Projet de conception de fin de deuxième année

Détection du tempo d'une chanson

Pierre Fritsch Thomas Dupas

Avril – Juin 2005

Supélec
Projet encadré par Jean-Luc Collette

Table des matières

1. Introduction 5
1.1. Étendue du projet 5
1.2. Que signifie BPM ? 5

2. Algorithmes de détection du rythme 7
2.1. Intercorrélation avec un peigne de Dirac 7
2.1.1. Principe et limitations théoriques 7
2.1.2. Mise en oeuvre pratique 7
2.1.3. Limitations 8
2.2. Autocorrélation de la chanson 8
2.2.1. Principe 8
2.2.2. Implémentation 11
2.2.3. Avantages et inconvénients 11
2.3. Méthode de détection des crêtes 11
2.3.1. Principe 11
2.3.2. Implémentation 12
2.3.3. Avantages et inconvénients 14

3. Implémentation de “fondu calé” 15
3.1. Précision requise 15
3.1.1. Précision de la mesure du BPM 15
3.1.2. Qualité du modificateur de rythme 16
3.2. Implémentation 16
3.2.1. Mesure du BPM des morceaux 16
3.2.2. Time stretching 16
3.2.3. Superposition des morceaux 16
3.2.4. Fondu 17
3.2.5. Reconstitution en un seul morceau 17

4. Interface graphique du programme de “fondu calé” 19

A. Code source 27
A.1. calcul_bpm.m 27
A.2. calcul_peigne.m 30
A.3. calcul_crete.m 32
A.4. calcul_peigne.m 35
A.5. donner_peigne_superposition.m 37

A.6. synchro_morceaux.m 38
A.7. transition.m 42

B. Algorithme basé sur la mesure de l'énergie **45**
B.1. Principe 45
B.2. Première analyse 45
B.3. Quelques optimisations directes 46
B.4. Sensibilité de la détection 46
B.5. Efficacité de l'algorithme 47

Chapitre 1.

Introduction

Simuler un phénomène physique régi par des équations mathématiques bien connues est toujours faisable, même si cela est souvent soumis à bon nombre d'approximations. Mais qu'en est-il de concepts plus abstraits, comme les sensations ressenties par l'homme, qui ne suivent aucune loi ? Les choses les plus simples à ressentir sont bien souvent les choses les plus dures à traduire en termes de programme informatique. La caractérisation de rythme illustre ce concept : ressentir le rythme d'une chanson est une chose naturelle pour les humains, même les non spécialistes. Néanmoins, il ne s'agit que d'une intuition, d'une impression que l'on ressent en écoutant une mélodie, une impression qui nous fera dancer en rythme ou tapoter sur le coin d'une table sur les batements du morceau.

Ainsi, comment faire acquérir cette capacité à un ordinateur qui ne sait effectuer que des opérations mathématiques ? En fait, il existe quelques algorithmes qui parviennent à approximer, avec plus ou moins de précision, cette détection. Nous avons, au cours de ce projet, étudié différentes méthodes d'extraction de rythme sur des chansons pré-enregistrées.

1.1. Étendue du projet

Après un rapide tour d'horizon des nombreux algorithmes de détection qui existent dans la littérature, nous tenterons d'en implémenter quelques-uns et étudierons leurs limites (chap. 2). Nous appliquerons ensuite cette connaissance à l'exemple concret d'une diffusion musicale, en essayant de faire une transition douce entre deux morceaux de rythmes différents (chap. 3). Ces opérations seront réalisées sous *Matlab*, à l'aide d'une interface graphique décrite au chap. 4. Pour conclure, nous comparerons les algorithmes utilisés.

1.2. Que signifie BPM ?

Les *beats* font référence à la perception d'une pulsation mettant en valeur des périodes de durée constante. Le *tempo* correpond à la fréquence à laquelle ces pulsations se présentent. C'est à cette fréquence que l'être humain tape "naturellement" du pied lorsqu'il entend de la musique. Le BPM (*beats par minute*) est une unité de mesure du tempo : elle représente le nombre de boums ou battements qu'il y a dans un morceau en une minute.

Typiquement, un être humain est capable de détecter la périodicité temporelle d'une chanson entre 40 et 150 BPM. Lorsque les beats se produisent à plus de deux secondes d'intervalle, il les interprète comme une série d'événements isolés, plutôt qu'une séquence unifiée et régulière. De même, en-dessous de 250 ms, les battements sont trop rapprochés.

Le but de ce projet est de mesurer le tempo d'une chanson et de l'exprimer en BPM.

Chapitre 2.

Algorithmes de détection du rythme

Voyons quelques algorithmes de calcul de BPM.

2.1. Intercorrélation avec un peigne de Dirac

Cet algorithme, décrit dans [Pat03], consiste à comparer le signal avec des peignes de Dirac à différents BPM.

2.1.1. Principe et limitations théoriques

Le principe mis en oeuvre par cet algorithme est la mesure de la ressemblance maximale entre une chanson et un peigne de Dirac. Cette mesure est effectuée grâce à l'intercorrélation des deux signaux $\gamma_{xy}[\beta] = \sum_{k=-\infty}^{k=+\infty} x[k]y[k-\beta]$.

L'énergie de la fonction d'*intercorrélation* nous fournit une idée de la *ressemblance entre le signal et le peigne de Dirac testé*. Cela permet en quelque sorte de mesurer combien le rythme du peigne est présent dans la chanson. On répète cette opération pour des peignes de Dirac à différentes fréquences, et on regarde pour lequel d'entre eux l'énergie est la plus grande. On notera E_y cette énergie, tandis que $x[k]$ est le signal représentant l'extrait considéré et $y[k]$ est le peigne de Dirac.

$$E_y = \sum_{k=-\infty}^{+\infty} \gamma_{xy}[k]^2$$

Cette expression étant assez gourmande en temps de calcul, on préfère passer dans le domaine fréquentiel pour s'affranchir du calcul de tous les décalages :

$$E_y = \sum_{k=0}^{N} |X[k] \cdot Y[k]|^2$$

2.1.2. Mise en oeuvre pratique

- On choisit 5 secondes quelque part dans le morceau.
- On calcule la transformée de Fourier $X[k]$ du signal.
- On note $BPMc$ le bpm instantané testé.

- On calcule la période T_i du peigne de Dirac correspondant à $BPMc$ en utilisant la formule

$$T_i = \frac{60}{BPM_c} \cdot fs.$$

- On calcule le peigne de Dirac et sa transformée de Fourier $Y[k]$.
- On calcule finalement l'énergie d'intercorrélation en utilisant la formule :

$$E_{BPMc} = \sum_{k=0}^{N} X[k] \cdot Y[k]$$

- Le rythme de la chanson est donné par la valeur du BPM testé pour laquelle cette quantité est maximale.

Le code source de cet algorithme est donné en annexe, au paragraphe A.2.

La figure 2.1 illustre le fonctionnement de l'algorithme sur le morceau *amber*. Sur la courbe qui donne l'énergie de la corrélation de la chanson avec les peignes, on observe un maximum à 131 BPM, qui est le rythme de la chanson.

2.1.3. Limitations

Cet algorithme présente quelques défauts :
- il est bien trop gourmand en temps de calcul pour être éxécuté en temps réel. Le calcul des transformées de Fourier prend énormément de temps.
- il ne fonctionnera pas du tout pour les chansons trop riches, dans lesquelles on ne voit pas des pics se dégager.

Par contre, il présente l'avantage majeur d'être utilisable directement sur des chansons dont le rythme varie. Par exemple, si une chanson présente deux beats par période, une période sur 4, cet algorithme sera insensible à cette difficulté.

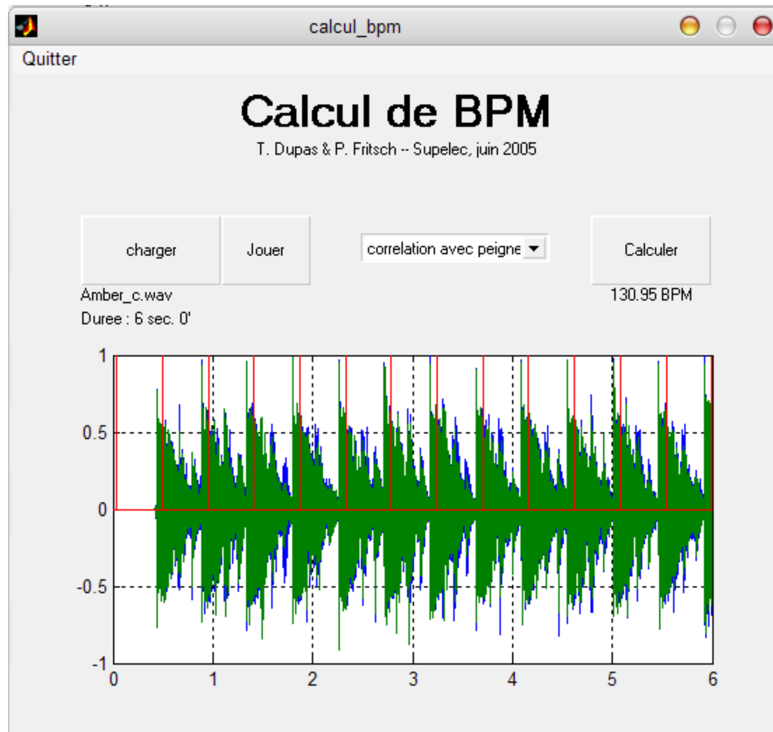
2.2. Autocorrélation de la chanson

Cette méthode s'inspire des travaux de Judith C. Brown [Bro03], qui a eu l'idée d'utiliser l'autocorrélation pour déterminer le *meter* d'une chanson.

