# CONTENTS

1 What is XDE Physics ?
  1.1 An interactive dynamical multibody systems simulator .......................... 1
  1.2 Use of generalized coordinates ......................................................... 1
  1.3 Precise collision detection .................................................................. 3
  1.4 Precise contact handling ...................................................................... 4
  1.5 Deformable object simulation ............................................................... 4
  1.6 All dynamical effects. ......................................................................... 5
  1.7 Easy prototyping of robot controller ..................................................... 5
  1.8 Robust simulation .............................................................................. 5
  1.9 Summary - What you can do with XDE Physics ........................................ 5

2 Quick Start
  2.1 Get XDE Physics ................................................................................. 7
  2.2 Installing XDE Physics .......................................................................... 7
  2.3 Setting your building chain for XDE Physics ............................................ 7
  2.4 A first dive into XDE Physics ................................................................. 7

3 Multibody dynamics
  3.1 Definition ............................................................................................ 11
  3.2 Kinematic Graph .................................................................................. 11
  3.3 Rigid Bodies ....................................................................................... 13
  3.4 Joints ..................................................................................................... 15
  3.5 Smooth interactions ............................................................................ 22
  3.6 Constraints ........................................................................................... 26
  3.7 Collision handling ................................................................................ 28
  3.8 Friction models .................................................................................... 28
  3.9 A complex example ............................................................................. 28

4 Collision detection .................................................................................. 29

5 Handling Contact
  5.1 Contact Laws provided by XDE Physics ............................................... 31
  5.2 Configuring XDE Physics for contact handling ....................................... 35

6 Beams simulation .................................................................................... 39

7 Controlling Robots .................................................................................. 41

8 Tuning your simulation to improve performances .................................... 43
WHAT IS XDE PHYSICS?

1.1 An interactive dynamical multibody systems simulator

XDE Physics is an interactive physics engine, featuring precise collision detection, multibody and beam dynamics. It is written in C++ and has been optimized to provide maximal efficiency.

It allows to animate complex scenes with multiple rigid bodies interconnected with mechanical joints very efficiently. As it has been designed for virtual prototyping, XDE Physics offers precise collision handling as well as very accurate simulation of dynamic effects, both at rates compatible with haptic rendering.

To achieve this objective, XDE Physics differs from the other physics engines on many points. We will discuss these points below.

1.2 Use of generalized coordinates

Almost all the other physics engines simulating multibody systems use forces to ensure constraints respect between rigid bodies. These forces are usually referred to as Lagrange Multipliers.

On the opposite, XDE Physics uses a completely different strategy to enforce constraints between rigid bodies. XDE Physics changes the system parametrization. For instance, if we want to simulate two rigid bodies linked by a hinge joint, in most physics engines, this system would be modeled by 2 rigid bodies with 6 Degree of Freedom each, (3 rotations, 3 translation) and an additional constraint to enforce hinge joint respect.

In XDE Physics, this system would only be described by one Degree of Freedom (DOF), the hinge rotation angle. Hence, the system is simply described by a single parameter. This parametrization is computed by XDE Physics. The coordinates of this parametrization are called generalized coordinates.

Important:

This strategy has two important consequences for the simulation:

- It reduces the numbers of DOF of the system, hence reducing the complexity of the simulation.
- The constraint can’t be violated, as they are naturally defined by the parametrization. There is not artificial constraint forces applied to ensure the constraint respect.

See multibody-ref for further information on generalized coordinates.
Figure 1.1: A typical Xde Physics simulation.
1.3 Precise collision detection

In physics simulation, objects are usually meant to interact with each other. This interaction occurs when objects are in contact with each other. These contacts are detected during a collision detection step.

XDE Physics offers two ways to detect collision between the geometry of rigid bodies.

1.3.1 LMD++

In physics engine, the rigid body geometry is described by a triangle soup. One possible way to detect collision between 2 rigid bodies consists in using the triangle soup of each rigid bodies geometry. This is what LMD++ does.

LMD++ is a very precise collision engine designed to compute collision information. More specifically, it provides qualitative information about the nature of the contact: plan/plan, edge/edge, vertex/edge, ...

This information, that is extracted from the mesh topology, allows the removal of many useless contact points. This reduction of the number of contact points allows a more efficient simulation as it simplifies the load of the contact solver. However, this come at a higher cost of the collision detection steps.

Hint:

Use LMD++ when:

- You want precise contact information, to perform “peg in the hole” insertion task for example
- You want to reduce the number of contact points to improve the performances of the contact solver.
• Your scene contains objects whose number of triangles is not too important (circa 100K triangles).

1.3.2 CDDIF

Jérémie.

Tip: If precision if important for your simulation, for instance if you plan to do “peg in the hole” insertion task, use LMD++. On the opposite, if simulation rate is the key point of your simulation, i.e. if you want to do haptic rendering with a huge geometrical model, prefer CDDIF.

1.4 Precise contact handling

XDE Physics also differs from the other simulation engine regarding the way it handles contact and friction. XDE Physics provides two ways to simulate contact:

• It can use contact regularization methods, i.e. penalty methods to handle contact without friction. This is commonly done in multi body simulation engine.

This method is very fast but can lead to numerical instability. Moreover, this kind of methods required the setting of a penalty stiffness factor, which can be tedious to set.

• It can use non smooth methods to solve the physical contact and friction laws. XDE Physics currently supports 3 contacts laws: Signorini, Coulomb and Coulomb-Contensou.

Once the contact information have been computed, they are used in the contact solver to enforce the non penetration constraints and to compute the friction forces. Again, XDE Physics differs from the other physics engines on the several points:

Important:

• XDE Physics solves the contact and friction constraints using differential inclusions. It can also provide penalty method for contact without friction.

• XDE Physics provides various physics based friction model: Signorini, Coulomb, and Coulomb-Contensou laws.

• XDE Physics does not discretise the friction laws. For instance, Coulomb friction cones are not discretized. This avoid to introduce artefacts due to the discretization.

See Handling Contact for more details on friction.

1.5 Deformable object simulation

XDE Physics provide a module to accurately simulates beams and cables. This modules, use physical and geometric properties like beams section radius, Young Modulus, Shear Modulus, Poisson Coefficient, ... to easily define the beams physical behaviour.

Error: Figure needed here
1.6 All dynamical effects.

XDE Physics can simulate all the dynamical effects of dynamically moving multibody systems: Coriolis forces, ..

