MathEngine Karma™ Dynamics

Developer Guide

MathEngine Karma Dynamics. Developer Guide.

Contents

Preface

1 Introduction to Karma Dynamics

2 Getting Started

3 Constraints: Joints and Contacts

4 Advanced Features, Optimization and Utilities

5 Constructing Good Simulations

Preface

About the Karma Dynamics Guide

This guide explains how to use Karma Dynamics to add realistic, complex physical behavior to real-time 3D environments. Karma Dynamics is written in C and hand tuned assembler. The provided source is in C with a C API. Karma is available for:

- The Sony PlayStation2 games console.
- Xbox as a beta to paying customers.
- Single precision Win32 built against the Microsoft LIBC, LIBCMT or MSVCRT libraries.
- Linux on request.

The single precision Win32 build of Karma against MSVCRT is provided for evaluation.

The Kea Solver, that is the core component of Karma Dynamics, is included.

This manual is aimed at developers of real-time entertainment simulation software, familiar with the following:

- The C programming language. Knowledge of Microsoft Visual C++ is an asset.
- Basic mathematical concepts.

Accompanying Documentation

Detailed information about each function is provided in the HTML Karma Dynamics Reference Manual that can be found by following the ‘Demos and Manuals’ hyperlink in the index.html file in the metoolkit directory.

Related Software

Karma Viewer

Karma Viewer is a basic cross platform wrapper around the GLUT and Direct 3D libraries. While this enables developers to build 3D applications with simple scenes, it is not meant to replace the chosen rendering tool. Rather the developer should hook Karma up to the renderer they are using. Some basic performance monitoring tools are provided. The Viewer is documented in the MathEngine Karma Viewer Developer Guide.

Karma Collision

Karma Collision is a collision detection package. It provides the contact information required to produce real-time, geometrically-realistic collisions between 3D models. Karma Collision can be used with Karma Dynamics or on its own. The following documentation discusses Karma Collision:

- *MathEngine Karma Collision. Developer Guide*.
- *MathEngine Karma Collision. Reference Manual*.

Karma Simulation

Karma Simulation (Mst Library) provides an API that bridges Karma Dynamics (Mdt Library) and Karma Collision (Mcd Library). The Mst Library simplifies the control of dynamics and collision by providing tools and utilities in an integrated programming environment. The following documentation discusses Karma Simulation:

- *MathEngine Karma Simulation Toolkit. Developer Guide*.
- *MathEngine Karma Simulation Toolkit. Reference Manual*.

About MathEngine

MathEngine is the provider of natural behavior technology for leading-edge developers committed to injecting life into 3D simulations and applications. Founded in Oxford, England in 1997 and staffed by a team of physicists, mathematicians and programmers, MathEngine provides tools that give software developers the ability to add natural behavior to applications for use in the games and entertainment markets.

Contacting MathEngine

Head Office

MathEngine plc, 60, St. Aldates, Oxford, UK, OX1 1ST.

Tel.+44 (0)1865 799400 Fax +44 (0)1865 799401

Web Site

www.mathengine.com

Customer Technical Support

support@mathengine.com

General Enquiries

sales@mathengine.com

Conventions

Units

There is no built-in system of units in Karma, which is not to say that quantities are dimensionless. Any system of units may be chosen, either meter-kilogram-seconds, centimeter-grams-seconds or foot-pound-seconds. However, the developer is responsible for the consistency of values and dimensions used. This analysis becomes important when tuning an application, or changing several parameters simultaneously.

Coordinate Systems and Reference Frames

Karma uses rectangular Cartesian coordinates for virtually all vector quantities. Karma does not use polar coordinates. The components of any 3-dimensional vector are the projections along the x, y, and z axes. Those axes are orthogonal to each other and the scale along each one is identical. Right-handed coordinate systems are used everywhere except in the Karma Viewer.

The reference frame used most is called alternately the world, global, inertial or Newtonian reference frame. These are all names for the same thing: a fixed reference frame which defines the overall orientation and position of the scene being worked on. This is the common frame of reference for all the objects in an MdtWorld or an MstUniverse.

There are other useful reference frames however. Each rigid body has defined a reference frame centered at its center of mass. Points on a rigid body, such as the attachment positions for joints, are naturally expressed in terms of a fixed location on the rigid body. Note that the joint position is given in the world reference frame in Karma. As the rigid body moves under the influence of applied forces, that attachment point will move as seen from the origin of the world reference frame.

Since a 3D model is designed independently of mass properties of the objects it is based on, an artist is likely to pick a convenient origin that is not the center of mass of the object. This reference frame is called the model reference frame. A relative transformation between this reference frame and the rigid body center of mass reference frame can be used to simplify manipulation of geometric models.

The relationship between reference frames consists of a translation vector and a 3 by 3 orthonormal rotation matrix. If the translation vector is X and the rotation matrix is R, then, a vector x with coordinates expressed in the non-inertial frame becomes the vector y in the world frame with: $y = X + Rx$.

This convention is not universal, hence care should be taken when comparing formulae with standard texts on mechanics and rigid body dynamics. As is customary in well-known 3D graphics API's, a transformation between reference frames is conveniently stored in a 4 by 4 affine matrix. The MeMatrix4 and MeVector4 data types implement matrices and vectors for use in this way. In

this representation, regular 3D vectors are represented as the first three components of the four dimensional vectors. The last element in the 4 dimenstional vector must be set to 1 if the translation is to be taken into account and 0 if one is only interested in the rotation. The convention used follows the one used in the graphics literature, OpenGL in particular.

Newcomers to this sort of algebra may like to consult a textbook on 3D graphics to familiarize themselves with the concepts and notations. Utility functions are provided to convert vectors from one reference frame to another.

Type Conventions

Karma Dynamics uses some special type definitions and macros that make it more portable.

For example:

- `MeReal`: floating point numbers.
- `MeVector3`: a vector of 3 `MeReal`s.
- `MeVector4`: a vector of 4 `MeReal`s.
- `MeMatrix3`: A 3x3 matrix of `MeReal`s.
- `MeMatrix4`: A 4x4 matrix of `MeReal`s.
- `MeSqrt()`: `sqrt()`

These and others are defined in `MePrecision.h`.

Typographical Conventions

Bold Face indicates:	• UI element names (except for the standard OK and Cancel) • Commands
`Courier indicates:`	• Program code • Directory and file names
Italics indicates:	• Document and book titles • Cross-references • Introduction of a new word or a new concept

Naming Conventions for C Identifiers

`Me`	MathEngine types and macros for controlling precision.

`Mdt`	Karma Dynamics
`MdtBcl`	Basic Constraint Library
`MdtKea`	Kea Solver
`Mcd`	Karma Collision
`Mst`	Karma Simulation Toolkit
`R`	Karma Viewer

Chapter 1 • Introduction to Karma Dynamics

Overview

Karma Dynamics can be used to add believable, realistic, complex physical behavior to real-time 3D environments, enabling developers to produce imaginative simulations quickly and easily without needing to know about the complex details of the underlying mathematics and physics. Simulations can be optimized to provide the most realism and speed out of the available hardware and software.