2.2.1. Principe

Plutôt que de tester la ressemblance de la chanson avec un peigne de fréquence arbitraire, on exploite le fait que la chanson est périodique et se ressemble donc à elle-même décalée dans le temps. Pour en trouver le rythme, on prend l'ensemble du morceau, on le décale de 4 beats à 100 BPM, et on regarde la corrélation entre ces deux extraits. On recommence l'opération à 101 BPM, et ainsi de suite jusqu'à une borne qu'on s'est fixée (à 160 BPM par exemple).

On obtient ainsi la ressemblance de la chanson avec elle-même décalée. Un exemple est donné en figure 2.2. Il suffit de repérer le maximum de cette courbe et de lire le BPM correspondant pour avoir le rythme de la chanson.



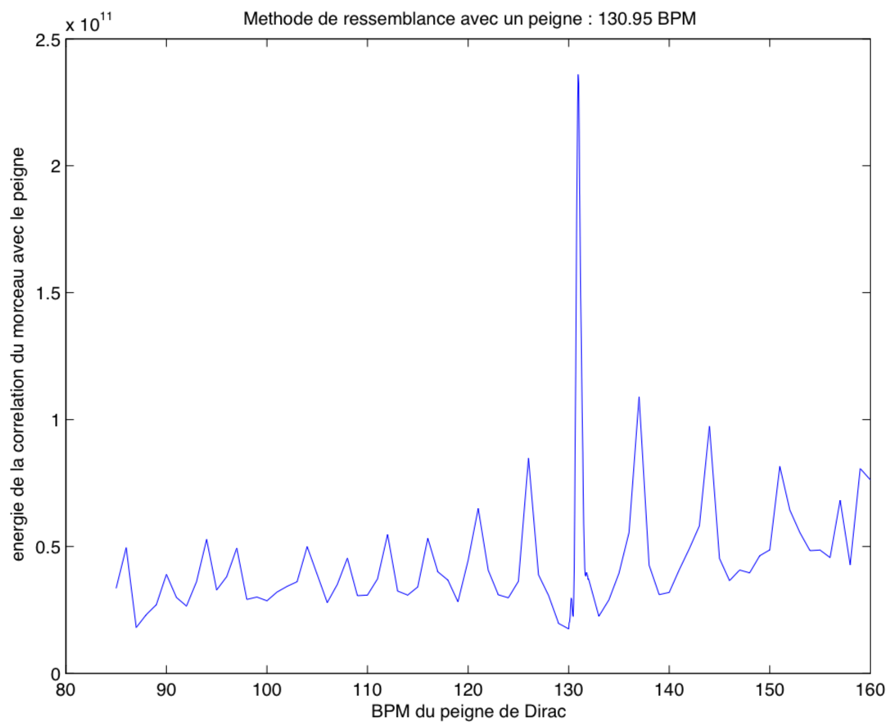
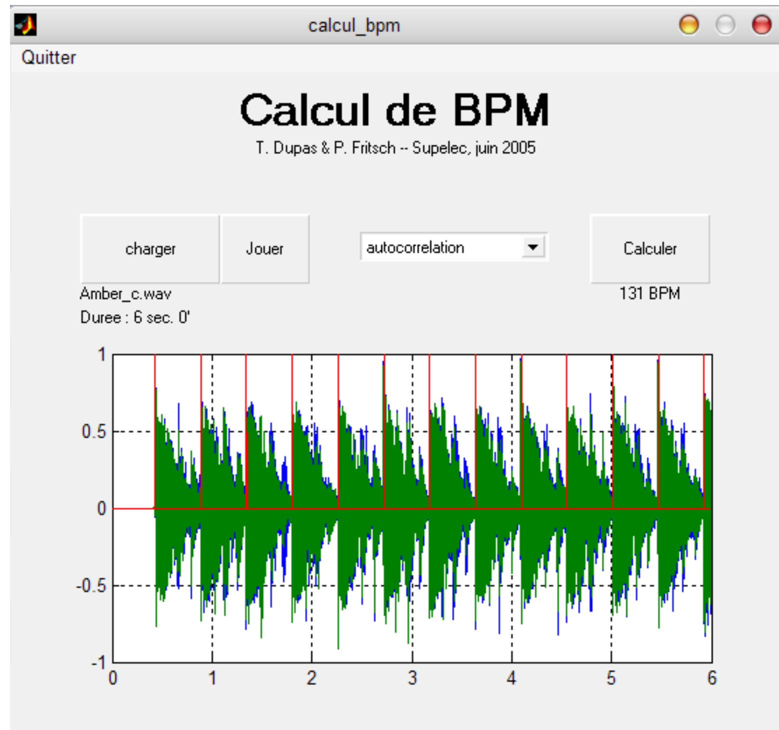


Figure 2.1.: Méthode d'intercorrélation avec un peigne de Dirac



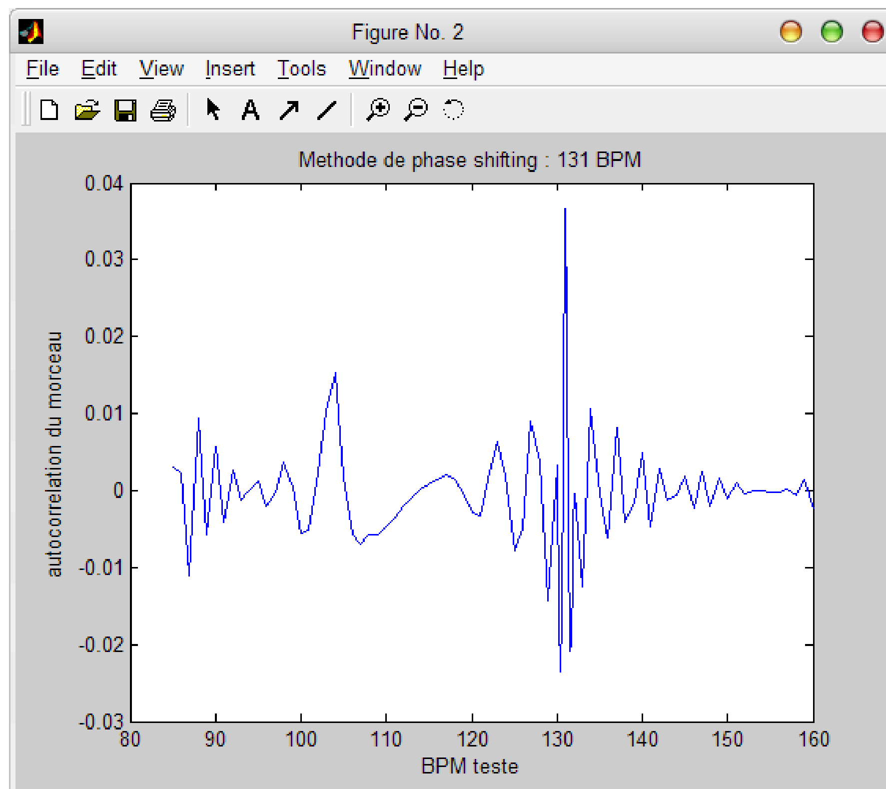


Figure 2.2.: Méthode d'autocorrélation

2.2.2. Implémentation

Le code source de cette méthode est donné en annexe, paragraphe A.2.

1. Pour chacun des BPMs à tester, calculer l'indice dont il faut décaler la chanson :

$$decalage = \frac{4 \cdot 60 \cdot fs}{BPM}$$

2. Pour chacun de ces décalages, calculer l'autocorrélation de la chanson $x[k]$, $k = 1 \ldots n$ avec elle-même décalée dans le temps, et normaliser par la durée de la superposition :

$$correl(decalage) = \sum_{k=1}^{k=n-decalage} \frac{x(k) \cdot x(k+decalage)}{n - decalage}$$

3. Le BPM de la chanson est donné par le maximum de cette courbe.

2.2.3. Avantages et inconvénients

Cette méthode présente des avantages :
- elle est assez robuste et fonctionne sur quasiment n'importe quel type de morceau, pour peu qu'on puisse y repérer des périodicités, et ce même s'il n'y a pas de battements à proprement parler ; de même, si la chanson présente des passages "vides", cela ne pose aucun problème ;
- elle est totalement autonome, et ne nécessite aucun paramètre complémentaire pour fonctionner ;
- elle permet d'obtenir un résultat très précis.

Elle présente néanmoins un inconvénient majeur, qui est son temps d'exécution. De plus, elle est incapable de repérer la position des beats.

2.3. Méthode de détection des crêtes

Cette méthode se base sur l'idée que la musique présente périodiquement des pics d'énergie. Pour détecter ceux-ci, on suit la crête du signal, comme Hess le fait sur des signaux de parole pour en déterminer le pitch ([Hes83], page 195).

Une méthode similaire, qui consiste à détecter les écarts entre l'énergie locale et l'énergie moyenne, est décrite en annexe B.

2.3.1. Principe

L'algorithme, dont le listing est donné en annexe au paragraphe A.3, procède comme suit :

1. On commence par faire un filtrage passe-bas de la chanson pour lisser la courbe sur laquelle on travaille. Pour ce faire, on calcule l'énergie moyenne de la chanson en chacun des points de calcul. Pour accélérer ce calcul, on suréchantillonne le signal.

