

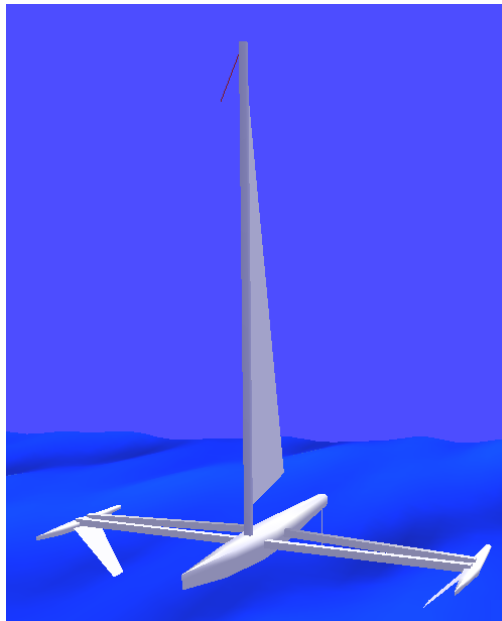
Modélisation et contrôle de l'Hydroptère

Projet de semestre

Basile Graf

Collaborateurs responsables : Sébastien Gros, Phillipe Müllhaupt
Prof. Dominique Bonvin
Ecole Polytechnique Fédérale de Lausanne
basile.graf@epfl.ch

19 juin 2006



Résumé

Ce document est une partie du compte-rendu concernant mon projet de semestre *Modélisation et contrôle de l'Hydroptère*. L'autre partie du compte rendu étant la documentation du programme. Les principaux buts du projet ont été de développer une implémentation rapide du modèle de l'hydroptère et de l'intégrer dans une réalité virtuelle en temps réel. Pour ce faire, un programme de traduction automatique de MATLAB en C/C++ pour certaines parties de code a été écrit. Le modèle entier a également été écrit en C/C++ et un solver d'équations différentielles partielles en C a été utilisé. Un graphisme 3D en OpenGL pour un rendu temps réel a été écrit.

Les résultats sont une vitesse de calcul multipliée par plus de mille pour la simulation du modèle ainsi que la possibilité de piloter et de voir évoluer l'Hydroptère, même sur un ordinateur de puissance courante.

Table des matières

1	Modèle MATLAB de l'hydroptère	4
1.1	Géométrie	4
1.2	Détermination du second membre	5
1.3	Limitations	5
2	Conversion MATLAB - C/C++	7
2.1	Exemple de conversion	8
2.2	Codes générés	9
3	Utilisation du programme principal : <i>hydroMEX3</i>	12
3.1	Lancer le programme	12
3.2	Utiliser le programme	13
4	Documentation du code source	16

Introduction

L'Hydroptère est un voilier à hydrofoils (surfaces portantes aquatiques). Les foils génèrent suffisamment de poussée pour porter le bateau ce qui réduit drastiquement les forces de frottement avec l'eau. Lorsque l'équilibre des forces est atteint, les coques de l'Hydroptère ne sont plus en contact avec l'eau et celui-ci "vole" grâce à ses foils et atteint ainsi des vitesses nettement supérieures à celles atteintes par des voiliers classiques. Cependant, l'Hydrofoil présente le désavantage d'être peu stable.

Ces considérations rendent intéressante l'idée de pouvoir disposer d'un modèle de la machine et de pouvoir ainsi tester différentes stratégies de contrôle.

1 Modèle MATLAB de l'hydroptère

On donne ici une brève description du modèle MATLAB de l'hydroptère qui a servi de base pour le projet [1].

1.1 Géométrie

Pour le calcul de la dynamique du modèle, on considère le bateau comme un objet solide dans l'espace. Comme le montre la figure 1, le bateau possède six degrés de liberté : trois pour sa position et trois pour son attitude.

