CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Connecting OpenFOAM with MATLAB

Developed for OpenFOAM-2.0.x
Requires: MATLAB and gcc

Author:
Johannes Palm

Peer reviewed by:
Jelena Andric

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, the content might be helpful and give some pointers to anyone who wants to know how to connect Matlab with OpenFOAM.

November 6, 2012

Chapter 1

1.1 Introduction

This tutorial describes how to connect OpenFOAM with MATLAB. The processes of how to send information to MATLAB, perform some calculations or post process and optionally return values to OpenFOAM for direct use are exemplified in this report. Simple calculation examples are used to demonstrate the procedure. The aim of the report is thus to show how to establish a working connection between MATLAB and OpenFOAM, and for that purpose the examples shown are focused on the key commands needed to establish such a connection. This tool can be used very effectively in the development stage of a model when different types of solutions can be rapidly coded within the MATLAB framework and used in an OpenFOAM solver.
This report is written as a tutorial for connecting OpenFOAM with MATLAB and uses the OpenFOAM tutorial `floatingObject` as a starting point. The `floatingObject` tutorial calculates the motion of a free floating box in water subjected to a dambreak using the `interDyMFoam` solver with dynamic meshes. The report is divided into four chapters with the first chapter being dedicated to introducing the `floatingObject` tutorial and modifying it with the addition of a linear spring restraint. Chapter two is devoted to estalishing a MATLAB connection from a C++ environment, and chapter three explains how this connection can be used to create a non-linear spring restraint on the floating object using MATLAB. The fourth chapter is an appendix containing complete source files.

1.2 The floatingObject tutorial

This section provides a brief explanation of the case setup, and tutorial results of the OpenFOAM tutorial `floatingObject` of the `interDyMFoam` solver. Start by copying the `floatingObject` tutorial to the run directory.

```
OF20x
cp -r  $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/floatingObject $FOAM_RUN
cd $FOAM_RUN/floatingObject
```

1.2.1 Case setup

Looking into the `Allrun` script the following utilities and solvers are used:

blockMesh (util.)
topoSet (util.)
subSetMesh (util.)
setFields (util.)
interDyMFoam (solver)

These will not be modified in this project and they will therefore not be gone through in detail here, but it is good to know how the case is setup.

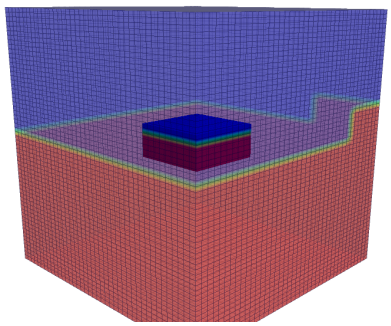

Figure 1.1: The initial geometry of the floatingObject tutorial.

In the end of the case `system/controlDict` file, the following `libs` command is found.

```
libs
(
    "libOpenFOAM.so"
    "libincompressibleRASModels.so"
    "libfvMotionSolvers.so"
    "libforces.so"
);
```

This allows the use of the functionalities of the listed libraries. In the `0.org` directory there is a field file called `pointDisplacement` which is needed for solvers using dynamic meshes and contains the displacements of all nodes in the mesh at each time. In this file, one of the boundary patches is named `floatingObject` and is of type `sixDoFRigidBodyDisplacement`, which is a function object within the `libforces.so` library. Here the dynamic properties of the floating box are specified.

```
 floatingObject
    {
        type            sixDoFRigidBodyDisplacement;
        centreOfMass    (0.5 0.5 0.5);
        momentOfInertia (0.08622222 0.08622222 0.144);
        mass            9.6;
        rhoInf          1;  // needed only for solvers solving for kinematic pressure
        report          on;
        value           uniform (0 0 0);
    }
```

The `sixDoFRigidBodyDisplacement` uses the `forces` function object to integrate the pressure along the wetted surface and calculate the motion of the body using the resulting force and torque. Quaternions are used to keep track of the present state of the body. The motion is then mapped to the individual cell faces of the surface of the body, which in turn specifies a time changing boundary condition on the velocity of the water.

1.2.2 Results

Executing the `./Allrun` script results in a 6 s simulation of the dambreak hitting the floating object.

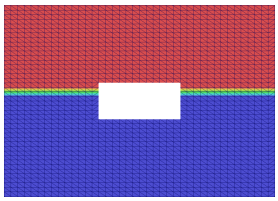

Figure 1.2: Results at plane y=0 and t=0 s of tutorial floatingObject.

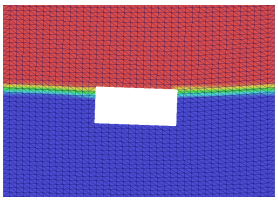

Figure 1.3: Results at plane y=0 and t=1 s of tutorial floatingObject.

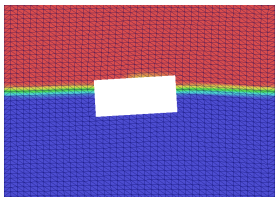

Figure 1.4: Results at plane y=0 and t=1.5 s of tutorial floatingObject.

