MIRIAD Technologies

Compilation de mex-fonctions qui utilisent la librairie C++ Matlab

Développement de codes algorithmiques

Jérôme Lacaille

Maître de Conférences
Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex
e-mail: lacaille@cmla.ens-cachan.fr
site: www.cmla.ens-cachan.fr/~lacaille

Responsable Recherche et Développement
Miriad Technologies
8, avenue Hoche
75008 Paris
e-mail: jerome.lacaille@miriadtech.com
site: www.miriadtech.com

Vendredi 26 Avril 2002
(20 pages)

Résumé

Les librairies C++ fournies par MathWorks permettent de développer des applications scientifiques efficaces sans avoir à se soucier des aspects informatiques pénibles comme la gestion de la mémoire ou des exceptions. Malheureusement la librairie C++ Matlab ne permet que de produire des exécutables stand-alones. Or pour des raisons évidentes de facilité de prototypage, l'utilisation de mex-fonctions sous l'interpréteur Matlab est incomparable. De plus, certaines toolboxes ne compilent pas nécessairement et tous les utilisateurs de Matlab n'ont pas non plus besoin de bibliothèque graphique en dehors des phases de prototypage.

Notons par ailleurs que les mex-fonctions peuvent elles aussi être incluses au sein d'applications stand-alones développées après coups. Il est donc vraiment dommage que MathWorks ne propose pas de solutions à ce problème.

Or il existe une solution. L'incompatibilité entre la librairie C++, la librairie C (avec laquelle il est possible de faire des mex-fonctions) et l'interpréteur Matlab peut être levée. Cette manipulation n'a pas été prévue par MathWorks, et il faut intervenir légèrement sur les options standards de compilation et recompiler la librairie "`libmmfile`" en ajoutant les fonctionnalités lui permettant d'être compatible à la fois avec l'interpréteur et le compilateur en mode stand-alone.

Je propose dans cet article de présenter les quelques étapes techniques permettant de comprendre le fonctionnement interne du compilateur, de l'interpréteur et de la librairie C++. Je donne ensuite un exemple d'utilisation du C++ dans une mex-fonction.

Je fournis également en libre accès une fonction "`cppmex.m`" qui a le même rôle que le "`mex.m`" de Matlab mais qui permet de définir des mex-fonctions en C++ dont le prototype devient :

```
void cppMexFunction( int nlhs, mwArray mlhs[],
                     int nrhs, const mwArray mrhs[])
```

Le code de "`cppmex`" ainsi que les documentations qui l'accompagnent peuvent être téléchargées depuis la page Web suivante :

http://lacaille.jerome.online.fr/matlab

Contraintes

Pour l'instant les codes développés utilisent le compilateur Visual C++ sous Windows, et la version 6.1 de Matlab. Une étude est encore à mener pour réaliser des outils compatibles sous Unix ou avec la librairie Cygwin et le compilateur "`gcc`".

Nomenclature

Ce document présente des morceaux de codes C, C++ et Matlab. Un jeu de couleurs est adopté pour les listings :

- J'écris en ***bleu*** les mots clés du langage utilisé.
- Les fonctions typiquement Matlab sont coloriées en ***vert***, qu'il s'agisse des fonctions du langage ou de celles des librairies C et C++.
- Finalement, les nouvelles fonctions spécifiques à "`cppmex`" sont coloriées en ***rouge***.

SOMMAIRE

A Introduction **4**
A.1 L'API Mex et la librairie C++ 4
A.2 Exemples 5
A.3 Tentative de liaison 7

B Une solution au problème **10**
B.1 Exemple d'utilisation de la commande "cppmex" 10
B.2 Gestion de la mémoire et des erreurs 11
B.3 Pour remplacer la librairie "libmmfile" 13
B.4 Interception des exceptions 15
B.5 Dérivation des flux de sortie standard 16
B.6 La commande "cppmex" 17

Index **19**

A INTRODUCTION

MathWorks propose deux outils très utiles dans sa distribution de Matlab.

– Le premier est une API[1] permettant d'étendre l'interpréteur. C'est la notion de mex-fonction.
– Le second est la librairie C++ qui permet d'écrire des applications stand-alones en C++, en utilisant une syntaxe proche de celle du langage Matlab.

L'utilisation de ces deux objets dans les contextes définis par MathWorks est déjà une grande opportunité pour les développeurs scientifiques.

