USER MANUAL
version 2.0.1

MOODY
Mooring Dynamics

JOHANNES PALM
CLAES ESKILSSON

Department of Mechanics and Maritime Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2019
Contact information: johannes.palm@chalmers.se

CONTENTS

1 Introduction

1.1 General description

Moody is developed as a module for computing cable dynamics, aiming primarily at marine applications. All examples in this manual are therefore different types of mooring configurations for an offshore structure. This manual aims to explain the usage of Moody as a stand-alone solver for mooring dynamics, and as a plug-in module to hydrodynamic codes via the Moody API. When using this version of moody for scientific purposes, please make sure to cite:

1. J. Palm, C. Eskilsson, and L. Bergdahl. An hp-adaptive discontinuous Galerkin method for modelling snap loads in mooring cables. *Ocean Engineering*, 144:266–276, 2017
2. J. Palm. *Mooring Dynamics for Wave Energy Applications*. PhD thesis, Chalmers University of Technology, 2017

Moody comes with a suite of MATLAB® scripts mainly used for post-processing. These are optional to use and are simply provided as a way of loading and processing the output data from Moody. All matlab functions and scripts, including the API-equivalent are found in `API/matlab`. The output data format of Moody is presented in chapter 4.

In stand-alone mode, Moody is most easily operated though the terminal window. Automated batch runs are typically made through bash-scripting or any other scripting language of choice. Alternatively, a MATLAB interface for running Moody is included. The simulation is controlled through an input file and standard flag-value pairs e.g. `-f <fileName>`. This mode is used to make stand-alone, uncoupled simulations of cable dynamics. Moody has an automated program interface (API) for coupling to external solvers, e.g. for hydrodynamics. The setup is very similar to using Moody in stand-alone mode with only minor changes to the input file.

The present version of Moody is a discontinuous Galerkin finite element method with high-order Legendre polynomial expansion bases. The formulation is in conservative form, employing a local Lax-Friedrich type Riemann solver for the numerical fluxes between the elements. For more information on the numerical schemes and the equations of motion, the reader is referred to the theory manual.

Moody is a stand-alone c++ code with no dependencies apart from the std-library. It was originally prototyped in Matlab, which is why many input files are labelled '.m' in the tutorials. This is not a requirement of Moody, and any text-file format would serve well as input file for a simulation.

1.2 Installation

Moody itself is released in pre-compiled form, however the API coupling code is released open source. Therefore (as of v.2.0.1), only the binaries are packed in .tar.gz files for each operating system (OSX, Linux and Windows). The compressed files includes the `etc, bin, lib` and `include` folders. The tutorials, the manual and the API are all accessed directly as sources when cloning the repository.

1. Clone or download the repository from `github.com/johannep/moodyAPI`.
2. Select your installation directory and unpack the appropriate tar-file binaries IN THE SAME LOCATION. Technically, this is only relevant for all tutorials and matlab-processing paths to work as intended.
3. In Linux or OSX environments, source the environment variables. For this, use `etc/bashrcMoody`. **NB: There is no Windows PATH-setting script provided at this time.**

Ex:

```
cd moodyAPI;
tar -xzvf moody-Linux.tar.gz;
source etc/bashrcMoody
```

1.3 Getting started

To run a 2.5 s simulation on the lindahl125.m input file, and place output one level up, do:

```
cd $moodyDir/tutorials/catenaryValidation;
moody.x -f lindahl125.m -o ../catenaryResults -time.end 2.5
```

To run tests with two different mesh resolutions on a case, do:

```
moody.x -f inputFile -o N10 -addInput cableObject1.N 10
moody.x -f inputFile -o N20 -addInput cableObject1.N 20
```

To post process a case in Matlab, use `readCase` to load cable data into a Matlab-structure. View tension propagation as a movie every tenth saved timestep through the `moodyMovie` function.

```
c=readCase('oneDShockOutput');
moodyMovie('T',c,c.t(1:10:end));
```

For tests using the API please see the API description in ch. 5.

1.4 Debug tip

If there are any problems running the software, a good starting point is to check the output of the following commands (in a Linux environment):

```
echo $moodyPath
echo $moodyDir
which moody.x
ldd libmoody.so
echo $LD_LIBRARY_PATH
echo $PATH
```

2

The input file

2.1 Overview

An input file is used to setup the mooring system to be studied. An overview of the required input data is given below.

General settings
: Controls environmental aspects like fluid density and gravity. Individual (top-level) keywords.

Time settings
: Controls time step size, start and end time of simulation, etc. Grouped under keyword `time`.

Vertices
: Specifies the location and the number of vertices in the simulation. A top-level array named `vertexLocations`.

Cable types
: Specify the cross-sectional and material parameters of a mooring cable. Grouped for each type under keyword `cableTypeX`.

Cable objects
: Specify cable properties like end-points, cable type and length. Grouped for each cable under keyword `cableX`, (old naming cableObjectX also works, but they do not function in combination).

Rigid bodies
: Dynamic rigid bodies are submerged Morison bodies, like floaters and clump-weights. Defined for each body under keyword `rbX`.

Boundary conditions
: Specify type and behaviour for all boundary conditions. Grouped for each boundary under keyword `bcX`.

Ground settings
: Specifies ground model type and characteristics. Grouped under keyword `ground`.

API settings
: When used as a mooring module, this contains API-specific information. Grouped under keyword `API`.

Print settings
: Print settings control the output format of Moody. Grouped under keyword `print`.

Numerical settings
: Numerical settings control the quadrature order of the simulation. Grouped under keyword `numSet`.

Static solver
: A way to relax the analytic initial conditions. Grouped under keyword `staticSolver`.

The input file can have any text-file format, with or without extension. It follows the format of a Matlab script, where each variable name is assigned a value via the = sign, e.g. `time.start = 5` will set the start time of the simulation to 5s. Multiply defined variables are allowed, but be aware that only the first instance of a keyword will be used. Sub-structures and fields of information are assigned through the `.` sign. The following sections describe each category of information required to setup a Moody simulation.

2.2 General settings

The general settings control the simulation environment.

`dimensionNumber(3)`
: The input is 1, 2 or 3 and defines a 1D, 2D or 3D simulation inside Moody. Default value 3 is recommended. Not all features are supported in 1D and 2D. All vector values in the input file must still be set in 3D format. The input order of ALL vector values in the input file is `[x,y,z]`. The values used are then extracted according to:

  `dimensionNumber=1` $\Rightarrow$ 1D simulation in [x] only

  `dimensionNumber=2` $\Rightarrow$ 2D simulation in [x,z] only

  `dimensionNumber=3` $\Rightarrow$ 3D simulation in [x,y,z]

`gravity` (1)
: Gravity $\left(9.81[\mathrm{m/s^2}]\right)$ can be toggled on or off by input value 1 or 0.

`waterLevel` (inf) [m]
: This refers to the still water level z-coordinate. Thus everything is assumed to be submerged by default.

`waterDensity` (1000.0) [kg/m^3]
: The water density applies to all fluid below the water level.

`airDensity` (0.9) [kg/m^3]
: The air density applies to all fluid above the water level.

```
 % Optional %
dimensionNumber = 3;
gravity = 1;
waterLevel = 0.0; % [m]
waterDensity = 1000.0; % [kg/m3]
airDensity = 1.0; % [kg/m3]
```

2.3 Time settings

The required time settings are described in detail below.

`time.start` [s]
: Start time of the simulation.

`time.end` [s]
: End time of the simulation.

`time.dt` [s]
: Time step size.

`time.scheme` (RK3)
: A string specifying the choice of integration method. Choices are:

  `eulerFwd`
  : The time step scheme is based on the simple 1st order Euler forward method. It is not recommended for use..

  `RK3`
  : Third order explicit Strong Stability Preserving Runge-Kutta scheme. This is the default.

  `RK45`
  : Five stage fourth order explicit Runge-Kutta scheme.

  `AB3`
  : Third order Adams-Bashford scheme. Use for computational efficiency but not in combination with adaptive time stepping (i.e. `cfl`).

`cfl`
: If present, it tells Moody to adapt the time step size of the simulation to match a given fraction of the CFL condition of the local cable mesh. Typically `cfl=0.9` works well for most applications, but in cases dominated by ground a lower value is often needed for stability. If field `cfl` is present, field `dt` is not used.
Also activated by keyword `courantNo`.

```
time.start = 0;
time.end = 10;
time.dt = 1e-4;
time.scheme= 'RK3';
```

2.4 VERTICES

Moody's geometry definition is based on vertices that define points in three dimensional space. A vertex is needed to assign a physical location to any cable or boundary condition object. The cell array (Matlab style) `vertexLocations` is used for this purpose. It is important to note that all points must be written in 3D coordinates, see the description of the `dimensionNumber` input in section 2.2.
The following example creates vertex `1` and `2` at points `[0 0 -50]` and `[50 0 0]` respectively.

```
vertexLocations = {
1 [0 0 -50];
2 [50 0 0]
};
```

2.5 Cable types

A cable type is used to define the material properties of a cable. Hydrodynamic coefficients, diameter, mass per meter and constitutive relations are all defined here. When defining the actual cable objects, several objects can be of the same cable type. It is analogous to buying a nylon rope, and cutting it into three pieces. The nylon rope properties are contained in the cable type, and the three pieces become three cable objects.
Cable types are defined as structures with a number following the key name `cableType`, and fields specifying the properties. Several cable types can be specified in the same simulation (e.g. `cableType1`, `cableType2`, ...). The required and optional fields of the cable type structure are described below.

