Chapitre 1

Utilisation de jMax

1.1 Configuration de l'environnement

1.1.1 Environnement Unix

L'interface graphique de jMax est gérée par Java, il est donc nécessaire de rajouter à son chemin de recherche des exécutables, représenté par la variable d'environnement unix **PATH**, le chemin d'accès aux exécutables java.
Par exemple, dans le cas où l'on utilise le C-Shell et où le répertoire /usr/java/bin contient les exécutables java, ou des liens vers ceux-ci:

```
setenv PATH=$PATH:/usr/java/bin
```

1.1.2 Environnement jMax

Ensuite il est utile de créer un fichier de configuration de jMax, nommé .jmaxrc. Ce fichier est recherché lors de chaque lancement de jMax. Celui-ci doit être placé à la racine de votre répertoire. Si jMax ne peux l'ouvrir en lecture, il utilisera la configuration par défaut livré avec la distribution.
Dans ce fichier, il est possible de positionner la majorité des variables d'environnement de jMax.

Exemple de fichier .jmaxrc :

```
defineHost local o2r10ki63
defineHost bonang.ircam.fr o2r10ki63

### start in debug mode
if {[systemProperty "debug"] == "true"}
 {
   set jmaxMode "debug"
   set jmaxConnection "udpclient"
```

```
   set jmaxPort 2000
 }
elseif {[systemProperty "pure"] == "true"}
 {
   set jmaxMode "debug"
   set jmaxConnection "udp"
   set jmaxServerName fts.pure
 }
elseif {[systemProperty "socket"] == "true"}
 {
   set jmaxMode "opt"
   set jmaxConnection "udpclient"
   set jmaxPort 2000
 }
else
 {
   set jmaxConnection "udp"
 }
#set jmaxSplashScreen "hide"
set jmaxFastFileBox "true"
set jmaxPkgPath /u/formes/tisseran/Trompette/max/

### startup  actions
when start
 {
   puts "root dir: $jmaxRootDir"
   ucs set fonts 8 9 10 12 14 16 18 20

   jmaxSetSampleRate 44100
   jmaxSetAudioBuffer 1024

   # audio config
   if {$jmaxHost == "bonang.ircam.fr"}
   {
     openDefaultAudioIn analogIn
     openDefaultAudioOut analogOut
     openDefaultMidi midi2
 }
elseif {$jmaxHost == "local"}
 {
     openDefaultAudioIn analogIn
```

```
     openDefaultAudioOut analogOut
     openDefaultMidi midi2
 }
else
 {
puts "NO AUDIO CONFIG!"
 }
puts "startup done! \n"
 }
### exit actions
when exit
 {
   puts "bye, bye!"
 }
```

Remarque 1:

Il est toujours possible de repositionner les variables d'environnement par le biais de commande saisie à l'intérieur de la console. Le langage de commande utilisé est proche du tcl.

Remarque 2:

Le nom du port MIDI utilisé comme port MIDI par défaut ne correspond pas au nom de l'interface MIDI créée par la commande Unix:
`startmidi [-n name] [-s speed] [-p proto]  -d /dev/tty{d,f}x`
Par défaut, il existe des correspondances entre les noms de port MIDI sous jMax et les noms des interfaces sous le système d'exploitation (Figure 1.1):

Il faut remarquer qu'il n'est pas possible pour l'utilisateur de modifier ces correspondances. Il est cependant possible de définir de nouvelles correspondances entre port MIDI jMax et interface MIDI Unix, dans le fichier de configuration. Dans l'exemple qui suit, nous définissons le port MIDI "midi_tutorial" dans jMax comme un lien vers l'interface midi "Tutorial" lancé par la commande "startmidi" du système d'exploitation (IRIX63). Après cette définition, nous configurons jMax pour utiliser ce port MIDI par défaut :

```
defineDevice midi_tutorial { sgi_midi port Tutorial } " Midi on startmidi port
Tutorial "
#On rajoute les lignes suivantes dans la boucle when start { } pour preciser
#quel port midi utiliser par defaut
openDefaultAudioIn analogIn
openDefaultAudioOut analogOut
openDefaultMidi midi_tutorial
```

host types / MIDI devices	**o2r5k**	**o2r5ki63**	**o2r10k**	**o2r10ki63**	**octane**	**origin**	**linuxpc**
midi	Serial Port 1	Serial Port 1	Serial Port 1	Serial Port 1	Serial Port 1	Serial Port 1	
midi1	Serial Port 1	Serial Port 1	Serial Port 1	Serial Port 1	Serial Port 1	Serial Port 1	
midi2	Serial Port 2	Serial Port 2	Serial Port 2	Serial Port 2	Serial Port 2	Serial Port 2	

FIG. 1.1 – *Correspondance entre port MIDI jMax et interface MIDI du système d'exploitation*

En ajoutant ces commandes dans le fichier de configuration .jmaxrc, la console affiche les lignes suivantes après le lancement de jMax (Figure 1.2):

```
Opening default audio in: Analog In
Opening default audio out: Analog Out
Opening default midi port:  Midi on startmidi port Tutorial
startup done!
```

FIG. 1.2 – *Affichage sur console*

Rappel : Les communications entre les objets peuvent être de deux types:

- Type "contrôle" : la communication s'effectue par valeurs envoyées de la sortie de l'émetteur vers l'entrée du récepteur. Les valeurs peuvent être de types entières, rèelles, des symboles et des ensembles de ces valeurs.
- Type "signal" : l'émetteur envoie au récepteur un paquet d'échantillons (la taille de ce paquet est généralement réprésentée par la variable **n_tick**). On peut considérer ce type de transmissions comme une communication de type contrôle dont la valeur envoyée est un tableau de réels de taille n_tick (cette variable a généralement pour valeur 64).

Il faut noter qu'il n'est pas possible, en pratique, de simuler une liaison de type "signal" par une de type "contrôle". En effet la fréquence de prise en compte des signaux de contrôle est beaucoup trop faible pour pouvoir asssumer une telle comparaison.

