University of Windsor

**Mechanical, Automotive, & Materials Engineering**
401 Sunset Avenue
Windsor, Ontario, Canada N9B 3P4
519 253 3000 (2616)
www.uwindsor.ca/mame

# EoM User Manual

BP Minaker
RJ Rieveley

January 12, 2017

# Contents

# List of Tables

Chapter 1

# EoM User Manual

This document explains the use of the University of Windsor Vehicle Dynamics and Control Research Group Equations of Motion (EoM) software. This software can be used to automatically generate the linear or linearised equations of motion for mechanical systems. These equations can be used for engineering analysis and simulation. The code is published under the GNU Public Licence (GPL), and so is free to distrbute. It is designed to run in Matlab® or the similar open-source code Octave.

## 1.1 Introduction

The EoM code can generate the equations of motion for a three dimensional mechanism composed of rigid bodies, and coupled by either rigid or flexible connectors. The input required by the code is a function file that returns a specially formatted structured variable describing the properties of the system, using a protocol described in this document. Complex systems and sub-items can be built from combinations of the base elements, or defined automatically within purpose developed add-on code. The equations of motion that are generated are linearized, and the code output is the descriptor state space form (A,B,C,D,E matrices) of these equations. Some typical linear analysis of the resulting equations is conducted, including eigenvalue and frequency response. A LaTeX interface has been developed, and the results are presented in a `report.tex` source file that can be compiled to give formatted output. If you are unfamiliar with the LaTeX document preparation system, much useful information can be found online at the websites `www.latex-project.org` or `miktex.org`.

## 1.2 Folder structure

The folder layout in EoM is quite simple. The main calling function file, `run_eom.m`, resides in the top level folder `inst`. Note that `run_eom.m` cannot be run directly from the Matlab® editor, as it requires arguments, and must be called from the command line. Alternatively, it can be used in a simple script, such as `sample_1.m`, which can be executed from the editor. There are also two sub-folders here, `data` and `apps`. The `data` folder contains two further subfolders; the `input` folder, where EoM searches for the system definition function, and the `output` folder, where the results of the analysis are stored. Each analysis generates subfolders that are named by date and by time to prevent overwriting of old results. The EoM code itself is stored in the `apps` folder, in the `eom` subfolder. The other subfolders, e.g., `anim`, `tex_utils`, etc., store various bits of code used alongside EoM to generate animations, or reports.

## 1.3 Developing a model

The EoM syntax is straightforward and can be arranged in a number of different ways, depending on the discretion of the user. Systems are defined in standard function `.m` files to allow for sub-functions, and iterative definitions. These definitions can create the system step by step by adding individual elements, or as members of a group structure with assembly at the end of the input file. The elements of the system are assembled in the input file using a structured variable, typically named `the_system`, containing a list (or more precisely, a one dimensional cell array) of items of predefined types, e.g., `the_system.item{1}=my_first_item`. Each item is in turn a structured variable with a precisely defined set of fields. The data structure of each item must contain a field `type`, holding a string that must be the name of one of the item types listed in this document, e.g., `my_first_item.type='body'`. Several other fields must be included, depending on the type of item, and several optional fields may also be included, e.g., `group` may be used to group items. Note that the system definition file should be a function that takes arguments as necessary, but returns only the structured variable containing the system definition.

If the system defintion file takes a single argument used to define some parameter, e.g. the speed of a vehicle used when conducting the linearisation, or some spring stiffness, EoM can be called in 'batch mode', to call the function with a range of values, and generate the equations of motion for each case. Several example input files can be found in the `examples` folder. These files can be used to understand the syntax and use of EoM, and range in complexity from simple spring-mass-damper sytems, to bicycles and vehicle suspensions.