`diameter` [m]
: The diameter (nominal diameter in the case of chains) of the cable. If not specified, the default value is computed from `gamma0` and `rho`, assuming a constant density in a circular cross-section.

`gamma0` [kg/m]
: The mass per meter of the cable. If not specified, the default value is computed from `diameter` and `rho`, assuming a constant density in a circular cross-section.

`rho` [kg/m^3]
: The density of the cable material. If not specified, the default value is computed from `diameter` and `rho`, assuming a constant density in a circular cross-section.

`materialModel`
: The material model is a substructure of the cable type. It contains information about the constitutive relations of the cable. See section 2.6 for detailed info.

Optional (parenthesis indicates default value)

`CDn` (0)
: The drag force coefficient in the normal direction of the cable.

`CDt` (0)
: The drag force coefficient in the tangential direction of the cable.

`CM` or `CMn` (0)
: The added mass coefficient of the cable in the normal direction of the cable.

`CMt` (0)
: The added mass coefficient in the tangential direction of the cable.

```
cableType1.diameter = 0.05;
cableType1.gamma0 = 1.5; % (default:  rho*area)
cableType1.rho = 7800; % (default:  gamma0/area)
cableType1.materialModel.type = 'bilinearCable';
cableType1.materialModel.EA = 1e4;
% Optional %
cableType1.CDn = 1; % (default:  0)
cableType1.CDt = 0.1; % (default:  0)
cableType1.CM = 2; % (default:  0)
```

2.6 Material models

The material model is a substructure of the `cableType` and contains the information about the constitutive relation between axial force and strain in the cable. Moody uses the cable elongation ($\delta L/L_0$) as strain input to the force-strain relation.

2.6.1 *Common fields*

`EA` [N]
: The mean axial stiffness of the cable. In nonlinear materials, this value is used as an approximation when computing the initial condition of the cable analytically.

`type`
: A string specifying which available type to use. Each type has its own additional input fields, described below.

`xi(0)` [Ns]
: The linear internal damping coefficient of the material. The internal damping force is added to the tension force from the material type and is computed as a linear strain-rate dependence as $T_\xi = \xi\dot{\varepsilon}$.

2.6.2 *Available types*

`type=biLinear`
: Computes the axial tension force of the cable according to Hooke's Law, but disallows compressive forces. Thus $T = \max(EA\,\varepsilon, 0)$.

`type=exponential`
: This type uses an exponential strain-force relation as:

$$T = \max\left(0, K\left(e^{a\varepsilon} - 1\right)\right). \tag{2.1}$$

Additional fields of input are:

`K` [N]
: Basic stiffness. See eq. (2.1).

`a`
: The growth-rate of the exponential function. See eq. (2.1).

`type=polynomial`
: Implements strain-force relation as polynomial according to

$$T = \max\left(0, \sum_{i=1}^{m} C_i \varepsilon^i\right), \tag{2.2}$$

where C_i is the polynomial coefficient of order i as defined by the `C` field below. Additional fields of input are:

`C` [N]
: A vector set of coefficients. First value is the linear coefficient, second is the quadratic strain dependence and so forth. The length of the vector specifies the order of the polynomial strain-force curve. Ex:
`materialModel.C = [1.0e3 2.0e3 3.0e3 4.0e3];`

2.7 Cable objects

Cable objects are the main object type in Moody. It specifies a cable of a given unstretched length between two vertices. Naming convention following `cable1`, `cable2`, etc. is used to define the cable objects in the input file.

`typeNumber`
: The cable type number.

`startVertex`
: The start vertex number of the cable. This will be used as $s = 0$, where s is the unstretched cable coordinate from 0 to L.

`endVertex`
: The end vertex number of the cable. This will be used as $s = L$, where s is the unstretched cable coordinate from 0 to L.

`length` [m]
: The unstretched length (L) of the cable. For some types of initial condition, e.g. `PreStrain`, the field is unused and is overwritten by the computed value.

`IC`
: The `IC` field is a substructure that defines the initial conditions of the cable. There are several options for the type of initial conditions and they are described in section 2.8.

`N`
: The number of elements in the cable.

`P`
: The polynomial order of the Legendre polynomial basis functions. In this release of Moody, this option is not available. A constant P=4 is always used.

```
cable1.typeNumber = 1;
cable1.startVertex = 1;
cable1.endVertex = 2;
cable1.length = 100; % [m]
cable1.IC.type = 'PreStrain';
cable1.IC.eps0 = 0.05; % [-]
cable1.N = 10;
```

2.8 Initial Conditions

The initial conditions are specified individually for each cable object, as a substructure of `cable`.

2.8.1 *Common fields*

`type`
: A string specifying which available type to use. Each type has its own additional input fields, described below.

2.8.2 *Available types*

`type=PreStrain`
: The cable is set to a straight line between the two end points. The length of the cable is recomputed based on the distance between the points and the specified initial pre-strain. Gravity is not taken into account in this initial condition.

  `eps0`
  : The initial pre-strain of the cable. Input is dimensionless and can be either a single value or a vector of values. If more than one value is specified, the length of the vector must match the length of `parts`, see below. Each part of the cable will then be given the matching pre-strain.

  `parts` (1.0)
  : The relative part of the cable subject to the strain specified in `eps0`. The input is a vector of the same size as `eps0` and the sum of all the values must be 1. The values of `parts` relates to the unstretched length fractions of the cable.

```
IC.type = 'PreStrain';
IC.eps0 = [0.01; 0.02];
IC.parts = [0.5; 0.5];
```

`type=CatenaryStatic`
: The cable is set to be an elastic catenary shape at rest. This option only works in 2D and 3D, when gravity is turned on. Ground interaction is detected analytically, so the cable is initially set to lie still exactly at the ground level. No further input is required.

`type=HalfSine`

Halfsine type is an extension of type `PreStrain`. A sinusoidally varying displacement is imposed on a pre-strained cable. The sinusoidal displacement is in the vertical direction only.

`amplitude` (0) [m]

The amplitude of vertical displacement. When set to 0, the result is the same as for type `'PreStrain'`.

`periods` (0.5)

The number of sinusoidal periods to use along the cable.

`eps0` (0.0)

The desired pre-strain in the cable before the sinusoidal displacement. The input is a single value that is applied to the whole cable.

```
IC.type = 'HalfSine';
IC.amplitude = 1; % [m]
IC.eps0 = 0.01;
% Optional %
IC.periods = 0.5;
```

2.9 Boundary Conditions

Boundary conditions are created through the input file in the same `structure.field` way as other objects.

2.9.1 Common fields

There are several modes of boundary conditions, requiring different fields of input, but some values are common for all.

`type`
: The type describes if the boundary condition is governing the force (Neumann condition) or the position (Dirichlet condition) of the point. The input is either in string format as `'dirichlet'` or `'neumann'`, or a 3 by 1 cell array where the mode of each dimension is specified individually.

`vertexNumber`
: Although several objects can be connected to the same vertex, each boundary condition can only be applied to one. The input specifies the vertex number to apply the boundary condition to.

`startTime` (startTime) [s]
: When defined, the boundary value is held fixed at that of `bcX.startTime` For $t <$`bcX.startTime`. The input is either a single value or a 3 by 1 vector specifying the start time of each coordinate direction independently. It has no effect on `mode='fixed'`. The input is in [s] and the default value is the same as the global start time of the simulation.

`endTime` (endTime) [s]
: Defines the time at which the boundary condition stops to be active. For $t >$`bcX.endTime`, the boundary value is held fixed at that of `bcX.endTime`. The input is either a single value or a 3 by 1 vector specifying the end time of each coordinate direction independently. It is not applicable to `mode='fixed'`. The input is in [s] and the default value is the same as the global end time of the simulation.

`rampTime` (0) [s]
: Dynamic boundary conditions are ramped up from static start time conditions over a given time interval $\Delta\tau$=`bcX.rampTime`. The ramp factor Q is defined as a function of $\tau = t-$`startTime` as

$$Q = 0.5\left(\sin\left(\pi\frac{\tau}{\Delta\tau} - \frac{\pi}{2}\right) + 1\right) . \tag{2.3}$$

`dampTime` (0) [s]
: Analogous to `rampTime` for smoothly stopping the motion of a boundary condition as the end time is approached.

`mode`
: The input is a string specifying which of the implemented modes that will be used. Each mode has its own additional input fields, both compulsory and optional. They are described below.

2.9.2 Available modes

`mode=fixed`
: The fixed mode is used to describe a stationary point / constant force.

  `value` (vertexLocation alt. [0,0,0]) [m alt. N]
  : Specify the boundary condition value. If specified for dirichlet degrees of freedom, this value has precedence over information in the vertex location list.

`mode=sine`
: This mode is used to generate sinusoidally varying values at the boundary. The offset, amplitude, frequency and phase of the sine motion can be specified for each coordinate direction. Scalar valued input is applied to all directions.

  `amplitude` ([0,0,0]) [m alt. N]
  : The amplitude of the sinusoidal excitation.

  `frequency` ([0,0,0]) [Hz]
  : The frequency of oscillation.

  `phase` ([0,0,0]) [deg.]
  : The phase of the excitation.

  `centerValue` (vertexLocation alt. [0,0,0]) [m alt. N]
  : If defined it determines the offset around which the oscillation takes place.

