Manuel du Programmeur

Traitement de séquences et manette Wii

Nicolas CHEIFETZ
Supervisé par Thierry Artières

juin-juillet 2008

Laboratoire d'Informatique de Paris 6

Table des matières

Chapitre 1

Rappel des objectifs

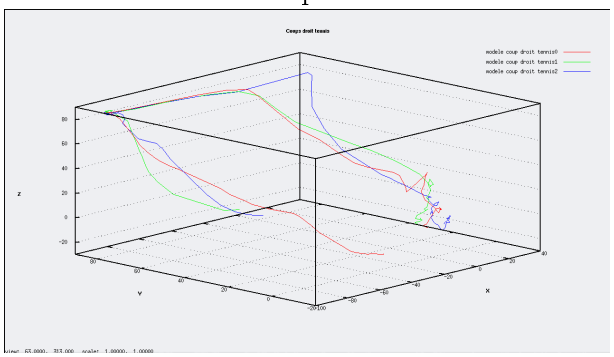
Ce Stage a pour but de concevoir un outil de base pour la manipulation, le traitement et la reconnaissance des gestes réalisés par une NINTENDO WIIMOTE. La recherche est basée sur l'étude des accélérations. Le langage de programmation utilisé est `Java`.

- **Manipulation des données :** stockage, visualisation, lecture et écriture des données, création de bases de données, Cross training ...
- **Traitement des données :** lissage, interpolation, centre-réduction, intégration, ajout de caractéristiques (features)...
- **Reconnaissance :** apprentissage de modèles (`DTW`, `CRFBranch`, `LibSVM`), reconnaissance de séquences à l'aide des modèles

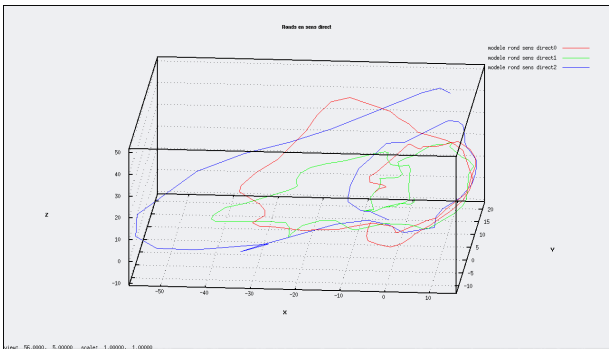
L'application visée est de manipuler l'interface de Windows XP au travers d'une Wiimote.

Exemples de gestes captés par une Wiimote (accélérations tridimensionnelles) :

Trois coups droits :



Trois ronds dans le sens direct :



1.1 Remarque :

Le projet succède au travaux effectués lors du PIAD n°25 [4] par Yasmina Seddik, Samuel Ortmans et Nicolas Cheifetz. Beaucoup d'aspects seront répétés, il n'est donc pas nécessaire de lire l'intégralité des documents mis à votre disposition sur le lien cité dans les Références ; ceci-dit, il vous est conseillé de le faire pour une meilleure compréhension du sujet.

Chapitre 2

Choix d'implémentation

L'architecture du projet est organisée autour du concept de boîtes, une structure amplement utilisée pour les jeux en temps réel. L'avantage des boîtes est qu'elles sont "imbriquables" et réutilisables.

Les boîtes sont *imbriquables* dans le sens où elles peuvent être combinées dans n'importe quel ordre. C'est important car cela facilite les tests. Si on décide par exemple de faire un ultérieur prétraitement, il suffira d'ajouter la boîte de prétraitement correspondante, à la séquence de boîtes appelées.

Elles sont *réutilisables* car chacune des boîtes peut être utilisée pour effectuer la tâche qui lui est propre, dans un contexte autre que celui de ce logiciel, à condition que les formats des entrées-sorties soient respectés.

