TD_xxx: Matlab code for simulation of floating body motions in the time domain

Andrew Hamilton

Contents

1 Documentation

1.1 Introduction

TD_xxx is a collection of Matlab code for computing the motions of one or more floating bodies in the time domain due to excitation by incoming waves. As illustrated in Figure 1, the code takes as inputs the characteristics of the various elements that make up the system (buoy size, tether characteristics, etc) and produces a time history of the motions and forces that will result from a specific set of environmental conditions (sea-state, wind speed, etc). The code is designed to perform these computations in three dimensions with six degrees of freedom, but to date only computations in two dimensions (heave, surge and pitch) have been performed as this tends to be a more useful case.

At its core, the code solves the equations of motion of one or more rigid bodies that are subject to external forces defined individually for each body (gravity, wave forcing, wind forcing, etc) and forces due to connections between each body (tethers consisting of springs, damping elements, or arbitrary force/elongation relationships). Use of the code involves defining the characteristics of each rigid body (mass, size, water-plane area, drag coefficients, etc), and specifying forces that act on each body (gravity, buoyancy, wave forces, tether forces, etc). With this definition complete the code integrates the equations of motion and produces a time-history of motions and forces for each body. Figure 1 illustrates the flow of the solution.

Several important force functions are supplied with the code. The most complicated and challenging to implement is the wave forcing which consists of wave excitation forces, wave-radiation forces, and wave-diffraction forces, following standard linear wave theory. The wave forces are computed from the impulse response functions for the specific body that must be pre-computed by a program such as WAMIT. With those functions supplied, the computation and application of all wave forces is done automatically. Less complicated but equally important are the forces on each body due to gravity, buoyancy, wind, and currents.

A second type of force that applies to pairs of bodies is also included. These forces are defined as functions of the relative positioning of two bodies and apply an equal an opposite force to specified attachment points on each body. The simplest example of this type of force is a spring tether that connects two bodies. In this case the force that applies to each body is a function of the spring constant and the relative position of each body. As the bodies move apart the spring force goes up, and as the bodies move together the spring force goes done. Non-linearities can be included such as a slack condition in which the tether exerts no force on the bodies if they are closer together than some specified distance. As present, there is no facility to solve for the cable-dynamics of the tethers themselves. The effect of drag on the tether can be included in a lumped sense as a force that acts on each body, but not in a way that captures the deformation of the tether from a straight line. Therefore, the code is best suited for relatively short tethers with minimal drag forcing that would cause the tether to deviate from a straight line between the two bodies.

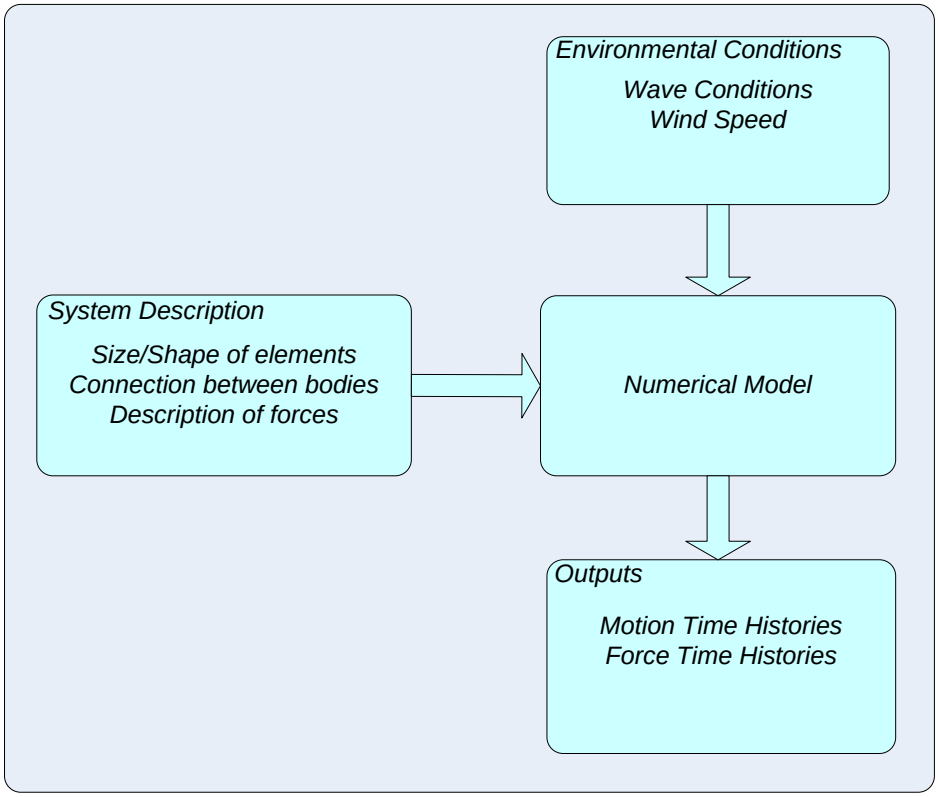

Figure 1: Flow and inputs and outputs through the solver.

1.2 Coordinate Systems

Each rigid body has two coordinate systems particular to it, a body fixed coordinate system and a locally fixed coordinate system. Each locally fixed coordinate system is placed at a known location in the global coordinate system (typically at the equilibrium position of the body). Therefore, as each body moves it is possible to determine the distance between two points on separate bodies by working through the coordinate transformations back to the global coordinate system. Figure 2 illustrates the local coordinate systems located at the equilibrium position of each body. Knowledge of these coordinate systems is helpful when writing new forcing functions that depend on the position of the rigid body relative to the global coordinate system or other rigid bodies.

