Extendable OpenSim-Matlab Infrastructure Using Class Oriented Mex Interface for C++

Introduction

The objective of this project is to provide an alternative interface between OpenSim and Matlab®, based on an extended C++ mex interface. Despite the fact that there is a user friendly OpenSim interface for Matlab, it lacks the ability to extend new functionalities based on the Java API (e.g. custom controller). Inspired by the relative project "Dynamic Simulation of Movement Based on OpenSim and MATLAB®/Simulink®", where the user can easily interface OpenSim with Simulink, the proposed framework moves one step further by providing new capabilities to link custom written C++ OpenSim extensions to Matlab and to harvest both the powerful OpenSim C++ API and Matlab functionalities. The implementation is based on Matlab mex interface, which is further extended to support more complex functionalities based on the project mexplus. The latter is a C++ Matlab mex development kit that contains a couple of C++ classes and macros to make mex development easy in Matlab.

This document provides a short description on how to employ the OpenSim-Matlab interface, by providing an example.

Dependences

OpenSim SDK: Ether use of the release version of OpenSim or pull from github.

https://simtk.org/project/xml/downloads.xml?group_id=91

https://github.com/opensim-org/opensim-core

Matlab: The user must have Matlab installed in a directory, suppose Matlab_ROOT

MEXPLUS: The source of the external library mexplus must be included in the project

https://github.com/mitkof6/mexplus

Overview

The process is a little tricky, but there are plenty examples on how to use the mexplus project to interface C++ code with Matlab. The classic mex interface is a little constraining because it provides a way to implement a single function (mexFunction) that is called when Matlab calls the corresponding dynamic library that is compiled (for Matlab the extension is .mexw32/64). The only problem is that only one entry point can be provided in the mexFunction and it is difficult to have a persistence data member and a dispatch function in order to implement more complex functionalities (refer to how to write mex functions).

The workflow has as follows:

1) Design a class that implements the C++ functionalities that you what to interface with Matlab, by providing the corresponding methods.
2) Define the mex and interface by using mexplus macros.
3) Compile the code and generate a dynamic linked library that has the name Name_.mexw32/64.

Dimitar Stanev stanev@ece.upatras.gr, 2015

4) Construct a Matlab class that has a similar function interface as the one developed, with the corresponding arguments. Place the .mexw32/64 library in the same folder as the Matlab class file.
5) Run the library using the class oriented interface.

Example

1) The following snippet of code implements the C++ OpenSim functionalities that we want to interface with Matlab. Ignore the FeedforwardController that is a custom developed controller with an extended functionality that we want to use.

```
/*
Copyright (C) 2015  Dimitar Stanev (stanev@ece.upatras.gr)

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
#ifndef MATLAB_ENGINE_H
#define MATLAB_ENGINE_H

#include <OpenSim/Simulation/Model/Model.h>
#include <OpenSim/Simulation/Manager/Manager.h>
#include <OpenSim/Analyses/Kinematics.h>

#include "mexplus.h"

#include "FeedforwardController.h"

using namespace std;
using namespace mexplus;
using namespace OpenSim;
using namespace SimTK;


class EngineMatlab
{
public:

    EngineMatlab()
    {

    }

    ~EngineMatlab()
    {

    }
    void setup(const string& fileName)
    {
        mexPrintf("Open model: %s \n", fileName.c_str());
        m_model = new Model(fileName);

        // kinematics
        m_kinematics = new Kinematics();
        m_kinematics->setModel(*m_model);
        m_kinematics->setRecordAccelerations(true);
        m_model->addAnalysis(m_kinematics);

        // controller
        m_feedController = new FeedforwardController();
        m_model->addController(m_feedController);

        // initialize
        m_model->buildSystem();
        m_state = m_model->initializeState();

        Array<string> muscleNames;
        m_model->getMuscles().getNames(muscleNames);
        m_feedController->setMuscleNames(muscleNames);

        // manager
        RungeKuttaMersonIntegrator* integrator =
            new RungeKuttaMersonIntegrator(m_model->getMultibodySystem());
        m_manager = new Manager(*m_model, *integrator);

        m_currentTime = 0;
    }

    bool step(double tf)
    {
        mexPrintf("Integration from %g to %g \n", m_currentTime, tf);

        if (tf < m_currentTime)
        {
            mexPrintf("tf must be grater than current simulation time t0 (tf:%g <
t0:%g)\n",
                tf, m_state->getTime());
            return false;
        }

        m_manager->setInitialTime(m_currentTime);
        m_manager->setFinalTime(tf);

        try
        {
            m_manager->integrate(*m_state);
        }
        catch (const std::exception& ex)
        {
            mexPrintf("Exception: '%s'\n", ex.what());
            return false;
        }
        catch (...)
        {
            mexPrintf("Unrecognized exception \n");
            return false;
        }

        mexPrintf("Integration finished\n");
        m_currentTime = tf;
        return true;

    }

    void setExcitations(const vector<double>& ext)
    {
        if (ext.size() != m_model->getNumControls())
        {
            mexPrintf("The number of controls does not agree ext_%d != act_%d\n",
                ext.size(), m_model->getNumControls());
            return;
        }

        m_feedController->setControls(ext);
    }

    void setExcitation(const string& muscle, double ext)
    {
        bool flag = m_feedController->setControls(muscle, ext);

        if (!flag)
        {
            mexPrintf("The %s muscle does not exist\n", muscle);
        }
    }

    vector<string> getMuscleNames() const
    {
        Array<string> temp;
        m_model->getMuscles().getNames(temp);

        vector<string> names;
        for (int i = 0; i < temp.size(); i++)
        {
            names.push_back(temp[i]);
        }

        return names;
    }

    double getCoordinateValue(const string& name)
    {
        return m_model->getCoordinateSet().get(name).getValue(*m_state);
    }

    void printResults(const string& dir)
    {
        mexPrintf("Printing result in: %s directory\n", dir.c_str());

        m_kinematics->printResults(m_model->getName(), dir);

        OpenSim::Storage stateStorage(m_manager->getStateStorage());
        m_model->updSimbodyEngine().convertRadiansToDegrees(stateStorage);
        stateStorage.print(dir + "/" + m_model->getName() + "_state.sto");
    }


private:

    ReferencePtr<Model> m_model;
    ReferencePtr<Kinematics> m_kinematics;
    ReferencePtr<Manager> m_manager;
    ReferencePtr<State> m_state;
    ReferencePtr<FeedforwardController> m_feedController;

    double m_currentTime;

};

#endif
```

