MBS guide

Joar Göransson Axås

June 2020

1 Introduction

This text gives an overview of how to set up and solve a multibody dynamics problem with the Python module `MBS`. All classes and functions are described in detail in the code documentation, and therefore the aim of this guide is rather to provide a general guideline and to answer common questions.

2 Code structure

The tool is imported via

```
import MBS
import MBS.constraints
import MBS.loads
import MBS.positioning
import MBS.simulation
import MBS.transformations
import MBS.visualization
```

for each respective module required for the given project. Generally, all modules are not needed, but in practice, `MBS`, `MBS.positioning`, and `MBS.simulation` are always required to solve any sensible problem.

The problem is set up by generating an object for each involved body, point, constraint, and load. The recommended order of setup is bodies – points – constraints – loads. The following is an example of such a setup.

```
import numpy
pendulum = MBS.Body(m=2, I=numpy.eye(3), force=[0.0, 0.0,
    -2*9.8])
ground = MBS.Ground()
bodies = [pendulum]

pendulum_point = MBS.Point([-0.5, 0.0, 0.0], shaft)
ground_point = MBS.Point([0.0, 0.0, 0.0], ground)
points = [pendulum_point, ground_point]
```

```
joint = MBS.constraints.Spherical(pendulum, pendulum_point,
    ground, ground_point)
constraints = [joint]
```

Here, we chose to implement the gravity as an argument to the `pendulum` body. Alternatively, we could have added a `load` object. The code above describes the components of a pendulum attached to the origin with a ball-and-socket joint.

A function call is needed to generate the generalised coordinate vectors of all bodies:

```
s, qdot = MBS.positioning.get_state(bodies)
```

We have set up the components of the system, but we did not yet define the position and orientation of the pendulum. For the position, this can be done automatically with

```
MBS.positioning.adjust_to_constraints(s, bodies, constraints)
```

The data contained in each object are used to set up the equations of motion in each timestep. We can now simulate the system for, say, a timestep 10^{-4} s and 1000 timesteps.

```
ssaved, qdotsaved = MBS.simulation.simulate(1e-4, 1000, s,
    qdot, bodies=bodies, constraints=constraints)
```

Finally, to interpret the results, it is nice to make a little video.

```
MBS.visualization.animate(ssaved, bodies)
```

3 Classes

3.1 Bodies

These entities have 6 degrees of freedom and move. Loads and constraints are applied to them. They also have points, which however are only used for visualisation.

3.2 Ground

This is a special object. Whenever a `Body` object can be passed as an argument, so can a `Ground` object. The ground is different from a body in that it does not move, and is not visualised.

3.3 Points

`Point` objects belong to bodies or the ground, and in the case of bodies, mark the outline of the body visualisation. A point object is a vector in the local coordinate system of the entity associated with the given point in the constraint or load.

3.4 Constraints

Each body has six degrees of freedom that can be constrained: three translational and three rotational. In fact, there are only two constraint types, one `Translational` for translations and one `Slider` for rotations. The number of locked degrees of freedom and the direction of constrained motion are defined as arguments to these two classes. All other joints can be defined as some combination of a `Translational` and a `Slider` constraint. For example, a fixed constraint is a combination of fully locked `Translational` and `Slider` constraints. For simplification, some common combinations, like the `Fixed` constraint, are built in as inheriting classes. Furthermore, the `Spherical` constraint is a special case of a fully locked `Translational` constraint.

Fewer than three axes can be locked by passing a matrix of the normals of the locked directions as the `constaxes` argument to constraint classes. It is important to define the axes in the correct coordinate system, which is specified in the `coord` argument. Generally, if the joint is attached to the bodies, it should be defined in local coordinates, by passing `coord='local1'` for the first body.

A revolute joint with one degree of freedom therefore can be realised by two constraints, locking three translations and two rotations. This can be written as

```
constraints = []
constraints.append(MBS.constraints.Spherical(body1, point1,
    body2, point2))
constraints.append(MBS.constraints.Slider(body1, point1, body2
    , point2, [[1,0,0], [0,0,1]]))
```

The two points of the bodies now coincide, and the bodies will have the same rotations about the x and z axis in the local coordinates of `body1`. This is equivalent to a revolute joint, with free rotation about the y axis in the coordinate system of `body1`.