1.7 Easy prototyping of robot controller

XDE Physics offers functions to compute the various mathematical operators required to create advance robot controller based on state of the art methods. This controller can be written in the cartesian as well as articular space. See Controlling Robots for further information robot controller design.

1.8 Robust simulation

Thanks to the advanced methods XDE Physics used to simulate multibody systems dynamics, XDE Physics offers very robust simulation, where rigid bodies can be very constrained and still remains stable. XDE Physics, using generalized coordinates, can be sued to design articular as well as cartesian controller.

1.9 Summary - What you can do with XDE Physics

As a summary, we’ll quickly review some example of systems that can interactively be simulated by XDE Physics.

1.9.1 Any kind of manipulating robots

Industrial robots are made of rigid members interconnected by joints. XDE Physics has been designed especially to handle this kind of systems.

Error: Figure needed here.

1.9.2 Virtual prototyping

As XDE Physics is a physically based simulation engine. Moreover, it supports bodies with complex geometries. This offers many possibilities to check the design of complex multi body systems at the very first steps of their design: * With XDE Physics you can check interactively that various part of a complex system can be assembled. * XDE Physics can provide all the information regarding the dynamical behaviour of the systems: body position speed, articular efforts, ... * The beam module of XDE Physics is also very useful to check the assembly of systems.

1.9.3 Serious Gaming

The two keys features of XDE Physics lie in its interactivity and its respect of the physical laws. This made of XDE Physics an excellent candidate to design serious games.

For instance, writing an application that allow student to experiment with body, gravity, and friction law can be done very easily.
XDE Physics can also be an excellent engine for serious game where an virtual human operates, controlled by a student. In such virtual environment, the virtual operator will be able to interact naturally with any objects on the virtual world.

1.9.4 Simulating wheeled engine

Xde Physics handles collision, friction, and simulation of various joint amongst which hinge joint. It can also simulate spring and damper in cartesian as well as articular space. Hence, XDE Physics can simulate wheeled engine out of the box.

Error: Figure needed here. (Accessim !)

As you can see, XDE Physics is a very versatile simulation engine, and it can be used to simulated many systems.
CHAPTER TWO

QUICK START

2.1 Get XDE Physics

You can download the latest revision of XDE Physics on xde.fr.

2.2 Installing XDE Physics

Simply unzip XDE Physics next to your projet root directory.

2.3 Setting your building chain for XDE Physics

XDE Physics is distributed with a XDE PhysicsConfig.cmake file. We strongly encourage you to use CMake to generate your build chain. It is the perfect tool for quickly creating multi-platforms build chains.

Simply specify in your cmake the path to the repository where you installed XDE Physics and use `find_package` to get access to XDE Physics:

```cmake
find_package(XDE Physics)
include_directories(${XDE Physics_INCLUDE_DIRS})
...
target_link_library(myTarget ${XDE Physics_LIBRARIES})
```

2.4 A first dive into XDE Physics

Now we are ready to see how easily you can simulate complexe multibody systems with XDE Physics, and still get interactive performances.

For the sake of simplicity, we will illustrate the use of XDE Physics on a very simple example: the simulation of a pendulum. You can find the whole code for this example in tutorial 1, and more complexe examples in the other tutorials.

2.4.1 A simple pendulum

To get you familiar with the use of XDE Physics, we will build a very simple simulation of a pendulum. This system is described by Figure 1, and is made of a unique rigid body connected to a fixed point using an hinge joint.
Under the effect of gravity, this pendulum will oscillate freely.

### 2.4.2 The mechanical scene

In XDE Physics, the object responsible for the simulation is the *Scene*. All the simulation objects, rigid bodies, joints, forces, interactions, collision meshes, ... are added to the simulation through this *Scene* object. To create it, we need to set at least three parameters:

1. The simulation time step
2. The kind of solver used: FRICTION_DYNAMIC_INTEGRATOR, NO_FRICTION_DYNAMIC_INTEGRATOR, ...
3. The direction of the gravity field, which is the z axis by default.

```cpp
mechanicalScene = xde::gvm::SceneRef::createObject("mechanicalScene");
mechanicalScene.setIntegratorFlags(xde::gvm::FRICTION_DYNAMIC_INTEGRATOR);
mechanicalScene.setTimeStep(1e-3);
mechanicalScene.setVerticalDirectionUp(Eigen::Vector3d(0., 1.0, 0.));
```

In this example, we create a mechanical scene whose solver handles dynamic and friction. The time step is set to 1 millisecond, and the up direction is set along the Y axis.
2.4.3 Creating Rigid Bodies

XDE Physics simulates the dynamics of multiple rigid bodies connected by joints. Rigid bodies creation is as simple as this:

```cpp
pendulum = xde::gvm::RigidBodyRef::createObject("pendulum");
```

Only one parameter is required: the name of the body. Note that in XDE Physics, each object must have a unique name.

However, to physically simulate the dynamics of such multibody systems, we need to define the inertial properties of the rigid bodies, i.e. we need to set their masses, inertia frames, and their principal moments of inertia:

```cpp
pendulum.setMass(1.0);
pendulum.setPrincipalInertiaFrame(Displacementd::Identity());
pendulum.setPrincipalMomentsOfInertia(Eigen::Vector3d::Constant(1.));
```

This sets the mass of `pendulum` to 1 kilogram. Its center of mass and inertia frame are defined by `setPrincipalInertiaFrame`. Its three principal moments of inertia, sometimes referred as angular mass, are also set to 1.0. If you are not familiar with these quantities, they are explained in section ‘Multibody dynamics’.

The `RigidBody` class provides more than these three methods. We will cover these methods in the section ‘Multibody dynamics’.

---

Attention: In XDE Physics, there is a specific rigid body, called the `GroundBody`, which is automatically created, and which is the root of all the kinematic chains.

2.4.4 Associating Rigid Bodies and Collision Mesh

It is possible to associate a rigid body with a collision mesh. When two rigid bodies have a collision mesh, and when the collision between this two objects is enabled, XDE Physics will ensure not only that the two objects do not interpenetrate, but also that the friction law chosen is respected.

---

Error: Example code needed here.