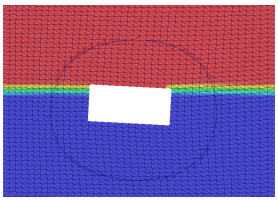

Figure 1.5: Results at plane y=0 and t=2 s of tutorial floatingObject.

Looking at the resulting heave motion of the centre of mass of the floating object over the entire simulation gives a more complete understanding.

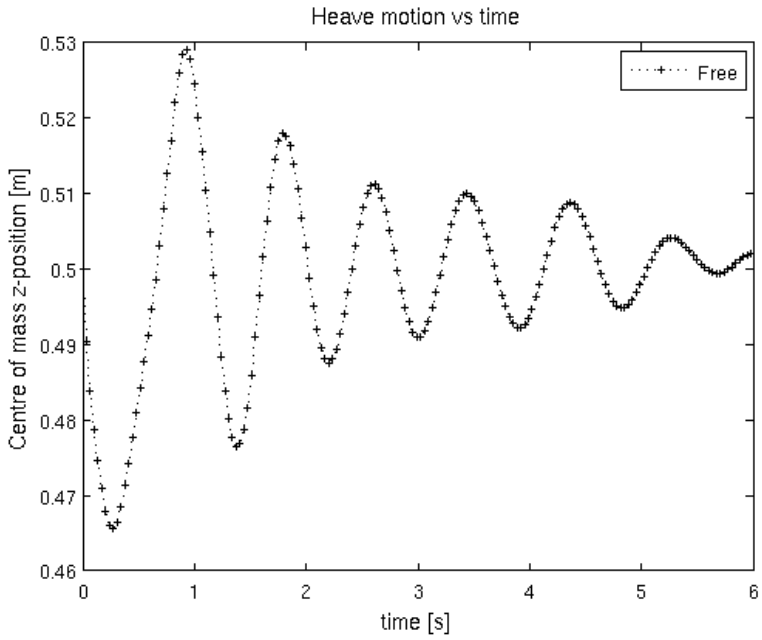

Figure 1.6: The time evolution of the heave motion of the floating object. The figure was produced using MATLAB and the heave position was saved at each time step using the scripts devloped in chapters 2 and 3.

1.3 Modifying the floatingObject tutorial

The sixDoFRigidBodyDisplacement object is prepared for constraints, e.g. preventing rotation around a specific axis or prescribing motion in a plane, and restraints, e.g. linear or rotational springs. This section deals with how to add a restraint to the tutorial.
To do this, commands will be taken from the tutorial `wingMotion2D_pimpleDyMFoam`.

```
pushd incompressible/pimpleDyMFoam/wingMotion/wingMotion2D_pimpleDyMFoam/
cd 0.org
#Open File pointDisplacement and copy restraints section#
popd
```

Go back to the `pointDisplacement` file of the floatingObject case and paste the restraints section between the `report` and the `value` statements in the file. Remove all other restraints so that only the one named `verticalSpring` remains, and change the values so that this part of the file now becomes:

```
floatingObject
    {
        type            sixDoFRigidBodyDisplacement;
        centreOfMass    (0.5 0.5 0.5);
        momentOfInertia (0.08622222 0.08622222 0.144);
        mass            9.6;
        rhoInf          1;  // needed only for solvers solving for kinematic pressure
        report          on;
        restraints{
            verticalSpring
            {
                sixDoFRigidBodyMotionRestraint linearSpring;

                linearSpringCoeffs
                {
                    anchor          (0.5 0.5 0.2);
                    refAttachmentPt (0.5 0.5 0.45);
                    stiffness       100;
                    damping         0;
                    restLength      0.25;
```

```
            }
        }
    }
    value           uniform (0 0 0);
}
```

Now, a vertical spring is acting between the body point (`refAttachmentPt`) and the global, static coordinate system point (`anchor`). The `stiffness`, `damping` and `restLength` have their straightforward meaning. Now the case can be run again using `./Allclean` and `./Allrun`.

1.3.1 Results

Here, a comparison between the heave motion of the free and the restrained floating object is presented. The spring stiffness is relatively low compared to the hydrodynamic forces acting on the body, but there is a clear restrain of the vertical motion of the centre of mass of the body.

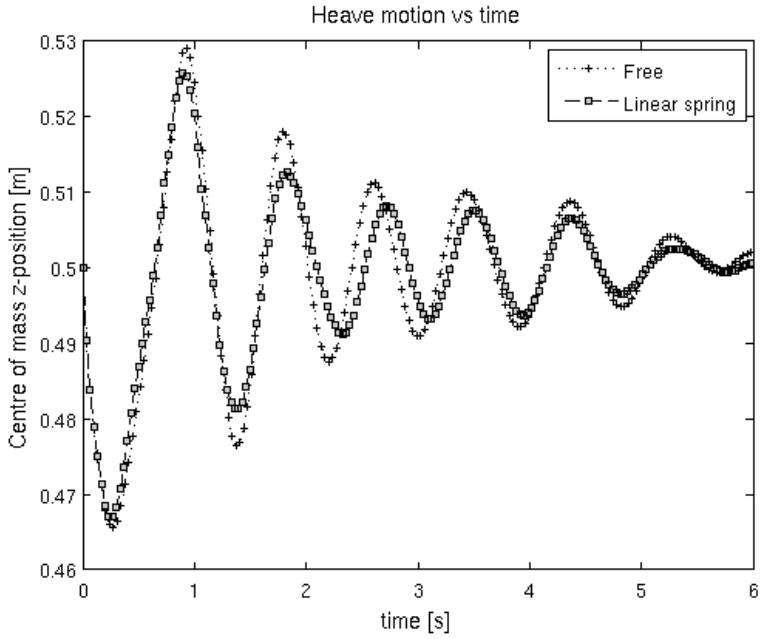

Figure 1.7: Comparison of the time evolution of the heave motion of the floating object with and without a vertical spring.

