponible sous GitHub (https://github.com/Lisa-Baget/pyanote) sous licence GPL (pour « préserver la liberté d'utiliser, d'étudier, de modifier et de diffuser le logiciel et ses versions dérivées »).

Nous souhaitions que *py@note* soit un logiciel dédié à la musique permettant de lire de la musique, de la visualiser sous différentes formes (sur le clavier d'un piano, le manche d'une guitare, une partition, …), de la créer à partir de ces différentes visualisations, et de l'enregistrer sous la forme d'un fichier texte compréhensible. En fait, nous voudrions une espèce de Guitar Pro (https://www.guitar-pro.com/fr) mais pour lequel le code Python serait libre pour que chacun puisse modifier l'application suivant ses besoins.

[Cahier des charges]

Notre objectif dans un premier temps était de pouvoir représenter de la musique en Python, de pouvoir jouer cette musique, de représenter les notes jouées sur un instrument virtuel au fur et à mesure de la lecture, et de sauvegarder ce qui est joué par l'instrument virtuel.

{Phase initiale : réflexion et premiers essais}

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 7 avec un volume de 130 » ou « mettre l'instrument 13 (xylophone) sur le canal 0 ». Ce qui est joué en MIDI est moins joli qu'un vrai son enregistré mais utiliser de vrais sons nécessiterait d'avoir un fichier son par note et par instrument ce qui n'est pas possible pour nous. Pour envoyer ces instructions à la sortie MIDI de notre ordinateur, nous avons trouvé le module pygame.midi (https://github.com/pygame).

Nous avons vite utilisé MIDI pour jouer de la musique mais écrire des listes d'accords et de durées était un travail énorme de copie et de calcul alors qu'on peut trouver toutes les chansons qu'on veut dans des fichiers MIDI (fichiers binaires qui codent des séquences d'instructions dans une forme définie en 1996 par l'International Midi Association https://www.midi.org/). Nous avons donc voulu déchiffrer ces fichiers, ce qui semblait possible avec Mido (https://github.com/mido/mido). Mais nous n'avons même pas pu installer ce module et nous avons décidé d'écrire nous-même la lecture d'un fichier MIDI.

Au début nous voulions réaliser une guitare virtuelle pour représenter des accords sur le manche d'une guitare et jouer le son correspondant à ces accords sur l'ordinateur. Ceci nous a posé de gros problèmes principalement liés au fait qu'une même note (même octave) peut être jouée à différentes positions sur le manche. Comme ce n'est pas le cas avec le piano (une seule touche associée à chaque note), nous l'avons choisi comme instrument virtuel.

Enfin, pour l'interface graphique, nous avons naturellement choisi le module Python Tkinter (https://docs.python.org/3/library/tk.html) que nous avions étudié en cours.

{Objectifs : établissement d'un cahier des charges}

Au bout de deux mois de recherche de code qu'on pouvait utiliser, d'essais et de tests nous avons décidé le cahier des charges suivant pour notre projet :

- Décodage d'un fichier MIDI
    - récupération de toutes les infos binaires contenues dans le fichier

    - sauvegarde des infos dans un objet utilisable pour la lecture (que nous appelons album)
- Lecture d'un album
    - compréhension des données contenues dans l'album
    - envois de ces données au bon moment sur une sortie MIDI pour jouer la musique
- Interface de lecture
    - Interface graphique Tkinter permettant de contrôler la lecture
    - Ouverture fichier, lecture, pause, accélérer
- Réalisation d'un piano virtuel
    - Interface graphique Tkinter 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
- Création de nouveaux albums :
    - Création d'un album composé de celui du fichier MIDI et des notes jouées sur le piano virtuel.
    - Enregistrement de ce nouvel album au format MIDI

{Outils : faciliter le travail collaboratif}

Afin de faciliter le travail collaboratif, nous avons décidé (trop tardivement) d'utiliser les outils suivants :

__ *GitHub :* notre projet est hébergé sur GitHub (https://github.com/). Ce site contient la dernière version de notre projet, et aussi les versions correspondant à chaque mise à jour. On peut ainsi revenir à une version précédente en cas d'erreur irrécupérable… Il gère les sous-projets et assure un suivi des erreurs (issues).

__ *git :* est le programme qu'on doit installer sur l'ordinateur (https://git-scm.com/) pour faciliter l'interaction avec GitHub. Après l'avoir configuré (voir les instructions dans https://github.com/Lisa-Baget/pyanote/blob/master/documentation/github.md), il suffit de taper git pull pour récupérer la dernière version du projet depuis GitHub et git commit (préparation de l'envoi) puis git push pour envoyer à GitHub les modifications qu'on a faites.

__ *Visual Studio Code :* est un éditeur de texte gratuit (https://code.visualstudio.com/). On peut installer une extension *Python* qui facilitera beaucoup le travail de codage par rapport à IDLE, surtout quand on a beaucoup de fichiers. On peut également installer les extensions *LiveShare* pour que les collaborateurs voient en temps réel les modifications dans un fichier et *GitHub* pour voir les modifications par rapport à la version de GitHub.

[Manuel d'utilisation]

Nous donnons ici les instructions d'installation et des conseils d'utilisation, pour ceux qui veulent jouer avec comme pour ceux qui veulent étendre le programme.

{Installation : récupération et installation de py@note}

Notre projet py@note est disponible sur GitHub à l'adresse : https://github.com/Lisa-Baget/pyanote

__ *Si vous avez installé git sur votre ordinateur :* ouvrir l'invite de commande dans un répertoire MONREP et tapez les commandes suivantes :

```
MONREP> git.clone "https://github.com/Lisa-Baget/pyanote"
MONREP> cd pyanote
MONREP/pyanote> python setup.py
```

Ceci va installer à la fois py@note et pygame. Si on veut mettre à jour py@note avec sa dernière version :

```
MONREP/pyanote> git pull
MONREP/pyanote> python setup.py
```

__ *Si vous n'avez pas installé git sur votre ordinateur :* téléchargez une archive compressée de py@note à l'adresse : https://github.com/Lisa-Baget/pyanote/archive/master.zip . Décompressez l'archive, allez dans le répertoire pyanote (qui contient setup.py) et dans l'invite de commande :

```
MONREP/pyanote> python setup.py
```

__ *Pour les progammeurss :* si vous souhaitez modifier notre code, il est préférable d'utiliser la commande ci-dessous. Ainsi, python utilisera votre version de py@note et pas celle installée. Attention, si vous rajoutez un fichier dans le module pyanote, il faudra changer la version dans le setup.py et relancer ce setup.