2) Next we must define the mex interface using the mexplus macros. What these macros do is: 1) to get an instance of the persistent object (class), 2) to dispatch the corresponding member function and 3) to convert the Matlab arguments to C++ datatypes and pass them to the dispatcher function.

```
#include "EngineMatlab.h"

template class mexplus::Session<EngineMatlab>;

namespace
{

MEX_DEFINE(new) (int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
{
    InputArguments input(nrhs, prhs, 0);
    OutputArguments output(nlhs, plhs, 1);

    output.set(0, Session<EngineMatlab>::create(new EngineMatlab()));
}

MEX_DEFINE(delete) (int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
{
    InputArguments input(nrhs, prhs, 1);
    OutputArguments output(nlhs, plhs, 0);

    Session<EngineMatlab>::destroy(input.get(0));
}

MEX_DEFINE(setup) (int nlhs, mxArray* plhs [], int nrhs, const mxArray* prhs [])
{
    InputArguments input(nrhs, prhs, 2);
    OutputArguments output(nlhs, plhs, 0);

    EngineMatlab* engine = Session<EngineMatlab>::get(input.get(0));
    engine->setup(input.get<string>(1));
}

MEX_DEFINE(step) (int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
{
    InputArguments input(nrhs, prhs, 2);
    OutputArguments output(nlhs, plhs, 1);

    EngineMatlab* engine = Session<EngineMatlab>::get(input.get(0));
    output.set<bool>(0, engine->step(input.get<double>(1)));
}

MEX_DEFINE(setExcitations) (int nlhs, mxArray* plhs [], int nrhs, const mxArray* prhs
[])
{
    InputArguments input(nrhs, prhs, 2);
    OutputArguments output(nlhs, plhs, 0);

    EngineMatlab* engine = Session<EngineMatlab>::get(input.get(0));
    engine->setExcitations(input.get<vector<double>>(1));
}

MEX_DEFINE(setExcitation) (int nlhs, mxArray* plhs [], int nrhs, const mxArray* prhs
[])
{
    InputArguments input(nrhs, prhs, 3);
    OutputArguments output(nlhs, plhs, 0);

    EngineMatlab* engine = Session<EngineMatlab>::get(input.get(0));
    engine->setExcitation(input.get<string>(1), input.get<double>(2));
}

MEX_DEFINE(getMuscleNames) (int nlhs, mxArray* plhs [], int nrhs, const mxArray* prhs
[])
{
    InputArguments input(nrhs, prhs, 1);
    OutputArguments output(nlhs, plhs, 1);

    EngineMatlab* engine = Session<EngineMatlab>::get(input.get(0));
    output.set< vector<string> >(0, engine->getMuscleNames());
}

MEX_DEFINE(getCoordinateValue) (int nlhs, mxArray* plhs [], int nrhs, const mxArray*
prhs [])
{
    InputArguments input(nrhs, prhs, 2);
    OutputArguments output(nlhs, plhs, 1);

    EngineMatlab* engine = Session<EngineMatlab>::get(input.get(0));
    output.set<double>(0, engine->getCoordinateValue(input.get<string>(1)));
}

MEX_DEFINE(printResults) (int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
{
    InputArguments input(nrhs, prhs, 2);
    OutputArguments output(nlhs, plhs, 0);

    EngineMatlab* engine = Session<EngineMatlab>::get(input.get(0));
    engine->printResults(input.get<string>(1));
}

} // namespace

MEX_DISPATCH // Don't forget to add this if MEX_DEFINE() is used.
```