1.3 Code Specifics

The code utilizes Matlab's object-oriented features and provides classes and functions to allow the solution of problems involving coupled motions of more than one body, subject to a variety of external forcing types. There are two main objects, one which represents the incoming sea-state (IncWave), and the other that represents each rigid body (RigidBody). In

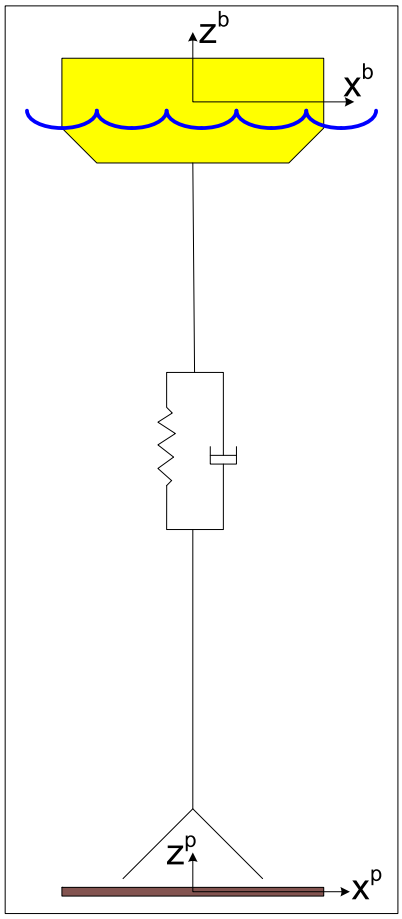

Figure 2: Motions of each body are measured relative to a coordinates system fixed at their equilibrium position.

a typical scenario there will be only one IncWave object and one or more RigidBody instances. Once the user sets up the characteristics of the objects that represent a specific system, the code computes the resulting motion and force histories. As the simulation progresses, the motion histories of each body are stored within the RigidBody objects. The functions that implement forces on each body take one or more RigidBody objects as arguments and therefore have access to the time history of that bodies motions. The wave-radiation forces in particular make use of this since the wave forcing at any given time depends on the bodies position at all past times as well as it's current position, velocity, and acceleration. When the simulation is complete it is only necessary to save the RigidBody objects and the IncWave object to store a complete history of the solution. Typically one Matlab script is used to perform the simulation and separate scripts are then used to read the data back and plot quantities of interest. This way it is convenient to perform a range of simulations across a

parameter automatically, the top level simulation script just needs to save each result in a filename that represents the parameter value.

1.4 Directory Structure

The top level directory of the distribution contains separate directories for the source code, the example simulations, WAMIT results and this documentation as follows:

```
/
├── Doc
├── Solutions
│   ├── SingleBodyExample
│   ├── TwoBodyExample
│   └── ...
├── src
│   ├── @IncWave
│   └── @RigidBody
└── WAMIT_Results
```

- The `Doc` directory contains this documentation and supporting files.
- The `Solutions` directory contains the top-level matlab scripts for various solutions, the distribution contains two examples described in this manual and users should add directories here for their own problems.
- The `src` directory contains the toolkit source including the object definitions, the solver source code, and a variety of force functions. Users can add force functions as described in section 3 to this directory or create them in the `Solutions` subdirectory they create for specific problems.
- The `WAMIT_Results` directory contains WAMIT output files, results are included in the distribution for a few buoy shapes. Shapes different than these require running WAMIT or a similar solver to generate the needed hydrodynamic characteristics for the specific floating body.

2 Examples

This section contains two examples that illustrate the use of the code. The listing shown is the top level Matlab script that the user generates for a given problem, the complete Matlab scripts are contained in the *Solutions* directory of the distribution. This script calls the underlying objects and functions that make up the simulation toolkit. The first example is a single floating body subject to incoming waves. The second example adds a submerged body that is tethered to the floating body by a spring-damper system. These input files are typical of the user generated component of a TD_xxx solution. If a solution only requires existing force functions, generating this top level file is all that is needed to simulate a new problem. If special forcing functions are required to capture the physics of a situation, then those forcing functions need to also be written as described in section 3.3

2.1 Single Body Example

The file *TD_SingleBodyExample.m* contains an example that uses the above classes and functions to solve the wave-excited time-domain motion problem for a freely floating body with three degrees of freedom (surge, heave, and pitch). In this example the body used is a truncated cylinder of draft 0.8m and diameter 2.5m. The hydrodynamic coefficients for this geometry have previously been computed by WAMIT are stored in the directory "WAMITResults" (see WAMIT documentation for details).
First the path to the needed matlab source code is established as relative to the directory this top level file is located in. Alternatively the absolute paths of this location could be added to the Matlab path and this wouldn't be necessary to include.

```
5 close all
6 clear all
7
8 path(path,'../../src');                %Add source code path
```

Next the water density (rho), simulation time (tf), time step size (dt) and the number of time-steps to compute (N) are established.

```
10 grav = 9.81;  %Define gravitational constant (m/s^2)
11 rho = 1025;     %Define water density (kg/m^3)
12 tf = 60;        %Set simulation duration in seconds
13 dt = .1;        %Set time-step size in seconds
14 N = tf/dt;      %Number of timesteps to compute
```