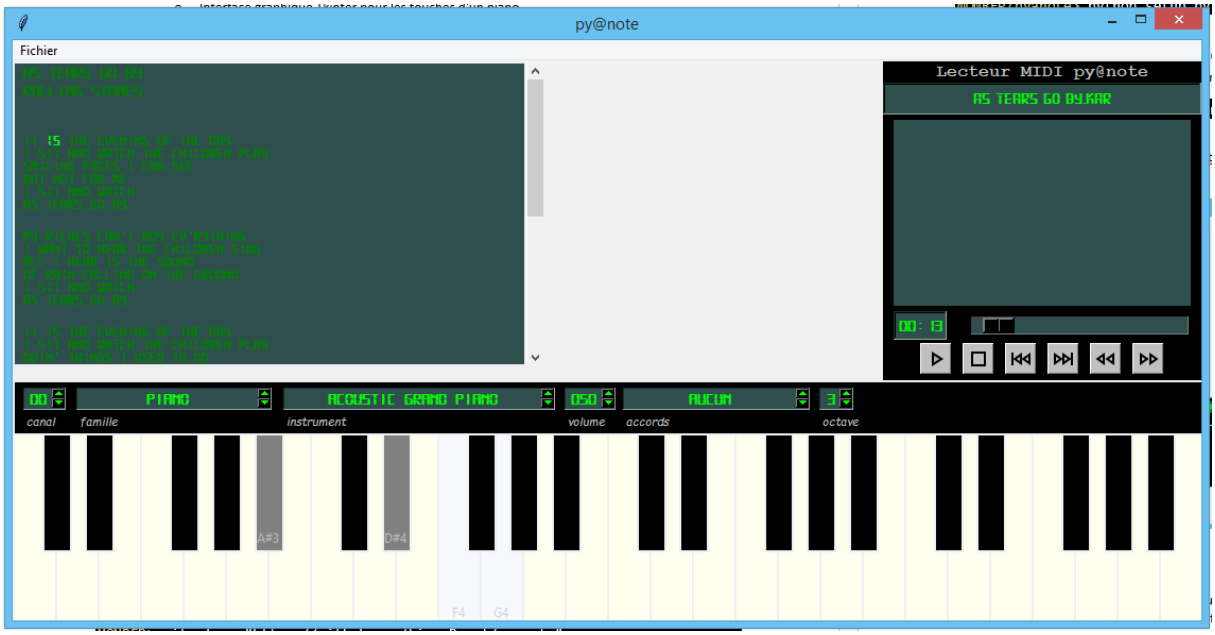
```
MONREP/pyanote> python setup.py develop
```

{Utilisation : que peut-on faire avec py@note}

Depuis le répertoire d'installation, ouvrez l'invite de commande et tapez :

```
MONREP/pyanote> python pyanote/main.py
```

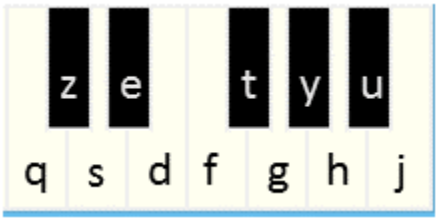
Ceci lancera la fenêtre principale de py@note, vous permettra d'écouter des fichiers MIDI, de jouer du piano et de vous entrainer au karaoke.



__ *jouer du piano :* c'est aussi simple que de cliquer sur une touche. Le nom de la note est affiché, la touche se colore, et la note est jouée. Ceci disparait quand vous arrêtez de cliquer mais la note résonne encore ¼ de secondes. Le comportement peut être modifié par les widgets de contrôle situés au-dessus du clavier.

- canal : sélectionne le canal MIDI sur lequel les notes sont jouées
- famille et instrument : sont utilisés pour modifier le son joué quand on appuie sur une note
- volume : change le volume des notes jouées
- accords : par défaut aucun, seule la note sur laquelle on clique est jouée. Si on sélectionne un accord, la note sur laquelle on clique est considérée comme la fondamentale de cet accord et toutes les notes sont jouées (son, coloration de la touche et affichage du nom de la note)
- octave : sélectionne l'octave qui sera joué quand on utilise les raccourcis clavier

Les raccourcis clavier que nous avons utilisé tentent de faire correspondre la position des touches sur le piano et celles sur le clavier français. Ils sont :



__ *lire un fichier midi :* dans le canevas Lecture MIDI py@note, cliquez sur la zone de texte prévue pour ouvrir le dialogue de sélection d'un fichier MIDI et lancer la boucle de lecture de l'album contenu dans ce fichier.

Les notes sont affichées sur le clavier sur le clavier au fur et à mesure qu'elles sont jouées. Ce n'est pour l'instant pas satisfaisant car on affiche toutes les notes jouées par tous les instruments (sauf percussions). Si de plus le fichier ouvert est un fichier KARAOKE (un fichier MIDI qui finit par .kar, et dont certains messages codent des paroles de chansons), les paroles sont affichées dans un canevas et changent de couleur au fur et à mesure de leur lecture.

Pour l'instant les seuls widgets de contrôle du lecteur qui fonctionnent sont les deux à gauche : lecture/pause (bouton qui change d'apparence quand on l'actionne) et arrêt.

__*bugs :* nous avons repéré de nombreux bugs dans cette interface. Nous en tiendrons la liste dans les « issues » de GitHub (https://github.com/Lisa-Baget/pyanote/issues). N'hésitez pas à nous faire part à cette adresse de tout nouveau problème détecté.