Chapter 2

2.1 Setting up a MATLAB engine script in C++

This section is dedicated to the communication between C++ and MATLAB and how to set up a MATLAB engine script in C++. It is at this stage completely independent of the OpenFOAM installation. The only things needed are functioning installations of C++, MATLAB and, in this case, the gcc compiler.

2.1.1 Communication syntax

The file `demoPipe.C` is shown below and contains code that generates a MATLAB-pipe class.

```
// Filename: demoPipe.C //

#include<iostream>
#include<cmath>
#include "engine.h"
using namespace std;

int main(){
// Open a portal to MATLAB through a pointer to an Engine object //

   Engine *eMatlabPtr=engOpen(NULL);

// Create an empty mxArray of size [1,1] //

    mxArray *aMxArray = mxCreateDoubleMatrix(1,1,mxREAL);

// Get pointer to the actual value of the mxArray //

    double *aPtr = mxGetPr(aMxArray);

// Set value of mxArray//

    aPtr[0] = 5;

// Set the value of matlab parameter a to the value of aMxArray. //

    engPutVariable(eMatlabPtr,"a",aMxArray);

// Execute commands in MATLAB //

    engEvalString(eMatlabPtr,"b=a.^2; plot(0:0.1:2*pi,sin(0:0.1:2*pi)); pause(5);");

// Collect the result from MATLAB back to the C++ code //

    mxArray *bMxArray = engGetVariable(eMatlabPtr,"b");
    double *bPtr = mxGetPr(bMxArray);

// Print the result //
    cout << "5*5 = " << bPtr[0] << endl;

// Close the pipe to Matlab //
    engClose(eMatlabPtr);
    return 0;
}
```

The mxArray data type is a complicated data structure containing much more information than the values of the matrix which are needed. Therefore the `mxGetPr(mxArrayObj)` is used to modify

the value of the cells in the array, in this case stored as pointers `*aPtr` and `*bPtr`.
The MATLAB engine object pointer `*eMatlabPtr` is used to specify which MATLAB process to use when calling the `engPutVariable`, `engEvalString` and `engGetVariable` functions. There are also other functions available for MATLAB communication and the full set is well documented on the Mathworks homepage, www.mathworks.com, however one can get quite a long way with these three routines. At the end of the program, the pipe to MATLAB is closed by calling the `engClose` function, and the MATLAB process is terminated. It is worth to note that a user defined .m-file can be executed by simply sending the name of it as the input command string to `engEvalString`. When running a case in OpenFOAM, MATLAB is opened in the main directory of the case files, and from there it will search for function- or script files.

2.1.2 Compilation instructions

The compilation of the program `demoPipe.C` is not completely straightforward. The first task is to locate the file `engine.h` and include it in the `gcc` command using the `-I` flag. It can be found in the `<matlabroot>/extern/include/` directory, where `<matlabroot>` is the installation directory of your MATLAB version.

Example flag to include search path for header files:
`-I/chalmers/sw/sup64/matlab-2011b/extern/include`

The `mxArray` data types and the `eng...` functions must be known to the program. This is done by linking the compilation to the libraries `libmx.so`, `libeng.so`. But it is not enough to just include them in the compilation using the `-l` flag, as the libaries are not located in the default library path (as set by the `LD_LIBRARY_PATH` variable of your system). Additional library paths can be specified using the `-L` flag, and the libraries needed are found in the `<matlabroot>/bin/lnxa64/` for a linux 64 system. This differs slightly depending on the architecture of the operating system.

Example flag to include library search paths and link to libraries:
`-L/chalmers/sw/sup64/matlab-2011b/bin/glnxa64 -leng -lmx -lmat`

However, it turns out that these libraries are dependent on additional libraries. To tell the compiler linker to look in the proper directory for these files at runTime one can use the `-rpath` option. This inserts the library path into the header of the executable or shared library, which then is read at runTime, and the program finds the neccessary libraries. In this case they are all found in the same directory.

Example flag to include library search path in compiled object:
`-Wl,-rpath,/chalmers/sw/sup64/matlab-2011b/bin/glnxa64`

So the final complete compilation command for this simple case is:

```
gcc -Wall -Wl,-rpath,/chalmers/sw/sup64/matlab-2011b/bin/glnxa64 \
-I/chalmers/sw/sup64/matlab-2011b/extern/include \
-L/chalmers/sw/sup64/matlab-2011b/bin/glnxa64 -leng -lmx \
demoPipe.C -o demoPipeProgram
```

which will create the executable file demoPipeProgram, executed by typing `./demoPipeProgram` in the directory of the file.