A RigidBody object is defined and the time-step size and physical characteristics of the buoy are assigned. This object represents the floating body in this example, there are

```
16 %Set up "Buoy"
17 Buoy = RigidBody;
18 Buoy = set(Buoy,'name','Surface Buoy');
19 Buoy = set(Buoy,'dt',dt,'N',N);      %Set timesteps
20 Buoy = set(Buoy,'dof',6);            %With 6 DOF.
```

```
Buoy = set(Buoy,'Fixed',0);          %Set free to move
Buoy = set(Buoy,'mass', 2050);       %Set mass in kg
Buoy = set(Buoy,'S',5.0);            %Set waterplane area in m^2
Buoy = set(Buoy,'disp',2.0);         %Set displacement in m^3;
Buoy = set(Buoy,'g',grav);           %Set grav. acceleration constant
Buoy = set(Buoy,'x0',[0 0 0]);       %Equilibrium Pos. in global coord.
Buoy = set(Buoy,'xG',[0 0 -0.38]);   %Set center of gravity in meters
Buoy = set(Buoy,'xB',[0 0 -0.22]);   %Set center of buoyancy in meters
Buoy = set(Buoy,'S11',1.92,'S22',1.92);  %S11 = S22 = 1.92 m^4
I = zeros(3);
I(1,1) = 702.6; I(2,2) = 702.6; I(3,3) = 588.7;
Buoy = set(Buoy,'I',I);              %Set moments of inertial
clear I;
Buoy = set(Buoy,'Area',[1.935 1.935 4.9]); %Set submerged proj. area in m^2
Buoy = set(Buoy,'Cd',[10.0 1.5 1]);  %Set drag coefficients (3 directions)
Buoy = set(Buoy,'AreaW',[.875 .875 0]); %Set area subject to wind forcing
Buoy = set(Buoy,'CdW',[1.5 1.5 0]);  %Set wind drag coefficients
```

The frequency domain hydrodynamic coefficients as generated from WAMIT are read and the infinite frequency added mass coefficients are assigned to the buoy RigidBody object. The function "ReadWamitFD" used here takes a filename as an argument and returns a structure containing the needed data read from the file.

```
%Read Frequency Domain Hydrodynamic Data from file.
filename = ['WAMIT_Results\SingleBodyExample' ];
disp('Reading frequency domain hydrodynamic coefficients');
FD_coefs = ReadWamitFD(filename);
for i = 1:6
   for j = 1:6
      mu_inf(i,j) = rho*FD_coefs.X{i,j}(2);
   end
end
Buoy = set(Buoy,'mu_inf',mu_inf);
clear mu_inf;
```

The impulse response functions that represent the floating bodies wave-hydrodynamic behavior is read and computed by the function "ComputeWamitIRF_coefs".

```
%Compute Impulse Response Functions from Frequency Domain Results
disp('Computing impulse response functions');
L = ComputeWamitIRF_coefs(filename);
```

The incident wave profile is computed next by using the IncWave object. In this example, a monochromatic wave-profile is chosen, with a period of 8 seconds. The time-steps that the wave elevation is computed at is generated by the "Buoy" object, ensuring that the time-steps of the incident wave profile match that of the solution. Because the forces on the body due to the incident wave persist after the peak of the wave has passed the centerline of the body, a longer incident wave profile is needed than the length of the simulation. The time specification in the call to "eta" reflects this.

```
55 %Compute incident wave profile
56 wave = IncWave;
57 wave = set(wave,'A',1);          %Amplitude = 1 meter
58 wave = set(wave,'T',8);          %Period = 8 seconds
59 wave = set(wave,'Type',1);       %Type 1 = monochromatic waves
60 wave = set(wave,'beta',-1);      %Waves moving towards negative x
61
62 disp('Computing incident wave profile');
63 t1= dt:dt:L{7,1}.breaks(end);
64 t = get(Buoy,'t');  %Get timesteps from Buoy object.
65 eta0 = eta(wave,0,[t t(end)+t1]);
```

Separate functions are assigned to the body that compute each of the various forces that act on the body. The "F_HydroStatic" and "F_Weight" functions depend only on characteristics of the Buoy object, so don't require any additional arguments. The function that computes the memory component of the radiation forces requires knowledge of the bodies impulse response functions. Matlab's "Anonymous Function" feature is used to pass this additional argument to the function as a parameter, while still allowing the Mem_Radiation function to be called later with the standard argument list. The function that computes the wave-exciting force requires the impulse response functions and wave elevation history so these are also passed as parameters using the anonymous function feature.

```
67 %Add forces to Buoy
68 Buoy = add_Force_fcn(Buoy,@F_HydroStatic);
69 Buoy = add_Force_fcn(Buoy,@F_Weight);
70 Buoy = add_Force_fcn(Buoy,@(body,RB,n) Mem_Radiation(body,RB,n,L));
71 Buoy = add_Force_fcn(Buoy,@(body,RB,n) F_Exciting(body,RB,n,L,eta0));
72 Buoy = add_Force_fcn(Buoy,@(body,RB,n) F_Drag_Inc(body,RB,n,'Surface Buoy',
       wave));
```

The solution must start from an initial condition and this is established by setting the solution to a prescribed value at the time equal zero timestep.

```
74 %Set Initial Conditions  %Offset position from equilibrium position.
75 Buoy = set_pos(Buoy,[0 0 eta0(1) 0 0 0],1);
```

