Multibodysim

Wietse van Dijk (2012)

contents

1 Conventions

Positive directions are depicted in Figure 1.

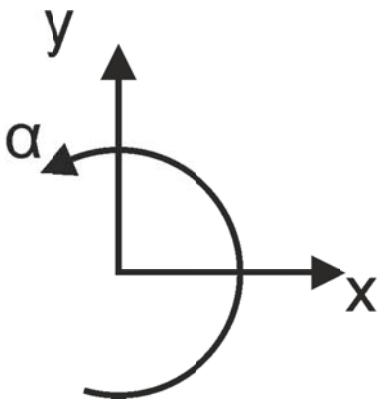

Figure 1: Positive directions

In the simulator the units from Table 1 are used.

Measure	Unit
Length	m
Mass	kg
Inertia	$kg \cdot m^2$
Force	N
Torque	N·m

Table 1: Units

2 Introduction

This is a short manual on the multibody physics simulator. The multibody simulator generates the equations of motion for a multibody system. Additionally controllers can be applied to exert external forces on the system. A integrator calculates the movements based on the equations of motion.

All code is written in MATLAB. However running MATLAB scripts is relatively slow to running compiled code. Therefore the program has the functionality to generate MEX code for the equations of motions and the controllers. It might be noticed that the code is therefore sometimes a little bit more complicated than necessary than it would be if only MATLAB script were evaluated.

The equations of motion are derived by the TMT-method [1]. To implement constraints an extension of the method is implemented. This is described in Appendix B.

3 The simulator

This function will discuss the basic functionality of the simulator. How to add parts and controllers and how to run the simulation. With the software comes the script example.m. This scripts shows you how to build some basic models. It can be useful to review this file complementary to the instructions given in this manual.

3.1 Initializing the simulator

The simulator should be initialized by running

```
initsim;
```

This adds the necessary directories to the search path

3.2 Creating a multibody system

To create a multibody system a multibody object should be created. This can be done as follows

```
mdl = multibody;
```

This creates a multibody system containing only one part, the world, to which new objects can be added. Throughout this manual we will use mdl to refer to a multibody object.

3.3 Adding parts and controllers

Parts and controllers can be added to the system. There are different kinds of parts and controllers like masses (parts), joints (parts), gravity (controller). The user is free to create new types. A part can be added to the multibody system by typing:

```
mdl.makepart(mypart);
```

Where mypart is a part object that is added to the system. All parts are member of an array (mdl.parts). Similarly a controller can be added:

```
mdl.makecontroller(mycontrol);
```

The controller are stored in the mdl.control array. A part or controller can be found by using its index in the array:

```
mdl.controller{1}
```

Or by finding it using its name

```
mdl.findpart('mypart');
mdl.findcontroll('mycontrol');
```

More details on parts and controllers can be found in the parts and the controllers section.

3.4 Simulating

After the model is build it needs to be pre-processed before the simulation can be run with:

```
mdl.preprocess
```

This command creates the necessary .m files to run the simulation. The .m files are stored in the generated directory of the working folder. The program is written with the assumption that the working folder is the folder that contains the multibody program. Changing the working folder to another folder might cause the simulator to give an error.
The time span of the simulation adjusted by setting the time span

```
mdl.t = [0 t_end]; % default [0 1]
```

The simulation can be started with

```
mdl.simulate(0, 0); % run simulation without compiled code
```

If a simulation is run multiple times it can be beneficial to compile some parts of the code. This can be done as follows:

```
mdl.makemex;
mdl.simulate(1, 0); % run simuation with compiled code
```

Simulations can be run multiple times without the need running the mdl.preprocess or mdl.makemex command again as long as no parts or controllers are added or removed. This can for example be useful to run a model using different initial conditions. To update changes the following command suffices:

```
mdl.initmatrix
```

After that the simulation can be rerun using the commands from above.

3.5 Assessing and animating the results

property	type	
simt	[1 x n] matrix	Simulation time on every timestep
simx	[m x n] matrix	Generalized coordinates positions, veleocities, and integrated controller states
simrec	[p, n]	Recorded controller states

The results can be animated with the following commands:

```
mdl.setshape; %defines a shape for all the parts
mdl.makefigure; % makes a figure with the model depected
mdl.animate; % animates the figure, and adds controls to the figure
```

3.6 Parts

Parts are objects that define the masses and the kinematics of the system. There are eg. joints, masses, constraints. An example of a system consisting out of different parts is shown in Figure 2.