## 1.4 System elements

There are currently thirteen type of system elements recognized in EoM:

**fundamental**: `body`, `load`

**flexible connectors**: `spring`, `flex_point`, `beam`, `triangle_3`, `triangle_5`

**rigid connectors**: `link`, `rigid_point`, `nh_point`

**input/output**: `actuator`, `sensor`

The basic building block of the system is the `body` type item, which adds a rigid body to the system. Any other item type in the system must be attached to one (or more) rigid bodies. The other items represent both rigid and flexible connectors. These connectors may be point connectors, i.e., their geometry is defined at a single point, like the `rigid_point` and `flex_point`, or line connectors that require two points, like the `link` and `spring`. The flexible connectors may be chosen to carry either tension or torsion. Additionally, there is a `beam` (spring) item that carries both bending and shear. Loads may be applied using an `actuator` item, and displacements may be measured using a `sensor` item. Constant preloads can be applied using a `load` item. Nonholonomic constraints may be applied using a `nh_point` item. A very crude finite element type analysis may be built using the `triange_3` or `triangle_5` items; these are planar elastic elements.

**item type** `body`

A rigid body. Properties include name, location of the centre of mass, mass, moments of inertia, and cross products of inertia. Items of type `body` must have a unique `name` property defined, all other item types can be optionally named for ease of identification, but this is not necessary. A ground body is predefined; it may not be defined in the input file. All other item types must be attached to a body, determined by name. Bodies can be defined with mass or inertia equal to zero if desired, if for example, one wishes to define a spring and damper acting in series. Please note the sign convention used for the cross products of inertia; EoM expects the positive integral as input, and changes the sign when assembling the inertia matrix (some sources define the individual entries as the negative integral). Note also the sequence of the inertia values (xy,yz,zx). The velocity or angular velocity can be specified; the system is assumed to be in equilibrium and the equations are linearised around the velocity condition specified. There is no check to see if these specified velocities satisfy the constraints (at least, not yet).

**Table 1.1**: item type `body`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `body` |
| name | text string | identifies the body, required, and must be unique |
| location | column 3-vector | location of centre of mass |
| mass | scalar | mass |
| momentsofinertia | column 3-vector | the xx, yy, and zz mass moments of inertia |
| productsofinertia | column 3-vector | the xy, yz, and zx cross products |
| velocity | column 3-vector | (*optional*) linearization velocity |
| angular_velocity | column 3-vector | (*optional*) linearization angular velocity |

### item type `spring`

A two point elastic spring, with optional linear damping and relative inertia. To be precise, the stiffness is also optional, but if none of stiffness, damping, or inertia are specified, the `spring` has no effect. Properties include the location of each end, names of the two bodies to which it is attached, and stiffness, damping and inertance values. If the `preload` field is specified, it will be used, if the system is otherwise statically indeterminate. If the preload is not specified, for statically determinate systems, it will be calculated from statics, and for indeterminate systems it will be calculated from stiffness and statics. It is not necessarily assumed to be zero if unspecified. An equilibrium check is conducted before the analysis begins to ensure the preloads are correctly specified. If there is insufficient information to uniquely determine the equilibrium preloads, EoM will make a 'best guess', throw a warning, and proceed if the equilibrium check passes. To define a torsional spring, simply set the field `twist` to 1.

**Table 1.2:** item type `spring`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `spring` |
| name | text string | (*optional*) identifies the spring |
| location1 | column 3-vector | location of end1 |
| location2 | column 3-vector | location of end2 |
| body1 | text string | identifies which body is attached to end1 |
| body2 | text string | identifies which body is attached to end2 |
| stiffness | scalar | (*optional*) stiffness (AE/L for simple tension) |
| damping | scalar | (*optional*) damping |
| inertia | scalar | (*optional*) inertance (relative inertia) |
| preload | scalar | (*optional*) specifies a spring preload force |
| twist | binary integer | (*optional*) specifies tension (default) or torsion |

**item type `link`**

Similar to a spring, but inextensible. Preload cannot be specified, but is determined as part of the solution process.

**Table 1.3**: item type `link`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `link` |
| name | text string | (*optional*) identifies the link |
| location1 | column 3-vector | location of end1 |
| location2 | column 3-vector | location of end2 |
| body1 | text string | identifies which body is attached to end1 |
| body2 | text string | identifies which body is attached to end2 |

