CONTENTS

1 Program description
  1.1 Background ........................................... 1
  1.2 Workflow ............................................ 2
  1.3 Theoretical framework ............................... 3
    1.3.1 Cable dynamics .................................. 3
    1.3.2 Discretization scheme ......................... 4
    1.3.3 Boundary Conditions ......................... 5
  1.4 Note on performance .............................. 6

2 The input file
  2.1 General settings .................................. 7
  2.2 Time settings .................................... 9
  2.3 Geometry ......................................... 10
  2.4 Cable types ....................................... 11
    2.4.1 Fields ........................................ 11
  2.5 Cable objects .................................... 13
    2.5.1 Fields ........................................ 13
  2.6 Initial Conditions ............................... 15
    2.6.1 PreStrain .................................... 15
    2.6.2 CatenaryStatic .............................. 15
    2.6.3 HalfSine type ................................ 16
  2.7 Material models ................................. 17
    2.7.1 linearCable .................................. 17
    2.7.2 bilinearCable ............................... 17
    2.7.3 multiLinearCable ........................... 17
  2.8 Boundary Conditions ............................ 19
    2.8.1 Common fields ................................ 19
    2.8.2 fixed mode ................................... 20
1 Program description

1.1 Background

Moody is a software developed for computing cable dynamics. The main objective has been to create a tool for in depth analysis of the tension force propagation in mooring cables. Sudden, large motions of the moored object may result in shock waves propagating in the cable. It is the aim of this software to be able to analyse and quantify the effects that these force propagations may have on the overall design of mooring cables in marine environments.

The code is at present written in MATLAB code language, and is based on the Finite Element Method with high order polynomial basis functions and discontinuous elements. The spatial discretisation is based on the Local Discontinuous Galerkin Method.
1. Program description

1.2 Workflow

To make a simulation in Moody, the sourceCode folder must first be on the Matlab path (easiest is to use the addpath('pathToMoodySource') command). Next, an input file according to the format needed by the software should be created. The exact appearance of the input file is described in section ??.

Then Moody can be called with the desired input. After the simulation, the results must be evaluated and transformed to physical space. At runtime Moody only saves its local, non-dimensionalised modal values. A movie of the results can be made with the mkMovie function as a final step. In short:

- Add source code to Matlab path
- Create an input file
- Call moody with the input file
- Evaluate results
- Call mkMovie to view the results easily. (optional)
- Remove source code from Matlab path (optional)

```matlab
addpath('sourceCode');
moody('inputFile');
evaluateResults('inputFile');
% Optional %
mkMovie('inputFile');
rmpath('sourceCode');
```
1.3 THEORETICAL FRAMEWORK

1.3.1 Cable dynamics

For a completely flexible cable, the equation of motion for the cable, expressed along the curvilinear abscissa \( s \) of the unstretched cable, can be written as a coupled set of one-dimensional, non-linear wave equations. Following [1, 3], the formulation for a linear elastic cable material appears as

\[
\frac{\partial^2 \mathbf{r}}{\partial t^2} = \frac{1}{\gamma_0} \frac{\partial}{\partial s} \left( \frac{T}{1 + \varepsilon} \frac{\partial \mathbf{r}}{\partial s} \right) + \frac{\mathbf{f}}{\gamma_0}, \quad (1.1)
\]

\[
T = EA \varepsilon, \quad (1.2)
\]

\[
\varepsilon = \left| \frac{\partial \mathbf{r}}{\partial s} \right| - 1, \quad (1.3)
\]

where \( \mathbf{r} = (r_x, r_y, r_z) \) is the cable position vector, \( \gamma_0 \) is the mass per meter cable, \( T \) represents the axial tension force magnitude and \( \varepsilon \) is the cable tangential strain. The linear, axial stiffness of the cable is denoted by \( EA_0 \). The \( \mathbf{f} = \mathbf{f}_1 + \mathbf{f}_2 + \mathbf{f}_3 + \mathbf{f}_4 + \mathbf{f}_5 \) on the right hand side represent the environmental forces acting on the cable segment.

- \( \mathbf{f}_1 \): The sum of gravity and buoyancy
- \( \mathbf{f}_2 \): Added mass force
- \( \mathbf{f}_3 \): Drag force
- \( \mathbf{f}_4 \): Froude-Krylov force
- \( \mathbf{f}_5 \): Contact forces

We introduce the unit tangential vector \( \mathbf{t} \) as

\[
\mathbf{t} = \frac{\partial \mathbf{r}}{\partial s} \left/ \left| \frac{\partial \mathbf{r}}{\partial s} \right| \right. = \frac{\partial \mathbf{r}}{\partial s} \left/ \left( 1 + \varepsilon \right) \right., \quad (1.4)
\]

and the decomposition notations \( \mathbf{x}_t = (\mathbf{x} \cdot \mathbf{t}) \mathbf{t} \) and \( \mathbf{x}_n = \mathbf{x} - \mathbf{x}_t \) for the tangential and normal components of a vector \( \mathbf{x} \). Now the external force contributions can be formulated as

\[
\mathbf{f}_1 = -\gamma_0 g, \quad (1.5)
\]

\[
\mathbf{f}_2 = A_0 \rho_w \left( C_M t \mathbf{a}_{r,t} + C_M n \mathbf{a}_{r,n} \right), \quad (1.6)
\]

\[
\mathbf{f}_3 = \frac{1}{2} \rho_w d \left( C_D t |\mathbf{v}_{r,t}| \mathbf{v}_{r,t} + C_D n |\mathbf{v}_{r,n}| \mathbf{v}_{r,n} \right) \left( 1 + \varepsilon \right), \quad (1.7)
\]

\[
\mathbf{f}_4 = A_0 \rho_w \mathbf{a}_w. \quad (1.8)
\]
Here \( \gamma = ((\rho_c - \rho_w) / \rho_c) \gamma_0 \) is the effective mass per unit length of the submerged cable. Indexes \( c \) and \( w \) indicate the cable and surrounding fluid (water) respectively. Thus, \( \rho_c \) and \( \rho_w \) are the cable and fluid densities. The terms \( C_{Mt} \), \( C_{Mn} \), \( C_{Dt} \) and \( C_{Dn} \) denote the hydrodynamic coefficients of added mass, tangential drag and normal drag forces respectively. The last three forces are functions of the relative velocity and relative acceleration of the fluid with respect to the mooring cable, \( v_{rel} \) and \( a_{rel} \), given by

\[
v_{rel} = v_w - \frac{\partial r}{\partial t}, \tag{1.9}
\]

\[
a_{rel} = a_w - \frac{\partial^2 r}{\partial t^2}. \tag{1.10}
\]

The contact forces contained in \( f_5 \) are implemented in different ways depending on the type of contact. At present, the only contact force applied is the ground model interaction, which is described in section 2.9.