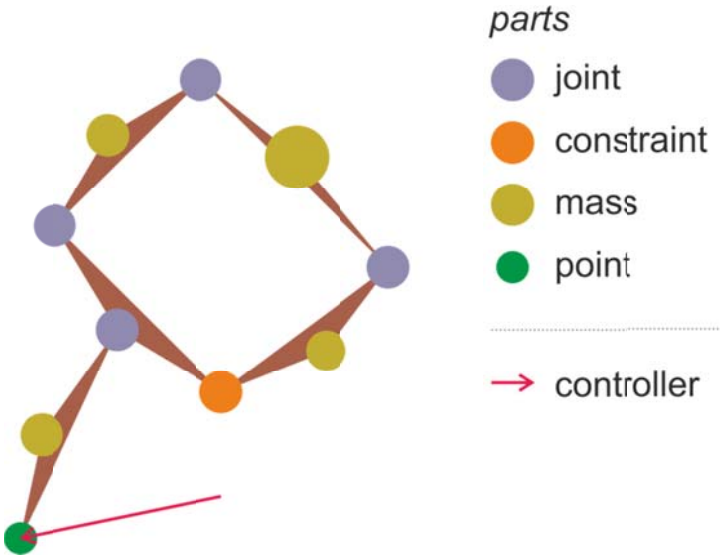

Figure 2: An example of a multibody system with different parts and a controller exerting a force on the system. The system is defined by the parts. The segments (brown) emerge from the different parts that are linked together (e.g. a joint and a mass), but do not have to be explicitly defined by the user.

The different part types have common and specific properties. The most important common properties are:

property	type	
pos	[3 x 1] matrix	The position of the part in [x, y , and α]
name	string	The name of the part. The name should be unique
parent	part	The parent part it is attached to. This should always be defined, except for the world part that does not have a parent. A parent should be joint or the world.

All the part types that are included in the simulator can be created as follows

```
p = parttype('propertyname1', val1, 'propertyname2', val2);
```

eg:

```
p = point('pos', [1; 1; 0], 'name', 'mypoint', 'parent',  pointparent)
```

Below the different parts that are currently implemented in the simulator are discussed in detail.

3.6.1 Joints

Joints define the kinematics of the system. A joint defines how the children of that joint can move with respect to the parent of that joint. In the current version two types of joints are implemented. Floating joints and rotational joints.

3.6.1.1 Floating joints

Floating joints are joints that form a floating base of a system. They have three degrees of freedom (x, y, rotation) and are thereby completely free to move.

3.6.1.2 Rotational joints

Rotational joints are joints are fixed on their parent joint and all children of this joint can rotate around the position of this joints. The rotational joint has one degree of freedom.

3.6.2 Masses

Masses represent the mass and the inertia of the system.

mass	[3 x 1] matrix	Mass of the part. Mass in the x-direction, mass in the y-direction, moment of iniertia.

3.6.3 Points

Points that are part of the multibody system do not play a role in the dynamics of the system, they do not have mass nor that have an influence on the kinematics. However they can be very helpful

for the control of the system. A point can for example which part of the system can make contact with the ground.

3.7 Constraints

A constraint limits the equations of motion of the system. Different constraints are possible, but so far only a rotational constraint is implemented.

3.8 Controllers

Controllers are elements in the simulator that have the capability to exert forces or torques on the multibody system. There are a few controllers already implemented, but it might be useful to create your own controllers. The first the controllers that are already implemented will be discussed, secondly it will be discussed how to create your own controller.

3.9 Events

The simulation is stopped either when the maximal simulation time is reached or if an event is detected. In the latter case an event function has to be defined. The event function outputs 1 if an event is detected. An event function can be defined by:

```
mdl.simevent{1} = myeventfunctions
```

4 Controllers

Different controllers are implemented in the simulation. This section discusses the general lay-out of the controllers, controllers that are implemented, and how to write your own controllers.

4.1 General properties of the controllers

Each controller has some general features. These features are discussed here.

4.1.1 Constants

Each controller can have constants. The constants can be found by using:

```
mdl.findcontrol(mycontrol).cons
```

This is a struct containing all the constants (e.g. the stiffness and damping values for a PD-controller). These constants can be changed in between of simulations. When the size of the struct is not changed creating new mex files is not necessary the run the model again.

4.1.2 (integrated) States

A controller can have additional states. These states are integrated by the integrator of the software. This state can be for example the state of a viscous-elastic property of the model. The model gets the state (s) and should return the derivative (ds) of that state.

4.1.3 Records (non-integrated states)

The record are values that are stored every time step and are not integrated. This can be for example the interaction force with the ground of a state of a state-machine. Record might come in handy when reviewing simulation results.