Nous avons choisi d'implémenter notre projet sous l'IDE `Eclipse`. Pour charger facilement le PIAD sous Eclipse, nous avions créé une page web [1]. La librairie à inclure n'est plus `LibPIAD.jar` mais la librairie `LibStage.jar`. La présente version du projet est exploitable sous le système d'exploitation *Windows XP*.

Pour aborder le projet, vous disposez de sept scripts batch dans `Stage_final/start` :

`Main.bat` :	propose 20 fonctionnalités utiles au développement (appel au programme `Main.java`)
`Create_bases_*.bat` :	crée des bases de données à partir d'une wiimote, construction des bases par Cross Training, création des fichiers lisibles par CRFBranch (appel au programme `Create_bases.java`)
`TestDTW.bat` :	test avec le modèle DTW au travers d'une Wiimote (appel au programme `TestDTW.java`)
`TestCRF.bat` :	test avec le modèle CRFBranch au travers d'une Wiimote (appel au programme `TestCRF.java`)
`TrainTestCRFBranch.bat` :	apprendre/tester un ou plusieurs modèle(s) CrfBranch (appel au programme `TrainTestCRFBranch.java`)

Vous pouvez ouvrir les fichiers `.bat` dans un éditeur de texte pour plus d'informations techniques. Il vous est aussi conseillé de jeter un coup d'oeil au Manuel de l'utilisateur [3] et au "Cahier des Charges" du PIAD [4].

[1] `Procédure pour travailler sur le piad sous Eclipse` : *ici*

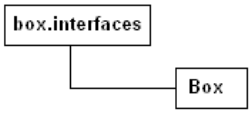
Chapitre 3

Architecture du logiciel

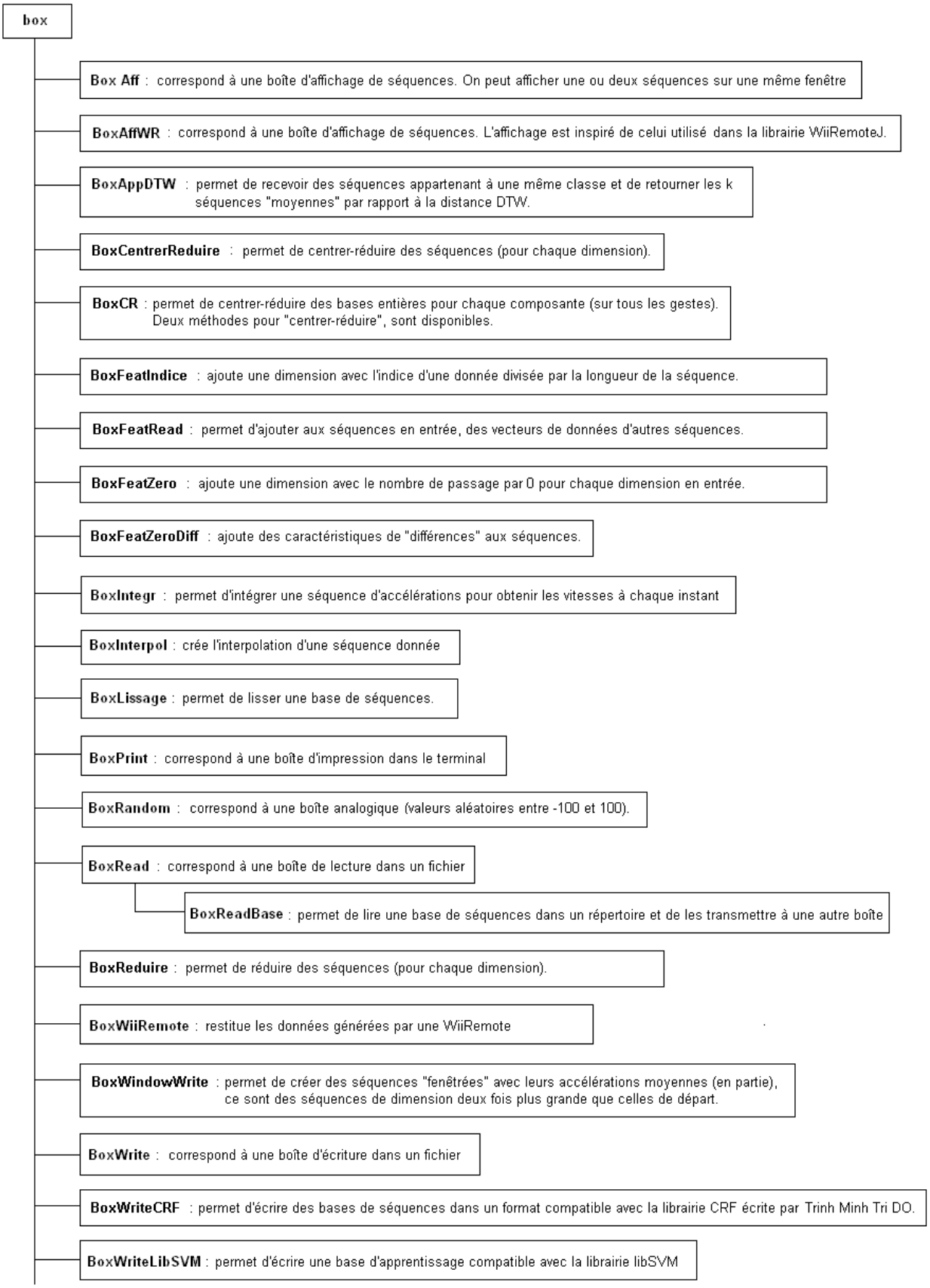
Voici des diagrammes de package réduits à l'essentiel. Vous trouverez des informations complémentaires plus fournies dans la JavaDoc du projet.