3) For the compilation we must include and link the Matlab mex infrastructure. The mex include directory is defined as **${Matlab_ROOT}/extern/include**, while the corresponding mex library directory is **${Matlab_ROOT}/extern/lib/win64/Microsoft**. The libraries that must be linked are **libmx libmex libmat.** The library must be compiled as a module with link flags "/export:mexFunction" and suffix .mexw32/64. The following cmake file is used where we must also include the mexplus project headers.

```
# EngineMatlab
```

```
####################################################################################
# Source files
FILE(GLOB SOURCE_FILES *.h *.cpp)
SET(SOURCES ${SOURCE_FILES})

####################################################################################
# Dependences
INCLUDE_DIRECTORIES(${CMAKE_SOURCE_DIR}/mexplus)
INCLUDE_DIRECTORIES(${Matlab_INCLUDE_DIRS})
#INCLUDE_DIRECTORIES(${Matlab_SIMULINK_INCLUDE_DIRS})
LINK_DIRECTORIES(${Matlab_LIBRARIES})

####################################################################################
# Build library
ADD_LIBRARY(EngineMatlab_ MODULE ${SOURCES})
ADD_DEFINITIONS(-DMATLAB_MEX_FILE)
SET_TARGET_PROPERTIES(EngineMatlab_ PROPERTIES LINK_FLAGS "/export:mexFunction")
SET_TARGET_PROPERTIES(EngineMatlab_ PROPERTIES SUFFIX ${Matlab_MEX_EXTENSION})
TARGET_LINK_LIBRARIES(EngineMatlab_ libmx libmex libmat ${OpenSim_LIBRARIES})
```

4) The next step is to implement the Matlab class interface that will dispatch the corresponding c++ method from the dynamic library. The class file and .mexw32/64 files must be in the same directory.

```
classdef EngineMatlab < handle
%EngineMatlab usage of the mexplus development kit.

properties (Access = private)
  id_ % ID of the session
end

methods
  function this = EngineMatlab()
    %EngineMatlab Create a new
    this.id_ = EngineMatlab_('new');
  end

  function delete(this)
    %delete Destructor
    EngineMatlab_('delete', this.id_);
  end

  function setup(this, fileName)
    %step integration step
    assert(isscalar(this));
    assert(ischar(fileName));
    EngineMatlab_('setup', this.id_, fileName);
  end

  function bool = step(this, tf)
    %step integration step
    assert(isscalar(this));
    bool = EngineMatlab_('step', this.id_, tf);
  end

  function setExcitations(this, excitations)
    %setExcitations sets the excitations of the muscle
    assert(isscalar(this));
    EngineMatlab_('setExcitations', this.id_, excitations);
  end
  function setExcitation(this, muslceName, excitation)
    %setExcitations sets the excitations of the muscle by name
    assert(isscalar(this));
    assert(ischar(muslceName));
    assert(isscalar(excitation));
    EngineMatlab_('setExcitation', this.id_, muslceName, excitation);
  end

  function muscleNames = getMuscleNames(this)
    %getMuscleNames integration step
    assert(isscalar(this));
    muscleNames = EngineMatlab_('getMuscleNames', this.id_);
  end

  function value = getCoordinateValue(this, name)
    %getLatestNormalizedMuscleForce gets the value of a spesific
    %coordinate
    assert(isscalar(this));
    assert(ischar(name));
    value = EngineMatlab_('getCoordinateValue', ...
        this.id_, name);
  end

  function printResults(this, dir)
    %printResults Save something to the database.
    assert(isscalar(this));
    EngineMatlab_('printResults', this.id_, dir);
  end

end

end
```

