Projet PPC - Concurrent UPMC LapTop Orchestra (CULTO)

Pour le projet de programmation concurrente, nous allons créer un évènement aux retombées mondiales, en formant le tout premier orchestre de laptop de l'UPMC. Le but de ce projet est donc dans un premier temps de comprendre les mécanismes internes de la programmation réactive. Pour cela nous allons créer nos propres objets réactifs, utilisable de manière transparente avec les objets internes donnés par PureData. Ainsi, nos objets seront capables de réagir à toutes sortes de signaux et d'effectuer des calculs sur ceux-ci. En utilisant des algorithmes de traitement de signal fournis par nos soins, chaque groupe pourra ainsi créer son propre instrument de musique expérimental, capable de générer des sons de haute-qualité (soit par synthèse directe, ou croisée entre des sources audios). Le projet se terminera par un concert donné par l'ensemble des groupes sous forme d'un orchestre.

Objectifs

1. Comprendre les mécanismes internes de la programmation réactive
2. Créer ses propres objets réactifs étendant un langage existant
3. Comprendre les principaux de signaux et réactivité
4. Utiliser des algorithmes de traitement de signal
5. Créer son propre instrument basé sur ces principes
6. Faire un concert avec l'ensemble des instruments du cours

1 Créer ses externals PureData

Bien que le langage PureData propose de très nombreuses fonctionnalités, certains comportement peuvent être très dur à implémenter même en utilisant des combinaisons de tous les objets primitifs fournis par le language (objets dits "*internals*"). Ainsi PureData permet d'être étendu de manière très simple en codant ses propres primitives (des objets dits "*externals*") depuis des languages classiques comme C/C++. Pour ce faire, PureData propose un ensemble d'include, mais il faut au préalable bien comprendre les mécanismes de la programmation réactive pour permettre de mettre en place un objet d'un environnement de programmation graphique réactive, capable de réagir à différents types de signaux.

De base le language est écrit en C, mais nous utiliserons ici le terme *classe* pour parler de la réalisation d'un concept combinant des données et la manipulation de ces données. Des *instances concrètes* de ces classes seront appelés *objets*. Pour vous faciliter la vie, nous fournissons en annexe une liste exhaustive de toutes les fonctions et types proposés par le SDK de PureData, ainsi que les propriétés du système de message.

Internal Un *internal* est une classe directement construite dans PureData. De nombreuses primitives du language comme `+`, `pack` ou `sig~` sont des internals.

External Un *external* est une classe qui n'est pas directement dans PureData mais qui est chargée au runtime. Une fois chargée dans la mémoire de PureData, les externals ne peuvent pas être distingués des internals.

1.1 Le premier external réactif *horloge*

D'habitude la première étape d'apprentissage d'un langage est le "hello world", mais vu qu'on cherche à étendre un language déjà complet (et votre niveau de *ouf malade*), nous allons commencer par créer un objet permettant d'obtenir l'heure système par un appel C (objet n'existant pas de base dans PureData). L'idée étant que notre objet doit envoyer l'heure sur son unique *outlet* à chaque fois qu'un message `bang` lui est envoyé.

1.1.1 Interface, classe et espace de données

L'interface vers PureData est fournie dans le fichier de header ‘‘`m_pd.h`’’ que vous pouvez donc importer au début de votre code. Pour arriver à construire des objets PureData, on utilise le type de ses classes (`t_class`) qui défini le type global, puis on utilise des `struct` C classiques, qui contiennent de manière minimale un champ de type `t_object` qui contient les propriétés de l'objet dans PureData. Les opérations minimales pour qu'un objet quelconque puisse être utilisé sont ensuite sa fonction d'initialisation (chargement en mémoire de l'objet) et de création (instantiation de la boîte). PureData vous simplifie l'étape de chargement en mémoire en cherchant de manière automatique la fonction `<nom-objet>_setup(void)` qui définit par la suite les opérations disponibles comme la création d'objet ainsi que les différents comportements en fonction de différents signaux reçus. Le squelette donné vous résume les fonctions à implémenter pour cet exercice.

```
1  #include "m_pd.h"
2
3  static t_class  *horloge_class;
4  typedef struct  _horloge
5  {
6    t_object      x_obj;
7    t_outlet      *h_out;
8  }
9  t_horloge;
10
11 // Q.4 - Comportement de l'objet en cas de message bang sur l'entrée principale (chaude)
12 void            horloge_bang(t_horloge *x);
13
14 // Q.3 - Création d'un nouvel objet horloge
15 void            *horloge_new(void);
16
17 // Q.2 - Chargement en mémoire des objets de type horloge
18 void            horloge_setup(void);
```

