QBlade Version 2.0.5.2: Matlab Tutorial

Livia Brandetti[1,2] and Daniel van den Berg[1]

[1]Delft Center for Systems and Control, Faculty of Mechanical Engineering, Delft University of Technology, Delft, The Netherlands
[2]Wind Energy Section, Faculty of Aerospace Engineering, Delft University of Technology, Delft, The Netherlands

06 April 2023

This report is the result of a collaboration between the Delft University of Technology and the Technical University of Berlin.

1 Introduction

QBlade is a highly advanced multi-physics code that covers the complete range of aspects required for the aero-servo-hydro-elastic design, prototyping, simulation, and certification of wind turbines. To use this interface, the QBlade code is loaded into the desired framework as a dynamic link library (.dll). In contrast, the scripting functions are exported through dedicated functions in an include.h header file.

Since, at present, only an example for using QBlade in Python is available, the main goal of this document is to provide a detailed explanation of how to use MATLAB to communicate with the QBlade Enterprise Edition (i.e. QBlade-EE) in a `.dll` format. However, since the EE version requires a license, the examples provided in this report have been adapted to work with the CE version. In both cases, the first step is to download the desired version (e.g. QBlade-CE or QBlade-EE) from the following website: `https://qblade.org/downloads/`. Then, by making use of the help function and the required syntax, the .dll interface is accessed in Matlab.

The report is structured as follows: Section 2 gives an overview of the motivation for using a .dll interface. A detailed description of every function included in the header file and the Matlab syntax required to build the interface are provided in Section 3. Section 4 illustrates several working examples together with their implementation in Matlab.

2 Motivation for using QBlade as DLL

This documentation will show how QBlade can be interfaced with Matlab by accessing it as a dynamic link library (.dll). The primary reason for this is to be able to run a number of different simulations, defined within Matlab, without having to set up a large number of simulations within QBlade. The desired workflow can be described as follows:

- Begin with a baseline simulation case.
- Make a copy of this simulation case and alter either controller or structural settings for this particular simulation.
- Initialize and run the simulation.
- Optional: Change wind/operating conditions during simulation.
- Optional: Read out structural and aerodynamic information during simulations.
- Optional: Apply control inputs.
- At the end of the simulation, save the project.

Figure 1 shows a graphical representation of the desired workflow. The main idea depicted is that you begin with a baseline set-up (In this case a floating wind turbine) and copy that whilst altering some properties of the system. The Matlab code is only added for illustrative purposes. Chapter 4 will have examples of working Matlab code for common wind turbine simulation cases.

Figure 1: Graphical representation of the desired workflow using QBlade as .dll within Matlab. The Matlab code included is only for illustrative purposes.

This workflow reduces the number of interactions required with the GUI version of QBlade, reducing the number of potential mistakes when setting up a simulation and allowing for easier batch simulations.

3 Interfacing QBlade and Matlab

This section will give a brief overview of how one can use Matlab to interact with a .dll file. It will provide a short introduction to the commonly used Matlab commands and an overview of all the functions associated specifically with the QBlade .dll. The final section also introduces a Matlab help function that contains information on all the associated functions, including examples on how to interact with it using Matlab.

3.1 Matlab syntax

To build the interface in Matlab, the user needs to create a folder called **QBladeDLLInterface** in the example, containing the license file, QBladeEE.dll, QBladeDLLInclude.h (i.e. the header file for all the functions of the .dll), QBladeEE.exe and the following sub-folders:

1. **Source** contains the QBlade project in .sim format or .qpr format and the folder with the information on the simulation.
2. **MATLAB files** is where the Matlab interface will be built and where a sub-folder, called **Functions** in the example, can be created to include any Matlab functions needed in the simulations. For instance, in the example, one of these functions is used to write the parameters for the discon.in of the Delft Research Controller .dll.

After creating the required folders, the user should create a Matlab script and apply the following syntax:

- `loadlibrary` to load the .dll library.
  Example:
  `loadlibrary`('<userpath> QBladeEE.dll',
  'QBladeDLLFunctions.h','alias','QBladeDLL')
- `libfunctions` to store the functions in a cell.
  Example:
  m = `libfunctions`('QBladeDLL')
- `libfunctionsview` to display the signature of the function, i.e. what a function includes in terms of data and datatype;
- `calllib` to call the functions. It is important to check the type of data the functions need (Section 3.2).
  Example:
  `calllib`('QBladeDLL','initializeSimulation')
- `unloadlibrary` to unload the library.

The order in which the commands are presented here should be the same order in which the commands are called in the Matlab code. The paths to load the library and the project should be set absolute. This is the reason why in the example, the <**userpath**> is used.

3.2 Help Function

Interacting with the .dll can be done by calling one of the 24 included functions. These functions are defined in C code and provided with the .dll. When loading the library using the `loadlibrary` command (see Section 3.1) a so-called header file also needs to be provided. The header file and .dll need to be the same programming language, i.e., if the .dll is compiled in C, then the header should also be written in C. The full list of functions is given by:

1. loadProject
2. loadSimDefinition
3. storeProject
4. setDebugInfo
5. setLibraryPath
6. createInstance
7. closeInstance
8. loadTurbulentWindBinary
9. addTurbulentWind
10. initializeSimulation
11. setTimestepSize
12. setRPMPrescribeType_at_num
13. setRampupTime
14. getWindspeed
15. getCustomData_at_num
16. setInitialConditions_at_num
17. setTurbinePosition_at_num
18. setPowerLawWind
19. setControlVars_at_num
20. getTurbineOperation_at_num
21. advanceController_at_num
22. advanceTurbineSimulation
23. runFullSimulation

24. setLibraryPath

25. getTowerBottomLoads_at_num

A full description of each function, and how to interact with it using Matlab, can be found in Appendix 5. All the descriptions there can also be had in Matlab directly, as an output of a Matlab help function called `QBladeFunctionHelp`. This function is provided with the example files (Section 4). To get information on one of the individual functions, call the help function with the name of the library call functions as a string. All these functions are called using the `calllib` function in Matlab, see Section 3.1.