Karma Dynamics is an integrated suite of dynamics simulation tools that includes:

- Libraries and header files.
- Sample programs and demo programs with source code.
- Implementations of high-level scene API with source code.
- Documentation.

This chapter summarizes the features of Karma Dynamics.

How A Game Uses Karma Dynamics

The following diagram shows how a game or other application uses Karma Dynamics and Karma Collision:

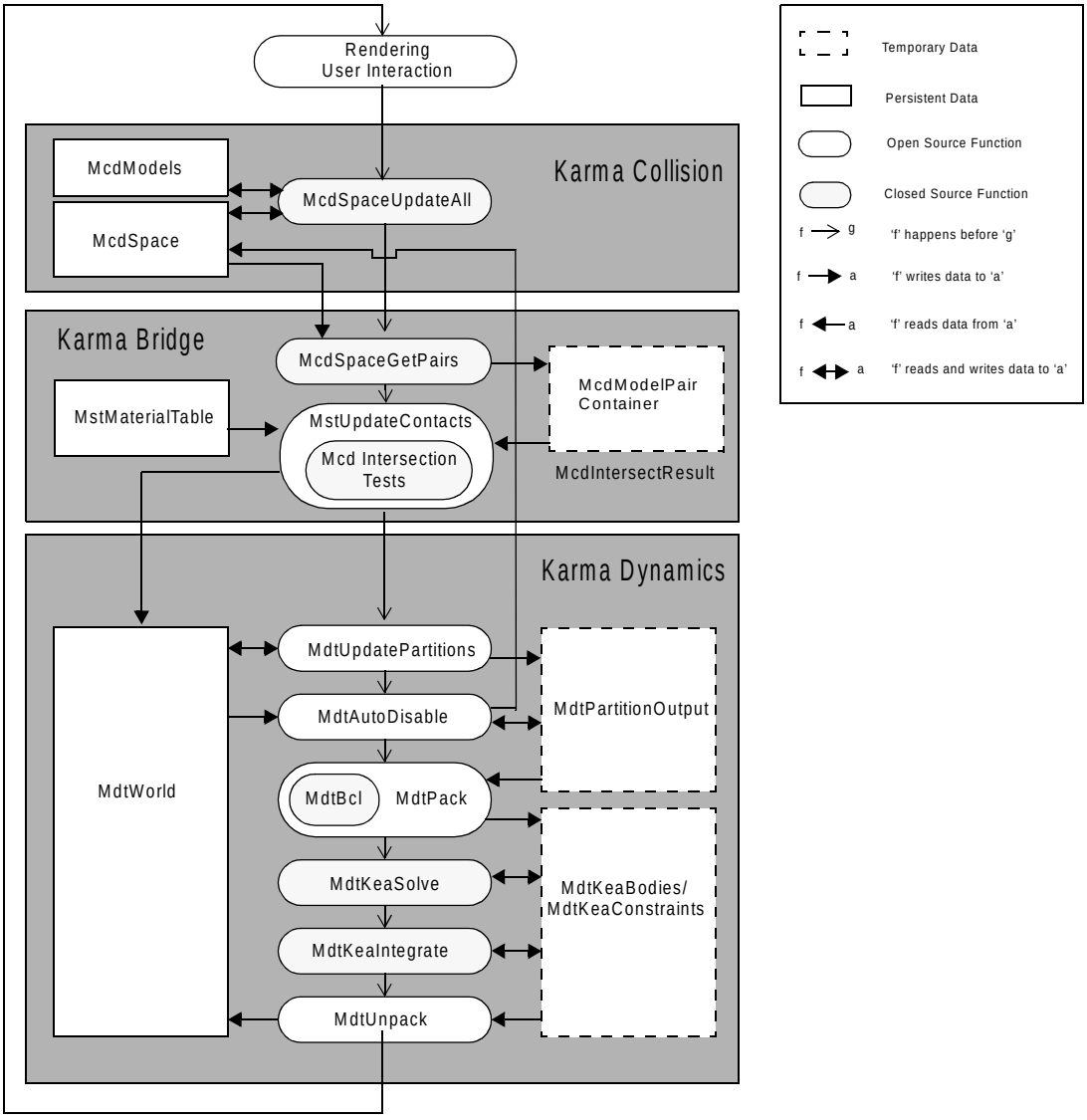


Figure 1*: Karma Architecture*

The Mdt Library

The Mdt Library contains functions that use physical data to simulate what would happen if, for example, a car crashed into a stack of barrels, a human walked across a rope bridge, or if one bipedal robot tripped another. Real world information (e.g. mass, friction, gravity) is contained within the data structures used in Karma to represent the simulated system of bodies.

The Mdt Library can be used to create rigid bodies and articulated bodies corresponding to the models (people, robots, vehicles, and other objects) used in games or other 3D simulations. The abstract space occupied by these bodies is called a *world*. This world typically corresponds to the "scene" described by the rendering software and to the *space* occupied by any collision models.

The world is a collection of data structures. More than one world can be used in an application, perhaps corresponding to different levels in a game. Different worlds are totally disconnected from each other and may never interact. Any interaction would be handled by the developer. A world may be partitioned into smaller groups of interacting rigid bodies. Rigid bodies can move from partition to partition.

A world may contain rigid bodies that interact via forces, constraints, and input and output signals. The bodies may have geometric properties assigned to them that can be used to determine the interaction mechanism. An input signal enables the user to interact with objects in the world. By reading the:

- (x, y) coordinates of the mouse pointer,
- angle signal from a joystick,
- or input from a steering wheel or special pedal,

the user can reposition or apply forces to scene objects.

Similarly, an output signal could be the orientation of one of the scene objects that could be sent to a force feedback device or a motion platform.

A constraint restricts the motion of a rigid body. There are two types of contacts (Chap 3):

- A *joint* between two bodies, such as a hinge to represent the elbow of a virtual human, is a constraint. This constraint restricts at least one of the six degrees of freedom of the attached pair of bodies (or a body attached directly to the world). The six degrees of freedom comprise three linear and three rotational degrees of freedom.
- A *contact* constraint is a type of constraint that prevents objects inter-penetrating. No permanent restriction is placed on any of the six degrees of freedom of the contacting objects.

Rigid bodies, and the forces that act on these bodies are the subject of this chapter.

Relationship to Other Libraries

The Mdt Library is the Karma Dynamics top level library. If the Mdt Library is used to build a simulation, then the lower level MdtBcl Constraint Library does not need to be called. Note that MdtBcl must be linked in though. The source code of the Mdt Library can be used as a guide as to how to use the lower level libraries. For developers using Karma Collision in addition to Karma Dynamics, the Mst library provides tools to integrate both of these into a simulation library.

Designing Efficient Simulations Using the Mdt Source Code

Karma Dynamics is distributed with full source code for the Mdt Library. Hence applications can be optimized by, for example:

- Selecting only the required parts of the Mdt Library.
- Customising the Mdt code.
- Creating new joint types, based upon Mdt joints.