1.1.3 Lancement de jMax

On lance jmax en utilisant la ligne de commande suivante:

```
jmax -new true
```

Remarque1 :
On suppose que la variable PATH contient les chemins d'accès aux exécutables Java et qu'il existe un alias jmax pointant sur l'exécutable jmax de la distribution (dans le cas de bonang : `/usr/LOCAL/max/bin/jmax`).

Remarque2 :
L'option `-new true` permet l'utilisation de la nouvelle interface graphique, cette option ne semble pas nécessaire pour la version 2.4.5 de jMax.

1.1.4 Lancement de jMax sur deux machines

Il est possible de faire tourner l'interface graphique sur une machine et le moteur de calcul sur une autre. Pour utiliser cette fonctionnalité il faut lancer jMax avec la ligne de commande suivante:

```
(marimba) jmax -jmaxHost marimba.ircam.fr -jmaxConnection udpclient -jmaxPort
20023 -new true
(bonang) /usr/LOCAL/max/fts/bin/irix63r10k/opt/fts udp
marimba.ircam.fr:20023
```

Dans cet exemple, l'interface graphique tourne sur marimba, FTS sur bonang et la communication se fait par le port TCP/IP 20023.

1.2 Interface graphique

Il est important d'avoir connaissance de quelques contraintes liées à l'interface graphique.

1.2.1 Contraintes de l'interface graphique

Visualisation des liens

Il faut noter qu'il n'y a pas de différence graphique entre des liens "contrôle" et des liens "signal". Il est évident que ceci peut compliquer la compréhension du patch. Cependant on ne peut pas faire d'erreur en reliant un lien de type "contrôle" à une entrée signal (au vis versa), car les types des inlets sont définis lors des instantiations des classes des objets utilisés.

Plusieurs liens sur un outlet

Lorsque l'on ajoute un outlet à un objet, jMax le place à droite des précédents. Dans le cas des objets où on ne visualise qu'un inlet, par exemple le bang ou le slider, il faut savoir que les outlets sont superposés. Ainsi la priorité ira du lien le plus récent au plus ancien.

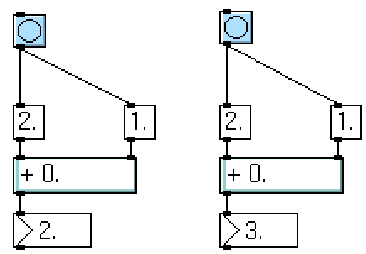


FIG. 1.3 – *Exemple illustrant le problème de la surcharge d'un outlet*

L'objet "+" effectue l'opération d'addition lors de la modification d'état de l'inlet gauche. Les opérandes sont les valeurs reçues par les inlets. Au cas où l'objet n'a pas encore reçu de valeur sur l'inlet de droite, il utilise son argument comme opérande complémentaire à celui reçu par l'entrée de gauche pour effectuer l'addition.

Cet exemple (Figure A.1) nous montre bien le problème lié à l'utilisation de plusieurs liens sur un même outlet. Il n'y a pas d'outil graphique permettant de visualiser la priorité entre les lines, de ce fait deux patches semblables, graphiquement, ne donnent pas le même résultat. Nous pouvons en conclure qu'il n'est pas possible de prévoir le comportement d'un patch contenant une surcharge d'outlet.

Ordre de prise en compte des entrées et des sorties

La prise en compte des entrées (inlets), ainsi que celle des sorties (outlets), se fait de droite à gauche. Il ne faut pas négliger cette règle lors de l'élaboration de patch. En effet le résultat pourrait être très différent de celui attendu, selon l'ordonnancement des objets.

L'objet "t b b" envoie un bang sur chacune de ses sorties après réception d'un bang à son entrée.

L'exemple de la figure 1.4 nous permet de voir une application de la priorité entre les entrées (sorties). On constate que pour l'addition de droite,

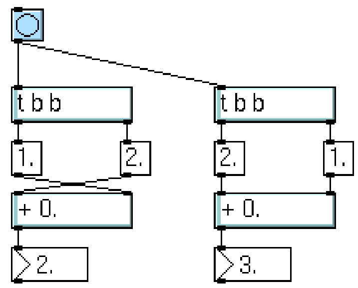


FIG. 1.4 – *Exemple illustrant la règle de priorité entre outlet et inlet*

selon la règle de priorité énoncée précédemment, la première valeur envoyée à l'objet est la valeur **1.** . Celle-ci est reçue par l'inlet de droite donc l'addition n'est pas effectuée. Lors de la réception de **2.** sur l'inlet de gauche, on additionne les deux opérandes ce qui nous donne comme résultat **3.** .
Pour l'addition de gauche, la première valeur reçue est la valeur **2.** sur l'inlet de gauche. L'addition est alors effectuée, d'où le résultat **2.** .

Charge machine apportée par l'utilisation de Java

Il est important de noter que plus le patch comporte de données à rafraîchir régulièrement, plus la partie de la charge de la machine dûe à Java sera importante. Il faut faire particulièrement attention aux sliders de grande taille et aux boîtes d'affichage de valeurs. Il faut bien sur faire un choix entre l'ergonomie d'utilisation et la charge induite par le patch.

Le nombre de valeurs possibles sur un slider est limité par sa taille. Celle-ci est modifiable par *edit, inspect* (il faut être en mode édition (CTRL+e)).

Il est possible de modifier la période de remise à jour des variables graphiques à l'aide de la commande saisie dans la console jMax ou dans un fichier de configuration (.jmaxrc, .jpk ou .env):

```
ucs set update_period <int>
```

Cette variable est positionnée par défaut à 10, l'unité étant la milliseconde.

1.2.2 Sous-patches et templates

On peut utiliser des sous patches pour faciliter la lisibilité du patch. Ceux-ci permettent de mettre différents objets dans une autre fenêtre et de relier l'ensemble avec le patch principal. Il existe un objet graphique permettant la création graphique de sous-patches (Figure 1.5):

FIG. 1.5 – *Objet graphique représentant un sous-patch*

Il est nécessaire de passer en mode exécution afin de pouvoir visualiser le contenu du sous-patch. On y accède en double-cliquant dessus. On peut maintenant (en repassant en mode édition) créer, ou modifier, le contenu de notre sous-patch. Les entrées et sorties sont représentées par des objets de forme carrée contenant un triangle (la pointe dirigée vers le haut dans le cas d'une sortie). Un sous-patch peut comporter les mêmes objets que le patch principal.