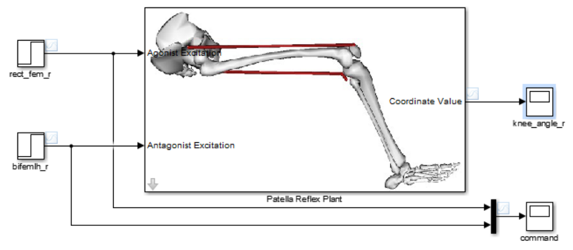
5) Finally, construct the object and call the corresponding methods based on the class oriented interface.

```
clear all;
clc;
close all;

%% initialization
type = 'Release';
copyfile(strcat(pwd, '/../build/', type, '/EngineMatlab_.mexw64'), pwd)

%% setup

engine = EngineMatlab();
engine.setup('patella_reflex_model.osim');
muscles = engine.getMuscleNames();

%% simulation

engine.setExcitations([0 0 0 0 0.5 0 0 0 0 0 0]);
%engine.setExcitation(char(muscles(5)), 1);
finished = engine.step(1)
engine.getCoordinateValue(char('knee_angle_r'))

engine.setExcitation(char(muscles(5)), 0);
engine.setExcitation(char(muscles(1)), 0.5);
finished = engine.step(2)
engine.getCoordinateValue(char('knee_angle_r'))
engine.printResults(pwd);

%% clear
engine.delete();

clear engine;
clear EngineMatlab_;
```

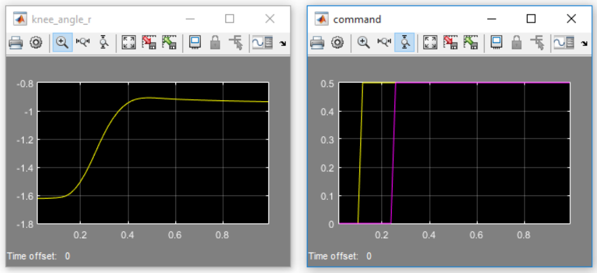
Simulink Interface

In order to use the class oriented interface with Simulink, we must provide a block component with specific inputs and outputs as defined by the C++ interface. One way to do this, is to use the Simulink S-Function infrastructure. Because S-Function is designed for solving dynamic systems, the states of the system are hidden (integration handled by OpenSim), so we implemented a direct feedthrough S-Function with no states, but only inputs and outputs that are interfaced with the OpenSim engine class. We are interested in continuous integration, so we must define the sampling of the S-Function component to be ts = [0 0].

```
function [sys, x0, str, ts] = SEngineMatlab(t, x, u, flag, engine, ...
    agonistMuscle, antagonistMuscle, coordinateName, t0, dt)
% S-function interface between EngineMatlab and Simulink
%
% parameters:
%   p(1) = MatlabEngine
%   p(2) = agonist muscle name
%   p(3) = antagonist muscle name
%   p(4) = coordinate name to observe
%   not used p(4) = initial simulation time (t0)
%   not used p(5) = simulation time step (dt)
%
% input:
%   u(1) = agonist muscle excitation
%   u(2) = antagonist muscle excitation
%
% output:
%   o(5) = coordinate variable (e.g. knee_angle_r)


switch flag

    %%%%%%%%%%%%%%%%%%
    % Initialization %
    %%%%%%%%%%%%%%%%%%
    case 0
        str = [];
        % of 0 0 it is a continuous integration
        % [dt t0]
        ts = [0 0];
        x0 = [];
        s = simsizes ;
        s.NumContStates = 0;
        s.NumDiscStates= 0 ;
        s.NumOutputs = 1;
        s.NumInputs = 2;
        % DirFeedthrough must be set to 1 when there is no state
        % so u will not be NaN for flag = 3
        s.DirFeedthrough = 1;
        s.NumSampleTimes = 1;
        sys = simsizes(s);
        %%%%%%%%%%%%%%%
        % Derivatives %
        %%%%%%%%%%%%%%%
    case 1

        sys = [];

        %%%%%%%%%%%%%%%%%%%%%%%%
        % Update and Terminate %
        %%%%%%%%%%%%%%%%%%%%%%%%
    case {2,4,9}
        sys = []; % do nothing

        %%%%%%%%%%
        % Output %
        %%%%%%%%%%
    case 3

        engine.setExcitation(char(agonistMuscle), u(1));
        engine.setExcitation(char(antagonistMuscle), u(2));

        if engine.step(t) == 1
            sys = engine.getCoordinateValue(char(coordinateName));
        else
            DAStudio.error('EngineMatlab faled during step');
        end

    otherwise
        DAStudio.error('Simulink:blocks:unhandledFlag', num2str(flag));
end
```



Go to model properties to check the initialization callback



Conclusion

This simple example presents the aspects of the proposed framework. The new framework is capable to extend the OpenSim functionalities (see FeedforwardController) and to directly link with Matlab. It is evident that the user can use the class oriented interface in order to connect OpenSim engine with Simulink too, simply by providing the corresponding functional interface for the two simulators to correspond.