Architecture des packages `outils.interfaces` *et* `outils` *:*

Architecture du package `box.interfaces` *:*



Architecture du package `box` *:*



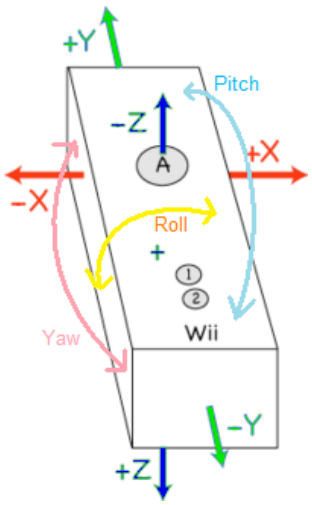
Chapitre 4

Description du code

4.1 Eléments utilisés

Le projet exploite des signaux multidimensionnels. Théoriquement, on peut étudier des séquences de n'importe quelle dimension. Chaque séquence brute (échantillonée sur une Wiimote) est formée par des signaux en trois dimensions (une par capteur). Le projet permet à l'utilisateur de reconnaître des gestes restitués par la Wiimote grâce à la technologie Bluetooth [1].

Pour faire le lien entre les signaux captés par Bluetooth et notre projet Java, nous avons utilisé la librairie WiiremoteJ. Nous utilisons actuellement la version 1.3 [1]. Cette librairie nous permet de collecter les accélérations de la manette grâce à trois accéléromètres, commander la vibration, l'allumage des diodes, récupérer deux angles d'inclinaison,... Bien que ses fonctionnalités soient assez complètes, il n'est pas encore possible de collecter avec précision les positions et les angles d'inclinaison de la manette.



Nintendo Wiimote

Remarque

La Wiimote est connectée à l'ordinateur par Bluetooth, grâce à une implémentation de JSR082 : Bluecove 2.0.2+ sous Windows XP (AvetanaBluetooth sous Unix).
Une procédure est décrite dans le Manuel de l'Utilisateur [3].

[1] Bluetooth - Version 2.0 + EDR

Nous utilisons trois modèles discriminants pour classifier les séquences.
Le premier est un algorithme de mesure de similarité entre deux séquences par programmation dynamique **DTW**, le second est une méthode à noyaux : machine à vecteurs support (**SVM**), et le dernier est une instance particulière des champs aléatoires : les champs de Markov conditionnels (Lafferty, 2001) ou Conditionnal Random Fiels (**CRF**).