2.4.5 Adding joint between Rigid Bodies

In XDE Physics, rigid bodies are interconnected by joints, that define motion between two rigid bodies. In this example, we’ll use the `HingeJoint`, which adds an hinge joint between two rigid bodies.

```cpp
hinge = xde::gvm::HingeJointRef::createObject("hinge");
```

---

Note: In fact, in XDE Physics, a joint does not add a constraint on the rigid body motion. It changes the parametrization of the system so that the constraint is naturally ensured. There is no artificial forces created to enforce the constraint.

This constraint must obviously be configured, to specify:

1. Its position relative to the previous body, in this case, the `GroundBody`
2. The 3d position of its center in the previous body frame
3. Its axis of rotation in the previous body frame
4. Its reference joint position

Once again, the configuration is quite straightforward:

```cpp
Eigen::Vector3d axis(0., 0., 1.);
Eigen::Vector3d center(0., 0., 0.);
std::cout << pendulum.getPosition() << std::endl;
```

2.4.6 Start the simulation and get results

Our simulation is now ready to use.

All we have to do to get the new position of our pendulum is call the `integrate` method and get the pendulum position:

```cpp
mechanicalScene.integrate();
std::cout << pendulum.getPosition() << std::endl;
```

In a typical simulation, which lasts for instance $t$ seconds, and whose timestep is $dt$, the method `integrate` is called $n$ times, with $n$ computed as follows:

```
\text{math:}
\begin{align*}
n &= \frac{t}{dt}
\end{align*}
```
CHAPTER
THREE

MULTIBODY DYNAMICS

Contents

• Multibody dynamics
  – Definition
  – Kinematic Graph
  – Rigid Bodies
  – Joints
  – Smooth interactions
  – Constraints
  – Collision handling
  – Friction models
  – A complex example

3.1 Definition

A multibody system is made of rigid bodies interconnected with joints. These joints define the way the successive rigid bodies move with respect to each other. A car for instance, is roughly made of 4 wheels connected to chassis using 4 hinge joints. An humanoid robot can also be modelized by limbs interconnected with hinge and ball joints.

3.2 Kinematic Graph

It is important to understand how XDE Physics represents complex multibody systems : it uses a kinematic graph. Each vertex of this graph is a rigid body, and each edge is a joint.

Hence, the kinematic graph of a simple car is describe by the Figure below:
This simply means that each car wheels is connected to the car frame with an hinge joint. Moreover, the car frame moves freely with respect to the inertia frame, that is called the ground body. See A special rigid body: the ground body for further details on this specific body.

XDE Physics supports any kind of graph, including graph with closed loop, thanks to a specific joint: the CutJoint:
3.3 Rigid Bodies

As stated above, multibody systems are made of rigid bodies. They are the vertices of XDE Physics kinematic graph. These rigid bodies have mainly three properties:

1. Their position and speed. As a Rigid Bodies moves in 3D, its position is defined by a 3D position and a 3D rotation. Similarly, its speed is defined by 3 translation speed, and 3 rotation speed both along the $x$, $y$, $z$ axis.

2. Their inertia parameters, that include mass, moments of inertia and inertia frame. We won’t discuss this point now. See Collision detection for more details.

3. Their geometry, essential for the collision handling. We will discuss this point in the section MB-collision-ref.

Inertia properties are very important, as they have significant effect on two points:

1. Inertia properties influence the dynamical behaviour of rigid bodies. Massive objects react more slowly to force than light bodies, i.e. their acceleration is inversely proportional to their mass. This is the Newton Second Law.

2. Incorrect inertia properties might also impact on the simulation numerical stability. For instance, mixing very light and very heavy objects might lead to an instable simulation. More information on this topic can be found in the Expert Manual.
They can have other properties, like a material defining friction properties, but they are less important and will be discussed specifically here: *Handling Contact.*

**Warning:** It is very important to remember that non fixed rigid bodies can’t have a null mass. One of the most frequent error responsible for unstable simulation is simply a rigid bodies whose mass was erroneously set to zero. This can be understand easily when you realize that the acceleration, necessary to update the speed, is inversely proportional to mass. Hence, if mass is null, acceleration is infinite! Fixed bodies don’t care has they can’t move and have no speed.

### 3.3.1 Inertia properties of rigid bodies

As stated above, a *RigidBody* evolves in 3D, and hence its position is described by a translation as well as a rotation. Therefore, to fully describes is dynamics, we need its inertia properties for the translation as well as the rotation.

These properties are defined:

- The rigid body mass. The mass is directly responsible for the translational inertia.
- The principal moments of inertia, that account for the rotational inertia.
- The principal frame of inertia, that specify the principal axes of inertia.

The principal moments are defined by a diagonal matrix $I_r$, of dimension $3x3$:

In the case of a *RigidBody* whose geometry is a sphere of radius $r$, and whose mass is $m$, this matrix is:

$$ I_r = \begin{pmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{pmatrix} $$

As stated above, the rotation inertia is expressed according the principal axes of inertia. Therefore you need to inform XDE Physics about this frame.

The inertia frame should always be defined with respect to the ground frame $x, y, z$.

Setting inertia properties is quite simple in XDE Physics, however, it is of some interest to understand why XDE Physics needs these information, and what it does with them.

#### The mass matrix

XDE Physics uses these data to build the mass matrix, which is defined for each rigid body of the simulation. This matrix is a 6x6 matrix, and completely defines rigid bodies inertia properties. This matrix is expressed in the world frame, and should be symmetric positive definite.

Since is it symmetric positive definite, it is possible to find a frame in which this mass matrix is diagonal, and its diagonal elements must all be positive. This frame is called the principal inertia frame. Hence, is is possible to build the mass matrix from its reduced, diagonal, 6x6 mass matrix and the principal frame of inertia, using the following formula:

$$ M = Ad((H^0_y)^{-1})^T \ast \begin{pmatrix} I_r & 0 \\ 0 & I_t \end{pmatrix} \ast Ad((H^0_y)^{-1}) $$

Where, as stated above $I_r$ holds the principal moment of inertia, and $I_t$ is the translational inertia:

$$ I_t = \begin{pmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix} $$
And $H^0_g \in SE(3)$ is the principal frame of inertia, and $Ad(.)$ is the adjoint operator. XDE Physics then use this inertia information to compute the dynamical motion of the rigid body.