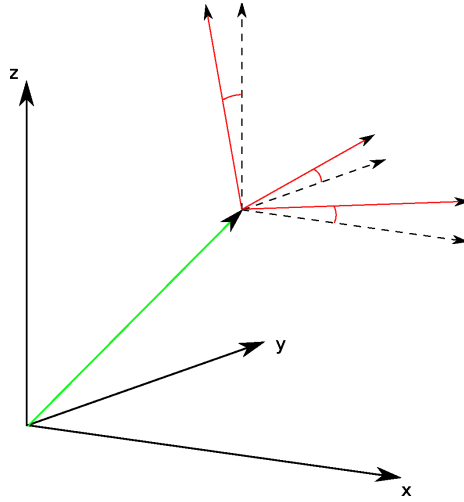


FIG. 1 – Six degrés de liberté pour un objet solide dans l'espace

$$\vec{r} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \vec{a} = \begin{pmatrix} \phi \\ \theta \\ \psi \end{pmatrix} \quad (1)$$

Pour décrire la dynamique du système, nous avons donc besoin des coordonnées $\begin{pmatrix} \vec{r} \\ \vec{a} \end{pmatrix}$ et de leur dérivées temporelles $\begin{pmatrix} \dot{\vec{r}} \\ \dot{\vec{a}} \end{pmatrix}$ comme variables d'état. L'état est donc décrit par

$$\vec{q} = \begin{pmatrix} \vec{r} \\ \vec{a} \\ \dot{\vec{r}} \\ \dot{\vec{a}} \end{pmatrix}, \quad \dim(\vec{q}) = 12 \quad (2)$$

L'évolution temporelle d'un tel système est décrite par un système d'équations différentielles du type

$$\dot{\vec{q}} = f(\vec{q}, \vec{Q}, \vec{p}) \quad (3)$$

\vec{q} : état

\vec{Q} : forces appliquées

\vec{p} : paramètres (positions des gouvernes, vent, etc...)

Le premier objectif est donc de déterminer la fonction vectorielle f . Cette fonction est (très) complexe. Ensuite, une intégration numérique du problème permet de simuler le comportement du bateau.

1.2 Détermination du second membre

La détermination de la fonction f (3) se fait en grande partie par le calcul du Lagrangien et de ses dérivées partielles. Ce calcul tient également compte des forces généralisées appliquées au système.

$$L = E_{cin} - E_{pot} \quad (4)$$

$$\frac{\partial}{\partial t} \frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_i \quad (5)$$

Mais à leur tour, les forces généralisées Q_i sont des fonctions des paramètres \vec{p} et de l'état \vec{q}

$$Q_i = g_i(\vec{p}, \vec{q}) \quad (6)$$

Les fonctions g_i expriment les dépendances des Q_i à la position et à l'attitude (matrice de rotation) de la machine ainsi qu'à tous les paramètres extérieurs (hauteur de la mer par rapport aux surfaces portantes, vent, etc...). La figure 2 montre un schéma bloc du programme MATLAB utilisé pour générer le modèle. On y utilise le toolbox de calcul symbolique pour générer des expressions qui pourront ensuite être évaluées numériquement lors de l'intégration.

1.3 Limitations

Les expressions à évaluer lors de chaque pas d'intégration générée par le programme sont très grandes et nécessitent des temps de calcul non négligeables (les chaînes de caractères qui représentent ces expressions dépassent 20000 caractères pour certaines d'entre elles).

Dans la version MATLAB de la simulation (utilisation du solver `ode45(...)`), le temps relatif de simulation sur un Pentium IV à 1.7GHz est de environ 400%. On est encore loin d'une simulation en temps réel avec affichage simultané.