1. L'algorithme **DTW** a entièrement été implémenté dans la classe `outils.DTW.java` (*voir* 3 Architecture du logiciel)
2. Pour le deuxième modèle, nous faisons appel à une librairie de référence en apprentissage : LibSVM version 2.86 [2]. L'interfaçage avec la librairie s'est fait par un stockage de séquences sous un format compatible.
3. Le modèle CrfBranch est performant pour des classes multimodales (plusieurs "expressions" d'un même geste). Ce modèle a été entièrement développé par Trinh Minh Tri DO [6] et s'applique à des bases de séquences, par exemple des séquences numériques d'accélérations. Ce modèle utilise l'algorithme de gradient avec la méthode de quasi-Newton avec mémoire limitée **LBFGS** (Nocedal, 1989).

4.2 Eléments produits

Hormis les librairies extérieures décrites précédemment, nous avons implémenté un canevas d'entrées-sorties typées. Nous avons créé des boîtes "imbriquables" et réutilisables (*voir* 2 Choix d'implémentation) dans le package `box`. Ces boîtes communiquent au moyen de la méthode `process()`, commune à toutes les boîtes.

Ces boîtes font appels à d'autres classes Java pour réaliser certains travaux. Ces classes annexes sont rassemblées dans le package `outils`. Par exemple, la boîte `BoxRead` manipule la classe `Lecture` qui gère entièrement la lecture dans un fichier.

Il existe aussi un package `ouils.viewer3D` destiné à la visualisation 3D lors de la manipulation de la manette par l'utilisateur. Les programmes Java qui s'y trouvent ont été développé par Frederic DE STEUR [5]. Cet outil est utilisable mais n'est pas encore très performant dans notre projet.

Enfin, j'ai créé une archive `CRF.jar`. Celle-ci contient les outils qui crée le lien avec le modèle CrfBranch écrit dans d'autres langages que Java [6] et permettent l'étude des séquences par le modèle CrfBranch. L'accés est centralisé dans le programe `CRF.java` situé à l'intérieur du .jar.

4.3 Paramètres d'implémentation

On échantillonne les signaux envoyés par la Wiiremote au rythme d'une centaine de mesures par seconde. La taille des paquets est inscrite dans la variable `buffer_length` : champ commun à toutes les boîtes ; cette variable est le seul champ de la classe abstraite `Box`.
`buffer_length` est fondamental pour le bon déroulement de notre projet. Cette variable est en général déterminée par la première boîte créée et sa valeur ne sera jamais modifiée au cours d'une tâche. Usuellement, on instancie la variable `buffer_length` par un entier entre 10 et 20.

Par convention, chaque séquence du projet est contenue dans un fichier texte ("`.txt`") ayant comme en-tête : le nombre d'entrées et le nombre de données pour chaque entrée. En ce qui concerne les bases de séquences lisibles par le modèle CrfBranch, le format est quelque peu différent : un fichier contient

l'ensemble (la base) des séquences. L'en-tête est alors composé du nombre de séquence dans le fichier et du nombre d'entrées (nombre de dimensions).
Les séquences "pré-traitées" ont des noms de fichiers indiquant les prétraitements qu'elles ont subis. Par exemple, un fichier `Interpol_Test.txt` contient l'interpolation d'une séquence brtutes, contenue dans le fichier `Test.txt`. De plus, dans une base de séquences, les séquences sont sauvées par ordre lexicographique dans le sens où la terminaison des fichiers croit en fonction de leurs positions. J'ajoute que les séquences peuvent être théoriquement être numérotées de 0 à 99. Par exemple, le répertoire `Dir` contiendra des fichiers du type `Dir00.txt`, `Dir01.txt`, `Dir02.txt`, etc.

Les modèles de gestes pour le modèle DTW seront donc sauvés de la même manière :
dans `Stage_final\Project\Datas\ModelesDTW\modele_oui`, on trouvera des fichiers `modeles_oui00.txt`, `modeles_oui01.txt`, etc.