A.1 L'API MEX ET LA LIBRAIRIE C++

Quels sont les plus apportés par l'API et la librairie C++ ?

L'API des mex-fonctions :

– Cet outil permet d'accéder sous l'interpréteur Matlab à des codes développés hors Matlab. Il peut s'agir de librairies de fonctions externes ou de codes développés en interne mais n'utilisant pas le langage Matlab. L'intérêt d'encapsuler des codes C/C++ (et même Fortran) dans une mex-fonction permet au développeur Matlab d'accéder à ces fonctionnalités sans sortir de l'environnement de développement.
– Réciproquement, une fois intégrés à Matlab, les codes C/C++ bénéficient de tous les outils de développement faisant partie intégrante de l'environnement. Il est possible par exemple de créer des prototypes graphiques de logiciels en utilisant "guidë. On peut aussi analyser les résultats des algorithmes à l'aide des nombreuses toolboxes livrées avec Matlab.

La librairie C++ :

– La librairie C++ Matlab intègre une classe de matrices (entre autres objets) très développée. Or lorsque l'on crée des codes mathématiques une telle classe est souvent nécessaire. Un des avantages fondamentaux de la librairie C++ Matlab est que celle-ci est disponible, libre de port, et maintenue par MathWorks. J'ai moi-même eu à développer ma propre classe de matrices au sein de ma société. Ce travail s'est vite avéré fastidieux et très couteux en temps de maintenance.
– La librairie C++ Matlab intègre par défaut l'accès à de nombreuses librairies mathématiques standards (blas, lapack, libpack, fftw, etc.). Ce qui évite d'avoir à recoder tous ces accès.
– Le fait que cette librairie soit en C++ permet d'employer les mécanismes de gestion de mémoire et de gestion des exceptions intégrés au langage. Le résultat est que le programmeur qui utilise cette librairie n'a plus à ce soucier de ces aspects informatiques fastidieux.

1. API : Application Programming Interface.

A.2 EXEMPLES

Cette section présente un exemple d'utilisation de l'API des mex-fonctions et un essai d'intégration de code C++ développé avec la librairie Matlab.

Création d'une mex-fonction

Il est possible de compiler une fonction Matlab ("`.m`") en une mex-fonction grâce au compilateur "`mcc`". Mais ce n'est pas l'objet du présent document. Nous nous intéressons ici à l'importation au sein de l'environnement Matlab d'un code développé en C/C++.

Pour cela Matlab propose une interface très simple : la ***mexFunction***. Il s'agit d'un point d'entrée dans une librairie dynamique. Pour créer une telle fonction, il suffit d'écrire un fichier C contenant le prototype de ce point d'entrée et de le compiler à l'aide de l'utilitaire ***mex***.

Exemple :

Supposons que nous disposions d'une fonction interne "`aber`" permettant de supprimer les valeurs aberrantes d'un signal.

```
void aber( double * out, double * in, unsigned len,
           unsigned window, double alpha);
```

Cette fonction accepte un vecteur de doubles "in" en entrée, et remplit un vecteur de doubles "out" de même longueur "len". Deux paramètres sont passés : une taille de fenêtre mobile "window" et un coefficient "alpha".

La fonction "`aber`" bénéficie d'une implémentation en code natif (C ou C++) car pour réaliser sa tâche elle doit faire évoluer une fenêtre mobile le long du signal, et ceci plusieurs fois sur l'ensemble du vecteur. Une telle implémentation en Matlab nécessiterait une imbrication de boucles `for` qui conduirait à un code très inefficace. Par la suite nous supposerons que la fonction "`aber`" est placée dans le fichier C "`libaber.c`" (ou "`libaber.cpp`" en C++).