`mode=externalPoint`
: This mode is used in MoodyAPI. It uses quadratic interpolation to generate intermediate boundary conditions in between coupling times of the external solver. In MoodyAPI, the type must be `dirichlet`, as the coupling is assumed to control the motion of a vertex in Moody, and return a cable force to the external solver. No further input is required in MoodyAPI. For standalone Moody, a quadratic interpolation can be set using the following fields, in combination with `bc.startTime` and `bc.endTime`:

  `startValue` (vertexLocation alt. [0,0,0]) [m alt. N]
  : The start value defines the value of the boundary at the beginning of the simulation.

`endValue` (vertexLocation alt. [0,0,0]) [m alt. N]
: The end value defines the value of the boundary at the `endTime`.

`startVelocity` ([0,0,0]) [m/s alt. N/s]
: The initial rate of change of the boundary at `startTime`.

`mode=externalRigidBody`

MoodyAPI comes with a rigid body framework, where the position state of a rigid body is used as input to the API, and where the total mooring forces and moments are returned to the external solver. As in `externalPoint`, only `type=dirichlet` is allowed for use with the API.

`inputDoFs` (6)
: The value must be 6 or 7. If 7, the input values are interpreted as a septernion. If 6, input is point position followed by radians of roll, pitch and yaw respectively.

`slaves`
: A list of vertices whose motions are governed by the rigid body.

`slavePositions`
: A list of slave positions relative to the control point position of the body (the vertexNumber of the BC). The order of the points correspond to the order in which the `slaves` vertices are listed.

```
bc1.type = 'dirichlet';
bc1.mode = 'sine';
bc1.vertexNumber = 2;
bc1.amplitude = [1;0;1]; % [m]
bc1.frequency = [0.5;0;0.5]; % [Hz]
bc1.phase = [90;0;0]; % [deg]
% Optional %
bc1.centerValue = [0;0;0];
bc1.rampTime = 5; % [s]
bc1.startTime = 0;
bc1.endTime = 100;
```

2.10 Ground models

There is at present only one type of ground model implemented in Moody. The `springDampGround` ground model is a combined linear spring and bilinear damper. The contact force acting on each quadrature point of the cable from the ground is a function of the penetration depth and the vertical velocity. Dynamic friction is also included. The input parameters listed below refer to the equations stated in the theory manual of appendix A.

`type`
: The `type` field must be set to `'springDampGround'`.

`level`
: This sets the level of the ground in the global coordinate system. The ground is assumed to be horizontal and thus have its normal vector aligned with the global z-axis. Goes into (A.29) as z_0.

`stiffness` [Pa/m]
: The stiffness of the ground. Goes into (A.29) as K.

`dampingCoeff` (1)
: The damping coefficient specifies the fraction of critical damping of the ground. Energy is only dissipated during the penetration of the ground. During the lifting phase, no vertical damping force is applied. The default value is 1, meaning critical damping. Goes into (A.29) as ξ.

`frictionCoeff` (0)
: This is the friction coefficient between any cable and the ground. Goes into (A.30) as μ.

`vc` (0.001) [m/s]
: The cut off speed for dynamic friction specifies the horizontal speed at which the friction force reaches its full value. This is a numerical relaxation of the friction force for the times when the cable is changing direction. Goes into (A.30) as v_μ.

```
ground.type = 'springDampGround';
ground.level = -50; // ground.stiffness = 1e6;
% Optional %
ground.dampingCoeff = 1;
ground.frictionCoeff = 0.3;
ground.vc = 0.1;
```

2.11 API SETTINGS

See the API chapter **??** for information about API control parameters in the input file.

2.12 PRINT SETTINGS

Print settings control the output folder content and format.

`dt` (time.dt) [s]
: The time between each output time in the result directory.

`mode` (1)
: The `mode=1` prints all cable values in both modal and nodal space. `mode=0` prints only modal coefficients. Use moodyPost.x to extract nodal output, see chapter 4.

`format` ('bin')
: Moody prints in binary format by default. `format='txt'` or `format='ascii'` prints in ASCII format. Binary format is significantly faster and more efficient for storage.

2.13 NUMERICAL SETTINGS

Numerical settings control the quadrature order of the simulations. The top-level keyword is `numLib`.

`qPointsAdded` (2)
: The number of quadrature points more than the polynomial order P, to use in the simulation. Default is 2, meaning that $Q = P + 2$ points will be used in each element. Moody uses Gauss-Legendre-Lobatto quadrature which is exact with $P + 2$ points for the Legendre polynomial of order P. However, in cases of nonlinear external forces (such as stiff ground interaction) a higher value might be needed to sample the force properly.

2.14 STATIC SOLVER

The static solver (`staticSolver`) option enables a number of Euler fwd time steps before the time loop begins. It is still a bit experimental and will probably develop during coming updates. For now, options are:

`relax` (0)
: Switch to 1 to enable the dynamic relaxation of the initial condition.

`relaxFactor` (0.9) Factor to scale the resulting cable velocity after each time step.

`maxIter` (100) How many iterations (of `timeSteps` time steps each) to do. The dynamics of the system is reset after each iteration.

`timeSteps` (100) How many time steps to take during 1 iteration.

2.15 Rigid bodies

Mooring elements such as submerged floaters and sinkers can be modelled using rigid body type objects, with keyword `rbX`. These are divided into two categories. One is the simple point mass with an assigned mass and volume but with neglected rotational motion. This is modelled using the `rigidPoint` type object. The other rigid body type also takes the rotation of the body into account and computes the motion from the force and moment exerted by the attached cables and the surrounding fluid.
The motion of submerged rigid bodies is assumed to be following Morrison's equation, and no check is made in the software that this is a valid approximation. Therefore one should take care when simulating large bodies in relation to the flow. The force and position fluxes between rigid bodies and the attached mooring cable(s) are such that the position of the body acts as a dirichlet boundary condition on the cable, while the cable acts as a Neumann boundary condition to the rigid body solver. Some fields are common to all rigid bodies while other are object type specific.

2.15.1 *Common fields*

`vertexNumber`
: The vertex number of the body. It should be the center of mass of the body.

`mass` [kg]
: The mass of the body. May not be 0.

2.15.2 *rigidPoint*

The rigid point is the object type used to simulate point masses and point floatation forces in Moody. It computes the motion of the body in the three translational degrees of freedom.

`type`
: The `type` field must be set to `'rigidPoint'`.

`volume`
: The volume of the body. The input is in [m^3] and the default value is 0.

`CDn and CM`
: The hydrodynamic coefficients of drag and added mass, `CDn` and `CM` respectively, are both specified as single input values and used according to the Morrison equation. The default values are 0.

`initialState`
: A substructure with subfield like `r`, and `v`. The body can be given a start

velocity and a startposition. . The input is a 3 by 1 vector in [m/s] of the global coordinate system. The default value is [0;0;0].

`area` [m^2]
: Area of body, used in drag force computation. If not specified, it is computed assuming the volume of a sphere.

```
rb1 = struct();
rb1.type = 'rigidPoint';
rb1.vertex = 2;
rb1.mass = 1; % [kg]
% Optional %
rb1.volume = 0;
rb1.CDn = 1;
rb1.CM = 2.5;
rb1.initialState.v = [1 0 0];
```

2.15.3 rigidCylinder

A submerged, vertical cylinder is modelled, including both translation and rotation. The rotation is modelled using quaternions and the variation of the surrounding fluid is taken into account by integration of the forces on each slice.

`type`
: The `type` field must be set to `'rigidCylinder'`.

`height` [m]
: The height of the cylinder must be specified. The input is in [m].

`diameter` [m]
: The diameter of the cylinder.

`momentOfInertia, I`
: This is the mass moment of inertia of the body, around the center of gravity. If not specified, the default value is that of the solid cylinder. The input should be in [kg m^2] and formatted as a vector of two values: `rbX.I = [Ixx Izz]`, where z is along the symmetry axis.

`centreOfBuoyancy, cob` (0) [m]
: Use cob to offset the center of buoyancy (along the symmetry axis only).

`slaves`
: There can be several cables attached to each cylinder and they need not be connected to the body main vertex. The `slaves` field specifies a list of vertices that are attached to the rigid body and whose motion is constrained

by the body translation and rotation. The input is a vector of vertex numbers matching vertices in `vertexLocations`. These need not be connected to any other object and can be used to visualise the time evolution of the position of a point of interest on the body as a hanging node.

`CM` ([0 0])
: A vector of two values `rbX.CM=[CMn CMt]`, where `CMn` is the normal direction coefficient. `CMt` is the cylinder lid coefficient.

`CD` ([0 0 0])
: A vector of three values `rbX.CD=[CDn CDt CDyaw]`, where `CDn` is the normal direction coefficient. `CDt` is the cylinder lid coefficient, and `CDyaw` is the The input is a single, unitless value with default value 0.

`initialState` (zeros(13,1))
: All instances of initial state can be set here. Subfields are: *r*-positions, *v*-velocity, *omega*-angular velocity and *q*-quaternion object.

`decoupledDrag` (0)
: Set to 1 to use the drag force in decoupled mode, i.e. to compute rotational drag moment based on ω of body alone, and compute the translational force based on the relative velocity of body and fluid. Default is 0, i.e. the strip theory is used to compute the drag forces and moments at each time-step using 7-point numerical quadrature along the symmetry axis.

3
Running the code

3.1 User input

Moody is operated though the terminal window. Generally, the `-<flagName> <value>` syntax is used to provide run-time information to the computation. All settings apart from the output file name prepends information in the input file.

3.1.1 Flags

`-f` (moodyInput.m)
: Filename. Must be followed by the filename of the input file.