Example:
Call information on the 'getCustomData_at_num' library function:
QBladeFunctionHelp('getCustomData_at_num')

4 Examples

This section presents five different applications of the QBlade Matlab interface. First, a minimum working example, which can be used as a starting point, is described. Then, the conventional and most-used torque control strategy, i.e. the baseline $K\omega^2$ controller, is implemented in a turbine simulation. Example 4.3 expands on the baseline $K\omega^2$ controller by also including a step in wind speed. This could be of interest to test controllers for a range of wind speeds. Example 4.4 contains two examples of how to interact with the Delft Research Controller DLL. These examples are specific to the work carried out at the TU Delft. However, these examples also show how one can set-up batch simulations, output data for each simulation and interact with a project that contains a controller using the bladed interface. The final example 4.5 is based on example 4.2, but contains two turbines. The first turbine is modelled using the free-vortex wake, such that its wake interacts with the second turbine. This example shows how two turbines can be used using the Matlab interface and can only run for the QBlade–EE version.

4.1 Minimum Working Example

A minimum working example of how to interface with QBlade in Matlab is presented by providing both the Matlab code and a detailed description of the steps that need to be performed. In this example, the NREL-5MW reference turbine bottom fixed [3] is simulated for ten-time steps and outputs the rotational speed in rpm. The QBlade project, called 'NREL5MW.qpr', was made available by the developers.

The following steps are performed in the example below:

1. load the .dll library with the `loadlibrary` command;

2. store the .dll functions in a cell with the `libfunctions` command. Note that if the cell is empty, it will mean that the library is not loaded correctly, and an error message is given as a warning;

3. set the library path with the following command
   `calllib`('QBladeDLL','setLibraryPath','../../QBladeCE2.0.5.2.dll')

4. create a new instance of QBlade with the `calllib` command;

5. define the project file that needs to be loaded, in this case, is the 'NREL5MW.qpr';

6. load the project with the `calllib` command;

7. initialize the simulation for a selected device, i.e. 1, and certain OpenCL parameters, i.e. 24, with the `calllib` command;

8. define the time length of the simulation, which equals the number of simulation steps times the time step with which the simulation has been set up. It is possible to check in the GUI under the simulation settings tab to avoid any mistakes;

9. define the variable that will be read and loaded during the simulation, i.e. the rotational speed. Note that the variable name should follow the label of the graphs in the GUI;

10. advance the structural part of the simulation and calculates all structural forces with the `calllib` command;

11. advance the aerodynamic part of the simulation and calculates all aerodynamic forces with the `calllib` command;

12. extract the rotational speed during the simulation with the `calllib` command;

13. end the simulation by closing the QBlade instance with the `calllib` command.

```
addpath('..\..\');
% Establish connection

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
loadlibrary('../../QBladeCE_2.0.5.2.dll',
'../QBladeDLLFunctions.h','alias','QBladeDLL')

m = libfunctions('QBladeDLL') ;

if isempty(m)
    fprintf('Error')
end

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
calllib('QBladeDLL','setLibraryPath','../../QBladeCE_2.0.5.2.dll')

calllib('QBladeDLL','createInstance',0,24)

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
projectFile = '../../Source/NREL5MW.qpr';

calllib('QBladeDLL','loadProject',projectFile)
calllib('QBladeDLL','initializeSimulation')
simTime = 100; %in timestep, actual time is timestep*#timesteps
valuestr = 'Rotational Speed [rpm]';

f = waitbar(0,'Initializing Simulation') ;

for i = 1:1:simTime
    calllib('QBladeDLL','advanceTurbineSimulation')
    a = calllib('QBladeDLL','getCustomData_at_num',valuestr, 0, 0);
    rpm(i,:) = a;

    waitbar(i/simTime,f,'Simulation Running')

end

close(f);
%calllib('QBladeDLL','storeProject','check.qpr')
calllib('QBladeDLL','closeInstance')
```

4.2 Baseline $K\omega^2$ controller

This example builds on the minimum work example by also adding a $K\omega^2$ controller to the for-loop that runs the wind turbine simulation. In this example, the $K\omega^2$ controller is used because it is a commonly used control scheme in wind turbines. However, within this example, there is room to change the $K\omega^2$ controller for more complex control methods. Furthermore, this example only uses generator torque as a control action, however, the other four inputs constitute the turbine yaw angle and individual blade pitch angles. These can all be used for control simultaneously if needed.

For this example, we use the same project as with the minimum working example, a bottom-fixed NREL 5MW reference turbine. The gain K equals 2.24 for the NREL 5MW turbine and the gearbox ratio, needed to change rotor rpm to Low-Speed-Shaft (LSS) rpm, is $N = 97$. The controller is implemented in line 42 of the Matlab code. A conversion from rpm to rad/s is needed for it to function correctly.

```
1  addpath('..\..\');
2
3  %this is setup using relative path and depends on the location ...
       of this file, if there are issues use absolute path to these ...
       files
4  loadlibrary('../../QBladeCE_2.0.5.2.dll',
5  '../QBladeDLLFunctions.h','alias','QBladeDLL')
6
7  m = libfunctions('QBladeDLL') ;
8
9  if isempty(m)
10     fprintf('Error')
11 end
12
13 %this is setup using relative path and depends on the location ...
       of this file, if there are issues use absolute path to these ...
       files
14 calllib('QBladeDLL','setLibraryPath','../../QBladeCE_2.0.5.2.dll')
15
16 calllib('QBladeDLL','createInstance',0,24)
17
18 %this is setup using relative path and depends on the location ...
       of this file, if there are issues use absolute path to these ...
       files
19 projectFile = '../../Source/NREL5MW.qpr';
20
21 calllib('QBladeDLL','loadProject',projectFile)
22 calllib('QBladeDLL','initializeSimulation')
23
24 simTime = 400; %in timestep, actual time is timestep*#timesteps
25 valuestr = 'Rotational Speed [rpm]';
26 valuestr2 = 'Gen. HSS Torque [Nm]';
27 % valuestr2 = 'Gen. Power (w.o. losses) [kW]';
28
29 K = 2.24;
30 N = 97;
```

```
31
32  f = waitbar(0,'Initializing Simulation') ;
33
34  for i = 1:1:simTime
35
36      calllib('QBladeDLL','advanceTurbineSimulation')
37      omega = calllib('QBladeDLL','getCustomData_at_num',valuestr, ...
            0.5, 0);
38      genTorqueQB = ...
            calllib('QBladeDLL','getCustomData_at_num',valuestr2, 0, 0);
39      genTorqueQB_store(i,:) = genTorqueQB;
40
41      omega_g = omega*N;
42      genTorque = K.*(omega_g*(2*pi/60))^2;
43      genTorque_store(i,:) = genTorque ;
44
45      calllib('QBladeDLL','setControlVars_at_num',[genTorque 0 0 0 ...
            0],0)
46
47      waitbar(i/simTime,f,'Simulation Running')
48
49  end
50  close(f)
51
52  calllib('QBladeDLL','closeInstance')
53
54  figure;
55  plot(genTorqueQB_store)
56  hold on
57  plot(genTorque_store)
58  grid on
59  legend('QB HSS Torque','K omega^2')
```

Once run, without altering the code, the output should look as in Figure 2. The blue line represents the generator torque that is being sent to QBlade using **setControlVars_at_num**. The red line shows the generator torque read out from QBlade using **getCustomData_at_num**. Notice how the data from QBlade is one step behind the calculated torque; this is because that data channel is read before applying the torque to the system (one step behind).