Pour appeler cette fonction sous Matlab il suffit de créer une fonction passerelle ***mexFunction*** qui appelle "`aber`". Nous disposons cette fonction dans le fichier "`aber.c`":

Fichier "aber.c" :

```
#include <mex.h>

// Prototype de la fonction algorithmique.
void aber( double * out, double * in, unsigned len,
           unsigned window,double alpha);

void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
  // Tests des paramètres.
  if (nrhs != 3)
    mexErrMsgTxt("Need exactly three input arguments!");
  if (mxGetM(prhs[1]) * mxGetN(prhs[1]) != 1)
```

```
    mexErrMsgTxt("Windows argument must be scalar.");
  if (mxGetM(prhs[2])*mxGetN(prhs[2])!= 1)
    mexErrMsgTxt("Coefficient must be scalar.");
  // On utilise mexErrMsgTxt() pour renvoyer les messages d'erreur.

  // Initialisation du vecteur de sorties comme étant égal au vecteur d'entrées.
  plhs[0] = mxDuplicateArray( prhs[0]);

  // Appel de la fonction algorithmique.
  aber( mxGetPr(plhs[0]), mxGetPr(prhs[0]),
        mxGetM(prhs[0])*mxGetN(prhs[0]),
        (int) mxGetScalar(prhs[1]), mxGetScalar(prhs[2]));
}
```

Ensuite il suffit de compiler les deux fichiers ensemble en prenant soin de placer "`aber.c`" en premier pour créer la DLL "`aber.dll`":

```
» mex aber.c libaber.c
```

La DLL produite est une mex-fonction qui peut être appelée depuis l'interpréteur Matlab :

```
» x = sin(0:pi/24:6*pi)' ;
» n = length(x) ;
» id = fix(rand(10,1)*n)+1 ;
» x(id) = 10*ones(size(id)).*sign(randn(size(id)));
» plot([x , aber(x,10,2)]) ;
```

Si le code de la fonction "`aber`" est écrite en C++, alors il faut écrire la fonction passerelle ***mexFunction*** en C++ (par exemple dans un fichier d'extension "`.cpp`"), tout en précisant que son interface est en C :

Fichier "aber.cpp" :

```
#include <mex.h>  // Définition des fonctions de l'API.
#include "aber.h" // Là où est décrite la classe contenant aber().

extern "C" void mexFunction( int nlhs, mxArray *plhs[],
                             int nrhs, const mxArray *prhs[])
{
  . . .
}
```

Mais si le code de "`aber`" utilise la librairie C++ Matlab (les objets ***mwArray***) alors il est impossible de créer une mex-fonction aussi simplement.

Utilisation de la librairie C++ Matlab

La librairie C++ facilite le travail des développeurs algorithmiques. Supposons que notre fonction "`aber`" utilise cette librairie. Proposons ci-dessous une nouvelle version simplifiée de l'implémentation de la fonction "`aber`" :

Fichier "libaber.cpp" :

```
#include <matlab.hpp> // Définitions des objets de la librairie C++
```

```
mwArray aber(const mwArray &X, int window,double alpha)
{
  int nrows = X.Size(1);
  int ncols = X.Size(2);
  mwArray Y(X);   // Copie de l'entrée.

  // Boucle sur les colonnes.
  for(int col=1; col<=ncols; col++)
  {
    int n;
    do     // Tant qu'il reste des valeurs aberrantes.
    {
      n=0;
      for(int row=2; row<=nrows-1; row++)
      {
        // Fenêtre mobile.
        mwIndex rows(max(1,row-window),min(nrow,row+window));
        mwArray m = mean(X(rows,col));
        if (tobool(abs(X(row,col)-m)>alpha*std_func(X(rows,col))))
        {
          Y(row,col) = (X(row-1,col)+(X(row+1,col))/2;
          n++;
        }
      }
    } while(n);
  }
}
```

Cette fonction est facile à appeler depuis un code C++ : si `X` est un ***mwArray***, il suffit d'écrire dans un code C++ quelque chose comme

```
mwArray Y = aber(X,10,2);
```

mais le prototypage de notre fonction ***aber*** depuis l'interpréteur Matlab n'est pas immédiat. Nous allons voir dans la section suivante qu'une tentative simple pour réaliser une mex-fonction à l'aide de ce fichier "`aber.cpp`" n'apporte pas les résultats attendus. Nous donnons finalement une solution dans la section B.

A.3 TENTATIVE DE LIAISON

Si ces deux objets proposés par MathWorks sont si utiles il leur manque un aspect assez évident : comment les relier. Je n'ai trouvé aucune information permettant de créer une mex-fonction qui utiliserait un code comme celui développé ci-dessus.