2. On construit un signal “alpiniste” qui va grimper les crêtes qu’il rencontre et redescendre en deltaplane. En des termes plus mathématiques, si $x[k]$ est l’énergie du signal musical, alors $h[k]$ le signal alpiniste est défini par

$$h[1] = x[1]$$
$$h[k] = \begin{cases} x[k] & \text{si } x[k] \geq h[k-1] - \varepsilon \\ h[k-1] - \varepsilon & \text{sinon (dcroissance linaire)} \end{cases}$$

3. Chaque fois que le signal alpiniste h est en train d’escalader, i.e. de recopier x, on est sur un beat. Ailleurs, on n’est pas sur un beat.
4. À la fin du morceau, on compte le nombre de tels beats et on divise par la longueur du morceau pour connaître son tempo.

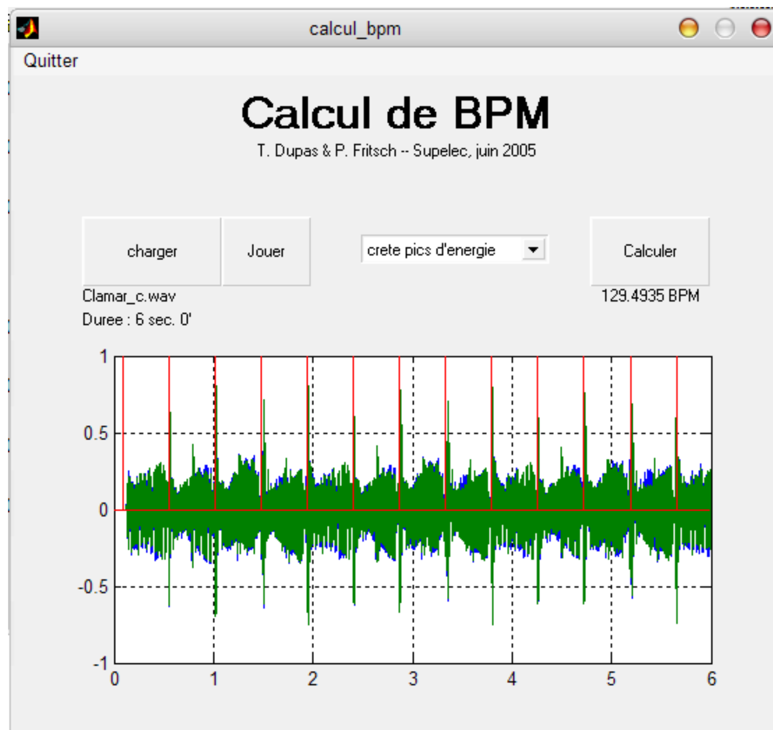
La figure 2.3 montre l’algorithme en situation sur un morceau présentant des pics nets.

2.3.2. Implémentation

L’implémentation de cet algorithme soulève néanmoins quatre problèmes, que nous allons résoudre :

– Au début du morceau, l’algorithme considère toujours qu’on est sur un pic, puisqu’il faut recopier x pour initialiser h. Pour éviter ce désagrément, on enregistre l’altitude des premiers pics, et on estime qu’il ne s’agissait pas de pics si on en rencontre un ultérieurement qui est sensiblement plus grand, par exemple un pic qui monte 40 % plus haut.
– On a besoin d’un nombre entier de périodes pour pouvoir faire la division. Pour obtenir ce résultat, une fois les pics repérés, on découpe le morceau de sorte à obtenir pile un multiple de la période, en commençant le découpage juste avant le premier (vrai) pic, et en le terminant juste avant le dernier pic. En divisant cette durée par le nombre de beats qu’elle contient, on arrive à une mesure précise du tempo.
– Si les pics ne sont pas suffisamment raides, c’est-à-dire si l’énergie met du temps à augmenter, l’algorithme risque de considérer que plusieurs pics se sont présentés. On laisse donc un peu de temps à l’alpiniste pour escalader la montagne (environ 20 ms), afin qu’il arrive au sommet sans qu’on ait l’impression de gravir plusieurs pics.
– Si deux pics sont trop rapprochés alors qu’il ne s’agit pas de deux beats consécutifs, la mesure est faussée. On s’affranchit de ce défaut en définissant une durée minimale entre deux beats consécutifs, qui correspond à cette durée pour le BPM maximal détectable.

En faisant ainsi, on parvient à obtenir des résultats cohérents avec cet algorithme, sans rallonger de beaucoup le temps de calcul (il s’agit uniquement de branchements conditionnels et de découpages).



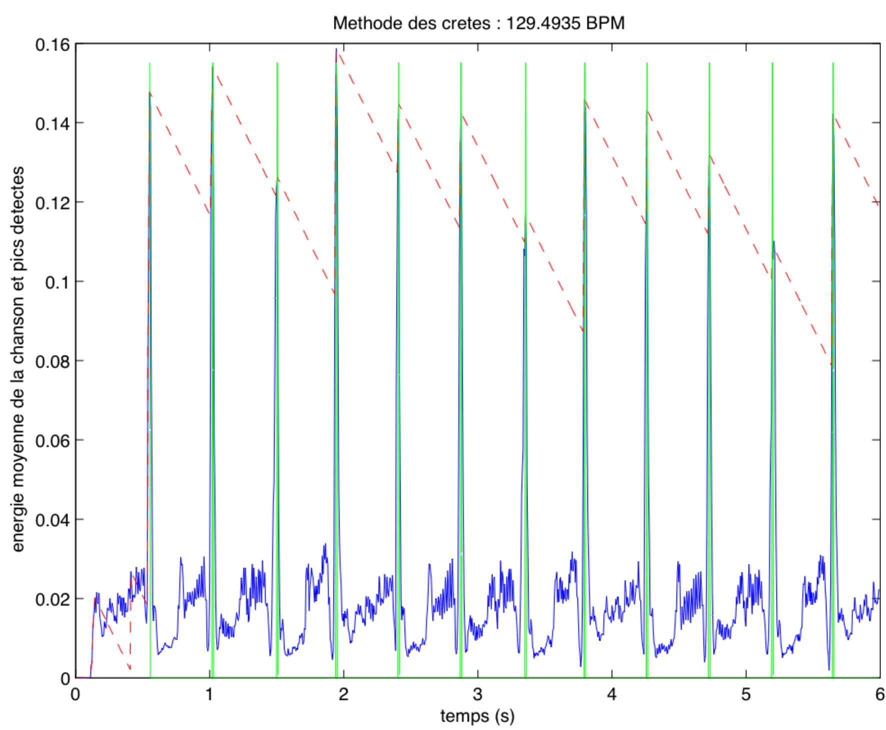


Figure 2.3.: Méthode de detéction des crêtes

2.3.3. Avantages et inconvénients

Contrairement aux autres algorithmes, celui-ci est très *rapide*, au point qu'on pourrait envisager une exécution en temps réel. Le seul calcul lourd effectué est le filtrage passe-bas, le reste n'est que la comparaison avec des seuils.

De plus, en cas de variation brutale du rythme, on obtient le rythme moyen de la chanson, là où les autres algorithmes donneraient un résultat sans aucun sens.

Malgré ces avantages certains, il présente des inconvénients majeurs :

- il présuppose que la chanson présente un beat par période, et est donc complètement perdu si elle en présente plus ;
- la constante de décroissance ε doit être réglée pour s'adapter au morceau étudié ;
- si des beats de faible énergie se présentent, alors ils risquent de ne pas être détectés.

C'est donc un excellent algorithme pour des chansons dont le rythme est bien marqué, mais qui sert difficilement dans les autres cas.

Chapitre 3.

Implémentation de “fondu calé”

Une application de cette détection de rythme est la réalisation de transitions entre des chansons de rythme différent. Pour ce faire, on procède en plusieurs étapes :

1. détection du rythme des deux chansons ;
2. “étirement” des chansons (*time stretching*) pour qu’elles aient le même rythme ;
3. “calage” des chansons, i.e. superposition des *beats* de deux chansons ; comme elles ont même rythme, cette superposition va se maintenir dans le temps ;
4. finalement, passage en douceur d’une chanson à l’autre (fondu).

C’est ainsi qu’opèrent les *disc jockeys* lorsqu’ils enchaînent deux chansons.

3.1. Précision requise

Cette transition nécessite une grande précision dans le calcul du BPM.

3.1.1. Précision de la mesure du BPM

En effet, disons qu’on veut pouvoir synchroniser deux chansons sur une durée de 30 secondes. Au cours de cette durée, il faudrait qu’il n’y ait pas de différence de rythme audible entre les deux chansons.

Une oreille non exercée est capable de détecter des décalages de battements à partir d’un 64ème de période, autrement dit un 16ème de beat. L’indication des BPMs devrait donc être précise à 0,0625 BPM près, disons 0,05 BPM. Avec une erreur de mesure de 0,05 BPM, on obtient, sur une durée de 30 secondes, un décalage entre beats qui vaut :

$$0,05\frac{\text{beat}}{\text{min}} \cdot 30\text{sec} = 0,25\text{beat},$$

ce qui, si on note x le tempo de la chanson en BPM, correspond à un décalage temporel

$$\frac{0,25\text{beat}}{x\frac{\text{beat}}{\text{min}}\frac{1\text{min}}{60\text{sec}}} = \frac{15}{x}\text{sec}$$

Ainsi, pour une chanson à 120 BPM, une erreur de mesure de 0,05 BPM induit, sur une durée de 30 secondes, un décalage temporel de l’ordre de 125 ms.