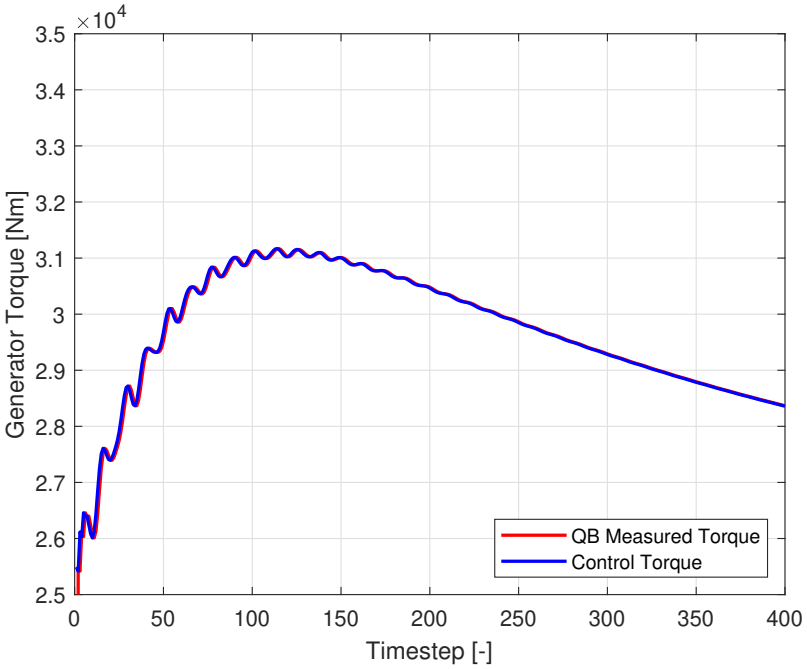

Figure 2: Output of the example code running the $K\omega^2$ controller.

4.3 Step in Wind Simulation

In this example, we will expand on the $K\omega^2$ controller build in Section 4.2. Specifically, during the simulation, we put a step on the wind speed of 1 m/s, raising it from 9 m/s to 10 m/s. Also included in the script is a measurement of the wind speed in x, y and z directions at hub height. An increase in wind speed also warrants a response from the $K\omega^2$ controller, hence its inclusion in this code.

```
addpath('..\..\');

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
loadlibrary('../../QBladeCE_2.0.5.2.dll',
'../QBladeDLLFunctions.h','alias','QBladeDLL')

m = libfunctions('QBladeDLL') ;

if isempty(m)
    fprintf('Error')
end

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
calllib('QBladeDLL','setLibraryPath','../../QBladeCE_2.0.5.2.dll')

calllib('QBladeDLL','createInstance',0,24)

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
projectFile = '../../Source/NREL5MW.qpr';

calllib('QBladeDLL','loadProject',projectFile)
calllib('QBladeDLL','initializeSimulation')

simTime = 3000; %in timestep, actual time is timestep*#timesteps
valuestr = 'Rotational Speed [rpm]';
valuestr2 = 'Gen. HSS Torque [Nm]';
f = waitbar(0,'Initializing Simulation') ;


K = 2.24;
N = 97;
startWind = 1500;
for i = 1:1:simTime

    calllib('QBladeDLL','advanceTurbineSimulation')
    if i > startWind
    calllib('QBladeDLL','setPowerLawWind', 10, 0, 0, 0, 87.6)
    end
    omega = calllib('QBladeDLL','getCustomData_at_num',valuestr, ...
        0.5, 0) ;
```

```
40      genTorqueQB = ...
            calllib('QBladeDLL','getCustomData_at_num',valuestr2, 0, ...
            0) ;
41      genTorqueQB_store(i,:) = genTorqueQB;
42
43      omega_g = omega*N;
44      genTorque = K.*(omega_g*(2*pi/60))^2;
45      genTorque_store(i,:) = genTorque ;
46
47      calllib('QBladeDLL','setControlVars_at_num',[genTorque 0 0 0 ...
            0],0)
48      V_hub(i,:) = calllib('QBladeDLL','getWindspeed', -20, 0, ...
            87.6, [0 0 0]);
49      waitbar(i/simTime,f,'Simulation Running')
50
51  end
52
53  close(f)
54  % calllib('QBladeDLL','storeProject','Test.qpr')
55  calllib('QBladeDLL','closeInstance')
56
57  figure;
58  subplot(2,1,1)
59  plot(genTorqueQB_store)
60  hold on
61  plot(genTorque_store)
62  grid on
63  legend('QB HSS Torque','K omega^2')
64
65  subplot(2,1,2)
66  plot(V_hub)
67  grid on
68  legend('Wind Speed [m/s]')
```

4.4 Delft Research Controller

The Delft Research Controller (DRC) provides an open, modular and fully adaptable wind turbine controller [6]. New controller implementations can be added to the existing controller, allowing the assessment of new algorithms. The DRC is developed in Fortran and uses the Bladed-style DISCON controller interface. The compiled controller is configured by a single control settings parameter file, i.e. discon.in, and can work with any wind turbine model and simulation software using the DISCON interface. These variables are called UserVars and are communicated to the controller .dll through the SWAP array. The relative path to the file containing the UserVars is hard coded into the compiled version of the DRC.

NREL recently acknowledged the potential of the DRC by adopting it as their baseline control solution of choice and dubbed it as the Reference Open-Source Controller (ROSCO) [5, 4]. To provide an easy and convenient way of controller development via a graphical interface, this controller design and compilation environment has been developed in Simulink and refer to SimulinkDRC [7]. These examples are primarily included for the people working in the Data-Driven Control group at the Delft University of Technology. These examples do show how QBlade can be used to run batch simulations and, at the same time, save data sets for post-processing.