4.4 Principaux algorithmes

4.4.1 DTW

Dynamic Time Warping est un algorithme de mesure de similarité entre deux séquences. La continuité est moins importante dans DTW que dans d'autres algorithmes de filtrage : DTW est adapté à l'appariemment des séquences avec des informations manquantes.

L'algorithme se déroule en deux phases :

1. phase avant (**forward**) : on calcule les coûts de tous les chemins valides
2. phase retour (**back-tracking**) : on détermine quel est le chemin optimal

L'algorithme suivant est implémenté dans la classe `outils.DTW.java` :

```
//DTW = distance entre deux séquences

//Soient les deux séquences à étudier (sous forme de tableaux de réels)
double[][] seq1
double[][] seq2

//les deux séquences doivent avoir le même nombre d'entrées
si seq1.length != seq2.length alors exit

//appariement linéaire
double[][] apLin
pour j1=0 à seq1[0].length-1
   pour j2=0 à seq2[0].length-1
      apLin[j1][j2] = distanceEuclidienne(seq1[ :][j1],seq2[ :][j2])
   finpour
finpour

//1ère phase de l'algorithme : phase avant (forward)
double[][] forward
//initialisation
forward[0][0] = apLin[0][0]
forward[ :][0] = apLin[ :][0] + forward[ :-1][0]
forward[0][ :] = apLin[0][ :] + forward[0][ :-1]
```

```
//itérations
pour j1=1 à seq1[0].length-1
   pour j2=1 à seq2[0].length-1
                                                  forward[j1-1][j2],
       forward[j1][j2] = apLin[j1][j2]+ min(  forward[j1-1][j2-1]   )
                                                  forward[j1][j2-1]
   finpour
finpour

//2e phase de l'algorithme : phase arrière (backtrack)
Vector<Double> backtrack
int j1 = seq1[0].length-1
int j2 = seq2[0].length-1
tant que j1!=0 et j2!=0
   si j1==0 alors backtrack.add(forward[0][j2-1]), j2-- fin si
   sinon si j2==0 alors backtrack.add(forward[j1-1][0]), j1-- fin sinon si
   sinon
      double cas1 = forward[j1-1][j2]
      double cas2 = forward[j1][j2-1]
      double cas3 = forward[j1-1][j2-1]
      si cas1<=cas2 et cas1<=cas3 alors backtrack.add(cas1), j1-- fin si
      sinon si cas2<=cas1 et cas2<=cas3 alors backtrack.add(cas2), j2-- fin sinon si
      sinon res.add(cas3), j1--, j2-- fin sinon
   fin sinon
fin tant que

double DTW = \sum_i backtrack.element(i)
```

4.4.2 Calcul d'un modèle DTW à k séquences

Soient $(S_i)_{i\in\{1,\cdots,n\}}$, les n séquences d'apprentissage.
$\forall i \in \{1,\cdots,n\}$, $D(i) = \sum_{j\neq i} d(Si, Sj)$, où d est la distance DTW de S_i à S_j.

Les k séquences pour lesquelles D est minimal, constituent le modèle.