The problem is now completely set up and the final step is to solve the equations of motion at each subsequent time-steps after an initial condition is established. The "Solve_RK4" function performs the solution and requires that all objects in the system be assembled in a single Matlab "cell array". Because there is only one rigid body in this example the cell array RB only has one cell. After the solution is complete the cell array contents are copied back for convenience in plotting the solution.

```
77 RB{1} = Buoy;
78
79 %Solve for motion
80 RB = Solve_RK4(RB);
81
```

```
82 Buoy = RB{1};
83
84 %Save results for subsequent plotting
85 save('SingleBodyExample','Buoy','wave');
```

At this point the solution is complete and the motion history of the floating body is contained in the Buoy object. The Buoy object and wave object are saved to a matlab file for subsequent plotting and manipulation. For creating a simple plot of basic results that will display when the simulation is complete, the "get" method of the Buoy object is used to make a copy of these values for plotting.

```
87 %Plot Results
88 x = get(Buoy,'x');
89 xdot = get(Buoy,'xdot');
90 xddot = get(Buoy,'xddot');
91 eta0 = eta(wave,0,t);
92 figure('Position',[520 548 864 550]);
93 plot(t,x(:,3),t,xdot(:,3),t,xddot(:,3));
94 set(gca,'FontSize',14);
95 legend('pos (m)','vel(m/s)','accel(m^2/s)','Location','NorthWest');
96 ylabel('Heave motions')
97 xlabel('time (s)');
98 print('-dpdf','SingleBodyExampleMotions');
```

Finally the path is restored:

```
100 rmpath('../../src');
```

The above code computes the motions of a floating truncated cylinder due to an incoming wave-field of unit amplitude and a period of 8 seconds. The incident wave field amplitude ramps up to its final amplitude over several wave periods, as defined by the IncWave object. Figure 3 shows the computed heave motion, which increases in amplitude and achieves a steady state motion after the transient motions introduced by the ramp die away. The drag forces on the buoy included with the "F_Drag_Inc" function are nonlinear and as a result the solution is not sinusoidal as would be the case in a perfectly linear problem.

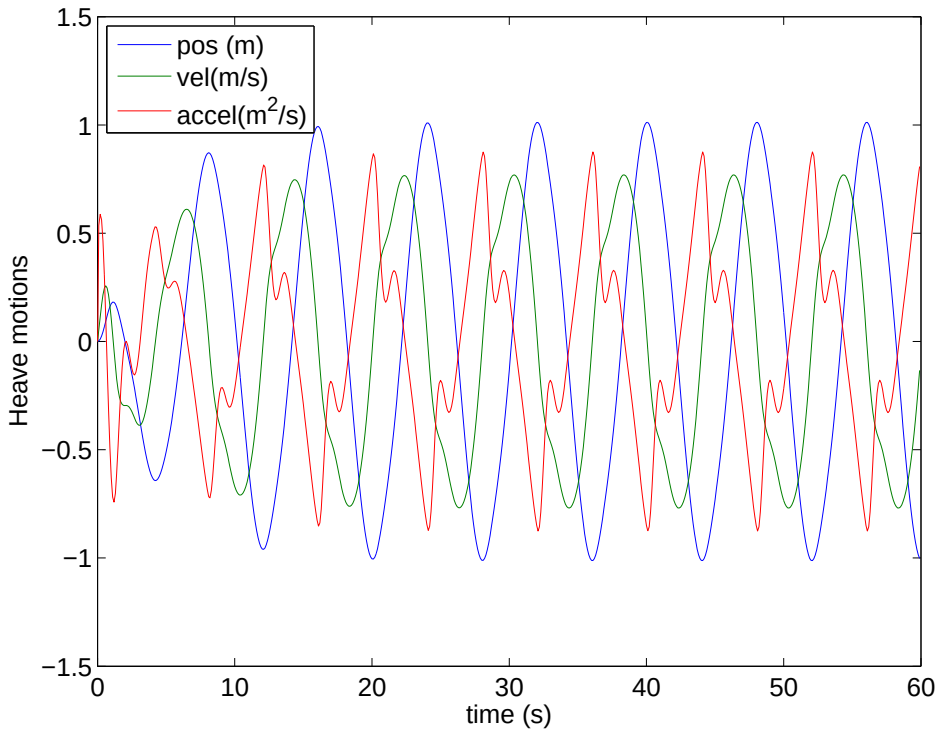

Figure 3: Heave motion results for the single body example.

