MatODE

Wouter Caarls and Erik Schuitema
{w.caarls, e.schuitema}@tudelft.nl

December 21, 2011

Contents

1 Introduction

1.1 What is ODE?

ODE is the Open Dynamics Engine by Russell Smith. While ODE is slower than direct model simulation for most of the toy problems you will be working on, the advantage is that it can be directly applied to the more complex systems you will encounter in your future projects. From the ODE website at `http://www.ode.org/`:

> ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools.

An ODE model consists of *bodies* (which have a certain mass and inertia), *geometries* (defining the shape of a body for collisions) and *joints* (connecting the bodies). A *motor* can be defined on a joint to apply a certain force to it. Different joint types – such as hinges or sliders – are available, providing all the tools to model many different robots.

ODE works by fixed-timestep numerical integration of the equations of motion that define the system of connected bodies. When bodies collide, it creates temporary "joints" at the collision point that make sure the bodies do not intersect. These joints constitute a spring-damper system with finite K and D. Both the fixed step size (causing collision to be detected some time *after* they've taken place) and finite K/D values (meaning bodies can "sink into" one another a tiny amount) lead to inaccuracies. With well-chosen values for these parameters, ODE can be remarkably accurate.

1.2 What is matode ?

`matode` is a Matlab interface to ODE. It uses Matlab's object oriented programming capabilities to make the interface as easy as possible. The main class is `odesim`, which provides global interaction with the simulator such as initialization, running a simulation step, and resetting to an initial condition. It also allows you to retrieve sensor and actuator indices, with which you can sense joint positions and drive motors. The bodies that constitute the robot and the joints that connect them are defined in an `.xml` file loaded during initialization.

1.3 Installation

1. Unzip the installation file.
2. *(Linux only)* Add the `matode/toolbox` directory to your `LD_LIBRARY_PATH`.
3. Add the `matode/toolbox` directory to your Matlab path.

2 Configuration file syntax

The configuration XML file describes all the bodies, joints and motors in the system, as well as collision information and settings for the integrator. We will now briefly explain the format of this file.

2.1 `constants`

The first important section is called `constants`. As the name implies, constants are defined here that can be used in the rest of the file. As with other numerical values, the constants may contain simple mathematical expressions. Constant expressions may also reference other constants, as long as those are defined earlier in the file (see Figure 1).

```
<configuration>
      <constants>
            <length>0.2</length>
            <radius>0.015</radius>
            <density>7874</density>
            <mass>length*_pi*radius^2*density</mass>
      </constants>
      ...
</configuration>
```

Figure 1: Using mathematical expressions in the `constants` section, referencing previously defined constants.

2.2 `ode`

The `ode` section defines the simulation environment, and takes up the rest of the file. It contains global settings, such as the `globalK` and `globalD` constants for the spring-damper physics simulation used by ODE, as well as the gravity vector `gravityZ` (and possibly `gravityX` and `gravityY`, if you're so inclined). Most important for now are the duration of a simulation step, `steptime`, and how many substeps are done for each step, `subsamplingfactor`. In general, the step time depends on the bandwidth requirements of your controller, while the subsampling factor should be determined by the expected speeds and forces in the simulation.

The `ode` section also defines the objects in the simulation, and collision information (material properties, and which objects may collide). As the system that you will be modeling does not include collisions, we will focus on defining objects.

2.3 `object`

An object is a collection of rigid bodies connected by joints. An object has a `name` and zero or more `initialconditions`, which determine the body orientations when the simulation is reset. As such, the `initialcondition` defines the `bodyname` and a desired `rotation` of that body (see Figure 2). The object may also define `geometries` associated with its bodies, which are used in collision detection.

Bodies represent the moving masses in an object. A `body` has a `name`, `mass` and a moment of inertia defined by `IXX`, `IYY` and `IZZ` (and possibly other combinations, depending on the shape of the body you're trying to simulate). It can also define how it should be drawn, using the `drawinfo` tag (see Figure 3). Note, though, that what is drawn may be completely unrelated to the moment of inertia, or even the collision geometry!

When drawing, the coordinate system is centered on the body. Any objects you draw (such as the cylinder in figure 3) are also placed in the center, unless you specifically move it with `x`, `y`, `z` position tags. The positive Z axis points

```
<object>
      ...
      <initialcondition>
            <bodyname>pole</bodyname>
            <rotation>
                  <axis>
                        <x>1</x>
                        <y>0</y>
                        <z>0</z>
                  </axis>
                  <angle>_pi</angle>
            </rotation>
      </initialcondition>
</object>
```

Figure 2: An initial condition for the `pole` body, rotating it 180 degrees around the X axis.

upwards.

```
<body>
      ...
      <drawinfo>
            <cylinder comment="pole">
                  <radius>radius</radius>
                  <length>length</length>
            </cylinder>
      </drawinfo>
</body>
```

Figure 3: Using a cylinder as the graphical representation of a body. `radius` and `length` are constants.

You can define an `anchor` on a body (and also in the plain `ode` section, in which case it is an anchor fixed in the world) in order to connect the body with others using `joints`. The anchor defines a point (`x`, `y`, `z`) on the body at which the joint is fixed (see Figure 4). The coordinate system is the same as that for drawing, so the anchor in the figure is located at the bottom of the body. Because you can have multiple anchors per body, it also has a `name`.

A `joint` connects two anchors (`anchor1` and `anchor2`) and can be of different `types` (such as a hinge, slider, universal, etc.). Depending on the type it can have a number of properties, such as the axis along which the movement can occur (see Figure 5).