{Développement : coder sous py@note}

Vous pouvez également tester tous les modules de pyanote indépendamment. Par exemple pour voir comment py@note représente une piste MIDI :

```
MONREP/pyanote> python pyanote/piste.py
```

Notre projet comporte pour l'instant 15 fichiers (modules) contenus dans le répertoire (package) pyanote. Ces 15 fichiers sont organisés comme expliqué dans le tableau suivant (nous avons compté le nombre de lignes de code, sloc = source lines of code en anglais sans compter les lignes de commentaires ni les tests). Avec le projet il y a aussi un répertoire archives qui contient des idées qui n'ont pas marché mais que nous voulions garder, un répertoire fichiersMidi qui contient 4 exemples de fichiers MIDI récupérés sur internet, ainsi qu'un répertoire documentation qui n'est pas vraiment à jour…

Nom du module	Contenu	sloc
Lecture (décodage) d'un fichier binaire MIDI		*152*
utils.py	Petites fonctions pour lire des informations élémentaires dans un fichier binaire et les transformer en un objet python comme un entier.	30
resume.py	Création d'un résumé. C'est un dictionnaire qui contient toutes les informations d'un fichier MIDI sauf la liste des évènements.	33
pistes.py	Création d'une piste. C'est une liste d'évènements correspondant à une piste du fichier MIDI.	62
album.py	Création d'un album. C'est un dictionnaire construit à partir du résumé en rajoutant une liste de chansons. Suivant le format du fichier MIDI, les chansons ne correspondent pas exactement aux pistes.	28
Lecture (son) d'un album créé à partir d'un fichier binaire MIDI		*179*
son.py	Petites fonctions utilisées pour interagir avec pygame.midi.	8

controleur.py	Création d'un controleur. C'est un dictionnaire construit à partir d'un album en rajoutant tous les parametres qui seront utilisés et éventuellement modifiés pendant la boucle de lecture. On peut démarrer un controleur pour commencer la boucle de lecture soit normalement, soit dans un thread.	78
modificateurs.py	Contient toutes les fonctions appelées pendant la boucle de lecture d'un controleur. Si on veut rajouter des fonctionnalités à la boucle de lecture, par exemple afficher les paroles d'un fichier Karaoke, il faudra modifier ces fonctions.	93
Musique (relation avec les notations musicales)		*49*
notes.py	Transforme une chaîne de caractères représentant une note (''Do#4')' en un entier utilisé par MIDI pour représenter cette note (61) et inversement.	38
accords.py	Calcule une liste de notes (un accord) à partir d'une note fondamentale et du nom d'un accord.	11
Interface graphique basée sur Tkinter		*365*
lecteur.py	Création d'un canevas Tkinter permettant de controler la boucle de lecture. Ouverture de fichier, pause, arrêt.	78
clavier.py	Création d'un canevas Tkinter correspondant à toutes les touches d'un piano, qui peuvent être utilisées pour jouer de la musique.	95
instruments.py	Création d'un dictionnaire permettant d'accéder aux numéros d'instruments MIDI à partir de leur famille et de leur nom. Utilise un fichier JSON créé par mobyvb	8
piano.py	Création d'un canevas contenant un piano et des widgets de contrôle.	90
karaoke.py	Interface graphique pour afficher les paroles d'un fichier karaoke	57
main.py	Création d'un canevas contenant un piano et un lecteur.	37
Total :		**742**

Si vous souhaitez développer à partir de py@note, faites un fork sous GitHub. Si votre travail vous satisfait, faites-le nous savoir et nous joindrons votre travail à py@note.

[Rapport technique]

Le package pyanote contient de nombreux algorithmes qu'une interface graphique fait fonctionner ensemble. Nous commençons par des généralités concernant l'ensemble de notre code puis feront des zooms sur des parties du code qui nous ont posé des problèmes ou qui nous semblent intéressantes.

{Généralités : à propos de notre code}

Dans cette partie nous parlons de tout ce qui a influé sur l'ensemble du projet py@note.

Organisation du code

Au fur et à mesure qu'on avançait dans le projet, le fichier pyanote.py dans lequel on codait devenait illisible et on n'arrivait plus à corriger les erreurs. Nous avons donc décidé d'organiser le code de la façon suivante :

__ *Utilisation de modules* **:** tout notre code est contenu dans un répertoire pyanote qui contient des fichiers Python appelés modules. Chaque module contient un ensemble de fonctions servant à résoudre un problème particulier. Par exemple le module pistes.py contient les fonctions nécessaires 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 :

```
import pyanote.resume
pyanote.resume.creer_resume("nom_fichier.mid")
```

Nous avons eu des problèmes avec la forme from pyanote.resume import * quand des fonctions de différents modules avaient le même nom et avons décidé d'éviter cette forme d'import. 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) . Un autre problème que nous avons rencontré est celui des imports circulaires (un fichier A ne peut importer un fichier B qui importe A). Ceci a été particulièrement casse-tête pour relier l'interface graphique à la boucle de lecture.

__ *Test du code :* chaque module a souvent dû être testé. Le problème est que si on laisse le test dans le module, quand on importe le module dans le programme final, le test est exécuté (et on ne veut pas voir les résultats de tous les tests dans ce programme). Cependant enlever ou commenter les tests demande à réécrire ou décommenter tous les tests dès qu'on modifie le code du module. La solution choisie est de mettre tous les tests d'un module sous la forme suivante :

```
if __name__ == "__main__":
    ## Faire les tests ici
```

Ainsi le test ne s'exécute que si on exécute directement le module et pas dans un fichier qui l'importe. Tous nos modules contiennent de tels tests que nous avons pu relancer sans problème dès que nous modifions un module ou un module importé par ce module.

__ *Commentaires :* en travaillant ensemble on s'est rendu compte de l'importance 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 ça les conventions PEP227 (voir https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python/235263-de-bonnes-pratiques). Ceci permet dans certains éditeurs (comme l'éditeur gratuit Visual Studio Code) de voir les commentaires de la fonction quand on tape son nom dans un autre module.

```
"""pyanote.piste
(C) Lisa Baget, 2018-2019

Ce module contient les fonctions permettant de construire la liste de tous les
évènements contenus dans une unique piste d'un fichier Midi.
"""
import pyanote.utils as utils

def creer_piste(resume, num_piste):
    ''' Retourne la liste de tous les evenements contenus dans une unique
piste du fichier Midi décrit.
    '''
```

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 exemple pour nous un événement MIDI est une liste E = [temps, numéro, message]. On accède au temps en utilisant E[0]. C'est pratique et rapide tant qu'on n'a pas des listes trop grandes. Nous avons par exemple un objet contrôleur qui contient plus de 30 éléments différents. Il est impossible de programmer en se souvenant des indexs de chacun de ces éléments. C'est dans ce cas que nous avons choisi d'utiliser des dictionnaires. Par exemple, si on a un contrôleur C = {''index_chanson'': 0,''index_evenement'': 1299, ''vitesse'': 1.2, …}, en tapant C[''vitesse''] on récupère 1.2 et en tapant C[''vitesse''] = 1.5 on change la vitesse dans le contrôleur 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 erreurs* **:** nous avons déjà vu en cours l'existence d'erreurs, par exemple quand on tape 1/0 Python affiche une erreur dont le type est ZeroDivisionError. Nous pouvons lancer nos propres erreurs. Par exemple 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 on ne lit pas dans le fichier MIDI une chaîne binaire qui devrait s'y trouver. Cependant les erreurs peuvent aussi nous poser des problèmes, par exemple notre interface graphique va ouvrir des fichiers MIDI mais si la lecture provoque une erreur (ce n'est pas un fichier MIDI ou il est défectueux, ce qui nous est arrivé), on ne veut pas que l'interface s'arrête.

```
def changer_fichier():
    fichier = askopenfilename()
    try:
        album = creer_album(fichier)
    except ValueError:
        return
    jouer_album(album)
```