Designing Efficient Simulations Using the Mdt Library

There are lots of ways of making your simulation go faster using the provided library functions. For example:

- Choice of friction model.
- Partitioning of objects in a world.
- Disabling bodies.
- Keeping the matrix size down.
- Level of detail representations.

Rigid Bodies: A Simplified Description of Reality

All the objects in the real world have finite extent and are therefore not well simulated using the 'point mass model' of classical physics. The rigid body is an idealisation of objects that do not deform easily. It represents objects that have finite extent but that never deform at all; any two points on the objects are always exactly the same distance apart, no matter what forces and stresses are applied to the body. In the real Newtonian world, objects are not infinitely hard or rigid since they all deform to some extent - even diamond. Having said that, infinite rigidity provides a good approximation when modelling the dynamics of solid objects. The assumption of infinite hardness allows faster simulation speeds to be realised, and hence is used by Karma.

Rigid bodies have fundamental physical properties such as mass and inertia tensors as well as extra physical properties. The rigid body model used by Karma uses real world physical properties such as mass and mass distribution. Friction and resitution are properties that describes how bodies in contact interact. Karma does not model all physical phenomena directly and properties such as viscous drag coefficient (related to the geometry of the body), electric charge, magnetic dipole etc. can be modelled by the developer to introduce more interesting behavior based on the existing API.

Rigid bodies have kinematic attributes that describe their position and movement:

- Position of the center of mass.
- Orientation of the body.
- Velocity of the center of mass.
- Angular velocity that describes the rate of change of orientation.

Rigid bodies have dynamic attributes that consist of net applied forces and torques.

These properties can be set to appropriate values using Karma Dynamics, and objects moved by the application of forces such as gravitational field strength, and impact with another object.

A brief description of the program flow is:

- Set the initial (at time, T = 0) world, object and joint properties.
- Specify any forces or contacts at T = 0.
- Set the simulation time step, T_S, i.e. the time amount by which the simulation should be repeatedly incremented. A typical value might be $T_S = 0.01$ seconds.
- Karma Dynamics then calculates the new positions at $(T + T_S)$ and the other dynamical properties for each body in the world according to the underlying newtonian physical model. This is called *solving*. The Karma Dynamics solver is called Kea.
- The positions, orientation and so forth of the models in the scene are updated, based on the data returned by Karma Dynamics. This renderer updates the scene from this information.
- Calculate the next position etc at T + 2Ts.
- etc

Joint Constraints and Articulated Bodies

Articulated bodies are rigid bodies connected by joint constraints. A body representing a forearm can be connected to a body representing an upper arm by an ‘elbow’ hinge joint. But joints don’t just connect bodies. They also *constrain* the motion of rigid bodies: the forearm cannot be pulled far without moving the upper arm. A hinge joint can have limited rotational movement - a real human elbow cannot move more than about 160 degrees without breaking.

The Mdt library includes models of, for example, hinges, ball and socket, universal, and other joints. Hinge and prismatic joints have *limits* so that simple models of real-world behavior can be created: a hinge can be limited, for example, so that it only opens 270 degrees.

Some joints have *soft limits*, that give ‘bouncy’ effects.

Other joints are motorized, allowing control of the movement of the bodies. Motorized joints with power limits provide stable modeling of, for example, engines, brakes, motors, and stiff springs.

Contact Constraints and Collision Detection

In a 3D simulation, models are likely to come into contact with each other. The point or points where two bodies intersect is a *contact*. A contact limits how bodies can move, and is therefore a type of constraint. The Mdt library includes high-level representations of *contact constraints*.

With either Karma Collision, or in-house collision code, it can be determined whether two bodies are in contact. To do this, create and populate a contact data structure, i.e. set the position of the point of contact as well as the unit normal vector in the direction perpendicular to the local contact plane - Karma Collision does this automatically.

Karma Dynamics then uses this contact information to model the behavior of the two bodies. For example, if a body representing a stone falls onto the floor, Karma Dynamics will detect the collision and apply the appropriate impulse so that the stone stops when it strikes the floor. The amount of rebound of the stone can be adjusted by varying the restitution between the stone and the floor.

Working With Karma Collision

When using both Karma Collision and Karma Dynamics, the Simulation Toolkit (Mst Library) may be used to handles operations and communication between collision and dynamics.

To use one of the lower-level libraries in Karma Dynamics, the devloper will need to write custom code to handle communication with Karma Collision in addition to Karma Dynamics.

MdtWorld

An `MdtWorld` structure, often called a *world*, is a container that tracks the `MdtBody` structures along with their joint and contact constraints. It also stores world (global) properties such as the gravitational field strength.

Partitioning the World

Physical simulation becomes more computationally expensive as the number of interacting objects increases. If there are ten objects in a world that are all constrained together, the scene evolution is more difficult to simulate than when there are ten uncoupled objects because of all the interactions between the linked sturcture. Speed can be gained by separating the world, where appropriate, into groups of objects that interacting among themselves, but with each group interacting little or not at all with the other groups.

The Mdt layer includes an optional partitioning feature that analyses the constraints in the world to produce lists of rigid bodies; all the rigid bodies within a list are constrained to at least one other in the same list and unconstrained to any rigid body in any of the other lists. The lists are interchangeably called either islands or partitions.

Externally applied forces may be ignored during partitioning. They affect the contact forces generated in linked systems, but do not increase the complexity of the solving, like the joining of objects to a system would.

Disabling Bodies

There is no need to calculate the behavior of a stationary body, unless it is struck by another body. The Mdt Library will automatically *disable* stationary bodies, removing them from the list of bodies passed to the solver. This reduces the computational time that the solver needs to update the dynamic properties of the bodies in the world.

To prevent a particular body from moving it is not enough to disable it unless custom partitioning is introduced and Mdt's auto partition features disabled. Any rigid body registered in a world will come to life when other rigid bodies collide with it or are constrained to it. Objects that are not moving and that have a very small acceleration will be disabled. This usually means that the forces acting on that rigid body have come near to equilibrium.

Bodies that never move at all, such as buildings, don’t have any dynamics properties associated with them. They are simply collision models that are part of the world and are not included in the evolution of the dynamic system. When dynamic objects collide with fixed objects, it is enough to supply the appropriate contact information that can be generated with Karma Collision.

Chapter 2 • Getting Started

Overview

In this chapter code samples show how to simulate:

- Rigid bodies using Karma Dynamics.
- What happens when a rigid body comes into contact with something.

The Mdt Library: Getting Started

In this chapter the basic procedure that must be followed in any program that uses Karma Dynamics is described.

The code extracts below are based on the tutorial `drop.c`, that can be found in \metoolkits\src\tutorials\Drop. This is a straightforward tutorial that demonstrates how to use Karma Dynamics to simulate a ball falling freely in an earth strength gravitational field of $9.8Nkg^{-1}$.

Rendering Solution

All of the provided examples and tutorials, except for GreaseMonkey, use the Karma Viewer. Developer applications may be driven by the requirements of the rendering software used.