`-o` (input filename without extension)
: Output filename. Must be followed by the name of the output folder.

`-time.start`
: Specifies the start time of the simulation. Overrides information specified in input file.

`-time.end`
: Specifies the end time of the simulation. Overrides information specified in input file.

`-startState`
: This is the reboot flag of moody. If followed by a value, the value should be an existing results folder name. If no value is specified, the default is to use the output folder if it exists. Moody will start from the mooring state at the start time of the simulation. If no results are found, Moody will start from static equilibrium, and print a warning message.

`-addInput`
A way to specify additional changes to the input file as value pairs: `-addInput <name1> <value1> <name2> <value2>`. All information prepends the input file and thereby overrides the information in the input file.

3.1.2 Examples

```
moody.x -f caseFile.m
moody.x -f moodyInput.m -o outFolder -time.start 10
-time.end 20 -addInput time.dt 0.01
```

4

Post processing

4.1 Moody post

Moody has a post-processing utility named `moodyPost.x`. Although nodal values are printed to the output directory, `moodyPost.x` can be used to create a smaller set of output data for post-processing. It can also be used to generate VTK-files of the cable lines and their tension force magnitude for visualisation in e.g. Kitware's Paraview.

4.1.1 Usage

moodyPost.x is a command line tool. The first parameter must be a moody result directory. Default is to process all times in output/time.dat. To control which times to process and output there are three flag options.

`-times`
  A list of output times.
  Ex. `moodyPost.x outName -vtk -times '0,1,2,4.5'`

`-timeList`
  Use times from a file. File should be in a single row vector format with no header.
  Ex. `moodyPost.x outName -vtk -timeList timeFile.txt`

`-dt` [s]
  Use time step size `dt` to step through the output history.
  Ex. `moodyPost.x outName -vtk -dt 0.5`

Output options are:

`-vtk`
: Print VTK files in sub-directory `VTK`.
Ex. `moodyPost.x moodyResults -vtk -times '0,1,2,4.5'`

`-p`
: Print nodal values in sub-directory `processed`. Optional flag `-var` is used to control which parameters to output. Default is to use all.

  `-var` Should be followed by a list of integer values in 1 to 7, where

  1. tension
  2. strain
  3. strain rate
  4. position
  5. velocity
  6. tangent
  7. end point values.

  Ex. `moodyPost.x moodyResults -p -var '[1,4]'`, prints tension and position.

`-clean`
: The clean flag (or `-c`) is used to repair incomplete output. So, for cases that have crashed or aborted, `moodyPost.x` makes the output causal in time and prints the `sPlot.dat` file required by the post-processing routines in Matlab.

4.2 MATLAB ROUTINES

A set of Matlab routines and functions are provided in `$moodyDir/API/matlab`. These can be used to load and visualise the results. Each cable result is analysed separately. The most important routine is `readCase.m` which returns a cell array of data structures containing the data of all cable objects and rigid bodies. See the help texts in each function for usage instructions.

4.3 OUTPUT FILE STRUCTURE

The first column of each output file contains the time stamp of the output, $t = [t_1, t_2, \ldots, t_m]$. This column is also printed to the `time.dat` file in the results directory. Thus all output data files have m number of rows. The values are then presented along the unstretched cable coordinate s for each n output quadrature points $[s_1, s_2, \ldots, s_n]$ of the cable. Output is separated into vector valued output and scalar output. For scalars such as tension, strain and strain rate, the output goes from the start to the end of the cable. For the vector valued output, such

Table 4.1: Output structure description for cable object data. Times are indexed as $t_1, t_2, \ldots, t_m$, vector components as $\overline{f} = [f_x, f_y, f_z]$ and the cable unstretched coordinate s is indexed as $s_1, s_2, \ldots, s_n$.

File suffix	Output structure
`_position.dat`	$\begin{array}{cccc} t_1 & x(t_1, s_1 : s_n) & y(t_1, s_1 : s_n) & z(t_1, s_1 : s_n) \\ \vdots & \vdots & \vdots & \vdots \\ t_m & x(t_m, s_1 : s_n) & y(t_m, s_1 : s_n) & z(t_m, s_1 : s_n) \end{array}$
`_velocity.dat`	$\begin{array}{cccc} t_1 & v_x(t_1, s_1 : s_n) & v_y(t_1, s_1 : s_n) & v_z(t_1, s_1 : s_n) \\ \vdots & \vdots & \vdots & \vdots \\ t_m & v_x(t_m, s_1 : s_n) & v_y(t_m, s_1 : s_n) & v_z(t_m, s_1 : s_n) \end{array}$
`_tension.dat`	$\begin{array}{cc} t_1 & T(t_1, s_1 : s_n) \\ \vdots & \vdots \\ t_m & T(t_m, s_1 : s_n) \end{array}$
`_strain.dat`	$\begin{array}{cc} t_1 & \varepsilon(t_1, s_1 : s_n) \\ \vdots & \vdots \\ t_m & \varepsilon(t_m, s_1 : s_n) \end{array}$
`_sPlot.dat`	$t_1 \quad s_1 : s_n$

as position and velocity, the data is stored consecutively for each coordinate direction of the simulation, in the order x, y, z. The structure is described in table 4.1.

5

Use Moody as a mooring module

5.1 Introduction to the API

Moody is primarily designed to be used as a modular add-on for an external solver for the fluid problem. When Moody is started up in API-mode the external solver is guiding the time evolution of some of Moodys boundary conditions. The interaction between the codes follows the schematics of Figure 5.1.
When the time step size of the external solver is larger than the internal time step of Moody, a sub-step is used internally in Moody. Moody interpolates in time between the old and the new boundary values received. Quadratic interpolation with optional time staggering is used. See [9] for more information on how different choices of interpolation can affect the solution.

5.2 Interface functions

The program interface is made up of global functions:

(i) `moodyInit` - an initialisation routine;

(ii) `moodySolve` - to compute the mooring forces; and

(iii) `moodyClose` - to close the moody objects and clean the output files.

(iv) `moodyGetNumberOfPoints` - return the number of quad-points in the moorings.

(v) `moodyGetPositions` - return the coordinates of all quad-points in the moorings.

(vi) `moodySetFlow` - sets the flow velocity, acceleration and density at each of the quadpoints of Moody.

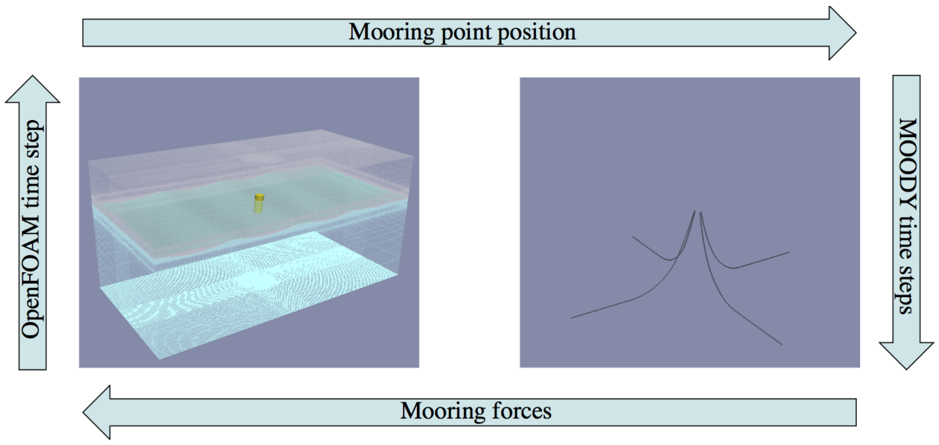

Figure 5.1: Schematic description of the information loop between Moody and an external software, represented by an OpenFOAM CFD simulation in this case.

5.2.1 *Description*

The interface is contained in `$moodyDir/include/moodyWrapper.h`, together with a description of all interface parameters required. The following is an excerpt from `moodyWrapper.h`, showing the definition of the interface functions.

```
 /** Initialisation call to moody. Required to setup the API.
   Parameter fName:         input file name (including path and extension)
   Parameter nVals:         number of values in parameter initialValues
   Parameter initialValues: array of initial values of the boundary conditions (
     global frame)
   Parameter startTime:     time at start of simulation (s)
*/
void moodyInit(const char* fName, int nVals,double initialValues[], double
     startTime );

/** Solves the internal system of mooring dynamics between time t1 and t2.
    Parameter X:     Boundary condition values at time t2.
    Parameter F:     Returned as the outward pointing mooring forces for each
        boundary condition dof.
    Parameter t1:    Time of last call, start time of time integration.
    Parameter t2:    End time of present time step simulation.

    Note: t1 and t2 are stored internally, with corresponding states of the
        mooring dynamics. If t1 has increased since the last call, moody will
        move forward in time and store the state at t1 as the new starting state
        of the mooring dynamics. Several iterations can be made with varying t2
        but the same t1 without loosing the backup starting state. Compatible
        with predictor/corrector type time stepping schemes.
*/
void moodySolve(const double X[], double F[], double t1, double t2);

/** Closes the moody simulation and stores data for post processing */
void moodyClose();

/** Interface function to collect mooring points from moody. Returns the number
    of points. XYZ is 3*nPoints. */
int moodyGetNumberOfPoints();

/** Collect mooring quadrature points in xyz list. xyz needs to be a container of
```

```
    at least nPointsIn*3 doubles. */
void moodyGetPositions(int nPointsIn, double xyz[] );

/** Set flow velocity (vF) acceleration (aF) and density (rhoF) at time t in all
    nPoints of the mooring system. nPoints should be output from
    moodyGetNumberOfPoints() and containers vF, and aF must be at least nPoints*3
     in size. rhoF must be at least nPoints. */
void moodySetFlow(const double t, int nPoints, const double vF[] ,const double aF
    [],const double rhoF[]);
```

In principle all input files can be used to run an API simulation, but some compulsory items are needed for the coupling to run smoothly.