3.1.2. Qualité du modificateur de rythme

Pour accélérer/ralentir les chansons de façon à pouvoir exploiter la résolution de mesure avec son pas de 0,05 BPM, il s'agit d'avoir un algorithme qui permette d'étirer des chansons de façon très précise.

Pour une chanson à 120 BPM par exemple, il s'agit de pouvoir modifier le rythme avec un facteur $0,0417\ \%$ si on veut pouvoir réaliser des calages sur des longues durées.

3.2. Implémentation

Décrivons les étapes de l'algorithme de fondu calé.

3.2.1. Mesure du BPM des morceaux

Dans un premier temps, on a besoin des BPMs des morceaux qu'il faut enchaîner. Pour calculer ceux-ci, on utilise un des algorithmes décrits dans le chapitre 2. Ce calcul étant assez long, on peut l'effectuer seulement sur une partie des chansons. En particulier, on peut se contenter d'utiliser la fin du premier morceau pour y calculer le BPM, ainsi que le début du second morceau. En ne conservant que ces parties des chansons, on accélère le processus de calcul, tout en s'assurant que la mesure est fiable, puisque le BPM risque moins de varier sur des parties courtes de chansons.

3.2.2. Time stretching

On accorde ensuite le rythme des deux chansons. Pour cela, nous avons besoin d'un algorithme dit de *time stretching* permettant d'étirer ou de compresser temporellement un morceau musical ou de voix sans en déformer la tonalité (sans que Celine Dion emprunte la voix de Barry White, par exemple).

Comme il ne s'agit que d'une question périphérique par rapport au projet initial, nous nous sommes résolus à chercher un script déjà implémenté. Nous avons commencé par utiliser un algorithme fourni par l'enseignant encadrant, Jean-Luc Collette, mais il s'est vite avéré que la précision de celui-ci était insuffisante pour l'utilisation que nous voulions en faire (cf. paragraphe 3.1.2).

Ainsi nous avons trouvé sur le net une librairie contenant un vocoder tout fait appelé *pVoc* dont la précision s'est révélée satisfaisante. Le fichier utile dans ce répertoire est `stretchFile.m` et nous l'avons modifié de façon à ce qu'il renvoie un vecteur et non un fichier d'extension `.wav` comme c'était le cas à l'origine. `stretchFile` prend comme argument un fichier d'extension `.wav` ainsi qu'un rapport de transformation sous forme du quotient de la longueur finale souhaitée sur la longueur initiale du morceau.

Le fichier *Matlab* qui permet d'utiliser cet algorithme s'appelle `mettre_au_bpm.m`.

3.2.3. Superposition des morceaux

Pour superposer les morceaux, on a besoin de connaître la position des pics dans chacun d'eux. On obtient celle-ci en cherchant le maximum de l'intercorrélation de la

chanson avec un peigne de Dirac au même BPM que la chanson.

Partant de là, on superpose les chansons de sorte qu'elles se recouvrent pendant la durée souhaitée de la transition, tout en ayant leurs beats calés. Autrement dit, on fait en sorte que les deux peignes soient confondus.

3.2.4. Fondu

Le résultat de l'opération est obtenu en gardant le début de la première chanson, accolé au fondu de la fin de la première chanson avec le début de la deuxième, le tout suivi par la fin de la deuxième chanson.

Le fondu est construit en sommant le fondu sortant du premier morceau avec le fondu entrant du deuxieme morceau. Pour réaliser un fondu sortant, on multiplie terme à terme la fin du morceau par un triangle décroissant de 1 à 0, de largeur égale à la durée du fondu. Pour réaliser un fondu entrant, on multiplie par le triangle complémentaire, i.e. le triangle croissant obtenu en soustrayant le premier à un vecteur de 1.

3.2.5. Reconstitution en un seul morceau

Il ne reste plus qu'à recoller les différentes parties de morceaux de musiques, c'est-à-dire les mettre bout à bout. On commence par la partie du premier morceau située avant le premier pic de la section finale testée, on lui accole la partie “fondu” préalablement calée comme expliqué ci dessus, il ne reste plus alors qu'à terminer en accolant la partie du second morceau située après l'extrait impliqué dans le fondu. Reste à écouter le morceau ainsi synthétisé.

Chapitre 4.

Interface graphique du programme de “fondu calé”

Nous avons réalisé une interface graphique pour la synthèse d’une transition, ce paragraphe a pour but de décrire son utilisation.

L’interface permettant d’effectuer le “fondu calé” est très simple d’utilisation.

En lançant synchro_morceau.m une fenêtre (figure : 4.1) apparait.

En cliquant sur le bouton “charger” un menu de selection du fichier se superpose à la fenêtre initiale (figure : 4.2). On peut naviguer dans nos données pour trouver le fichier d’extension wav désiré.

Une fois sélectionné le morceau est affiché dans une fenêtre sous le menu de sélection avec sa durée (figure : 4.3).

On peut sur le même mode sélectionner le second morceau.

Une fois les deux pistes chargées, il ne reste plus qu’à sélectionner le Bpm visé et la durée de transition souhaitée puis cliquer sur le bouton calculer.

Après un temps de calcul conséquent s’affiche la fenêtre figure 4.3.

Le résultat du calcul s’affiche sous forme de trois fenêtres. La première qui reprend l’interface graphique générale présente le graphique de la partie dite de transition avec les peignes de Dirac superposés (figure : 4.4). Deux autres fenêtres présentent l’autocorrélation des morceaux avec le BPM en abscisse afin de permettre d’évaluer leur cadence initiale (figure : 4.5).

FIG. 4.1.: Fenêtre d'accueil

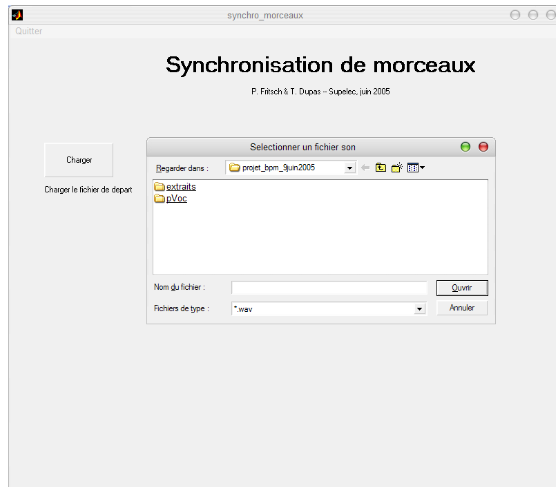


FIG. 4.2.: Sélection du morceau N°1

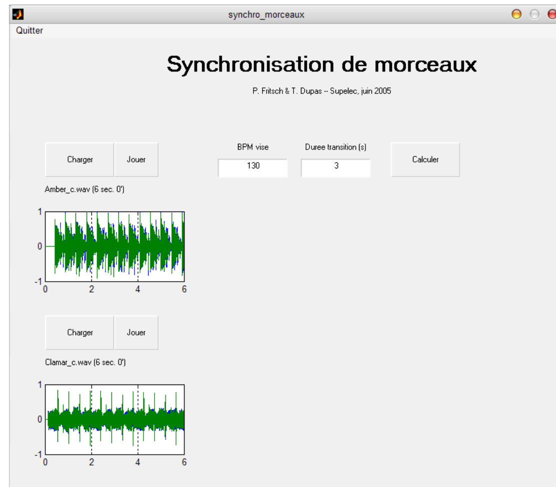


Fig. 4.3.: Sélection terminée

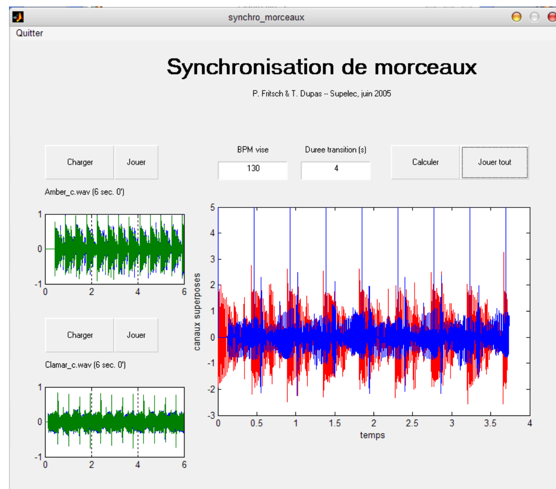


Fig. 4.4.: Représentation de la transition

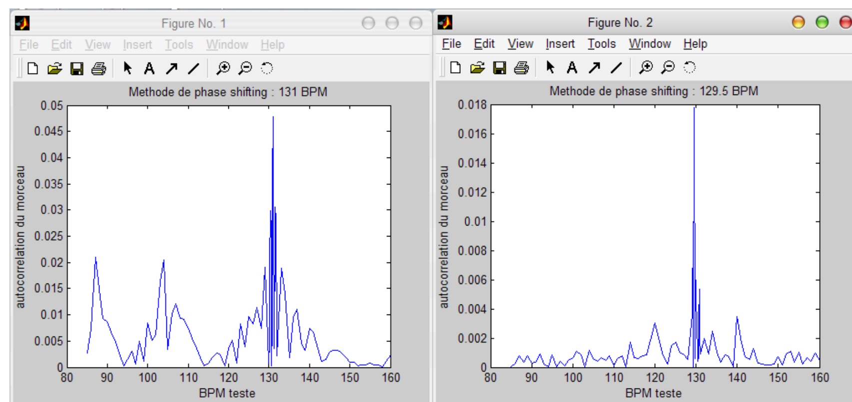


FIG. 4.5.: Autocorrélations et rythmes initiaux

Conclusion

Nous avons implémenté trois algorithmes de détection du tempo d'une chanson :

– la ressemblance spectrale avec un peigne de Dirac ;
– l'autocorrélation de la chanson avec elle-même ;
– la détection des pics d'énergie par crêtes.