> **Note:** For any applications using the Karma Viewer, press F1 to bring up application-specific Help.

Stepping Through the Simulation Using Callback Functions

In the `drop.c` tutorial program, `drop.c` does not call `MdtWorldStep()` directly to evolve the simulation by one time-step. A callback function void Tick(RRender *rc) is defined and called from main through the render callback function call RRun(rc, Tick, 0). The world may be dynamically evolved in this callback.

```
void Tick(RRender * rc)
{
...
... /* Dynamics code here */
...

}
```

When the program is terminated, the callback function void cleanup(void) is called to free the allocated memory. This must be used because some systems do not exit from the evolve loop and execute the remaining code. Please refer to the Karma Viewer Guide for further information.

```
void cleanup(void)
{
...
... /* Cleanup code here. This includes dynamics cleanup code.*/
...
}
```

A callback function must be used to clean up because the Viewer terminates the program itself.

Initializing the World

Here, we create (initialize) the world that will contain the simulation:

```
MdtWorldID world = MdtWorldCreate(1, 0);
```

The Mdt Library uses a world as a framework for a simulation. A simulation can use one or more worlds, each of which may use one or more solvers, where each solver has its own integrator and collision algorithms.

> **Note:** In this release of Karma Dynamics, only one Kea solver is supported: the MdtKea Library (Kea Solver).

A world is a `MdtWorld` structure that contains all the properties of the world to be created. Many of these properties are used by the Mdt Library to determine whether a body is *enabled* (used by the solver) or *disabled* (not used by the solver).

The function `MdtWorldCreate` creates a world and returns a `MdtWorldID` identifier that points to a `MdtWorld` structure that identifies the world that has been created.

```
MdtWorldID MEAPI MdtWorldCreate ( const unsigned int maxBodies,
                                  const unsigned int maxConstraints );
```

Where `maxBodies` and `maxConstraints` represent the maximum number of bodies and constraints in the world.

Many functions are used to manage world properties. Most (but not all) are prefixed by `MdtWorld`.

Setting the Timestep for the Simulation

First the time step - that is the amount of time that will elapse between states of the simulation - needs setting.

```
/* Simulation time step in seconds */
MeReal step = (MeReal)(0.03);
```

The floating-point constant is cast to `MeReal`, one of many types defined in the MathEngine Definitions and Tools Library, that are documented in the *MathEngine Karma Documentation Set*. The underlying type for `MeReal` can change from single to double precision, depending on how it is defined.

The time increment can be set to a value to suit the complexity of the simulation and the operating platform . For further information, see: *Stepping Through Time* on page 17.

Setting the Gravity of the World

The gravity (gravitational field strength) for this virtual world may be set to that at the earth surface using:

```
void MEAPI MdtWorldSetGravity ( const MdtWorldID world,
                  const MeReal gx, const MeReal gy, const MeReal gz);
```

Where gx, gy and gz are the components of the gravity vector.

that is:

```
MdtWorldSetGravity(world, 0, -(MeReal)(9.8), 0);
```

In the real world, where the plane on which objects rest (e.g., a floor) can be conveniently described by the x and z axis, the gravity would be set only along the negative y axis to earth's gravity value, such that `gx=gz=0` and `gy=-9.8`. Usually, gravity is be set to act in the "down" direction, but it may be set along any vector direction and it's magnitude changed.

Defining a Body

The next step is to create a body and place it in the world. Typically, a body corresponds to a rendered graphical object such as a sphere, cube, or more complex shape:

```
MdtBodyID body;
...
body = MdtBodyCreate(world);
```

A body is represented internally by a structure storing the following physical properties.

`MdtBody` **Members**	**Default Value**
mass	`1`
moment of inertia	`{{0.4,0,0},{0,0.4,0},{0,0,0.4}}`
position	`{0,0,0}`
quaternion	`{1,0,0,0}`
transformation matrix	`{{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}}`
fast spin axis	`{0,1,0} (fast spin is disabled by default)`
force applied	`{0,0,0}`
torque applied	`{0,0,0}`

`MdtBody` **Members**	**Default Value**
velocity	`{0,0,0}`
angular velocity	`{0,0,0}`
acceleration	`{0,0,0}`
angular acceleration	`{0,0,0}`
velocity damping	`0`
angular vel damping	`0`

The `MdtBodyCreate()` function creates such a structure:.

```
MdtBodyID MEAPI MdtBodyCreate ( const MdtWorldID world );
```

Creates a new body with default parameters.

See the *Karma Dynamics Reference Manual* for additional details about the `MdtBody` structure.

The body must be enabled in order for it to take part in the simulation:

```
MdtBodyEnable(body);
```

The formal description of this function is:

```
void MEAPI MdtBodyEnable ( const MdtBodyID body );
```

Enables simulation of `body`. By default, bodies are disabled from simulation. This function enables simulation of that body. Bodies are automatically enabled when hit by other enabled bodies.

In `drop.c`, a reset routine initializes the following relevant physical properties: the body's position, linear velocity, angular velocity and quaternion value:

```
void Reset(void)
{
    MdtBodySetPosition(body, 0, 10, 0);
    MdtBodySetLinearVelocity(body, 0, 0, 0);
    MdtBodySetAngularVelocity(body, 0, 0, 0);
    MdtBodySetQuaternion(body, 1, 0, 0, 0);
}
```

Note: A *quaternion* may be used to represent the orientation of a body in space. Rotation matrices or Euler angles are alternative ways that may be used to represent orientation. Please refer to the glossary for further details.

This routine is called whenever the user of the simulation presses the Enter key. This routine uses *mutators*, that is functions that change the value of a structure member. Usually, any structure member that can be modified can also be read. The functions that read structure member values are called *accessors*.

The mutators and accessors for any `MdtBody` structure all have the `MdtBodySet` and `MdtBodyGet` prefixes respectively. A complete list of accessors and mutators can be found in the *Karma Reference Manual.*

Stepping Through Time

To start the simulation, `drop.c`, call the renderer and pass it the name of `drop.c`'s callback function:

```
RRun(rc, Tick);
```

The Viewer in turn calls the `Tick()` callback function (see *Stepping Through the Simulation Using Callback Functions* on page 13) in its main loop. This steps the simulation forward by the timestep set in *Setting the Timestep for the Simulation* on page 14. The following discusses this:

Here is pseudo-code for the Viewers `RRun()`:

```
while no exit-request
{
   Handle user input
   call Tick() to evolve the simulation and update graphic transforms
   Draw graphics
}
```

Each time `Tick()` is called, it first calls the Mdt Library's step function `MdtWorldStep()` to evolve the world by the time step. The choice of timestep should meet the simulation requirements. If the timestep is too large, a simulation may become unrealistic. For example, a falling ball can pass through a floor of finite thickness if its position is not determined at small enough intervals i.e. the contact is missed. If the timestep is too small, the simulation may run at an unacceptably slow speed.