5.3 Input file entries

Two modes of boundary condition are supported for control via the API: `externalPoint` and `externalRigidBody`. See section 2.9 for information on their individual setup. Other input file entries for the API are grouped under the `API` keyword. The following fields are available to control the simulation behaviour, where some are simply input-file versions of the command-line flags described in chapter 3:

`bcNames`

This is the only compulsory field of the API. It lists the boundary conditions that are to be externally controlled. The order of the input tells Moody how to interpret the input in the `initialValues` parameter of `moodyInit` and in the `X` parameter in `moodySolve`. By extension it also controls the order of the forces returned in parameter `F` in `moodySolve`.
Ex: `API.bcNames = {'bc4'; 'bc5'; 'bc1'};`

`staggerTimeFraction` (0.0)

Used to control the API time staggering for a smoother mooring behaviour. Value α should be a scalar $\alpha \in [0,1]$, specifying a fraction of the latest time step at which to stop the mooring simulation and return the mooring force. Default is $\alpha = 0.0$, but recommended value is $\alpha = 0.5$. End time of each coupling step is then computed from $t_{\text{end}} = (1-\alpha)t_2 + \alpha t_1$.
Ex: `API.staggerTimeFraction = 0.5`

`output` (input file name)

The name of the Moody output directory of the simulation. Equivalent to the `-o` flag.
Ex: `API.output = 'moodyResults'`

`reboot` (yes)

This flag controls the startup behaviour of Moody in API mode. Default value is `'yes'` and tells Moody to attempt a reboot from a dynamic state

at `time.start` in the output directory. New output is then appended to the previous results. To disable use of old results and start from static initial conditions, use `API.reboot='no'`. If API.reboot takes any other string value, this should be a result directory that can be used as a source state for the simulation. The source directory is then unchanged by the simulation.
Ex: `API.reboot = 'no'`

`syncOutput` (0)
Integer value 0 or 1 allowed. If 1, Moody results are printed at each coupling time (in addition to at each `saveInterval`). If 0, output is only printed based on Moodys' internal `print.dt`.
Ex: `API.syncOutput = 1`

5.4 Dynamic mooring restraint in OpenFOAM

Again, Moody was built for numerical simulation of the moored motion of wave energy converters (WECs) and other offshore structures. The mooring dynamic restraint on a floating body is an important part of the survivability and reliability of the offshore installation. The original coupling of Moody to an external CFD software was made with OpenFOAM [7] in 2013 [11] using a prototype Matlab version of Moody together with OpenFOAM 2.1.x. This coupling has been updated to compile with the v1712+, v1806 and v1906 versions of OpenFOAM. The source code is available in the `thirdParty` folder. The details of required changes to the native code of OpenFOAM are presented in implementationNotes.

In addition to the source code of the mooring restraint and the changes to the motion library, there is in `thirdParty/OpenFOAM` a tutorial (mooredFloatingObject) on how to use Moody. It simulates mooring of the classical floating object tutorial of OpenFOAM without the need of any motion constraints.

5.5 Fortran and MATLAB interface functions

There is a Fortran module with interface functions to Moody, to facilitate coupling to Fortran codes. There is also a Matlab API which allows for easy prototyping of coupled simulations from MATLAB. Both interfaces are located in the `thirdParty`.

5.6 Test the API

Finally, there is also a `test_API.x` program for testing a Moody api setup. The following example simulates the `mooringSystem.m` run via the time series specified in the `positions.txt` file. The time series should be in simple text-file format with no header line. The first column specifies the time, and the remaining columns specify the values of each api input dof respectively. See the matlab api tutorial for an example of `positions.txt`.

```
test_API.x mooringSystem.m positions.txt
```

REFERENCES

[1] G. Barter and D. Darmofal. Shock capturing with PDE-based artificial viscosity for DGFEM: Part i. Formulation. *J. Comp. Phys*, 229:1810–1827, 2010.

[2] P. Bernard. *Discontinuous Galerkin methods for Geophysical Flow Modeling*. PhD thesis, Ecole polytechnique de Louvain, 2008.

[3] B. Cockburn and C.W. Shu. Runge-Kutta discontinuous Galerkin methods for convection-dominated problems. *J. Sci. Comp.*, 16:173–261, 2001.

[4] L. Krivodonova, J. Xin, J.-F. Remacle, N. Chevaugeon, and J. Flaherty. Shock detection and limiting with discontinuous Galerkin methods for hyperbolic conservation laws. *J. Appl. Num. Math.*, 48:323–338, 2004.

[5] J. Lindahl. Implicit numerisk lösning av rörelseekvationerna för en förankringskabel. Technical Report Report Series A:11, Chalmers University of Technology, 1984.

[6] J.R. Morison, M.P. O'Brien, J.W. Johnson, and S.A. Schaaf. The force exerted by surface waves on piles. *Petroleum Transactions, Amer. Inst. Mining Engineers*, 186:149–154, 1950.

[7] OpenCFD Ltd. *OpenFOAM Homepage*, 2014. Available http://www.openfoam.org.

[8] Orcina Inc. *OrcaFlex manual – version 9.5a*, 2012.

[9] J. Palm. *Mooring Dynamics for Wave Energy Applications*. PhD thesis, Chalmers University of Technology, 2017.

[10] J. Palm, C. Eskilsson, and L. Bergdahl. An hp-adaptive discontinuous Galerkin method for modelling snap loads in mooring cables. *Ocean Engineering*, 144:266–276, 2017.

[11] J. Palm, C. Eskilsson, G. Paredes, and L. Bergdahl. CFD simulations of a moored floating wave energy converter. In *Proc. 10th European Wave and Tidal Energy Conference*, Aalborg, Denmark, 2013.

A
Theory manual

A.1 BACKGROUND

The present version of Moody is implemented as an *hp*-adaptive cable solver based on the discontinuous Galerkin method with an approximate Riemann solver of local Lax-Friedrich type. This manual outlines the governing equations and highlights the modelling approach used in the implementation. This manual is mostly compiled from Palm et al [10] where the present formulation was first presented. In addition, the section on API boundary condition interpolation is collected from the PhD thesis of Palm [9].

A.2 GOVERNING EQUATIONS

For a cable of length L_c, we use the unstretched cable coordinate $s \in [0, L_c]$ to express the global coordinate position vector of the cable as $\mathbf{r} = [r_1(s), r_2(s), r_3(s)]^{\mathrm{T}}$. Under the assumption of negligible bending stiffness, the equation of motion becomes

$$\gamma_0 \ddot{\mathbf{r}} = \frac{\partial}{\partial s}\left(T\hat{\mathbf{t}}\right) + \mathbf{f}, \tag{A.1}$$

$$\hat{\mathbf{t}} = \frac{\partial \mathbf{r}}{\partial s}\left|\frac{\partial \mathbf{r}}{\partial s}\right|^{-1}, \tag{A.2}$$

where γ_0 is the cable mass per unit length, T is the cable tension force magnitude, $\hat{\mathbf{t}}$ is the tangential unit vector of the cable and $\mathbf{f}$ represents all external forces. For notation we use $\dot{x} = \dfrac{\partial x}{\partial t}$ to indicate time derivatives and $|\mathbf{x}| = \sqrt{x_i x_i}$ to denote the L_2 - norm of a vector quantity x, Vector components are denoted by their index as $x_i,\ i \in [1,2,3]$, and summation over repeated indices is implied.

Written as a first order system in terms of the cable position $\mathbf{r}$, its spatial derivative $\mathbf{q} = \frac{\partial \mathbf{r}}{\partial s}$ and its momentum density $\nu = \dot{\mathbf{r}}\gamma_0$, eq. (A.1) becomes

$$\dot{\mathbf{r}} = \frac{\nu}{\gamma_0}, \tag{A.3}$$

$$\dot{\mathbf{q}} = \frac{\partial}{\partial s}\left(\frac{\nu}{\gamma_0}\right), \tag{A.4}$$

$$\dot{\nu} = \frac{\partial}{\partial s}\left(T\hat{\mathbf{t}}\right) + \mathbf{f}, \tag{A.5}$$

where we have assumed that the cable mass is constant in time. In terms of a state vector $\mathbf{u} = [\mathbf{r}, \mathbf{q}, \nu]^{\mathrm{T}}$ the conservative form of the problem is written as

$$\dot{\mathbf{u}} = \frac{\partial \mathbf{F}(\mathbf{u})}{\partial s} + \mathbf{Q}(\mathbf{u}), \tag{A.6}$$

with a flux function

$$\mathbf{F}(\mathbf{u}) = \left[\emptyset, \frac{\nu}{\gamma_0}, T\hat{\mathbf{t}}\right]^{\mathrm{T}}, \tag{A.7}$$

and a non-linear source term

$$\mathbf{Q}(\mathbf{u}) = \left[\frac{\nu}{\gamma_0}, \emptyset, \mathbf{f}\right]^{\mathrm{T}}. \tag{A.8}$$

A.2.1 Measures of strain

Constitutive modelling in Moody is based on a strain-tension relation. We use the elongation of the cable as a measure for the strain throughout the code. This is to be most easily compatible with nonlinear strain-tension curves that originate from physical experiments on cables. Therefore, it is assumed that input parameters to material models are in accordance with this criteria. The elongation strain ε is computed from

$$\varepsilon = \sqrt{\mathbf{q}\cdot\mathbf{q}} - 1, \tag{A.9}$$

so that

$$|\mathbf{q}| = 1 + \varepsilon, \tag{A.10}$$

$$\hat{\mathbf{t}} = \frac{\mathbf{q}}{1+\varepsilon} \tag{A.11}$$

A.2.2 Strain rate

The strain rate is not formally included in the formulation. It is computed from

$$\dot{\varepsilon}^{(k)} = \dot{\mathbf{q}}^{(k-1)} \cdot \hat{\mathbf{t}}^{(k)}, \tag{A.12}$$

where superscript (k) indicates at which time step the variable is evaluated. $\dot{\mathbf{q}}^{(k-1)}$ thus means that the value from the last time step is used to approximate the strain rate. For highly damped cables, a lower CFL condition might be needed to ensure stability due to this approximation.