2.2 Incorporation into OpenFOAM

This section shows how to compile the MATLAB connection into a shared dynamic library, here named `myFirstMatlabPipe`, and to use this library in the compilation of a new restraint, which in this case is named `mylinearSpring`. The compilations in this section are made with the `wmake` compiler of OpenFOAM.

2.2.1 Creating a MATLAB pipe library

The following steps are rather straight forward for an experienced user of C++, however they are described here for the completeness of the tutorial. The necessary steps to create a MATLAB pipe class are basically the same as those described in the previous section.
Begin by creating a source directory for your connection pipe files.

```
mkdir -p $WM_PROJECT_USER_DIR/src/externalPipes/myFirstMatlabPipe/
```

in which the files `myFirstMatlabPipe.H` and `myFirstMatlabPipe.C` are used to create a class that initiates a connection to a MATLAB process.

```
// Filename: myFirstMatlabPipe.H //

#ifndef myFirstMatlabPipe_H
#define myFirstMatlabPipe_H

#include "engine.h"

using namespace std;

class myFirstMatlabPipe
{
        // Type definition //

        // Private data objects //
        Engine *eMatlabPtr;

        // Number of calltimes //
        int ii;

    public:
        // Constructor //
        myFirstMatlabPipe();

        // Destructor //
        virtual ~myFirstMatlabPipe();

        // Send and return a double array to a matlab script //
        virtual double matlabCallScript(const char* matlabFilename,double inputArg) const;

        // Close the pipe to MATLAB //
        virtual void close() const;
};

#endif
//************************************ END OF FILE ************************************//

// Filename: myFirstMatlabPipe.C //

#include<iostream>
#include<cmath>
#include "engine.h"
#include "myFirstMatlabPipe.H"
//
using namespace std;
// Constructor //
myFirstMatlabPipe::myFirstMatlabPipe()
{
    cout << "Matlab engine pointer initialized" << endl;
    // Create matlab engine pointer //
    eMatlabPtr=engOpen(NULL);
    // Create a matlab call no. iterator ii //
    engEvalString(eMatlabPtr,"ii=0;");
}
// Destructor //
```

```
myFirstMatlabPipe::~myFirstMatlabPipe(){};

double myFirstMatlabPipe::matlabCallScript(const char* matlabFilename,double inputArg) const
{
    // Increase iterator value //
    engEvalString(eMatlabPtr,"ii=ii+1;");

    // Create scalar mxArray object compatible with MATLAB and C++ //
    mxArray *inMxArray = mxCreateDoubleMatrix(1,1,mxREAL);
    double *inPtr = mxGetPr(inMxArray);

    // Send value of inMxArray to MATLAB //
    inPtr[0] = inputArg;
    engPutVariable(eMatlabPtr,"inputFromCpp",inMxArray);

    // Execute MATLAB script //
    engEvalString(eMatlabPtr,matlabFilename);

    // Extract value to C++ and return //
    mxArray *outMxArray = engGetVariable(eMatlabPtr,"outputToCpp");
    double *outPtr = mxGetPr(outMxArray);

    return outPtr[0];
};

void myFirstMatlabPipe::close() const
{
    engClose(eMatlabPtr);
};
//************************************ END OF FILE ************************************//
```

`myFirstMatlabPipe.C` can be compiled using the `wmake libso` command of OpenFOAM. The wmake compiler is an OpenFOAM version of gcc/g++, with several flags automatically included. In order to use it, a `Make` directory has to be created. Since the MATLAB class is specific for each type of calculation, and it might be that I want to do several different types of MATLAB calculations in the future, I choose to build the library one level up, so that the `Make`-directory is located in `$WM_PROJECT_USER_DIR/src/externalPipe/`. The corresponding files `Make/files` and `Make/options` are displayed below.

Make/files

```
myFirstMatlabPipe/myFirstMatlabPipe.C

LIB = $(FOAM_USER_LIBBIN)/libexternalMatlabPipes
```

Make/options

```
EXE_INC = \
  -Wl,-rpath,/chalmers/sw/sup64/matlab-2011b/bin/glnxa64 \
       -I/chalmers/sw/sup64/matlab-2011b/extern/include

LIB_LIBS = \
  -L/chalmers/sw/sup64/matlab-2011b/bin/glnxa64 \
  -leng \
  -lmx
```

where the resulting library will be put in `$FOAM_USER_LIBBIN` and the `-rpath` option is including the library link in the header of the library file as before.