2.2 Two Body Example

The file *TD_TwoBodyExample.m* contains an example that adds the effects of a second submerged body that is tethered to the surface buoy. The layout of the top level program is similar to the single body example but there is an added forcing function that defines the influence of the tether on both the floating body and the submerged body. This tether is modeled as a spring-damper system and therefore the force it exerts on each body depends upon the positions and velocities of both rigid bodies in the system. The first 49 lines of this example that set up the floating body are the same as the single body example and are not repeated here. However, the second rigid body in the system (submerged in this case) is created by instantiating a second rigid body object with a different name.

```
67 %Set up Submerged Plate
68 S = 10;  %For 10 m^2 plate
69 thickness = 0.9*0.0254;  %(For .9" thick flat plate)
70 displ = S*thickness;  %m^3
71 wet_weight = 1.5*454.54*grav;  %Newtons (1500lbs)
72 mass = wet_weight/grav+displ*rho;  %kg
73 Plate = RigidBody;
74 Plate = set(Plate,'name','Plate');
75 Plate = set(Plate,'x0',[0  0 -20]);  %Equilibrium Pos. in global coords.
```

```
76 Plate = set(Plate,'dt',dt,'N',N);  %Set timesteps
77 Plate = set(Plate,'dof',6);  %With 6 DOF.
78 Plate = set(Plate,'Fixed',0);  %Set free to move
79 Plate = set(Plate,'mass',mass,'S',0,'disp',displ,'g',grav);
80 Plate = set(Plate,'xG',[0.0 0 0]);
81 Plate = set(Plate,'xB',[0.0 0 0]);
82 Plate = set(Plate,'S11',0.0,'S22',0.0);  %m^4
83
84 I = zeros(3);
85 I(1,1) = 693.0; I(2,2) = 3382; I(3,3) = 3965;
86 Plate = set(Plate,'I',I);
87 clear I;
88 Plate = set(Plate,'Area',[2*S  2*S  S]);
89 Plate = set(Plate,'Cd',[.1  .1  1.2]);
90
91 mu_inf = zeros(6,6);
92 Len = sqrt(S);  %Determine square plate dimension from area.
93 mu_inf(1,1) = Len*pi*rho*(thickness/2)^2;
94 mu_inf(3,3) = Len*pi*rho*(Len/2)^2;   %added mass for 2-d ellipse plate.
95 mu_inf(5,5) = (1/8)*Len*pi*rho*((Len/2)^2-(thickness/2)^2)^2;
96
97 Plate = set(Plate,'mu_inf',mu_inf);
```

At this point the submerged plate object has been created and assigned all the necessary properties. The center of gravity and center of buoyancy are located at the same place as is the case for a submerged body of uniform density. The waterplane areas and moments are set to zero because the body is submerged. The added mass numbers are taken from estimates of 2-D flows described by Newman (pp145). Note that the pitch added mass is included and important since we will be applying forcing that can cause motion in three degrees of freedom (Surge, Heave, Pitch).
All the forces that act on the two bodies must be specified, this is done exactly the same as for the single body example for the floating buoy, and similarly for the submerged plate:

```
99 %Add forces to Buoy
100 Buoy = add_Force_fcn(Buoy,@F_HydroStatic);
101 Buoy = add_Force_fcn(Buoy,@F_Weight);
102 Buoy = add_Force_fcn(Buoy,@(body,RB,n) Mem_Radiation(body,RB,n,L));
103 Buoy = add_Force_fcn(Buoy,@(body,RB,n) F_Exciting(body,RB,n,L,eta0));
104 Buoy = add_Force_fcn(Buoy,@(body,RB,n) F_Drag_Inc(body,RB,n,'Surface Buoy',
      wave));
105
106 %Add forces to Plate
107 Plate = add_Force_fcn(Plate,@F_HydroStatic);
108 Plate = add_Force_fcn(Plate,@F_Weight);
109 Plate = add_Force_fcn(Plate,@(body,RB,n) F_Drag_Inc(body,RB,n,'Surface Buoy
      ',0));
```

The submerged plate does not have any wave exciting or radiation forces since it is presumably far enough from the free surface that these forces are insignificant. The `0` as the last

argument of the F_Drag_Inc in place of a wave object indicates that there is no wave forcing and the large scale motion of the water due to the waves should be ignored, appropriate for deep submergence.
At this point the forces due to the tether between the two bodies has not been included. This is a special case since the forcing depends upon the state of more than one rigid body object. Additionally, it is easy to see that the force on one body must be equal and opposite to the force on the other. Here's the code that sets this up:

```
111 %Add forces that act on both bodies
112 p_aB = [0.0  0.0  -0.9];  %Tether attachment point on Buoy (local CoordSys)
      .
113 p_aP = [0.0  0.0  2];  %Tether attachment point on Plate (local CoordSys).
114
115 k = wet_weight/1;  %Spring Elongates 1m to support static wet weight
116 [Spring_Force_Coeffs] = LinearSpringForce(k);
117
118 b = 10000;  %Damping coefficient   N/(m/s)
119 [Damping_Force_Coeffs] = LinearDampingForce(b);
120
121 Buoy = add_Force_fcn(Buoy,@(body,RB,n) SpringDamperTether(body,RB,n,'Plate'
      ,p_aB,p_aP,Spring_Force_Coeffs,Damping_Force_Coeffs));
122
123 Plate = add_Force_fcn(Plate,@(body,RB,n) SpringDamperTether(body,RB,n,'
      Surface Buoy',p_aP,p_aB,Spring_Force_Coeffs,Damping_Force_Coeffs));
```

The first two lines define the position (x,y,z) on each body that the tether is attached, this is important since the magnitude of the moments due to this force depend on the position of attachment. Although it would be simpler to define a forcing function that simply takes a linear spring and damping coefficient as an argument, a Matlab interpolation function is used and the interpolation constants are passed to the force function. There is a performance impact of this, but the forcing function is them more readily adapted to non-linear tether dynamics. If a non-linear spring or damping characteristic is needed, it is just a matter for creating interpolation constants for that relationship instead of re-writing the forcing function itself. With those details set up the force is added to each body in turn using the add_Force_fcn as before. The anonymous function features are again used to pass in additional useful such as the spring and damper characteristics and the attachment points on each rigid body. More importantly however, the forcing function must be told what other rigid bodies are needed to compute the force. In line 105 above a force is being added to the Buoy that depends upon the state of the object named 'Plate' (listed in the fourth argument). In line 107 the same forcing function is applied to the Plate, but in that case the secondary rigid body is the object named 'Surface Buoy'. Also, in this second call the attachment points must be switched so that the first attachment point always applies to the object the force is being added to. Logic in the forcing function determines which body is where when these are called during the simulation and performs all needed sign changes. If in programming errors are made in setting this up, the system can be unstable and the two bodies fly apart instead of settling into the steady oscillation of a stable system.

