RPI-MATLAB-Simulator
User Guide

Jedediyah Williams
Ying Lu
Jeffrey Trinkle

November 25, 2013

Contents

Chapter 1

Introduction

RPI-MATLAB-Simulator (RPIsim) is an open source simulation framework for research and education in multibody dynamics available from rpi-matlab-simulator. RPIsim is designed and organized to be extended. Its modular design allows users to edit or add new components without worrying about extra implementation details. RPIsim has two main goals:

1. Provide an intuitive and easily extendable platform for research and education in multibody dynamics

2. Maintain an evolving code base of useful algorithms and analysis tools for multibody dynamics problems.

Throughout the document you will see multiple examples of different types of code. Where this code starts with $, it is a Linux bash terminal. Where the code starts with >>, it is the MATLAB (or Octave) command line inside the program. Finally, where the code is line-numbered and has no prompt, it is a MATLAB script.

Chapter 2 discusses how to setup and get going with RPIsim. Chapter 3 demonstrates standard methods for generating a simulation scene and interacting with the simulation. Chapter 4 discusses details of the workings of RPIsim. Chapter 5 demonstrates how to write custom modules for extending the functionality of RPIsim. Chapter 6 covers how to integrate Bullet collision detection through a MATLAB MEX interface. Chapter 7 walks through an example of using RPIsim to both kinematically and dynamically simulate a robotic arm.

Thanks to all those who have helped during the development of the RPI-MATLAB-Simulator, and to those who have been patient enough to use it in its early stages. In particular, thanks to Hans Vorsteveld, Michael Andersen, Daniel Montrallo Flickinger, and Rohinish Gupta.

Chapter 2

Installation and quick start

2.1 Installation

2.1.1 MATLAB

MATLAB is a popular numerical computing environment developed by MathWorks and available for purchase from http://www.mathworks.com/products/matlab/. Follow MATLAB's installation instructions for your environment.

2.1.2 GNU Octave

GNU Octave is a free MATLAB alternative that is available for Linux and Windows systems. Windows users must use Cygwin to run Octave, which is available from http://www.cygwin.com/. Octave is available from http://www.gnu.org/software/octave/download.html, but on many popular Linux systems, Octave can be quickly installed by running

```
$ sudo atp—get install octave3.2
```

Octave is run from the command line by simply typing *octave* (on Windows systems, this is done in Cygwin).

2.1.3 RPIsim

RPI-MATLAB-Simulator is available online at
http://code.google.com/p/rpi-matlab-simulator/. In a Linux environment, the RPIsim source code can be downloaded using subversion by running

```
 $ svn checkout
http://rpi—matlab—simulator.googlecode.com/svn/simulator
     rpi—matlab—simulator
```

If you do not have subversion, you will have to install it. For Windows, we suggest TortoiseSVN, available from http://tortoisesvn.net/. For Linux systems, simply run

```
$ sudo apt-get install subversion
```

2.2 Getting started

For the remainder of the document, we will discuss using RPIsim in the context of MATLAB, however everything should be analogously possible in Octave.

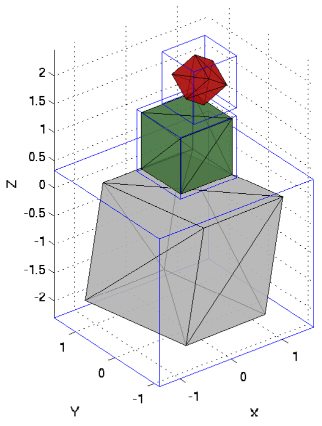
Once you have a system setup and have downloaded RPIsim, open MATLAB and change directories to RPIsim and run *setupRPIsim.m*:

```
>> cd ~/rpi-matlab-simulator/
>> setupRPIsim
```

setupRPIsim.m simply adds all of the necessary directories to the path. Now run a test script such as

```
>> simtest
```

You should be presented with the graphical representation of a small sample scene that will run automatically and look similar to the following image. The



blue wire-frame cubes are axis-aligned bounding boxes and will turn red when the corresponding body has active contacts with another body (visualization of bounding boxes is not on by default). The largest cube is static, and will not move. The other two bodies will fall with gravity and slide off of the static cube. You should see contacts plotted in green as they are identified between bodies.

Chapter 3

Creating & running scenes

In this chapter we introduce the basics of writing a simulation script and running it. We cover several of the common options in 3.1, then we will learn by example in 3.2.

3.1 Simulator options

There are several options when running a simulation including whether to use the GUI, whether to show contacts or bounding boxes, what step size to use, whether to record the simulation, etc. Many of these options are exemplified in section 3.2, but we will describe several of the options in Table 3.1 for reference. You can find all of these fields in *Simulation.m*. When used below, *sim* represents a *Simulator* struct.

3.2 Examples

3.2.1 Hello world

```
sim = Simulator();                          % Initialize the simulator
sim = sim_addBody( sim, mesh_cube() );      % Add a cube to the scene
sim = sim_run( sim );                       % Run the simulator
```

Script 1. Hello world.

Script 1 depicts a minimal simulation with only a single cube at the origin. The first line initializes a simulator struct, the second line adds a default cube to the simulator, and the third line runs the simulator.

3.2.2 A rolling cart

Let's create a rectangular chassis and put four wheels on it!

```
1    sim = Simulator(.01);
2    sim.H_dynamics = @mLCPdynamics;
3    sim.drawContacts = true;
4    sim.drawJoints = true;
5
6    % Ground
7    ground = Body_plane([0; 0; 0],[0; .1; 1]);
8        ground.color = [.7 .5 .5];
9        ground.mu = 1;
10
11   % Chassis
12   chassis = mesh_rectangularBlock(1,3,0.25);
13       chassis.u = [0; 0; 1];
14       chassis.color = [.3 .6 .5];
15
16   % Wheels
17   wheel = mesh_cylinder(20,1,0.25,0.2);
18   wheel.quat = qt([0 1 0],pi/2);
19   w1 = wheel;
20   w2 = wheel;                % Copy wheel struct to four wheels
21   w3 = wheel;
22   w4 = wheel;
23   w1.u = [ .5;  1.3; .75];    % Position wheels around chassis
24   w2.u = [-.5;  1.3; .75];
25   w3.u = [ .5; -1.3; .75];
26   w4.u = [-.5; -1.3; .75];
27
28   % Add bodies to simulator
29   sim = sim_addBody(sim, [ground chassis w1 w2 w3 w4]);
30
31   % Create joints
32   sim = sim_addJoint( sim, 2, 3, w1.u, [1;0;0], 'revolute');
33   sim = sim_addJoint( sim, 2, 4, w2.u, [1;0;0], 'revolute');
34   sim = sim_addJoint( sim, 2, 5, w3.u, [1;0;0], 'revolute');
35   sim = sim_addJoint( sim, 2, 6, w4.u, [1;0;0], 'revolute');
36
37   % Run the simulator
38   sim = sim_run( sim );
```

Script 2. A rolling cart.