### 3.3.2 Creating Rigid Bodies in XDE Physics

Let’s now see how to feed XDE Physics with these essential information:

```cpp
xde::gvm::RigidBodyRef aRigidSphere = xde::gvm::RigidBodyRef::createObject("aRigidSphere");
double mass = 1.0;
double r = 0.1;
// sphere principal moments of inertia
double Irxyz = 2./5.*mass*r*r;
aBody.setMass(mass);
//inertia axis are aligned with reference frame axes
aBody.setPrincipalInertiaFrame(Eigen::Displacement::Identity());
aBody.setPrincipalMomentsOfInertia(Eigen::Vector3d(Irxyz, Irxyz, Irxyz));
```

This block of code is quite self-explanatory, especially the first 2 lines. The third line, is more complicated. It sets the principal inertia frame using an `Eigen::Displacement`. This means that its set the orientation of the frame as well as its position. I.e. this methods set the orientation of the principal inertia axis and the position of the center of mass.

The fourth line sets the principal moments of inertia, also called the rotational inertia or the angular mass, expressed in the principal inertia frame. 

**Tip:** If your rigid body has been associated with a geometrical mesh, it is possible to use this mesh to compute the principal moments and the inertia frame. Simply set the mass and invoke `computePrincipalFrameAndMomentsOfInertiaUsingCompositeObbAndMass()`. Your body will behave like a rigid box oriented like the collision composite Oriented Bounding Box (OBB). See Collision detection for more details.

### 3.3.3 A special rigid body : the ground body

Every XDE Physics simulation contains at least a rigid body: the ground body. You will see in section Joints, that this body is the root node of the kinematic graph: all the kinematic graph branches are connected to this first node.

### 3.4 Joints

In the previous section, we have described the vertices of a kinematic graph: the rigid bodies. In this section we detail the edges of the kinematic graph, i.e. the joints.

Joint are the objects that describe the way rigid bodies move with respect from each other. To state this in a different manner, joint describes the relative kinematic of two successive rigid bodies. For instance, in our car example, the wheels are attached to the car frame by hinge joint. This allow only one motion from the wheels with respect to the car frame: they can only rotate around a specific axes.

So you can see that with this simple modelisation, we cannot steer our car in a defined direction, has the wheel are only allowed to rotated only their rotation axis. So if we want to allow the control the car direction, we need to add another rotation for the two front steering wheels. This can be done with XDE Physics using a SerialHinge2, a joint that allow the rotation around two predefined axis:

---

1 See LGSM manual for more information.
In our example, the two rotation axis would be the wheel rotation axis and the vertical axis.

Note: Rigid bodies cannot be added to the simulation without specifying a preceding body and a joint. By default, an XDE Physics simulation contains at least one body, the ground body. So the first body you’ll add in the simulation will be connected to the ground body.

### 3.4.1 Joints and Degrees of Freedom

You can see in our multibody systems, the updated car, the wheels position can be described by only six angles: 2 for the rear wheels, 4 for the front steering wheels.

These 6 parameters are called the degrees of freedom. In XDE Physics the degrees of freedom (DoFs) are defined by the joints, depending on their kind.

Note: You can understand the effect of a joint in two ways:

1. The joints parameterize the motion of the rigid bodies, i.e. they specify the number of DoFs and how the bodies move with respect to each other.
2. The joints add constraints on normally freely moving bodies with 6 DoFs (3 translations, 3 rotations), and hence reduce the bodies admissible motion.

So basically there is two kinds of simulators:

1. Those, like XDE Physics, that parameterize the motion of their bodies, hence reducing the simulation total number of DoFs as well as the simulation cost. This method is very efficient but can be difficult to implement.
2. Those who describe all their bodies as body freely moving in 6D (3 translations, 3 rotations), and add kinematic constraints. This leads to simpler implementations, but less efficient code, as you have to solve motion equation for $6 \times n$ DoFs plus $n$ constraints of various dimension.
3.4.2 Fixed Joint

![Diagram of two bodies linked by a fixed joint]

Figure 3.1: Two bodies linked by a fixed joint.

Description

The simplest joint available in XDE Physics is the `FixedJoint`. It adds a rigid body that is fixed with respect to its preceding body. This means that this joint does not add any DoFs to the simulation, as the new body cannot move and there is not need for a parameters to described its relative motion.

Note: Thus, we can also see the `FixedJoint` as a kinematic constraints on all the DoFs of a body with respect to the preceding, i.e. the object cannot move with respect to its preceding body.

Configuration

Being one of the simplest joint, its is also the easiest to configure:

```cpp
xde::gvm::FixedJointRef aFixedJoint = xde::gvm::FixedJointRef::CreateObject("aFixedJoint");
//add aBody to aPrecedingBody using aFixedJoint
mechanicalScene.addRigidBody(aPrecedingBody, aBody, aFixedJoint);
//set the relative position of aBody wrt aPrecedingBody
Eigen::Displacement H_aBody_in_aPrecedingBody(1, 2, 3, 1.0, 0., 0., 0.);
aFixedJoint.configure(H_aBody_in_aPrecedingBody);
```

There is an even simpler methods when you want to add a fixed body whose preceding body is the ground body:

```cpp
//add a body to the scene
mechanicalScene.addFixedRigidBody(aBody);
Eigen::Displacement H_aBody_in_groundBody(1, 2, 3, 1.0, 0., 0., 0.);
//set the body position
//Warning : you can call setPosition on a RigidBody only when
// its preceding body is the groundBody
// and its is connected with a Fixed or Free Joint
aBody.setPosition(H_aBody_in_groundBody);
```
3.4.3 Free Joint

![Free Joint Diagram]

Figure 3.2: Two bodies linked by a free joint.

Description

The **FreeJoint** is another simple joint provided by XDE Physics. As you might have already guessed, this joint let a rigid body moves freely in the 6D space, i.e. the body can freely translate and rotate. This joints adds 6 DoFs to the simulation, 3DoFs for the translation, and 3DoFs for the rotation.

**Note:** The **FreeJoint** adds no kinematic constraint to the body with respect to the preceding.

Configuration

As for the **FixedJoint**, the configuration of the **FreeJoint** is done very naturally:

```cpp
//add aBody to aPrecedingBody using aFreeJoint
mechanicalScene.addRigidBody(aPrecedingBody, aBody, aFreeJoint);
//set the relative position of aBody wrt aPrecedingBody
Eigen::Displacement H_aBody_in_aPrecedingBody(1.,2.,3., 1.0, 0., 0., 0.);
aFreeJoint.configure(H_aBody_in_aPrecedingBody);
```

Similarly with the **FixedJoint**, you can use the code below to quickly add a free body to the simulation:

```cpp
//add a body to the scene
mechanicalScene.addFreeRigidBody(aBody);
Eigen::Displacement H_aBody_in_groundBody(1.,2.,3., 1.0, 0., 0., 0.);
//set the body position
//Warning : you can call setPosition on a RigidBody only when
//its preceding body is the groundBody
// and its is connected with a Fixed or Free Joint
aBody.setPosition(H_aBody_in_groundBody);
```
3.4.4 Prismatic Joint

Description

The PrismaticJoint is a joint that adds only one DoF to the simulation. So the newly added body motion can be described only by one parameter. In the case of the PrismaticJoint, this single parameter describes the translation of the body along a predefined axis.