### 1.3.2 Discretization scheme

As previously stated, Moody is based on the Local Discontinuous Galerkin Method (LDG). In this setting the cable domain \( \Omega \) is discretized into \( N_{el} \) elemental domains \( \Omega^e \in [s_e, s_u] \), where \( s_e \) and \( s_u \) are the define the upper and lower edges of the element.

Equation (1.1) is first rewritten as a first order equation system in space with an auxiliary variable \( q \).

\[
\frac{\partial^2 r}{\partial t^2} = \frac{1}{\gamma_0} \frac{\partial}{\partial s} \left( \frac{T}{1 + \epsilon} q \right) + \frac{f}{\gamma_0}, \tag{1.11}
\]

\[
q = \frac{\partial r}{\partial s}. \tag{1.12}
\]

\[
\epsilon = |q| - 1 \tag{1.13}
\]

Introducing the cable vector force \( T \) as

\[
T = EA \epsilon \frac{q}{1 + \epsilon} = Tt, \tag{1.14}
\]

equation (1.11) can be condensed into

\[
\frac{\partial^2 r}{\partial t^2} = \frac{1}{\gamma_0} \frac{\partial T}{\partial s} + \frac{f}{\gamma_0}. \tag{1.15}
\]

Within the \( e^{th} \) elemental region, the solution for an arbitrary function \( f \) is approximated by setting \( f(s, t) \approx f^e(s, t) = \sum_{i=0}^{p} \phi_i(s) \tilde{f}_i^e(t) \). Here \( \tilde{f}_i^e(t) \) denotes the local degrees of freedom of expansion coefficients and \( \phi_i \) are the expansion basis of order \( i \).
Denoting the inner product with $(\cdot, \cdot)$, the Galerkin approximation of equations (1.14) and (1.12) is obtained as

\[
\left( \phi_k, \frac{\partial^2 \mathbf{r}_h}{\partial t^2} \right)_{\Omega^e} = \frac{1}{\gamma_0} \left( \phi_k, \frac{\partial \mathbf{T}_h}{\partial s} \right)_{\Omega^e} + \frac{1}{\gamma_0} (\phi_k, \mathbf{f}_h)_{\Omega^e}, \quad \forall \ k,
\]

\[
(\phi_k, \mathbf{q}_h)_{\Omega^e} = \left( \phi_k, \frac{\partial \mathbf{r}_h}{\partial s} \right)_{\Omega^e}.
\]

Integrating the terms involving derivatives of $s$ by parts, exchanging the boundary flux terms with numerical fluxes, denoted by $\hat{\cdot}$, and integrating by parts once more yields

\[
\left( \phi_k, \frac{\partial^2 \mathbf{r}_h}{\partial t^2} \right)_{\Omega^e} = \frac{1}{\gamma_0} \left( \phi_k, \frac{\partial \mathbf{T}_h}{\partial s} \right)_{\Omega^e} + \frac{1}{\gamma_0} \left[ \phi_k (\hat{\mathbf{T}}_h - \mathbf{T}_h^u) \right]_{s_f}^u + \frac{1}{\gamma_0} (\phi_k, \mathbf{f}_h)_{\Omega^e},
\]

\[
(\phi_k, \mathbf{q}_h)_{\Omega^e} = \left( \phi_k, \frac{\partial \mathbf{r}_h}{\partial s} \right)_{\Omega^e} + \left[ \phi_k (\hat{\mathbf{r}}_h - \mathbf{r}_h^u) \right]_{s_f}^u.
\]

For the numerical fluxes, a modified version of the local discontinuous Galerkin (LDG) method developed by Cockburn and Shu [2] is used,

\[
\hat{\mathbf{r}}_h = \{ \mathbf{r}_h \} + \beta [ \mathbf{r}_h ],
\]

\[
\hat{\mathbf{T}}_h = \{ \mathbf{T}_h \} - \beta [ \mathbf{T}_h ] + \frac{\eta_1}{h} [ \mathbf{r}_h ] + \eta_2 h [ \mathbf{v}_h ],
\]

where $\eta_1$ and $\eta_2$ are constant mesh-independent parameters, $h$ is the non-dimensional element size, $\beta \in [-1/2, 1/2]$ controls the level of up- and downwinding of the fluxes, and the trace $\{ x \}$, and jump $[ x ]$ operators are defined as

\[
\{ x^e_h \} |_{s} = \frac{1}{2} \left( x^e_h |_{s_u} + x^{e+1}_h |_{s_f} \right) \quad \text{if} \quad s = s_u^e,
\]

\[
\{ x^e_h \} |_{s} = \frac{1}{2} \left( x^e_h |_{s_f} + x^{e-1}_h |_{s_u} \right) \quad \text{if} \quad s = s_f^e,
\]

\[
[ x^e_h ] |_{s} = \left( x^e_h |_{s_u} - x^{e+1}_h |_{s_f} \right) \quad \text{if} \quad s = s_u^e,
\]

\[
[ x^e_h ] |_{s} = \left( x^{e-1}_h |_{s_u} - x^e_h |_{s_f} \right) \quad \text{if} \quad s = s_d^e.
\]

The full description of the discretisation schemes and governing equations can be found in [4]. For more information about LDG methods, see [2].

1.3.3 Boundary Conditions

The boundary conditions of the cable objects are implemented weakly through the fluxes of the LDG scheme. Instead of the internal fluxes in (1.20) and (1.21),
specific boundary fluxes are used.

\[
\begin{align*}
\hat{r}_h &= g_D \quad \text{on } \Gamma_D, \\
\hat{r}_h &= r_e \quad \text{on } \Gamma_N, \\
\hat{v}_h &= g_D \quad \text{on } \Gamma_D, \\
\hat{v}_h &= v_e \quad \text{on } \Gamma_N, \\
\hat{T}_h &= T_e + \eta_1 (r_e - g_D) + \eta_2 (v_e - g_D) \quad \text{on } \Gamma_D, \\
\hat{T}_h &= g_N \quad \text{on } \Gamma_N,
\end{align*}
\]

(1.26) (1.27) (1.28)

where \( g_D \) and \( g_N \) are the Dirichlet and Neumann boundary condition values at the cable end points.

1.4 Note on performance

Moody combines the LDG method with high-order elements. Therefore there is a case-dependent trade-off between the number of elements and the elemental basis function order, and a suitable resolution is not always straightforward to accomplish.