1.1.2 Chargement en mémoire

La fonction `horloge_setup` est la fonction permettant à PureData de prendre connaissance de votre objet et de charger sa signature en mémoire (vous permettant ensuite de créer plusieurs instances de cet objet). Elle va ainsi utiliser la gamme de fonction `class_*` décrites en annexes. Ecrivez la fonction `horloge_setup` qui crée une nouvelle classe (en utilisant la fonction `class_new`) et ajoute les comportements de l'objet (sa réactivité à différents types de messages). Notez bien que nous avons créé une variable globale (`static t_class *horloge_class`), c'est elle qui nous permet de conserver les informations de ce type d'objet dans PureData

1.1.3 Création des objets

La fonction `horloge_new` permet à PureData d'instancier un nouvel objet horloge (lorsque vous insérez une boîte horloge dans votre chemin réactif). Elle va ainsi construire un nouvel objet (en utilisant la fonction `pd_new` décrite en annexe), mais c'est également elle qui va se charger de construire les *inlets* et *outlets* (vous remarquez donc que ceux-ci sont créés directement lors de l'instanciation, grâce à la fonction `outlet_new`. Ecrivez la fonction `horloge_new` qui instancie une nouvelle classe et construit son *outlet*.

1.1.4 Fonctionnalités

La fonction `horloge_bang` indique à PureData les fonctions à réaliser lors de la réception d'un message de type `bang`. Il s'agit donc de la fonction qui sera appellée à chaque message. Vous devez donc récupérer l'heure grâce à un appel système C et écrire le résultat sur l'outlet de votre objet, en utilisant la gamme de fonctions `outlet_*` décrites en annexe.

1.1.5 Compilation et tests

La compilation est relativement simple mais nécessite quelques précautions, notamment sur le nommage des fichiers en sortie. Un exemple de ligne de compilation est donnée

```
gcc -I"<pdPath>/src" -Wall -W -g -ftree-vectorize -o horloge.o -c horloge.c
```

```
gcc -undefined dynamic_lookup -o horloge.pd_darwin horloge.o
```

Deux choses sont très importantes à noter. D'une part nous utilisons l'option du compilateur GNU `-undefined dynamic_lookup`, cette astuce nous permet d'enlever la nécessité d'avoir les objets PureData au linker statique, et permettra de faire ce link uniquement lors du chargement de l'objet. D'autre part, l'objet final se nomme `horloge.pd_darwin` car notre objet est compilé sur un Mac. Ce nommage permettra à PureData de retrouver l'objet external dans le répertoire courant en fonction de l'architecture. (Donc n'oubliez pas de compiler en `horloge.pd_linux` pour les archis Linux ou `horloge.dll` pour les archis Windows).

Pour tester votre objet, il suffit de créer un patch dans le même répertoire que votre objet, puis d'ajouter un Objet nommé "horloge" dans PureData.

1.2 Un deuxième external complexe *multipouet*

Nous allons maintenant monter en puissance en créant un objet plus complexe ayant plusieurs *inlets*, plusieurs *outlets*, permettant de travailler avec des listes, avec un nombre d'arguments variables à la construction de l'objet et permettant de réagir à *plusieurs types de signaux*. Un objet simulant bien cela est l'objet multipouet. Il s'agit en réalité d'un compteur entier ayant une valeur qu'on peut incrémenter et auquel on ajoute la possibilité de *reset* (remettre le compteur à sa valeur initiale), définir les bornes inférieures et supérieures (message *bound*) et de contrôler le pas "*step*" (nombre d'unités ajoutées à chaque incrémentation). Chaque message `bang` produit une liste constituée de N "pouet" sur l'outlet principal, et le dépassement de borne supérieure génére un message `bang` sur un 2ème outlet et remet le compteur à sa valeur initiale.

```
1  #include "m_pd.h"
2
3  static t_class  *multipouet_class;
4  typedef struct  _multipouet
5  {
6    t_object      x_obj;
7    t_int                 i_count;
8    t_float         step;
9    t_int                 i_min, i_max;
10   t_outlet       *p_out, *b_out;
11 }
12 t_multipouet;
13
14 // Q.4 - Comportement de l'objet en cas de message reset, set ou bound
15 void            multipouet_reset(t_multipouet *x);
16 void            multipouet_set(t_multipouet *x, t_floatarg f);
17 void            multipouet_bound(t_multipouet *x, t_floatarg min, t_floatarg max);
18
19 // Q.3 - Comportement de l'objet en cas de message bang sur l'entrée principale (chaude)
20 void            multipouet_bang(t_multipouet *x);
21
22 // Q.2 - Création d'un nouvel objet multipouet
23 void            *multipouet_new(t_symbol *s, int argc, t_atom *argv);
24
25 // Q.1 - Chargement en mémoire des objets de type multipouet
26 void            multipouet_setup(void);
```