Ce code est présent dans notre interface graphique. Lorsque on veut changer de fichier, on essaie (try) de créer un album. Si ça marche on ne regarde pas ce qu'il y a dans la partie except et on continue le programme qui joue l'album. Si ça ne marche pas, l'erreur est interceptée par le except et n'arrête pas le programme. On exécute le code qui est dans la partie except (ici un return pour sortir de la fonction qui ne fera donc rien).

__ *Les fonctions anonymes* **:** nous en discuterons plus longuement dans la partie interface graphique, où elles ont été nécessaires. Quand on écrit :

```
def test(x, y):
    return x + y
```

c'est presque exactement la même chose que d'écrire :

```
test = lambda x, y: x + y
```

L'expression lambda x, y : x + y veut dire « une fonction de deux paramètres x et y qui retourne (le return est implicite) x + y. Avec test = lambda … on donne le nom test à cette fonction. Il y a des différences : par exemple, le code du lambda ne peut contenir qu'une instruction.

{pygame.midi : Lecture du son}

Nous avons dû commencer par comprendre ce qu'est un message MIDI pour utiliser le module pygame.midi (https://github.com/pygame). Ce module nous permet d'ouvrir une sortie son de la manière suivante :

```
import pygame.midi
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 des messages, 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 hexadécimale) 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 nécessite qu'un seul argument donc le dernier

est à zéro. 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). Après avoir dormi deux secondes le programme va éteindre cette même note (8 = note off). 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 9 réservé à la batterie (on n'a pas besoin de les éteindre). La sortie midi n'a que 16 canaux différents (0 à 15).

Nous avons décrit ici des messages de contrôle que nous expliquerons mieux dans la partie fichier MIDI. Mais on peut aussi envoyer des messages système. Dans py@note, nous avons décidé d'envoyer à la sortie MIDI tous les messages système sans essayer de les comprendre car ils sont le plus souvent spécifiques aux constructeurs de matériel. Pour les messages contrôle on aurait pu faire la même chose mais nous avons besoin d'en comprendre au moins certains : par exemple pour changer la couleur d'une touche du piano quand la note correspondante est jouée.

__ *pyanote/son.py :* dans ce module nous avons juste écrit de petites fonctions qui permettent d'utiliser de façon plus simple les fonctions de pygame.midi.

{Musique : 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 aigus.

__ *pyanote/note.py :* contient des fonctions qui permettent de transformer un numéro de note MIDI en une chaine de caractères (par exemple 60 donnera Do4) et inversement de transformer une chaine de caractères représentant une note en entier représentant la note en MIDI.

```
print(nombre_vers_note(82))
print(nombre_vers_note(82, "b"))
```

Ce programme affichera A#5 puis Bb5. 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 le paramètres par défaut # (préféré). Dans le deuxième appel on a indiqué qu'on préférait bémol. Pour écrire ces traductions, il nous faut une correspondance entre chaque nom de note (ici en anglais) et le nombre de ½ tons représenté par cette note.

```
EN = {0: "C", 2: "D", 4: "E", 5: "F", 7: "G", 9: "A", 11: "B"}
```

On applique alors la formule (découverte en étudiant les tableaux de correspondance des notes MIDI):

NUMERO_NOTE_MIDI = (OCTAVE + 1) * 12 + VALEUR(NOTE)

Cette formule est en fait celle d'une division euclidienne : la division euclidienne d'un numéro de note midi par 12 donne (octave + 1) et a pour reste la valeur de la note. Donc :

OCTAVE + 1 = NUMERO_NOTE_MIDI // 12 ## division euclidienne VALEUR(NOTE) = NUMERO_NOTE_MIDI % 12 ## reste de la division euclidienne

Les fonctions que nous avons écrites sont plus compliquées puisqu'il fallait gérer les dièses (augmenter la note d'un ½ ton) et les bémols (diminuer la note d'un ½ ton). Dans tout notre projet nous ne manipulons les notes que sous la forme d'entiers. Le seul moment où nous utilisons ce module est pour afficher le nom des notes sur les touches d'un piano quand la note est jouée.

__ *pyanote/accords.py :* nous avons écrit une bibliothèque d'accords dans un dictionnaire dont le début est :

```
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. Si on appelle construire_accord(82, ''majeur'') ça nous donnera la liste [82+0 ,82+4,82+7] qui est l'accord de A#majeur.

{Lecture d'un fichier MIDI : se plonger dans des spécifications}

Pouvoir déchiffrer un fichier MIDI a été un énorme travail rendu nécessaire 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.

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

Le format MIDI

Un fichier MIDI est un fichier binaire composé de plusieurs parties. La première est le header (en-tête) qui est suivie d'une ou plusieurs pistes (tracks).

__ *header :* le header commence par 4 octets qui codent la chaîne binaire b'MThd'. Suivent 4 octets qui codent un entier (la taille du header), 2 octets qui codent un entier (le format du fichier), 2 octets qui codent un entier (le nombre de pistes) et 2 octets qui codent un entier (le tempo en ticks/noire, nous en reparlerons). Attention, si le premier bit du premier octet du tempo est à 1, ça veut dire qu'il s'agit d'une façon différente de coder le temps. Nous n'avons pas codé ceci car nous n'avons jamais trouvé d'exemple de fichier MIDI qui faisait ça. Dans ce cas, nous envoyons une erreur. Les taille – 6 octets suivants sont réservés aux constructeurs de matériel MIDI et nous n'avons pas à nous en soucier.

__ *track :* une piste commence toujours par 4 octets qui codent la chaîne binaire b'MTrk'. Suivent 4 octets qui codent un entier (la taille de la piste). Viennent ensuite, jusqu'à la fin de la piste, une séquence d'évènements.