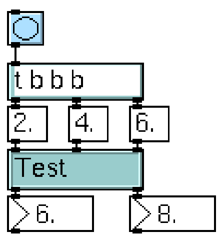


FIG. 1.6 – *Patch principal*

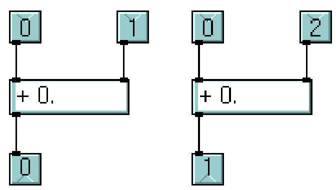


FIG. 1.7 – *Contenu du sous patch "Test"*

FIG. 1.8 – *Exemple de patch et sous patch*

On peut remarquer qu' à chaque ajout d'un objet représentant une entrée (sortie), le nombre spécifiant l'entrée (sortie) est incrémenté de 1. En effectuant un copier-coller de celui-ci, on obtient plusieurs entrées (sortie) de même numéro. On peut se servir de ce fait, pour éviter de tendre des liens de part et d'autre du patch. Un sous-patch permet ainsi de gagner en lisibilité mais malheureusement il n'est pas possible d'affecter une aide au sous-patch. Ce qui est un incovénient non négligeable dans le cas d'élaboration de sous-patch assez complexe.

On peut remédier à ceci en utilisant un template: on crée le sous-patch, comme décrit précedemment, puis on effectue un **Save To... (menu File)**. Ensuite il suffit d'indiquer à **jMax** le (ou les) chemin(s) d'accès du (des) répertoire(s) contenant les templates, on utilise pour cela la commande :

`templateDirectory chemin_acces`

Cette commande peut être saisie dans le fichier de configuration de jmax (.jmaxrc), le fichier de configuration du package (.jpk), le fichier de configuration du patch (.env) et dans la console jMax.

Lorsque l'on veut utiliser les templates, il suffit de placer une “boîte” de type objet contenant le nom du templates. En double-cliquant dessus, on visualise le contenu de ceux-ci.

Remarque :
: On ajoute généralement la ligne de commande précédente au fichier de description du package. On peut aussi l'utiliser dans les fichiers de type .env. Dans ce cas la définition est locale aux patchs qui utilisent ce fichier (voir plus loin des précisions)

Différence entre sous patch et template:
: Comme il est précisé précédemment, il est possible de rajouter de l'aide aux templates. De plus, il faut noter que la définition d'un sous-patch est locale au patch principal contrairement à celle d'un template. Si l'on utilise des sous-patches, l'utilisateur aura seulement besoin du fichier jMax contenant le patch.

1.2.3 Ajouter de l'aide aux patches sous jMax

On crée pour cela un patch d'aide, qui comprend tout ce que l'on juge utile de préciser. On sauvegarde sous le format .jmax, ainsi il est possible d'exécuter ce patch et de créer de cette façon des exemples de comportement (utile pour les objets de type contrôle). Notre fichier d'aide étant créé, il faut maintenant le rattacher à l'objet (ou le template) auquel il se rapporte. Pour cela on édite le fichier du package contenant l'objet (ou le template) et on ajoute à celui-ci la ligne de commande suivante:

```
helpPatch nom_objet chemin_acces_au_patch_aide
```

L'aide définie de cette manière est consultable en cliquant avec le bouton du milieu sur l'objet. Il est aussi possible de rajouter un lien dans le menu help, situé à droite de la fenêtre d'édition, en utilisant la ligne de commande suivante (toujours ajoutée au fichier de déclaration du package):

```
helpSummary nom_du_sommaire_dans_le_menu_help chemin_acces_au_patch_sommaire
```

1.3 Fichier de configuration des patches

1.3.1 Description des fichiers .jpk

Ce sont des fichiers de langage tcl/tk comportant les spécifications des packages. Ils comportent la liste des modules à charger en mémoire ainsi que les déclaration des aides des objets contenus dans ce package. On précise aussi le répertoire contenant les templates.

Remarque :
: Il est possible d'effectuer les mêmes commandes dans ces fichiers que

celles disponibles partir de la console de jMax. On peut ainsi repositionner des variables afin de permettre une bonne exécution des packages.

Exemple de fichier .jpk: trompette.jpk

```
# package declaration
package provide trompette

#On affecte le taux d'echantillonnage a 32000.
jmaxSetSampleRate 32000

# Modules a charger en memoire
global jmaxArch jmaxMode
ucs load module trompette $dir/c/lib/$jmaxArch/$jmaxMode/libtrompette.so

# Repertoire contenant les templates
templateDirectory $dir/templates

# help
# REMARQUE
# Les helpPatch sont appelles
# lorsque l'on clique avec le bouton du milieu sur l'objet
# Les helpSummary sont des elements du menu help
helpPatch tromp_param $dir/help/tromp_param.help.jmax
helpPatch tromp~ $dir/help/tromp.help.jmax
helpSummary Inlet&Outlet_tromp_param
            $dir/help/tromp_param.description_inlet_outlet.jmax
helpSummary Description_panneau_controle
            $dir/help/tromp.description_panneau_controle.jmax

puts "package: trompette" #Affichage sur la console
```

1.3.2 Fichiers .env

Lors de l'ouverture d'un fichier *file.jmax*, jMax recherche la présence d'un fichier de nom *file.env* ou ,dans le cas où celui-ci n'existe pas, de nom *project.env*. Si aucun de ces deux fichiers existent, les packages utilisés seront les packages définit dans le fichier `$JMAXDIR/config/packages.tcl` (`$JMAXDIR` étant le répertoire contenant jMax). Les fichiers .env sont des fichiers d'environnement, ils précisent quels sont les packages utilisés par le patch. Contrairement au fichier de spécification des packages, ils ne permettent pas de repositionner des variables d'environnement de jMax.