4.1.4 Motors

Many controllers might have an action that acts as a motor around a rotational joint. In order to make this easier a controller might have motors. By using motors the bookkeeping of calculating relative joints angles and torques is made easier.

4.2 Build in controllers

4.2.1 Gravity

This controller simulates earth gravity with an acceleration of -9.81 in the axes. To add the controller

```
mdl.addcontrol(gravity)
```

4.2.2 Contacts with the ground

This controller simulates contacts with the ground. The used contact model is derived from [2]. For this controller it is necessary which points make contact with the ground. The contact controller can be added to the model as follows:

```
c = contact;
c.points = mdl.parts([10:15 22:27]);
mdl.makecontroller(c);
```

4.2.3 Human controllers

The simulator was specifically designed to simulate human motions. Therefor some human specific controllers have been implemented. The controllers here are specifically designed to control the hip knee and ankle of both legs. The motors of these controllers refer to (hip left, knee left, ankle left, hip right, knee right, and ankle right). Angles of the segments should be zero when the human is in upright position.

4.2.3.1 Ligaments

The ligaments prevent the human to move outside the joint limits.

4.2.3.2 Reflex

This controller models the human reflex model by [3].

4.2.3.3 XPED – exotendons

This controller models the exotendons as described by [4]

4.3 Appendix A: Reviewing simulations

As described above the simulations can be used with the mdl.animate method (page 5). For evaluating of walking data an additional function is available.

```
averagestep(mdl, 'arg1', 'arg2', 'argn')
```

The arguments define the type of plots for which the following options are available:

activations	Muscle activations
muscle force	Muscle forces
joint torques	Joint torques by the individual controllers
joint torque area	Joint torques by the individual controllers as area plot
joint angles	Joint angles

4.4 Appendix B: Constraints in the TMT-method

It can sometimes be difficult to find a set of generalized coordinates that describe a multibody system. This can for example be the case is closed links appear in the structure. A possible way to solve this problem is to add constraints. The following paragraphs with an example how the constraints are implemented in the multibody simulator as an extension to the TMT method.

4.4.1 Example

Throughout this section we will use the example from Figure 3. The figure shows a two link system masses m_1 and m_2, link lengths l1 and l2, and the angles of the links are given by q_1 and q_2.

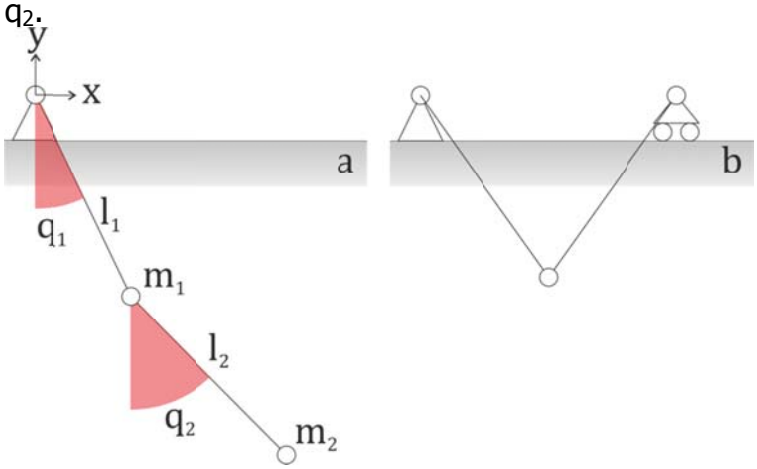

Figure 3: this example shows the a two link system. In a the system is unconstrained, in b the system is constrained.

With the original TMT-method we can derive the equations of motion for the system shown in a.

$$T(q)=\begin{bmatrix} l_1\sin(q_1) \\ -l_1\cos(q_1) \\ l_2\sin(q_2) \\ -l_2\cos(q_2) \end{bmatrix}, M=\begin{bmatrix} m_1 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 \\ 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & m_2 \end{bmatrix} \tag{1.1}$$

In our case we assume the forces acting on the system are gravitational forces:

$$F=\begin{bmatrix} 0 \\ -m_1 g \\ 0 \\ -m_2 g \end{bmatrix} \tag{1.2}$$

And

$$\ddot{q}=\left(\mathbf{T}^T\mathbf{M}\mathbf{T}\right)\mathbf{T}^T\left(\mathbf{F}-\mathbf{M}\mathbf{S}\dot{q}\dot{q}\right) \tag{1.3}$$

In figure b a constraint is added. The y-position of the second mass is always zero. One way of solving this problem is reduce the number of generalized coordinates and express everything a function of q_1. In this case we solve the problem by adding a constraint equation.