### item type `rigid_point`

A generic constraint. Properites include the number of forces that are carried, and the number of moments, plus location, and the names of the two bodies to which it is attached. Acceptable choices for the number of forces and moments is 0, 1, 2, or 3. If there are either 1 or 2 forces or moments carried, a direction vector must be included as well; this is not necessary for the 0 or 3 cases. In the case where 1 force or moment is carried, the axis defines the direction of the force or moment. If 2 forces or moments, the axis defines the normal to the plane in which the forces or moments lie. For example, a ball joint passes three forces, zero moments, no axis definition is necessary. For a hinge, three forces, two moments, and an axis are necessary, and it is the axis of rotation of the hinge. For point-on-plane contact, 1 force and 0 moments are transmitted, the axis is required and defines the normal to the plane. Both forces and moments use the same axis, so in some cases it becomes necessary to put two `rigid_points` at the same location to build a compound constraint. If an axis is specified in the case where it is not needed, EoM will ignore it. Although perhaps confusing initially, it is a compact method that can define many different types of constraints.

To define constraints with rolling contact, the optional field `rolling_axis` should be defined; this field is used only in the calculation of the tangent stiffness matrix, and has no effect unless the contact carries preload when the system is in equilibrium. Important: for constraints where sliding contact is present, it is assumed that the location of the point of contact remains fixed relative to body 1, and that the orientation of the contact surface is determined by body 2. For most cases then, be sure that body 1 is the rolling or sliding body, and that body 2 is the (flat) surface.

**Table 1.4**: item type `rigid_point`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `rigid_point` |
| name | text string | (*optional*) identifies the constraint |
| location | column 3-vector | location of constraint |
| body1 | text string | identifies which body is attached to end1 |
| body2 | text string | identifies which body is attached to end2 |
| forces | integer | number of forces transmitted (0, 1, 2, or 3) |
| moments | integer | number of moments transmitted (0, 1, 2, or 3) |
| axis | column 3-vector | direction or normal vector |
| rolling_axis | column 3-vector | (*optional*) direction of rolling axis |

### item type `flex_point`

A single point spring with translational and rotational stiffness and damping, i.e., a flexible version of the rigid point above, useful to model bushings. Properties include the location, the names of the two bodies to which it is attached, the number of forces and moments passed, axes as necessary, linear and torsional stiffness, and linear and torsional damping. Inertance is not modeled. The stiffness that is given is assumed to be the same in all directions that are defined as flexible. Flex points cannot have different stiffnesses in different directions, but the same result can be accomplished with two flex points at the same location. Both stiffness and damping values are optional, but if neither are set, the `flex_point` has no effect. Note that rolling contact can be defined as in the `rigid_point` above, in the case of a flexible tire. Again, the rolling axis has no effect unless the connection is carrying a preload force when the system is in equilibrium.

**Table 1.5:** item type `flex_point`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `flex_point` |
| name | text string | (*optional*) identifies the flex point |
| location | column 3-vector | location of flex point |
| body1 | text string | identifies which body is attached to end1 |
| body2 | text string | identifies which body is attached to end2 |
| forces | scalar | number of forces transmitted (0, 1, 2, or 3) |
| moments | scalar | number of moments transmitted (0, 1, 2, or 3) |
| axis | column 3-vector | direction or normal vector |
| stiffness | column 2-vector | (*optional*) stiffness and torsional stiffness values |
| damping | column 2-vector | (*optional*) damping and torsional damping values |
| rolling_axis | column 3-vector | (*optional*) direction of rolling axis |

### item type `beam`

A massless beam spring, with identical area moments of inertia in both bending directions, and no cross products of area. Bending and shear stiffness are modelled, but not tension or torsion. Note that a 'stress softening' effect (a reduction in flexural stiffness in the presence of compressive loads) can be included by placing a `spring` item collinearly with the beam (although it does not include shortening effects). Properties include the location of each end, the names of the two bodies to which it is attached, and the stiffness value. No damping or inertance is modeled.

**Table 1.6**: item type `beam`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `beam` |
| name | text string | (*optional*) identifies the beam |
| location1 | column 3-vector | location of end1 |
| location2 | column 3-vector | location of end2 |
| body1 | text string | identifies which body is attached to end1 |
| body2 | text string | identifies which body is attached to end2 |
| stiffness | scalar | bending stiffness (EI/L) |

### item type `load`

Constant forces or moments applied to the system. Used to determine preloads in the connecting elements, and eventually to find the tangent stiffness matrix. Allows one to model the oscillation of a simple pendulum due to gravity, for example, or the weave and wobble of a bicycle. Properties include the location of point of application, the name of the body to which it is attached, and the force and moment vectors. To simplify the addition of `load` items, the helper function `weight(body_item,g)` is included, which applies an appropriate weight load in the negative z direction to the mass centre of the `body` sent as an argument. The gravity argument is optional, and defaults to 9.81 if not specified.