Everything else in this file is the same as the SingleBody example, the wave object is created and initial conditions are assigned. The solver itself `Solve_RK4` requires a cell array of the rigid bodies. For convenience these are reassigned back to their original names after the solution is complete

```
129 RB{1} = Buoy;
130 RB{2} = Plate;
131
132 %Solve for motion
133 RB = Solve_RK4(RB);
134
135 Buoy = RB{1};
136 Plate = RB{2};
137
138 %Save results for subsequent plotting
139 save('TwoBodyExample','Buoy','Plate','wave');
```

Again the data is stored after simulation for subsequent plotting and analysis. This time the 'Plate' object needs to be saved as well to capture the motion history of the submerged plate. Figure 4 shows the resulting heave motion of the buoy and the plate. Again the non-linear drag causes the results to be non sinusoidal. The occasional high accelerations are due to a nonlinearity in the tether forcing function that models the tether going slack if the buoy and the plate get too close together.

Note the plate positions oscillate around a position one meter below the local coordinate system for the plate rigid body. This is because the spring constant was selected to be such that the weight in water of the plate extends the spring 1 meter.

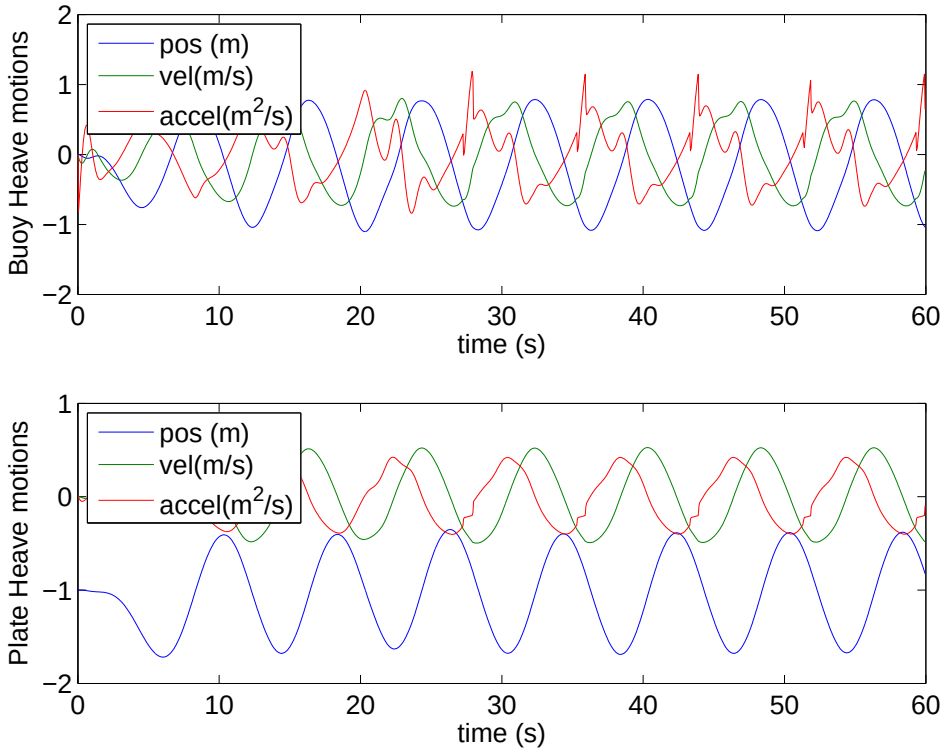

Figure 4: Heave motion results for the two body example.

3 Force functions

3.1 Existing Force Functions

Currently, the code contains routines to implement a variety of forces that could exist in a typical floating body situation. These are documented in detail in the Reference section, but table 1 provides a summary listing. There are two categories of forces, those that depend only on the characteristics and state of a single rigid body in the system (such as gravity or drag forcing), and those that depend on the characteristics of two or more rigid bodies in the system (such as a tether that connects two bodies). Each are added to a problem in the same way using the "add_Force_fcn" function, but in the case of a force that acts on more than one body the same force must be added to each of the bodies it acts on as shown in the two-body example above.

Function Name	Description
F_Weight	Computes forcing due to gravity
F_Hydrostatic	Computes hydrostatic forcing
F_Wind	Computes forcing on rigid body due to windage
F_Exciting	Computes wave exciting forces on floating body
F_Radiation	Computed wave radiation forces on floating body
F_Drag	Computes viscous drag forces due to body motion
F_Drag_inc	Computes viscous drag forces due to body motion and incoming waves
SpringDamper_Tether	Computes forces due to spring/damper connection between two bodies.

Table 1: Forcing functions currently defined in TD_xxx.