1.2.1 Chargement en mémoire

Cette fois-ci nous voulons ajouter plusieurs argument optionnel à la création de l'objet, on pourra ainsi passer l'option `A_GIMME` qu'il faut passer à la création de la classe (`class_new`). Celui-ci indique que l'objet peut à sa création recevoir un ensemble de paramètres (comme créer une boite `osc~ 440` au lieu de `osc~`). Le deuxième mécanisme à mettre en place est d'informer le système de l'existance de nouvelles méthodes (qui réagiront à des messages dans PureData). On utilisera pour cela les fonctions `class_addmethod.`

1.2.2 Création de l'objet

Pour la création, vous noterez que plusieurs arguments sont passés à la fonction `multipouet_new`. En l'occurence le `t_symbol` nous donne le nom symbolique de l'objet. La suite est formée comme une fonction main C classique.

Ainsi, notre objet peut recevoir une liste quelconque de paramètres. Ici nous recevons un nombre `argc` d'arguments qui sont stockés dans la liste `argv` (le type `t_atom` est décrit dans les annexes). Ce tableau va nous permettre de remplir certains champs de la struct, pensez donc à bien gérer les cas ou 0, 1, 2 ou 3 arguments sont passés à la création de l'objet (`min, max, step`). Encore une fois, il faut également initialiser les inlets et outlets avec les fonctions `*outlet_new`. On ajoute enfin une dernière difficulté supplémentaire car on veut que le paramètre "*step*" puisse être modifié **directement** dans un 2ème inlet, et que le message *bound* arrive dans le 3ème inlet, on utilisera pour cela les fonctions `*inlet_new` et les autres messages seront gérés par l'inlet principal (donc pas besoin de paramètrage normalement).

1.2.3 Méthodes de l'objet

Cette fois-ci notre objet réagit à plusieurs signaux (messages) différents. Il faut donc coder le comportement réactif de l'objet en fonction de chaque type de messages

1. Le message `bang` est toujours géré par `multipouet_bang`, il incrémente le compteur, crée une liste de N "pouet" et sort cette liste sur le 1er outlet. Si le compteur dépasse la borne `max`, ce compteur est remis à la borne `min`, un message `bang` est émis sur le 2ème outlet et une liste de `min` "pouet" sur le 1er outlet
2. Le message `reset` est géré par `multipouet_reset`, il remet le compteur à la borne `min`.
3. Le messsage `set` est géré par `multipouet_set`, il met le compteur à la valeur indiquée en argument du message
4. Le message `bound` est géré par `multipouet_bound`, il remplace les bornes `min` et `max` par les valeurs données en argument du message.

1.3 Un external de signal *duck~*

Le *ducking* est une technique de mastering audio utilisé par les plus grands fumistes (comme David Guetta ou les DJs de camping), qui consiste à moduler le volume d'un signal entrant par celui d'un autre signal de contrôle. L'idée est que lorsqu'on commence à parler, le volume du micro diminue de manière proportionelle le volume de la musique. Ceci est également utilisé sur des synthétiseurs dont le volume est controlé par le volume des grosses caisses (l'effet *pompe* classique de la techno). Etant donné qu'on va maintenant monter en complexité, car cet objet va devoir gérer des entrées de signal continu, certaines nouvelles fonctions vont devoir s'ajouter à notre schéma classique d'externals. La structure de donnée reste relativement similaire, nous gardons cette fois-ci les deux *inlets* de signaux car nous allons travailler régulieurement avec ceux-ci.

```
1  #include "m_pd.h"
2
3  static t_class  *duck_tilde_class;
4  typedef struct  _duck_tilde
5  {
6    t_object              x_obj;
7    t_sample              f_pan;
8    t_sample              f;
9    t_inlet               *x_in2;
10   t_outlet              *x_out;
11 }
12 t_duck_tilde;
13
14 // Q.5 - Fonction centrale effectuant le calcul sur le signal
15 t_int           *duck_tilde_perform(t_int *w);
16
17 // Q.4 - Ajout de l'objet à l'arbre de traitement DSP
18 void             duck_tilde_dsp(t_duck_tilde *x, t_signal **sp);
19
20 // Q.3 - Libération de la mémoire de l'objet duck
21 void             duck_tilde_free(t_pan_tilde *x);
22
23 // Q.2 - Création d'un nouvel objet duck
24 void             *duck_tilde_new(void);
25
```

```
26 // Q.1 - Chargement en mémoire des objets de type duck
27 void             duck_tilde_setup(void);
```