A joint can also have a `motor` associated with it, which can be actuated to provide a certain force or torque (see Figure 6).

```
<body>
      ...
      <anchor>
            <name>world</name>
            <x>0</x>
            <y>0</y>
            <z>-length/2+radius</z>
      </anchor>
</body>
```

Figure 4: Defining an anchor.

```
<joint>
      <name>joint</name>
      <type>hinge</type>
      <anchor1>
            <bodyname>world</bodyname>
            <anchorname>pole</anchorname>
      </anchor1>
      <anchor2>
            <bodyname>pole</bodyname>
            <anchorname>world</anchorname>
      </anchor2>
      <axisX>1</axisX>
      <axisY>0</axisY>
      <axisZ>0</axisZ>
      ...
</joint>
```

Figure 5: A hinge joint definition.

```
<joint>
      ...
      <motor>
            <type>torque</type>
      </motor>
</joint>
```

Figure 6: A torque-controlled hinge joint motor.

3 Matlab interface

A typical script involving `matode` looks as follows:

```
sim = odesim('mountaincar.xml');                % Load configuration
vel = sim.sensor('robot.base.velocity.y');      % Define sensor
motor = sim.actuator('robot.motorjoint1.torque');% Define actuator
actuators = sim.actuate();                      % Get actuation vector
for t = 0:sim.step():6                          % Simulation loop (6s)
```

```
  sensors = sim.sense();                          % Measure sensor values
  if sensors(vel) > 0                             % Read sensor
    actuators(motor) = 0.5;                       % Set actuator
  else
    actuators(motor) = -0.5;
  end
  sim.actuate(actuators);                         % Run simulation step
  pause(sim.step());                              % Run in real-time
end
sim.close()                                       % Destroy simulation
```

Note how the sensor has been defined: it gets the absolute `velocity` in the `y` direction of the `base` body in the `robot` object.

This simulation uses a simple controller to drive a car up a hill. While the car doesn't have enough torque to accomplish this task immediately, it can use another hill to gain speed. This is called the mountain car task.

3.1 Help page

```
ODESIM Open Dynamics Engine interface

   OBJ = ODESIM(FILE) returns an ODE simulator object.
   OBJ = ODESIM(..., 'nodialog') suppresses the OpenGL dialog window.

   Members:
      OBJ.SENSOR Get sensor index.
         S = OBJ.SENSOR(PATH) returns the sensor index for reading the
         value of PATH. PATH is of the form
            <object>.<joint>.<angle|anglerate|position|positionrate>
            which reads the angle or angular velocity in case of a hinge
            joint, or position and velocity in case of a slider.
         or
            <object>.<body>.<position|velocity>.<x|y|z>
            which reads the absolute position or velocity of a body in
            a certain direction
         or
            <object>.<body>.orientation.<x|y|z|w>
            which an element of the orientation quaternion of a body
         or
            <obj>.<body>.<x|y|z|azimuth|elevation|distance>@<obj2>.<body2>
            which reads the relative position, angle or distance of a
            body in the coordinate frame of body2.

      OBJ.ACTUATOR Get actuator index.
         A = OBJ.ACTUATOR(PATH) returns the actuator index for setting the
         value of PATH. PATH is of the form
         <object>.<joint>.<torque|force>
         Torque is used for hinge joints, force for slider joints.

      OBJ.SENSE Retrieve sensor data.
```

```
      V = OBJ.SENSE() Waits for the simulator to finish the last
      step and returns the sensor values.

   OBJ.ACTUATE Dispatch actuator values.
      V = OBJ.ACTUATE() returns a vector for setting actuator values.
      OBJ.ACTUATE(V) writes the actuator values V to the simulator
      and initiates a simulation step.

   OBJ.RESET Reset simulation
      OBJ.RESET() resets the simulation to the initial condition.
      OBJ.RESET(SEED) sets the random seed on which the initial
      condition is based to SEED.

   OBJ.STEP Simulator step time.
      S = OBJ.STEP is the simulated time between successive steps.

   OBJ.CLOSE Close dialog.
      OBJ.CLOSE() closes the OpenGL dialog and destroys the simulation.

Example:
   sim = odesim('mountaincar.xml');
   vel = sim.sensor('robot.base.velocity.y');
   motor = sim.actuator('robot.motorjoint1.torque');
   actuators = sim.actuate();
   for i = 1:sim.step():6
     sensors = sim.sense();
     if sensors(vel) > 0
       actuators(motor) = 0.5;
     else
       actuators(motor) = -0.5;
     end
     sim.actuate(actuators);
     pause(sim.step());
   end
   sim.close();

Authors:
  Wouter Caarls <w.caarls@tudelft.nl>
  Erik Schuitema <e.schuitema@tudelft.nl>
```

4 Questions and answers

Why is the first set of sensor data incorrect?

After defining a sensor, you must first perform a simulation step by calling `actuate` before the proper data for that sensor is available. Because no sensors were defined before the first step, all sensor values are 0 at that point. Note that it is still important to read the first sensor data, because otherwise the

actuation would run a step behind.

The same holds for `reset`; a simulation step must first take place before the new state can be read. `sim.actuate(sim.actuate());` is a reasonable way of performing such a step.

Can I set joint angles and body positions directly from Matlab?

That feature is not available at this time. To set positions and angles, use the `fixedpoint` and `initialcondition` clauses in the configuration file.