In lines 1-4, we initialize the simulator and set some properties for run-time. Then we create a ground plane (7-10), chassis (12-14), and four wheels (17-26). Notice that when we created the four wheels, we re-used a generic “wheel” struct that had the rotation and body properties we wanted to share between all four wheels. The one attribute we needed to change for each wheel was its starting position u (23-26). Line 29 adds the bodies to the simulator. Lines 32-35 create the four revolute joints between the wheels and chassis. Unfortunately at the time of this writing, there is no “nice” way to reference the bodies when creating a joint, so we must use the bodyID attribute. For example, line 32 specifies a joint between body 2 (the chassis) and body 3 (w1), where body 1 was the ground plane. This requires the user to be able to keep track of what order bodies were added.

3.2.3 Cascading polyhedra

3.3 Recording and replaying simulations

3.3.1 Basic record and playback

Recording a simulation is done by setting the *Simulation* attribute *record* to true as on line 2 of the script below. To improve performance, you may turn off the GUI during simulation. MATLAB graphics tend to be quite slow, often taking up about half the time of simulation!

```
1    sim = Simulator();      % Create a simulator
2    sim.record = true;      % Turn on recording
3    sim.MAX_STEP = 500;
4    sim.draw = false;
5    ...
6    sim = sim_run( sim );   % Run the simulator
```

Script 3. Turning on recording

Each time a simulator is run with record enabled, a directory is created with a name formatted as "*sim_data_*"[*year*]_[*month*]_[*day*]_[*hour*]_[*minute*]_[*second*]. This looks like a long name, but it will help identify simulation data. For example, *sim_data*_13_10_14_18_22_59 was created at 6:22:59 pm on October 10, 2013.

Playing a recorded simulation is done by running *sim_replay*([directory]), for example

```
>> sim_replay( 'sim_data_13_10_14_18_22_59' )
```

Figure 3.1 depicts the GUI with which the user is presented when re-animating a simulation using *sim_replay()*. The slider allows the user to browse frames of the simulation, and the "Play" button animates over all frames.

3.3.2 Simulation Data Standard

3.4 Plotting while simulating

There are two fields in the Simulator struct that make interacting with data during simulation straightforward: *userFunction* and *userData*. *userFunction* allows the user to to specify a function to be evaluated at the end of every time step. *userData* is a struct where values generated in the user function can be stored. The following sections exemplify how to use these two variables.

3.4.1 Position, velocity, acceleration

Script 4 demonstrates a user function for plotting the vertical position, velocity, and acceleration of a body. Script 5 is an example simulation that utilizes Script 4. Notice that Script 4 takes *sim* as an argument and also returns *sim*. Important things to remember when writing your userFunction:

- userFunction should take *sim* as an argument and return *sim*.
- All variables that you wish to store go in *sim.userData*.
- When plotting, you may have to check if it is the simulation's first step in order to initialize things.
- You don't just have to use custom functions for plotting!

```
function sim = plotPVA( sim )
 body = sim.bodies(2);   % The body we'll plot P, V, and A for.

 P = body.u(3);              % Verticle position
 V = body.nu(3);             % Verticle velocity
 A = body.Fext(3)/body.mass; % Verticle acceleration

 if sim.step == 1
   % On the first time step, the plot doesn't exist yet,
   % so we'll create it.
   figure();
   sim.userData.P = plot(sim.time, P, 'r');   hold on;
   sim.userData.V = plot(sim.time, V, 'g');
   sim.userData.A = plot(sim.time, A, 'b');
   xlabel('Time (s)');
   legend('Position','Velocity', 'Acceleration',2);
 else
   % After the first time step the plots exist,
   % so we'll just update their X and Y data.
   set(sim.userData.P,'xdata',[get(sim.userData.P,'xdata') sim.time]);
   set(sim.userData.P,'ydata',[get(sim.userData.P,'ydata') P]);

   set(sim.userData.V,'xdata',[get(sim.userData.V,'xdata') sim.time]);
   set(sim.userData.V,'ydata',[get(sim.userData.V,'ydata') V]);

   set(sim.userData.A,'xdata',[get(sim.userData.A,'xdata') sim.time]);
   set(sim.userData.A,'ydata',[get(sim.userData.A,'ydata') A]);
 end
end
```

Script 4. A user function for plotting position, velocity, and acceleration.

```
function sim = hangingRod()
  % Initialize simulator
  sim = Simulator();
  sim.H_dynamics = @mLCPdynamics;
  sim.userFunction = @plotPVA;

  % Create an invisible static body
  staticBody = mesh_cylinder(7,1,0.2,1);
      staticBody.dynamic = false;
      staticBody.visible = false;

  % Create a hanging rod
  angle = pi/5;
  hangingBody = mesh_cylinder(7,1,0.1,0.5);
      hangingBody.u = [-0.25*sin(angle); 0; 0.25-0.25*cos(angle)];
      hangingBody.quat = qt([0;1;0], angle);

  % Gather simulation bodies and add to simulator
  bodies = [staticBody hangingBody];
  sim = sim_addBody( sim, bodies );

  % Create joint
  sim = sim_addJoint(sim, 1, 2, [0;0;0.25], [0;1;0], 'spherical');

  % Run simulation
  sim = sim_run( sim );
end
```

Script 5. A simulation that uses plotPVA().

Running hangingRod will generate two plots: the simulation (Figure 3.2a) and the plot generated by the user function (Figure 3.2b).

3.4.2 Joint Error

Let's take a look at joint error for a ranging rod pendulum. The simulation is depicted in Figure 3.2a. We can plot the joint error during simulation with the following function:

```
function sim = plotJointError( sim )
    [C,~] = joint_constraintError(sim,1);  % Error in FIRST joint
    if sim.step == 1
       figure(2);
       sim.data.error = plot(sim.time,norm(C));
       xlabel('Time (s)');
       ylabel('norm(Joint Error)');
    else
       set(sim.data.error,'xdata', ...
            [get(sim.data.error,'xdata') sim.time]);
       set(sim.data.error,'ydata', ...
            [get(sim.data.error,'ydata') norm(C)]);
    end
end
```

Script 6. Function for plotting joint error.

To tell the simulator to call this function during simulation, we add the single line

```
sim.userFunction = @plotJointError;
```

to our simulation script. Figure 3.3 depicts the magnitude of the joint error for hangingRod.m over five seconds of simulation.

Table 3.1: Simulator options (See *Simulator.m* for the complete list).

Option syntax	**Default value**	**Description**
$sim.draw = [BOOL]$	*true*	Sets whether to graphically display the simulation as it runs.
$sim.drawBoundingBoxes = [BOOL]$	*false*	Sets whether to graphically display bounding boxes of bodies.
$sim.drawContacts = [BOOL]$	*false*	Sets whether to graphically display contacts as they occur during simulation.
$sim.drawJoints = [BOOL]$	*false*	Sets whether to graphically display joints.
$sim.gravity = [BOOL]$	*true*	Turns gravity on (true) and off (false). This must be done before calling *sim_run()*.
$sim.gravityVector = [DOUBLE_{3\times1}]$	$\begin{bmatrix} 0 \\ 0 \\ -9.81 \end{bmatrix}$	A 3×1 vector that defines the magnitude and direction of the gravitational force.
$sim.H\_collision\_detection = @[function]$	*collision_detection*	The function handle that points to the function which performs collision detection.
$sim.H\_dynamics = @[function]$	$mLCPdynamics$	The function handle that points to the function that will formulate the time-stepping subproblem at each simulation time step.
$sim.num\_fricdirs = [INT]$	7	The number of friction directions to use when linearizing the friction cone during the dynamics formulation stage.