2.2.2 Creating a new restraint

Go to `$WM_PROJECT_DIR` and copy the `linearSpring` restraint of the rigid body motion function object to your `$WM_PROJECT_USER_DIR` including the folder structure.

```
foam
cp -r --parents $WM_PROJECT_DIR/src/postProcessing/functionObjects/forces/\
pointPatchFields/derived/sixDoFRigidBodyMotion/\
sixDoFRigidBodyMotionRestraint/linearSpring $WM_PROJECT_USER_DIR
```

Go to the new directory (not printed below) and rename the folder `linearSpring` to `mylinearSpring`. Then rename all the files within the directory, and all instances of `linearSpring` to `mylinearSpring` within the files themselves.

```
mv linearSpring mylinearSpring
cd mylinearSpring
mv linearSpring.C mylinearSpring.C
mv linearSpring.H mylinearSpring.H
sed -i s/"linearSpring"/"mylinearSpring"/g mylinearSpring.C
sed -i s/"linearSpring"/"mylinearSpring"/g mylinearSpring.H
```

There is no `Make` directory in the folder, and looking for it on each level while moving up in the file tree, it can be found in `$WM_PROJECT_DIR/src/postProcessing/functionsObjects/forces`. Copy that `Make` directory to `myLinearSpring/`.

```
cp -r $WM_PROJECT_DIR/src/postProcessing/functionsObjects/forces/Make .
```

The `Make/files` file has to be changed so that only `mylinearSpring.C` is included in the file. Also, the target file must be changed to `libmylinearSpring`.
Make/files:

```
mylinearSpring.C

LIB = $(FOAM_USER_LIBBIN)/libmylinearSpring
```

In the options file, start with adding only the lnInclude directory of the old `forces` directory from which the `Make` folder was copied.
Make/options:

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/turbulenceModels \
    -I$(LIB_SRC)/turbulenceModels/LES/LESdeltas/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(WM_PROJECT_DIR)/src/postProcessing/functionObjects/forces/lnInclude

LIB_LIBS = \
    -lincompressibleTransportModels \
    -lincompressibleRASModels \
    -lincompressibleLESModels \
    -lbasicThermophysicalModels \
    -lspecie \
    -lcompressibleRASModels \
    -lcompressibleLESModels \
    -lfiniteVolume \
    -lmeshTools \
    -lsampling
```

Not all of these directories and libraries are needed, but no investigation of which ones are actually necessary has been made within this project so all flags are left included. Now do a test compilation of the `mylinearSpring`. From the `mylinearSpring` directory write:

```
wmake libso
```

This should compile without any errors.

2.2.3 Modifying the restraint

Now we want to use `mylinearSpring` to call the `myFirstMatlabPipe` library which we have already created. The following is an extraction of the modifcations made in the restraint header file. The complete file is given in section 4.1.3

```
//********************************** START OF EXTRACT **********************************//
#include "sixDoFRigidBodyMotion.H"
//---------------- Changes start --------------//
#include "myFirstMatlabPipe.H"
//----------------- Changes end ---------------//

//********************************** END OF EXTRACT ************************************//
...
//********************************** START OF EXTRACT **********************************//
        //- Rest length - length of spring when no forces are applied to it
```

```
        scalar restLength_;

    //---------------- Changes start --------------//

        //- The object containing the pipe to MATLAB -//
        myFirstMatlabPipe mlObj;

    //----------------- Changes end ---------------//

public:

    //- Runtime type information
    TypeName("mylinearSpring");
//************************************ END OF EXTRACT ***********************************//
```

The only changes made to the header file are the `#include myFirstMatlabPipe.H` statement and the declaration of the private object `mlObj` of type `myFirstMatlabPipe`. The source file has mostly been alterered within the `restrain` function, and the following is an extraction of the modified parts of `mylinearSpring.C`. The complete file is given in section 4.1.4.

```
//************************************ START OF EXTRACT ***********************************//
#include "sixDoFRigidBodyMotion.H"
//---------------- Changes start --------------//
#include "myFirstMatlabPipe.H"
//----------------- Changes end ---------------//

//************************************ END OF EXTRACT ***********************************//
...
//************************************ START OF EXTRACT ***********************************//
void Foam::sixDoFRigidBodyMotionRestraints::mylinearSpring::restrain
(
    const sixDoFRigidBodyMotion& motion,
    vector& restraintPosition,
    vector& restraintForce,
    vector& restraintMoment
) const
{
    restraintPosition = motion.currentPosition(refAttachmentPt_);

    vector r = restraintPosition - anchor_;

    scalar magR = mag(r);

    // r is now the r unit vector
    r /= (magR + VSMALL);

    vector v = motion.currentVelocity(restraintPosition);
    double extension = magR-restLength_;

//---------------- Changes start --------------//

    const char *mScriptFilename = "mooringScript;";//mooringScript.m needed.

    double effExtension=mlObj.matlabCallScript(mScriptFilename,extension);

    restraintForce = -stiffness_*effExtension*r - damping_*(r & v)*r;

    restraintMoment = vector::zero;
    if (motion.report())
    {
        Info<< " attachmentPt - anchor " << r*magR
            << " spring length " << magR
            << " force " << restraintForce
            << " moment " << restraintMoment
            << endl;
        const char *mScriptFilename = "saveInfoScript;";//saveInfoScript.m needed

    // Extract the heave position and send it to MATLAB for storage//
        vector printCoM = motion.centreOfMass();
        mlObj.matlabCallScript(mScriptFilename,printCoM[2]);