1.3.1 Chargement en mémoire

De base, la méthode de chargement mémoire restera sensiblement identique (création de la classe avec `class_new` puis ajout des méthodes). Cependant, il faut cette fois-ci en plus de la méhtode de création `class_new` également indiquer une fonction de libération mémoire `class_new`, car nous allons travailler avec des buffers de signaux qui doivent être libérés proprement à la destruction de notre objet. Il faut également noter que la fonction de traitement principale n'est plus celle du message bang, mais de traitement des signaux `dsp` (Digital Sound Processing). On enregistrera donc cette methode sous le symbole `gensym(‘‘dsp’’)`. Enfin, il faut indiquer à PureData que cette classe traite des signaux dans son inlet principal, pour cela on utilisera la macro `CLASS_MAINSIGNALIN` (ceci permet d'utiliser l'inlet principal comme entrée de signal)

1.3.2 Création de l'objet

La création de l'objet se fait également sans trop de modification, on crée les inlets et outlets de la même manière qu'avant (`inlet_new` et `outlet_new`) en spécifiant bien le type des entrées/sorties comme étant de type `s_signal`.

1.3.3 Libération de l'objet

Il est important de savoir que même si dans le cas de la fonction `duck_tilde_new`, on ne change presque rien, PureData considère que la mémoire créée pour des inlets et outlets de type est gérée par l'utilisateur directement. Il faut donc s'occuper de libérer la mémoire soit même en utilisant les fonctions `inlet_free` et `outlet_free`.

1.3.4 Ajout à l'arbre de traitement DSP

La gestion du traitement de signal DSP dans PureData est gérée de manière particulière. Vous avez sûrement remarqué qu'au démarrage de l'application l'option DSP est désactivée (donc pas de traitement audio ne se fait). Ainsi lorsque le DSP est activé, PureData cherche dans son arbre de traitement les objets à activer. Il suffit donc dans cette fonction de prévenir PureData qu'une méthode perform devra être lancée pour traiter les signaux audio en utilisant la fonction `dsp_add` pour indiquer l'existence de la fonction `duck_tilde_perform`. On devra également indiquer les buffers qui seront nécessaires (expliqués dans la question suivante).

1.3.5 Méthodes de traitement du signal

Cette fois-ci notre objet reçoit des buffers de signaux et c'est la méthode qui permet d'effectuer le traitement que nous voulons sur les signaux entrants pour générer un signal sortant. Comme vous pouvez le noter, la fonction de traitement `duck_tilde_perform` prends étrangement en entrée un pointeur sur `t_int`. Il s'agit en fait d'une facilité (similaire à celle de `void*`) qui contient toutes les informations réunies dans un tableau. Il faut donc caster les différents objets, nous vous aidons en donnant les types des différentes entrées du tableau.

w[1] type `t_duck_tilde*` - Pointeur vers l'objet `duck~` lui-même

w[2] type `t_sample*` - Pointeur vers le buffer entrant dans le premier inlet

w[3] type `t_sample*` - Pointeur vers le buffer entrant dans le second inlet

w[4] type `t_sample`* - Pointeur vers le buffer de sortie (celui qui contiendra le résultat de notre calcul)

w[5] type `int` - Taille des buffers (tous sont alloués par PureData à la même taille)

Le but de l'objet est que cette fonction calcule le volume du signal dans le 2ème inlet et modifie en conséquence le volume du signal entrant dans le 1er inlet. Pour calculer le volume, on peut utiliser la formule simpliste (et à peine fausse) `moyenne(valeur_absolue(signal_2))`. On devra ensuite multiplier le 1er signal par `1-volume` et écrire le résultat dans le buffer de sortie. **Attention à la taille des buffers reçus par votre fonction**.