Dans pyanote/resume.py, la fonction créer_résumé(nom_de_fichier) permet de créer un dictionnaire contenant toutes les informations dont nous venons de parler, mais ne lira pas les évènements. Par exemple elle retournera:

```
{'fichier': 'fichiersMidi/Dave Brubeck - Take Five.mid',
 'format': 1,
 'nb_pistes': 17,
 'ticks/noire': 192}
 'resumes_pistes': [
        {'id': 0, 'début': 22, 'fin': 82},
        {'id': 1, 'début': 90, 'fin': 4713},
        # 15 autres pistes nom montrées]
}
```

Dans cet exemple (simplifié, car le vrai fichier fait 17 pistes) la piste 0 commence à l'octet 22 et finit à l'octet 82. Ces informations seront nécessaires pour lire une piste.

__ *évènements :* un évènement MIDI est un delta-temps suivi d'un message. Le delta-temps est la durée en ticks (une unité de mesure de temps spécifique à MIDI) entre l'évènement et l'évènement précédent. Ce delta-

temps est stocké sur un nombre variable d'octets (voir lecture d'un entier variable ci-dessous). Pour nous, dans notre représentation Python, un évènement sera une liste :

```
evenement = [delta-temps : int, num_piste : int, message : list]
```

__ *messages :* un message MIDI est codé sous un nombre variable d'octets. Ils contient l'information qu'il faut traiter après l'écoulement du delta-temps. Pour déchiffrer un message, il faut commencer par lire son premier octet, qui est normalement (voir running status ci-dessous pour l'exception qui nous a embêté) le statut du message. Il y a 3 types de messages, suivant la valeur du statut. Tous seront codés par des listes.

__ *messages système :* quand le statut vaut b'\xF0' ou b'\xF7', ce qui suit est un entier variable qui code la longueur de la chaîne binaire (bytes en Python) qui est la chaîne du message système. Le message système complet est égal à statut + chaîne du message système. Comme on l'a déjà dit, nous envoyons les messages système à la sortie MIDI sans essayer de les comprendre. Pour nous, un message système sera une liste de taille 1 (1 seul argument, c'est-à-dire len(message_systeme) = 1)

```
message_systeme = [chaine_binaire : bytes] # liste de taille 1
```

__*messages meta :* quand le statut vaut b'xFF', l'octet qui suit code le type du message meta, l'entier variable qui suit la *taille* nécessaire pour coder sa valeur, et les *taille* octets qui suivent codent effectivement cette valeur. Suivant le type du message meta, la valeur doit être comprise de différentes façons, nous donnons juste ici quelques exemples que nous utilisons vraiment dans py@note.

Type	**Signification**	**Taille**	**Valeur**
0x01	*Text Event.* Utilisé dans les fichiers karaoke pour les paroles.	var	Chaine de caractère codée dans un format non spécifié
0x2F	*End of Track.* Dernier message (obligatoire) de chaque piste.	0	Rien
0x51	*Set Tempo.* Changement de tempo.	3	Un entier codé sur 3 octets qui donne les microsecondes/noire

Le message meta n'est pas à envoyer à la sortie MIDI. Certains doivent être interprétés pour que la lecture marche (changement de tempo), d'autres sont purement informatifs (texte, nom de la chanson). Pour nous en Python le message meta sera une liste de taille 2 dont le deuxième élément peut-être un entier, une chaîne, une liste d'entiers, suivant la valeur du premier argument.

```
message_meta = [type_meta : int, valeur : object] # liste de taille 2
```

__ *messages de contrôle :* dans tous les autres cas, le message est un message de contrôle. L'instruction est codée sur les 4 premiers bits du status (division euclidienne par 16), et le canal sur les 4 derniers bits (reste de la division euclidienne par 16). Viennent ensuite 1 ou 2 arguments, suivant l'instruction. Les messages de contrôle doivent être envoyés à la sortie MIDI, car ils contiennent les informations sur la musique qui doit être jouée. Certains types de messages ont utilisés dans l'interface graphique de py@note.

Inst.	**Signification**	**arg 1**	**arg 2**	**Utilisation py@note**
0x8	Note off	Note	Vélocité	OUI : on change la couleur du clavier
0x9	Note on	Note	Vélocité	OUI : on change la couleur du clavier
0xA	Note Aftertouch	Note	Quantité	NON
0xB	Controller Event	Type	Valeur	NON
0xC	Program Change	Instrument	x	OUI : envoyé quand on change d'instrument
0xD	Channel Aftertouch	Quantité	x	NON
0xE	Pitch bend	Valeur		NON

Pour nous, un message de contrôle sera une liste de taille 3 (quand il n'y a pas de deuxième argument, on le met à 0 et c'est quand même compris par pyanote.midi).

```
message_controle = [statut : int, arg1 : int, arg2 : int] # liste de taille 3
```

L'appel de la fonction pyanote.piste.creer_piste(resume, num_piste) va retourner la liste de tous les évènements contenus dans cette piste. Si resume est celui créé avant et le numéro de piste est 0, cette fonction vas nous donner (on a enlevé plusieurs évènements) :

```
[[0,0,[81, 359281]], [7,0,[b'\xf0~\x7f\t\x01\xf7']], ..., [178059,0,[47, []]]]
```

Le premier évènement arrive immédiatement (au bout de 0 ticks). Comme tous les autres événements de la piste 0, sont deuxième élément vaut 0. Son message est une liste de taille 2, donc un meta. Son type est 81 = 0x51, donc un changement de tempo. Le nouveau tempo doit être 359281 microsecondes/noire. Le deuxième événement arrive au bout de 7 ticks, et contient un message système (liste taille 1). On enverra à la sortie MIDI la chaîne binaire b'\xf0~\x7f\t\x01\xf7'. Le dernier événement arrive 178059 ticks après le précédent, c'est le message meta fin de piste codé par b'\xFF\x2F\x00' (on vérifie sa présence quand on construit le résumé).