4.4.1 WSE-TSR tracking controller

This example uses a stripped-down version of the Delft Research Controller specifically designed for partial load torque control either with the most common $K\omega^2$ controller or with the combined wind speed estimator and tip-speed ratio (WSE-TSR) tracking controller [1]. In this example, the WSE-TSR tracking controller is enabled with two different settings of gain. The controller and estimator gains are governed by an input file called discon.in. The values in this file are passed to it through Matlab and the `writeDisconWSETSR` function.

Main:

```
%%
% Writing DISCON loop.

%%
clear all
close all
clc

%% Define paths

%set the absolute path to your qblade directory here
UserPath = '<userpath>\QBladeCE_2.0.5.2\';
disconPath = UserPath ;
MatlabPath = [UserPath ...
    'MATLAB_Files\4p4_DRC_Cases\4p41_BaselineTorqueControl\'];
SourcePath = [UserPath 'Source'] ;
DllPath = [UserPath 'QBladeCE_2.0.5.2.dll'];
addpath('..\Functions');
addpath('..\..\..\');
%%

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
loadlibrary(DllPath,'../../QBladeDLLFunctions.h','alias','QBladeDLL')

m = libfunctions('QBladeDLL') ;

if isempty(m)
    fprintf('Error')
end

%% Defining Settings
Pitchb1 = 0;
Pitchb2 = 0;
Pitchb3 = 0;
KpTSR = [-1635.1714 -2079.5682];
KiTSR = [-6451.3008 -11.6565];
KpWSE = [8.0258 11.2468];
KiWSE = [0.2796 0.2775];
TSRref = [7.2385 7.7187];
K = [0.0000 0.0000];
Tg0 = [25000.0000 25000.0000];

SimNr = length(TSRref);
tic
for i_for1 = 1:1:SimNr

    Sim_name_folder = ['Sim_', num2str(i_for1)] ;

    copyfile([SourcePath],Sim_name_folder);

    cd(disconPath)

```

```
52      writeDisconWSETSR(Pitchb1,Pitchb2,Pitchb3, ...
53          KpTSR(i_for1),KiTSR(i_for1),KpWSE(i_for1),KiWSE(i_for1), ...
54          TSRref(i_for1),K(i_for1),Tg0(i_for1))
55      cd(MatlabPath)
56
57      run Torque_Simulation.m
58
59      cd(Sim_name_folder)
60
61      save(['Output_',num2str(i_for1)],'DATA')
62      cd('..\..\..\..\')
63      delete('discon.in');
64      cd(MatlabPath)
65  end
66  %%
67  toc
68
69  figure();grid on;
70  subplot(1,4,1)
71  title('Generator Torque')
72  plot(DATA.time,DATA.Tg); hold on;
73  plot(DATA.time,DATA.TgSWAP);
74  subplot(1,4,2)
75  title('Rotational speed')
76  plot(DATA.time,DATA.Omega);
77  subplot(1,4,3)
78  title('Tip-speed ratio')
79  plot(DATA.time,DATA.TSR);
80  hold on;
81  plot(DATA.time,DATA.TSRest);
82  legend('Measured','Estimated');
83  subplot(1,4,4)
84  title('Rotor-effective wind speed')
85  plot(DATA.time,DATA.U);
86  hold on;
87  plot(DATA.time,DATA.Uest);
88  legend('Measured','Estimated');
```

Function:

```
simFile_loc = [MatlabPath Sim_name_folder '\'];

calllib('QBladeDLL','setLibraryPath',DllPath)
calllib('QBladeDLL','createInstance',1,24)
simName = 'NREL5MW_WSETSRK.sim';
calllib('QBladeDLL','loadSimDefinition',[simFile_loc simName])
calllib('QBladeDLL','initializeSimulation')

simTime = 4000; %in timestep, actual time is timestep*#timesteps ...
    %TO DO CHANGE
valuestr = 'Time [s]';
valuestr1 = 'Rotational Speed [rpm]';
valuestr2 = 'Gen. HSS Torque [Nm]';
valuestr3 = 'SWAP[46] [-]'; %Generator torque
valuestr4 = 'Abs Wind Vel. at Hub [m/s]';
valuestr5 = 'SWAP[84] [-]'; %Estimated wind speed
valuestr6 = 'Tip Speed Ratio [-]';
valuestr7 = 'SWAP[85] [-]'; %Estimated tip-speed ratio

for i_for2 = 1:1:simTime

    calllib('QBladeDLL','advanceTurbineSimulation')
    calllib('QBladeDLL','advanceController_at_num',[0 0 0 0 0],0)

    Time = calllib('QBladeDLL','getCustomData_at_num',valuestr, ...
        0, 0);
    Rpm = calllib('QBladeDLL','getCustomData_at_num',valuestr1, ...
        0, 0);
    GenTg = ...
        calllib('QBladeDLL','getCustomData_at_num',valuestr2, 0, 0);
    GenTgSWAP = ...
        calllib('QBladeDLL','getCustomData_at_num',valuestr3, 0, 0);
    U = calllib('QBladeDLL','getCustomData_at_num',valuestr4, 0, 0);
    Uest = calllib('QBladeDLL','getCustomData_at_num',valuestr5, ...
        0, 0);
    TSR = calllib('QBladeDLL','getCustomData_at_num',valuestr6, ...
        0, 0);
    TSRest = ...
        calllib('QBladeDLL','getCustomData_at_num',valuestr7, 0, 0);

    DATA.time(i_for2,:) = Time;
    DATA.Omega(i_for2,:) = Rpm;
    DATA.Tg(i_for2,:) = GenTg;
    DATA.TgSWAP(i_for2,:) = GenTgSWAP;
    DATA.U(i_for2,:) = U;
    DATA.Uest(i_for2,:) = Uest;
    DATA.TSR(i_for2,:) = TSR;
    DATA.TSRest(i_for2,:) = TSRest;

end

calllib('QBladeDLL','closeInstance')
```

4.4.2 Individual Pitch Control

This example uses a stripped-down version of the Delft Research Controller specifically designed for dynamic induction control (Pulse [8]) or dynamic individual pitch control (Helix [2]). In this example, the Helix is enabled with an amplitude of degrees and at two different frequencies. The excitation frequency and technique are governed by an input file called discon.in. The values in this file are passed to it through Matlab and the `writeDisconIPC` function.