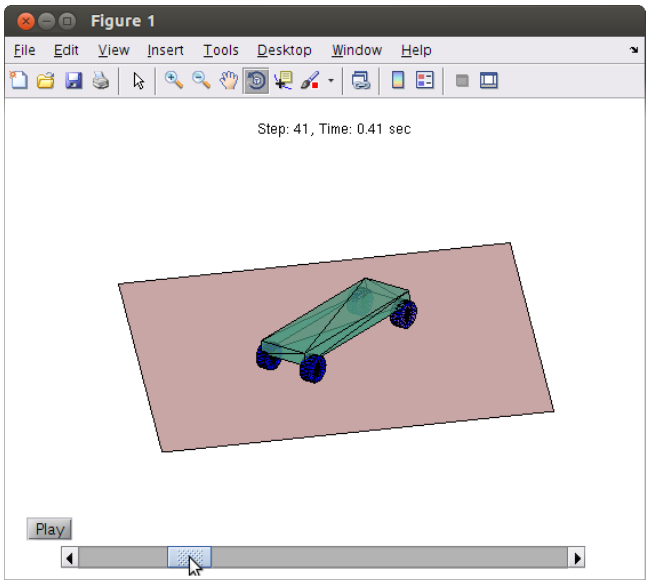
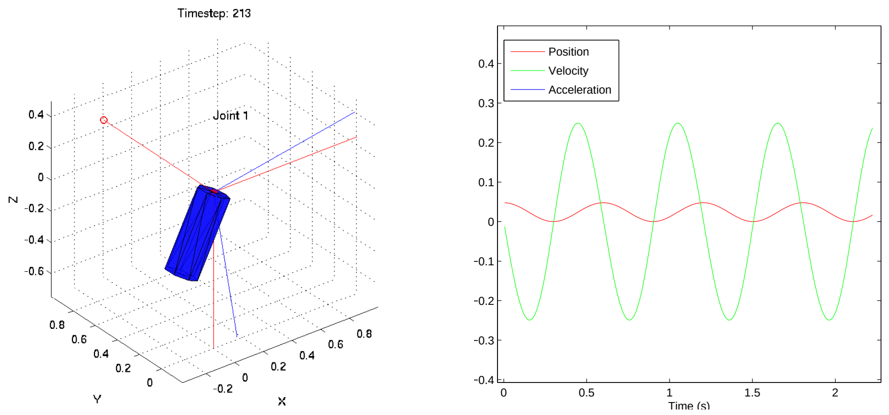


Figure 3.1: The *sim_replay()* GUI. The “Play” button will animate the simulation, or dragging the slider will set the simulation time manually.



(a) hangingRod(), a simulation that calls the userFunction plotPVA.

(b) Position and velocity (acceleration is constant -9.81) in vertical direction.

Figure 3.2: A hanging rod simulation with userFunction plotting.

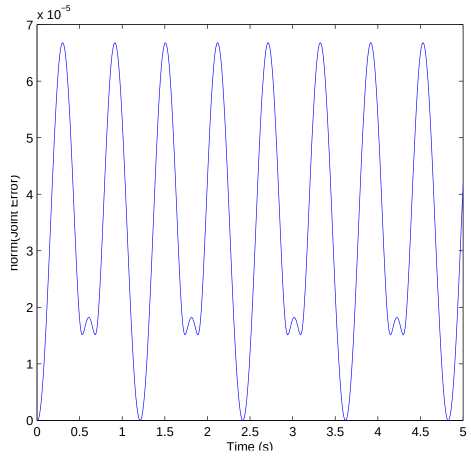


Figure 3.3: Error in joint position for the hanging rod pendulum.

Chapter 4

The default simulator

This chapter discusses the workings of RPIsim "out of the box" and goes into greater detail about what functionality is already implemented.

The file structure of the simulator is depicted in Figure 4.1. This structure is meant to be intuitive and help guide the user when editing or adding new components. The "examples" directory contains several examples of scenes which demonstrate how to specify options, and run a simulation. The simulator itself is entirely contained in the "engine" directory. The directories found there are fairly self-explanatory. The "dynamics" directory contains the functions defining each available time-stepping formulation, all of which construct a time-stepping subproblem formulated as a complementarity problem (CP). The "solvers" directory contains various functions for solving the complementarity problem: linear complementarity problem (LCP), mixed linear complementarity problem (mLCP), and nonlinear complementarity problem (NCP). See [BETC12] for a comprehensive review of these topics.

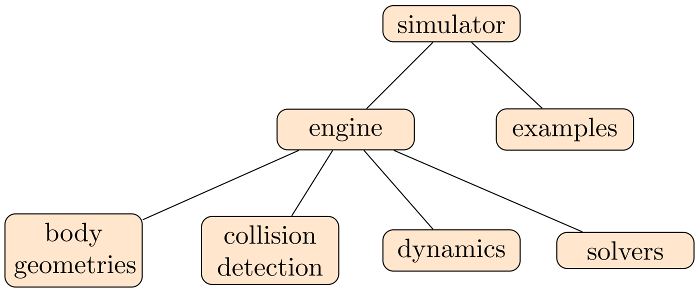


Figure 4.1: RPIsim file structure. Simulator code is organized in order to reflect the stages of simulation and to give the user intuition about the connectedness of these stages. The existing code serves as templates for extending the simulator with custom modules.

The flow of simulation is depicted in Figure 4.2. As soon as a simulation script is executed, the scene is rendered so that it may be inspected. When run, the simulator proceeds through the various stages of the simulation loop.

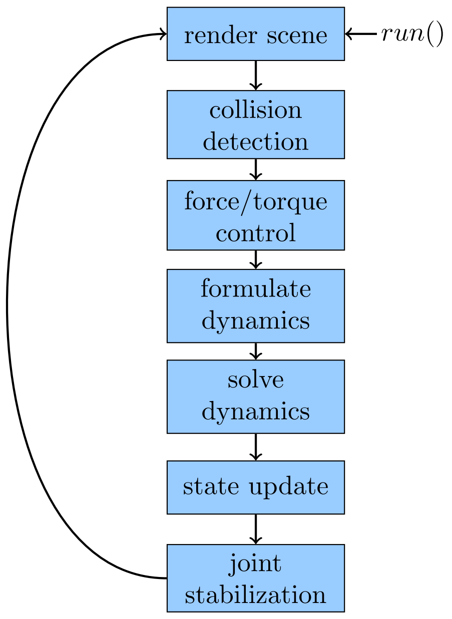


Figure 4.2: Simulation loop. At each stage, a user can replace or plug in custom modules.

4.1 Body representations

In previous versions of RPIsim, each body type i.e. cubes, spheres, etc., was an instance of a corresponding MATLAB object. This has changed in the current version for multiple reasons (in part because Octave does not use objects in the MATLAB sense), and now all simulation bodies are represented by the *Body* struct. All common body attributes such as position, rotation, mass, inertia, etc. are contained in Body, but because all bodies are represented by this one struct, it also contains type-specific attributes such as radius for spheres, point and normal for planes, etc. See *Body.m* for more details.