pyanote/utils.py : lecture d'un entier variable

Le format MIDI définit une façon 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 intervalles 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 place 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 o_1= 129. Le premier bit de cet octet est à 1 car 129 $\geqslant$ 128 (en effet si le premier bit est à 1 le nombre est supérieur ou égal à $1000\ 0000_{(2)} = 2^7{}_{(10)} = 128_{(10)}$). Aussi, il faut lire l'octet suivant o_2 = 145. Comme 145 $\geqslant$ 128 nous lisons également l'octet o_3 = 25 et comme 25 < 128 on s'arrête et on ne lit plus rien. Le VARINT que nous cherchons est codé sur les entiers o_1, o_2 et o_3. Cependant le premier bit de o_1 et o_2 ne sert pas à coder l'entier mais seulement à dire de continuer la lecture. Il faut donc faire comme si il n'était pas là. Pour enlever ce bit il suffit d'enlever 128 au nombre. On a $o'_1 = o_1 - 128$, 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 réellement codés sur 7 bits. Chacun ne peut coder que $2^7 = 128$ nombres possibles et le nombre VARINT est donc $o'_1 \times 128^2 + o'_2 \times 128 + o'_3$. C'est ce qui est calculé par notre fonction lire_entier_variable.

```
def lire_entier_variable(fichier):
    entier = 0
    octet = ord(fichier.read(1))
    while octet >= 128: ## si le bit de poids fort est 1 il faut continuer
        entier = entier * 128 + octet - 128
        octet = ord(fichier.read(1))
    return entier * 128 + octet
```

Dans ce code, 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 et donc nous n'avons pas pu l'utiliser pour lire un entier codé sur n octets en faisant ord(fichier.read(n)).

pyanote/pistes.py : le problème du running status

La grosse difficulté que nous avons rencontré dans cette partie est liée 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 (et nous ne l'avons pas expliqué dans notre explication du format MIDI). 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'existence de ce running status.

A chaque fois qu'on lit un message de contrôle dans le fichier, il faut enregistrer le statut de ce message (on n'enregistre que le dernier). Au prochain message de contrôle, si le statut qu'on lit ne correspond pas à celui d'un message de contrôle (car en divisant le statut par 16 on n'obtient pas une instruction), alors il faut comprendre que le vrai statut de ce message est celui qu'on avait sauvegardé, et que ce qu'on vient de lire est en fait le premier argument du message. Le running status économise donc des octets.

Nous présentons ci-dessous une version simplifiée des fonctions de piste.py qui montre comment nous avons géré ce running status grâce à une variable sauvegarde.

```
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 #mise a jour de la sauvegarde
        arg1 = ord(fichier.read(1))
    else: #besoin du running statut
        arg1 = statut
        statut = sauvegarde[0]
    #ici il faudrait tester qu'il faut un 2eme argument
    arg2 = ord(fichier.read(1))
    return [statut, arg1, arg2]
```

Cette fonction est appelée dans notre boucle de lecture de tous les événements (elle s'appelle creer_piste) où nous initialisons sauvegarde = [None]. Au début il n'y a pas de sauvegarde, nous mettons cette information dans une liste pour pouvoir la changer 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 annuler 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 (ceux qui expliquent le format ne sont pas toujours d'accord). Nous avons décidé d'annuler la sauvegarde dans ce cas et pour l'instant aucune erreur n'a été trouvée.

pyanote/album.py : la fusion de pistes

Pour comprendre ce besoin de fusionner les pistes, revenons au format que nous avons lu dans le résumé. Il y a 3 valeurs de format possible :

- 0 (monopiste) : le fichier ne contient qu'une chanson dont les 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. Cette fusion veut dire qu'on doit coder une liste d'évènements qui, si on la jouait, donnerait la même chose que de jouer toutes les pistes en même temps.
- 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.

Imaginons que la fonction creer_piste nous ait donné deux pistes (dans ces pistes, nous avons remplacé les messages par des lettres pour ne pas encombrer l'exemple avec des messages qui n'influencent pas la fusion).

```
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] ]
```

Les premiers éléments de chaque événement sont des temps relatifs, et se lisent de la manière suivante : [20,0,E] veut dire que E devra être traité 20 ticks (unité de mesure du temps chez les événements MIDI) après le message de l'événement B précédent. Le problème est que ces delta temps n'ont de sens que dans une piste et ne peuvent être utilisés pour comparer des événements dans des pistes différentes. Une première étape de la fusion va donc être de transformer ces temps relatifs en temps absolus qui pourront être 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 essai en rajoutant la deuxième piste à la première puis en appelant une fonction de tri. Mais la fusion était très 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, dans cette liste, les éléments ne sont pas des événements car ils expriment des temps absolus et pas des temps relatifs (qui sont prévus dans le format MIDI). Nous avons donc du écrire une fonction similaire à la précédente qui remplace les temps absolus par des temps relatifs. Avec cette fonction nous obtenons la liste fusionnée définitive :

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

{Jouer les notes d'un fichier MIDI : nombreux essais}

Grâce à nos modules pyanote.son et pyanote.album nous pensions n'avoir aucun problème avec la lecture d'un fichier midi. Grave erreur, nous avons du essayer de multiples versions de boucles de lecture.

Lecture basique et gestion du temps

Pour écrire notre première boucle de lecture, la seule difficulté devait être la gestion du temps car il faut calculer un temps en secondes à partir des ticks. Notre première 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)
```

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 meta nous indiquent des changements de tempo M en microseconde/beat. Quand on connaît ces deux valeurs, M/H nous donne une mesure T en micro/ticks. Quand on multiplie un delta temps en ticks par T on obtient une durée en microseconde, qu'il suffit de diviser par 10^6 pour obtenir le paramètre de time.sleep.

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

Il reste un problème : si on n'a pas reçu 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. Nous avons calculé une valeur par défaut M = 500000 grâce au calcul suivant.

120 noires/minute = 1/120 minutes/noire= 60/120 secondes/noire= $0{,}5 \times 10^6$ microsecondes/noire

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 remplaçant la fonction time.sleep par la fonction after de Tkinter.