On pourrait penser qu'il suffit d'adapter une passerelle capable de faire l'interface entre la

librairie C++ et la ***mexFunction*** :

Fichier "aber.cpp" :

```
#include <matlab.hpp>  // Définitions des objets de la librairie C++
#include <mex.h>       // Définition des fonctions de l'API.

// Prototype de la fonction algorithmique.
mwArray aber(const mwArray &X, int window,double alpha);

extern "C" void mexFunction( int nlhs,mxArray *plhs[],
                             int nrhs,const mxArray *prhs[])
{
  . . .
  mwArray X(mxDuplicateArray(prhs[0])); // Recopie des entrées.
  mwArray Y = aber( X, (int) mxGetScalar(prhs[1]),mxGetScalar(prhs[2]));
  plhs[0] =mxDuplicateArray(Y.GetData()); // Recopie des sorties.
}
```

Pour compiler il est alors nécessaire de passer par une succession d'étapes permettant d'accéder aux ressources du C++ :

```
» mbuild -lang cpp -c aber.cpp
» mbuild -lang cpp -c libaber.cpp
» mex aber.obj libaber.obj ...
     D:/MatlabR12/extern/lib/win32/libmatpm.lib ...
     D:/MatlabR12/extern/lib/win32/microsoft/msvc60/libmmfile.lib
```

Notons que pour boucler l'étape de compilation et d'édition de liens il a fallu importer deux librairies : "`libmatpm.lib`" et "`libmmfile.lib`" :

- La première ("`libmatpm`") car c'est là que se trouvent les définitions des objets C++ (***mwArray***, ***mwIndex***, etc.).
- La seconde ("`libmmfile`") contient une version compilée des fonctions Matlab qui ne sont pas dans la librairie de base "`libmatlb`"

Sous l'interpréteur Matlab

Si on essaye de d'appeler la mex-fonction ainsi créée depuis l'interpréteur Matlab :

```
» y = aber(x,10,20) ;

------------------------------------------------------------
  Segmentation violation detected at Tue Apr 09 18:10:46 2002
------------------------------------------------------------
```

on voit que le résultat n'est pas très efficace. Et ceci malgré la recopie du contenu de la matrice "`Y`" par la fonction ***mxDuplicateArray***. Nous montrons dans les sections suivantes comment résoudre ce problème.

Dans un fichier ".m" compilé en stand-alone

Ce qui apparaît très étonnant c'est que si l'on compile un programme stand-alone qui appelle la DLL "aber.dll" construite par cette méthode, tout se passe pour le mieux. Voici un tel fichier "main.m":

Fichier "main.m":

```
function main(window,len)

if ischar(window)
  window =str2num(window);
end

if ischar(len)
  window =str2num(len);
end

x = sin(0:pi/24:6*pi)';
n = length(x);
id = fix(rand(10,1)*n)+1;
x(id)=10*ones(size(id)).*sign(randn(size(id)));
y = aber(x,window,len);
r = std(y);
fprintf('std(x)=%g std(y)=%g\n', std(x), std(y));
```

La compilation et l'exécution sont immédiates. Il faut juste penser à créer l'exécutable "main.exe" en utilisant la librairie C++ Matlab (option "-p" du compilateur ***mcc***).

```
» mcc -p main
» !main 10 2
std(x)=2.71776  std(y)=0.706644
»
```

B
UNE SOLUTION AU PROBLÈME

Même s'il est vrai que MathWorks a fait de grands efforts pour faciliter la programmation en C avec les librairies Matlab, l'utilisation de la structure ***mxArray*** demande une maîtrise non négligeable de la gestion mémoire et des erreurs. En parallèle, la sur-couche C++ avec son objet ***mwArray*** permet de s'absoudre de tous les problèmes informatiques pour n'avoir à se soucier que de l'algorithmique.

Matlab permet d'écrire des mex-fonctions en C mais pas l'utilisation de la sur-couche C++. Il semble en effet qu'il existe une incompatibilité entre les bibliothèques C++ et l'interpréteur Matlab qui ne bénéficient pas du même type de gestion mémoire, ni du même protocole d'accès aux fonctions contenues dans `"libmmfile"`

Le code ***cppmex*** permet de relier la librairie C++ Matlab et l'API des mex-fonctions. Son principe est simple : séparer les zones mémoires entre la mex-fonction et la librairie C++, intercepter les exceptions, et modifier la librairie `"libmmfile"` pour qu'elle devienne compatible avec l'interpréteur.