Using the exponential notation, the position $H_0^i(q_1)$ of a body $i$ connected to a body $j$ with a PrismaticJoint of translation unit axis $u = (u_x, u_y, u_z)$ can be defined as:

$$H_0^i(q_1) = H_0^j * H_i^j * e^{t_p q_1}$$

where:

- $H_0^j$ is the position of body $j$ with respect to the ground body
- $H_i^j$ is the position of body $i$ with respect to the body $j$
- $t_p$ is the screw matrix of twist: $t_p = (0, u)$
- $q_1$ is the joint parameter describing the translation.

Configuration

To correctly configure a PrismaticJoint you have to feed the simulation with the following information:

- The position of the body with respect to the preceding body: $H_i^j$
- The translation unit axis: $(x, y, z)$, with $||(x, y, z)|| = 1.0$
- The initial initial value of the translation $q_1^0$

```c++
xde::gvm::PrismaticJointRef aJoint = xde::gvm::PrismaticJointRef::CreateObject("aJoint");
//add aBody to aPrecedingBody using aJoint
mechanicalScene.addRigidBody(aPrecedingBody, aBody, aJoint);
//set the relative position of aBody wrt aPrecedingBody
Eigen::Displacement H_aBody_in_aPrecedingBody(1, 2, 3, 1.0, 0., 0., 0.);
Eigen::Vector3d axis(1., 0., 0.);
// The default joint position is 0.
```
double q0 = 0.;
aJoint.configure(H_aBody_in_aPrecedingBody, axis, q0);

Note: The PrismaticJoint adds the following kinematic constraint to the body with respect to the preceding:
  • no rotations are authorized.
  • only one translation along a predefined axis is allowed.

3.4.5 Hinge Joint

Description

As for the PrismaticJoint, the HingeJoint is a joint that add only one DoF to the simulation. Again, the newly added body motion can be described by a single parameter. In the case of the HingeJoint, this single parameters describes the rotation around a predefined axis.

Using the exponential notation, the position $H^0_i(q_1)$ of a body $i$ connected to a body $j$ with a HingeJoint of rotation unit axis $u = (u_x,u_y,u_z)$ and rotation center $c_j = (c_x,c_y,c_z)$ expressed with respect to the preceding body can be defined as:

$$H^0_i(q_1) = H^0_j * H^j_i * e^\hat{t}_p q_1$$

where:

• $H^0_j$ is the position of body $j$
• $H^j_i$ is the position of body $i$ with respect to the body $j$
• $\hat{t}_p$ is the screw matrix of twist: $t_p = (u, c \wedge u)$
• $q_1$ is the joint parameter describing the rotation.
Configuration

To correctly configure a HingeJoint you have to feed the simulation with the following information:

- The position of the body with respect to the preceding body: \( H^j_i \)
- The rotation unit axis \( u = (u_x, u_y, u_z) \), with \( ||u|| = 1.0 \)
- The center of rotation \( c_j = (c_x, c_y, c_z) \)
- The initial initial value of the translation \( q_0 \).

```cpp
xde::gvm::PrismaticJointRef aJoint = xde::gvm::HingeJoint::CreateObject("aJoint");
//add aBody to aPrecedingBody using aJoint
mechanicalScene.addRigidBody(aPrecedingBody, aBody, aJoint);
//set the relative position of aBody wrt aPrecedingBody
Eigen::Displacement H_aBody_in_aPrecedingBody(1,,2,3, 1.0, 0., 0., 0.);
Eigen::Vector3d axis(1., 0., 0.);
Eigen::Vector3d center(1., 2., 3.);
// The default joint position is 0.
double q0 = 0.;
aJoint.configure(H_aBody_in_aPrecedingBody, center, axis, q0);
```

Note: The HingeJoint adds the following kinematic constraint to the body with respect to the preceding:

- no translations can occur.
- only one rotation around a predefined axis is allowed.

See tutorial-ref for an introduction to the use of HingeJoint.

### 3.4.6 SerialHinge

![Serial Hinge Joint](image)

Figure 3.5: Two bodies linked by a fixed joint.

Description

In addition to simple and essential joints like the Free, Fixed, hinge, ... joints, XDE Physics provides more complex joints. The SerialHinge, for instance, allows to add two or three HingeJoint in serie between to
rigid bodies. Hence the position $H_i^0(q_1, q_2)$ of the body $i$ with respect to the body $j$ is described by two or three parameters.

$$H_i^0(q_1, q_2) = H_j^0 * H_1^j * e^{e^{-t_p (q_1, q_2)}}$$

where:

- $H_j^0$ is the position of body $j$
- $H_1^j$ is the position of HingeJoint 1 with respect to the body $j$
- $H_2^1$ is the position of HingeJoint 2 with respect to the HingeJoint 1
- $t_p$ is the screw matrix of twist: $t_p = (u, c \wedge u)$
- $q_1$ is the joint parameter describing the rotation.
- $q_2$ is the joint parameter describing the rotation.

**Configuration**

To configure a SerialHinge you just have to configure each HingeJoint it contains:

```cpp
xde::gvm::SerialHinge2Ref aJoint = xde::gvm::SerialHinge2Ref::CreateObject("aJoint");
//add aBody to aPrecedingBody using aJoint
mechanicalScene.addRigidBody(aPrecedingBody, aBody, aJoint);
//set the relative position of aBody wrt aPrecedingBody
Eigen::Displacement H_1_in_j(1, 2, 3, 1.0, 0., 0., 0.);
Eigen::Displacement H_2_in_1(0, 7, 3, 1.0, 0., 0., 0.);
Eigen::Vector3d axis1(1., 0., 0.);
Eigen::Vector3d center1(1., 2., 3.);
Eigen::Vector3d axis2(0., 1., 0.);
Eigen::Vector3d center2(4., 5., 6.);
// The default joint position is 0.
double q1_0 = 0.;
double q2_0 = 0.;
aJoint.getHinge (0).configure (H_1_in_j, center1, axis1, q1_0);
aJoint.getHinge (1).configure (H_2_in_1, center2, axis2, q2_0);
```

### 3.5 Smooth interactions

Once you’ve described your multi bodies system, using RigidBody and Joint, you might want to add additional interaction between the rigid bodies, or even between a cartesian position and the rigid bodies. XDE Physics propose several objects to satisfy this requirement, depending on their nature.