In drop.c the time increment is set using:

```
void MEAPI MdtWorldStep ( const MdtWorldID world, const MeReal stepSize );
```

To evolve the world. This function partition the world and use the Kea solver to work out the forces required to maintain constraints. The integrator then evolves the world, calculating the new positions and velocities. The variable `world` is the `MdtWorld` to simulate and `stepSize` is the amount of time to evolve world by.

i.e.

```
MdtWorldStep(world, step);
```

After stepping forward in time, some or all of the enabled bodies may have moved to new positions, and certain of the bodies' properties may have changed. (In `drop.c` there is one enabled body).

Then `Tick()` sets the graphic bodies transformation matrix to that of the dynamic bodies transformation matrix, i.e.: the renderer is updated with the new object positions and orientations.

```
sphereG->m_matrix = MdtBodyGetTransformPtr(body);
```

Note: The time increment does need to be fixed.

Cleaning Up

The API call `MdtWorldDestroy` in the function `cleanup()` frees the memory at the end of the simulation. See *Stepping Through the Simulation Using Callback Functions* on page 13:

```
MdtWorldDestroy(world);
```

The formal description of `MdtWorldDestroy` is:

`void MEAPI` **`MdtWorldDestroy`** `( const MdtWorldID` *`world`* `);`
Destroys an `MdtWorld` and all bodies and constraints contained in it.

Geometric Properties

There are no dynamic properties that describe the body's shape (that is, its geometry). The only information that a dynamic body has of its extent is its mass distribution. For simulations that do not involve collisions, knowledge of geometry is not required.

> **Note:** Bodies should be assigned inertia tensors that are appropriate to their geometry and mass. Failing to set inertia tensors properly can cause non-physical behavior.

A body does need to have a specific geometric shape assigned to it to determine whether it has collided or come into contact with another body. It is not necessary to know anything about an objects shape in order to simulate it moving freely under gravity.

Both Karma Collision and the Karma Viewer support *models* with *geometries* (shapes). There may or may not be a one-to-one correlation between simulated bodies and the "models" used by Karma Collision, Karma Viewer, or any other third-party software.

One way that the Mdt Library interacts with the geometry of a model is through contacts. See *Creating Contacts: MdtContact* on page 65.

Chapter 3 • Constraints: Joints and Contacts

Overview

The interaction of objects through constraints, such as joints or contacts, is dealt with in this chapter through the use of the high level Karma Dynamics API. An articulated body is comprised of two or more jointed bodies.

By partitioning a world into a subset of objects that interact only with other objects in the sub-set, fast and efficient simulations can be created.

Constraints

A constraint is a restriction on the allowed motion of a physical object. This restriction gives rise to a force acting on the constrained body that effects it’s motion. Constraints may be used to avoid having to model the detailed interactions - the large scale effect is dealt with, without worrying about the underlying physical model. For instance, the forces that prevent a coffee cup from falling through a table arise from electrostatic forces between the respective molecules making up the solid material comprising the cup. However, the net effect of all those complicated forces is simply that the cup doesn't penetrate the table. This can be expressed as a kinematic restriction on the motion and the force resulting from that familiar constraint is called the normal force.

There are many restrictions that can be imposed on rigid bodies using mechanical coupling, all of which are based on a constraint of some description. The more familiar ones are the revolute or hinge joint, the prismatic or sliding joint, the universal joint, the ball and socket or spherical joint, strut joints, etc. Note however, that these joints are an idealization of the real couplings that one can construct with physical components and that are, for example, commonly found attaching doors to door frames, and to transmit drive forces from an engine to a car wheel. No real joint behaves exactly like an ideal joint since real physical bodies are never perfectly rigid, there is always some small clearance in any given assembly.

There are numerous types of constraints that can be imposed on position, angular freedom, velocity, angular velocity or certain combinations of these, in addition to restrictions on the forces required to impose a constraint (if pulled hard enough the elastic limit of a spring can be exceeded) etc. The motion of a single rigid body or the relative motion of two or more rigid bodies can be restricted. Finally, what are known as either *equality* or *inequality* constraints, can be set up. This can become complicated, but it is the purpose of the Karma Dynamics library to deal with all the details.

Constraints are a very powerful modelling tool. Karma Dynamics provides an API through the Mdt Library for a selection of useful constraint types.

Joint Types

Articulated bodies are made up of two or more rigid bodies connected together by joints. Joints, like contacts, are constraints upon the behavior of bodies. This following joint types are supported by the Mdt Library.

- Ball and Socket or spherical.
- Hinge or revolute.
- Prismatic or slider.
- Universal.
- Angular3.
- Car Wheel.
- Linear1.
- Linear2.
- Fixed Path.
- Fixed Position Fixed Orientation (FPFO). Deprecated in favor of RPRO (below).
- Relative Position Relative Orientation (RPRO).
- Spring.
- Cone Limit constraint.

These joints can be used to attach two simulated objects to each other - that is, their relative position or orientation (or both) is constrained. Although the discussion below refers always to attached bodies, these joints may be used to attach a single object to the inertial reference frame (world) by not specifying a second attached body (or by setting it to `NULL`). Complex dynamic structures can be created by linking bodies together using these joints. When such structures are cross-linked (multiply connected), the model must be physically realistic, because the Mdt Library cannot deal with unrealistic structures.

limits (stops) and *actuation* (motors) that can be applied to Hinge and Prismatic joints are discussed in this chapter.

Degrees of Freedom

A free body has six degrees of freedom, that allow it to:

- move freely in any direction in 3D space relative to another object
- freely rotate about any axis in 3D space relative to another object.

Hence, by joining two objects together up to six degrees of freedom from the attached pair of objects can be removed. To have an effect at least one degree of freedom must be removed. The number of degrees of freedom removed by a joint is a measure of the computational cost of using that joint in a simulation, the more degrees of freedom a joint removes, the more costly it is to implement.

Common Constraint Functions

Contacts and joints each have a dedicated set of functions to manage their properties. In order to describe these functions while avoiding redundancy, an abstract, generic set of functions (that unless specified otherwise is common to all joint structures) is first discussed. The specific type of joint will be identified with the wildcard character * to replace one of the following identifiers:

- `BSJoint`, the Ball and Socket joint.
- `Hinge`, the Hinge Joint.
- `Prismatic`, the Prismatic joint.
- `Universal`, the Universal joint.
- `Angular3`, the Angular3 joint.
- `CarWheel`, the Car Wheel Joint.
- `Linear1`, the Linear1 joint.
- `Linear2`, the Linear2 joint.
- `FixedPath`, the Fixed-Path joint.
- `FPFOJoint`, the Fixed-Position-Fixed-Orientation joint.
- `RPROJoint`, the Relative-Position-Relative-Orientation joint.
- `Spring`, the Spring joint.
- `ConeLimit`, the Cone-Limit constraint.
- `Contact`, a contact.

A constraint can only be created by using the appropriate `Mdt*Create()` function for that constraint. A joint must be created if an articulated body is needed. First create a `Mdt*ID` variable (called `joint_or_contact` in the descriptions below) that will point to the `Mdt*` structure where all information about that joint will be stored. The function that creates a joint and returns a `Mdt*ID` variable is:

`Mdt*ID MEAPI` **`Mdt*Create`**`( const MdtWorldID` *`world`* `);`

This function creates a new joint or contact in the `world. This should be followed with the` **`Mdt*SetBodies`**`(const Mdt*ID` *`joint_or_contact`*, const MdtBodyID *body0*, const MdtBodyID *body1*) function to assign bodies to the contact.