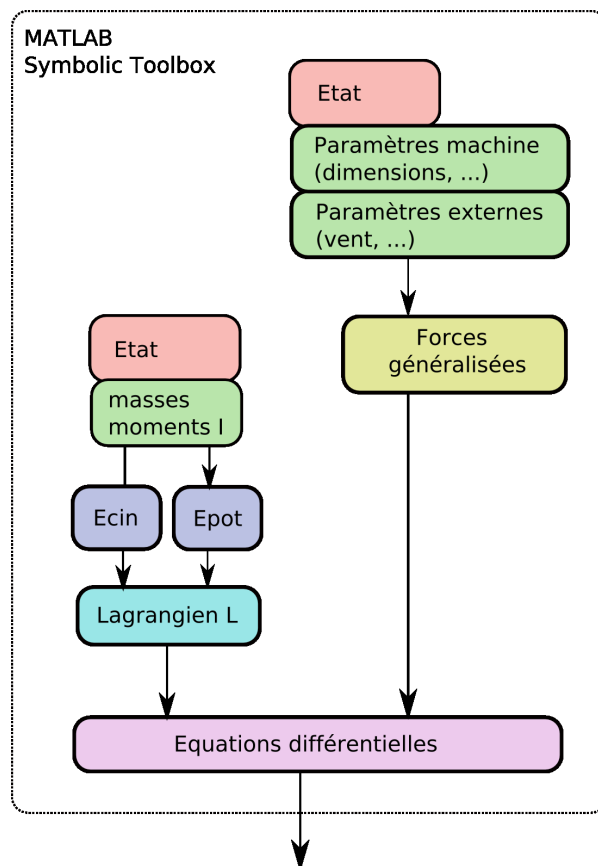


FIG. 2 – Schéma bloc de la génération du modèle

2 Conversion MATLAB - C/C++

La raison pour laquelle la simulation est 'lente' tient au fait que MATLAB est un langage interprété. Comme nous le verrons, la traduction du modèle en C/C++ et sa compilation permettront un gain de vitesse dramatique. MATLAB propose une routine de traduction en C (MATLAB compiler), mais le code généré n'est en réalité qu'un interpréteur réduit exécutant des versions encryptées des fichiers de code (.m) fournis. Le programme résultant est donc bien un exécutable indépendant de MATLAB, mais n'est en aucun cas plus rapide que le programme de départ.

Comme nous l'avons vu, les expressions à évaluer sont très longues et les syntaxes de base de MATLAB et du C/C++ diffèrent légèrement. Par exemple, 3 doit être remplacé par 3., 3.0 ou 3.0f pour ne pas être interprété comme un nombre entier, a^b doit être remplacé par `pow(a,b)`, \wedge étant un opérateur logique en C/C++ (la surcharge serait également une solution). Une stratégie de type *copier-coller-remplacer* est donc suicidaire, voire impossible.

Une première fonction a donc été écrite, `m2c()`, qui a pour tâche de traduire une expression d'opérateurs MATLAB en une expression d'opérateurs C. Par exemple

```
m2c( 'a^(1/2)+exp(b)+c^d' )
```

donne

```
sqrt(a)+exp(b)+pow(c,d)
```

Attention, la confusion entre nombres entiers et nombres à virgule flottante n'est pas traitée par `m2c`.

Nous n'en sommes pas pour autant au bout de nos peines, car si une expression MATLAB peut directement être évaluée, on ne peut pas directement compiler son équivalent en C/C++. Par ailleurs, les expressions du modèle sont complexes, mais contiennent beaucoup de sous-expressions redondantes. Pour un calcul efficace, il est donc intéressant de pouvoir précalculer ces sous-expressions. La nature du modèle fait également que les fonctions trigonométriques sinus et cosinus (projections) sont très fréquemment répétées, elles seront donc également recherchées et précalculées.

Une deuxième fonction `sym2c()` a été écrite, qui utilise entre autre `m2c()` et la fonction de recherche de sous-expressions du Symbolic Toolbox `subexpr()`.

```
[cfun, hdecl] = sym2c(varName, expr, funcname, cfun, hdecl)
```

- **varName** Cell-array contenant les noms (*strings*) des variables évaluées par `expr`.
- **expr** Expression(s) symbolique(s) (classe `sym`). La dimension de `expr` doit être la même que le nombre de strings donnés dans `varName`.
- **funcname** Nom de la fonction C/C++ à générer qui contiendra les évaluations.
- **cfun** Cell-array à compléter. Chaque élément de `cfun` contient une ligne de code du fichier `.cpp` (code source) à générer sous la forme d'un string.