$$\begin{aligned} D_j(q_k)&=0 \\ -l_1\cos(q_1)-l_2\cos(q_2)&=0 \end{aligned} \tag{1.4}$$

4.4.2 Velocities

We take the derivative of this equation:

$$D_{j,k}(q)\dot{q}_k = 0$$

$$\begin{bmatrix} l_1 \sin(q_1) & l_2 \sin(q_2) \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} = 0 \tag{1.5}$$

We can split the generalized coordinates into independent (q_i) and dependent (q_d) coordinates. The number of dependent coordinates should be equal to the number of constraints. In our example we choose q_1 to be the independent coordinate and q_2 to be the dependent coordinate. Equation 1.2 can be rewritten as:

$$\begin{bmatrix} D_{j,i}(q_i) & D_{j,d}(q_d) \end{bmatrix} \begin{bmatrix} \dot{q}_i \\ \dot{q}_d \end{bmatrix} = 0 \tag{1.6}$$

Or:

$$D_{j,i}(q_i)\dot{q}_i + D_{j,d}(q_d)\dot{q}_d = 0 \tag{1.7}$$

And thereby find an expression for the dependent velocities:

$$\begin{aligned} \dot{q}_d &= -D_{j,d}^{-1}(q_d) D_{j,i}(q_d)\dot{q}_i \\ \dot{q}_2 &= \frac{-1}{l_2 \sin(q_2)} \cdot l_1 \sin(q_1)\dot{q}_1 \end{aligned} \tag{1.8}$$

4.4.3 Accelerations

We differentiate equation 1.4:

$$\begin{aligned} &D_{j,ik}\dot{q}_i\dot{q}_k + D_{j,i}\ddot{q}_i + D_{j,de}\dot{q}_d\dot{q}_e + D_{j,d}\ddot{q}_d = 0 \\ &l_1 \cos(q_1)\dot{q}_1\dot{q}_1 + l_1 \sin(q_1)\ddot{q}_1 + l_2 \cos(q_2)\dot{q}_2\dot{q}_2 + l_2 \sin(q_2)\ddot{q}_2 = 0 \end{aligned} \tag{1.9}$$

Rewriting

$$\begin{aligned} \ddot{q}_d &= -D_{j,d}^{-1}\left(D_{j,ik}\dot{q}_i\dot{q}_k + D_{j,i}\ddot{q}_i + D_{j,de}\dot{q}_d\dot{q}_e\right) \\ \ddot{q}_2 &= \frac{-1}{l_2 \sin(q_2)}\left(l_1 \cos(q_1)\dot{q}_1\dot{q}_1 + l_1 \sin(q_1)\ddot{q}_1 + l_2 \cos(q_2)\dot{q}_2\dot{q}_2\right) \end{aligned} \tag{1.10}$$

Integration in the TMT-method

To make the integration in the TMT-method easier we define the following matrices

$$\begin{aligned} \mathbf{C} &= \begin{bmatrix} I \\ -D_{j,d}^{-1}(q_d) D_{j,i} \end{bmatrix} \\ \mathbf{E} &= \begin{bmatrix} 0 \\ -D_{j,d}^{-1}(q_d) D_{j,mn} \end{bmatrix} \end{aligned} \tag{1.11}$$

With

$$q = \begin{bmatrix} q_i \\ q_d \end{bmatrix} \tag{1.12}$$

Integration in the TMT-method:

$$\ddot{q} = \left(\mathbf{C}^T\mathbf{T}^T\mathbf{M}\mathbf{T}\mathbf{C}\right)^{-1}\mathbf{C}^T\mathbf{T}^T\left(\mathbf{F} - \left(\mathbf{M}\left(\mathbf{S}\dot{q}\dot{q} + \mathbf{T}\mathbf{D}\dot{q}\dot{q}\right)\right)\right) \tag{1.13}$$

4.5 Literature

1 van der Linde, R.Q., and Schwab, A.L.: 'Multibody Dynamics B', in Editor (Ed.)^(Eds.): 'Book Multibody Dynamics B' (Delft University of Technology, 2002, edn.), pp. 1-30

2 Ackermann, M., and Van den Bogert, A.J.: 'Optimality principles for model-based prediction of human gait', Journal of Biomechanics

3 Geyer, H., and Herr, H.: 'A muscle-reflex model that encodes principles of legged mechanics produces human walking dynamics and muscle activities', Neural Systems and Rehabilitation Engineering, IEEE Transactions on, 2010, 18, (3), pp. 263-273

4 van den Bogert, A.J.: 'Exotendons for assistance of human locomotion', BioMedical Engineering OnLine, 2003, 2, (1), pp. 17