Moody exhibits exponential convergence for smooth solutions such as the static catenary shape, see [4]. Thus engineering accuracy can be achieved with only a few elements of high order. On the other hand, the high-order scheme is well known to yield oscillatory results in the case of discontinuities in the solution. This is balanced by how the LDG method allows the solution to be discontinuous over element boundaries, thus relaxing the overshoot of the oscillations in the highest modes. Therefore snap loads are more genuinely captured by many elements of lower order, as the relaxation effect becomes more pronounced.

The present version of MOODY is not a commercial software. It is a development version written in Matlab code language for ease of feature development. As such, the code is not optimised and therefore computationally heavy in its present form, and it is therefore too early to draw conclusions on the final computational cost of the LDG and spectral \( hp \) formulation of the equations of cable dynamics. A comparison with other mooring models regarding simulation time cannot be made until the code is translated and compiled into a stand-alone software package.
2

The input file

2.1 General settings

The input file is a Matlab script file describing the mooring cable system that is to be simulated. The amount of information needed is dependent on the desired complexity of the system, but some compulsory parts pertaining to general information about the environment must always be included. In general, the format is based on several keywords that need to have assigned values in the input file. The general settings include gravity, still water level coordinate and the number of dimensions to compute in.

gravity
Gravity can be toggled on or off by input value 1 or 0.

dimensionNumber
The input is 1, 2 or 3 and defines a 1D, 2D or 3D simulation inside Moody. All vector values in the input file must still be set in 3D format. The input order of vector values in the input file is \([x, y, z]\), and the values used are extracted according to:

\[
\begin{align*}
\text{dimensionNumber} &= 1 \Rightarrow \text{1D simulation in } [x] \text{ only} \\
\text{dimensionNumber} &= 2 \Rightarrow \text{2D simulation in } [x, z] \text{ only} \\
\text{dimensionNumber} &= 3 \Rightarrow \text{3D simulation in } [x, y, z]
\end{align*}
\]

waterLevel
This refers to the still water level \(z\)-coordinate. The input is a scalar in \([m]\) and the default value is \(\text{inf}\). Thus everything is assumed to be submerged by default.
2. The input file

waterDensity
The water density applies to all fluid below the water level. The input is a compulsory scalar and the unit is \([\text{kg/m}^3]\).

airDensity
The air density applies to all fluid above the water level. The input is a scalar and the unit is \([\text{kg/m}^3]\). The default value is 0 kg/m³.

grid = 1;
dimensionNumber = 2;
waterDensity = 1000; \% [\text{kg/m}^3]
\% Optional \%
waterLevel = 0; \% [\text{m}]
airDensity = 0; \% [\text{kg/m}^3]
2.2 TIME SETTINGS

The time settings define the start and end time of the simulation. Here, also the time integration scheme, the time step size and the output time interval are set. Generally the input value for time should be in [s].

startTime
The start time of the simulation. Compulsory input in [s].

endTime
The end time of the simulation. Compulsory input in [s].

timeStep
The time step size used in the time integration. Compulsory input in [s].

saveInterval
The time between each output time in the result directory. This defines the output frequency of Moody. Compulsory input in [s].

timeStepScheme
A string specifying the choice of integration method. Compulsory input. Choices are:

expLeapFrog
The time step scheme is based on the explicit leap frog algorithm where the velocity is half a time step offset from the position.

expStormerVerlet
The time step scheme is based on the explicit Stormer-Verlet algorithm.

expRK3
Third order explicit Runge-Kutta scheme.

impAlpha
The time step scheme uses the general implicit alpha method with a default value of timeStepSchemeSettings = [α, δ] = [1/2, 1/2], matching the Newmark integration method. Here timeStepSchemeSettings is an additional, optional parameter.

```
startTime = 0;
endTime = 10;
timeStep = 1e-4;
saveInterval = 0.01;
timeStepScheme = 'expLeapFrog';
```
2.3 GEOMETRY

Moody’s geometry definition is based on vertices that define points in three dimensional space. A vertex is needed to define a physical location to any cable, rigid body or boundary condition object. The cell array `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 2.1. The following example creates vertex 1 and 2 at points \([0 \ 0 \ -50]\) and \([50 \ 0 \ 0]\) respectively.

```matlab
vertexLocations = {
    1 [0 0 -50];
    2 [50 0 0]
};
```
2.4 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.

2.4.1 Fields

diameter
The diameter field of the cable type defines the diameter (or nominal diameter in the case of chains) of the cable. If not specified it is computed from \( \gamma_0 \) and \( \rho \), assuming a constant density in a circular cross-section.
The input is in \([m]\).

\( \gamma_0 \)
This is the mass per meter of the cable. If not specified it is assumed that the cable is perfectly cylindrical and of constant density in the cross section. The default value is thus calculated from \( \rho \) and diameter. The input is in \([kg/m]\).

\( \rho \)
This is the density of the cable material. It is used to calculate the mass per meter (\( \gamma_0 \)) or the area of the cable, as well as defining the buoyancy of the submerged cable. The input is in \([kg/m^3]\).

\( CDn \)
Defining the drag force coefficient in the normal direction of the cable. Default value is 0.

\( CDt \)
Defining the drag force coefficient in the tangential direction of the cable. Default value is 0.

\( CM \) or \( CMn \)
Defining the added mass coefficient of the cable in the normal direction of the cable. Default value is 0.

\( CMt \)
Defining the added mass coefficient in the tangential direction of the cable. Default value is 0.
The material model is a substructure of the cable type. It contains information about the constitutive relations of the cable. Here you can specify axial stiffness $EA$, and internal damping coefficient, depending on the type of material. This is described in the section on material models, see section 2.7.

```matlab
cableType1 = struct();
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)
```
2.5 Cable objects

Cable objects are the main object type in Moody. It specifies a cable between two vertices and creates two files in the result directory. One contains the non-dimensionalised modal coefficients of position and velocity, and the other contains the modal coefficients of the spatial derivative vector along the cable. These are later post-processed to produce the physical results of position velocity, tension, strain and strain rate. Structures named cableObject1, cableObject2 and so on, are used to define the cable objects in the input file.

2.5.1 Fields

name
It is possible to define a name on the cable object. This name will be used to name result files in the result directory. There can be no duplicates of names on cables. The input is a string and the default value is the name of the structure itself. In the example below, it would have been 'cableObject1'.

typeNumber
Defining the cable type number that the cable is made up of. The input is a scalar.