```
def continuer_lecture(chanson, i , album, sortie_midi):
    evenement = chanson[i]
    traiter_evenement(evenement, sortie_midi)
    if i + 1 < len(chanson): # il y a un evenement suivant
        ev_suivant = chanson[i+1]
        temps = calculer_temps(album, evenement[0])
        Tk().after(temps, continuer_lecture, chanson, i+1, album, sortie_midi)
```

Avec cette solution notre fonction de lecture est découpée en pleins de petites fonctions continuer_ lecture qui s'appellent lpar l'intermédiaire de after. La dernière ligne du programme ci-dessus veut dire que, après temps, on appelle continuer_lecture(chanson, i+1, album). Ceci semble faire la même chose que la version basique mais il y a une différence. La fonction time.sleep utilisée dans la première version est bloquante et provoque le freeze. La fonction after de Tkinter est non bloquante et laisse à Tkinter le temps de gérer ses événements. Nous pensions avoir ici la bonne solution mais la fonction after est beaucoup moins précise que la fonction sleep Appeler after(130, …) peut en fait créer un décalage de 180 millisecondes voire plus). En écoutant le morceau on ne reconnaissait plus le rythme. Malheureusement, nous n'avons pas gardé cette version du programme que nous aurions dû garder en archives.

La lecture dans un thread

On nous a alors 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 appelant cette fonction dans un thread elle se déroule en parallèle du reste du programme (le programme principal). En particulier, la chaine 'le thread est vivant' s'affiche de suite et n'attend pas la fin de l'exécution de la fonction jouer_album. Avec cette méthode, Tkinter ne freeze plus et comme la fonction de lecture utilise time.sleep, les intervalles de temps sont correctement respectés.

Introduction du contrôleur

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 principal (pour mettre la lecture en pause, arrêter la lecture, …). Pour ce faire, nous profitons que le programme principal et le thread ont la même mémoire. Si le programme principal et le thread partagent un dictionnaire que nous allons appeler le contrôleur, ce contrôleur 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'on peut lancer dans un thread, il est possible de contrôler la lecture depuis le programme principal. Si 'fin' et 'pause' sont à False 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 principal 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 changée. Le contrôleur que nous avons utilisé est plus riche que 'pause' et 'fin', il contient les clés 'index_chanson' et 'index_évenement' qui donnent le numéro de la chanson qu'on joue et celui de l'événement dans cette chanson, 'vitesse' (par défaut à 1) permet de ralentir/accélérer une chanson en divisant le temps d'attente 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 presque immédiatement. Ceci est pratique pour faire des tests sur la lecture avant de lancer la 'vraie' lecture.

Les modificateurs de contrôleur

Au fur et à mesure qu'on avançait 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, l'affichage des paroles d'une chanson (pour un karaoké), la correction des problèmes liés à la pause... En particulier les note on qui continuaient à jouer car les note off étaient après la pause. On ne se retrouvait plus dans notre code. Nous avons choisi d'appeler, à chaque étape spécifique du contrôleur, une fonction qui gére tout ce qui doit se passer à ce moment précis de la boucle de lecture. Par exemple, dans la partie traitement des messages:

```
if len(message) == 1: # systeme
```

```
        mod.executer_modificateurs_message_systeme(controleur, n_piste, msg)
    elif len(message) == 3: # controle
        mod.executer_modificateurs_message_controle(controleur, n_piste, msg)
    else: # meta
        mod.executer_modificateurs_message_meta(controleur, n_piste, msg)
```

Les trois fonctions appelées sont codées dans le module pyanote.modificateurs (nommé mod ici). Elles contiennent tous les instructions à réaliser à cette étape de la boucle de lecture. Cette organisation nous a permis de rajouter des possibilités à la boucle de lecture sans plus jamais toucher au code de cette boucle. Nous avons pour l'instant écrit dans pyanote/modificateurs.py les fonctionnalités suivantes :

- envoie les messages de contrôle et système à la sortie MIDI
- stocke les note on qui ne sont pas fermés pour faire des note off quand on se met en pause
- stocke la durée totale des sleep d'une chanson, à la fin d'une analyse on connait donc sa durée.
- signaler à l'interface graphique qu'il faut mettre à jour l'horloge dès qu'une seconde s'est écoulée
- enregistre les canaux utilisés par des note on. A la fin de l'analyse, on peut connaitre les canaux disponibles pour le piano.
- signaler les note on, note off à l'interface graphique pour modifier l'apparence des touches
- envoie les messages meta de type 1 utilisés dans les fichier .kar (MIDI + karaoke) vers l'interface graphique pour qu'ils soient éclairés au bon moment.

{Interface graphique : peu d'algorithmes, presque 50% du programme}

Notre interface graphique contient pour l'instant 4 objets principaux Tkinter, ce sont tous des canevas qui peuvent contenir plein de widgets. Le clavier est un canevas qui contient plusieurs 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…). Nous avons aussi une interface de lecture de fichier midi. Pour l'instant elle ne permet que d'ouvrir des fichiers, de les mettre en pause et de les arrêter. Enfin, un Frame du module pyanote/main.py réunit tous ces canevas et les fait interagir.

Généralités sur notre façon de coder en Tkinter

Dans notre cours d'ISN nous avons beaucoup utilisé les variables globales pour nos interfaces graphiques. Dans le projet, ceci nous a posé des problèmes.

Si par exemple on utilise une variable globale pour stocker quel instrument on joue dans un piano c'était pratique car toutes les fonctions appelées par l'interface permettent de lire et de modifier la variable globale. Ceci pose problème quand on utilise un nombre variable de widgets (par exemple les touches du clavier, ou même les pianos). Dans ce cas, on ne sait pas combien de variables déclarer. De plus, à mesure que le code grossissait nous ne savions plus dans quelles fonctions des variables globales étaient modifiées et ça rendait la recherche d'erreurs de plus en plus difficile. Nous nous sommes donc obligés à coder de la façon suivante.

__ *Passer paramètres, utiliser mutables :* quand une fonction f déclare une variable a, puis appelle une fonction g qui doit lire a, il faut que a soit passé par les paramètres de g. Si f appelle f1 qui appelle f2 … qui appelle g, il faudra passer a en paramètre dans toutes les fonctions f1, f2, … Si f déclare la variable a, mais n'appelle pas (directement ou indirectement) la fonction g qui doit lire a, nous avons un problème de conception. Il faut alors repérer une fonction h qui appelle à la fois f et g, déclarer la variable a dans h et la passer en paramètre dans f et g. Le passage par paramètre ne permet pas automatiquement de modifier la variable, il faut qu'elle soit mutable (exemple : liste, dictionnaire). Le code suivant illustre ce principe (c'est celui utilisé dans la sauvegarde du running status):

```
def h():
    a = [None] ## déclaration de la variable a,  mutable
    f(a) ## on envoie a à f
```

```
    g(a) ## on envoie a à g

def f(x): ## c'est f qui calcule la valeur de a
    x[0] = "Modification" ## x = ["Modification"] ne marche pas

def g(x): ## on affiche a dans g
    print(x[0]) ## Affichera bien Modification
```

Ainsi une variable déclarée dans une fonction est initialisée dans une autre fonction et lue dans une troisième.

__ *Enregistrer les données dans les objets, naviguer depuis un évènement :* le passage de paramètres dans une fonction n'est pas toujours possible dans Tkinter. En effet, appuyer sur un bouton ou cliquer dans un canevas va pouvoir, grâce au bind, déclencher l'application d'une fonction f. Cependant cette fonction f ne peut être définie qu'avec un seul argument, l'événement qui est créé au moment du clic. Par exemple si on clique sur une touche de notre piano, on veut utiliser la sortie MIDI pour jouer la note correspondant à cette touche dans le canal selectionné, avec le volume selectionné. Le programme suivant ne marche pas.