Les limites d'utilisation de ces algorithmes sont illustrés par le tableau 4.1, qui compare leurs possibilités. Le tableau 4.2 donne leurs performances sur deux morceaux de test.

Dans tous les cas, pour s'affranchir des variations lentes du tempo, il faut prendre un extrait de 5 secondes dans la chanson et appliquer les algorithmes sur ce court morceau. Dès lors on considère que le tempo de la chanson ne varie pas au cours de celle-ci ou très peu et que cette partie de la chanson est représentative du reste au niveau tempo.

Possibilité	Ressemblance peigne	Autocorrélation	Crête énergie
temps réel	non	non	oui
rythme compliqué	oui	oui	non
chanson “riche”	non	oui	non
variation brutale du tempo	non	non	oui
autonome = aucun réglage	oui	oui	non

Tab. 4.1.: Comparaison des caractéristiques des algorithmes

Extrait utilisé	*amber*		*clamar*	
	BPM	Temps de calcul	BPM	Temps de calcul
Autocorrélation	131	6.9s	129	6.9s
Crêtes énergie	133	0.5s	129	0.43s
Ressemblance peigne	131	48.32s	155	48.35s

Table 4.2.: Performance des algorithmes

Bibliographie

[Sch97] Eric D. Scheirer, Tempo and beat analysis of acoustic musical signals, Acoustical Society of America, 1997

[Sch00] Eric D. Scheirer, Music-Listening Systems, MIT, 2000

[Bro03] Judith C. Brown, Determination of the meter of musical scores by autocorrelation, J. Acoust. Soc. Am., 2003

[Pat03] Frédéric Patin, Beat Detection Algorithms, http ://yov408.free.fr, 2003

[Alg] Masoud Alghoniemy and Ahmed H. Tewfik, Rhythm and periodicity detection in polyphonic music

[McK04] Martin F. McKinney & Dirk Moelants, Extracting the perceptual tempo from music, Universitat Pompeu Fabra, 2004

[Hes83] Wolfgang Hess, Pitch Determination of Speech Signals : Algorithms and Devices,Springer-Verlag Berlin, 1983

Annexe A.

Code source

Cette annexe contient les codes source du projet. Les fichiers d'interface ne sont pas inclus.

A.1. calcul_bpm.m

```
function varargout = calcul_bpm(varargin)
% CALCUL_BPM M-file for calcul_bpm.fig
%      CALCUL_BPM, by itself, creates a new CALCUL_BPM or raises the existing
%      singleton*.
%
%      H = CALCUL_BPM returns the handle to a new CALCUL_BPM or the handle to
%      the existing singleton*.
%
%      CALCUL_BPM('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in CALCUL_BPM.M with the given input arguments.
%
%      CALCUL_BPM('Property','Value',...) creates a new CALCUL_BPM or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before calcul_bpm_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to calcul_bpm_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help calcul_bpm

% Last Modified by GUIDE v2.5 09-Jun-2005 10:24:41

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @calcul_bpm_OpeningFcn, ...
                   'gui_OutputFcn',  @calcul_bpm_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
```

```
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before calcul_bpm is made visible.
function calcul_bpm_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to calcul_bpm (see VARARGIN)

% Choose default command line output for calcul_bpm
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes calcul_bpm wait for user response (see UIRESUME)
% uiwait(handles.fenetre);


% --- Outputs from this function are returned to the command line.
function varargout = calcul_bpm_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in load.
function load_Callback(hObject, eventdata, handles)
% hObject    handle to load (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[FileName,PathName] = uigetfile('*.wav','Selectionner un fichier son');
if isequal(FileName,0) | isequal(PathName,0)
    return
end
try
    [son,fe] = wavread(fullfile(PathName, FileName));
catch
    return
end
disp(['Chargement du fichier ' FileName]);
handles.nom = FileName;
set(handles.fichier, 'String', FileName);
set(handles.duree, 'String', ['Duree : ' duree_texte(length(son)/fe)]);
set(handles.play, 'Visible', 'on');
set(handles.choix, 'Visible', 'on');
set(handles.start, 'Visible', 'on');
set(handles.text_bpm, 'String' , '');
handles.son = son;
handles.fe = fe;
handles.temps = (0 : size(handles.son,1) - 1)/handles.fe;
plotter_son(handles);
guidata(hObject, handles);
```

```
% --- Executes on button press in play.
function play_Callback(hObject, eventdata, handles)
% hObject    handle to play (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
sound(handles.son, handles. fe);

% --- Executes during object creation, after setting all properties.
function choix_CreateFcn(hObject, eventdata, handles)
% hObject    handle to choix (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


% --- Executes on selection change in choix.
function choix_Callback(hObject, eventdata, handles)
% hObject    handle to choix (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns choix contents as cell array
%        contents{get(hObject,'Value')} returns selected item from choix


% --- Executes on button press in start.
function start_Callback(hObject, eventdata, handles)
% hObject    handle to start (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

t_depart = cputime;

texteintro = 'Lancement du calcul -- methode choisie : ';

choix = get(handles.choix, 'Value');
switch choix
    case 1
        % autocorrelation
        disp([texteintro 'autocorrelation']);
        bpm = calcul_autocorrelation(handles.son, handles.fe);
    case 2
        % energie
        disp([texteintro 'detection des pics d''energie (crete)']);
        bpm = calcul_crete(handles.son, handles.fe);
    case 3
        % peignes de Dirac
        disp([texteintro 'correlation avec des peignes de Dirac']);
        bpm = calcul_peigne(handles.son, handles.fe);
end
duree = cputime - t_depart;

disp(['Le calcul a utilise ' num2str(duree) 's. de temps CPU']);

% mise a jour de l'affichage du BPM dans la fenetre
```

```
set(handles.text_bpm, 'String', [num2str(bpm) ' BPM']),

% plot de la chanson avec un peigne en surimpression
disp(['Recherche du peigne qui cadre au mieux avec la chanson a ' ...
    num2str(bpm) 'BPM']);
peigne = donner_peigne_superposition(handles.son, bpm, handles.fe);
plotter_son(handles);
hold on;
plot(handles.temps, peigne, 'r');
hold off;

% --------------------------------------------------------------------
function quitter_Callback(hObject, eventdata, handles)
% hObject    handle to quitter (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
delete(handles.fenetre);

% plotte handles.son dans handles.courbe
function plotter_son(handles)
    axes(handles.courbe);
    plot(handles.temps, handles.son);
    ech = axis;
    axis([ech(1:2) -1 1]);
    grid on
```

A.2. calcul_peigne.m

```
% Calcul du BPM par autocorrelation (phase shifting)
% Pierre Fritsch, 8 juin 2005
% inspire par Bro03
%
% bpm = calcul_autocorrelation(son, fs, [pas_fin, [bpm_debut, bpm_fin,
% [pas_grossier]]])
%
% calcule le BPM du vecteur son echantillonne a la frequence fs
% avec une precision pas_fin, en supposant que ce BPM est inclus dans la
% plage [bpm_debut : bpm_fin] parcourue rapidement avec une precision
% de pas_grossier

function bpm = calcul_autocorrelation(son, fs, pas_fin, ...
    bpm_debut, bpm_fin, pas_grossier)

% arguments par defaut
pas_fin_defaut = 0.05;
bpm_debut_defaut = 85;
bpm_fin_defaut = 160;
pas_grossier_defaut = 1;

switch nargin
    case 6
        % rien a faire, youpie !
    case 5
        pas_grossier = pas_grossier_defaut;
    case 3
        pas_grossier = pas_grossier_defaut;
        bpm_debut = bpm_debut_defaut;
        bpm_fin = bpm_fin_defaut;
    case 2
        pas_grossier = pas_grossier_defaut;
        bpm_debut = bpm_debut_defaut;
```

```
        bpm_fin = bpm_fin_defaut;
        pas_fin = pas_fin_defaut;
    otherwise
        disp('Erreur : pas le bon nb d''arguments');
        return;
end

% initialisation du calcul
pas = pas_grossier;
trouve_grossier = 0;
trouve_bpm = 0;


%son = son.^2;

while (trouve_bpm == 0)

    % Plage de BPMs a parcourir :
    plage_bpm = bpm_debut : pas : bpm_fin;

    % correspondant, sur 4 mesures, a des decalages (en echantillons) :
    decalage = 4./plage_bpm*60*fs;

    correl = zeros(size(decalage));

    tic;
    disp(['Parcours de la plage ' num2str(bpm_debut) ':' ...
        num2str(pas) ':' num2str(bpm_fin) ' :']);
    disp(['Calcul de ' num2str(length(decalage)) ...
        ' autocorrelations en cours sur des vecteurs de taille > ' ...
        num2str(length(son) - decalage(end))]);

    % Pour chacun des decalages
    for i = 1:length(decalage)
        decal = floor(decalage(i));
        % caculer l'intercorrelation du morceau
        % avec lui-meme decale de decalage
        % et normaliser par la largeur de l'echantillon

        extrait2 = son(min(1 + decal, end) : end, :);
        duree = length(extrait2);
        extrait1 = son(1 : duree, :);

        % Le mean est la pour le cas des morceaux stereo
        % on se ramene a un seul canal
        correl(i) = abs(mean(sum(extrait1 .* extrait2)) / duree);
    end

    t = toc;

    % Recuperation du BPM max de ce passage
    [correl_max, indice_max] = max(correl);
    bpm = plage_bpm(indice_max);
    disp(['BPM proche de ' num2str(bpm) ' (trouve en ' num2str(t) 's)']);

    if trouve_grossier == 0
        % On vient de faire le tour avec un pas grossier
        trouve_grossier = 1;

        % la prochaine fois on fera un refinement
        pas = pas_fin;
```

```
        % autour de la valeur grossiere trouvee ici
        bpm_debut = bpm - pas_grossier + pas_fin;
        bpm_fin = bpm + pas_grossier - pas_fin;

        % on sauvegarde les resultats pour affichage final
        correl_grossier_debut = correl(1 : indice_max - 1);
        correl_grossier_fin = correl(indice_max + 1 : end);
        bpm_grossier_debut = plage_bpm(1 : indice_max - 1);
        bpm_grossier_fin = plage_bpm(indice_max + 1 : end);
    else
        % On vient de faire le deuxieme passage (fin)
        trouve_bpm = 1;
        % On reconstruit l'ensemble des correlations pour le trace
        correl = [correl_grossier_debut correl correl_grossier_fin];
        plage_bpm = [bpm_grossier_debut plage_bpm bpm_grossier_fin];
    end
end

figure
plot(plage_bpm, correl);
title(['Methode de phase shifting : ' num2str(bpm) ' BPM']);
xlabel('BPM teste');
ylabel('autocorrelation du morceau');
```