4.1.1 Triangle meshes

All mesh bodies are represented as triangle meshes. There are several meshes already available including mesh_cube, mesh_cylinder, mesh_rectangular_block, and the five Platonic solids.

New meshes can be created by either generating the vertex and face data, or using $mesh_read_poly_file.m$ to read in a text file like *cube.poly*. The format of this file must first list the vertices in order, then list faces as triples of vertex indices. The function *test_mesh_object.m* is useful when creating your own meshes as it will plot your mesh along with the face normals. It's important for collision detection to make sure that the normals are all pointing "out" of the body.

4.2 Collision detection

Consider two bodies $\mathcal{A}$ and $\mathcal{B}$ near contact as in Figure 4.3. A contact between

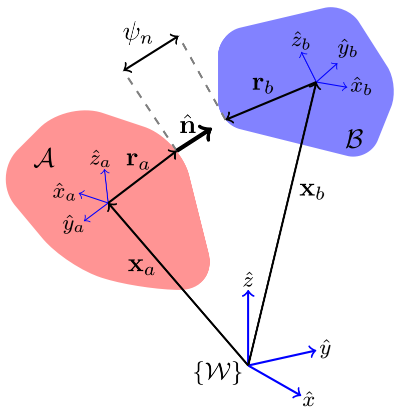


Figure 4.3: Two bodies, $\mathcal{A}$ and $\mathcal{B}$, and contact vectors in the world frame $\{\mathcal{W}\}$.

them may be represented by a 5-tuple $\mathbf{c} = \{\mathcal{A}_{id}, \mathcal{B}_{id}, \mathbf{p}_a, \hat{\mathbf{n}}, \psi_n\}$, where $\mathcal{A}_{id}$ and $\mathcal{B}_{id}$ are body indexes or IDs for $\mathcal{A}$ and $\mathcal{B}$ respectively, $\mathbf{p}_a$ is the contact point on $\mathcal{A}$ in world coordinates, $\hat{\mathbf{n}}$ is the contact normal vector in the direction of $\mathcal{A}$ onto $\mathcal{B}$ by convention, and ψ_n is the signed gap distance between the two bodies. This notion of a contact is represented by *Contact.m* and such contacts are generated during collision detection.

The default collision detection routine is *collsion_detection.m* which loops over all pairs of bodies and does some form of broad-phase detection before performing a narrow-phase. Pairs of meshes use axis-aligned bounding boxes whereas spheres against planes skip the broad phase and test the distance directly. Specific tests are contained in engine/collision_detection/body_tests.

4.3 Contact constraint

Using complementarity conditions to represent normal constraints, or non-penetration constraints, at points of contact is common. For the non-penetration constraint on a set of contacts, the complementarity condition takes the form

$$\mathbf{0} \leq \boldsymbol{\lambda}_n \perp \boldsymbol{\psi}_n \geq \mathbf{0} \tag{4.1}$$

where the vector $\boldsymbol{\lambda}_n$ represents the forces in the direction of the contact normals, and $\boldsymbol{\psi}_n$ is the vector of gap distances. Equation (4.1) states that for a given contact, if the force being applied is positive then the gap distance must be zero, and if the gap distance is positive then the force must be zero. Further, both the force and the gap distance are constrained to be non-negative.

From the contacts identified by collision detection, the unilateral contact constraint between bodies A and B may be written as

$$\mathbf{0} \leq \frac{\boldsymbol{\psi}_n}{h} + \mathbf{G}_n^T \boldsymbol{\nu}^{\ell+1} + \frac{\partial \boldsymbol{\psi}_n}{\partial t} \perp \boldsymbol{\lambda}_n \geq \mathbf{0} \tag{4.2}$$

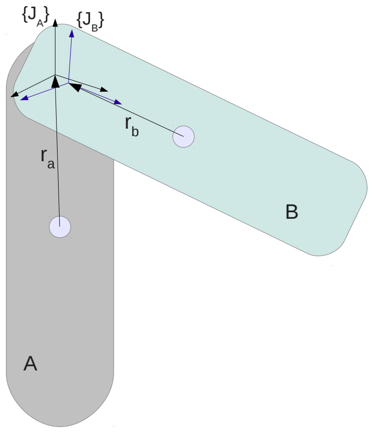
where the Jacobian is given by

$$\mathbf{G}_n = \begin{bmatrix} \hat{\mathbf{n}} \\ \mathbf{r}_a \times \hat{\mathbf{n}} \\ -\hat{\mathbf{n}} \\ \mathbf{r}_b \times -\hat{\mathbf{n}} \end{bmatrix} \tag{4.3}$$

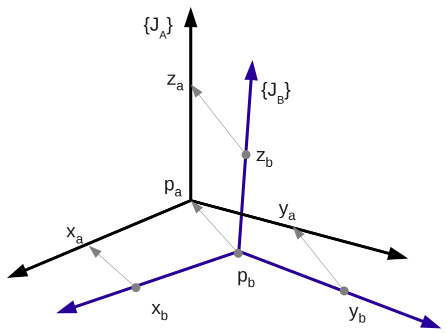
4.4 Joints

4.4.1 Joint representation

Joints are represented in RPIsim as bilateral constraints. Take for example the joint depicted in Figure 4.4 Joint constraints attempt to constrain the relative



(a) Two joined bodies and their respective joint frame coordinate systems $\{J_A\}$ and $\{J_B\}$.



(b) The set of joint axes and possible errors in origins as well as unit points along constraint directions.

Figure 4.4: A joint constraint between two bodies A and B.

positions of joint frames of either one or two bodies. Given two bodies A and B, each has their own joint frame $\{J_a\}$ and $\{J_b\}$ fixed relative to their respective bodies. Let $\mathbf{r}_a$ be the vector from the center of mass of body A to the joint frame $\{J_a\}$, and let $\hat{\mathbf{x}}_a$, $\hat{\mathbf{y}}_a$, and $\hat{\mathbf{z}}_a$ be the unit vectors corresponding to the primary axes of $\{J_a\}$, all expressed in the world frame. Body b has analogously defined vectors for its joint frame $\{J_b\}$. We may write the joint Jacobian as the stacked matrix $\mathbf{G}_b = \begin{bmatrix} \mathbf{G}_A \\ \mathbf{G}_B \end{bmatrix}$. If we were to constrain both position and rotation about all three axes, we would have a "fixed" joint where

$$\begin{aligned} \mathbf{G}_A &= \begin{bmatrix} \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a & \hat{\mathbf{z}}_a & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{r}_a \times \hat{\mathbf{x}}_a & \mathbf{r}_a \times \hat{\mathbf{y}}_a & \mathbf{r}_a \times \hat{\mathbf{z}}_a & \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a & \hat{\mathbf{z}}_a \end{bmatrix} \\ \mathbf{G}_B &= \begin{bmatrix} -\hat{\mathbf{x}}_a & -\hat{\mathbf{y}}_a & -\hat{\mathbf{z}}_a & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{r}_b \times -\hat{\mathbf{x}}_a & \mathbf{r}_b \times -\hat{\mathbf{y}}_a & \mathbf{r}_b \times -\hat{\mathbf{z}}_a & -\hat{\mathbf{x}}_a & -\hat{\mathbf{y}}_a & -\hat{\mathbf{z}}_a \end{bmatrix} \end{aligned} \tag{4.4}$$

We can think of the six columns of equation (4.4) as respectfully constraining the x position, y position, z position, x rotation, y rotation, and z rotation with regard to the assumed initially coincident joint frames $\{J_a\}$ and $\{J_b\}$. From here, we can create any common joint we wish by simply masking out the columns for which we would like to introduce a degree of freedom. A revolute joint about the $\hat{\mathbf{z}}$ axis would include columns one through five, but eliminate column six, thus allowing non-zero rotational velocity about the joint's $\hat{\mathbf{z}}$ direction. Table 4.1 contains the Jacobian definitions (for $\mathbf{G}_A$) for several common joint types.