Main:

```
%%
% Writing DISCON loop.

%%
clear all
close all
clc

%% Define paths
%set the absolute path to your qblade directory here
UserPath = '<userpath>\QBladeCE_2.0.5.2\';
disconPath = UserPath ;
MatlabPath = [UserPath ...
    'MATLAB_Files\4p4_DRC_Cases\4p42_IndividualPitchControl\'];
SourcePath = [UserPath 'Source'] ;
DllPath = [UserPath 'QBladeCE_2.0.5.2.dll'];
addpath('..\Functions');
addpath('..\..\..\');
%%

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
loadlibrary(DllPath,'../../QBladeDLLFunctions.h','alias','QBladeDLL')

m = libfunctions('QBladeDLL') ;

if isempty(m)
    fprintf('Error')
end

%% Defining Settings
Str = [0.25 0.5];

IPC = 0;
Pulse = 0/57.3;
Helix_CM2 = 4/57.3;
Helix_CM3 = 4/57.3;
Freq = Str*9/126*2*pi ;
Pitch_off = 0/57.3;
uservar7 = 0;
uservar8 = 0;
uservar9 = 0;
```

```
41 uservar10 = 0;
42
43 SimNr = length(Freq);
44 tic
45 for i_for1 = 1:1:SimNr
46
47     Sim_name_folder = ['Sim_', num2str(i_for1)] ;
48
49     copyfile([SourcePath],Sim_name_folder);
50
51     cd(disconPath)
52
53     writeDisconIPC(IPC,Pulse,Helix_CM2,Helix_CM3,Freq(i_for1),Pitch_off)
54     cd(MatlabPath)
55
56     run IPC_Simulation.m
57
58     cd(Sim_name_folder)
59
60     save(['Output_',num2str(i_for1)],'PitchAngles')
61
62     cd('..\..\..\..\')
63     delete('discon.in');
64     cd(MatlabPath)
65 end
66 %%
67 toc
68
69 figure;
70 plot(PitchAngles(:,1))
71 hold on
72 plot(PitchAngles(:,2))
73 plot(PitchAngles(:,3))
74 grid on
75 legend('Blade 1','Blade 2','Blade 3')
```

Function:

```
simFile_loc = [MatlabPath Sim_name_folder '\'];

calllib('QBladeDLL','setLibraryPath',DllPath)
calllib('QBladeDLL','createInstance',1,24)
simName = 'NREL5MW_DIPC.sim';
calllib('QBladeDLL','loadSimDefinition',[simFile_loc simName])
calllib('QBladeDLL','initializeSimulation')

simTime = 1000; %in timestep, actual time is timestep*#timesteps
valuestr = 'Pitch Angle Blade 1 [deg]';
valuestr2 = 'Pitch Angle Blade 2 [deg]';
valuestr3 = 'Pitch Angle Blade 3 [deg]';

for i_for2 = 1:1:simTime

    calllib('QBladeDLL','advanceTurbineSimulation')
    calllib('QBladeDLL','advanceController_at_num',[0 0 0 0 0],0)

    Pitch1 = ...
        calllib('QBladeDLL','getCustomData_at_num',valuestr, 0, ...
        0) ;
    Pitch2 = ...
        calllib('QBladeDLL','getCustomData_at_num',valuestr2, 0, 0);
    Pitch3 = ...
        calllib('QBladeDLL','getCustomData_at_num',valuestr3, 0, 0);

    PitchAngles(i_for2,:) = [Pitch1 Pitch2 Pitch3] ;
end

calllib('QBladeDLL','closeInstance')
```

4.5 Two Turbine Simulation

This example will show how to interact with two turbines in the same simulation. For this example, example 2 has been expanded with a second turbine. This second turbine is placed 376 metres downstream (3 rotor diameters). This first wind turbine is set up to use a free-vortex wake to model the aerodynamics. Different to example 2, using the free vortex model generates a wake behind the first turbine that will interact with the second turbine. This second turbine runs a BEM code. Both turbines are controlled using the $K\omega^2$ controller implementation from example 2. Once the wake comes starts to interact with the second turbine it lowers the effective wind speed. The $K\omega^2$ will lower the torque when the wind speed decreases.

NOTE: This example only works with QBlade Enterprise Edition (i.e. QBlade-EE).

```
if libisloaded('QBladeDLL')
    unloadlibrary 'QBladeDLL'
end;

addpath('..\..\');

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
loadlibrary('../../QBladeEE_2.0.5.2.dll',
'../QBladeDLLFunctions.h','alias','QBladeDLL')

m = libfunctions('QBladeDLL') ;

if isempty(m)
    fprintf('Error')
end

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
calllib('QBladeDLL','setLibraryPath','../../QBladeEE_2.0.5.2.dll')

calllib('QBladeDLL','createInstance',1,24)

%this is setup using relative path and depends on the location ...
    of this file, if there are issues use absolute path to these ...
    files
projectFile = '../../Source/NREL5MW_TwoTurbines.qpr';

calllib('QBladeDLL','loadProject',projectFile)
calllib('QBladeDLL','initializeSimulation')

simTime = 1400; %in timestep, actual time is timestep*#timesteps
valuestr = 'Rotational Speed [rpm]';
valuestr2 = 'Gen. HSS Torque [Nm]';
% valuestr2 = 'Gen. Power (w.o. losses) [kW]';
f = waitbar(0,'Initializing Simulation') ;
```

```
33
34  K = 2.24;
35  N = 97;
36  for i = 1:1:simTime
37      calllib('QBladeDLL','advanceTurbineSimulation')
38      omega_WT1 = ...
            calllib('QBladeDLL','getCustomData_at_num',valuestr, 0, ...
            0) ;
39      genTorqueQB_WT1 = ...
            calllib('QBladeDLL','getCustomData_at_num',valuestr2, 0, ...
            0) ;
40      genTorqueQB_store_WT1(i,:) = genTorqueQB_WT1;
41
42      omega_g_WT1 = omega_WT1*N;
43      genTorque_WT1 = K.*(omega_g_WT1*(2*pi/60))^2;
44      genTorque_store_WT1(i,:) = genTorque_WT1 ;
45      V_hub_WT1(i,:) = calllib('QBladeDLL','getWindspeed', -20, 0, ...
            87.6, [0 0 0]);
46
47      calllib('QBladeDLL','setControlVars_at_num',[genTorque_WT1 0 ...
            0 0 0],0)
48
49      omega_WT2 = ...
            calllib('QBladeDLL','getCustomData_at_num',valuestr, 0, ...
            1) ;
50      genTorqueQB_WT2 = ...
            calllib('QBladeDLL','getCustomData_at_num',valuestr2, 0, ...
            1) ;
51      genTorqueQB_store_WT2(i,:) = genTorqueQB_WT2;
52
53      omega_g_WT2 = omega_WT2*N;
54      genTorque_WT2 = K.*(omega_g_WT2*(2*pi/60))^2;
55      genTorque_store_WT2(i,:) = genTorque_WT2 ;
56      V_hub_WT2(i,:) = calllib('QBladeDLL','getWindspeed', 350, 0, ...
            87.6, [0 0 0]);
57
58      calllib('QBladeDLL','setControlVars_at_num',[genTorque_WT2 0 ...
            0 0 0],1)
59      waitbar(i/simTime,f,'Simulation Running')
60  end
61  close(f)
62  % calllib('QBladeDLL','storeProject','Test.qpr')
63  calllib('QBladeDLL','closeInstance')
64
65  figure(1);
66  subplot(2,1,1)
67  plot(genTorqueQB_store_WT1,'LineWidth',1.5)
68  hold on
69  plot(genTorqueQB_store_WT2,'LineWidth',1.5)
70  grid on
71  legend('QB HSS Torque WT 1','QB HSS Torque WT ...
        2','Location','southwest')
72  title('Generator Torque')
73
74  subplot(2,1,2)
75  plot(V_hub_WT1(:,1),'LineWidth',1.5)
76  hold on
```

```
77  plot(V_hub_WT2(:,1),'LineWidth',1.5)
78  grid on
79  legend('Wind Speed Turbine 1','Wind Speed Turbine ...
        2','Location','southwest')
80  title('Wind speed')
```