A.3. calcul_crete.m

```
% Calcul du BPM par detection des cretes d'energie
% Pierre Fritsch, 8 juin 2005
% inspire par Hess, p. 195
%
% bpm = calcul_crete(son, fs, [coeff_decroissance, ...
%  [ecart_pretraitement, [largeur_moyennage, [bpm_max, [duree_montee]]]]])
%
% calcule le BPM du vecteur son echantillonne a la frequence fs
% on compare la valeur a chaque instant avec la valeur obtenue en
% decroissance lineaire avec coeff_decroissance
%
% Au prealable, on calcule la moyenne de l'energie sur des
% tranches de longueur largeur_moyennage, et on ne fait ça que tous les
% ecart_pretraitement echantillons du signal
% On suppose que le BPM a detecter ne depasse pas bpm_max pour eliminer
% les false positive, et on laisse au signal la possibilite d'atteindre
% son maximum en duree_montee secondes lorsqu'il monte.

% IL FAUT AU MOINS DEUX PICS 'propres' DANS LA CHANSON !!!!!

function bpm = calcul_crete(son, fs, coeff_decroissance, ...
    ecart_pretraitement, largeur_moyennage, bpm_max, duree_montee)

% valeurs par defaut
largeur_moyennage_defaut = 1024;
ecart_pretraitement_defaut = largeur_moyennage_defaut/4;
coeff_decroissance_defaut = 0.0004;

% si un pic est 40% plus grand que le premier pic detecte,
% alors il est considere comme etant le premier pic
coeff_detection_premierPic = 1.4;

% bpm maximal detectable
% sert a calculer l'ecart minimal entre deux pics consecutifs
bpm_max_defaut = 160;
```

```
% duree de montee d'un pic, en s
duree_montee_defaut = 0.070;

switch nargin
    case 2
        coeff_decroissance = coeff_decroissance_defaut;
        ecart_pretraitement = ecart_pretraitement_defaut;
        largeur_moyennage = largeur_moyennage_defaut;
        bpm_max = bpm_max_defaut;
        duree_montee = duree_montee_defaut;
    case 3
        ecart_pretraitement = ecart_pretraitement_defaut;
        largeur_moyennage = largeur_moyennage_defaut;
        bpm_max = bpm_max_defaut;
        duree_montee = duree_montee_defaut;
    case 4
        largeur_moyennage = largeur_moyennage_defaut;
        bpm_max = bpm_max_defaut;
        duree_montee = duree_montee_defaut;
    case 5
        bpm_max = bpm_max_defaut;
        duree_montee = duree_montee_defaut;
    case 6
        duree_montee = duree_montee_defaut;
    case 7
    otherwise
        disp('Probleme : pas le bon nombre d''arguments');
        return;
end

% Pretraitement carrement passe-bas :
% on remplace le signal par son energie locale
disp('Pretraitement : lissage de la courbe par calcul de l''energie locale');
disp(['  sur ' num2str(largeur_moyennage) ' points tous les ' ...
        num2str(ecart_pretraitement) ' points' ]);
if size(son, 2) ~= 1
    son = mean(son')'; % passage en mono
end

% on surechantillonne le signal
energie = zeros(floor(length(son)/ecart_pretraitement), size(son, 2));

% vecteur temps du signal surechantillonne :
t = 0 : ecart_pretraitement / fs : ...
    (length(energie) - 1) * ecart_pretraitement / fs;

% le calcul de l'energie moyenne se fait autour de l'echantillon
% sur une largeur largeur_moyennage
larg = floor(largeur_moyennage/2);

tic;
for i = 1 : length(energie)
    % position correspondante dans son
    k = i * ecart_pretraitement;

    % calcul de l'energie moyenne du signal
    energie(i,:) = mean(son(max(1, k - larg) : min(end, k + larg), :).^2);
end
duree_calcul = toc;
disp(['Pretraitement : ' num2str(length(energie)) ...
        ' valeurs obtenues en ' num2str(duree_calcul) ' s']);
```

```
% Traitement proprement dit
% Detection des pics d'energie par decroissance lineaire

% ecart minimal entre deux pics consecutifs
% exprime en nombre d'echantillons d'energie
ecart_minimal = round( 60 * fs / bpm_max * 1/ecart_pretraitement);

% largeur de montee d'un pic
% exprime en nombre d'echantillons d'energie
largeur_montee = duree_montee*fs*ecart_pretraitement;

% signal alpiniste
h = zeros(size(energie));

% enregistrement des pics du signal
pics = zeros(size(energie));

h(1) = energie(1);

% on enregistre la position du premier pic
% ainsi que sa valeur pour eliminer le bruit du debut
indicePremierPic = 1;
valeurPremierPic = h(1);

% on enregistre la position du dernier pic
% pour avoir un nombre entier de periodes
indiceDernierPic = 1;

% Parcours du vecteur a la recherche des montees brusques
for i = 2 : length(energie)

    h(i) = h(i-1) - coeff_decroissance;
    % Si
    if h(i) <= energie(i)
        % on vient de recontrer un debut de pic
        valeur_de_h = h(i);
        h(i) = energie(i);

        % Mise a jour de l'enregistrement du premier pic
        if (energie(i) >= valeurPremierPic * coeff_detection_premierPic)
            indicePremierPic = i;
            valeurPremierPic = energie(i);
        end
        pics(i) = 1;

        % Post-traitement
        % Si on est sur un pic et qu'il y a eu un pic juste avant
        % on est vraisemblablement sur une montee.
        % on en garde le sommet
        pics_precedents = pics(max(i - largeur_montee, i - 1) : i - 1);
        if (pics(i) ~= 0 && sum(pics_precedents) ~= 0)
            pics(max(i - largeur_montee, i - 1) : i - 1) = zeros(size(pics_precedents));
        end

        if sum(pics(max(i - ecart_minimal, indicePremierPic) : i - 1)) ~= 0
            % il y a eu un pic recemment
            % donc celui-ci n'en est pas un
            pics(i) = 0;

            % on fait comme si on l'avait pas vu
```

```
            h(i) = valeur_de_h;
        else
            % il n'y a pas eu de pic recemment
            % donc celui-ci en est un
            % c'est peut-etre le dernier pic
            indiceDernierPic = i;
        end
    end

end


% RESULTAT :
% =========
nbPics = sum(pics(indicePremierPic:indiceDernierPic - 1));
dureeTotale = (indiceDernierPic - indicePremierPic) * ecart_pretraitement / fs;

disp([num2str(nbPics) ' periodes utilisent une duree de ' num2str(dureeTotale) ' sec. ']);

% une simple regle de trois nous dit combien il y en a par minute
bpm = nbPics/dureeTotale * 60;

figure;
plot(t', energie, 'b', t', h, 'r--', ...
    t(indicePremierPic : end)', pics(indicePremierPic : end) * valeurPremierPic * 1.05, 'g');
xlabel('temps (s)');
ylabel('energie moyenne de la chanson et pics detectes');
title(['Methode des cretes : ' num2str(bpm) ' BPM']);
```