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

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

length Defines the unstretched length of the cable. It is needed to compute the correct catenary at rest when that choice of initial condition is made. In other cases of initial conditions it is sometimes unused and overwritten by the computed length. The input is in [m].

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 2.6.

mesh
The mesh field is a substructure specifying the mesh properties of the cable. The possible fields of the mesh substructure are
2. The input file

\( N \) The number of elements in the cable.

\( P \) The polynomial order of the basis functions. Default is 1, meaning linear elements.

\( Q \) The number of quadrature points used in each element. Default is \( P+2 \).

type String specifying the polynomial basis type. Choices are Legendre, which is the default, Jacobi and Spectral.

cableObject1 = struct();
cableObject1.typeNumber = 1;
cableObject1.startVertex = 1;
cableObject1.endVertex = 2;
cableObject1.length = 100; % [m]
cableObject1.IC.type = 'PreStrain';
cableObject1.IC.eps0 = 0.05; % [-]
cableObject1.mesh.type = 'Legendre';
cableObject1.mesh.N = 10;
cableObject1.mesh.P = 7;
% Optional %
cableObject1.name = 'seawardMooringCable';
cableObject1.mesh.Q = 9; % (same as default)
2.6 Initial Conditions

The initial conditions are specified individually for each cable object, as a substructure of `cableObject`. There are different types of initial conditions, requiring particular fields of input.

2.6.1 PreStrain

The cable is initially a straight line between the two end point coordinate positions. The length of the cable is recomputed based on the distance between the points and the specified initial prestrain. Gravity is not taken into account in this initial condition.

- **type**
  
  The `type` field must be set to 'PreStrain' for this initial condition to be activated. The pre-strain type overrides the input field `length` in the cable object structure and instead computes it from the distance between the start and end points of the cables and the given pre-strain.

- **eps0**
  
  This defines 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`. Each part of the cable will then be given the matching pre-strain.

- **parts**
  
  This field specifies 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 default value of `parts` is 1, and is used when only a single value is given to `eps0`. The values of `parts` relates to the unstretched length fractions of the cable.

```matlab
IC = struct();
IC.type = 'PreStrain';
IC.eps0 = [0.01; 0.02];
% Optional %
IC.parts = [0.5; 0.5];
```

2.6.2 CatenaryStatic

The cable is initially set to be a hanging elastic catenary shape at rest. This option only works in 2D and 3D, when gravity is turned on.
2. The input file

2.6. The input file

`type`

The `type` field must be set to `'CatenaryStatic'` for this initial condition to be activated.

```
IC = struct();
IC.type = 'CatenaryStatic';
```

2.6.3 `HalfSine` type

This first creates straight, pre-strained cable between its two end points. Then it imposes a sinusoidally varying displacement on the vertical coordinate the cable. The default is a half period of sine.

`type`

The `type` field must be set to `'HalfSine'` for this initial condition to be activated.

`amplitude`

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

`periods`

Optional field that specifies the number of sinusoidal periods to use along the cable. The default is 1/2.

`eps0`

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 = struct();
IC.type = 'HalfSine';
IC.amplitude = 1; % [m]
IC.eps0 = 0.01;
% Optional %
IC.periods = 0.5;
```
2.7 Material models

The material model is a substructure of the cableObject and contains the information about the constitutive relation between axial force and strain in the cable. There are different types implemented.

2.7.1 linearCable

The linearCable type will compute the axial tension force of the cable according to Hooke’s Law, as \( T = EA \varepsilon \).

**type**

The type keyword for this material model is linearCable.

**EA**

The axial stiffness of the cable, specified as one single value in [N].

2.7.2 bilinearCable

The bilinearCable type will compute the axial tension force of the cable as a linear visco-elastic material, following \( T = EA \varepsilon H(\varepsilon) + C \dot{\varepsilon} \), where \( H(\varepsilon) \) is the step function outputting 1 for \( \varepsilon > 0 \) and 0 for \( \varepsilon < 0 \). This results in a linear cable for positive strains, and no force in compression. This type can also include internal damping through the coefficient \( C = 2\xi \sqrt{\gamma_0 EA (1m)^2} \), where \( \xi \) is the ratio of critical damping of a 1m cable segment equivalent oscillating single degree of freedom system.

**type**

The type keyword for this material model is bilinearCable.

**EA**

The axial stiffness of the cable, specified as one single value in [N].

**internalDamping**

Specifies the ratio of critical damping, \( \xi \). Here 1 is critical damping, 0 is no damping. Default value is 0.

2.7.3 multiLinearCable

The multiLinearCable type will compute the axial tension force of the cable following a partially linear stress-strain relation. The tension force in each part is computed as a bilinearCable.

**type**

The type keyword for this material model is multiLinearCable.
2. The input file

keyStrains

The keyStrains field is needed if there is more than one range of elasticity specified. The input is a vector of strains at which the linear range starts.

EA

The axial stiffness of the cable, specified as a vector of values in [N]. Each value corresponds to the same position in the keyStrains field so that \( EA(k) \) is the stiffness between index \( k \) and \( k+1 \) in keyStrains. The length of EA must be the same as keyStrains if the latter is defined.

internalDamping

Specifies the ratio of critical damping, \( \xi \). Here 1 is critical damping, 0 is no damping. Each value corresponds to the same position in the keyStrains field so that internalDamping\((k)\) is the damping ratio between index \( k \) and \( k+1 \) in keyStrains. The default value is 0 for all regions.

```matlab
materialModel = struct();
materialModel.type = 'multiLinearCable';
materialModel.EA = [1e5;1e3;10]; % [N]
materialModel.keyStrains = [0; 0.05; 0.1]; % [-]
% Optional %
materialModel.internalDamping = [0;0.05;0.1]; % [-]
```

The field keyStrains is only compulsory if the type ‘multiLinearCable’ is used with several inputs to EA. When EA is scalar, it is not needed. The use of other material model types is analogous to this example but with single value inputs to EA and internalDamping.
2.8 **Boundary Conditions**

In order for the FEM model to work, the appropriate boundary conditions for the problem must be applied. Boundary conditions are created through the input file in the same `structure.field` way as other objects.

### 2.8.1 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.

Ex: `bc1.type = {'neumann';'dirichlet';'dirichlet'};`

**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. The input is a scalar.

**mode**

The mode describes which boundary condition option that will be used. 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.

**startTime**

Defines the time at which the boundary condition starts to be active. For \( t < \text{startTime} \), the boundary value is held fixed at that of \( \text{startTime} \). The input is either a single value or a 3 by 1 vector specifying the start time of each coordinate direction independently. It is not applicable to \( \text{mode} = 'fixed' \). The input is in [s] and the default value is the same as the global start time of the simulation.

**endTime**

Defines the time at which the boundary condition stops to be active. For \( t > \text{endTime} \), the boundary value is held fixed at that of \( \text{endTime} \). 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 \( \text{mode} = 'fixed' \). The input is in [s] and the default value is the same as the global end time of the simulation.
2. The input file

2.8.2 fixed mode

Fixed mode is used to create a constant boundary value. It has only one possible additional field.

mode
The mode must be set to 'fixed' to activate this boundary condition.

direction
The direction of the value. The value is interpreted as the point resultant force in [N] for Neumann conditions, and as the global coordinate system position, in [m], for Dirichlet type conditions. The default value is the zero vector for force and the vertexNumber coordinates specified in the vertexLocations array for position. The input format is a 3 by 1 vector of values.