3.2 Example Force Functions

3.2.1 Force Depends on One Rigid Body

The simplest forcing functions specify a force that depends only on the most recent state of one rigid body. An instructive example of this is the hydrostatic forcing that provides six degree-of-freedom force vector that is a function of the position of the rigid body. In a case like this the standard argument list for a forcing function is sufficient, this argument list specifies the rigid body the force depends on, the cell-array of all rigid bodies, and the current timestep (n). In this example only the first argument is used but the others are part of the standard argument list since they are often useful as will be seen in the more complicated examples.
The first line of code in `F_HydroStatic.m` identifies the function name, argument list, and returned value.

```
5 function [F_HS] = F_HydroStatic(body,RB,n)
```

All the needed information to perform the hydrostatic force computation is contained in the rigid body object, these values are broken out into local variables for convience.

```
7 g = get(body,'g');
8 rho = get(body,'rho');
9 S = get(body,'S');
10 S11 = get(body,'S11');
11 S22 = get(body,'S22');
12 disp = get(body,'disp');
13 xG = get(body,'xG');
14 xB = get(body,'xB');
15 x = get(body,'sv');
16
17 rhog = rho*g;
```

The computation of the force vector as a function of the floating bodies position and orientation is a standard computation (see Newman, 1977 pp. 293). The six force components

corresponding to force in surge, sway, heave, and moments in roll, pitch, and yaw are computed and stored in a six element vector that the function returns.

```
19 C = zeros(6);
20 C(3,3) = rhog*S;
21 C(4,4) = rhog*(S22+disp*xB(3));
22 C(4,6) = -rhog*disp*xB(1);
23 C(5,6) = -rhog*disp*xB(2);
24 C(5,5) = rhog*(S11+disp*xB(3));
25
26 F_HS = zeros(1,6);
27
28 F_HS(3) = rhog*disp;
29 F_HS(4) = -rhog*disp*xB(2);
30 F_HS(5) = rhog*disp*xB(1);
31
32 for i = 1:6
33    for j = 1:6
34     F_HS(i) = F_HS(i)-C(i,j)*x(j);
35   end
36 end
```

In this simple example the force that was computed depended only on the current position of the rigid body. Much more complicated cases can arise in which the force can depend on time, the time history of the rigid bodies position and/or velocity, or the time-history of the incident wave field. The next example describes such a case.

3.2.2 Force Depends on Time History of Incoming Wave

The linearized wave exciting forces are computed as a convolution integral of the recent wave height history and the wave exciting force impulse response function. The function in `F_Exciting.m` computes this force. The timestep number that is passed into these functions is not an integer, the fractional part indicates which step of the 4th order Runge-Kutta algorithm is calling the force function. This matters for forcing functions that depend on time. A small bit of code separates the fractional part of the timestep and the time-step size itself is recovered from the rigid body object. Additionally, the function call contains two additional arguments, the impulse response function for the wave exciting force, and the wave-height history.

```
5 function F_E = F_Exciting(body,RB,n,L,eta0)
6
7 frac = mod(n,1);
8 n = floor(n);
9 dt = get(body,'dt');
```

The computation to compute the exciting force does not depend on states stored within the rigid body object, but instead depends on the impulse response function and the wave elevation history. The actual convolution integral is complicated but the code below performs

that integration and returns the six element vector of the wave-exciting forces.

```
N = (L{7,1}.breaks(end))/dt;

tau = -(N-1)*dt:dt:dt*(N-1);

n = n + (length(eta0)-1);  %Adjust current time index for padding.
eta0 = [zeros(1,length(eta0)-1) eta0];  %Pad eta0 to represent no waves
    before t = 0.

NN = 2*N-1;

F_E = zeros(1,6);
Integrand = zeros(1,NN);
for i = 1:6
      if (isstruct(L{7,i}) == 0)
        L{7,i} = interp1(tau,L{7,i},'spline','pp');  %Convert only if
            coefficient structure not passed.
      end
        L{7,i} = ppval(L{7,i},tau+dt*frac);
    L_loc = L{7,i};
    for nn = 1:NN
      Integrand(nn) = eta0(n-(nn-N))*L_loc(nn);
    end
  F_E(i) = trapz(tau,Integrand);
end
```

3.3 Adding New Force Functions

The convenience of this code is the ability to easily add just about any needed forcing functions easily without changing the underlying solver. Any function can be implemented that depends on the state of all bodies in the system and any external parameters that are passed to the forcing function. Once that function is written then it can be immediately included in the solution using the `add_Force_function` call. The above examples and others contained in the code show most of what is needed to implement a new force function, a few important points are highlighted here:

- In all cases the functions have access to information stored in the rigid body objects and the timestep number.
- If additional information is needed, it must be passed in as additional arguments.
- The time-step supplied contains a fractional part that indicates which step of the Runge-Kutta algorithm is calling this function.

4 Reference

4.1 Classes

In addition to the methods described below for each class, each class has methods to *set/get* properties and display the object. Additional methods for each class are documented below.

4.1.1 IncWave

The IncWave class contains parameters that describe an incident wave field and methods that return the appropriate wave elevation at a specified location and time. Currently three types of incident wave are supported, a wave field that is characterized by the Pierson-Moskowitz spectrum, and a monochromatic incident wave profile, and the ability have the object generate a wave-elevation time history that has a specified spectrum. In each case, the wave elevation is zero everywhere at time zero and ramps to the full amplitude specified over the first few cycles of motion.

Properties

- Type – Wave type: 0=Pierson-Moskowitz Spectrum, 1 = monochromatic wave, 2 = specified spectrum.
- A – Wave Amplitude
- T – Wave Period
- g – Acceleration due to gravity in appropriate units relative to A and T.
- n_phases – Number of phases to use in time history generation from spectrum. Default = 300.
- $freq$ – Specified frequencies, array of arbitrary length.
- *spectrum* – Specified wave spectrum, array of length equal to the number of specified frequencies.
- *beta* – Wave Direction. +1 = moving towards positive x, -1 = moving towards negative x.