//----------------- Changes end ---------------//
    }
}
//************************************ END OF EXTRACT ***********************************//
```

Apart from the `#include myFirstMatlabPipe.H`, changes have only been made within the marked area of the file. Here the present `extension` is sent to MATLAB and the effective extension is returned and used in the calculation of `restraintForce`.

Successful compilation of `mylinearSpring.C` requires some additions to the `Make/options` file. First of all the files `myFirstMatlabPipe.H` and its dependent file `engine.h` must both be in the compiler search path. Secondly, since the file is using the functionality of the `libexternalMatlabPipes` library, this must be included in the linking process of the compilation. When these modifications have been made, the `Make/options` file looks like:

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/turbulenceModels \
    -I$(LIB_SRC)/turbulenceModels/LES/LESdeltas/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(WM_PROJECT_DIR)/src/postProcessing/functionObjects/forces/lnInclude \
    -I/chalmers/sw/sup64/matlab-2011b/extern/include \
    -I$(WM_PROJECT_USER_DIR)/src/externalPipe/myFirstMatlabPipe

LIB_LIBS = \
    -lincompressibleTransportModels \
    -lincompressibleRASModels \
    -lincompressibleLESModels \
    -lbasicThermophysicalModels \
    -lspecie \
    -lcompressibleRASModels \
    -lcompressibleLESModels \
    -lfiniteVolume \
    -lmeshTools \
    -lsampling \
    -L$(FOAM_USER_LIBBIN) \
    -lexternalMatlabPipes
```

Chapter 3

3.1 floatingObject tutorial with MATLAB

The case files can largely be kept in the same way as they were described in chapter 1. However, to use `mylinearSpring` as a restraint three things must be changed. Primarily `linearSpring` must be exchanged to `mylinearSpring` in `0.org/pointDisplacement`.

```
sed -i s/"linearSpring"/"mylinearSpring"/g 0.org/pointDisplacement
```

Secondly the library `"libmylinearSpring.so"` must be added to the list of used libraries at the end of `system/controlDict`.

3.1.1 Matlab m-files for the case

Finally, the two MATLAB-files specified in `mylinearSpring`, namely `mooringScript.m` and `saveInfoScript.m` must be created. Since the point is to prove that the connection works, a very simple bilinear version of the spring is calculated by MATLAB, where the extension is multiplied by 50 if it is positive and by 0 if it is negative. In this way it represents a bilinear mooring line, which becomes very stiff if extended but cannot respond to compressive forces.

mooringScript.m

```
%----- Calculate effective extension -----%
if inputFromCpp <=0
    outputToCpp = 0;
else
   outputToCpp = 50*inputFromCpp;
end

%----- Plot the results runtime -----%
plot(ii,inputFromCpp,'k.',ii,outputToCpp,'rs');
hold on
if ii==1
  title('Evolution of actual and effective extension');
  legend('actual extension','used extension');
  xlabel('Number of time steps [-]')
  ylabel('Spring extension [m]');
end
```

saveInfoScript.m

```
zHeave=[zHeave;inputFromCpp];

save("matlabSession.mat","ii","zHeave");
```

After the creation of these two files, the `./Allrun` command should work without problems.

3.1.2 Results

The difference between the resulting heave motion of the body for different mooring conditions is presented below.

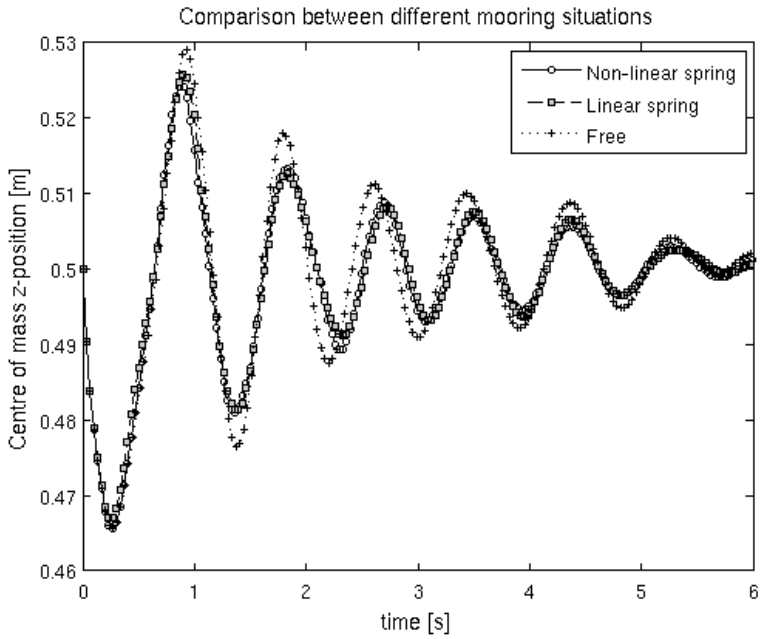

Figure 3.1: Comparison of the time evolution of the heave motion of the floating object with and without a vertical spring and with the non-linear MATLAB-calculated spring.