La section suivante présente un exemple d'utilisation de cette fonction. Finalement, je détaille les procédures adoptées pour obtenir le résultat souhaité.

B.1 EXEMPLE D'UTILISATION DE LA COMMANDE "CPPMEX"

Pour rester dans la norme des mex-fonctions, je propose que le point d'entrée de la cppmex-fonction soit de type analogue à ***mexFunction***. Nous définissons ainsi une fonction passerelle à notre fonction `"aber"` :

Fichier "aber.cpp" :

```
#include <matlab.hpp> // Définitions des objets de la librairie C++

// Prototype de la fonction algorithmique.
mwArray aber(const mwArray &X, int window,double alpha);

// Point d'entrée C++.
void cppMexFunction( int nlhs,mwArray mlhs[],
                     int nrhs,const mwArray mrhs[])
{
  if (nrhs!= 3)
    error("Need exactly three input arguments!");
  if (mrhs[1].EltCount()!= 1)
    error("Windows argument must be scalar.");
  if (mrhs[2].EltCount()!= 1)
    error("Coefficient must be scalar.");
```

```
  int window =mxGetScalar(mrhs[1].GetData());
  double alpha =mxGetScalar(mrhs[2].GetData());
  mlhs[0] =aber( mrhs[0], window, alpha);
}
```

Notez l'utilisation de la fonction ***error*** de la librairie C++ et le passage des arguments sous la forme de tableaux d'objets de type ***mwArray***.

Nous utilisons ensuite la commande ***cppmex*** pour créer la mex-fonction :

```
» cppmex aber.cpp libaber.cpp
» y = aber(x,10,20) ;
» plot([x,y])
```

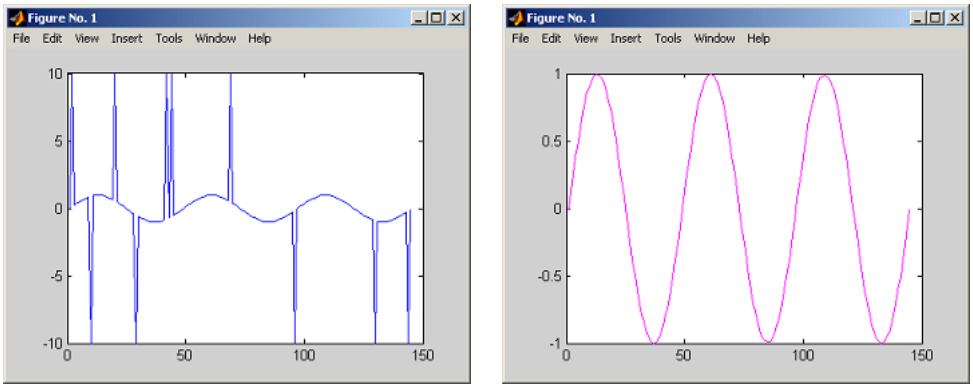

FIG. 1 – *Le signal et sa version sans valeurs aberrantes.*

Le résultat de cette compilation produit une DLL "`aber.dll`" exactement comme si on avait utilisé la commande ***mex***.

B.2 GESTION DE LA MÉMOIRE ET DES ERREURS

La ***mexFunction*** est codée dans un fichier template auxiliaire : "`cppmex.cpp`" Elle se charge des copies mémoires, de l'appel de la passerelle C++ ***cppMexFunction*** et de l'interception des exceptions.