A.4. calcul_peigne.m

```
% Calcul du BPM par ressemblance (intercorrelation) avec un peigne de Dirac
% Pierre Fritsch, 8 juin 2005
% d'apres Pat03, "Filtering rhythm detection"
%
% bpm = calcul_peigne(son, fs)
%
% calcule le BPM du vecteur son echantillonne a la frequence fs

function bpm = calcul_peigne(son, fs, pas_fin, ...
    bpm_debut, bpm_fin, pas_grossier)

% arguments par defaut
pas_fin_defaut = 0.05;
bpm_debut_defaut = 85;
bpm_fin_defaut = 160;
pas_grossier_defaut = 1;

switch nargin
    case 6
        % rien a faire, youpie !
    case 5
        pas_grossier = pas_grossier_defaut;
    case 3
        pas_grossier = pas_grossier_defaut;
        bpm_debut = bpm_debut_defaut;
        bpm_fin = bpm_fin_defaut;
    case 2
        pas_grossier = pas_grossier_defaut;
        bpm_debut = bpm_debut_defaut;
        bpm_fin = bpm_fin_defaut;
```

```
        pas_fin = pas_fin_defaut;
    otherwise
        disp('Erreur␣:␣pas␣le␣bon␣nb␣d''arguments');
        return;
end

% initialisation du calcul
pas = pas_grossier;
trouve_grossier = 0;
trouve_bpm = 0;

% longueur du fichier son
longueur = length(son);

if size(son, 2) ~= 1
son = mean(son')'; % passage en mono
end

while (trouve_bpm == 0)

    % Plage de BPMs a parcourir :
    plage_bpm = bpm_debut:pas:bpm_fin;

    energie = zeros(size(plage_bpm));

    tic;
    disp(['Parcours␣de␣la␣plage␣' num2str(bpm_debut) ':' ...
        num2str(pas) ':' num2str(bpm_fin) '␣:']);

    % Pour chacun des BPMs
    for i = 1:length(plage_bpm)

        % generation d'un peigne de Dirac au bon BPM
        % necessite de faire taire les warnings (indices non entiers)
        periode = 60/plage_bpm(i) * fs; % periode du peigne
        etat_warning = warning; warning off;
        peigne = zeros(longueur, 1);
        peigne(1 : periode : longueur) = 1; % creation du peigne
        warning(etat_warning); % reactivation des warning

        energie(i) = sum(abs(fft(son).*fft(peigne)).^2);
    end

    t = toc;

    % Recuperation du BPM max de ce passage
    [energie_max, indice_max] = max(energie);
    bpm = plage_bpm(indice_max);
    disp(['BPM␣proche␣de␣' num2str(bpm) '␣(trouve␣en␣' num2str(t) 's)']);

    if trouve_grossier == 0
        % On vient de faire le tour avec un pas grossier
        trouve_grossier = 1;

        % la prochaine fois on fera un refinement
        pas = pas_fin;

        % autour de la valeur grossiere trouvee ici
        bpm_debut = bpm - pas_grossier + pas_fin;
        bpm_fin = bpm + pas_grossier - pas_fin;

        % on sauvegarde les resultats pour affichage final
```

```
        energie_grossier_debut = energie(1 : indice_max - 1);
        energie_grossier_fin = energie(indice_max + 1 : end);
        bpm_grossier_debut = plage_bpm(1 : indice_max - 1);
        bpm_grossier_fin = plage_bpm(indice_max + 1 : end);
    else
        % On vient de faire le deuxieme passage (fin)
        trouve_bpm = 1;
        % On reconstruit l'ensemble des correlations pour le trace
        energie = [energie_grossier_debut energie energie_grossier_fin];
        plage_bpm = [bpm_grossier_debut plage_bpm bpm_grossier_fin];
    end
end

figure
plot(plage_bpm, energie);
title(['Methode de ressemblance avec un peigne : ' num2str(bpm) ' BPM']);
xlabel('BPM du peigne de Dirac');
ylabel('energie de la correlation du morceau avec le peigne');
```

A.5. donner_peigne_superposition.m

```
% Recherche de la position optimale d'un peigne de Dirac
% pour superposition au-dessus d'une chanson de bpm connu
% Pierre Fritsch, 8 juin 2005
%
% peigne = donner_peigne_superposition(vecteur_son, bpm, fs)

function peigne = donner_peigne_superposition(vecteur_son, bpm, fs)

    % longueur du vecteur son donne en entree
    longueur_son = length(vecteur_son);

    % passage en mono si le son est en stereo
    if (size(vecteur_son, 2) == 2)
    vecteur_son = mean(vecteur_son')';
    end

    % calcul de la duree entre 2 beats consecutifs a partir du BPM
    periode = 60/bpm * fs;

    % desactivation des warnings
    % (on va manipuler des indices pas entiers, mais c'est fait expres)
    etat_warning = warning;
    warning off;

    % on se debrouille pour que le peigne contienne un nombre entier
    % de periodes
    longueur = round(periode*ceil(longueur_son/periode));

    % creation du peigne de dirac
    peigne = zeros(longueur,1);
    peigne(1:periode:longueur) = 1;

    % reactivation des warning a l'etat pre-desactivation
    warning(etat_warning);

    % on complete le vecteur son avec des zeros pour qu'il ait
    % la meme taille que le peigne et on en garde la valeur absolue
    % pour qu'il ressemble davantage a un peigne de pics = +1
    vecteur_son = [abs(vecteur_son)' zeros(longueur - longueur_son,1)']';
```

```
    % calcul de l'intercorrelation sur une periode, pas la peine d'aller
    % plus loin puisque c'est periodique...
    [c, lags] = xcorr(vecteur_son, peigne, ceil(periode), 'unbiased');

    % recuperation du max de cette intercorrelation
    [bof, decal] = max(c);

    % decalage du peigne
    peigne = circshift(peigne, lags(decal));

    % on coupe le peigne pour qu'il ait la meme longueur que le fichier
    % d'entree
    peigne = peigne(1 : longueur_son);
```

A.6. synchro_morceaux.m

```
function varargout = synchro_morceaux(varargin)
% SYNCHRO_MORCEAUX M-file for synchro_morceaux.fig
%      SYNCHRO_MORCEAUX, by itself, creates a new SYNCHRO_MORCEAUX or raises the existing
%      singleton*.
%
%      H = SYNCHRO_MORCEAUX returns the handle to a new SYNCHRO_MORCEAUX or the handle to
%      the existing singleton*.
%
%      SYNCHRO_MORCEAUX('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in SYNCHRO_MORCEAUX.M with the given input arguments.
%
%      SYNCHRO_MORCEAUX('Property','Value',...) creates a new SYNCHRO_MORCEAUX or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before synchro_morceaux_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to synchro_morceaux_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help synchro_morceaux

% Last Modified by GUIDE v2.5 09-Jun-2005 15:23:08

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @synchro_morceaux_OpeningFcn, ...
    'gui_OutputFcn',  @synchro_morceaux_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT
```

```
% --- Executes just before synchro_morceaux is made visible.
function synchro_morceaux_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to synchro_morceaux (see VARARGIN)

% Choose default command line output for synchro_morceaux
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes synchro_morceaux wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = synchro_morceaux_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in load1.
function load1_Callback(hObject, eventdata, handles)
% hObject    handle to load1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[FileName,PathName] = uigetfile('*.wav','Selectionner un fichier son');
if isequal(FileName,0) | isequal(PathName,0)
    return
end
try
    [son,fe] = wavread(fullfile(PathName, FileName));
catch
    return
end
disp(['Chargement du fichier ' FileName]);
handles.nom = FileName;
set(handles.fichier1, 'String', [FileName ' (' duree_texte(length(son)/fe) ')']);
set(handles.play1, 'Visible', 'on');
handles.son1 = son;
handles.fe1 = fe;
handles.temps1 = (0 : size(handles.son1,1) - 1)/handles.fe1;
plotter_son1(handles);
set(handles.load2, 'Visible', 'on');
set(handles.fichier2, 'Visible', 'on');
guidata(hObject, handles);

% --- Executes on button press in play1.
function play1_Callback(hObject, eventdata, handles)
% hObject    handle to play1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
sound(handles.son1, handles.fe1);

% --- Executes on button press in load2.
function load2_Callback(hObject, eventdata, handles)
% hObject    handle to load2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[FileName,PathName] = uigetfile('*.wav','Selectionner un fichier son');
if isequal(FileName,0) | isequal(PathName,0)
    return
end
try
    [son,fe] = wavread(fullfile(PathName, FileName));
catch
    return
end
disp(['Chargement du fichier ' FileName]);
handles.nom = FileName;
set(handles.fichier2, 'String', [FileName ' (' duree_texte(length(son)/fe) ')']);
set(handles.play2, 'Visible', 'on');
handles.son2 = son;
handles.fe2 = fe;
handles.temps2 = (0 : size(handles.son2,1) - 1)/handles.fe2;
plotter_son2(handles);
set(handles.text8, 'Visible', 'on');
set(handles.duree, 'Visible', 'on');
set(handles.text7, 'Visible', 'on');
set(handles.target_bpm, 'Visible', 'on');
set(handles.start, 'Visible', 'on');
guidata(hObject, handles);

% --- Executes on button press in play2.
function play2_Callback(hObject, eventdata, handles)
% hObject    handle to play2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
sound(handles.son2, handles. fe2);

% --- Executes during object creation, after setting all properties.
function target_bpm_CreateFcn(hObject, eventdata, handles)
% hObject    handle to target_bpm (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function target_bpm_Callback(hObject, eventdata, handles)
% hObject    handle to target_bpm (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of target_bpm as text
%        str2double(get(hObject,'String')) returns contents of target_bpm as a double


% --- Executes on button press in start.
```

```
function start_Callback(hObject, eventdata, handles)
% hObject    handle to start (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% lance le calcul
bpm = str2num(get(handles.target_bpm, 'String'));
duree = str2num(get(handles.duree, 'String'));

[handles.debut1, handles.fin1, handles.debut2, handles.fin2] ...
    = transition(handles.son1, handles.fe1, handles.son2, handles.fe2, ...
    bpm, duree );
set(handles.play3, 'Visible', 'on');
axes(handles.axes3);
temps = 0 : 1/handles.fe1 : (length(handles.debut2) - 1) / handles.fe1;
plot(temps, handles.fin1, 'r', temps, handles.debut2, 'b');
xlabel('temps');
ylabel('canaux␣superposes');
guidata(hObject, handles);
play3_Callback(hObject, eventdata, handles)

% --- Executes on button press in play3.
function play3_Callback(hObject, eventdata, handles)
% hObject    handle to play3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
duree = length(handles.debut2);

% Construction du fondu
if size(handles.debut2,2) == 1
    triangle_entrant = (0:1/(duree-1):1)';
else
    triangle_entrant(:,1) = (0:1/(duree-1):1)';
    triangle_entrant(:,2) = (0:1/(duree-1):1)';
end
triangle_sortant = ones(size(triangle_entrant)) - triangle_entrant;

mix = triangle_sortant.*handles.fin1 + triangle_entrant.*handles.debut2;

sound([handles.debut1' mix' handles.fin2'], handles.fe1);


function plotter_son1(handles)
axes(handles.axes1);
plot(handles.temps1, handles.son1);
ech = axis;
axis([ech(1:2) -1 1]);
grid on

function plotter_son2(handles)
axes(handles.axes2);
plot(handles.temps2, handles.son2);
ech = axis;
axis([ech(1:2) -1 1]);
grid on


% --- Executes during object creation, after setting all properties.
function duree_CreateFcn(hObject, eventdata, handles)
% hObject    handle to duree (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
```

```
% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function duree_Callback(hObject, eventdata, handles)
% hObject    handle to duree (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of duree as text
%        str2double(get(hObject,'String')) returns contents of duree as a double


% --- Executes during object creation, after setting all properties.
function edit2_CreateFcn(hObject, eventdata, handles)
% hObject    handle to duree (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function edit2_Callback(hObject, eventdata, handles)
% hObject    handle to duree (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of duree as text
%        str2double(get(hObject,'String')) returns contents of duree as a double


% --------------------------------------------------------------------
function Untitled_1_Callback(hObject, eventdata, handles)
% hObject    handle to Untitled_1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
delete(handles.fenetre);
```