2 Calculer la transformée de Fourier

La transformée de Fourier est une opération mathématique permettant de calculer le spectre de fréquences d'un signal sonore (celui-ci représente l'ensemble des "hauteurs" trouvées dans un son, et donc sa "couleur"). Cet algorithme nous sera d'une utilité fondamentale lors de la construction des différents instruments spectraux. L'algorithme de transformée rapide de Fourier (FFT) peut déjà être visible dans PureData grâce à l'objet `fft~`. Cet algorithme faisant appel à de nombreuses connaissances mathématiques, nous vous fournissons la fonction de calcul, mais c'est à vous d'arriver à ré-adapter ce traitement en un external de signal PureData `myfft~`. **Il faudra faire notamment très attention aux problèmes de taille des buffers donnés par PureData.**

1. Faire un patch PureData utilisant `fft~`
2. Afficher le résultat de calcul de `fft~` (n'utilisant que *l'internal* de PureData)
3. Adapter l'algorithme de FFT fourni dans un external de type signal nommé `myfft~`
   (a) Pour vous aider, vous pouvez repartir de l'external `duck~` que nous avons défini à l'exercice précédent.
   (b) Modifiez l'external pour prendre un unique signal sur l'entrée principale.
   (c) Utilisez l'algorithme de FFT en C qui est fourni (fonction `rdft`) en annexe sur le site web. **Attention de bien lire les sections suivantes avant de vous lancer dans l'utilisation de cette fonction.**
   (d) Pour la compilation, vous pouvez pour le moment copier les fonctions dans le code
4. Comparer les résultats de `fft~` et `myfft~`

2.1 Gestion des tailles de buffers

Comme vous l'avez peut-être remarqué, PureData est configuré de manière native pour travailler sur des buffers de 64 points. Ainsi lorsque vos objets de signal sont appelés à travers la fonction `*_perform`, ils reçoivent des vecteurs de 64 points, ce qui ne correspond qu'à 1.5 milisecondes de son ! Autant cette taille est idoine pour travailler avec des objets ne traitant que des problèmes sur le signal lui-même, cette taille ne permet pas d'effectuer des calculs convenables pour des problèmes spectraux (comme la FFT). Plusieurs solutions s'offrent à vous pour gérer le problème.

1. Calculer la FFT sur des vecteurs de 64 points, *mais ceci implique de n'avoir accès qu'aux fréquences supérieures à celles de la voix d'un castra.*
2. PureData possède un réglage de taille des buffers qui pourrait permettre de forcer les buffers à utiliser des buffers d'une taille convenable pour la FFT (au moins 2048 points) *mais celui-ci a une tendance à rendre le logiciel instable et ne permet pas une bonne réactivité.*
3. Placez un objet `[block~]` n'importe ou dans votre patch en lui donnant pour argument 2048, ce qui force les signaux audios à être traités dans des buffers de 2048 échantillons, *mais cela implique de mettre cette objet dans tous les sous-patch de PureData ce qui réduit sa modularité.*
4. La meilleure solution consiste à gérer les buffers soit-même. L'idée est donc d'allouer un buffer de la taille voulue (4096 ou 8192 points) à la création de l'objet (`*_new`) puis de gérer l'arrivée de nouvelle données (lors de l'appel à la méthode `*_perform`) sous forme de buffers circulaires. Ainsi l'arrivée d'un nouveau vecteur de données décale l'ensemble des données existantes dans le buffer de 64 points et ajoute les nouvelles données dans les 64 dernières positions du buffer. Si assez de données sont présentes, on lance alors le calcul de FFT. **Attention dans ce cas il faudra également gérer la libération de la mémoire associée au buffer.**

2.2 Application d'une fenêtre

Pour obtenir de meilleurs résultats, la FFT requiert d'être calculée sur des données dites fenêtrées. Le fenêtrage est une opération assez simple qui consiste à multiplier le vecteur d'entrée (à transformer) par une "fenêtre" (vous pouvez voir que cette opération est faite pour l'objet internal `fft~` de PureData dans le patch qui vous est fourni). Il suffit donc de générer un vecteur de la même taille que notre buffer, puis de multiplier point par point les deux vecteurs. Les deux types de fenêtres les plus efficaces pour la FFT sont celles de Hamming et Blackman, dont les formules sont données par

$$Hamming\,(t) = \begin{cases} 0.54 - 0.46cos\left(2\pi\frac{t}{T}\right) & \text{si } t \in [0,T] \\ 0 & \text{sinon} \end{cases}$$

$$Blackman\,(t) = \begin{cases} 0.42 - 0.5cos\left(2\pi\frac{t}{T}\right) + 0.08cos\left(4\pi\frac{t}{T}\right) & \text{si } t \in [0,T] \\ 0 & \text{sinon} \end{cases}$$