Figure 3 shows the output of the two turbine example. The interaction between the two turbines starts after 1000 timesteps (roughly 50 seconds). The drop in generator torque is due to the drop in wind speed. As it settles to its new steady state, the small fluctuations in the generator torque that remain are a result of the free-vortex implementation of the wake.

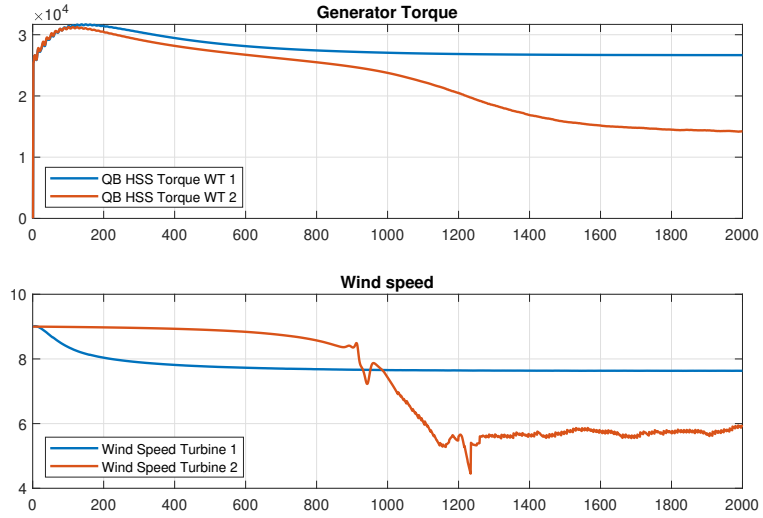

Figure 3: Output of the example code running the $K\omega^2$ controllers for the two turbine simulation.

4.6 Common mistakes

- It is important to remember that in the function **get custom data**, the string should correspond to the label of the graphical data in the GUI and not to the header of the export simulation data.
- When running an n-turbines simulation, check which is the first turbine loaded in the dropdown menu in the simulation setup screen. Note that the order in the dropdown menu is alphabetical and not the order of adding the turbine.
- When QBlade receives an update it is not uncommon for channel names to change. Whenever updating to a new version of QBlade double check the channel names within the GUI.

References

[1] L Brandetti et al. "On the ill-conditioning of the combined wind speed estimator and tip-speed ratio tracking control scheme". In: vol. 2265. Journal of Physics: Conference Series. 2022. DOI: 10.1088/1742-6596/2265/3/032085. URL: https://dx.doi.org/10.1088/1742-6596/2265/3/032085.

[2] Joeri A. Frederik et al. "The helix approach: Using dynamic individual pitch control to enhance wake mixing in wind farms". In: *Wind Energy* 23.8 (2020), pp. 1739–1751. DOI: https://doi.org/10.1002/we.2513.

[3] J Jonkman et al. *Definition of a 5-MW Reference Wind Turbine for Offshore System Development*. Tech. rep. 2009.

[4] National Renewable Energy Laboratory. *ROSCO. Version 1.0.0*. Tech. rep. 2020. URL: https://github.com/NREL/rosco.

[5] S P Mulders. "Wind turbine control: Advances for load mitigations and hydraulic drivetrains". PhD thesis. 2020. DOI: 10.4233/uuid:521577f0-a361-4f92-94c5-02a3bc61ef44. URL: https://doi.org/10.4233/uuid:521577f0-a361-4f92-94c5-02a3bc61ef44.

[6] S P Mulders and J W van Wingerden. "Delft Research Controller: an open-source and community-driven wind turbine baseline controller". In: vol. 1037. Journal of Physics: Conference Series. 2018. DOI: 10.1088/1742-6596/1037/3/032009. URL: https://dx.doi.org/10.1088/1742-6596/1037/3/032009.

[7] S P Mulders et al. "Wind turbine control: open-source software for control education, standardization and compilation". In: Journal of Physics: Conference Series. 2020. DOI: 10.1088/1742-6596/1452/1/012010. URL: https://dx.doi.org/10.1088/1742-6596/1452/1/012010.

[8] Wim Munters and Johan Meyers. "Towards practical dynamic induction control of wind farms: analysis of optimally controlled wind-farm boundary layers and sinusoidal induction control of first-row turbines". In: *Wind Energy Science* 3 (2018), pp. 409–425. DOI: 10.5194/wes-3-409-2018.