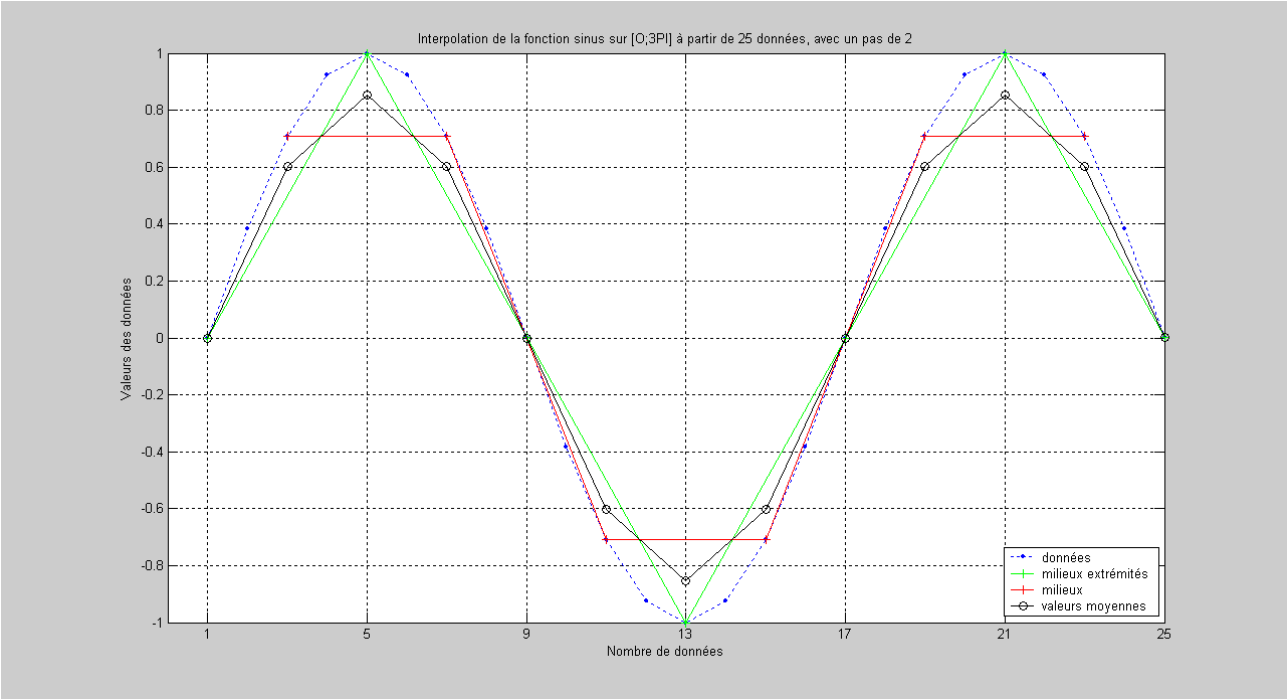
4.4.3 Interpolation

On utilise toujours la même nomenclature pour désigner les différentes courbes et leurs couleurs :

- données
- extrémités
- milieux
- valeurs moyennes

Les "valeurs moyennes" représentent les points de la courbe interpolée. Pour trouver cette courbe, nous construisons deux courbes annexes : extrémités et milieux, elles-mêmes déterminées à partir des données.

L'interpolation est décrite par l'exemple jouet suivant :



Interpolation de la fonction sinus sur $[0; 3\Pi]$ *à partir de 25 données, avec un 'pas de lissage' de valeur 2*

Chapitre 5

Interfaçage avec d'autres langages

Nous avons été amené à créer un interfaçage de programmes développés sous `C`, `C++` et `Fortran` vers le langage de programmation informatique orienté objet `Java`. En effet, le projet a été codé en `Java` mais le modèle CRFBranch [6] a été écrit en `C`, `C++` et `Fortran`. Nous avons fait appel au software `SWIG`[1]. C'est un logiciel de développement qui relie les programmes écrits en `C` et `C++` avec une variété de haut niveau des langages de programmation.

Cependant, il n'est pas possible d'utiliser SWIG directement sous Windows. Il faut créer un environnement UNIX pour le développement d'applications natives Win32. Nous avons choisi l'environnement `MinGW` [2]. Les versions de développement de `SWIG` sont disponibles sur le serveur SVN situé à SourceForge. Une fois `MinGW` et `MSYS` installés, téléchargez `Subversion` puis installez `SWIG` dans `msys` [3] avec une commande du type (dans l'invite de commandes) :

```
svn co https ://swig.svn.sourceforge.net/svnroot/swig/trunk swig
```

La prochaine étape est de créer une librairie dynamique `.dll` (`.so` sous Unix), des fichiers java, et un fichier C++, créant ainsi le lien entre des programmes `C`, `C++` et `Fortran` et le reste du projet.