If the interaction needed can be defined as a smooth application, then, you’ll find it in among the Smooth Interactions.

#### 3.5.1 Internal PD Coupling

**Description**

InternalPD Coupling provides a convenient way to control the position of one rigid body using the position of another rigid body. Depending on its configuration, you can chose to control only rotation, translation or both. ..
Figure 3.6: Two bodies attached by an internalPDCoupling with a non null offset. The desired position is rendered in transparent red.

**Tip:** The name InternalPDCoupling stands for the fact that, in opposition with CartesianPDCoupling, the coupling is done internally, *i.e.* between objects in the mechanical scene. As you may have already guessed, PD coupling stands for Proportional Derivative Coupling.

You can also think of the InternalPDCoupling as a damped spring attached to the two rigid bodies. The effort exerted by the spring on the two rigid bodies forces their position. Note that you can specify an offset, or a rest length/orientation. The default rest length and orientation are null.

**Configuration**

To configure an InternalPDCoupling, you need the following data:

- Two rigid bodies
- The proportional and derivative gains, *i.e.* the stiffness and the damping of the damped spring. These gains must be given for translation and rotation.
- An optional offset, or rest length/orientation

```cpp
xde::gvm::InternalPDCouplingRef ipdc = xde::gvm::InternalPDCouplingRef::CreateObject("ipdc");
Eigen::Displacement H_offset(0,7,3, 1.0, 0., 0., 0.);
ipdc.setRigidBody_i(rbi);
ipdc.setRigidBody_j(rbj);

double gainPTranslation = 1.0;
double gainPRotation = 1.0;
double gainDTranslation = 0.1;
double gainDRotation = 0.1;

ipdc.setGainsP(gainPRotation, gainPTranslation);
ipdc.setGainsD(gainDRotation, gainDTranslation);
ipdc.enable();
mechanicalScene.add(ipdc);
```

3.5. Smooth interactions
Warning: The rigid bodies attached to the InternalPDCoupling must have been added to the simulation before adding the InternalPDCouling.

3.5.2 Cartesian PD Coupling

Figure 3.7: A body controlled by a CartesianPDCoupling with a non null offset. The frame is associated with the desired position.

Description

The CartesianPDCoupling can be used to control the position and the velocity of a rigid body. It’s behaviour is somewhat similar to that of an InternalPDCoupling, except that the coupled rigid body is not controlled by another rigid body of the simulation. It is instead controlled externally by simply giving a desired position and velocity.

Tip: As for the InternalPDCoupling, the CartesianPDCoupling controls the coupled rigid body using a Proportional Derivative Coupling.

Again, you can thinks of the CartesianPDCoupling as a damped spring:

- Attached at one end to the coupled rigid body
- Controlled at the other end by the desired position and velocity.

This spring can have a non null rest position.

Configuration

To configure a CartesianPDCoupling, you need the following data:

- One rigid bodies
- The proportional and derivative gains, i.e. the stiffness and the damping of the damped spring. These gains must be given for translation and rotation.
- An optional offset, or rest length/orientation
Obviously, you can update the desired position on velocity at any time:

- The desired position $H^0_{j,\text{des}}$. This position is expressed with respect to the ground body.
- The desired velocity at $t_n$ and $t_{n+1}$. Note that the desired velocity can be given:
  - In the body frame, i.e. the desired velocity is **projected and reduced** at the rigid body origin.
  - In the Koenig frame, i.e. the desired velocity is **reduced at the rigid body origin** and **projected in the ground body frame**.

```cpp
xde::gvm::CartesianPDCouplingRef cpdc = xde::gvm::CartesianPDCouplingRef::CreateObject("cpdc");
Eigen::Displacement H_offset(0,7,3, 1.0, 0., 0., 0.);
ipdc.setCoupledRigidBody(rbj);

double gainPTranslation = 1.0;
double gainPRotation = 1.0;
double gainDTranslation = 0.1;
double gainDRotation = 0.1;

cpdc.setGainsP(gainPRotation, gainPTranslation);
cpdc.setGainsD(gainDRotation, gainDTranslation);
cpdc.enable();
mechanicalScene.add(cpdc);

Eigen::Displacement H_n(0,7,3, 1.0, 0., 0., 0.);
Eigen::Displacement H_np1(0,7,3, 1.1, 0., 0., 0.);
cpdc.setDesiredPosition(H_n, H_np1);

Eigen::Twistd T_n(0., 0.,0., 1.0, 0., 0.);
Eigen::Twistd T_np1(0., 0.,0., 1.0, 0., 0.);
cpdc.setDesiredVelocityInBodyFrame(T_n, T_np1);
```

**Warning:** The rigid body attached to the **CartesianPD Coupling** must have been added to the simulation before adding the **CartesianPD Coupling**.

### 3.5.3 External Wrench

**Description**

To control the rigid body position and velocity, the two previous smooth interactions were computing forces based on a damped spring model. This is the standard way to control a rigid body using a Proportional Derivative Coupling. But you might need to control a body in a different way, applying forces that you have computed yourself. The **ExtWrench** object offers this functionality..

**Configuration**

Configuring an **ExtWrench** is straightforward:

- Set the application rigid body,
- Set the applied wrench. As for the **CartesianPD Coupling**, the wrench can be defined in the body frame or the Koenig frame.
It is also possible to provide the position derivative, \textit{i.e.} the stiffness of the applied forces, as well as the velocity derivative, \textit{i.e.} the damping. In this case, the integrator will use these information to integrate implicitly.

```cpp
xde::gvm::ExtWrench extWrench = xde::gvm::ExtWrench::CreateObject("extWrench");
Eigen::Displacement H_offset(0,7,3, 1.0, 0., 0., 0.);
extWrench.setApplicationRigidBody(rbj);

Eigen::Wrenchd wrench(0., 1., 0., 0., 0., 1.);
Eigen::Matrix<double, 6, 6> gammaDiff;
Eigen::Matrix<double, 6, 6> B;

extWrench.setModeToBodyFrame();
extWrench.setWrench(wrench);
extWrench.setWrenchDiff();
extWrench.setWrenchB();
extWrench.enableImplicitIntegration();
extWrench.enable();
mechanicalScene.add(extWrench);
```

### 3.6 Constraints

In the beginning of section \textit{Joints}, we introduce the \textbf{FixedJoint}. This joint constraints the relative motion of two rigid bodies by simply setting the number of DOFs between theses two bodies to zero.