The bilateral constraint may be written as

$$\frac{\boldsymbol{\psi}_b}{h} + \mathbf{G}_b^T \boldsymbol{\nu}^{\ell+1} + \frac{\partial \boldsymbol{\psi}_b}{\partial t} = 0 \tag{4.5}$$

where $\boldsymbol{\psi}_b$ is the current constraint error $\boldsymbol{\psi}_b = \begin{bmatrix} \mathbf{p}_a - \mathbf{p}_b \\ \boldsymbol{\theta}_a - \boldsymbol{\theta}_b \end{bmatrix}$, which is the stacked vector of position an rotation error. Note that (4.5) does not currently include joint friction, nor does it contain a mechanism for enforcing joint limits.

4.4.2 Joint stabilization

It is possible (and depending on the implementation details, probable) for the bilateral constraints formulated in 4.4.1 to "drift." That is, error builds up over time steps which allows the origin of the joint frames to separate. Because of this, we require joint stabilization in order to correct this drift during simulation.

4.5 Friction

To simulate friction at points of contact, a commonly used model is Coulomb friction [dC21]. Coulomb's friction law models stick-slip friction and states

$$f_f \leq \mu f_n \tag{4.6}$$

where F_f is the frictional force opposing motion between two bodies, F_n is the normal force perpendicular to the plane of contact, and μ is the coefficient of friction between the two materials in contact. When the relative velocity of the two bodies is zero at the point of contact (sticking), the magnitude of the frictional force may lie anywhere withing $[0, \mu F_n]$. When the relative velocity of the two bodies is non-zero, then the frictional force is exactly μF_n. This definition is conveniently similar to that of the complementarity problem.

In 3D, this model represents a quadratic cone of feasible forces. In order to simplify the dynamics solution, this cone may be linearized by a polygonal approximation as depicted in Figure 4.5.

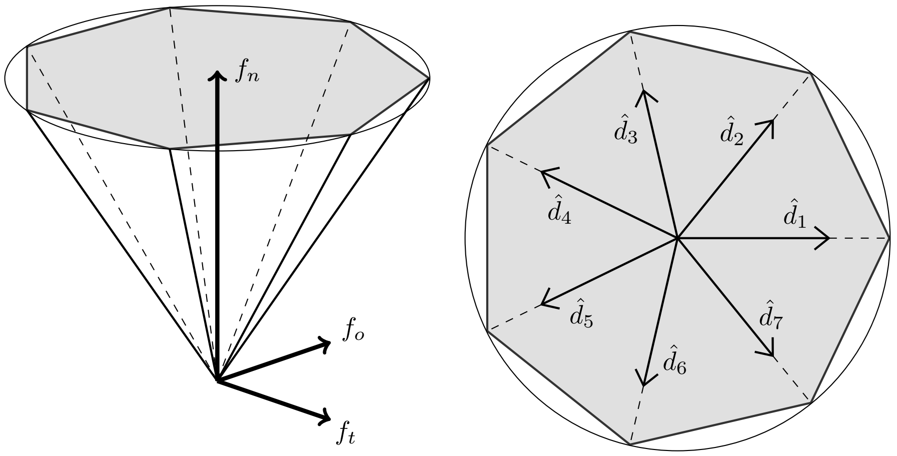


Figure 4.5: The friction cone, and its polygonal approximation.

The friction directions in the linearized friction cone are represented by n_d vectors that are perpendicular to the normal force f_n. Although one may choose any number of friction directions and orient them in any direction (within the contact plane), it is better in practice to use $n_d \geq 7$, generally use an odd number of directions, and keep them equally spaced. Equal spacing simply makes the result more numerically stable and robust. Increasing n_d gives better results (the classic tradeoff of speed versus accuracy), but keeping $n_d \geq 7$ is found to be large enough in practice to eliminate most error introduced by linearization.

Given the linearized friction cone and unit direction vectors $\hat{\mathbf{d}}_1$ through $\hat{\mathbf{d}}_{n_d}$, the friction constraint is written as

$$\begin{gathered} 0 \leq \mathbf{p}_f^{\ell+1} \perp \mathbf{G}_f^T \boldsymbol{\nu}^{\ell+1} + \mathbf{E}\boldsymbol{\sigma}^{\ell+1} + \frac{\partial \boldsymbol{\psi}_f}{\partial t} \geq 0 \\ 0 \leq \mathbf{s}^{\ell+1} \perp \mathbf{U}\mathbf{p}_n^{\ell+1} - \mathbf{E}^T \mathbf{p}_f^{\ell+1} \geq 0 \end{gathered} \tag{4.7}$$

where

$$\mathbf{G}_f = \begin{bmatrix} \hat{\mathbf{d}}_1 & ... & \hat{\mathbf{d}}_{n_d} \\ (\mathbf{r} \times \hat{\mathbf{d}}_1) & ... & (\mathbf{r} \times \hat{\mathbf{d}}_{n_d}) \end{bmatrix} \tag{4.8}$$

where r is the vector from the center of mass of a contact body to the contact point,

$$\mathbf{U} = diag(\mu_1, \ldots, \mu_{n_c}) \tag{4.9}$$

where μ_j is the coefficient of friction for the j^{th} contact of n_c contacts, and

$$\mathbf{E} = blockdiag\left(\begin{vmatrix} 1 \\ 1 \\ \vdots \end{vmatrix}_1, \ldots, \begin{vmatrix} 1 \\ 1 \\ \vdots \end{vmatrix}_{n_c} \right) \tag{4.10}$$

where each column vector of ones has length equal to the number of friction directions in the friction cone. The term $\mathbf{s}^{\ell+1}$ denotes the slip speed and $\boldsymbol{\sigma}^{\ell+1}$ is the remaining potential impulse, essentially $\boldsymbol{\sigma} = \boldsymbol{\mu}\mathbf{p}_n - \mathbf{p}_f$. These definitions are expounded in 4.6.