The Reset function is common to all joints and contacts, but the default values it resets to are specific to each of them. See the individual constraint descriptions for details.

```
void MEAPI Mdt*Reset( const Mdt*ID joint_or_contact );
```

Set `joint_or_contact` to its default value. Note that the bodies attached to the `joint_or_contact` will have their parameters reset too.

When a joint or a contact is no longer needed, remove it using the following function:

```
void MEAPI Mdt*Destroy( const MdtBSJointID joint_or_contact );
```

This function destroys the joint or contact named `joint_or_contact.`

When a constraint is created it needs to be *enabled* to be processed by the system. Conversely, disable a constraint that is not required.

```
void MEAPI Mdt*Enable( const MdtConstraintID joint_or_contact );
```

Enable the simulation of *`joint_or_contact`*.

```
void MEAPI Mdt*Disable( const MdtConstraintID joint_or_contact );
```

Disabling a constraint or joint stops it being simulated.

```
MeBool MEAPI Mdt*IsEnabled( const MdtConstraintID joint_or_contact );
```

Returns TRUE if the constraint is enadled.

A function to obtain a Joint or Contact ID from a Constraint ID: i.e. the converse of `Mdt*QuaConstraint.`

```
Mdt*ID MEAPI MdtConstraintDCast*( const MdtConstraintID cstrt );
```

Returns an `Mdt*ID` from an `MdtConstraintID`. If this constraint is not of the expected * type, 0 is returned.

Common Accessors

The `Mdt*GetPosition` function that accesses the position of the contact or joint in *world coordinates,* is common to contact and all of the joints except `Prismatic` (does not exist) and `Spring` (an additional argument is needed):

```
void MEAPI Mdt*GetPosition( const Mdt*ID joint_or_contact,
                                    MeVector3 posVector);
```

The position vector of `joint_or_contact` is returned in `posVector`. The undocumented constraint accessor function MdtConstraintGetPosition(MdtConstraintID constraint, MeVector3 position) should not be used. This function does not exist for Angular3. Excludes FixedPath, FPFO and RPRO. The following function is used for these joints.

```
void MEAPI Mdt*GetPosition( const Mdt*ID joint_or_contact,
                   const unsigned int bodyindex, MeVector3 posVector);
```

The position vector of `joint_or_contact` is returned in `posVector` for the FixedPath, FPFO and RPRO joints. . The undocumented constraint accessor function MdtConstraintGetPosition(MdtConstraintID constraint, MeVector3 position) should not be used.

```
MdtBodyID MEAPI Mdt*GetBody( const Mdt*ID joint_or_contact,
                            unsigned int bodyindex );
```

Return one of the bodies connected to this *`joint_or_contact`*. The value of `bodyindex` is 0 for the first body, 1 for the second body.

```
void MEAPI Mdt*GetForce( const Mdt*ID joint_or_contact,
                         unsigned int bodyindex, MeVector3 force );
```

Return the force applied to a body identified by bodyindex by *`joint_or_contact`* (on the last timestep). Forces are returned in the world reference frame.

```
void MEAPI Mdt*GetTorque( const Mdt*ID joint_or_contact,
                         unsigned int bodyindex, MeVector3 torque );
```

Return the torque applied to a body by *`joint_or_contact`* (on the last timestep). The torque is returned in the world reference frame in `torque`.

```
MdtWorldID MEAPI Mdt*GetWorld( const Mdt*ID joint_or_contact );
```

Return the world that *`joint_or_contact`* is in.

```
void *MEAPI Mdt*GetUserData( const Mdt*ID joint_or_contact );
```

Return the user-defined data of *joint_or_contact.*

```
MeI32 MEAPI Mdt*GetSortKey( const Mdt*ID joint_or_contact );
```

Return the sort key of *joint_or_contact.*

Common Mutators

The `Mdt*SetPosition` function that mutates the position of the contact or the joint in *world coordinates,* is common to contact and all of the joints except `Prismatic` (does not exist) and `Spring` (an additional argument is required).

```
void MEAPI Mdt*SetPosition( const Mdt*ID joint_or_contact,
                  const MeReal xPos, const MeReal yPos, const MeReal zPos );
```

Set the `joint_or_contact` position in world coordinates at `(xPos, yPos, zPos)`. The undocumented constraint mutator function MdtConstraintSetPosition(MdtConstraintID constraint, MeReal x, MeReal y, MeReal z) should not be used. This function does not exist for Angular3.

```
void MEAPI Mdt*SetBodies( const Mdt*ID joint_or_contact,
                        const MdtBodyID body0, const MdtBodyID body1 );
```

Attach `body0` and `body1` to *joint_or_contact.*

```
void MEAPI Mdt*SetUserData( Mdt*ID joint_or_contact, void *data )
```

Set the *joint_or_contact* user data.

> **NOTE:** The similarity of the constraint functions used by joints and contacts arises because most of the joints and contact functions are macros that are defined through the constraint function. Karma header files for the joints and contacts give a complete function listing. Descriptions of some useful base constraint functions follow.

```
void MEAPI Mdt*SetSortKey( const Mdt*ID joint_or_contact, MeI32 key );
```

Assign a sort key to *joint_or_contact.*

Base Constraint Functions

The `MdtConstraint` functions are a set of functions that apply to all constraints. The base constraint data consists of one attachment point per rigid body that is, a position and an orientation. This is the position of the joint as seen from each rigid body.

These base constraint functions have been used in implementing many of the individual constraint functions and, in some cases, offer an alternative to the individual functions. However, the effects of the base constraint functions are not well defined or documented and the individual constraint functions should always be used in preference.

When a joint or contact constraint is created, the functions can be accessed by converting the contact or joint ID to a `MdtConstraintID` variable using the following function:

```
MdtConstraintID Mdt*QuaConstraint( const Mdt*ID joint_constraint );
```

This function is used to convert a specific joint identifier to its abstract representation of a constraint identifier `MdtConstraintID`.

A constraint is destroyed by using:

```
void MEAPI MdtConstraintDestroy( const MdtConstraintID constraint );
```

Destroys a constraint. The constraint is disabled automatically if necessary.

The `MdtConstraint*` functions include a function to enable and a function to disable a constraint. The difference between the *destroy* function and the *disable* function is that the *disable* keeps the constraint structure in memory for later use. Destroying a constraint removes it from memory so that the it cannot be used at a later time.

```
void MEAPI MdtConstraintEnable( const MdtConstraintID constraint );
```

Enables simulation of a constraint.

```
void MEAPI MdtConstraintDisable( const MdtConstraintID constraint );
```

Disables simulation of a constraint.

```
MeBool MEAPI MdtConstraintIsEnabled( const MdtConstraintID constraint );
```

Determine if a constraint is currently enabled. Returns 1 if enabled, or 0 if not.