A.7. transition.m

```
% Realise une transition entre deux morceaux
% Pierre Fritsch, 8 juin 2005
%
% [debut1, passage12, fin2] = transition(morceau1, morceau2, bpm, duree)
%
% Calcule le BPM des deux morceaux, ajuste leur BPM et les assemble
% sur une duree duree
```

```
function [debut1, fin1, debut2, fin2] = transition(morceau1, fs1, morceau2, fs2, bpm, duree)

t = cputime;

disp('TRAITEMENT DU MORCEAU DE DEPART');
% On calcule le BPM du morceau 1 sur sa fin
bpm1 = calcul_autocorrelation(morceau1(max(end - 5 * fs1, 1) : end, :), fs1);
% Ajustement de la vitesse des deux morceaux
nouveau1 = mettre_au_bpm(morceau1, fs1, bpm1, bpm);

% on cherche la position du dernier beat du morceau 1
% et on garde la duree qu'il reste apres
peigne1 = donner_peigne_superposition(nouveau1, bpm, fs1);
reste = 0;
while peigne1(end - reste) == 0
    reste = reste + 1;
    if reste == length(peigne1)
        disp('Duree trop courte; impossible de caler les morceaux');
        return;
    end
end

disp('');
disp('TRAITEMENT DU MORCEAU D''ARRIVEE');
% BPM du morceau 2 sur son debut
bpm2 = calcul_autocorrelation(morceau2(1 : min(end, 5 * fs2), :), fs2);
nouveau2 = mettre_au_bpm(morceau2, fs2, bpm2, bpm);
peigne2 = donner_peigne_superposition(nouveau2, bpm, fs2);

disp('coin');

% et l'endroit où celui va se caler sur le morceau 2
% (pres de duree apres le debut du morceau 2)
indice = min(duree * fs2, length(peigne2));
while peigne2(indice) == 0
    indice = indice - 1;
    if indice == 0
        disp('Duree trop courte ! Chansons impossibles a caler')
        return
    end
end

    % duree effective de la transition
    % pour que les deux chansons soient calees
    duree_effective = indice + reste;

    debut1 = nouveau1(1 : end - duree_effective);
    fin1 = nouveau1(end - duree_effective + 1 : end);

    debut2 = nouveau2(1 : duree_effective);
    fin2 = nouveau2(duree_effective + 1 : end);

    t = cputime - t;
    disp(['Duree effective de la transition : ' num2str(duree_effective/fs1)]);
    disp(['Il a fallu ' num2str(t) ' sec. CPU pour ce calcul']);
```

Annexe B.

Algorithme basé sur la mesure de l'énergie

Cet algorithme est emprunté à [Pat03]. Nous ne l'avons pas implémenté.

B.1. Principe

Tout signal sonore audible perçu par l'oreille humaine est converti en un signal électrique qui sera analysé par le cerveau. En pratrique, plus l'énergie transportée par le son est importante, plus le son va paraître fort. Ainsi un son sera perçu comme un beat seulement si son énergie instantanée est largement supérieure à l'énergie moyenne reçue par l'oreille aux instants précedents et suivants, c'est à dire si le cerveau détecte une variation brutale de l'énergie du son. Selon ce modèle on va calculer l'énergie moyenne du signal et la comparer avec l'énergie instantanée.

B.2. Première analyse

On note (a_n) et (b_n) les échantillons de l'extrait (stéréophonique) pris avec une période d'échantillonnage T_e. (a_n) représente la voie de gauche tandis que (b_n) représente celle de droite. L'énergie instantanée sera par exemple prise comme l'énergie contenue dans 1024 échantillons (ce qui représente environ 5 centièmes de seconde). L'énergie moyenne ne doit pas être calculée sur les chansons entières car celles-ci contiennent souvent des variations de rythme et surtout d'intensité. On peut calculer l'énergie moyenne sur 44032 échantillons ce qui représente approximativement 1 seconde de musique. Cela est rendu possible par la supposition que l'oreille humaine ne se souvient que d'une seconde de son pour la détection de beat.

L'algorithme peut être décrit de la manière suivante :

1. Calcul de l'énergie instantanée :

$$e = e_{stereo} = e_{droite} + e_{gauche} = \sum_{k=i_0}^{i_0+1024} a_k[k]^2 + b[k]^2$$

2. Calcul de l'énergie moyenne locale : le tampon dans lequel on va stocker les 44032 derniers échantillons va contenir 2 listes, correspondant aux 2 voies, c'est-à-dire $B[0]$ et $B[1]$:

$$< E >= \frac{1024}{44100} \sum_{i=0}^{44032} B[0][i]^2 + B[1][i]^2$$

3. Suite du procédé :
   - on deplace les éléments du tampon de 1024 indices vers la droite afin de faire de la place pour les 1024 échantillons arrivants ;
   - on place ces 1024 échantillons en haut du tampon ;
   - on compare e à $C \cdot < E >$ avec C constante, pour déterminer si l'on est ou non en présence d'un beat ;
   - si e est plus grand que $C \cdot < E >$, on a affaire à un beat.

   En général une bonne valeur pour la constante est $C \simeq 1,3$.

B.3. Quelques optimisations directes

On peut déjà améliorer l'algorithme simplement en gardant la valeur instantanée de l'énergie calculée sur 1024 échantillons plutôt que la valeur des 1024 échantillons. On passe ainsi de 44032 valeurs à 44 valeurs déjà calculées sur 1024 échantillons.

Il est possible de calculer l'énergie moyenne sur un temps supérieur à une seconde (mais il faut toujours rester sous les 3 secondes). On peut aussi reproduire l'algorithme plus souvent, par exemple tous les 512 échantillons et plus tous les 1024.

Au niveau algorithmique , ces améliorations se traduisent par :

$$< E >= \frac{1}{43} \cdot \sum_{i=0}^{43} E[i]$$

À chaque étape, on empile la nouvelle valeur de l'énergie instantanée e dans $E[0]$ et on supprime $E[43]$ en décalant toutes les valeurs vers la droite.

B.4. Sensibilité de la détection

Le principal problème de cet algorithme est le choix de la constante C.

Par exemple, si on a affaire a de la techno ou a du rap, deux types de musique dans lesquels les beats sont assez intenses, alors C doit être assez élevée, de l'ordre de 1,4. Si on a affaire à du rock, c'est à dire à un son beaucoup plus riche, les beats sont moins distincts et C doit prendre une valeur de l'ordre de 1,1 soit une valeur beaucoup plus faible.

C peut être calculée en calculant la variance de l'énergie contenue dans le tampon. La variance va quantifier le caractère plus ou moins affirmé des beats et nous permettre de calculer la valeur de C. Plus la variance est importante, plus l'algo doit être sensible et plus la valeur de C doit être faible. La variance peut se calculer avec la formule suivante :

$$V = \frac{1}{43} \cdot \sum_{i=0}^{43} (E[i] - < E >)^2$$

On peut choisir une relation linéaire liant C et V. Par exemple, quand V vaut 200, C vaut 1 ; et quand V vaut 25, alors C vaut 1,45. De là :

$$C = (-0.0025714 \cdot V) + 1.5142857$$

B.5. Efficacité de l'algorithme

L'algorihme avec ajustement de la constante C est très efficace sur le rap et la techno. La richesse des autres types de musique a souvent tendance à prendre ce programme en défaut.