3.5 Loads

All loads act between two entities. Calls to any of the `Load` classes therefore require two entities as arguments. To implement a gravitational force, or any other load that for the purposes of the problem only acts on one body, the second entity should be set to a `Ground` instance.

Note that a spring is not a constraint, but a load. Setting a very high stiffness however makes the spring act similar to a constraint.

4 Positioning

At the start of the simulation, all bodies must be positioned so that all constraints are fulfilled. This is a result of the fact that the constraint equations are differentiated before solution, that is, the explicit solver makes sure that

constrained points move at the same speed, but not that they coincide. The solver has some limited functionality to assist the user with the positioning.

In addition, all bodies must be initialised so that the time derivative of all constraint equations is also zero. Points that are to coincide must move at the same velocity at the start of the simulation. There is currently no functionality to adjust the initial velocities automatically.

Manual positioning can be achieved by passing arguments to the bodies, or by adjustment before or during simulation. The arguments are `r, p, v, omega`, referring to translation, orientation, translational velocity, and angular velocity, respectively. `p` is a quaternion, while `omega` is a vector. To facilitate manual initialisation of the orientation of bodies, there are two additional arguments `n` and `phi`, describing a rotation of ϕ rad around the axis $\boldsymbol{n}$. This automatically sets the quaternion `p` such that the body is properly rotated. (For more complicated initial rotations, one would have to calculate the appropriate `p` and set it after creating the body object, as `p` currently can't be set in the constructor.)

The automatic positioning is achieved with the function

```
MBS.positioning.adjust_to_constraints(s, bodies, const,
    posconst)
```

The automatic positioning is limited to setting the translations of n bodies so that n spherical constraints are fulfilled. This is because the function requires a system with fully constrained translations. The function adjusts translations such that all spherical constraints are fulfilled. The constraints do not need to be included in the subsequent simulation. This means that it is possible to position bodies such that, for example, springs are initially relaxed. For convenience, the constraints that are not included in the simulation and only used for positioning can be passed as a list as the `posconstraints` argument, if it is desired to keep them separated from the simulation constraints, passed as `constraints`.

The bodies are not rotated. The reason for this is that there would generally exist multiple solutions, and the Jacobian of the constraint equations would be singular. This requires a more sophisticated (iterative) solution method.

If the Jacobian is singular, it means that too many or too few degrees of freedom have been locked, or that some bodies are locked with too many constraints while others have too few. This raises an error.

In other words, the function requires that all involved bodies are connected as a non-looping chain of spherical constraints, with the number of joints equal to the number of bodies. However, all bodies and constraints in the simulation do not need to be subject to the automatic positioning.

Note also that using the `adjust_to_constraints` function overwrites the translations of all bodies as specified with `r`. Manual positioning should therefore be performed after the automatic positioning.

5 Simulation

During simulation, the equations of motion are integrated using a forward Euler scheme. The error accumulates over the integration time, and is best reduced by decreasing the timestep. While more sofisticated algorithms are available, currently, their implementation was determined to be beyond the scope of this project. Nevertheless, a `correction` option has been added to the simulation tool that reduces the constraint errors in each timestep.

Although the full state for each timestep is returned in `ssaved, qdotsaved`, the same quantities are also saved as lists of arrays in each body object as `bodyObj.rsaved`, `bodyObj.psaved`, `bodyObj.vsaved`, and `bodyObj.omegasaved`. This is handy for plotting the state of individual bodies.

6 Visualisation

A very rudimentary plotting-based tool is included to visualise the output of the simulation. The `animation` function plots the body system for given timestep intervals (`pt`). The geometry visualisation draws lines between all points belonging to each body. The mass centre of each body is drawn as a sphere. It is a good idea to add some redundant points to the bodies to better capture their shape, even though it does not influence the simulation. In addition, a zooming coefficient can be defined. Finally, a functionality that rotates the point of reference around a vector has been added. In some cases, movement of the camera makes it easier for the user to grasp the three-dimensional geometry. Regardless of what the axis labels say, the system itself is not rotated. The visualization tool is envisioned as a debugging tool. Refer to e.g. Paraview for proper rendering of the simulation results.