**Table 1.7**: item type `load`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `load` |
| name | text string | (*optional*) identifies the load |
| location | column 3-vector | location of load |
| body | text string | identifies which body is loaded |
| force | column 3-vector | applied force |
| moment | column 3-vector | applied moment |

### item type `actuator`

A linear actuator. Similar to the `spring` item described above, but the force is determined from an input. Allows definition of systems with forced response. Properties include the location of each end, the names of the two bodies to which it is attached, and the `gain`. Torsional actuators are defined by setting the `twist` field to 1. Can be used to simulate loads applied by defined motion of a spring, where the gain can be set equal to the stiffness of a `spring` item in parallel. As an extention, the `rategain` allows a parallel damping effect, applying a force proprtional to the rate of change of input.

**Table 1.8**: item type `actuator`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `actuator` |
| name | text string | (*optional*) identifies the actuator |
| location1 | column 3-vector | location of end1 |
| location2 | column 3-vector | location of end2 |
| body1 | text string | identifies which body is attached to end1 |
| body2 | text string | identifies which body is attached to end2 |
| gain | scalar | ratio of force to input value |
| rategain | scalar | (*optional*) ratio of force to rate of change of input value |
| twist | binary integer | specifies force (default) or moment |

### item type `sensor`

A linear sensor. Like the actuator item described above, but instead determines the outputs of the linear system. Properties include the location of each end, names of the two bodies to which it is attached, and the gain. Torsional sensors are defined by setting the `twist` field to 1. The sensor may be defined to sense velocity or acceleration rather than displacement. The reference frame may be set to measure local rather than global velocities. In addition, the sensor may be paired with an actuator, where the actuator input is added directly to the sensor output (used to generate the feedthrough matrix).

**Table 1.9**: item type `sensor`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `sensor` |
| name | text string | (*optional*)identifies the sensor |
| location1 | column 3-vector | location of end1 |
| location2 | column 3-vector | location of end2 |
| body1 | text string | identifies which body is attached to end1 |
| body2 | text string | identifies which body is attached to end2 |
| gain | scalar | ratio of output value to sensor length |
| twist | binary integer | (*optional*) specifies length (default) or twist |
| order | integer | (*optional*) set to 1 for position (default), 2 for velocity, and 3 for acceleration |
| frame | integer | (*optional*) frame of reference; 1 for global (default), 0 for local |
| actuator | text string | (*optional*) identifies the paired actuator |

### item type `nh_point`

A nonholonomic constraint. Similar in nature to the `rigid_point`, but does not prevent displacement, only velocity. Properties include the location, the names of the two bodies to which it is attached, the number of forces and moments passed, axes as necessary. Useful for specifying the behaviour of a skate or rigid wheel that does not allow lateral sliding. The `nh_point` is assumed to not carry any preloads when the system is in equilibrium.

**Table 1.10**: item type `nh_point`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `nh_point` |
| name | text string | identifies the constraint |
| location | column 3-vector | location of constraint |
| body1 | text string | identifies which body is attached to end1 |
| body2 | text string | identifies which body is attached to end2 |
| forces | scalar | number of forces transmitted (0,1,2 or 3) |
| moments | scalar | number of moments transmitted (0,1,2 or 3) |
| axis | column 3-vector | direction or normal vector |

### item type `triangle_3`

A planar spring, commonly defined in the finite element literature as a constant strain triangle (CST). It has no out of plane stiffness. Properties include the locations of the nodes, the names of the three bodies to which it is attached, the modulus of elasticity and Poisson's ratio of the material used, and the thickness. The '3' in the name refers to three elastic modes, derived from six input motions (translation in two directions at each node), less three 'rigid body' modes.