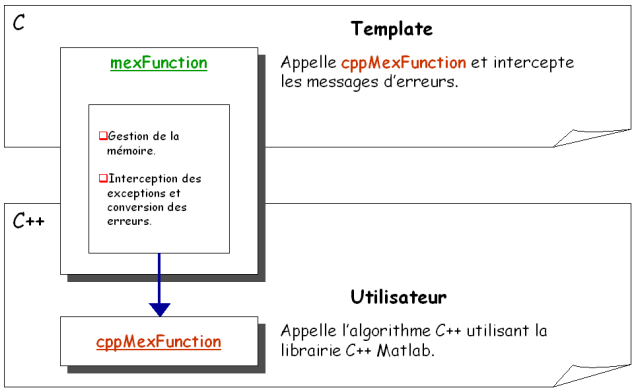

FIG. 2 – *Enchaînement des passerelles.*

Fichier "cppmex.cpp" :

```
#include <matlab.hpp> // Définitions des objets de la librairie C++
#include <mex.h>      // Définition des fonctions de l'API.

extern "C" void mexFunction( int nlhs, mxArray *plhs[],
                             int nrhs, const mxArray *prhs[])
{
  // Un tampon pour la sauvegarde des messages d'erreurs.
  strstream errmsg;

  // Création des sorties.
  // Le premier élément existe toujours, il correspond à la variable "ans".
  int k = nlhs?nlhs:1;
  mwArray *mlhs =new mwArray[k];

  // Conversion des entrées.
  mwArray *mrhs = NULL;
  if (nrhs)
  {
    mrhs =new mwArray[nrhs];
    for(int i=0; i<nrhs; i++)
    {
      mwArray U(mxDuplicateArray(prhs[i])) // Constructeur de conversion.
      // La duplication évite la destruction des entrées par la librairie C++.
      mrhs[i] = U;
    }
  }

  try
  {
    // Appel de la fonction utilisateur.
    cppMexFunction( nlhs, mlhs, nrhs, mrhs);
```

```
    // Récupération des sorties.
    for(int i=0; i<k; i++)
      plhs[i] = mxDuplicateArray( mlhs[i].GetData());
  }
  catch( mwException &ex)
  {
    // Interception des erreurs.
    errmsg<< ex;

    // En cas d'erreur les retours sont des matrices vides.
    for(int i=0; i<k; i++)
      plhs[i] = mxCreateDoubleMatrix(0,0,mxREAL);
  }

  // Destruction des sorties C++ intermédiaires.
  delete [] mlhs;

  // Retransmission des messages d'erreurs.
  if (errmsg.pcount())
    mexErrMsgTxt(errmsg.str());
}
```

Le coût à payer pour que ce système fonctionne est une double duplication des zones mémoires échangées entre les mondes C et C++.

La fonction ***cppMexFunction*** est le point d'entrée utilisateur qui permet de développer le code algorithmique en utilisant la totalité de la puissance de la librairie C++ Matlab.

Point d'entrée utilisateur :

```
#include <matlab.hpp>

void cppMexFunction( int nlhs, mwArray mlhs[],
                     int nrhs, const mwArray mrhs[])
{
  . . .
}
```

Notez le fait que les paramètres sont directement passés sous la forme de ***mwArray***. Les messages d'erreurs peuvent être produits par la fonction ***error*** car ils seront interceptés dans la boucle ***try***/***catch*** de la ***mexFunction***. L'entête "`matlab.hpp`" est nécessaire pour l'utilisation de la librairie C++.

B.3 POUR REMPLACER LA LIBRAIRIE "LIBMMFILE"

L'opération précédente suffit si l'on compile avec les librairies adéquates ("`libmatpm.lib`" et "`libmmfile.lib`"), mais quelques fonctions ne passent pas en mode interprété (c'est ce

que l'on a vu section A.3). En effet la compilation des fichiers ".m" de Matlab dans la librairie "libmmfile.lib" n'est prévue que pour une utilisation en mode stand-alone. Il faut donc intercepter les appels à cette librairie.

Pour cela on utilise le fichier "libmmfile.mlib" qui contient un descriptif sommaire de l'ensemble des fonctions à problème. Ce fichier se trouve dans %MATLAB%/extern/include

Un bout du fichier "libmmfile.mlib" :

```
mean {                  /* M function name */
  nin      = 2;         /* number of M inputs */
  nout     = 1;         /* number of M outputs */
  scope    = global;    /* where function defined */
  cname    = mlfMean;   /* mxArray in and out */
  nargout  = 0;         /* nargout not used */
}
```

A partir de ce fichier, il est facile de construire un fichier temporaire "pmfile.m" qui contient un appel pour chacune des fonctions de la "libmmfile"

Un morceau du fichier "pmfile.m" :

```
function varargout = pmfile(varargin)
...
varargout{1} = mean(varargin{1},varargin{2}) ;
varargout{1} = median(varargin{1},varargin{2}) ;
[varargout{1},varargout{2}] = meshdom(varargin{1},varargin{2}) ;
...
```

Ensuite, grâce à la commande

```
» mex -x pmfile
```

on va produire les interfaces nécessaires à la construction d'une nouvelle librairie "libpmfile.cpp"