2.3 Utilisation de la fonction `rdft`

Nous vous fournissons la fonction rdft permettant de calculer la transformée de Fourier rapide basé sur des nombres Réels et Discrets (RDft).

```
void rdft(int n, int isgn, float *a, int *ip, float *w)
```

n — Taille des vecteurs à transformer

isgn — Flag permettant d'effectuer une FFT normale (1) ou inverse (-1)

a — Vecteur de données à transformer. **Ce vecteur contient les données en entrée, mais contiendra aussi le résultat de la FFT après application de la fonction (réécriture des données au même endroit)**

ip — Vecteur de bitshuffling du taille n*2 **qui doit être initialisé grâce à la fonction** `init_rdft`

w — Vecteur de pondération de taille n*2 **qui doit être initialisé grâce à la fonction** `init_rdft`

3 Construire son instrument réactif

Nous allons maintenant passer au projet en soit, la création d'un external représentant son propre instrument de musique réactif. Vous devez choisir un des modules parmis les 3 proposés, effectuer son implémentation et tester ses possibilités musicales. Nous sommes évidemment ouvert à toute proposition intéressante de nouvel external (ou amélioration, modification ou extension d'un external existant) n'entrant pas nécessairement dans l'une des catégories que nous vous proposons. De manière générale, toutes les méthodes que nous vous proposons rentrent dans la classe des *synthèses croisées*. L'idée de ce type de synthèse est d'utiliser une source sonore qui contient l'information de hauteur (notes) et de la moduler (croiser) par une source spectrale qui lui donne sa couleur (timbre). Ainsi on découple

- D'un côté la source dite *harmoniquement riche*. Celle-ci définit la mélodie et son évolution et va typiquement être jouée par un musicien pour définir la musique
- D'un autre côté la source dite *modulatrice*. Celle-ci va définir la modification à appliquer à la source harmonique. Ainsi, cette source est indispensable pour définir la "couleur" du son (généralement on utilise une source vocale à cet effet)

Cette approche permet donc de séparer l'aspect musical (mélodies) de la couleur des sons. Il sera donc possible pour les musiciens de jouer des mélodies, tout en assurant que les non-musiciens aient une large influence sur le rendu de la texture sonore et donc que tout le monde puisse participer au concert du laptop orchestra.

3.1 Mise en place des objets

Tout comme les objets de signal précédents, un ensemble de considérations doivent être prises. Ainsi toutes les méthodes de synthèse croisées utilise fortement la FFT. Il vous faudra donc gérer les mêmes problèmes que pour l'exercice précédent.

1. Implémenter sa propre gestion des buffers, pour permettre de travailler sur des vecteurs d'une taille suffisante.
2. Gestion d'une taille de vecteurs variable (pouvant être mise en tant que paramètre de l'objet)
3. Gestion mémoire (alloc et free) pour tous les buffers qui seraient nécessaires en chemin
4. Application d'une fenêtre sur les différents signaux sur lesquels on doit calculer une FFT

Comme à ce stade du projet, vous devriez avoir déjà codé deux externals de signal et toutes les fonctions correspondantes, nous ne reviendrons plus sur la gestion des différentes fonctions de mise en place (`*_setup, *_new`, etc...). Comme il s'agit de problèmes de traitement de signal, nous vous aidons grandement en vous fournissant le pseudo-code de la fonction de traitement (`*_perform`) pour chaque type de synthèse.

3.2 Synthèse croisée spectrale

L'idée de la synthèse croisée est de filtrer la première entrée en utilisant la seconde pour créer un effet de synthèse spectrale croisée. Pour cela les valeurs de spectre sont calculées par FFT, puis on utilise la distribution de la première source que l'on multiplie par celle de la deuxième. On peut utiliser un seuil pour déterminer si il faut effectuer la multiplication spectrale à chaque instant ou plutôt maintenir le dernier vecteur de calcul pour intensifier l'effet perçu de la modulation.