5 Appendix

5.1 HelpFunction details

```
[1] loadProject:
    The loadProject library function is used to load a QBlade project
    (.qpr) into the created QBlade instance. loadProject takes the name
    of the project as string.

    Example:
    calllib('<library alias>','loadProject','<projectname>.qpr')

    ------------------------
    [2] loadSimDefiniton:
    The loadSimDefinition library function is used to load a QBlade Simulation export
    (.sim) into the created QBlade instance. loadSimDefinition takes the name
    of the exported simulation as string.

    Example:
    calllib('<library alias>','loadSimDefinition','<projectname>.sim')

    ------------------------
    [3] storeProject:
    Allows to save the project after a simulation has been run.
    storeProject saves the completed simulation as .qpr file

    Example:
    calllib('<library alias>','storeProject','<savename>.qpr')

    ------------------------
    [4] setDebugInfo:
    Enables the option to see debug information from QBlade.
    Takes true or false as input. Unclear if it fully works in Matlab.

    Example:
    calllib('<library alias>','setDebugInfo','true')

    ------------------------
    [5] createInstance:
    Creates a new instance of QBlade for a selected device
    and certain OpenCL parameters. First function to be called
    after having connected to the dll library.

    Example:
    calllib('<library alias>','createInstance',0,24)
```

```
------------------------
[6] closeInstance:
Closes the current open instance of QBlade. Call this function to close
QBLade instance after having run a simulation.
This is particulary important if looping over more than 1 simulation.

Example:
calllib('<library alias>','closeInstance')

------------------------
[7] loadTurbulentWindBinary:
Allows to open a turbulent wind field file in binary data.
Such a file can be exported from QBlade.

Example:
calllib('<library alias>','loadTurbulentWindBinary','<windfieldname>.<extension>')

------------------------
[8] addTurbulentWind:
This function can be used to create a turbulent wind field.
The function has the following inputs, entered into the function in the same order:
windspeed: the mean windspeed at the reference height [m/s]
refheight: the reference height [m]
hubheight: the hubheight, more specifically the height of the windfield center [m]
dimensions: the y- and z- dimensions of the windfield in meters [m]
gridpoints: the number of points in the y and z direction for which velocities are evaluated [-]
length: the simulated length of the windfield in seconds [s]
dT: the temporal resolution of the windfield [s]
turbulenceClass: the turbulence class, can be "A", "B" or "C"
turbulenceType: the turbulence type, can be "NTM", "ETM", "xEWM1" or "xEWM50"
- where x is the turbine class (1,2 or 3)
seed: the random seed for the turbulent windfield
vertInf: vertical inflow angle in degrees [deg]
horInf: horizontal inflow angle in degrees [deg]])

Example:
In this example we will create a turbulent wind field with turbulence class A and Type ETM.
The inflow speed is 14 m/s, at reference height of 89 meters (NREL5MW).
The windfield will be sized on a 50 by 50 grid, spanning 100 by 100 meters, centered at 89 meters.
The wind field will be simulated for 100 seconds, with time steps of 0.05s and finally
it will have no horizontal or vertical inflow angle (i.e. the wind is perpendicular to the rotor plane).
The seed is chosen as a random integer, in this example we take 42:

Function Call:
calllib('<library alias>','addTurbulentWind',14,89,89,[100 100],[50 50],100,0.05,'A','ETM',42,0,0,[])
```

```
IMPORTANT! This function can not be used together with setPowerLawWind.

------------------------
[9] initializeSimulation:
Initiates the simulation.

Example:
calllib('<library alias>','initializeSimulation')

------------------------
[10] setTimestepSize:
Set the simulation step size in seconds [s]. Call this function before initializeSimulation!

Set time step at 0.05 seconds example:
calllib('<library alias>','setTimestepSize',0.05)

------------------------
[11] setRPMPrescribeType_at_num:
Sets the phase in which the RPM is prescribed to the turbine.
Call this function before initializeSimulation!
If the RPM is prescribed no control over RPM is possible. There exist three options:
Prescribed during ramp-up only: 0
Prescribed during entire simulation: 1
No prescribed RPM :2
The first input is prescribed mode, the second input specifies the turbine.
The first turbine can be accessed by passing a 0, the second turbine by 1 etc.

Example with prescribed RPM during ramp-up only:
calllib('<library alias>','setRPMPrescribeType_at_num', 0, 0)

------------------------
[12] setRampupTime:
Set the simulation ramp up time in seconds [s]. Call this function before initializeSimulation!

Set ramp up time to 15 seconds example:
calllib('<library alias>','setRampupTime', 15)

------------------------
[13] getWindspeed:
Gets information of the wind speed [m/s] at a specified x-y-z location.
The fourth input to this function specifies the output vector, this is not used in Matlab.
Passing a 0 for that input was found to work to generate an output.
This output can be passed to any Matlab variable by setting it equal to the calllib.

Example where we read the windspeed at the hub of the turbine (x,y,z --> 0,0,89)
```

```
and pass it to a variable called V_hub:
V_hub = calllib('<library alias>','getWindspeed', 0, 0, 89, 0)

-------------------------
[14] getCustomData_at_num:
Gets information from any of the available data channels in QBlade for the current time step.
All data channels that can be found in the graph view are also available for reading.
The first input is the name of the channel as string, the second input specifies the turbine.
The first turbine can be accessed by passing a 0, the second turbine by 1 etc.
This output can be passed to any Matlab variable by setting it equal to the calllib.

In this example we will read the rotational speed of the turbine
and pass it to a Matlab variable called omega_r.

Example:
omega_r = calllib('<library alias>','getCustomData_at_num','Rotational Speed [rpm]', 0)

-------------------------
[15] setInitialConditions_at_num:
Sets the turbine initial conditions. If multiple turbines are present in the simulation
then the turbine to be set can be specific by passing 0,1,2,etc as final input to the library call.
The first turbine corresponds to passing a 0. If only 1 turbine is present, pass 0 as well.

The following initial conditions can be set:
turbine initial yaw [deg]
collective pitch [deg]
azimuthal angle [deg]
initial rotSpeed [rpm]
Call this function before initializeSimulation!

In the following example we will set the first turbine with an initial rotor speed of 10 rpm
,10 degree initial yaw, 2 degree collective pitch and 0 degree azimuthal angle.

Example:
calllib('<library alias>','setInitialConditions_at_num', 10, 2, 0, 10, 0)

-------------------------
[16] setTurbinePosition_at_num:
Sets the turbine position. If multiple turbines are present in the simulation
then the turbine to be set can be specific by passing 0,1,2,etc as final input to the library call.
The first turbine corresponds to passing a 0. If only 1 turbine is present, pass 0 as well.

Turbine position is specified using x, y, z coordinates and rotation around those axes in degrees.
In the following example we set our first turbine at (630,0,0)
and pitch (rotation around y) is with 10 degrees.
```

```
calllib('<library alias>','setTurbinePosition_at_num', 630, 0, 0, 0, 10, 0, 0)

------------------------
[17] setPowerLawWind:
This function defines a power law wind profile and the inflow direction.
This function can be called after the simulation has been initialized
and can be used to simulate a time varying wind field.
The function has the following inputs:
windspeed: constant windspeed in m/s [m/s]
horAngle: the horizontal inflow angle in degrees [deg]
vertAngle: the vertical inflow angle in degrees [deg]
shearExponent: set to 0 if windspeed is constant with height [-]
referenceHeight: [m]])

In the following example we will set a normal wind shear profile with a wind speed of 10 m/s,
0 degree horizantal inflow direction, 5 degree vertical inflow direction, shear of 1/7 and
reference height of 89 metres

Example:
calllib('<library alias>','setPowerLawWind', 10, 0, 5, 1/7, 89)

------------------------
[18] setControlVars_at_num:
Sets controller variables of the selected turbine. If multiple turbines are present in the simulation
then the turbine to be set can be specific by passing 0,1,2,etc as final input to the library call.
The first turbine corresponds to passing a 0. If only 1 turbine is present, pass 0 as well.

This function does not interact with any dll controller added to QBlade project.
If this function is called after advanceController and before advanceStructure
it will override control inputs from the dll.
The following control parameters can be set directly:
generator torque [Nm]
yaw angle [deg]
pitch blade 1 [deg]
pitch blade 2 [deg]
pitch blade 3 [deg]

 The example here will set the collective pitch of all blades to 1 deg

Example:
calllib('<library alias>','setControlVars_at_num', 0, 0, 1, 1, 1, 0)

------------------------
[19] getTurbineOperation_at_num:
This functions gives 40 different variables commonly used in turbine control.
Data is read for the specified turbine. If multiple turbines are present in the simulation
```