- **hdecl** Cell-array à compléter. Chaque élément de **hdecl** contient une ligne de code du fichier **.h** (fichier header) à générer sous la forme d'un string.

Cette fonction peut-être utilisée pour générer des fichiers de code source compilable.

2.1 Exemple de conversion

On donne ici un exemple d'un script MATLAB pour générer du code source C/C++ à partir d'expressions symbolique. Ce code est fourni dans le fichier **CconversionExample.m**.

```
clear all;

syms x y alpha beta          % variables
syms a b c real

%define (example) symbolic expressions:
symExpr1 = 2*sin(alpha)^2 + (1+y)^(3/2) + (5/4)*(1+2*sin(alpha)^2)
                                % First C/C++ function to be generated

%example with vectorial symbolic variable:
symExpr21 = solve(sym('x^3 + a*x^2 + b*x + c')) %result
% is a sybolic expression with 3 elements (the 3 solutions to the
% equation). The equations are pretty complicated and contains
% common (repeated) subexpressions.
symExpr22 = x + y + cos(alpha) % second expression (scalar) for the
                                % second C/C++ function to be
                                % generated

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generate C/C++ files from these expressions %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% A .cpp source file and a .h header file must be generated.
% All variables in C/C++ code are assumed to be part of a
% datastructure (d) of type datas.
% Each code line is stored as a string in a cell-array, cfun for
% .cpp file and hdecl for the .h header file.

tic
disp('Starting C++ code generation')

clear cfun;
clear hdecl;

%.cpp:
cfun{1}=[ '#include "conversionDemo.h" \n' ...
          '#include "toBeIncluded1.h" \n' ...
          '#include "toBeIncluded2.h" \n\n']; %files your code needs
cfun{2}=[ '// File generated on ' date ...
          ', Basile.Graf@epfl.ch \n\n\n'];

%.h:
hdecl{1}=[ '#ifndef CONVERSION_DEMO_HDEF \n'
           '#define CONVERSION_DEMO_HDEF \n \n\n'];
           %preprocessor constant, !!! MUST be a different name
           %for each file you create !!!
hdecl{2}=[ '// File generated on ' date ...
```



```

', Basile.Graf@epfl.ch \n\n\n'];

%function for symExpr1:
varName = {'symExpr1'};
expr = {symExpr1};
[cfun, hdecl] = sym2c(varName, expr, 'cfunction1', cfun, hdecl);

%function for symExpr21 and symExpr22:
varName = {'symExpr21', 'symExpr22'};
expr = {symExpr21, symExpr22};
[cfun, hdecl] = sym2c(varName, expr, 'cfunction2', cfun, hdecl);

hdecl{length(hdecl)+1}=['#endif \n']; %%endif preprocessor line
                                     %at end of .h

%write to .cpp:
fid = fopen('conversionDemo.cpp', 'wt');
for k = 1:length(cfun)
    fprintf(fid, cfun{k});
end
fclose(fid);

%write to .h
fid = fopen('conversionDemo.h', 'wt');
for k = 1:length(hdecl)
    fprintf(fid, hdecl{k});
end
fclose(fid);

toc
disp('C++ code generation finished') %be happy

```

L'évaluation de l'expression `symExpr1` se fera donc dans une fonction nommée `cfunction1()` et le résultat sera stocké dans la variable nommée `symExpr1`. La fonction nommée `cfunction1()` évaluera les expressions `symExpr21` et `symExpr22` et stockera les résultat dans `symExpr21[0]` à `symExpr21[2]` et `symExpr22`.

Vu le grand nombre de variables d'entrée/sortie, on créera une structure de données de nom `datas` et c'est un pointeur vers celle-ci qui sera passé aux fonctions C/C++. Dans notre exemple, la structure pourrait-être écrite comme suit :

```

struct datas
{
    double x, y;
    double alpha, beta;
    double a, b, c;
};

```

2.2 Codes générés

Dans notre exemple, on obtient les codes suivants.