However, when you want to create a multi body systems with a closed loop, as illustrated below:
you cannot use, for numerical reasons, a FixedJoint to close the loop. You have to use a CutJoint Instead.

### 3.6.1 CutJoint

**Description**

The CutJoint is completely different from the other joint you have already discovered. Instead of using a specific parametrization to describe the relative motion between two successive rigid bodies, the CutJoint adds constraints to prevent relative motion. ..

This has two important consequences:

<table>
<thead>
<tr>
<th>Warning:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The two rigid bodies connected by the CutJoint will keep the same DOFs. Using a FreeJoint would have reduced the number of DOFs of one of the bodies to zero.</td>
</tr>
<tr>
<td>• 6 additional constraints (3 for translation, 3 for rotation) are added to the solver.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding a CutJoint doesn’t change the kinematic graph of a multi body system. It simply adds constraints.</td>
</tr>
</tbody>
</table>
Configuring a **CutJoint** is very easy. You will need the following data:

- Two rigid bodies
- The position of the body \( j \) with respect to the body \( i \): \( H_{j_i} \).

```cpp
xde::gvm::CutJoint cutJoint = xde::gvm::CutJoint::CreateObject("cutJoint");

cutJoint.setRigidBody_i(rbi);
cutJoint.setRigidBody_j(rbj);

Eigen::Displacement H_j_i(0,7,3, 1.0, 0., 0., 0.);
cutJoint.configure(H_j_i);

cutJoint.enable();
mechanicalScene.add(cutJoint);
```

**Note:** XDE Physics offers three additional methods to ease the configuration of **CutJoint**:

1. `configureToIdentity()`: This simply sets \( H_{j_i} = I \).
2. `configureFromCurrentRigidBodyPositions()`: This uses the current rigid body positions to compute \( H_{j_i} \).
3. `configureFromRigidBodyPositions(Eigen::Displacement const& H_i_0, Eigen::Displacement const& H_j_0)`: This uses the given rigid body positions to compute \( H_{j_i} \).

### 3.7 Collision handling

See *Collision detection*.

### 3.8 Friction models

See *Handling Contact*

### 3.9 A complex example

```cpp
xde::gvm::CutJointRef.
```

This is for latex only!
COLLISION DETECTION

One of the key features of XDE Physics lies in its precise collision detection. This functionality involves two steps:

1. A collision detection step.
2. A contact solving step.
CHAPTER FIVE

HANDLING CONTACT

XDE Physics provide various friction models, to help you add more physical realism in your simulation.

5.1 Contact Laws provided by XDE Physics

5.1.1 Signorini Contact Law

![Signorini Law](image)

Figure 5.1: This figure represents two freely moving spheres. The right one is in contact with the ground and have a strictly positive contact force. The left one is above the ground, i.e. it is at a strictly positive distance from the ground. Hence its contact force is null.

**Description**

The simplest one uses the Signorini Law. This law can be stated as follows:

- **If contact occurs between two objects:**
  - the distance between the two objects at the contact point is zero: \( d_c = 0 \)
  - the force between the two objects at the contact point is strictly positive: \( f_c > 0 \), and is oriented along the surface normal at the contact point.

- **If no contact occurs between two objects:**
the distance between the two objects at the contact point is strictly positive: \( d_c > 0 \)

the force between the two objects at the contact point is null: \( f_c = 0 \)

As you can see in this definition, the contact force is oriented along the surface normal. There is no tangential generated by the contact, so no friction occurs when the Signorini law is used.

Tip: This law doesn’t account for friction. So use this law when you want to simulate object with sliding contact.

Performances

XDE Physics considers contact points as constraints. Hence, each Signorini contact point adds exactly one constraint to the mechanical solver. This obviously impact performances.

5.1.2 Coulomb Law

![Coulomb Law](image)

Figure 5.2: This figure represents freely moving spheres with different friction coefficient with the ground. The left sphere has a larger friction coefficient than the right one. As the contact forces of the left sphere is inside the friction cone, this sphere will roll without sliding. On the opposite, the contact force of the second sphere, having a smaller friction coefficient, will slide without rolling.

Description

The Coulomb Law simply states that:

- If the contact force is inside the friction cone, there is no relative motion between two objects.
- If the contact force is on the boundary of the friction cone, there is a sliding motion between two objects, and the friction force is directed in the opposite direction of the movement.

Note: The friction cone angle is given by \( \tan(\alpha) = \mu \).

This law can also been expresses as:

- No motion occurs if the norm of the tangential force \( f_t \) is inside the disk of radius \( r = \mu \cdot f_n \) where \( \mu \) is the friction coefficient, and \( f_n \) the contact force along the normal, i.e. \( |f_t| < \mu \cdot f_n \).
• Sliding occurs when the norm of the tangential force \( f_t \) is equal to \( r = \mu f_n \), i.e. \( |f_t| = \mu f_n \)

**Performances**

When a frictional contact point appears in an XDE Physics simulation, 3 constraints are added to the mechanical solvers:

• One for the normal contact force
• Two for the frictional tangent forces

### 5.1.3 Coulomb Contensou Law

![Contensou Ellipsoid](image)

Figure 5.3: This figure represents the Contensou ellipsoid. In the Coulomb law, the 2D friction force was included in a disk. For the Contensou law, the friction force being a 3D force (2D tangential force, 1D torque), the friction force is included in an ellipsoid. The base of this ellipsoid is a disk of radius \( r_c = \mu f_n \). The height is given by \( \frac{2}{3} r_c \times R \), where \( R \) is the radius of the surface involved in the frictional torque.