Exemple de fichier .env

```
package require trompette
package require graphique
```

Remarque:

> On peut constater que les packages sont appellés par leurs noms. On ne précise pas de chemin d'accès. En fait, les chemins d'accès sont définis dans une variable d'environnement **jmaxPkgPath** (affectée dans le fichier **.jmaxrc**). Cette variable est repositionnable en entrant la commande suivante dans la console:
> `set jmaxPkgPath /chemin/package`
> On peut remarquer que ce path ne contient pas le chemin d'accès direct aux répertoires contenant les packages, mais celui des répertoires pères à ceux-ci. Il faut que le package et le répertoire le contenant portent le même nom.

1.3.3 Pour rendre les patches moins austères

Il est possible d'ajouter des images pour rendre le patch moins austère ou bien aporter un complément d'information sur le fonctionnement par le biais d'un schéma. On utilise pour cela, les déclarations d'icônes:

```
declareIcon %warning /u/formes/tisseran/Image/warning.gif
declareIcon %jmax /u/formes/tisseran/Image/jmax_logo.gif
```

Remarque :

> Les images sont considérées comme des commentaires, on les appelle en créant une boite de commentaire contenant:
> **%warning**
> **%jmax**

1.4 Programmation d'objet jMax

1.4.1 Définition de la classe

La définition d'une classe correspond à la définition de:
- la taille mémoire à réserver aux objets de la classe.
- le nombre d'inlets des objets de la classe.
- le nombre d'outlets des objets de la classe.
- pour chaque inlet, la liste des types de messages et des méthodes associées.
- et éventuellement, le type de message pour chaque outlet.

Remarque:

Selon cette définition, tous les objets de la classe ont le même nombre d'inlets et le même nombre d'outlets.

Les classes sont définies par la méthode :

```
fts_status_t my_class_instantiate( fts_class_t *class, int ac, fts_atom_t *at)
```

Les arguments de cette méthode sont:
- fts_class_t *class : un pointeur sur une classe.
- int ac : le nombre d'argument passés à la méthode.
- fts_atom_t *at : un tableau comportant les atomes des arguments, c'est à dire les types des arguments.

On déclare premièrement les variables utilisées dans cette méthode. Le passage de paramètres aux méthodes de la classe s'effectue par un tableau de type fts_symbol_t.
Après avoir déclaré les variables, on définit la taille, le nombre d'inlets et d'outlets des objets de la classe:

```
fts_status_t fts_class_init( fts_class_t *class, unsigned int size,
int ninlets, int noutlets)
```

On peut maintenant effectuer l'association entre les méthodes et les inlets des objets de la classe.

1.4.2 Déclaration des méthodes

On définit premièrement les types des atomes contenus dans le message envoyé à la méthode associée à l'inlet.

Il est bien sur possible d'affecter plusieurs méthodes à une inlet, il faut juste que chacune des méthodes soient associées à un type de message propre ou à un symbole propre.

Remarque :

En général, les méthodes sont définies par **fts_method_define** et ont comme paramètres d'appel :
- Un pointeur sur objet : fts_object_t *o
- Le numéro de l'inlet associée : int winlet
- Le type de message associé : fts_symbol_t s
- Le nom de la méthode associé : methode
- Le nombre d'argument attendu : int ac
- Le pointeur sur la liste d'atomes du mesage : const fts_atom_t *at

Si on préfère utiliser une méthode à nombre d'arguments variable , les deux derniers paramètres ne sont pas utilisés:

```
fts_method_define_var_args(fts_object_t *o, int winlet, fts_symbol_t
s, nom_methode)
```

Ainsi on peut choisir que l'objet soit appellé avec son nom seul ou avec son nom suivi d'arguments dans le cas d'une méthode d'initialisation à nombre d'arguments variable.
Dans ce cas il convient de tester dans la méthode init, la présence des arguments.

Après avoir spécifié les méthodes, on affecte le symbole dsp avec la routine :

```
fts_new_symbol(chaine de caractere representant le symbole)
```

On déclare la fonction dsp liée au symbole dsp:

```
dsp_declare_function(symbole dsp, nom de la fonction dsp)
```

On déclare les inlets et outlets signal:

```
dsp_sig_inlet(classe, numero inlet)
dsp_sig_outlet(classe, numero outlet)
```

Et à la fin, on retourne le succès de l'opération:

```
return(fts_Success)
```

1.4.3 Initialisation d'un objet de la classe: méthode init

Elle est appellée lors de l'existence d'une *"boîte"* avec le nom de l'objet. On affecte un pointeur sur la structure de l'objet par le biais d'une redéfinition de type.
Pour récupérer les arguments placés après le nom de l'objet, on utilise les fonctions de type :

```
fts_get_symbol_arg
fts_get_int_arg
```

Avec comme paramètres:
- Le nombre d'arguments
- Le tableau des types des argument
- La position de l'argument à récupérer
- La valeur par défaut

Une fois que les variables sont définies, on effectue les allocations mémoires pour les pointeurs. Celle-ci sont formulées de la même manière que en C ANSI, à la différence que l'on utilise l'opérateur **fts_malloc** au lieu de **malloc**.
On peut ensuite initialiser les variables de l'objet que l'on désire.
Ensuite on insère l'objet dans la liste dsp (dans le cas d'un objet dsp):
`dsp_list_insert(fts_object_t *)`
Remarque:

> Si l'on désire effectuer des initialisations qui dépendent du taux d'échantillonnage ou du nombre d'échantillons contenus dans les paquets de transfert, il faut faire ces initialisations dans la méthode put (**cf plus loin**. En effet on ne peut connaître le taux d'échantillonnage et la taille des paquets, seulement après lancement de la dsp ($\Longrightarrow$ execution de la méthode put)

1.4.4 Suppresion d'un objet de la classe: méthode delete

La supression de la classe en mémoire se fait par le biais de la méthode delete.
Cette méthode est appellée lorsque l'on efface la *"boîte"* de nom de l'objet ou que l'on change le nom de celle-ci.
L'objet n'étant plus utilisé, il faut donc revenir à l'état précédant le chargement en mémoire de la classe.
On pointe donc sur la structure de l'objet. Puis on libère les espaces mémoire alloués par l'appel de la fonction:
`free(type *)`
Enfin on supprime l'objet de la liste dsp:
`dsp_list_remove(fts_object_t *)`