3.2.1 Structure de l'objet

L'objet final permettant d'effectuer la synthèse croisée spectrale une fois compilé sera constitué de 4 inlets (de gauche à droite). L'explication complète de l'effet des différents paramètres est donnée dans les sections suivantes.

1. Un inlet de signal permettant de récupérer la source harmoniquement riche
2. Un inlet de signal recevant la source modulatrice
3. Un inlet flottant permettant de gérer le seuil de synthèse (intensifiant l'effet)
4. Un inlet de messages permettant de gérer les messages
   (a) *autonorm* [0/1] permet d'activer une normalisation pour diminuer l'amplitude des pics
   (b) *bypass* [0/1] permet de désactiver le croisement (on entend la source riche uniquement).

3.2.2 Algorithme de traitement

Nous fournissons dans l'algorithme suivant le pseudo-code permettant d'effectuer la synthèse croisée spectrale.

```
1  Si (bypass)
2          Recopier l'in1␣dans␣l'out et sortir
3  Gestion des différents buffers de signal pour in1 et in2
4  Duplication des buffers pour future application de FFT
5  Appliquer la fenêtre aux buffers dupliqués (dup1 et dup2)
6  Application de la fonction rdft aux deux buffers
7  Si (autonorm)
8          ingain = 0;
9          Pour (i = 0; i < tailleFFT; i += 2)
10                 ingain = distance_euclidienne(dup1[i], dup1[i + 1])
11         Fin Pour
12 Fin Si
13 Pour (i = 0; i < tailleFFT; i += 2)
14         a1 = dup1[i]; b1 = dup1[i+1];
15         a2 = dup2[i]; b2 = dup2[i+1];
16         curGain = distance_euclidienne(a2, b2);
17         Si (curGain > seuilSynthese)
18                 tmp1[i] = distance_euclidienne(a1, b1) * curGain;
19         Fin Si
20         tmp1[i + 1] = -atan2(b1, a1);
21         tmp2[i] = tmp1[i] * cos(tmp1[i + 1]);
22         tmp2[i + 1] = -tmp1[i] * sin(tmp1[i + 1]);
23 Fin Pour
24 Si (autonorm)
25         outgain = 0
26         Pour (i = 0; i < tailleFFT; i += 2)
27                 outgain = distance_euclidienne(tmp2[i], tmp2[i + 1])
28         Fin Pour
29         finalGain = ingain / outgain
```

```
Sinon
        outgain = 1;
Fin Si
Effectuer la FFT inverse sur tmp2
Normaliser le resultat par finalGain
Ecrire dans le vecteur d'output
```

3.3 Synthèse croisée par convolution

Un autre technique classique est d'utiliser une opération dite de *"convolution par bloc"*. L'idée est d'effectuer la multiplication complexe des spectres des deux signaux d'entrée. La multiplication des spectres peut causer d'importantes baisses dans l'amplitude générale du signal de sortie. Inversement, il est possible d'utiliser la division complexe du spectre entrant à la place de la multiplication, ceci aura pour effet de multiplier grandement les amplitudes.

3.3.1 Propriétés de l'objet

L'objet final permettant d'effectuer la synthèse croisée par convolution une fois compilé sera constitué de 4 inlets (de gauche à droite). L'explication complète de l'effet des différents paramètres est donnée dans les sections suivantes.

1. Un inlet de signal permettant de récupérer la source harmoniquement riche
2. Un inlet de signal recevant la source modulatrice
3. Un inlet flottant permettant de gérer l'exposant d'échelle (diminuant l'effet)
4. Un inlet flottant permettant de gérer le seuil d'inversion (diminuant l'effet)
5. Un inlet de messages permettant de gérer les messages
   (a) *invert* [0/1] permet d'utiliser soit une multiplication soit une division complexe.
   (b) *bypass* [0/1] permet de désactiver le croisement (on entend la source riche uniquement).