```matlab
bc1 = struct();
bc1.type = 'neumann';
b1c.mode = 'fixed';
b1c.vertexNumber = 1;
% Optional %
b1c.value = [-100;0;0]; % [N]
```

2.8.3 sine mode

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.

mode
The mode must be set to 'sine' to activate this boundary condition.

amplitude
The amplitude of the sinusoidal motion is specified by a 3 by 1 vector. The unit of the input is read as either [N] or [m] depending on the boundary condition type.

frequency
The frequency of the sinusoidal motion is specified by a 3 by 1 vector. The unit of the input is [Hz].

phase
The phase of the sinusoidal motion is specified by a 3 by 1 vector. The unit of the input is [deg].
2.8. Boundary Conditions

centerValue
If defined it determines the static offset around which the oscillation takes place. If omitted the force is assumed to be the zero vector or the coordinates of the vertexNumber, depending on the choice of type. The input is a 3 by 1 vector read as either [N] or [m].

rampTime
This is the ramping time, in [s], over which the amplitude is linearly increased from 0 to its full value. It starts the ramp from the set startTime of the boundary condition. The input is in [s] and the default value is 0, i.e. no ramping is applied by default.

bc1 = struct();
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;

2.8.4 linearInterp mode

The boundary condition is linearly interpolated between specified start- and end values over a time interval prescribed by the startTime and endTime keywords. In this mode, endTime and endValue are compulsory fields.

mode
The mode must be set to 'linearInterp' to activate this boundary condition.

startValue
The start value defines the value of the boundary at the beginning of the simulation. Input is a 3 by 1 vector read as either [N] or [m] depending on the boundary condition type. The default is the zero vector for force and the coordinates of the vertexNumber for position.

endValue
The end value defines the value of the boundary at the endTime. Input is a 3 by 1 vector read as either [N] or [m] depending on the boundary condition type. This is a compulsory field.
2. The input file

```matlab
bc1 = struct();
bc1.type = {'dirichlet';'dirichlet';'dirichlet'};
b1.mode = 'linearInterp';
b1.vertexNumber = 2;
b1.endValue = [10;0;0];
b1.endTime = 1;
% Optional %
b1.startValue = [5;0;0];
b1.startTime = 0;
```

2.8.5 *quadraticInterp* mode

The boundary condition is quadratically interpolated between specified start- and end values \(P_1\) and \(P_2\) over a time interval \([t_1, t_2]\), prescribed by the `startTime` and `endTime` keywords. Compulsory fields are `endTime` and `endValue`. The interpolated value is computed in the time interval \(\tau \in [0, t_2 - t_1]\) as

\[
P(\tau) = P_1 + v\tau + 0.5at^2, \tag{2.1}
\]

assuming initial velocity \(v\) and constant acceleration \(a\) computed from the constraint condition at \(t = t_2\) that \(P(t_2 - t_1) = P_2\).

In API mode (see chapter 4), the starting velocity is the average velocity over two time steps, computed as

\[
v_{API} = \frac{1}{2} \left( \frac{P_1 - P_0}{t_1 - t_0} + \frac{P_2 - P_1}{t_2 - t_1} \right), \tag{2.2}
\]

with index 0 indicating the value of the starting value of the previous time step.

**mode**

The mode must be set to `quadraticInterp` to activate this boundary condition.

**startValue**

The start value defines the value of the boundary at the beginning of the simulation. Input is a 3 by 1 vector read as either \([\text{N}]\) or \([\text{m}]\) depending on the boundary condition type. The default is the zero vector for force and the coordinates of the `vertexNumber` for position.

**endValue**

The end value defines the value of the boundary at the `endTime`. Input is a 3 by 1 vector read as either \([\text{N}]\) or \([\text{m}]\) depending on the boundary condition type. This is a compulsory field.
2.8. Boundary Conditions

Start Velocity
The start velocity defines the value of the velocity or rate of change of the boundary at startTime. Input is a 3 by 1 vector read as either [N/s] or [m/s] depending on the boundary condition type. The default is to start from rest.

```matlab
cbc = struct();
cbc.type = {'dirichlet';'dirichlet';'dirichlet'};
cbc.mode = 'quadraticInterp';
cbc.vertexNumber = 2;
cbc.endValue = [10;0;0];
cbc.endTime = 1;
% Optional %
cbc.startValue = [5;0;0];
cbc.startTime = 0;
cbc.startVelocity = [2;0;0];
```

2.8.6 cubicInterp mode
The boundary condition is cubically interpolated between a specified start- and end value over a time interval prescribed by startTime and endTime. In this case endTime and endValue are compulsory fields, and there is an option to specify the start- and end velocities.

**mode**
The mode must be set to 'cubicInterp' to activate this boundary condition.

**startValue**
The start value defines the value of the boundary at the beginning of the simulation. Input is a 3 by 1 vector read as either [N] or [m] depending on the boundary condition type. The default is the zero vector for force and the coordinates of the vertexNumber for position.

**endValue**
The end value defines the value of the boundary at the endTime. Input is a 3 by 1 vector read as either [N] or [m] depending on the boundary condition type. This is a compulsory field.

**startVelocity**
The start velocity defines the value of the velocity or rate of change of the boundary at startTime. Input is a 3 by 1 vector read as either [N/s] or [m/s] depending on the boundary condition type. The default is to start from rest.
2. The input file

endVelocity
The end value defines the value of the boundary at endTime. Input is a 3 by 1 vector read as either [N/s] or [m/s] depending on the boundary condition type. The default is to end at zero velocity.