1.4.5 Lancement de la dsp: méthode put

Cette méthode est appellée lors du lancement de la dsp. La dsp est lancée par en cochnant la case `dsp on` du `control panel` ou en envoyant le message `start` à l'objet `dac~`. On commence par créer un pointeur sur

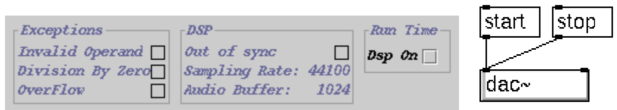


FIG. 1.9 – *Lancement de la dsp*

la structure de l'objet. Ensuite on crée des pointeurs (descripteur de dsp: `fts_dsp_descr_t`)qui seront utilisés pour écrire (lire) sur les sorties (entrées) signal:

`fts_get_ptr_arg(ac, at, 0, 0)`

Ces descripteurs permettent de connaître le taux d'échantillonnage:
`fts_get_input_srate`**(descripteur dsp, numéro inlet signal)** pour une entrée
`fts_get_output_srate`**(descripteur dsp, numéro outlet signal)** pour une sortie)
On peut aussi récupérer la taille des paquets de transfert de signal, généralement représentée par la variable `n_tick` (ceci est nécessaire pour effectuer un traitement des échantillons):
`fts_get_input_size`**(descripteur dsp, numéro inlet signal)**
Maintenant il faut envoyer les renseignement utiles à la fonction dsp. On affecte un pointeur sur l'objet:
`fts_set_ptr`**(adresse d'une case du tableau d'atomes, pointeur local sur l'objet)**, on précise les entrées (ou sorties) à utiliser :
`fts_set_symbol`**(adresse d'une case du tableau d'atomes utilisé,** `fts_dsp_get_input_name`**(descripteur dsp, numéro inlet signal)**
On appelle la fonction dsp en lui fournissant le nombre de paramètres contenus dans le tableau d'atomes et l'adresse de celui-ci:
`dsp_add_funcall`**(nom du symbole dsp, nombre d'arguments, adresse du tableau contenant les arguments)**

1.4.6 Fonction dsp

C'est la fonction qui s'occupe du traitement du signal. Elle est appellée par la méthode put. Cette fonction s'exécute à chaque paquet d'échantillons, elle ne s'arrête que lorsque l'on stoppe la dsp.
La méthode **put** lui fournit les adresses des pointeurs sur les entrées (sorties) dsp et du pointeur sur l'objet. On récupère ces adresses avec les routines suivantes:
- `fts_word_get_ptr`**(adresse de la case du tableau contenant le paramètre attendu)** - `fts_word_get_int` pour récupérer un entier.

Les signaux sont vus comme des tableaux de réels, on utilise donc des pointeurs sur réels (**float ***).
Ensuite on affecte généralement des variables locales avec les valeurs de celles contenues dans l'objet.
Le traitement des échantillons des paquets peut commencer.
On écrit le code comme en C ANSI.

1.4.7 Installation de la classe

L'installation de la classe est spécifiée dans la fonction `nomclasse_config`. Cette fonction est appelé par la fonction d'initialisation du module `module_init`. La fonction d'initialisation du module est spécifié dans le fichier de compilation du module (premier fichier listé dans le fichier Sources).

Remarque:

> Selon les versions de jMax, les installations de métaclasse ne sont pas effectuées par la même instruction. Sous une version 2.3.19, type de celle actuellement installée sur bonang, cette installation de métaclasse s'effectue par l'instruction:
>
> `fts_metaclass_create(fts_new_symbol, nom_fonction_instantiate, type de l'equivalence)`
>
> Alors que sous la version 2.4.5, le type de celle actuellement téléchargeable sur le site de l'IRCAM, l'instruction à utiliser est:
>
> `fts_metaclass_install(fts_new_symbol, non fonction instantiate, type de l'equivalence)`

1.5 Routines de bases

- Pour envoyer un mesage par un outlet, il faut utiliser la routine:
  `fts_outlet_send`**(pointeur sur l'objet, numéro de l'outlet, type du message, nombre d'atomes dans le message, adresse du tableau d'atome)**
- Pour affecter une valeur dans un tableau d'atome, on utilise la routine (dans le cas suivant avec un réel):
  `fts_set_float`**(adresse de la case du tableau a affecter, valeur)**
- Pour obtenir le taux d'échantillonnage:
  - d'une entrée signal:
  `fts_dsp_get_input_srate`**(descripteur dsp, numéro inlet signal)**
  - d'une sortie signal:

`fts_dsp_get_output_srate`**(descripteur dsp, numéro outlet signal)**

- Pour appeler des méthodes avec des messages:
  - Cas où il y a 3 arguments (réels):

```
a[0] = fts_s_float;
a[1] = fts_s_float;
a[2] = fts_s_float;
fts_method_define(cl, 0, fts_new_symbol(``open''), xxx_open, 3, a);
```

  Dans cet exemple, le message **open**, suivi de 3 arguments de type réel, reçu sur l'inlet numéro 0 entraîne l'appel de la méthode **xxx_open**.
  - Cas où il n'y a pas d'arguments:

```
fts_method_define(cl, 0, fts_new_symbol(``close''), xxx_close, 0, 0);
```

  Ici c'est le message **close** reçu par l'inlet numéro 0 qui entraîne l'appel de la méthode **xxx_close**.
  Il est bien sur possible d'effectuer des appels de fonctions par des messages avec ou sans paramètre:

```
fts_method_define_var_args(cl, 0, fts_new_symbol(``test''), xxx_test);
```

  On peut affecter toutes les méthodes précédentes à une même inlet:

```
a[0] = fts_s_float;
a[1] = fts_s_float;
a[2] = fts_s_float;
fts_method_define(cl, 0, fts_new_symbol(``open''), xxx_open, 3, a);
fts_method_define(cl, 0, fts_new_symbol(``close''), xxx_close, 0, 0);
fts_method_define_var_args(cl, 0, fts_new_symbol(``test''), xxx_test);
```

  Remarque :
  Dans l'exemple précédent, il faut noter qu'il peut y avoir au plus 2 arguments saisis avec le symbole “test”. De plus ces arguments doivent être de type float.