3.2 Outlook

This project has provided a basic tutorial for connecting MATLAB with OpenFOAM. Although the examples here shown have been on a basic level, the potential for this tool is very much larger. MATLAB is a tool used world wide and could be a powerful complement to the OpenFOAM functionality.
This project only covers one way of setting up the libraries and how they interconnect. What would be very useful, but that has not been investigated in this project, is to use the runTimeSelectionTable of OpenFOAM to connect to MATLAB and send information to it without being forced to hardcode it into the specific library in which it is used. An established general MATLAB-class library with a proper dictionary setup would prove a valuable asset to many engineers.
I hope however that this report can provide the neccessary initial steps and information to OpenFOAM users who whish to achieve the coupling between OpenFOAM and MATLAB.

Johannes Palm
October 2012

Chapter 4

Appendix

4.1 Complete source files

4.1.1 myFirstMatlabPipe.H

```
// Filename: myFirstMatlabPipe.H //

#ifndef myFirstMatlabPipe_H
#define myFirstMatlabPipe_H

#include "engine.h"

using namespace std;

class myFirstMatlabPipe
{
        // Type definition //

        // Private data objects //
        Engine *eMatlabPtr;

        // Number of calltimes //
        int ii;

    public:
        // Constructor //
        myFirstMatlabPipe();

        // Destructor //
        virtual ~myFirstMatlabPipe();

        // Send and return a double array to a matlab script //
        virtual double matlabCallScript(const char* matlabFilename,double inputArg) const;

        // Close the pipe to MATLAB //
        virtual void close() const;
};

#endif
```

4.1.2 myFirstMatlabPipe.C

```
// Filename: myFirstMatlabPipe.C //

#include<iostream>
#include<cmath>
#include "engine.h"
#include "myFirstMatlabPipe.H"
//
using namespace std;
// Constructor //
myFirstMatlabPipe::myFirstMatlabPipe()
{
    cout << "Matlab engine pointer initialized" << endl;
    // Create matlab engine pointer //
    eMatlabPtr=engOpen(NULL);
    // Create a matlab call no. iterator ii //
```

```
    engEvalString(eMatlabPtr,"ii=0;");
}
// Destructor //
myFirstMatlabPipe::~myFirstMatlabPipe(){};

double myFirstMatlabPipe::matlabCallScript(const char* matlabFilename,double inputArg) const
{
    // Increase iterator value //
    engEvalString(eMatlabPtr,"ii=ii+1;");

    // Create scalar mxArray object compatible with MATLAB and C++ //
    mxArray *inMxArray = mxCreateDoubleMatrix(1,1,mxREAL);
    double *inPtr = mxGetPr(inMxArray);

    // Send value of inMxArray to MATLAB //
    inPtr[0] = inputArg;
    engPutVariable(eMatlabPtr,"inputFromCpp",inMxArray);

    // Execute MATLAB script //
    engEvalString(eMatlabPtr,matlabFilename);

    // Extract value to C++ and return //
    mxArray *outMxArray = engGetVariable(eMatlabPtr,"outputToCpp");
    double *outPtr = mxGetPr(outMxArray);

    return outPtr[0];
};

void myFirstMatlabPipe::close() const
{
    engClose(eMatlabPtr);
};
```

4.1.3 mylinearSpring.H

```
*---------------------------------------------------------------------------*\
Class
    Foam::sixDoFRigidBodyMotionRestraints::mylinearSpring

Description
    sixDoFRigidBodyMotionRestraints model.  My linear spring.

SourceFiles
    mylinearSpring.C
\*---------------------------------------------------------------------------*/
#ifndef mylinearSpring_H
#define mylinearSpring_H

#include "sixDoFRigidBodyMotionRestraint.H"
#include "point.H"
//---------------- Changes start --------------//
#include "myFirstMatlabPipe.H"
//----------------- Changes end ---------------//

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

namespace sixDoFRigidBodyMotionRestraints
{

/*---------------------------------------------------------------------------*\
                        Class mylinearSpring Declaration
\*---------------------------------------------------------------------------*/

class mylinearSpring
:
    public sixDoFRigidBodyMotionRestraint
{
    // Private data

        //- Anchor point, where the spring is attached to an immovable
        //  object
        point anchor_;

        //- Reference point of attachment to the solid body
        point refAttachmentPt_;
```

```
        //- Spring stiffness coefficient (N/m)
        scalar stiffness_;

        //- Damping coefficient (Ns/m)
        scalar damping_;

        //- Rest length - length of spring when no forces are applied to it
        scalar restLength_;

    //---------------- Changes start --------------//

        //- The object containing the pipe to MATLAB -//
        myFirstMatlabPipe mlObj;

    //----------------- Changes end ---------------//

public:

    //- Runtime type information
    TypeName("mylinearSpring");


    // Constructors

        //- Construct from components
        mylinearSpring
        (
            const dictionary& sDoFRBMRDict
        );

        //- Construct and return a clone
        virtual autoPtr<sixDoFRigidBodyMotionRestraint> clone() const
        {
            return autoPtr<sixDoFRigidBodyMotionRestraint>
            (
                new mylinearSpring(*this)
            );
        }

    //- Destructor
    virtual ~mylinearSpring();

    // Member Functions

        //- Calculate the restraint position, force and moment.
        //  Global reference frame vectors.
        virtual void restrain
        (
            const sixDoFRigidBodyMotion& motion,
            vector& restraintPosition,
            vector& restraintForce,
            vector& restraintMoment
        ) const;

        //- Update properties from given dictionary
        virtual bool read(const dictionary& sDoFRBMRCoeff);

        //- Write
        virtual void write(Ostream&) const;
};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace solidBodyMotionFunctions
} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#endif
```