```
bc1 = struct();
bc1.type = 'dirichlet';
bc1.mode = 'cubicInterp';
bc1.vertexNumber = 2;
bc1.endValue = [10;0;0];
bc1.endTime = 1;
% Optional %
bc1.startValue = [5;0;0];
bc1.startTime = 0;
bc1.startVelocity = [0;0;0];
bc1.endVelocity = [0;0;0];
```
2.9 GROUND MODELS

There is at present only one type of ground model implemented in Moody.

2.9.1 springDampGround

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.

\[
f_{5z} = Kd \left( r_z - z_g \right) - 2\xi \sqrt{\gamma_0 Kd \max(\dot{z}, 0)}
\]

\[
\dot{r}_{\text{hor}} = \frac{\left[ \dot{r}_x, \dot{r}_y \right]}{\max(v_c, ||\dot{r}_x, \dot{r}_y||)}
\]

\[
f_{5xy} = -\gamma_e g \mu \dot{r}_{\text{hor}}
\]

\[
f_5 = \left[ f_{5xy}, f_{5z} \right]
\]

Here \( d \) is the cable diameter, \( \gamma_e \) the effective mass per meter, \( r_z \) notes the \( z \)-component of the position of the cable and \( \dot{r}_z \) its time derivative. Other notation is explained in the input description below.

**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 (2.3) as \( z_g \).

**stiffness**

The stiffness of the ground is specified in [Pa/m]. Goes into (2.3) as \( K \).

**dampingCoeff**

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 (2.3) as \( \xi \).

**frictionCoeff**

This is the friction coefficient between any cable and the ground. The default value is 1. Goes into (2.5) as \( \mu \).

**vc**

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. The input should be in [m/s] and the default value is 0. Goes into (2.4) as \( v_c \).
2. The input file

```matlab
ground = struct();
ground.type = 'springDampGround';
ground.level = -50; // ground.stiffness = 1e6;
% Optional %
ground.dampingCoeff = 1;
ground.frictionCoeff = 0.3;
ground.vc = 0.1;
```
2.10 RIGID BODIES

Mooring elements such as submerged floaters and sinkers can be modelled using rigid body type objects. 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. 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.

2.10.1 Common fields

name
It is possible to input a name of the rigid body. The default is the name of the rigid body structure, such as ‘rigidBody1’.

centerVertex
The center vertex defines which vertex the body is appointed to. It is considered as the center of floatation of the body.

mass
The mass of the body, specified in [kg]. This may not be 0, as the solver needs some inertia.

startVelocity
The body can be given a start velocity. The input is a 3 by 1 vector in [m/s] of the global coordinate system. The default value is [0;0;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.

2.10.2 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.
2. The input file

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

volume
The volume of the body. This is used to simulate floaters. The input is in [m³] and the default value is 0.

```matlab
rigidBody1 = struct();
rigidBody1.type = 'rigidPoint';
rigidBody1.name = 'sinker1'
rigidBody1.centerVertex = 2;
rigidBody1.mass = 1; % [kg]
% Optional %
rigidBody1.volume = 0;
rigidBody1.CDn = 1;
rigidBody1.CM = 2.5;
rigidBody1.startVelocity = [1; 0; 0];
```

2.10.3 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
The height of the cylinder must be specified. The input is in [m].

diameter or area
Either the diameter or the cross-sectional area of the cylinder must be specified as single values. The input is thus either in [m] or in [m²].

mass or density
Either the mass or the density of the cylinder must be specified as single values. The input is thus either in [kg] or in [kg/m³]. If density is specified, the body is assumed to be made of homogenous material, and the mass will be that of a solid cylinder.

momentOfInertia
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 [kgm²] and formatted as a 3 by 3 matrix defining the values of the inertia matrix of the body.
2.10. Rigid bodies

cogOffset

To model bodies that are non-homogenous, there is an option to offset the center of gravity. In this case, cogOffset defines a scalar distance along the symmetry axis from the center of floatation where the center of gravity is located. All the moments acting on the body are computed around this point. The input is a single value in [m], specified in the body-fixed coordinate system.

slaves

There can be several cables attached to each cylinder and they need not be connected to the center of floatation of the body. The slaves field specifies a list of vertices that are attached to the rigid body and whose motion is thereby 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.

CDt

Also the tangential drag can be specified through the input of this field. The input is a single, unitless value with default value 0.

omega

It is possible to specify an initial rotational speed of the object. This is done by setting the field omega to any 3 by 1 vector of angular frequency [rad/s].

q

By setting the value of q in the input, it sets the initial rotation of the body compared to the global inertial coordinate system. The 4 by 1 vector input is read as the quaternion of rotation for the body, with total norm 1. The default value is [1;0;0;0], meaning that the local body z-coordinate is aligned with the global vertical direction.

```matlab
rigidBody1 = struct();
rigidBody1.type = 'rigidCylinder';
rigidBody1.centerVertex = 2;
rigidBody1.mass = 5; % [kg]
rigidBody1.height = 0.5;
rigidBody1.diameter = 0.2;
% Optional %
rigidBody1.name = 'sinker1';
rigidBody1.momentOfInertia = 100*eye(3);
rigidBody1.cogOffset = 0;
rigidBody1.slaves = [1; 3];
```
2. The input file

```plaintext
rigidBody1.CDn  = 1;
rigidBody1.CDt  = 0.1;
rigidBody1.CM   = 2.5;
rigidBody1.startVelocity = [0; 0; 0];
rigidBody1.omega = [0;0;0];
rigidBody1.q = [1; 0; 0; 0];
```
After the setting up of the input file, the software can be started in a number of different modes. The main function is called `moody` and takes a number of different input arguments. In its most basic form, the simulation is performed completely as it is prescribed by the input file. The example below will generate an output folder with the same name as the input file in your present working directory.

```matlab
moody('inputFileName');
```

Other input options are also available and are in general possible to combine in the same call. Here follows a list of keyword strings that can be used as additional input.

- **outputFilename**
  Specify a name of the output directory.

  ```matlab
  moody('inputFile','outputFilename','test1');
  ```

- **startState**
  Using this option enables restarting the simulations from an existing result folder. In this case, the first input is interpreted as the name of the result directory. When no start time is specified, the simulation will resume at the last saved time entry.

  Example 1, start from the last time of the results:

  ```matlab
  moody('resultDir','startState');
  ```
3. Running the code

Example 2, start from a particular time:

```matlab
moody('resultDir','startState',10);
```

**addInput**

This option enables the user to overwrite, and expand information given in the input file in the call to Moody. The values are overwritten with the eval command.

```matlab
inArray={'bc1.frequency',2,'waterLevel',50};
moody('inputFile','addInput','inArray');
```

**endTime**

A special, simplified option for controlling the end time of simulation as a function call.