1.6 Pointeur sur fonction:

Comme en C, il est possible d'utiliser des pointeurs sur fonctions. On peut ainsi éviter d'effectuer un test sur un paramètre constant dans la boucle DSP, en définissant au préalable le traitement des échantillons lors de la modification du paramètre critique. Ainsi lors de l'appel des fonctions dites d' “update”, il est possible d'effectuer le test et d'affecter le pointeur sur la fonction adéquate.

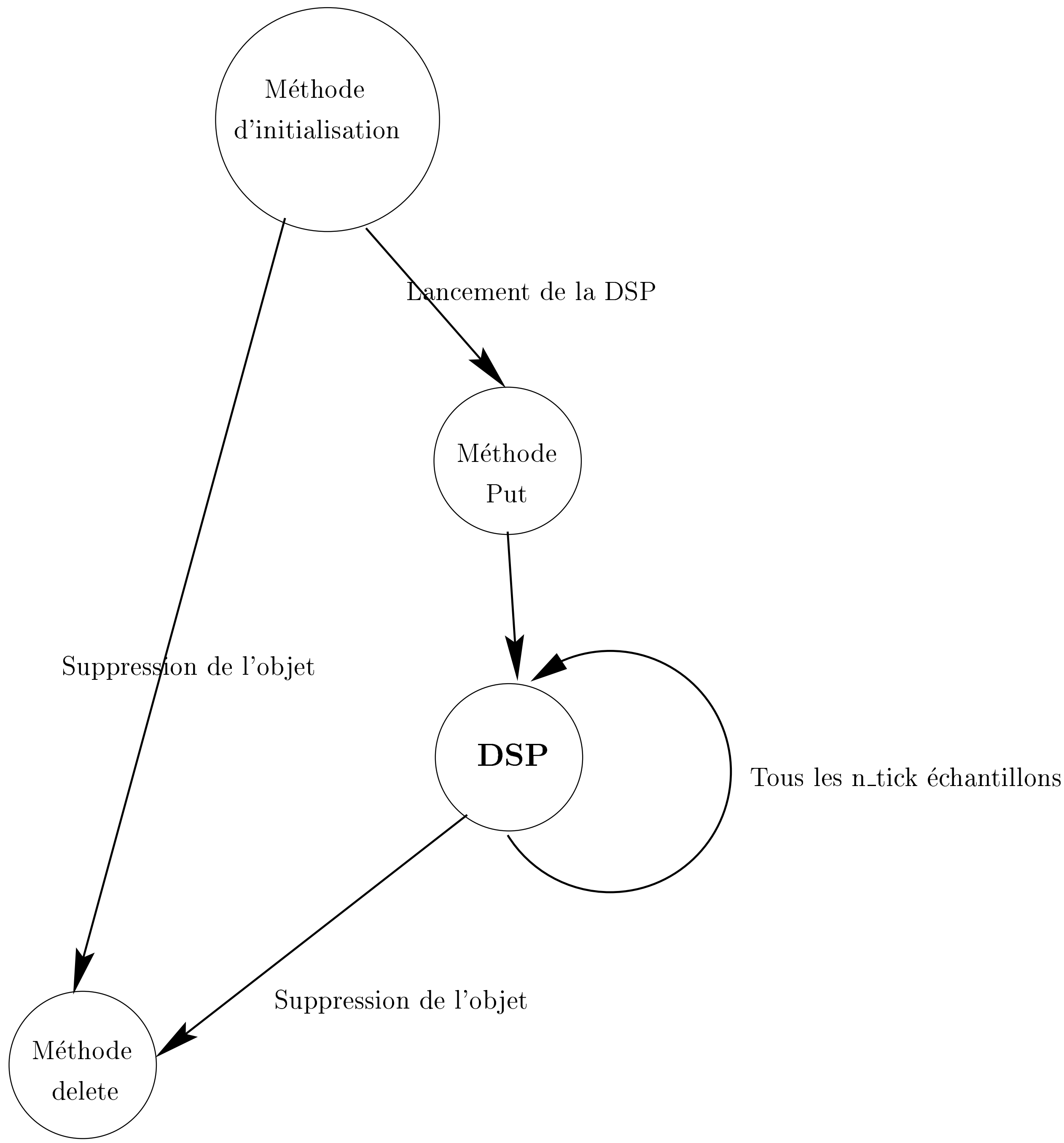
Afin de se prévenir des erreurs à la compilation, il est préférable de créer un fichier comportant tous les prototypes des fonctions utilisées. Ce fichier nous permettra aussi de vérifier que les fonctions sur lesquelles on est susceptible de pointer, sont affectables. En effet, pour utiliser cette fonctionnalité il est nécessaire que les fonctions, sur lesquelles on veut pointées, soient de même type et les mêmes arguments.

On essayera d'utiliser des fonctions qui prennent comme paramètre un pointeur sur la structure de l'objet, ou l'objet lui-même. Dans ce cas, il faudra juste faire attention au type de retour des fonctions.

Remarque:

> On peut effectuer une analogie entre l'utilisation de pointeur sur fonction au sein d'une structure et l'utilisation de la surcharge des méthodes en C++:
> La surcharge des méthodes correspond à des appels de fonction différent selon le nombre de paramètres. L'utilisation de pointeurs sur fonction correspond, elle, à des appels de fonction différent selon la valeur de un ou plusieurs paramètres.

Schéma d'exécution des méthodes associées à l'objet



1.7 Compilation et débuggage

1.7.1 Fichier Sources

Le fichier Sources contient la liste des fichiers à compiler. Il faut commencer la liste de fichiers par celui (dans l'exemple qui suit: non_linearite_fichier_config.c) qui appelle les fonctions de création de classe et précise le nom du module crée par la compilation. Ce nom sera utilisé par les fichiers de description de package.