The Constraints Mutator Functions

To attach bodies to a constraint use:

```
void MEAPI MdtConstraintSetBodies( const MdtConstraintID constraint,
                                   const MdtBodyID body0, const MdtBodyID body1 );
```

Set the bodies `body0` and `body1` to be attached to the constraint.

Note that a constraint must be disabled before changing the bodies attached to it. The constraint library provides a number of Set/Get functions to mutate and access the variables of a constraint structure. Please consult the Karma Dynamics Reference Manual.

```
void MEAPI MdtConstraintSetAxis( const MdtConstraintID constraint,
          const MeReal px, const MeReal py, const MeReal pz );
```

Set the constraint primary axis in the world reference frame.

```
void MEAPI MdtConstraintSetAxes( const MdtConstraintID constraint,
          const MeReal px, const MeReal py, const MeReal pz,
          const MeReal ox, const MeReal oy, const MeReal oz )
```

Set the primary and secondary constraint axes in the world reference frame.

The axes will be normalized automatically.

The axes must be orthogonal.

This effectively sets the rotational orientation of a constraint frame consisting of the two given axes and a third orthogonal axis corresponding to the cross product of the given axes.

An older, deprecated name for this function is **`MdtConstraintSetBothAxis`** (sic).

```
void MEAPI MdtConstraintBodySetPosition( const MdtConstraintID constraint,
                          const unsigned int bodyindex,
                          const MeReal x, const MeReal y, const MeReal z );
```

Set the constraint position for the given body in the world reference frame.

```
void MEAPI MdtConstraintSetUserData( const MdtConstraintID constraint,
                                     void *data )
```

Set the constraint userdata.

```
void MEAPI MdtConstraintSetSortKey( const MdtConstraintID constraint,
                                                          MeI32 key );
```

Set the constraint sort key.

The Constraint Accessor Functions

Most (but not all) mutator functions are paired to equivalent accessor functions:

```
MdtBodyID MEAPI MdtConstraintGetBody( const MdtConstraintID constraint,
                                      const unsigned int bodyindex );
```

Return the body connected to this constraint as determined by bodyindex. The value of `bodyindex` is 0 for the first body, 1 for the second body.

```
void MEAPI MdtConstraintGetAxis( const MdtConstraintID constraint,
                                 MeVector3 axis );
```

Get the `constraint` primary axis in the world reference frame and store its value in `axis`.

```
void *MEAPI MdtConstraintGetUserData( const MdtConstraintID constraint );
```

Return the user-defined data of this constraint.

```
void MEAPI MdtConstraintBodyGetAxes( const MdtConstraintID constraint,
                                     const unsigned int bodyindex,
                                     MeVector3 primary, MeVector3 ortho );
```

Get both the primary constraint axis and the orthogonal secondary constraint axis in the world reference frame for the given body.

An older, deprecated name for this function is **`MdtConstraintBodyGetBothAxes`**.

```
void MEAPI MdtConstraintGetAxes( const MdtConstraintID constraint,
                                   MeVector3 primary,  MeVector3 ortho );
```

Get both the primary constraint axis and the orthogonal secondary constraint axis in the world. ref. frame.

An older, deprecated name for this function is **`MdtConstraintGetBothAxes`**.

To get the value of the force and torque applied to a given body by a constraint use:

```
void MEAPI MdtConstraintGetForce( const MdtConstraintID constraint,
                        const unsigned int bodyindex, MeVector3 force );
```

Return the force applied to the body (identified by bodyindex) by this constraint on the last timestep. Forces are returned in the world reference frame.

```
void MEAPI MdtConstraintGetTorque( const MdtConstraintID constraint,
                        const unsigned int bodyindex, MeVector3 torque );
```

Return the torque applied to the body (identified by bodyindex) by this constraint on the last timestep. Torque is returned in the world reference frame.

To determine which world a constraint belongs to use:

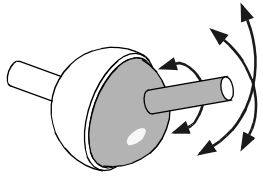
```
MdtWorldID MEAPI MdtConstraintGetWorld( const MdtConstraintID constraint );
```

Return the world that the constraint is in.

```
MeI32 MEAPI MdtConstraintGetSortKey( const MdtConstraintID constraint );
```

Return the constraint sort key.

Ball-and-socket (BS) Joint: MdtBSJoint



A ball and socket joint forces a point fixed in one bodies reference frame to be at the same location in the world reference frame as that of a point fixed in another bodies reference frame. This removes three (linear) degrees of freedom. In the diagram above, the center of the sphere always coincides with the center of the socket. This ideal joint allows all rotations about the common point. Real ball and socket joints have joint limits because a body attached to the ball will collide with the sides of the socket. The MdtBSJoint does not have limits built in but the MdtConeLimit constraint can be used with it to provide limits. This joint is sometimes referred to as a spherical joint.

A ball and socket joint, in conjuction with a cone limit, may be used to model a shoulder joint, or to connect links in a chain.

Ball-and Socket Joint Functions

There are no functions specific to `MdtBSJoint`. The reset function sets the joint position in each bodies' reference frame to {0, 0, 0}. The joint position can be set to change this default.

```
MdtBSJointID bs = MdtBSJointCreate(world);
```

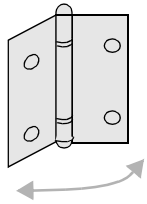
creates a ball and socket joint in an `MdtWorld world`. To use the joint to constrain a pair of objects (`body1` and `body2`) use;

```
MdtBSJointSetBodies(bs, body1, body2);
MdtBSJointSetPosition(bs, pos_x, pos_y, pos_z);
```

This sets the position of the joint in the world frame. The fixed positions of the joint relative to each body are then initialized. Alternatively the joint positions can be set individually for each body using the base constraint interface.

The bodies should already have been created and positioned in their requisite initial positions before attaching them to the joint.

Hinge Joint: MdtHinge



A hinge constrains a pair of bodies to rotate freely about a specific hinge axis. The remaining five degrees of freedom between the joined bodies are fixed. Because of this the hinge is more computationally costly than the previously discussed ball and socket joint. The hinge axis has fixed positions and orientations in each rigid body, and the hinge constraint forces those axes to coincide at all times. A hinge is sometimes referred to as a revolute joint.

A hinge joint could be used to attach a door to a doorframe, a lever, drawbridge or seesaw to its fulcrum, or to attach rotating parts such as a wheel to a chassis, a propeller shaft to a ship or a turntable to a deck.

Hinge Limits

Up to two stops, or limits, can be set to restrict the relative rotation of the bodies attached by a hinge joint. These limits may be specified independently to be either *hard* (if the limit stiffness factor is high) or *soft*. In a Karma Dynamics simulation, a hard bounce reverses the bodies' angular velocities in a single timestep, while a soft bounce may take many timesteps to reverse the angular velocity.

If the limits are soft, damping can be set so that, beyond the limits, the hinge behaves like a damped spring.