4.6 Dynamics formulation

There are several ways available for modelling the dynamic behaviour of the simulation. The default formulation is in *LCPdynamics.m* and constructs the matrices in vectors needed to solve the linear complementarity problem [ST96]

$$\mathbf{0} \leq \begin{vmatrix} \mathbf{G}_n^T\mathbf{M}^{-1}\mathbf{G}_n & \mathbf{G}_n^T\mathbf{M}^{-1}\mathbf{G}_f & \mathbf{0} \\ \mathbf{G}_f^T\mathbf{M}^{-1}\mathbf{G}_n & \mathbf{G}_f^T\mathbf{M}^{-1}\mathbf{G}_f & \mathbf{E} \\ \mathbf{U} & -\mathbf{E}^T & \mathbf{0} \end{vmatrix} \begin{vmatrix} \mathbf{p}_n^{\ell+1} \\ \mathbf{p}_f^{\ell+1} \\ \mathbf{s}^{\ell+1} \end{vmatrix} + \begin{vmatrix} \mathbf{G}_n^T(\nu^\ell + \mathbf{M}^{-1}p_{ext}^\ell) + \frac{\boldsymbol{\Psi}_n^\ell}{h} + \frac{\partial \boldsymbol{\Psi}_n^\ell}{\partial t} \\ \mathbf{G}_f^T(\nu^\ell + \mathbf{M}^{-1}p_{ext}^\ell) + \frac{\partial \boldsymbol{\Psi}_f}{\partial t} \\ \mathbf{0} \end{vmatrix} \perp \begin{vmatrix} \mathbf{p}_n^{\ell+1} \\ \mathbf{p}_f^{\ell+1} \\ \mathbf{s}^{\ell+1} \end{vmatrix} \geq 0 \tag{4.11}$$

where after a solution is found for $\mathbf{p}_n^{\ell+1}$ and $\mathbf{p}_f^{\ell+1}$, new velocities are calculated for all bodies with

$$\boldsymbol{\nu}^{\ell+1} = \boldsymbol{\nu}^\ell + \mathbf{M}^{-1}\mathbf{G}_n\mathbf{p}_n^{\ell+1} + \mathbf{M}^{-1}\mathbf{G}_f\mathbf{p}_f^{\ell+1} + \mathbf{M}^{-1}\mathbf{p}_{ext} \tag{4.12}$$

Note that (4.11) does not include bilateral constraints. Currently when simulating joints, you must use *mLCPdynamics.m* which constructs the mixed linear complementarity problem as

$$\begin{vmatrix} \mathbf{0} \\ \boldsymbol{\rho}_b^{\ell+1} \\ \boldsymbol{\rho}_n^{\ell+1} \\ \boldsymbol{\rho}_f^{\ell+1} \\ \boldsymbol{\sigma}^{\ell+1} \end{vmatrix} = \begin{vmatrix} -\mathbf{M} & \mathbf{G}_b & \mathbf{G}_n & \mathbf{G}_f & \mathbf{0} \\ \mathbf{G}_b^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{G}_n^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{G}_f^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{E} \\ \mathbf{0} & \mathbf{0} & \mathbf{U} & -\mathbf{E}^T & \mathbf{0} \end{vmatrix} \begin{vmatrix} \boldsymbol{\nu}^{\ell+1} \\ \mathbf{p}_b^{\ell+1} \\ \mathbf{p}_n^{\ell+1} \\ \mathbf{p}_f^{\ell+1} \\ \mathbf{s}^{\ell+1} \end{vmatrix} + \begin{vmatrix} \mathbf{M}\boldsymbol{\nu}^\ell + \mathbf{p}_{ext} \\ \frac{\boldsymbol{\Psi}_b^\ell}{h} + \frac{\partial \boldsymbol{\Psi}_b^\ell}{\partial t} \\ \frac{\boldsymbol{\Psi}_n^\ell}{h} + \frac{\partial \boldsymbol{\Psi}_n^\ell}{\partial t} \\ \frac{\partial \boldsymbol{\Psi}_f^\ell}{\partial t} \\ \mathbf{0} \end{vmatrix} \tag{4.13}$$

where $\boldsymbol{\rho}_b$, $\boldsymbol{\rho}_n$, $\boldsymbol{\rho}_f$, and $\boldsymbol{\sigma}$ are slack variables.

4.7 Solving dynamics

Several solvers are available including an implementation of Lemke's method [FM07], the PATH solver [FM00], and a suite of solvers num4lcp [EASM11]. These solvers are classified into pivoting methods and iterative methods.

4.7.1 Pivoting method

Lemke's algorithm is one robust pivoting method, although there is no general solution method which guarantees the solution of any given LCP except for a total enumeration of 2^n for a problem of size n. Lemke's algorithm works well for a small problem size while for larger-size problem, it tends to be slow.

4.7.2 Iterative method

The iterative method, such as Projected Gauss Seidel (PGS), generalized Newton's method and fixed-point iteration methods are included in the RPIsim. For more details about these methods, please refer to [**?**] (todo: needs a bib). The following table shows briefly about the available solvers inside the RPIsim: To call a specific solver, say, to use PATH solver, you need to call it by setting:

```
sim.H_solver = @pathsolver;
```

Joint Type	**Jacobian**
Rigid (fixed)	$\mathbf{G}_A = \begin{bmatrix} \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a & \hat{\mathbf{z}}_a & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{r}_a \times \hat{\mathbf{x}}_a & \mathbf{r}_a \times \hat{\mathbf{y}}_a & \mathbf{r}_a \times \hat{\mathbf{z}}_a & \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a & \hat{\mathbf{z}}_a \end{bmatrix}$
Revolute	$\mathbf{G}_A = \begin{bmatrix} \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a & \hat{\mathbf{z}}_a & \mathbf{0} & \mathbf{0} \\ \mathbf{r}_a \times \hat{\mathbf{x}}_a & \mathbf{r}_a \times \hat{\mathbf{y}}_a & \mathbf{r}_a \times \hat{\mathbf{z}}_a & \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a \end{bmatrix}$
Prismatic	$\mathbf{G}_A = \begin{bmatrix} \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{r}_a \times \hat{\mathbf{x}}_a & \mathbf{r}_a \times \hat{\mathbf{y}}_a & \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a & \hat{\mathbf{z}}_a \end{bmatrix}$
Cylindrical	$\mathbf{G}_A = \begin{bmatrix} \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a & \mathbf{0} & \mathbf{0} \\ \mathbf{r}_a \times \hat{\mathbf{x}}_a & \mathbf{r}_a \times \hat{\mathbf{y}}_a & \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a \end{bmatrix}$
Spherical	$\mathbf{G}_A = \begin{bmatrix} \hat{\mathbf{x}}_a & \hat{\mathbf{y}}_a & \hat{\mathbf{z}}_a \\ \mathbf{r}_a \times \hat{\mathbf{x}}_a & \mathbf{r}_a \times \hat{\mathbf{y}}_a & \mathbf{r}_a \times \hat{\mathbf{z}}_a \end{bmatrix}$

Table 4.1: Joint Constraint Jacobian Definitions

Table 4.2: Solvers available with RPIsim.

Name	**Applicable Model**	**Category**
Lemke	LCP	pivotal method
PATH	mLCP	mixed pivotal and iterative method
PGS	LCP, mLCP	iterative method with projection
Projected Jacobi	LCP, mLCP	iterative method with projection
Fixed-point	LCP, NCP	iterative method
Interior-point	LCP	iterative method
Fischer-Newton	LCP, NCP	non-smooth Newton method with line search
Minmap-Newton	LCP, NCP	non-smooth Newton method with line search

Chapter 5

Customizing the simulator

One of the main goals of RPIsim is to enable easy customizations and extensions through its modular structure. This chapter describes some of the default modules and gives examples on how to replace them with custom modules. Essentially this comes down to understanding the interface for a given component in the simulation loop (Figure 4.2).