Exemple de fichier Sources:

```
SOURCES = \
non_linearite_fichier_config.c \
miditofreq.c \
non_linearite_fonction_auxiliaire.c \
non_linearite_rationnelle.c \
non_linearite_segment.c \
non_linearite_totale.c \
```

Remarque:

> Lorsque l'on utilise des fonctions définies dans un fichier C annexe, il est nécessaire de créer un fichier contenant les prototypes. Celui-ci devra être inclus dans le code de l'objet afin d'éviter tout problème de compilation.

Contenu du fichier non_linearite_fichier_config.c :

```
#include "fts.h"

extern void test_config(void);
static void
nonlinearite_config_init(void)
{
  miditofreq_config();
  non_linearite_rationnelle_config();
  non_linearite_segment_config();
  non_linearite_config();
}

fts_module_t nonlinearite_module = {"nonlinearite",
 "nonlinearite synthesis classes",nonlinearite_config_init};
```

La dernière ligne de ce fichier précise le nom du module créé ainsi qu'une description de la classe. On peut remarquer que ce module contient 4 classes (il y a 4 appels de fonctions de création de classes).

1.7.2 Fichier Makefile

Ce fichier nous laisse la possibilité de rajouter des flags au compilateur pour inclure d'autres librairies (librairies X...), d'autres options nécessaire au fonctionnement de la classe créée.

Exemple de fichier Makefile:

```
DISTDIR = /usr/LOCAL/max/fts
PACKAGEROOT = ..

include $(DISTDIR)/Makefiles/Makefile.lib
include Sources

MODULE_CFLAGS= -I/usr/local/include -I/usr/include
MORESOLIBS= -lX11

all: $(LIBDIR)/libgraphique.a $(LIBDIR)/libgraphique.so

$(LIBDIR)/libgraphique.a: $(OBJECTS)
	$(AR) $(ARFLAGS) $(LIBDIR)/libgraphique.a $(OBJECTS)

TOCLEAN = $(LIBDIR)/libgraphique.a $(LIBDIR)/libgraphique.so

include $(DISTDIR)/Makefiles/Makefile.lib.end
```

Dans cet exemple, on utilise la librairie de X11. On crée les librairies libgraphique.a et libgraphique.so qui seront placées dans le réperetoire: $PWD/../lib/type_architecture/opt.

1.7.3 Débuggage

Pour débugguer un objet, on utilise l'option de compilation debug all:

```
gmake ARCH=irix63r10k MODE=debug all
```

Il faut modifier le répertoire où l'on va chercher le module (ce répertoire est défini dans le fichier de package .jpk), pour que le module chargé soit celui qui est compilé avec l'option de débuggage. Cette modification doit être effectué dans le fichier de configuration du package.

Ensuite, on lance jMax avec la commande :

```
jmax -jmaxHost nom_hote -jmaxConnection udpclient -jmaxPort
numero_port_tcp\ip
```

Il faut maintenant lancer le débuggueur (dbx sur sgi) avec l'executable fts :

```
dbx $CHEMIN_EXECUTABLE_FTS/fts
```

Dans le débuggueur, on entre la commande :

```
run udp nom_hote:numero_port_tcp\ip
```

Remarque1 :

En toute logique, la commande pour lancer jMax devrait comporter l'option `-jmaxMode debug` mais sur la version 2.3.19 (celle actuellement installé sur bonang), les objets standard n'ont pas été compilé en mode debug. Ainsi si on lance jMax avec l'option `-jmaxMode debug`, jMax ne pourra pas trouver les objets standards (dac∼, scale....) et dans ce cas on ne pourra pas générer de sons.

Remarque2 :

Ce que l'on appelle `jmaxHost` est le nom de la machine sur laquelle l'interface graphique de jMax est exécutée. La communication entre FTS et l'interface graphique est effectuée via le port TCP/IP spécifié par le flag `-jmaxPort numero_port`.

On peut maintenant ouvrir le patch contenant l'objet à débugguer. Puis on utilise le débuggueur comme avec n'importe quel exécutable. On peut évidemment remarquer que le lancement de fts crée plusieurs threads.

Voici un exemple sur bonang, les lignes commençant par # sont des commentaires:

```
# Lancement de jMax :
(bonang) jmax -jmaxHost bonang.ircam.fr -jmaxConnection udpclient -jmaxPort 20023
# On peut prendre n'importe quel port TCP\IP non utilise
# Maintenant on lance le debuggueur
(bonang) dbx /usr/LOCAL/max/fts/bin/irix63r10k/opt/fts
# Une fois que le debuggueur est lance
(dbx) run udp bonang.ircam.fr:20023
# Et voila jMax et fts tournent
```

Configuration de l'environnement

Chemin d'acces aux éxécutables

Il faut que le chemin d'accès aux éxécutables Java soit contenu dans la variable d'environnement PATH.

Chemin d'accès aux packages

La variable de l'environnement jMax `jmaxPkgPath` spécifie le chemin d'accès aux packages. En fait il s'agit du chemin d'accès aux répertoires pères du package. Le répertoire du package doit avoir le même nom que le package.

Package

Un package est un ensemble d'objets. Ainsi pour utiliser des objets additionnels à la version standard de jMax, il faut préciser que le patch à besoin d'un package suppplémentaire, par exemple dans le cas de l'utilisation de l'objet non_linearite_rationnelle~, il faut saisir la commande suivante dans la console:

```
package require non_linearite
```

Pour éviter de taper cette commande à chaque lancement de jMax, on peut l'ajouter dans le fichier de configuration correspondant au patch créé. Par exemple pour un patch sauvegardé sous le nom toto.jmax, le fichier de configuration sera toto.env .