`conversionDemo.h :`

```

#ifndef CONVERSION_DEMO_HDEF
#define CONVERSION_DEMO_HDEF

```

```

void cfunction1(datas *d);
void cfunction2(datas *d);
#endif

```

[illegible]

Il est à noter que dans cet exemple, l'apparition de la variable complexe i poserait un problème, la classe des nombres complexes n'existant pas par défaut dans C/C++. Mais ceci n'arrive pas dans le cas d'un modèle physique tel que l'hydroptère. Les codes générés dans le cas de l'hydroptère sont évidemment bien plus conséquents et deviennent tout à fait illisibles...

3 Utilisation du programme principal : *hydroMEX3*

On présente ici une description du programme principal *hydroMEX3* (c'est à dire le programme C/C++ compilé et exécutable depuis MATLAB) qui se présente sous la forme d'un exécutable MEX. Un MEX (Matlab EXecutable) est une librairie de fonctions à linkage dynamique (.dll sous Windows) que MATLAB peut exécuter et qui se comporte comme une fonction MATLAB normale. L'avantage est qu'elle peut-être écrite en C/C++, qu'elle est précompilée et qu'elle est donc beaucoup plus rapide qu'un script MATLAB habituel. Un MEX peut également profiter de toutes les possibilités qu'offre C/C++, ce dont on fait usage dans le programme *hydroMEX3* pour l'affichage 3D en OpenGL et pour l'intégration du système d'équations différentielles partielles avec une librairie spécialisée (CVODE de Sundials [4]).

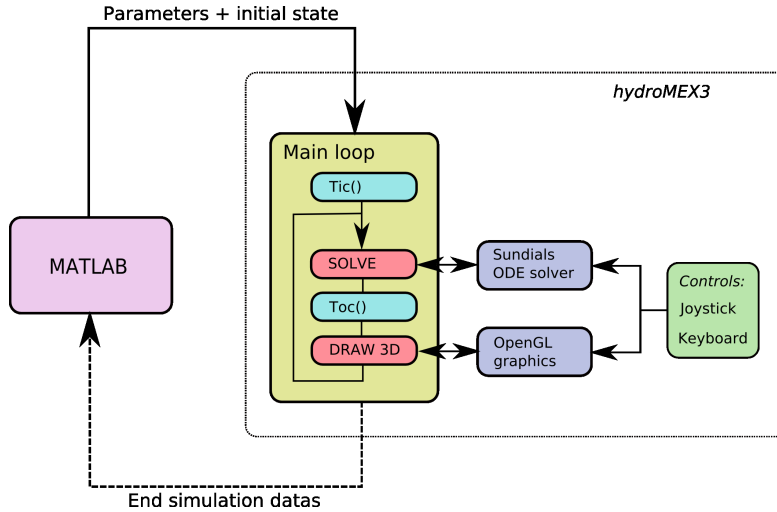


FIG. 3 – Structure du programme *hydroMEX3*

La figure 3 donne la structure générale du programme. Le programme est lancé depuis le prompt MATLAB et en reçoit divers paramètres ainsi que les conditions initiales du système. Ensuite, la boucle principale tourne, simule et affiche de manière indépendante de MATLAB. Lorsque la simulation est terminée et selon les paramètres passés, la fenêtre d'affichage est fermée et les résultats de simulation sont retournés ou non au workspace de MATLAB (pour d'éventuelles analyses supplémentaires).

3.1 Lancer le programme

Il faut tout d'abord définir les structures de paramètres et le vecteur de conditions initiales. Ces données sont par exemple générées par le script `Main.m` qui calcule un point d'équilibre pour les conditions initiales et des dimensions adaptée pour les dimensions de la machine qui sont à préciser. Ces paramètres

peuvent avantageusement être sauves dans un fichier `.mat`.