```matlab
moody('inputFile','endTime',50);
```

**API**

This flag is used the first time moody is called and starts up a moody simulation in API mode. The exact usage is described in chapter 4.
3.2 POST PROCESSING

During the simulation, data is saved to files in the output directory. All the objects and the general setup of the simulation are saved in setup.mat in the result directory. This is needed if the simulation is to be restarted or continued from a certain output state.

The following subsections explains how to evaluate, interpret and visualise the output data from a Moody simulation.

3.2.1 Evaluate results

The modal values of each object are saved in .dat-files during the computation. The modal file for position and velocity is named after the object and the file for the fluxed spatial derivative $q$ is named with a $qM$ suffix. These values need to be projected into physical space to make them understandable. This is done by evaluating the results in the function evaluateResults('resultDir').

The result evaluation creates several new files for each object in the result directory. The default evaluation is that used in the computation, but one can also specify the number of quadrature points in the output, through the keyword-value pair: (‘Q’, scalar).

The files created have the same name as the objects in the result directory, followed by the description of the file content. Here examples are given in the three dimensional format although the actual output from a simulation is dependent on the choice of dimension number in the input file.

cableObject1.dat
cableObject1_qM.dat
cableObject1_position.dat
cableObject1_velocity.dat
cableObject1_tension.dat
cableObject1_tensionVector.dat
cableObject1_strain.dat
cableObject1_strainRate.dat
cableObject1_sPlot.dat
rigidBody1.dat
rigidBody1_position.dat
rigidBody1_velocity.dat
time.dat
setup.mat
Table 3.1: Output structure description for cable object data. Times are indexed as $t_1, t_2, \ldots, t_m$, vector components as $\mathbf{f} = [f_x, f_y, f_z]$ and the cable unstretched coordinate $s$ is indexed as $s_1, s_2, \ldots, s_n$.

<table>
<thead>
<tr>
<th>File suffix</th>
<th>Output structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>_position.dat</td>
<td>$t_1 \quad x(t_1, s_1 : s_n) \quad y(t_1, s_1 : s_n) \quad z(t_1, s_1 : s_n)$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$t_m \quad x(t_m, s_1 : s_n) \quad y(t_m, s_1 : s_n) \quad z(t_m, s_1 : s_n)$</td>
</tr>
<tr>
<td>_velocity.dat</td>
<td>$t_1 \quad v_x(t_1, s_1 : s_n) \quad v_y(t_1, s_1 : s_n) \quad v_z(t_1, s_1 : s_n)$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$t_m \quad v_x(t_m, s_1 : s_n) \quad v_y(t_m, s_1 : s_n) \quad v_z(t_m, s_1 : s_n)$</td>
</tr>
<tr>
<td>_tensionVector.dat</td>
<td>$t_1 \quad T_x(t_1, s_1 : s_n) \quad T_y(t_1, s_1 : s_n) \quad T_z(t_1, s_1 : s_n)$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$t_m \quad T_x(t_m, s_1 : s_n) \quad T_y(t_m, s_1 : s_n) \quad T_z(t_m, s_1 : s_n)$</td>
</tr>
<tr>
<td>_tension.dat</td>
<td>$t_1 \quad T(t_1, s_1 : s_n)$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$t_m \quad T(t_m, s_1 : s_n)$</td>
</tr>
<tr>
<td>_strain.dat</td>
<td>$t_1 \quad \varepsilon(t_1, s_1 : s_n)$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$t_m \quad \varepsilon(t_m, s_1 : s_n)$</td>
</tr>
<tr>
<td>_strainRate.dat</td>
<td>$t_1 \quad \dot{\varepsilon}(t_1, s_1 : s_n)$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$t_m \quad \dot{\varepsilon}(t_m, s_1 : s_n)$</td>
</tr>
<tr>
<td>_sPlot.dat</td>
<td>$t_1 \quad s_1 : s_n$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.2.2 Cable output

The first column of each output file contains the time stamp of the output, $t = [t_1, t_2, \ldots, t_m]$. Thus all output 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 as tension vector, 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 3.1.
3.2. Post processing

Table 3.2: Output structure description for rigid body object data. Times are indexed as $t_1, t_2, \ldots, t_m$, vectors and vector components of property $f$ as $\vec{f} = [f_x, f_y, f_z]$ and the slave points are indexed as $p_1, p_2, \ldots, p_n$, where index 1 indicates the center of gravity. Here also $\vec{q}$ is the 1 by 4 quaternion for the present orientation of the body, and $\vec{\omega}$ is the angular velocity vector.

<table>
<thead>
<tr>
<th>File suffix</th>
<th>Output structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dat</td>
<td>$t_1 \quad \vec{x}(t_1, p_1) \quad \vec{y}(t_1) \quad \vec{v}(t_1, p_1) \quad \vec{\omega}(t_1)$</td>
</tr>
<tr>
<td></td>
<td>$\vdots \quad \vdots \quad \vdots \quad \vdots$</td>
</tr>
<tr>
<td></td>
<td>$t_m \quad \vec{x}(t_m, p_1) \quad \vec{y}(t_1) \quad \vec{v}(t_m, p_1) \quad \vec{\omega}(t_m)$</td>
</tr>
<tr>
<td>_position.dat</td>
<td>$t_1 \quad x(t_1, p_1 : p_n) \quad y(t_1, p_1 : p_n) \quad z(t_1, p_1 : p_n)$</td>
</tr>
<tr>
<td></td>
<td>$\vdots \quad \vdots \quad \vdots \quad \vdots$</td>
</tr>
<tr>
<td></td>
<td>$t_m \quad x(t_m, p_1 : p_n) \quad y(t_m, p_1 : p_n) \quad z(t_m, p_1 : p_n)$</td>
</tr>
<tr>
<td>_velocity.dat</td>
<td>$t_1 \quad v_x(t_1, p_1 : p_n) \quad v_y(t_1, p_1 : p_n) \quad v_z(t_1, p_1 : p_n)$</td>
</tr>
<tr>
<td></td>
<td>$\vdots \quad \vdots \quad \vdots \quad \vdots$</td>
</tr>
<tr>
<td></td>
<td>$t_m \quad v_x(t_m, p_1 : p_n) \quad v_y(t_m, p_1 : p_n) \quad v_z(t_m, p_1 : p_n)$</td>
</tr>
</tbody>
</table>

3.2.3 Rigid body output

Rigid bodies are calculated directly in the physical domain and therefore the modal values in the .dat-file of the object are more easily readable. The structure of the position and velocity output is analogous to the cable object only that instead of points along the cable, the indexing is made for each slave point of the body, beginning with the center of gravity as index 1. Table 3.2 presents the structure of the different output files.

3.2.4 Make a movie

To visualise the data, Moody comes with a make movie function called `mkMovie`. In its basic call, with only the output directory name as an input, `mkMovie` creates a movie of the position evolution of all objects with one frame per saved timestep. Example:

```matlab
h=mkMovie('outputFolderName');
```

Here $h$ is a handle to the figure of the movie. Below, other possible (keyword, value) pair inputs are listed.

- **tension**

  When this key string is found in the input list, `mkMovie` plots the tension of each cable of the simulation in individual subplots. It thereby disregards all object positions. No value is assigned to this keyword.
3. Running the code

**objects**
Used to modify which objects to plot for. The following input value must be a cell array of valid object names in string format. For a single object, a single string file name can be specified directly.

**timeWindow**
This option is used to specify a start- and an end time of the movie. The following input value must be a 2 by 1 vector as $[t_{\text{start}}, t_{\text{end}}]$.

**frames**
Specifies the total number of frames in the movie. This is essential for long simulations with small time steps. The ensuing value must be a scalar.

**dims**
Controls the dimensions of the plotted output. Only applies when tension is not present. The following input value must be in the format $[1, 1, 1]$. This example gives a 2D plot of the x-z plane of a three dimensional simulation.

Example:

```matlab
h=mkMovie('outputName','frames',200,'objects',
{'cableObject1'},timeWindow,[0 10], 'dims',[1 0 1])
```
Use Moody as an external module

4

4.1 INTRODUCTION TO API

Moody is designed to be used as a modular add-on for an external solver for the fluid problem. This is done by means of a so called API, an Automated Program Interface. When Moody is started up in API-mode the external solver is guiding the time evolution of the boundary conditions of Moody. The interaction between the codes is made as an information loop, see figure 4.1.

When the time step size of the external solver, such as the OpenFOAM CFD simulation in the case of figure 4.1, is larger than the internal time step of Moody, a sub-stepping is performed. Thus Moody interpolates in time between the old and the new boundary values received. The interpolation can be either cubic or linear, where the cubic interpolation requires also the velocity of the boundary at each time step.

The usage of the API can be separated into a startup call, where all Moody objects are initialised and the API determines which boundary conditions to control, and the time loop call that is used for moving the mooring solver forward in time.
4. Use Moody as an external module

Figure 4.1: Schematic description of the information loop between Moody and an external software, represented by an OpenFOAM CFD simulation in this case.

4.2 START UP THE API

The API is activated by including the keyword ‘API’ in the initial call to Moody. The initial call is used to set up the case according to the input file and create all the objects. It does not perform any dynamic simulation during the startup call. There are two arguments needed after the ‘API’ keyword. First, the time in [s] of the start of the simulation and second, the boundary condition matrix (or array). This last entry must contain the boundary condition numbers that the API controls and the value of these boundary conditions at the given starting time. The format of this input must be either a matrix or an array.

```matlab
[edgeVals,moodyState] = ... moody('inputFileName','API',tStart,values);
```

Here:

- **tStart**
  - Defines the start time of the moody environment. The input should be scalar.

- **values**
  - This contains the information of the boundary conditions and their values. The numbers specified as `bcNum` below must correspond to defined boundary conditions of mode `quadraticInterp`, `cubicInterp` or `linearInterp` in the input file. The input can be in either array or matrix format.

  Array format example:
4.2. Start up the API

```plaintext
{[bcNum1 bcVals1]; [bcNum2 bcVals2]; ...}
```

Matrix format example:

```
[bcNum1, bcVals1; bcNum2, bcVals2; ...]
```

The appearance of `bcVals` changes depending on the mode of the controlled boundary condition. For `linearInterp`, only the position or force is necessary. These values need to be appended with the velocity or force rate of change for `cubicInterp` boundaries. The following is an example how to send a vector property \( \mathbf{g} = [g_x, g_y, g_z] \), and its velocity \( \mathbf{v} \) as an API boundary condition to Moody.

**linearInterp format example:**

```
bcVals1 = [g_x, g_y, g_z]
```

**cubicInterp format example:**

```
bcVals1 = [g_x, g_y, g_z, v_x, v_y, v_z]
```

`edgeVals`

This is the matrix of the output from Moody. Each line in `edgeVals` contains the boundary condition number, `bcNum` as defined in `values`, and the output value at the boundary in all active coordinate directions. If `bcNum` refers to a Dirichlet boundary, the returned vector is the vectorial sum of forces from cables attached to the controlled point. The forces are directed in the outward pointing normal direction of each cable. In the case of `bcNum` referring to a Neumann condition, the values are simply the present coordinates of the vertex.

```
edgeVals = [1 10 0 10; 2 5 0 0]
```

`moodyState`

This is a 3 by 1 cell array containing the objects, settings and the present time value. This is a needed input for the time loop call to `moody` described below.
4. Use Moody as an external module

4.3 Time Loop Call

Based on the edgeVals the external solver can now integrate and move forward in time to produce new, updated values of the boundary conditions. These can then be sent back to Moody.

\[
[\text{edgeVals},\text{moodyState}] = \text{moody}(\text{moodyState}, t, \text{values});
\]

Here \( t \) is the new time, \( \text{values} \) contain the boundary condition information at time \( t \) and is otherwise formatted as in the initial API call. The input \( \text{moodyState} \) should be as returned by Moody’s initial call. The output version of \( \text{moodyState} \), as well as the \( \text{edgeVals} \) matrix, corresponds to time \( t \).

It is important to note that \( \text{moodyState} \) contain references to handle objects in Matlab, meaning that a renaming of the output to something else will not leave the input unaffected by the computation.

4.3.1 Using trial time steps

As Moody can be writing the results to the result directory several times per time loop call, the use of time step schemes that use trial time steps in the external solver have to be treated somewhat differently. Such time step schemes include among others higher order Runge-Kutta schemes and implicit schemes. In order to allow Moody to differentiate between trial steps and final steps, a slight modification is made to the input of the time loop call. The present time \( t \) is here instead sent as a time interval defined by two values, a start- and an end time.

\[
[\text{edgeVals},\text{moodyState}] = \text{moody}(\text{moodyState}, [t_0 \ t], \text{values});
\]

Here \( t_0 \) is the time of the last decisive step of the scheme, and \( t \) is the time of the trial step, at which \( \text{values} \) are to be applied. By saving a copy of \( \text{moodyState} \) corresponding to \( t_0 \) internally, Moody can then rewind results connected with trial, intermediate steps that are not part of the final solution.
REFERENCES