A.2.3 Tension force

The axial tension force is evaluated as $T = T(\varepsilon, \dot{\varepsilon})$, for each material model implemented. As an example, the `bilLinear` cable model has a truly linear relation between T and ε, with the stiffness EA. There is no deviation from the linear curve due to cross-sectional area diminishing for large strains, as that is assumed to be inherent in the input relation between elongation and tension. Cross-sectional area change due to strain is taken into account in the evaluation of the viscous drag force on the cable, see section A.3.4.

A.3 External forces

The total external force $\mathbf{f}$ of eq. (A.1) is given by

$$\mathbf{f} = \mathbf{f}_\mathrm{a} + \mathbf{f}_\mathrm{b} + \mathbf{f}_\mathrm{c} + \mathbf{f}_\mathrm{d}\,, \tag{A.13}$$

where $\mathbf{f}_\mathrm{a}$ is the added mass and Froude-Krylov forces, $\mathbf{f}_\mathrm{b}$ is the net force of gravity and buoyancy, $\mathbf{f}_\mathrm{c}$ represent contact forces, typically from sea-floor interaction, and $\mathbf{f}_\mathrm{d}$ is the viscous drag force.

For notation, a vector quantity $\mathbf{x}$ can be expressed in a tangential and normal component with respect to the cable tangent respectively as

$$\mathbf{x}_{\hat{t}} = \mathbf{x} \cdot \hat{\mathbf{t}}\,, \tag{A.14}$$

$$\mathbf{x}_{\hat{n}} = \mathbf{x} - \mathbf{x}_{\hat{t}}\,. \tag{A.15}$$

A.3.1 Added mass

The added mass effect on the cable dynamics is divided into a fluid part $\mathbf{f}_{\mathrm{a}+}$ dependent on the fluid acceleration, and an inertial part $\mathbf{f}_{\mathrm{a}-}$ which scales with the cable acceleration,

$$\mathbf{f}_\mathrm{a} = \mathbf{f}_{\mathrm{a}+} - \mathbf{f}_{\mathrm{a}-}\,. \tag{A.16}$$

The added mass contributions are computed from Morison's equations [6] using coefficients C_Mn and C_Mt in the normal and tangential direction respectively.

$$\mathbf{f}_{\mathrm{a}+} = A_c \rho_f \left((1 + C_\mathrm{Mn})\, \dot{\mathbf{v}}_f + (C_\mathrm{Mt} - C_\mathrm{Mn})\, \dot{\mathbf{v}}_{f\hat{\mathbf{t}}} \right)\,, \tag{A.17}$$

$$\mathbf{f}_{\mathrm{a}-} = \frac{A_c \rho_f}{\gamma_0} \left(C_\mathrm{Mn} \dot{v} + (C_\mathrm{Mt} - C_\mathrm{Mn})\, \dot{v}_{\hat{t}} \right)\,, \tag{A.18}$$

with ρ_f as the fluid density and $\mathbf{v}_f$ as the fluid velocity.

The inertial contribution is moved to the left hand side of eq. (A.5), effectively changing it to

$$\mathbf{F}_{\text{tot}} = \frac{\partial}{\partial s}\left(T\hat{\mathbf{t}}\right) + \mathbf{f} + \mathbf{f}_{\text{a}-} \,, \tag{A.19}$$

$$\mathbf{F}_{\text{tot}} = \dot{v} + \mathbf{f}_{\text{a}-} \,. \tag{A.20}$$

The computation of $\dot{v}$ is then reduced to solving

$$\mathbf{F}_{\text{tot}} = \left[a\mathbf{I} + b\hat{\mathbf{t}} \otimes \hat{\mathbf{t}}\right] \dot{v} = \psi\dot{v} \,, \tag{A.21}$$

$$a = 1 + A_c\rho_f \frac{C_{\text{Mn}}}{\gamma_0} \,, \tag{A.22}$$

$$b = A_c\rho_f \frac{C_{\text{Mt}} - C_{\text{Mn}}}{\gamma_0} \,, \tag{A.23}$$

where $\mathbf{I}$ is the 3×3 identity matrix, ψ is the added inertia matrix and $\otimes$ denotes the open vector product. In effect, this is then handled as if ψ^{-1} was a relaxation step on $\dot{v}$ for each computational node:

$$\dot{v} = \psi^{-1}\mathbf{F}_{\text{tot}} \,, \tag{A.24}$$

and the inverse matrices ψ^{-1} are analytically obtained for each node as

$$\psi^{-1} = \frac{1}{a(a+b)}\left((a+b)\mathbf{I} - b\hat{\mathbf{t}} \otimes \hat{\mathbf{t}}\right) \,. \tag{A.25}$$

A.3.2 Buoyancy

The buoyancy force is straightforward and computed from gravitational acceleration $\mathbf{g}$, the cable density ρ_c, the fluid density ρ_f, and the cable mass γ_0 as

$$\mathbf{f}_{\text{b}} = \mathbf{g}\frac{\rho_c - \rho_f}{\rho_c}\gamma_0 \,. \tag{A.26}$$

$$\tag{A.27}$$

A.3.3 Contact

The implementation of the ground model has a potentially large influence on the quality of the simulation. Moody supports contact forces from the ground based on a bilinear spring and damper. The implementation is close to that used in [8]. A tangential friction model from [5] using dynamic friction is also implemented. For a horizontal sea floor with (x, y, z) coordinates corresponding to vector index

1, 2 and 3 respectively, the contact force is computed as

$$\mathbf{f}_\mathrm{c} = \begin{cases} \mathbf{G}_\mathrm{v} + \mathbf{G}_\mathrm{h} & \text{if } (z_\mathrm{G} - r_{\hat{z}}) \geq 0\,, \\ 0 & \text{otherwise}\,, \end{cases} \tag{A.28}$$

$$\mathbf{G}_\mathrm{v} = \left(K_\mathrm{G} d_\mathrm{c} \left(z_\mathrm{G} - r_{\hat{z}}\right) - 2\xi_\mathrm{G}\sqrt{K_\mathrm{G}\gamma_0 d_\mathrm{c}} \min\left(\dot{r}_{\hat{z}}, 0\right)\right)\widehat{\mathbf{z}}\,, \tag{A.29}$$

$$\mathbf{G}_\mathrm{h} = \mu f_{\mathrm{b}\hat{z}} \tanh\left(\frac{\pi \dot{r}_{\widehat{xy}}}{v_\mu}\right)\frac{\dot{\mathbf{r}} - \dot{r}_{\hat{z}}\widehat{\mathbf{z}}}{\left|\dot{r}_{\widehat{xy}}\right|} \tag{A.30}$$

where indices $\widehat{xy}$, and $\hat{z}$ denote the horizontal and vertical projections of a vector respectively. Further, $r_{\hat{z}}$ is the vertical coordinate of the cable position, z_G is the vertical position of the ground, K_G is the ground stiffness, ξ_G is the ratio of critical damping for the ground–cable pair. The coefficient of kinetic friction is in eq (A.30) denoted as μ, with a corresponding velocity of maximum friction v_μ.

Please note that to model the ground interaction as described in eq. (A.28), we need the position of the cable r as an independent variable, which is why it is included as an independent variable in eq. (A.3).