**Description**

The Coulomb-Contensou friction Law adds rotating friction to the Coulomb Law. In addition to the 2D tangential friction force opposed to the motion, the Coulomb-Contensou Law defines a 1D friction torque opposed to the rotation at the contact point. The link between the 1D normal force, 2D tangential friction force and the 1D rotation force is described by a slightly smashed ellipsoid. In XDE Physics, we simplify the Coulomb-Contensou Law using an ellipsoid. With this simplification, this law simply states that:

• No motion occurs if the friction force is inside the friction ellipsoid.
• Sliding occurs when friction force is on the boundary of the friction ellipsoid.

As explained in the figure above, the ellipsoid geometry is defined by two parameters:

• \( \mu \), which defines the base radius \( r_c = \mu f_n \)
• \( R \), with defines the radius of the surface contact involved in the rotational friction. The bigger \( R \) is, the larger is the rotational friction.
Figure 5.4: In this picture you can see two contact points handled by the Coulomb Contensou Law. The left one has an large surface contact for the rotational friction. The right one as a smaller contact surface.

Figure 5.5: This figure represents the 4 possible cases of friction for the Contensou law:

1. The contact force is inside the ellipsoid. There is no motion
2. The contact force is on the boundary of the ellipsoid and there is no tangential friction force: only rotational motion occurs
3. The contact force is on the boundary of the ellipsoid and there is no friction torque: only translational motion occurs
4. The contact force is on the boundary of the ellipsoid and there is friction torque as well as a tangential friction force: a mixed translational and rotational motion occurs
Note: Note that there is less tangential friction when there is a friction torque! This physical phenomenon is used in cleaning machines like rotative scrubbers: it is easier to push the machine when it’s rotating.

Performances

When a Contensou contact point appears in an XDE Physics simulation, 4 constraints are added to the mechanical solvers:

- One for the normal contact force
- Two for the frictional tangent forces
- One for the frictional torque

5.1.4 Penalty based contact

Description

All the contact law presented above are solved very precisely using constraint-based methods. This brings a high level of accuracy to XDE Physics simulation. However, for some applications, performance is more important than precision. This is usually the case for haptic simulation.

In these cases, it might be a better option to use penalty-based methods to handle contact. XDE Physics also supports this kind of method for contact without friction, i.e. for the Signorini law.

Warning: Penalty-based methods do not guarantee non-interpenetration between objects. They can also introduce instability, as they add stiffness to the system.

Performances

As penalty-based methods do not add constraints to the systems, they include a significant speed up of the simulation.

5.2 Configuring XDE Physics for contact handling

5.2.1 Choosing the right solver

As we have seen in the Quick Start section Quick Start, just after having created a xde::gvm::SceneRef, we need to set the integrator flags. These flags define not only the kind of simulation that we want to perform, but also the solver used for contacts.

Regarding contact, there is only two options:

1. We don’t want to simulate friction, so we use the NO_FRICTION_* flags.
2. We want to simulate friction; so we use the FRICTION_* flags.
5.2.2 Setting the contact restitution mode

There is currently two contact restitution mode:

1. The **Non Smooth Mode**, where the various contact law presented above are solved using a constraints solver.
2. The **Penalty Mode**, which support only the *Signorini* Law, and that solves the constraints using penalty based method.

**Using the Non Smooth Mode**

This mode is very accurate, and allow the handling of friction, but is less fast than the **Penalty** mode. This mode doesn’t need you to set some parameters. You can just have control on the number of iteration done by the constraint solver with the method `xde::gvm::SceneRef::setGaussSeidelMaxIter()`.

This mode is enable by default. To enable this mode, call `xde::gvm::SceneRef::setContactRestitutionMode()`, with the argument `xde::gvm::ContactRestitutionMode::NonSmooth`.

**Using the Penalty Mode**

This mode is very fast, but is less accurate than the **Non Smooth Mode**, and requires the tedious setting of the penalty stiffness. Moreover, this method add stiffness to the modelised system, and might leads to numerical instabilities.

The stiffness settings is done calling `xde::gvm::SceneRef::setContactPenaltyStiffness()`. A standard value for this stiffness is $2 \times 10^6$.

To enable this mode, call `xde::gvm::SceneRef::setContactRestitutionMode()`, with the argument `xde::gvm::ContactRestitutionMode::Penalty`.

5.2.3 Contact Materials

Friction between two objects depends on the nature of their constitutive materials. For instance, the friction of a rubber ball on a glass ground is very different from the friction of a glass ball on an ice ground. So friction properties must be defined for a pair of material.

**Description**

In XDE Physics, contact materials are just defined by a name, i.e. a simple c++ string. They don’t have physical properties. Each **RigidBody** has a contact material.

On the opposite, pairs of contact material have physical properties. These physical properties are represented by `xde::gvm::ContactLawDesc`. Depending on the contact law chosen, the physical properties can be:

- nothing, if the *Signorini* law is chosen
- a friction coefficient, when using the *Coulomb* law.
- a friction coefficient and a contact radius, when using the *Coulomb-Contensou* law.

**Configuration**

By default, each rigid body is associated to a predefined contact material whose contact law is the *Signorini* law. To change this default material, we first need to change the contact material of the rigid bodies:
aRigidBody.setContactMaterial("rubber");
anotherRigidBody.setContactMaterial("copper");

Then, we have to specify the contact law associated with this pair of contact material:

```cpp
dx::gvm::ContactLawDesc copperRubberLaw(dx::gvm::ContactLaw::Coulomb, 0.8);
mechanicalScene.setContactLawForMaterialPair("rubber", "copper", copperRubberLaw);
```

And that’s it. When `aRigidBody` and `anotherRigidBody` are in collision, the contact is handled using the *Coulomb* law with a friction coefficient of 0.8.

### Getting contact information

XDE Physics provide an easy access to the following contact information:

- contact point position
- contact point normal
- contact normal force
- contact friction force

These information are available through the `dx::gvm::ContactAccessor` and `dx::gvm::PointContactData` classes.

```cpp
dx::gvm::ContactAccessor contactAccessor(mechanicalScene);
std::vector<PointContactData> pointContacts;
contactAccessor.getContactPoints(aBody, anotherBody, pointContacts);
for (std::vector<PointContactData>::iterator cIt = pointContacts.begin() ; cIt != pointContacts.end() ; cIt++)
    std::cout << cIt->ai << std::endl;
```
CHAPTER

SIX

BEAMS SIMULATION
CHAPTER
SEVEN

CONTROLLING ROBOTS
CHAPTER EIGHT

TUNING YOUR SIMULATION TO IMPROVE PERFORMANCES