**Table 1.11**: item type `triangle_3`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `triange_3` |
| name | text string | identifies the item |
| location1 | column 3-vector | location of corner1 |
| location2 | column 3-vector | location of corner2 |
| location3 | column 3-vector | location of corner3 |
| body1 | text string | identifies which body is attached to corner1 |
| body2 | text string | identifies which body is attached to corner2 |
| body2 | text string | identifies which body is attached to corner3 |
| modulus | scalar | material modulus of elasticity |
| thickness | scalar | material thickness |
| psn_ratio | scalar | Poisson's ratio |

### item type `triangle_5`

A planar spring, commonly defined in the finite element literature as an Ellis triangle. It is similar to the constant strain triangle, with the addition of rotational degrees of freedom at each node (sometimes referred to as 'drilling' rotations). The resulting moments only depend on relative rotation of the nodes, and not absolute rotations. The '5' refers to five elastic modes, derived from nine input motions (translation in two directions, plus one rotation at each node), less three 'rigid body' modes, and one 'false' zero stiffness mode where all three nodes have identical non-zero rotations.

**Table 1.12**: item type `triangle_5`

| Fields | Data Type | Comments |
|---|---|---|
| type | text string | must be `triange_5` |
| name | text string | identifies the item |
| location1 | column 3-vector | location of corner1 |
| location2 | column 3-vector | location of corner2 |
| location3 | column 3-vector | location of corner3 |
| body1 | text string | identifies which body is attached to corner1 |
| body2 | text string | identifies which body is attached to corner2 |
| body2 | text string | identifies which body is attached to corner3 |
| modulus | scalar | material modulus of elasticity |
| thickness | scalar | material thickness |
| psn_ratio | scalar | Poisson's ratio |

## 1.5 Running EoM

Once the model definition file is complete, the analysis can be conducted by calling the `run_eom()` function, with the name of the model definition file as the argument. For example, a sample spring-mass-damper model can be executed by running:
`run_eom('input_ex_smd')`
Or one can run a sweep over a range of speeds from 0 to 10 m/s on the sample Whipple rigid-rider bicyle model by running:
`run_eom('input_ex_bicycle_rider',[0:0.2:10])`
Note the file name must be given as a string, or as a function handle. The `run_eom.m` function file is found in the top level folder folder `inst`. The result of the analysis is stored in the automatically generated output folder:
`inst/data/output/todays_date/current_time/`

The output will be a number of data files, including a LaTeX file called `report.tex`. Compiling the `report.tex` file will generate a nicely formatted analysis of the system. Alternatively, the state matrices alone can be returned directly as an argument from the `run_eom()` function. An optional text flag can be passed to modify the default behavior of EoM, to turn off report generation, or generation of modal animations (both can be somewhat time consuming and often take much longer than the actual numerical analysis). If conducted, the modal analysis will be stored in the `VRML` folder. A standard standalone VRML viewer (e.g., `freewrl` in Linux, or `view3dscene` in many platforms), can be used to animate the mode shapes.

### run_eom

```
[sys]=run_eom(func)

[sys]=run_eom(func,flag)

[sys]=run_eom(func,flag,option)
```

Create state-space model and analyse using EoM.
**Inputs**
`func`: The system to be analyzed, where `func` is either a string containing the name of the system definition function, or its function handle.
`flag`: Optional flag to control the behavior of EoM. A string variable containing `noreport` to suppress the LaTeX report, `noanimate` to suppress the generation of the modal animations, or `quiet`, for both.
`option`: Optional float or struct argument, passed along to the system definition function. (The system definition function must be defined to take this argument if it is passed.) If the option argument is a vector of floats, then `run_eom` will call the system definition function once for each entry in the vector, passing each entry in turn. In this case, animations are automatically suppressed, the the format of the report is modifed to summarize the results. Some of the example inputs are designed to accept a struct argument that can modify the system parameters directly from the `run_eom` function call.
**Outputs**
`sys`
Linear equations of motion of the system, in descriptor state-space form (A,B,C,D,E ma-

trices). Note that the descriptor form of the state space equations can represent linear differential algebraic equations (DAEs), rather than ordinary differential equations (ODEs), depending on the condition of the E matrix, but if so, they can be easily converted to an equivalent lower dimensional ODE using the `minreal()` function in Matlab®/Octave. This most commonly occurs when defining systems with massless bodies.