3.3.2 Algorithme de traitement

Nous fournissons dans l'algorithme suivant le pseudo-code permettant d'effectuer la synthèse croisée par convolution.

```
Si (bypass)
        Recopier l'in1␣dans␣l'out et sortir
Gestion des différents buffers de signal pour in1 et in2
Duplication des buffers pour future application de FFT
Appliquer la fenêtre aux buffers dupliqués (dup1 et dup2)
Application de la fonction rdft aux deux buffers
// Conversion des valeurs complexes en coordonnées polaires
Pour (i = 0; i < tailleFFT; i += 2)
        a1 = dup1[i]; b1 = dup1[i+1];
        a2 = dup2[i]; b2 = dup2[i+1];
        Si (invert)
                mag_1 = distance_euclidienne(a1, b1);
                mag_2 = distance_euclidienne(a2, b2);
                tmp[i] = (mag2 > seuil ?
                        mag_1 / mag_2 : mag_1 / threshold);
                tmp[i + 1] = ((mag_1 != 0 && mag_2 != 0) ?
                        atan2(b2, a2) - atan2(b1, a1) : 0.);
        Sinon
                f_real = (a1 * a2) - (b1 * b2);
                f_imag = (a1 * b2) + (b1 * a2);
                tmp[i] = distance_euclidienne(f_real, f_imag);
                tmp[i + 1] = -atan2(f_imag, f_real);
        Fin Si
        tmp[i] = pow(tmp[i], exponent);
```

```
Fin Pour
// Reconvertir en valeurs complexes
Pour (i = 0; i < tailleFFT; i += 2)
        resultat[i] = tmp[i] * cos(tmp[i + 1]);
        resultat[i+1] = -tmp[i] * sin(tmp[i + 1]);
Fin Pour
Effectuer la FFT inverse sur resultat
Eventuellement normaliser le resultat
Ecrire dans le vecteur d'output
```

3.4 Synthèse croisée par shaping

L'idée de cette synthèse consiste à retravailler la forme (*shaping*) de l'évolution fréquentielle d'un signal par celle d'un autre. Pour cela on analyse l'envelope de la source modulatrice qu'on applique à la première source (harmonique). Le contrôle de la largeur de shaping permettra de contrôler l'intensité de cet effet.

3.4.1 Propriétés de l'objet

L'objet final permettant d'effectuer la synthèse croisée par shaping une fois compilé sera constitué de 4 inlets (de gauche à droite). L'explication complète de l'effet des différents paramètres est donnée.

1. Un inlet de signal permettant de récupérer la source harmoniquement riche
2. Un inlet de signal recevant la source modulatrice
3. Un inlet flottant permettant de gérer la largeur du shaping (*shapeWidth* intensifiant l'effet)
4. Un inlet de messages permettant de gérer les messages
   (a) *autonorm* [0/1] permet d'activer une normalisation de la forme temporelle
   (b) *bypass* [0/1] permet de désactiver le croisement (on entend la source riche uniquement).

3.4.2 Algorithme de traitement

Nous fournissons dans l'algorithme suivant le pseudo-code permettant d'effectuer la synthèse croisée par shaping.

```
Si (bypass)
        Recopier l'in1␣dans␣l'out et sortir
Gestion des différents buffers de signal pour in1 et in2
Duplication des buffers pour future application de FFT
Appliquer la fenêtre aux buffers dupliqués (dup1 et dup2)
Application de la fonction rdft aux deux buffers
// Conversion des valeurs complexes en coordonnées polaires
Pour (i = 0; i < tailleFFT; i += 2)
        a1 = dup1[i]; b1 = dup1[i+1];
        a2 = dup2[i]; b2 = dup2[i+1];
        tmp1[i] = distance_euclidienne(a1, b1);
        tmp1[i + 1] = -atan2(b1, a1);
        tmp2[i] = distance_euclidienne(a2, b2);
        tmp2[i + 1] = -atan2(b2, a2);
Fin Pour
Pour (i = 0; i < tailleFFT; i += shapeWidth * 2)
        ampSum = 0; freqSum = 0;
        Pour (j = 0; j < shapeWidth * 2; j += 2)
                ampSum += tmp2[i + j];
                freqSum += tmp1[i + j];
        Fin Pour
        factor = ampSum / freqSum;
        Pour (j = 0; j < shapeWidth * 2; j += 2)
                tmp1[i + j] *= factor
        Fin Pour
```

```
26 // Reconvertir en valeurs complexes
27 Pour (i = 0; i < tailleFFT; i += 2)
28         resultat[i] = tmp1[i] * cos(tmp1[i + 1]);
29         resultat[i+1] = -tmp1[i] * sin(tmp1[i + 1]);
30 Fin Pour
31 Effectuer la FFT inverse sur resultat
32 Eventuellement normaliser le resultat
33 Ecrire dans le vecteur d'output
```