5.1 Collision detection

The default collision detection function in RPIsim is *collision_detection.m*, which can find contacts between combination pairs of planes, spheres, and triangle meshes (of course we never test plane against plane). Unfortunately, the algorithms implemented in the default function are slow, so it is up to you to write optimized routines (and commit them to the repository!).

A collision detection routine in RPIsim will iterate over all bodies contained in *sim.bodies* and determine and store a set of contacts in *sim.contacts*.

Important things to remember when writing a collision detection routine in RPIsim:

- When adding a contact between bodies A and B, make sure to "activate" those bodies with

  ```
  sim = sim_activateBodies( sim, Aid, Bid );
  ```

  This marks each body as dynamically "active" for the current time step, and is important for all of the indexing that will happen later in the simulation loop (when building matrices for the dynamics).

- Some bodies may be static, so improve efficiency by checking:

  ```
  if ~A.dynamic && ~B.dynamic, continue; end
  ```

- Some bodies are set to not collide (e.g. two joined bodies), so avoid errors by checking:

  if any(A.doesNotCollideWith == B.bodyID), continue; end

There are several functions included in *engine/collision_detection/body_tests* that may be helpful in writing a collision detection routine.

5.2 Formulating dynamics and solving

In order to incorporate a custom dynamics formulation or solver, the user should be familiar with how dynamics components are passed from one block of the simulation loop to the next. By default, the function *preDynamics.m* puts together the necessary matrices and vectors including the mass-inertia matrix and the Jacobians for unilateral contact, bilateral joints, and friction constraints. These matrices are stored in the struct *sim.dynamics* in order to pass them along.

A custom dynamics formulation may choose to use the value in *sim.dynamics* or not. If not, then it is of course most efficient to remove the call to *preDynamics*() inside of *sim_run.m*.

Chapter 6

Using Bullet collision detection

IMPORTANT: Unfortunately, the Bullet interface will only work with MATLAB 32-bit versions, and will not work with Octave nor MATLAB 64-bit versions. This is because 1. it requires use of MEX files, which Octave does not support and 2. it requires use of 32-bit MEX files and 64-bit MATLAB cannot compile 32-bit MEX files.

6.1 Installing Bullet

In order to use the Bullet collision detection library, you first need Bullet installed. This can be done by following the instructions on http://bulletphysics.org/mediawiki-1.5.8/index.php/Installation.

Under Linux, I was able to do this with revision 2672 of Bullet:

```
$ svn checkout http://bullet.googlecode.com/svn/trunk/ bullet-read-
    only
$ cd bullet-read-only
$ mkdir bullet-build
$ cd bullet-build
$ cmake .. -G "Unix Makefiles" -DINSTALL_LIBS=ON
                          -DBUILD_SHARED_LIBS=ON
$ make -j4
$ sudo make install
```

6.2 Using Bullet with RPIsim

Once Bullet is installed, we need to build the interface that will pass the RPIsim body information to Bullet. This is done in RPIsim with

```
>> Compile_BULLET
```

One possible error may occur if you do not have a compiler set in MATLAB for compiling MEX functions.

Chapter 7

Sample application: proportional-derivative controlled robotic arm

In this section, we will demonstrate how to use RPIsim to simulate a robotic arm and a joint space controller for the arm.

7.1 Modeling the arm

Many thanks go to Rohinish Gupta who did the work of importing the mesh of the robotic arm into MATLAB. The arm we model is the Schunk Powerball arm, and the files are contained in *examples/powerball*. Specifically the script powerball.m loads the mesh bodies, defines the joints between all bodies, and initializes the simulator.

7.2 Kinematics

Directly controlling the arm by setting the joints to specific angles is straight forward as it is handled by the forward kinematics in the model. The variable *jointAngle* in powerball.m is initially set to all zeros, but may be altered to any set of joint angles.

7.3 Dynamic simulation

Dynamic control of the arm through a proportional-derivative controller utilizing the userFunction functionality as described in section 3.4. Because the userFunction is called at the beginning of each time step, we can observe the

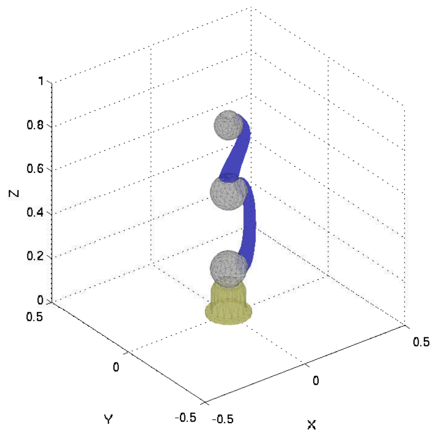


Figure 7.1: Simulation of a robotic arm in RPIsim. A PD-controller is used to control the arm in joint-space. Here, the arm is depicted in its zero-configuration.

current state of each joint as well as apply torques to any dynamic bodies. Our PD-controller userFunction will look like

```
function sim = powerball_controller( sim )
 theta_desired = ones(length(sim.joints),1) * sin(sim.step*.01);
 % Clear body torques
 for j=1:length(sim.joints)
  sim.bodies(sim.joints(j).body1id).Fext(4:6) = 0;
  sim.bodies(sim.joints(j).body2id).Fext(4:6) = 0;
 end

 % Calculate new torques
 for j=1:length(sim.joints)
  J = sim.joints(j);

  Perror = theta_desired(j) - J.theta;         % Proportional error
  Deriv  = (J.theta - J.theta_prev) / sim.h;   % Derivative
  joint_frame_torque = 50*Perror - 5*Deriv;    % PD controller

  t1 = J.T1world(1:3,1:3) * [0;0; -joint_frame_torque];
  sim.bodies(J.body1id).Fext(4:6) = sim.bodies(J.body1id).Fext(4:6)+t1;

  t2 = J.T2world(1:3,1:3) * [0;0; joint_frame_torque];
  sim.bodies(J.body2id).Fext(4:6) = sim.bodies(J.body2id).Fext(4:6)+t2;

  % For now, we need to keep track of this in the controller.
  sim.joints(j).theta_prev = J.theta;
 end
end
```

Script 7. PD-control userFunction.

Chapter 8

A framework for solver comparison in multibody dynamics

We have presented many solvers in the previous sections, solvers in minimal coordinate, in maximal coordinate; pivoting solvers, iterative solvers and even hybrid solvers. However, there is no currently available uniform comparison of all these solvers. Testing is usually conducted by using a small number of synthetic CPs, which will cause problem when applied to the CPs arising in “real” physics engines. Based on this, we have developed a standard interface based on Hierarchical Data Format 5 (HDF5) to fairly compare the various solvers [LLWT13].

8.1 Framework structure

A high-level view of the data hierarchy is shown in figure 8.1. The frames 1 to n represent n CPs corresponding to n simulation time steps (not necessarily consecutive). Each frame contains the information about the bodies, their joint constraints, the contacts (which are enough to set up any desired CP), the size of the time step, and solver data including the initial guess (or warm start value) for the solution, the solver’s error at each iteration, and the solution returned.