A.3.4 Drag

The viscous drag contribution is modelled via Morison's formulas [6] for a circular cross-section with volume-preserving properties during axial strain. Hence, the elongation factor $(1+\varepsilon)$ and the contraction factor $(1+\varepsilon)$ do not fully cancel. This is why the contraction factor for the diameter is $(\sqrt{1+\varepsilon})^{-1}$, and $\mathbf{f}_\mathrm{d}$ is expressed as

$$\mathbf{f}_D = 0.5\rho_f d_c \sqrt{1+\varepsilon}\left(C_\mathrm{Dt}\left|\mathbf{v}^*_{\hat{t}}\right|\mathbf{v}^*_{\hat{t}} + C_\mathrm{Dn}|\mathbf{v}^*_{\hat{n}}|\mathbf{v}^*_{\hat{n}}\right)\,, \tag{A.31}$$

where C_Dn, C_Dt are the drag coefficients of the normal and tangential direction, and $\mathbf{v}^*$ is the relative velocity between fluid and the cable:

$$\mathbf{v}^* = \mathbf{v}_f - \dot{\mathbf{r}}\,. \tag{A.32}$$

A.4 The Discontinuous Galerkin method

The discontinuous Galerkin method is used for spatial discretisation of the cable domain. Consider eq. (A.6) for a cable domain Ω of unstretched coordinate $s \in [0, L_c]$. Ω is discretised in a set of N elemental regions Ω^e of mesh size h^e. In each element a function $y(s,t)$ is approximated as a Legendre polynomial of order p as

$$y(s,t) \approx y^e(s,t) = \sum_{k=0}^{k=p} \phi_k(s)\tilde{y}^e_k(t)\,, \tag{A.33}$$

where $\tilde{y}_k^e$ is the k^{th} expansion coefficient corresponding to the basis function of order k, $\phi_k(s)$. The discontinuous Galerkin (DG) formulation of eq. (A.6) is then

$$\left(\phi_k, \frac{\partial u^e}{\partial t}\right)_{\Omega^e} = \left(\phi_k, \frac{\partial F^e}{\partial s}\right)_{\Omega^e} + (\phi_k, Q^e)_{\Omega^e}\,, \qquad \forall k \in [0,p]\,, \tag{A.34}$$

using

$$(a(s,t), b(s,t))_{\Omega^e} = \int_{\Omega^e} a(s,t)b(s,t)\mathrm{d}s\,, \tag{A.35}$$

to denote the inner product operator on the elemental domain. The Gauss-Lobatto-Legendre quadrature points and quadrature rules are used for numerical evaluation of the elemental integrals in (A.35)

Two integration by parts of the weak derivative term $\left(\phi_k, \frac{\partial F^e}{\partial s}\right)_{\Omega^e}$, allows eq. (A.34) to be written

$$\left(\phi_k, \frac{\partial u^e}{\partial t}\right)_{\Omega^e} = \left(\phi_k, \frac{\partial F^e}{\partial s}\right)_{\Omega^e} + \left[\phi_k \left(\widehat{F^e} - F^e\right)\right]_{s_L^e}^{s_U^e} + (\phi_k, Q^e)_{\Omega^e}\,, \tag{A.36}$$

$\forall k \in [0,p]$. Here s_U^e and s_L^e denote the elemental upper and lower bounds of the unstretched cable domain coordinate s respectively. Please note the introduction of the numerical boundary flux $\widehat{F^e}$. As the elements are discontinuous at the element boundaries, the numerical flux provides the coupling between neighbouring elements.

Finally, separation of time and space dependence defined in eq. (A.33) allows us to rewrite eq. (A.36) in terms of the modal coefficients of our polynomial space:

$$(\phi_k, \phi_i)\frac{\partial \tilde{u}^e}{\partial t} = \left(\phi_k, \frac{\partial \phi_i}{\partial s}\right)\tilde{F}^e + \left[\phi_k \left(\widehat{F^e} - F^e\right)\right]_{s_L^e}^{s_U^e} + (\phi_k, Q^e)_{\Omega^e} \tag{A.37}$$

$\forall k, i \in [0,p]$.

A.5 Time step schemes

We use the strong stability preserving 3rd order Runge-Kutta (RK) scheme as described in [3] to advance eq. (A.6) in time. Let $\dot{\mathbf{u}}_k^{(i)} = \mathscr{L}\left(\mathbf{u}_k^{(i)}\right)$, where $\mathbf{u}_k^{(i)}$ is the state vector at time step k and sub-cycle i. The 3rd order RK scheme is then

$$
\begin{aligned}
\mathbf{u}_{k+1}^{(0)} &= \mathbf{u}_k\,, \\
\mathbf{u}_{k+1}^{(1)} &= \mathbf{u}_{k+1}^{(0)} + \Delta\mathscr{L}\left(\mathbf{u}_{k+1}^{(0)}\right)\,, \\
\mathbf{u}_{k+1}^{(2)} &= \frac{1}{4}\left(\mathbf{u}_{k+1}^{(1)} + \Delta\mathscr{L}\left(\mathbf{u}_{k+1}^{(1)}\right)\right) + \frac{3}{4}\mathbf{u}_{k+1}^{(0)}\,, \\
\mathbf{u}_{k+1}^{(3)} &= \frac{2}{3}\left(\mathbf{u}_{k+1}^{(2)} + \Delta\mathscr{L}\left(\mathbf{u}_{k+1}^{(2)}\right)\right) + \frac{1}{3}\mathbf{u}_{k+1}^{(0)}\,, \\
\mathbf{u}_{k+1} &= \mathbf{u}_{k+1}^{(3)}\,.
\end{aligned}
\tag{A.38}
$$

A.6 Riemann solver

The numerical fluxes are central to the performance of the numerical scheme used in Moody. The present version uses a local Lax-Friedrich type flux, with $\widehat{F^e}$ in (A.37) evaluated as

$$\widehat{F^e} = \{F^e\} + |\lambda|_{\max}\,[[u^e]] \quad \text{on } \Gamma \in \Omega\,. \tag{A.39}$$

Notations $\{x^e\}$ and $[[x^e]]$ are

$$\{x^e\} = 0.5\left(x^+ + x^-\right)\,, \tag{A.40}$$

$$[[x^e]] = 0.5\left(n^+x^- + n^-x^+\right)\,, \tag{A.41}$$

where x^+ means taking the value from the internal side of the boundary and x^- from the neighbouring element. The normal vector n refers to the outward pointing normal, defined for each element boundary as $n^+ = -1$ on s_L^e and as $n^+ = 1$ on s_R^e. The maximum eigenvalue needed in eq. (A.39) is in this version chosen for each equation as:

$$|\lambda|_{\max} = \begin{cases} 0.8\left|\frac{[[F^e]]}{[[u^e]]}\right| & \text{if } u = q \\ & \\ |c_t| & \text{otherwise} \end{cases}\,, \tag{A.42}$$

where c_t is the tangential wave speed in the cable. It is defined by the maximum eigenvalue of the hyperbolic problem,

$$c_t = \sqrt{\frac{\partial T}{\partial \varepsilon}\gamma_0}\,. \tag{A.43}$$

A.7 *hp*-adaptivity

The goal of the adaptive mesh refinement scheme is to limit the discretisation error to below a pre-set tolerance level, ε^*. To do this we need an indicator of the relative error, a smoothness indicator to let us know which type of error that we observe, and a mechanism to efficiently adapt the spatial discretisation to best fit the solution.

A.7.1 Error indicator

In this work, we use the tension force magnitude T as indicator variable for the quality of the solution. For smooth solutions, we expect a convergence rate of $\mathcal{O}\left(h^{p+1}\right)$, but close to discontinuities the solution converges as $\mathcal{O}(h)$ [4]. The elemental jump of the solution is a common measure of the numerical error [4, 2, 1]. Barter et al. [1] used the relative jump

$$\tau = \left|\frac{[[T]]}{\{T\}}\right|,$$

at element edges to indicate regions of shocks. We incorporate the relative jump into Bernard's formula for the relative error [2] to get

$$\varepsilon^e = \frac{1}{\sqrt{8}}\sqrt{(\tau_L^e)^2 + (\tau_R^e)^2}, \tag{A.44}$$

with τ_L^e and τ_R^e as the relative jumps at the left and right elemental borders respectively.

A.7.2 Shock detection

We follow Krivodonova [4] in order to decide on the nature of the error, i.e. locate regions of sharp gradients or shocks. An intermediate convergence rate $h^{0.5(p+1)}$ is used to establish a shock indicator of element e. Computing

$$I^e = \max\left(\tau_L^e, \tau_R^e\right) h^{-0.5*(p+1)}, \tag{A.45}$$

results in the shock criteria as $I^e \geq 1$ [4]. When the cable goes slack ($T \to 0$), a snap load is expected to occur in the near future. We therefore treat it as a shock criteria. Thus, we also introduce a low-tension criteria T^*, that indicate shock behaviour if $\min\left(T^e\right) \leq T^*$, with $\min\left(T^e\right)$ evaluated at the quadrature points of element e. The shock criteria and detectors are combined into the shock detecting function S^e as

$$S^e = \begin{cases} 1 & \text{if} \quad I^e \geq 1 \text{ and } \varepsilon^e \geq \varepsilon^* \\ 1 & \text{if} \quad \min\left(T^e\right) \leq T^* \\ 0 & \text{otherwise} \end{cases}. \tag{A.46}$$

Please note that the jump-based shock detector is only applied to elements that have errors higher than the tolerance level, while the low tension indicator is applied to all elements in the cable domain.

A.7.3 Adaptivity control

We let p-refinement have precedence over h-refinement in all elements of smooth solution. If shocks are detected, the mesh is forced into maximum h-refinement and the order is reduced to linear approximation $p = 1$. h-refinement is restricted to splitting elements in half and merging two equally sized elements with the same parent. The initial mesh h-resolution is not allowed to coarsen, and the splitting hierarchy of elements is therefore confined to one element of the initial mesh.