La compilation précédente à produit une DLL absolument inutile : "pmfile.dll" Ce qui nous intéresse uniquement, c'est le code intermédiaire d'interface mex : "pmfile_mex.c" Dans ce fichier on trouvera trois types de fonctions C par fonction Matlab. Prenons l'exemple de la moyenne "mean" :

- ***static mxArray * Mmean()*** correspond au callback vers Matlab dans le cas d'une exécution sous la forme de mex-fonction depuis l'interpréteur. Dans le cas d'une fonction sans retour (comme tic) le type ***mxArray**** est remplacé par un ***void***.
- ***mxArray * mlfMean()*** est l'interface standard à Mmean appelé comme une fonction de la bibliothèque C. Seule cette interface doit être codée dans "libmmfile" car c'est elle qui sert dans la librairie C.
- ***void mlxMean()*** est une passerelle employée par feval

Ces trois types de fonctions sont alors recopiées dans un fichier "libpmfile.cpp"

Fichier "libpmfile.cpp" :

```
#include <matlab.hpp>

...
```

```
static mxArray * Mmean(int nargout_,mxArray * x, mxArray * dim)
{
 ...
}

mxArray * mlfMean(mxArray * x, mxArray * dim)
{
  ...
}

void mlxMean(int nlhs,mxArray * plhs[],
             int nrhs,mxArray * prhs[])
{
  ...
}

...
```

Finalement ce fichier est compilé en un objet par la commande :

```
» mbuild -lang cpp -c libpmfile.cpp
```

Ce fichier "`.obj`" sera inclus juste avant la librairie "`libmatpm.lib`" dans la commande de compilation finale qui devient :

```
» mex aber.obj libaber.obj libpmfiles.obj ...
     D:/MatlabR12/extern/lib/win32/libmatpm.lib
```

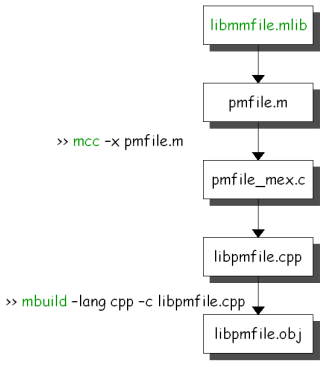

FIG. 3 – *Enchaînement des opérations.*

B.4 INTERCEPTION DES EXCEPTIONS

Deux sortes d'exceptions peuvent être lancées par un code C++/Matlab.

– Celles qui sont générées par la librairie C++ (des ***mwException*** qui peuvent être inter-

ceptées par un bloc ***try***/***catch***. Ces exceptions sont générées lorsque le programme est compilé en mode stand-alone.
– Et celles qui sont produit par l'interpréteur Matlab et qu'il est possible d'intercepter par les bloc ***mlfTry***/***mlfCatch***.

Pour intercepter simultanément ces deux types d'erreurs, nu fichier d'entête décrit les macros ***mwTry***/***mwCatch*** qui regroupe les deux types d'interception et produisent systématiquement une ***mwException***.

Fichier "`cppmex.h`" :

```
#define MLF_ENABLE_TRYCATCH
#include <matlab.hpp>
#include <mex.h>

#define mwTry \
        try { mlfTry

#define mwCatch \
        mlfCatch \
        { \
          mxArray *err = mlfLasterr(NULL) ; \
          int buflen = mxGetN(err)*mxGetM(err)+1 ; \
          char *buf = new char[buflen] ; \
          mxGetString(err,buf,buflen) ; \
          mwString str = buf ; \
          delete [] buf ; \
          throw mwException(str,__LINE__,__FILE__) ; \
        } \
        mlfEndCatch } catch
```

B.5 Dérivation des flux de sortie standard

Si l'on veut développer du code en C++, alors il est utile de proposer un couple de flux standards (***cout*** et ***cerr***) fonctionnel. Pour cela on utilise le fait que ***cout*** et ***cerr*** sont des ***ostream_withassign***, c'est à dire qu'il est possible de les remplacer par des instances dérivées de ***streambuf***.

Une classe ***mexstream*** dérivée de ***streambuf*** est placé en entête du fichier "`cppmex.cpp`" et permet de créer un ***streambuf*** qui remplace efficacement ***cout*** en imprimant les messages grâce à ***mexPrintf***. Cette classe est ensuite redérivée à nouveau en ***mexerrstream***, elle utilise plutôt ***mexWarnMsgTxt*** et sert à remplacer ***cerr***.