Remarque :

> Le fichier de configuration est lu lorsque l'on ouvre le patch (file, open). Si, dans le cas de l'exemple précédent, le fichier toto.env n'est pas trouvé, il y a lecture du fichier project.env (dans le cas ou celui-ci existe). Ce fichier correspond à la configuration par défaut des patchs contenu dans le même répertoire que ce fichier. Dans le cas ou le fichier project.env n'est pas trouvé, les packages utilisés sont ceux par défaut appelés par jMax, spécifié dans le fichier `$JMAXDIR/config/packages.tcl` (`$JMAXDIR` est le répertoire contenant jMax).

Utilisation de jMax

Lancement de jMax

On lance jmax en utilisant la ligne de commande suivante:

```
jmax -new true
```

Remarque1 :
On suppose que la variable PATH contient les chemins d'accès aux exécutables Java et qu'il existe un alias jmax pointant sur l'exécutable jmax de la distribution (dans le cas de bonang : `/usr/LOCAL/max/bin/jmax`).

Remarque2 :
L'option `-new true` permet l'utilisation de la nouvelle interface graphique, cette option ne semble pas nécessaire pour la version 2.4.5 de jMax.

Lancement de jMax sur deux machines

Il est possible de faire tourner l'interface graphique sur une machine et le moteur de calcul sur une autre. Pour utiliser cette fonctionnalité il faut lancer jMax avec la ligne de commande suivante:

```
(marimba) jmax -jmaxHost marimba.ircam.fr -jmaxConnection udpclient -jmaxPort
20023 -new true
(bonang) /usr/LOCAL/max/fts/bin/irix63r10k/opt/fts udp
marimba.ircam.fr:20023
```

Dans cet exemple, l'interface graphique tourne sur marimba, FTS sur bonang et la communication se fait par le port TCP/IP 20023.

Note sur les codes sources des objets

Fichier de données

Les fichiers de données (tableaux caractéristiques des notes, coefficient pour l'interpolation de Lagrange...) sont contenus dans le répertoire:
`/u/formes/tisseran/DATA/`
Si l'on désire modifier le chemin d'accès des fichiers de données, il suffit d'éditer les fichiers de définition des objets et modifier la constante `DATADIR`.

Modification d'une fonction entre jMax 2.3.19 et jMax 2.4.5

Les codes sources sont compatibles avec la version 2.3.19 de jMax. Pour les utiliser sous la version 2.4.5, il faut modifier les fonctions de création de classes (ce sont les fonctions appellées dans le premier fichier de SOURCES). La modification consiste à changer le nom de la fonction `fts_metaclass_create` en `fts_metaclass_install`.

Compilation

Pour compiler un objet, il faut se placer dans le répertoire contenant les sources et exécuter la commande suivante: `gmake ARCH=irix63r10k MODE=opt clean all`
Cette commande compile les sources pour une machine R10000 avec IRIX6.3, les anciens fichiers de compilation sont effacés (`clean all`).

Annexe A

Incovénient de l'interface graphique

A.1 Charge apportée par l'utilisation de Java

A.2 Problème de la surcharge des outlets

Visualisation des liens

Il faut noter qu'il n'y a pas de différence graphique entre des liens "contrôle" et des liens "signal". Il est évident que ceci peut compliquer la compréhension du patch. Cependant on ne peut pas faire d'erreur en reliant un lien de type "contrôle" à une entrée signal (au vis versa), car les types des inlets sont définis lors des instantiations des classes des objets utilisés.

Plusieurs liens sur un outlet

Lorsque l'on ajoute un outlet à un objet, jMax le place à droite des précédents. Dans le cas des objets où on ne visualise qu'un inlet, par exemple le bang ou le slider, il faut savoir que les outlets sont superposés. Ainsi la priorité ira du lien le plus récent au plus ancien.

L'objet "+" effectue l'opération d'addition lors de la modification d'état de l'inlet gauche. Les opérandes sont les valeurs reçues par les inlets. Au cas où l'objet n'a pas encore reçu de valeur sur l'inlet de droite, il utilise son argument comme opérande complémentaire à celui reçu par l'entrée de gauche pour effectuer l'addition.

Cet exemple (Figure A.1) nous montre bien le problème lié à l'utilisation de plusieurs liens sur un même outlet. Il n'y a pas d'outil graphique

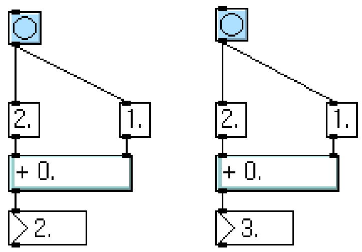


FIG. A.1 – *Exemple illustrant le problème de la surcharge d'un outlet*

permettant de visualiser la priorité entre les lines, de ce fait deux patches semblables, graphiquement, ne donnent pas le même résultat. Nous pouvons en conclure qu'il n'est pas possible de prévoir le comportement d'un patch contenant une surcharge d'outlet.

Annexe B

Problème de documentation

B.1 Seule documentation consultable le site WEB

Les seuls documents consultables se trouvent sur le site web de l'équipe Temps Réel. Malheureusement la documentation pour la programmation des objets FTS n'est pas mis à jour. Ainsi les prototypes de certaines fonctions ont été modifiés ou pire certaines fonctions ont été renommées. Actuellement, la documentation la plus à jour semble se trouver dans l'archive comprenant les sopurces de jmax-2.4.5. Mais cette documentation est très succincte. Il faudra donc pour le futur développeur consulter la dopucmentation du site internet, celle livrée avec la distribution et les sources d'objets déjà écrit pour pouvoir écrire ses propre objets. Cependant il restera encore le problème des versions incompatibles entre elles.

B.2 Différences entre les versions

Suite à l'installation de la version jamx-2.4.5 sur mon ordinateur personnel, j'ai put m'aprecevoir que les codes des objets écrit lors de mon stage, ne pouvaient pas s'exécuter chez moi. Après une brève lecture de la documentation livrée avec cette distribution, j'ai pu m'aprecevoir du changement de nom de la fonction d'installation de métaclasse. Je suis donc allé sur le site de l'équipe Temps Réel pour consulter les différences entre les différentes versions. On peut effectivement trouver ces différences, si l'on se limite aux versions 2.3.12 à 2.3.15.