On peut également définir soi-même un vecteur de condition initiales, par exemple :

```
>> X0 = [0 0 1.9410 0.01400 0 0 20 0 0 0 0 0];
```

Une structure de paramètres, par exemple :

```
>> P.reltol = 1e-6;
>> P.abstol = 1e-4;
>> P.param = [1.9613 ... % x foil position
              0.0017... % tan(cal angle of yaw)
              1.9410...
              0.0140...
              0.2291... % tan(cal angle of sail)
              1.2938... % G-shift
              42.5347... % bridge-foil angle (deg)
              20...
              2.0915... % wind angle
              1];      % z foil position
```

Une structure de paramètres pour le joystick, par exemple :

```
>> jP4.doX: 1 % use joystick x-axis, 0: keyboard
>> jP4.doY: 1 % use joystick y-axis, 0: keyboard
>> jP4.doZ: 1 % use joystick z-axis, 0: keyboard
>> jP4.doR: 1 % use joystick r-axis, 0: keyboard
>> jP4.dirX: -1 % inverse positive joystick x direction
>> jP4.dirY: 1 % do not inverse
>> jP4.dirZ: -1 % inverse
>> jP4.dirR: -1 % inverse
```

On peut ensuite lancer le programme :

```
>> [sim_out, t_sim_out, numStep] = hydroMEX3(20,X0,P,jP4);
>> clear hydroMEX3
```

Le premier paramètre donné est le temps de simulation désiré, ici, 20 secondes. Si le temps de simulation désiré est 0, la simulation est lancée et ne s'arrête que lorsque la fenêtre graphique est fermée; dans ce cas, aucun résultat de simulation n'est retourné. Les trois derniers paramètres sont ceux décrits plus haut. Le dernier paramètre (joystick) est optionnel (valeurs par défaut). Les paramètres retournés sont : une matrice (`sim_out`) contenant le vecteur d'état à chaque instant défini par le vecteur temps (`t_sim_out`). `numStep` donne le nombre total de pas de simulation effectués par le solver (alors que les pas retournés sont ceux correspondant à chaque image affichée).

3.2 Utiliser le programme

La figure 4 est une copie d'écran du programme en action.

En haut à gauche se trouve une girouette indiquant la direction du vent (flèche rouge) et celle de la baume (jaune). En bas à droite se trouve une cible