4.1.4 mylinearSpring.C

```
#include "mylinearSpring.H"
#include "addToRunTimeSelectionTable.H"
#include "sixDoFRigidBodyMotion.H"
//---------------- Changes start --------------//
#include "myFirstMatlabPipe.H"
//----------------- Changes end ---------------//
```

```
// * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * * //

namespace Foam
{
namespace sixDoFRigidBodyMotionRestraints
{
    defineTypeNameAndDebug(mylinearSpring, 0);

    addToRunTimeSelectionTable
    (
        sixDoFRigidBodyMotionRestraint,
        mylinearSpring,
        dictionary
    );
}
}


// * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * //

Foam::sixDoFRigidBodyMotionRestraints::mylinearSpring::mylinearSpring
(
    const dictionary& sDoFRBMRDict
)
:
    sixDoFRigidBodyMotionRestraint(sDoFRBMRDict),
    anchor_(),
    refAttachmentPt_(),
    stiffness_(),
    damping_(),
    restLength_()
{
    read(sDoFRBMRDict);
}


// * * * * * * * * * * * * * * * * * Destructors * * * * * * * * * * * * * * * * //

Foam::sixDoFRigidBodyMotionRestraints::mylinearSpring::~mylinearSpring()
{}


// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //

void Foam::sixDoFRigidBodyMotionRestraints::mylinearSpring::restrain
(
    const sixDoFRigidBodyMotion& motion,
    vector& restraintPosition,
    vector& restraintForce,
    vector& restraintMoment
) const
{
    restraintPosition = motion.currentPosition(refAttachmentPt_);

    vector r = restraintPosition - anchor_;

    scalar magR = mag(r);

    // r is now the r unit vector
    r /= (magR + VSMALL);

    vector v = motion.currentVelocity(restraintPosition);
    double extension = magR-restLength_;

//---------------- Changes start --------------//

    const char *mScriptFilename = "mooringScript;";//mooringScript.m needed.

    double effExtension=mlObj.matlabCallScript(mScriptFilename,extension);

    restraintForce = -stiffness_*effExtension*r - damping_*(r & v)*r;

    restraintMoment = vector::zero;
    if (motion.report())
    {
        Info<< " attachmentPt - anchor " << r*magR
            << " spring length " << magR
            << " force " << restraintForce
            << " moment " << restraintMoment
            << endl;
        const char *mScriptFilename = "saveInfoScript;";//saveInfoScript.m needed
```

```
        // Extract the heave position and send it to MATLAB for storage//
        vector printCoM = motion.centreOfMass();
        mlObj.matlabCallScript(mScriptFilename,printCoM[2]);
    }
//----------------- Changes end ---------------//
}

bool Foam::sixDoFRigidBodyMotionRestraints::mylinearSpring::read
(
    const dictionary& sDoFRBMRDict
)
{
    sixDoFRigidBodyMotionRestraint::read(sDoFRBMRDict);

    sDoFRBMRCoeffs_.lookup("anchor") >> anchor_;

    sDoFRBMRCoeffs_.lookup("refAttachmentPt") >> refAttachmentPt_;

    sDoFRBMRCoeffs_.lookup("stiffness") >> stiffness_;

    sDoFRBMRCoeffs_.lookup("damping") >> damping_;

    sDoFRBMRCoeffs_.lookup("restLength") >> restLength_;

    return true;
}

void Foam::sixDoFRigidBodyMotionRestraints::mylinearSpring::write
(
    Ostream& os
) const
{
    os.writeKeyword("anchor")
        << anchor_ << token::END_STATEMENT << nl;

    os.writeKeyword("refAttachmentPt")
        << refAttachmentPt_ << token::END_STATEMENT << nl;

    os.writeKeyword("stiffness")
        << stiffness_ << token::END_STATEMENT << nl;

    os.writeKeyword("damping")
        << damping_ << token::END_STATEMENT << nl;

    os.writeKeyword("restLength")
        << restLength_ << token::END_STATEMENT << nl;
}
// ************************************************************************* //
```

4.2 Matlab case files

4.2.1 mooringScript.m

```
%----- Calculate effective extension -----%
if inputFromCpp <=0
    outputToCpp = 0;
else
    outputToCpp = 50*inputFromCpp;
end

%----- Plot the results runtime -----%
plot(ii,inputFromCpp,'k.',ii,outputToCpp,'rs');
hold on
if ii==1
  title('Evolution of actual and effective extension');
  legend('actual extension','used extension');
  xlabel('Number of time steps [-]')
  ylabel('Spring extension [m]');
end
```

4.2.2 saveInfoScript.m

```
zHeave=[zHeave;inputFromCpp];
save("matlabSession.mat","ii","zHeave");
```