Methods

- IncWave() – Constructor, creates default object with Type = 1, $A = 1$, $T = 2$, $g = 9.81$
- eta(p,x,t) – Method to compute and return wave elevation at specified position and time. Method returns an array of values with the number of rows equal to the dimension of x and the number of columns equal to the dimension of t. The p argument is an IncWave object.

- eta(p,x,t) – Method to compute and return wave elevation height rate of change at specified position and time. Method returns an array of values with the number of rows equal to the dimension of x and the number of columns equal to the dimension of t. The p argument is an IncWave object.

- vel(p,x,z,t) – Method to return the fluid velocity due to the wave-field at any position and time. x and z specify the position and t specifies the time. This method returns a three element vector that represents the x, y, and z velocities respectively.

4.1.2 RigidBody

The RigidBody class represents the characteristics, position, and velocity of a rigid body. The class can be assigned one or more forces that act on the body and that are a function of the body's position, velocity, acceleration, and time. Each RigidBody object contains the position, velocity, acceleration, and force history for the body it represents. External functions that compute forces on the body have access to these to allow forces to be dependent on past motions of the bodies involved as well as the current state.

Properties

- *dof* – Degrees of freedom, this is the number of positions, velocities, and accelerations that are kept track of by this object.

- *dt* – Time-step size.

- *n_tsteps* – Time-step size.

- *t* – Array of times the position is represented at (i.e. 0:dt:n_tsteps*dt).

- $x0$ – (x,y,z) components of the equilibrium position of the body in the global coordinate system.

- x – dof components of the bodies position at each time-step. Is an array with dimension [n_tsteps X dof].

- *xdot* – dof components of the bodies velocity at each time-step. Is an array with dimension [n_tsteps X dof].

- *xddot* – dof components of the bodies acceleration at each time-step. Is an array with dimension [n_tsteps X dof].

- *Fixed* – Flag indicating if the bodies position is to be considered prescribed or is to be solved for.

- *mass* – Bodies mass.

- *disp* – Bodies displacement at equilibrium.

- S – Bodies undisturbed water-plane area.
- $S11$ – 2nd moment of water-plane area, about x-axis.
- $S22$ – 2nd moment of water-plane area, about y-axis.
- xG – Bodies center of gravity (in body fixed coordinates)
- xB – Bodies center of buoyancy (in body fixed coordinates in equilibrium position)
- g – Acceleration due to gravity. (default = 9.81)
- rho – Density of fluid body is floating in. (default = 1025)

Methods

- RigidBody() – Constructor, creates default object
- MassMatrix(p) – Returns the mass matrix for the body based upon the properties $mass$, xG, and I.
- add_Force_fcn(p,fcn) – Adds a function to the list of functions that are called by the object when evaluating the forces acting on the body. fcn is a function handle for a function that takes a RigidBody object and a time-step specification as arguments. i.e. fcn(RigidBody,n). Additional parameters can be passed to the fcn through the "anonymous function" features of Matlab. See section 3 for details.
- SumForces(p,n) – Iterates through all force functions that have been added to the object, calls those functions to compute the force acting on the body at time-step n, and creates a sum that represents the total force vector acting on the body at time-step n.
- RHS(p,n) – Combines the bodies mass matrix, infinite frequency added mass matrix, and the result from SumForces into the a right hand side vector evaluated at time-step n.

4.2 Functions

There are a number of functions that take various objects as arguments but that aren't part of a class. Each RigidBody object can be assigned a collection of functions that compute the force on the body based upon the bodies properties (position, velocity, acceleration history are most useful), and a time-step specification. In order to present a uniform interface to the RigidBody class, these functions must have an argument list of (p,n), where p is a RigidBody object and n is the time-step specification. If one of these functions requires additional parameters that are not a property of the RigidBody class, these parameters can be passed to the function by using the "anonymous function" feature of Matlab. In this way, the additional parameters are specified when the function is added to the RigidBody class

(using add_Force_fcn). Because the parameters are assigned to the function at this time, the parameters are fixed and can not change through the simulation. In this reference section the force functions that are part of the distribution are listed with their arguments and a short description. Not all of the functions listed here can be used directly as a force function, some are simply auxiliary functions that are called by the top level script or by actual force functions.

Functions

- F_Weight(body,n) – Returns force and moment vector due to the bodies weight.
- F_Hydrostatic(body,n) – Returns force and moment vector due to the hydrostatic force, which depends on the bodies position for surface piercing bodies with $S > 0$.
- F_Exciting(body,n,L,eta0) – Returns the wave exciting force and moment vector due to the incident and diffracted wave field. This force depends upon the local wave elevation $eta0$, and the incident wave impulse response function, contained in L.
- Mem_Radiation(body,n,L) – Returns the memory component of the force due to the waves radiated by the bodies motion. This force depends upon the radiation impulse response function, contained in L.
- ReadWamitFD(filenm) – Reads frequency domain hydrodynamic data from file created by WAMIT program.
- ReadWamitIRF(filenm) – Reads impulse response function hydrodynamic data from file created by WAMIT program.
- L = ComputeWamitIRF_coeffs(w,F,tau) – Convert frequency domain hydrodynamic data to time domain impulse response functions (L) as a function of time-steps specified in tau. w is a vector of frequencies that the hydrodynamic coefficients in F are computed at.
- Solve_RK4(bodies{N}) – This is the main solver that takes a cell array of rigid bodies and computes