Les commandes exactes à lancer dans le shell, dépendent de l'architecture de votre système. J'ai utilisé deux commandes pour créer la librairie `crfbranch.dll` et neuf fichiers java.
La première :

```
/usr/src/swig/swig.exe -c++ -java SWIGTYPES.i
```

crée les neufs fichiers java et un fichier C++ :

- `CrfBranch.java` : contient les méthodes natives, appelle les méthodes de la librairie `crfbranch.dll`
- `SWIGTYPES.java` : contient les méthodes pour manipuler les types ci-dessous
- `SWIGTYPESJNI.java` : interface de `SWIGTYPES.java`
- `SWIGTYPE_p_double.java` : type correspondant au `double*`
- `SWIGTYPE_p_FILE.java` : type correspondant au `FILE*`
- `SWIGTYPE_p_int.java` : type correspondant au `int*`
- `SWIGTYPE_p_p_char.java` : type correspondant au `char**`
- `SWIGTYPE_p_p_double.java` : type correspondant au `double**`
- `SWIGTYPE_p_p_p_double.java` : type correspondant au `double***`
- `SWIGTYPES_wrap.cxx` : contient les méthodes créant le lien avec les fichiers Java

[1] `SWIG` : *http://www.swig.org/*
[2] `MinGW` : *http://www.mingw.org/*
[3] `instructions SVN` : *http://www.swig.org/svn.html*

Puis la seconde commande est :

```
gcc -Wall -mrtd -I./ -I"C:\Program Files\Java\jdk1.6.0_10\include"
-I"C:\Program Files\Java\jdk1.6.0_10\include\win32" -I/mingw/lib
-shared useful.cxx maths.cpp lbfgs.f lbfgs_wrapper.c crfbranch.cxx
SWIGTYPES_wrap.cxx -Wl,--add-stdcall-alias,-L/bin -o crfbranch.dll
-lstdc++ -lg2c
```

Une librairie `crfbranch.dll` devrait alors être créée.

5.1 Remarque

Si vous décidez de créer votre propre librairie, il faut :

- veiller à inclure le bon en-tête dans l'interface appelée lors de la première commande.
  Pour moi, l'en-tête était :

  ```
  /* SWIGTYPES.i */
   %module SWIGTYPES
   %{
    /* Put header files here or function declarations like below */
  #include "crfbranch.hxx";
  %}

  /* First we'll use the pointer library */
  %include cpointer.i
  %pointer_functions(int      , intp);
  %pointer_functions(char*    , charpp);
  %pointer_functions(double   , doublep);
  %pointer_functions(double*  , doublepp);
  %pointer_functions(double**, doubleppp);
  ```

  Les appels à la méthode `pointer_functions` de l'interface `cpointer.i` (dans SWIG) permettent de créer les types correspondants aux `int*`, `char**`, `double*`, `double**` et `double***`.

- se procurer les bonnes librairies nécessaires au programmes natifs.
  Pour moi : `useful.cxx`, `maths.cpp`, `lbfgs.f`, `lbfgs_wrapper.c`, `lstdc++`, `lg2c`.

Références

[1] Michael Diamond (aka ChaOs). *WiiRemoteJ library*, 2007. Software available at http://www.wiili.org/index.php/WiiremoteJ.

[2] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM : a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[3] Nicolas Cheifetz. Manuel de l'utilisateur, 2008.

[4] Yasmina Seddik Nicolas Cheifetz and Samuel Ortmans. *Projet IAD - Master 1 IAD*, 2008. Project available at http://che.nico.ifrance.com/PIAD.

[5] Frederic DE STEUR. Wiimotecommander, 2008. Software available at http://sourceforge.net/project/showfiles.php?group_id=222335.

[6] DO Trinh Minh Tri. Multi branch conditional random fields implementation, 2007. http://webia.lip6.fr/~do/pmwiki/index.php/Main/Codes.