then the turbine to be set can be specific by passing 0,1,2,etc as final input to the library call.
The first turbine corresponds to passing a 0. If only 1 turbine is present, pass 0 as well.
[0] = rotational speed [rad/s]
[1] = power [W]
[2] = HH wind velocity [m/s]
[3] = yaw angle [deg]
[4] = pitch blade 1 [deg]
[5] = pitch blade 2 [deg]
[6] = pitch blade 3 [deg]
[7] = oop blade root bending moment blade 1 [Nm]
[8] = oop blade root bending moment blade 2 [Nm]
[9] = oop blade root bending moment blade 3 [Nm]
[10] = ip blade root bending moment blade 1 [Nm]
[11] = ip blade root bending moment blade 2 [Nm]
[12] = ip blade root bending moment blade 3 [Nm]
[13] = tor blade root bending moment blade 1 [Nm]
[14] = tor blade root bending moment blade 2 [Nm]
[15] = tor blade root bending moment blade 3 [Nm]
[16] = oop tip deflection blade 1 [m]
[17] = oop tip deflection blade 2 [m]
[18] = oop tip deflection blade 3 [m]
[19] = ip tip deflection blade 1 [m]
[20] = ip tip deflection blade 2 [m]
[21] = ip tip deflection blade 3 [m]
[22] = tower top acceleration in global X [m/s^2]
[23] = tower top acceleration in global Y [m/s^2]
[24] = tower top acceleration in global Z [m/s^2]
[25] = tower top fore aft acceleration [m/s^2]
[26] = tower top side side acceleration [m/s^2]
[27] = tower top X position [m]
[28] = tower top Y position [m]
[29] = tower bottom force along global X [Nm]
[30] = tower bottom force along global Y [Nm]
[31] = tower bottom force along global Z [Nm]
[32] = tower bottom bending moment along global X [Nm]
[33] = tower bottom bending moment along global Y [Nm]
[34] = tower bottom bending moment along global Z [Nm]
[35] = current time [s]
[36] = azimuthal position of the LSS [deg]
[37] = azimuthal position of the HSS [deg]
[38] = HSS torque [Nm]
[39] = wind speed at hub height [m/s]
[40] = horizontal inflow angle [deg]

The following example will pass all variables to a Matlab variable called Turb_Out.

```
Example:
Turb_Out = calllib('<library alias>','getTurbineOperation_at_num', ones(1,40), 0)

-------------------------
[20] advanceController_at_num:
Advances the controller dll of the selected turbine. If multiple turbines are present in the simulation
then the turbine to be set can be specific by passing 0,1,2,etc as final input to the library call.
The first turbine corresponds to passing a 0. If only 1 turbine is present, pass 0 as well.

The following variables are written as output of the controller:
generator torque [Nm]
yaw angle [deg]
pitch blade 1 [deg]
pitch blade 2 [deg]
pitch blade 3 [deg]

This output can be passed to any Matlab variable by setting it equal to the calllib.

In this example we will read the control inputs sent to the turbine
and pass it to a Matlab variable called control_out.

Example:
control_out = calllib('<library alias>','advanceController_at_num', ones(1,5), 0)

-------------------------
[21] advanceSimulation:
Advances both the aerodynamic as the structural time step by one

Example:
calllib('<library alias>','advanceTurbineSimulation')

-------------------------
[22] runFullSimulation:
Runs the full simulations for the time specified in the .qpr or .sim file.
You cannot interact with the simulation (get_data for example) when using this option.

Example:
calllib('<library alias>','runFullSimulation')

-------------------------
[23] setLibraryPath:
MANDATORY FUNCTION! Sets the path to the dll location.
The path specified must be absolute for reading license!.

Example:
calllib('<library alias>','setLibraryPath','<userpath>/QBladeEE_2.0.5.2/')
```

```
    ------------------------
    [24] getTowerBottomLoads_at_num:
    Outputs at the bottom of the tower loads of specified turbine.
    If multiple turbines are present in the simulation
    then the turbine to be set can be specific by passing 0,1,2,etc as final input to the library call.
    The first turbine corresponds to passing a 0. If only 1 turbine is present, pass 0 as well.

    The following variables are written as output:
    Example:
    Towerloads = calllib('<library alias>','getTowerBottomLoads_at_num','loads',0)
------------------------
```