`Fichier "cppmex.cpp" :`

```
#include <cppmex.h>

class mexstream : public streambuf
```

```
{ ... }
class mexerrstream : public mexstream
{ ... }

...
```

Finalement les flux standards sont modifiés par assignement dans la fonction passerelle ***mexFunction***. Mais ce remplacement ne doit être fait que si la mex-fonction est exécutée sous l'interpréteur Matlab. Dans le cas ou le code est utilisé par une application stand-alone, ces flux ne doivent pas être modifié. Un bloc ***mwTry***/***mwCatch*** est donc utilisé conjointement à un appel à l'interpréteur Matlab (On utilise pour cela deux appels successif à la fonction "`echo`"). Si aucune erreur est générée, alors c'est qu'il faut changer le flux.

```
mexstream cmex;
mexerrstream errmex;

mwTry
{
  mexCallMATLAB(0,NULL,0,NULL,"echo");
  mexCallMATLAB(0,NULL,0,NULL,"echo");

  cout = &cmex;
  cerr = &errmex;
} mwCatch( mwException &ex)
  { }
```

Finalement une fonction ***cppMexErrorHandler*** est définie, et remplace le gestionnaire d'erreur par défaut :

```
extern "C" void cppMexErrorHandler(const char *msg, bool isError)
{
  if (isError)
     error(msg);
  else
     cerr << msg << endl;
}

:

  mlfSetErrorHandler( cppMexErrorHandler);
```

B.6 LA COMMANDE "CPPMEX"

Le fichier "`cppmex.m`" doit être placé dans un répertoire du PATH Matlab. Elle est accompagnée de deux sous répertoires : "`private`" qui contient la fonction auxiliaire "`cppmex_setup`"

créant les template "`cppmex.obj`" et "`libpmfile.obj`", et le sous répertoire "`templates`" dans lequel seront stockés ces fichiers objets ainsi que le source "`cppmex.cpp`"

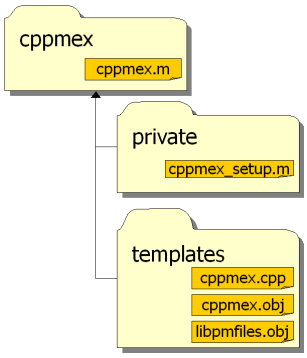

FIG. 4 – *Structure des fichiers **cppmex**.*

La syntaxe d'appel de ***cppmex*** est exactement la même que celle de ***mex*** :

cppmex `-setup`
Initialise les templates.

cppmex `[-options] base.cpp [ ... ] [librairie.lib ...]`
Compilation de *base*.dll.

Les options sont passées telles quelles à ***mbuild*** puis ***mex***. On peut ainsi produire une version débuggable de la mex-fonction avec l'option "`-g`".

Notons qu'il est aussi possible de passer des librairies statiques ("`.lib`") mais aussi des interfaces de DLL à ***cppmex***. Cela permet de regrouper plusieurs algorithmes au sein d'une même DLL et de générer plusieurs mex-fonctions pour y accéder.

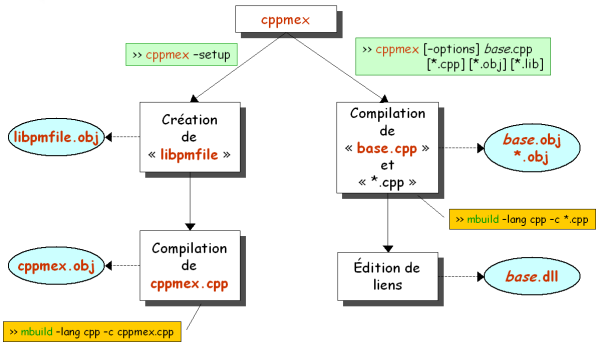

FIG. 5 – *Organigrame de traitement de **cppmex**.*

INDEX

mex-fonction, 4

RÉFÉRENCES

[1] MathWorks. *Matlab C/C++ Math Library*. Online HTML Manual, 6.1 edition.
[2] MathWorks. *Matlab compiler*. Online HTML Manual, 6.1 edition.
[3] MathWorks. *Matlab External API*. Online HTML Manual, 6.1 edition.