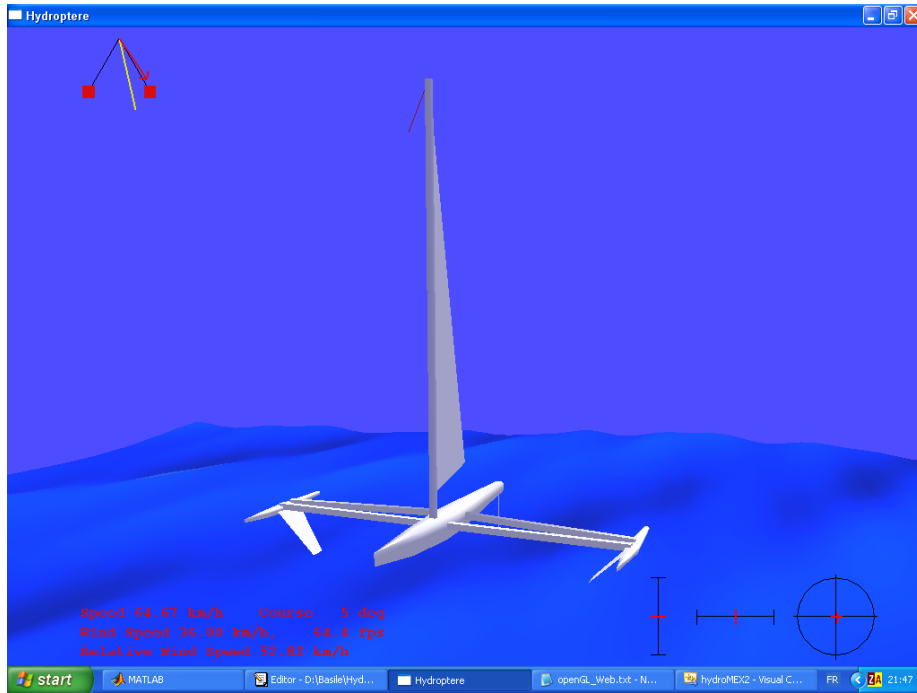


FIG. 4 – *hydroMEX3* en action

indiquant la position des deux axes principaux du joystick (profondeur et différentiel du calage des foils) ainsi que deux sliders, un horizontal pour la commande de direction (lacet) et un vertical pour le calage commun des foils. En bas à gauche apparaissent certaines informations importantes.

Contrôle par le clavier :

- 'Q' et 'V' : Tourner la caméra autour du bateau sur un plan horizontal.
- 'E' et 'R' : Tourner la caméra autour du bateau sur un plan vertical.
- 'T' et 'Z' : Rapprocher / éloigner la caméra du bateau.
- 'X' : Inverser la direction positive de l'axe X du joystick.
- 'Y' : Inverser la direction positive de l'axe Y du joystick.
- 'V' : Inverser la direction positive de l'axe Z du joystick.
- 'C' : Inverser la direction positive de l'axe R du joystick.
- '1' : Utiliser le joystick ou le clavier pour l'axe X.
- '2' : Utiliser le joystick ou le clavier pour l'axe Y.
- '3' : Utiliser le joystick ou le clavier pour l'axe Z.
- '4' : Utiliser le joystick ou le clavier pour l'axe R.
- ↑ et ↓ : Axe X lors du pilotage au clavier.
- ← et → : Axe Y lors du pilotage au clavier.
- 'M' et 'N' : Axe Z lors du pilotage au clavier.
- 'H' et 'B' : Axe R lors du pilotage au clavier.
- 'K' et 'L' : Relâcher / ramener la baume (angle maximal).

'8' et '9' : Moins / plus de vent.
'6' et '7' : Moins / plus de vagues.

Le programme retourne à MATLAB si l'on ferme la fenêtre graphique.

4 Documentation du code source

Une documentation complète du code source de *hydroMEX3* a été réalisée [7]. Elle se présente en deux versions : une version HTML et une version PDF (LateX). Il est conseillé de consulter la documentation HTML (les deux versions contiennent références croisées par hyperliens et graphes structurels, mais la version HTML est passablement plus claire). Les deux versions se trouvent dans les sous-répertoires suivants :

`/doxygen/html/index.html`

`/doxygen/latex/refman.pdf`

Références

- [1] Modèle MATLAB de l'Hydroptère ayant servi de base à ce projet
Sébastien Gros,
EPFL
- [2] Classical Mechanics,
Herbert Goldstein,
Addison-Wesley
- [3] The Waite Group's C++ Primer Plus,
Stephen Prata
Sams Publishing
- [4] Sundials ODE solver documentation :
User Documentation for cvode v2.4.0
Alan C. Hindmarsh and Radu Serban
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
[http ://www.llnl.gov/casc/sundials/documentation/cv_guide.pdf](http://www.llnl.gov/casc/sundials/documentation/cv_guide.pdf)
- [5] Neon Helium's online OpenGL tutorial,
[http ://nehe.gamedev.net/](http://nehe.gamedev.net/)
- [6] OpenGL Programming Guide : The Official Guide to Learning OpenGL,
Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner,
OpenGL Architecture Review Board
- [7] Manuel du programme de documentation Doxygen,
Dimitri van Heesch,
[http ://tesla.desy.de/doocs/doxygen-1.3/doxygen_manual-1.3.pdf](http://tesla.desy.de/doocs/doxygen-1.3/doxygen_manual-1.3.pdf)
Doxygen download (Linux, Win, Mac OSX,...) :
[http ://www.stack.nl/~dimitri/doxygen/download.html](http://www.stack.nl/~dimitri/doxygen/download.html)