In the “bodies” class, the body “id” is a unique reference for each body. Each body experiences generalized forces, which cause it to accelerate in a manner consistent with its mass and inertia tensor. The pose of each body is stored as a position vector and a unit quaternion, where the first element is the real part of the quaternion. The last piece of the “bodies” data structure is the generalized velocity.

In the “joints” class, the information needed to enforce the constraints imposed by the bilateral joints in systems in maximal coordinates is stored. The

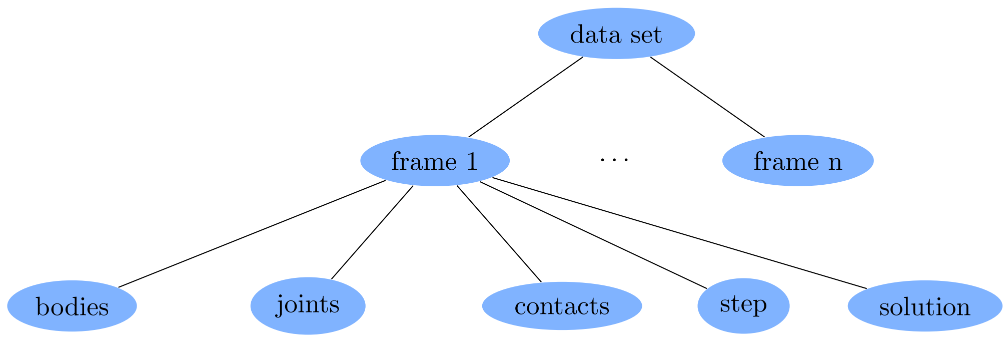


Figure 8.1: Simulation data hierarchy

Jacobian matrix which transforms from the joint frame to the world frame is saved in the field of "jacobian." For different kinds of joints, the size of Jacobian matrix varies, so we use a "row" vector to record the size of each Jacobian component for each joint. Joint limits, if any, are stored in the field "bounds." "Pairs" is the pair of body ids of the two bodies that form the joint.

For each contact, the signed gap distance between the two bodies is saved in the field "gap," with positive values being the separation distance and values corresponding to penetration depth. The coefficient of friction is saved in the field of "mu." For unilateral contacts, we may reconstruct the Jacobian matrix which transforms from the contact frame to the world frame, so the fields of "normals" and "points" are saved. The normals are the unit vector from the contact point and is perpendicular to and away from the surface of the 1^{st} body in the pair. The points are represented by vectors from the center of mass of the 1^{st} body to the contact point on the body.

The "step" is a field containing the single value of time step size in seconds.

8.2 Using the framework with RPIsim

The following is a sample scripts to compare several different solvers.

```
function results = TESTManySolversOneProb( )
% SolverHandles
solverHandles = {'Lemke'
                 'PATH'
                 'LCP_fixed_point'
                 'NCP_strict_PGS'
                 'FischerNewton'
                 'interior_point'
                 'minmap_newton'
                 'psor'
                };

% Parameters for the different error metrics (4 different metrics)
% 1. norm_CCK, 2. square_CCK,  3. norm_mCCK  4. square_mCCK
ErrorParams = struct();
ErrorParams.metricName = 'norm_CCK';

% Standard hdf5 file
problemFile = 'h5data/particle_problem.h5';
problem_indicies = (106:106);

% Other tunable parameters
% max_iter ---------- maximum number of iterations
% tol  -------------- tolerance
% alpha ------------- cfm value
% SaveOrNot---------- 1: save the results as eps; 0: otherwise
TuneParams = struct();
TuneParams.max_iter = 100;
TuneParams.tol = 10^(-6);
TuneParams.alpha = eps;
TuneParams.SaveOrNot = 1;
results = CompareManySolversOneProb(solverHandles, problemFile,
     ErrorParams, problem_indicies, TuneParams);
end
```

Script 8. Compare 8 different solvers.

Table 8.1: Standard notations in HDF5

	Fields	**Names**	**Math Notation**	**Sizes**	**Physical meaning**
HDF5	bodies (nb)	ids	i	$nb \times 1$	body ids
		masses	M	$nb \times 1$	masses
		forces	$\lambda = \begin{bmatrix} \mathbf{f} \\ \tau \end{bmatrix}$	$6nb \times 1$	external forces
		inertia	$\mathbf{I}$	$3nb \times 3$	inertia
		positions	$\mathbf{u} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$	$3nb \times 1$	positions
		quaternion	$Q = \begin{bmatrix} q_{\text{real}} \\ \mathbf{q}_{\text{img}} \end{bmatrix}$	$4nb \times 1$	body orientation
		velocities	$\nu = \begin{bmatrix} \mathbf{v} \\ \omega \end{bmatrix}$	$6nb \times 1$	generalized velocities
	joints (nj)	violations	ϕ	$nj \times 1$	joint violations
		jacobians	J	$(nj \times Type) \times 3$	Jacobians
		pairs	$pair$	$nj \times 2$	body pairs in joints
		bounds	$bound$	$nj \times 2$	joint limits
		rows	row	$Type \times 3$	Jointwise Jacobian rows
	contacts (nc)	gap	ψ	$nc \times 1$	gap distance
		pairs	$pair$	$nc \times 2$	body pairs in contact
		mu	μ	$nc \times 2$	coefficients of friction
		normals	$\hat{\mathbf{n}}$	$nc \times 3$	contact normals
		points	$\hat{\mathbf{p}}$	$nc \times 3$	contact points
	solutions	solution	z	$6nb \times 1$	solution
		iterations	$iter$	$sim \times 1$	solver iterations
		total_error	$totalError$	$iter \times 1$	total errors
		normal_error	$normError$	$iter \times 1$	normal errors
		friction_error	$fricError$	$iter \times 1$	friction error
		stick_or_slide	$state$	$iter \times 1$	state of contacts

Bibliography

[BETC12] Jan Bender, Kenny Erleben, Jeff Trinkle, and Erwin Coumins, *Interactive simulation of rigid body dynamics in computer graphics*, Conference of the European Association for Computer Graphics, State of The Art Report (STAR), 2012.

[dC21] Charles-Augustin de Coulomb, *Thorie des machines simples*, IHES (1821).

[EASM11] K. Erleben, M. Andersen, Niebe S., and Silcowitz M., *num4lcp*, Published online at code.google.com/p/num4lcp/, October 2011, Open source project for numerical methods for linear complementarity problems in physics-based animation.

[FM00] M.C. Ferris and T.S. Munson, *Complementarity problems in gams and the path solver*, Journal of Economic Dynamics and Control **24** (2000), no. 2, 165 – 88 (English).

[FM07] Paul Fackler and Mario Miranda, *Lemke, solver for linear complementarity problems*, `http://people.sc.fsu.edu/~jburkardt/m_src/lemke/lemke.html`, 2007.

[LLWT13] C. Lacoursiere, Y. Lu, J. Williams, and J.C. Trinkle, *Standard interface for data analysis of solvers in multibody dynamics*, Canadian Conference on Nonlinear Solid Mechanics (CanCNSM), July 2013.

[ST96] D.E. Stewart and J.C. Trinkle, *An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction*, International Journal for Numerical Methods in Engineering **39** (1996), no. 15, 2673 – 91.