```
touche = tk.Canvas(piano, ...)
touche.bind("<Button-1>", appuyer_touche_souris)

def appuyer_touche_souris(evenement):
    # dans cette fonction on ne connait que evenement
    message = [0x90 + canal, note, vol] # on connait pas canal, note, vol
    pyanote.son.message_controle(sortie_midi, message) # ni sortie_midi
```

Bien sûr, on peut utiliser des variables globales pour le canal, le volume et la sortie_midi, mais la solution n'est pas raisonnable pour les notes : il faudrait créer une variable globale et une fonction par note. Une possibilité est d'utiliser que tous les widgets Tkinter sont des objets. On peut leur ajouter des informations de la façon suivante :

```
piano = tk.Canvas(racine, ...)
piano.midi = sortie_midi
piano.volume = 150
piano.canal = 4
touche = tk.Canvas(piano, ...)
touche.note = 60
touche.bind("<Button-1>", appuyer_touche_souris)
```

Ici on a rajouté 3 informations à l'objet piano : piano.midi est une sortie MIDI qui a été calculée auparavant, piano.volume est initialisé à 150 et piano.canal est à 4. A chaque fois qu'on pourra accéder à ce piano, on pourra récupérer ces informations. De la même façon, on rajoute l'information touche.note = 60 à la touche (et si on rajoute touche.note = 61 à une autre touche, elles auront des informations différentes). On peut maintenant écrire notre fonction :

```
def appuyer_touche_souris(evenement):
    touche = evenement.widget # le widget d'1 evenement est l'objet bindé
    piano = touche.master # le master d'1 widget est son contenant
    message = [0x90 + piano.canal, touche.note, piano.volume]
    pyanote.son.message_controle(piano.midi, message)
```

__ *Fonctions anonymes :* il est resté de rares cas où la méthode précédente ne marchait pas. En effet, certains widgets de Tkinter utilisent une sorte de fonction de bind sans argument (nous avons utilisé les Spinbox pour les contrôles du piano, c'est joli mais ça nous a embêté) :

```
piano.canal = 5
canal = tk.Spinbox(controles, ...)
canal.configure(command = essai_changement_canal) # bind d'1 Spinbox

def essai_changement_canal():
   # sans argument, que peut-on faire?
```

Pas d'événement, donc pas de possibilité de récupérer le widget à l'origine de l'événement, donc pas possible de récupérer le widget piano et son piano.canal que l'on veut modifier. La solution que nous avons retenue est l'utilisation de fonctions anonymes. On peut écrire :

```
canal.configure(command = lambda : changement_canal(piano))

def changement_canal(piano):
   # a partir de piano, on peut naviguer
```

Maintenant, on a bindé à la Spinbox une fonction anonyme à 0 arguments (il n'y a rien entre lambda et ' :') qui appelle changement_canal(piano). La construction de cette fonction anonyme se fait au moment du configure, et elle connait bien piano qui est dans l'environnement. Attention, si piano n'est pas connu au moment du configure, ceci nous fera une erreur (difficile à identifier).

Le widget clavier

Mis à part les difficultés d'organisation du code et la longueur (en lignes de codes et en temps) de la réalisation d'une interface graphique, le seul algorithme intéressant a été celui de création d'un octave dans un canevas.

```
def creer_octave(clavier, x_octave, note_debut, w_note, h_note):
    octave = tk.Canvas(clavier, width = w_note * 7, height = h_note)
    octave.place(x=x_octave) # pour decaler les octaves
    intervalles = [0, 2, 4, 5, 7, 9, 11]
    for i in range(len(intervalles)):
        creer_touche(octave, note_debut + intervalles[i],
                     i * w_note, w_note, h_note, "ivory")
    for i in range(1, len(intervalles)):
        if intervalles[i] - intervalles[i-1] == 2: # il y a une touche noire
            x = (i * w_note) - (w_note / 2)
            creer_touche(octave, note_debut + intervalles[i] - 1,
            x, w_note, h_note, "black")

def creer_touche(contenant, note, x_touche, w_note, h_note, couleur):
    touche = tk.Canvas(contenant, width = w_note, height = w_note, bg =
couleur)
    touche.place(x=x_touche)
    touche.note = note
    touche.bind("<Button-1>", appuyer_touche_souris)
```

Le widget karaoke

Nous nous sommes aperçus en téléchargeant des fichiers MIDI sur le net et en les ouvrant avec QuickTime Player que certains contenaient toutes les informations pour afficher un karaoke. Ce sont les fichiers d'extension .kar, qui sont de vrais fichiers MIDI. Maleheureusement, nous n'avons trouvé sur internet aucune information sur ce format. Nécessairement, ces informations devaient être contenues dans les messages meta,

et nous avons d'abord pensé que c'était dans les messages de type 0x05 (Lyrics). En modifiant modificateurs.py pour afficher ces messages, rien. En affichant tous les messages meta, nous avons vu qu'ils étaient dans les messages de type 0x01 (Text Event). Nous avons d'abord trouvé ça idiot, mais nous avons ensuite compris pourquoi : les messages 0x05 sont obligatoirement codés en ASCII, tandis que les messages 0x01 sont en format libre (et rien n'indique le format qu'il faut utiliser pour le décodage). Nous les considérions auparavant comme binaires, mais il fallait maintenant essayer de les décoder. Dans pyanote/utils.py, nous avons réécrit notre fonction de décodage pour qu'elle essaie plusieurs formats dans la liste qu'on lui donne en paramètre. Pour lire la valeur d'un message meta 0x01, nous appelons donc :

```
valeur = utils.lire_chaine(fichier, taille, ['ascii', 'utf-8', 'latin-1'])
```

Pourquoi avoir choisi ces formats ? ASCII est le plus fréquent en MIDI, UTF-8 est apparemment devenu le format d'échange, et LATIN-1 nous permet de récupérer les accents en français.

{Travail collaboratif : répartition des tâches}

[Conclusion]