The inverse operation of coarsening the resolution follows the recipe of Bernard [2], assuming the error relation between the new and the old mesh to be:

$$\frac{\varepsilon^e}{\varepsilon^*} = \frac{h^{p+1}}{h^{*p^*+1}}\,, \tag{A.47}$$

where p^* and h^* denote the resolution parameters in the modified mesh. Thus, for merging elements with constant order p, the criterion is: $\varepsilon^e < 0.5^{p+1}\varepsilon^*$; and for lowering the polynomial order under constant h, consequently: $\varepsilon^e < h\varepsilon^*$. Combining the above, we arrive at the hp-adaptive control algorithm

$$\begin{aligned}
&\text{if}\quad S^e = 1: \quad p = 1, \quad h = h_{\min}\\
&\text{if}\quad S^e = 0: \begin{cases} p = \min(p+1, p_{\max}) & \text{if} \quad \varepsilon^e \geq \varepsilon^* \\ h = \max(0.5h, h_{\min}) & \text{if} \quad \varepsilon^e \geq \varepsilon^* \text{ and } p = p_{\max} \\ h = \min(2h, h_{\max}) & \text{if} \quad \varepsilon^e < 0.5^{p+1}\varepsilon^* \\ p = \max(p-1, 1) & \text{if} \quad \varepsilon^e < h\varepsilon^* \text{ and } h = h_{\max} \end{cases},
\end{aligned}$$

where S^e is the shock indicator function explained in eq. (A.46). The application of the control algorithm to pure h is straightforward, as we simply skip the p-adaptation parts. For pure p refinement, we skip the h-adaptation as well as the shock detection step. The adaptive scheme is applied after a complete time step, so that all stages of the Runge-Kutta scheme share the same spatial resolution.

A.7.4 Mesh refinement

The $hp-$adaptive mesh refinement in Moody is implemented in the following way. First, let h be the non-dimensional element size, P be the polynomial order of an element, and H denote the global level of $h-$refinement. For a given target change in mesh with ∂H levels and ∂P orders, the mesh is changed according to

(i) $p-$refinement,

$$P = P + \max(\partial H, 0)\,; \tag{A.48}$$

(ii) $h-$adaptivity, including splitting and merging of elements,

$$H = H + \partial H\,; \tag{A.49}$$

(iii) $p-$coarsening,

$$P = P + \min(\partial H, 0)\,, \tag{A.50}$$

to avoid loss of accuracy.

A.7.5 p-adaptivity

We use the Legendre polynomials as hierarchic modal basis functions. Increasing the polynomial order of an element is therefore reduced to simply padding the solution vector $\tilde{u}_i$ with 0 when P is increased. Conversely, $p-$coarsening is achieved by truncating $\tilde{u}_i$ at the target order.

A.7.6 h-adaptivity

In $h-$adaptivity, elements are added or removed dynamically as the solution progresses. Each element has two level flags used in to keep track of the $h-$refinement hierarchy. A global level index I_{G} and a local level index I_{L}, controlled during splitting and merging according to:

$$e \to \left(e^-, e^+\right) \begin{cases} I_{\mathrm{G}}^- = I_{\mathrm{G}}^+ = I_{\mathrm{G}} + 1 \\ I_{\mathrm{L}}^- = \min\left(I_{\mathrm{L}}, 0\right) - 1 \\ I_{\mathrm{L}}^+ = \max\left(I_{\mathrm{L}}, 0\right) + 1 \end{cases}, \tag{A.51}$$

$$\left(e^-, e^+\right) \to e \begin{cases} I_{\mathrm{G}} = I_{\mathrm{G}}^+ + 1 = I_{\mathrm{G}}^- + 1 \\ I_{\mathrm{L}} = I_{\mathrm{L}}^- + I_{\mathrm{L}}^+ \end{cases}. \tag{A.52}$$

Here we use notations $-$ and $+$ to denote the left and right child element respectively. The prerequisites for allowing merging of two neighbouring elements who both have been flagged for $h-$coarsening can now be simplified to

$$I_{\mathrm{L}}^- < 0, \quad \text{and} \quad I_{\mathrm{L}}^+ > 0.$$

The global order flag is only used to avoid refinement beyond a pre-set maximum limit and to keep track of the elemental size.

When an element is marked for h-refinement, it is divided in two equally sized elements. Thus the local Gauss-Lobatto-Legendre (GLL) quadrature points , $\xi \in [-1, 1]$, of the original parent element will always have the same relation to the points of the two new child elements. This allows for a constant, pre-computed split map matrix S_{s} between the modal values of the parent and its children. If superscript $^-$ and $^+$ describe values in the left and right child elements respectively, then

$$\begin{bmatrix} \tilde{u}^- \\ \tilde{u}^+ \end{bmatrix} = S_{\mathrm{s}}\,\tilde{u}, \tag{A.53}$$

Let S_{m2n}, and S_{n2m} denote the mapping matrices between the modal solution and the nodal values at the GLL points u of an element, i.e.

$$\begin{aligned} u &= S_{\mathrm{m2n}}\tilde{u} = \phi\left(\xi\right)\tilde{u}, \\ \tilde{u} &= S_{\mathrm{n2m}}u = M^{-1}Wu, \end{aligned}$$

where M^{-1} is the inverse mass matrix and W is the quadrature weighted $\phi^{\mathrm{T}}(\xi)$ matrix. Here we use the matrix $\phi(\xi)$ as a q by $p+1$ matrix where ϕ_{ij} contains the value of basis j at quadrature point i. Then the splitting transform S_{s} is constructed from

$$S_{\mathrm{s}} = \begin{bmatrix} S_{\mathrm{n2m}} & \emptyset \\ \emptyset & S_{\mathrm{n2m}} \end{bmatrix} \phi(\xi_{\mathrm{s}}), \tag{A.54}$$

$$\xi_{\mathrm{s}} = \begin{bmatrix} 0.5(\xi-1) \\ 0.5(\xi+1) \end{bmatrix}, \tag{A.55}$$

where ξ_{s} are the GLL points of the split elements projected on the local parent domain.

The merging operation is the inverse of the splitting, but is computed as a separate transform S_{m}. To avoid the ambiguous midpoint of the sibling elements, the merge-matrix is based on an even number of quadrature points in the final (parent) element. If Q is odd, $Q+1$ is used to compute the merge matrix.

So, assuming that Q is even, S_{m} is constructed from

$$S_{\mathrm{m}} = S_{\mathrm{n2m}}(Q) \begin{bmatrix} \phi(2\xi+1), \forall \xi < 0 \\ \phi(2\xi-1), \forall \xi > 0 \end{bmatrix}. \tag{A.56}$$

A.8 Limiters

The generalised minMod slope limiter as it is described in [3] is used to avoid overshoots in the presence of shocks. It is not strictly monotone and TVD as the original minMod method, but is classified as a TVB (Total Variation Bounded) method.

$$\tilde{u}_1^e = \mathrm{minMod}\left(\theta_l \frac{\tilde{u}_0^e - \tilde{u}_0^{e-1}}{2}, \theta_l \frac{\tilde{u}_0^{e+1} - \tilde{u}_0^e}{2}, \tilde{u}_1^e\right), \theta_l \in [1,2], \tag{A.57}$$

where $\tilde{u}_0^e$ is the mean value and $\tilde{u}_1^e$ is the linear slope of element e. The min-mod function is defined as

$$\mathrm{minMod}(a,b,c) = \begin{cases} \min(a,b,c) & \text{if } a,b,c > 0 \\ \max(a,b,c) & \text{if } a,b,c < 0 \\ 0 & \text{otherwise} \end{cases}.$$

The value of θ_l blends the limiter between the classical min-mod for $\theta_l = 1$ and the less restrictive, generalised min-mod from [3] for $\theta_l = 2$. The limiter is applied after each stage of the Runge-Kutta time-stepping scheme.

A.9 API BOUNDARY CONDITIONS

When coupled to an external solver (i.e. in API mode), the timestep requirement on the cable dynamics is potentially orders of magnitude smaller than that required by the external solver. To save computational effort, the boundary conditions that are sent to Moody at each coupling time are interpolated to achieve intermediate boundary conditions. The position of the attachment point is interpolated based on constant acceleration over the time step, and using a staggered process to maintain smoothness.

To explain, we let t_k and t_{k+1} be two consecutive coupling times, with corresponding mooring point positions r_k and r_{k+1}. We introduce the staggered-time fraction $\phi \in [0,1]$ and identify a corresponding mooring time $t^{\mathrm{m}}_{k+1} \in [t_k, t_{k+1}]$ as

$$t^{\mathrm{m}}_{k+1} = \phi t_k + (1-\phi)\, t_{k+1}\,. \tag{A.58}$$

The mooring boundary conditions $r_{\mathrm{D}}(t)$ and $v_{\mathrm{D}}(t)$ are interpolated over the mooring time step interval $t \in \left[t^{\mathrm{m}}_k, t^{\mathrm{m}}_{k+1}\right]$ as

$$r_{\mathrm{D}}\left(t^{\mathrm{m}}_k + \tau\right) = r^{\mathrm{m}}_k + v^{\mathrm{m}}_k \tau + 0.5 a_k \tau^2\,, \tag{A.59}$$

$$v_{\mathrm{D}}\left(t^{\mathrm{m}}_k + \tau\right) = v^{\mathrm{m}}_k + a_k \tau\,, \tag{A.60}$$

where $\tau \in \left[0, t^{\mathrm{m}}_{k+1} - t^{\mathrm{m}}_k\right]$, while $r^{\mathrm{m}}_k = r_{\mathrm{D}}\left(t^{\mathrm{m}}_k\right)$ and $v^{\mathrm{m}}_k = v_{\mathrm{D}}\left(t^{\mathrm{m}}_k\right)$ are taken from the previous coupling interval. To close the system, we only need to define a_k. Here, we choose a_k as the constant acceleration needed to satisfy $r_{\mathrm{D}}\left(t_{k+1}\right) = r_{k+1}$, i.e.

$$a_k = \frac{r_{k+1} - r^{\mathrm{m}}_k - v^{\mathrm{m}}_k \Delta_k}{0.5\Delta_k^2}\,, \tag{A.61}$$

with $\Delta_k = t_{k+1} - t^{\mathrm{m}}_k$.