If the limits are hard, the limit restitution can be set to a value between zero and one to govern the loss of angular momentum as the bodies rebound.

Hinge joint limits range from $-n\pi$ through $n\pi$ for real number n hence multiple rotations are supported and a hinge passing a limit will always be detected and the correct response simulated.

Hinge Actuators

A hinge joint can be actuated (powered). This simulates a motor acting on the hinge's remaining degree of freedom, the hinge angle. To characterize a hinge motor, set a desired angular speed and the motor's maximum torque. The motor is assumed to be symmetric, so that the maximum torque can be applied in either direction. A torque no greater than this is applied to the hinged bodies to change their relative angular velocity, until either the desired velocity is achieved, or the hinge angle hits a limit (if set).

The response of an actuated hinge hitting a limit depends on the stiffness and restitution or damping properties that have been chosen for the relevant limit, but in general the hinge will (quickly or slowly) come to rest at the set limit. If a soft limit has been specified, the rest position will be beyond the limit by an angle determined by the motor's maximum torque and the limit stiffness factor.

Whenever a hinge is actuated, or is at (or beyond) one of its limits, the computational cost is equivalent to constraining six degrees of freedom.

Hinge Joint Functions

A Hinge joint is described by the position and direction of its axis. The reset function zeros the position in each body's reference frame, and sets the axis direction to each body's x-axis, i.e., position = {0,0,0}, axis={1,0,0}.

Accessors

The accessor functions specific to the hinge are:

```
MdtLimitID MEAPI MdtHingeGetLimit( const MdtHingeID joint );
```

Provide read and write access with the `MdtLimit` functions to the constraint limits parameters of the `joint` by providing the corresponding `MdtLimitID` identifier.

```
void MEAPI MdtHingeGetAxis( const MdtHingeID joint, MeVector3 axisVec );
```

The Hinge joint axis is returned in the vector `axisVec`.

Mutators

The mutator functions specific to the hinge are:

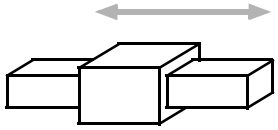
```
void MEAPI MdtHingeSetLimit( const MdtHingeID joint,
                             const MdtLimitID NewLimit );
```

Reset the joint limit and then copy the public attributes of `NewLimit`.

```
void MEAPI MdtHingeSetAxis( const MdtHingeID joint,
                            const MeReal xAxis,
                            const MeReal yAxis,
                            const MeReal zAxis );
```

Set the hinge axis of `joint` to `(xAxis, yAxis, zAxis)`

Prismatic: MdtPrismatic



The prismatic, or slider, joint is like the hinge in that two axes, one fixed in each of the two constrained bodies reference frames, are forced to coincide. In the prismatic however, the 2 bodies move along the axis, not around it. Like a hinge, a prismatic joint removes five degrees of freedom from the relative motion of the attached bodies, leaving one linear degree of freedom. The relative orientation of the bodies are maintained by the joint. The prismatic can be imagined as a bar sliding inside a block with a hole in it, where the area of the hole matches the cross sectional area of the bar.

Prismatic Limits

Two limits may be set to restrict the linear motion of a prismatic joint. These limits may be either hard or soft, with the ability to set the stiffness, restitution and damping properties independently for each limit.

Prismatic Actuators

Speed and maximum force may be set to actuate the movement of a prismatic joint. The actuation force will be applied to slow down or speed up the attached bodies until their relative velocity reaches the specified speed, unless a limit is reached first.

Whenever a prismatic joint is actuated, or is at (or beyond) one of its limits, the computational cost is equivalent to constraining six degrees of freedom rather than five.

Prismatic Joint Functions

A Prismatic joint is described by the direction of its sliding axis. The reset function sets this to the bodies' x-axis, i.e. {1,0,0}.

When the constrained bodies have been initialized and set to their starting positions, all that is required to initialize the Prismatic joint is to set the direction of its sliding axis.

The initial position of the bodies specifies the zero displacement of the sliding degree of freedom used for the Prismatic limits. This is set automatically when the axis is set.

Accessors

Here are the accessor functions specific to Prismatic:

```
MdtLimitID MEAPI MdtPrismaticGetLimit ( const MdtPrismaticID joint );
```

Provides read/write access to the constraint limits of `joint`.

```
void MEAPI MdtPrismaticGetAxis ( const MdtPrismaticID joint,
                                 MeVector3 axisVec );
```

The prismatic joint axis is returned in the vector `axisVec`.

Mutators

Here are the mutator functions specific to Prismatic:

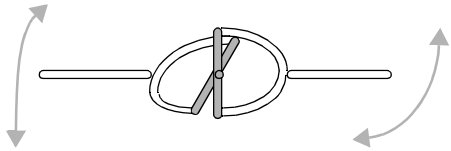
```
void MEAPI MdtPrismaticSetLimit ( const MdtPrismaticID joint,
                                  const MdtLimitID NewLimit );
```

Reset the joint limit and then copy the public attributes of `NewLimit`.

```
void MEAPI MdtPrismaticSetAxis ( const MdtPrismaticID joint,
          const MeReal xAxis, const MeReal yAxis, const MeReal zAxis );
```

Set the prismatic axis of `joint` to `(xAxis, yAxis, zAxis)`.

Universal Joint: MdtUniversal



In this joint, two axes, one fixed in each of the two constrained bodies, are forced to have a common origin and to be perpendicular at all times. This is a lot like the ball and socket joint but here the ball is not allowed to twist in its socket.

A universal joint removes four degrees of freedom from the attached bodies. It fixes their relative position and constrains them not to twist about a third axis, perpendicular to the two given axes. This joint may be pictured as a joystick mechanism in which two hinges are joined, one on top of the other with perpendicular axes, to allow an attached stick to move first in the *x*-direction then in the *y*-direction.

This mechanism is also known as gimbal, and the mechanism suffers from dreaded ‘gimbal-lock’ at 90°. This relates to its use as an ‘engineering’ universal joint that can be used to transmit torque from one body to another around a small bend. The transmission becomes increasingly unsmooth as the angle of bend approaches 90° and finally cannot transmit torque at all.

In Karma, the singularity at 90° can be avoided by applying a Cone-Limit constraint in parallel with the Universal.

Universal Joint Functions

A Universal joint is described by the position of the joint and the directions of the axes fixed in each body. The reset function zeros the position in each body frame and defaults the axes to the x-axis {1,0,0} in body1 and the y-axis {0,1,0} in body2.

Accessors

The accessor function specific to Universal is:

```
void MEAPI MdtUniversalGetAxis ( const MdtUniversalID joint,
                      const unsigned int bodyindex, MeVector3 axis );
```

The joint axis corresponding to `bodyindex` is returned in the vector `axisVec`.

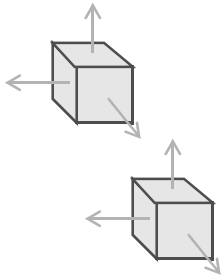
Mutators

The mutator function specific to Universal is:

```
void MEAPI MdtUniversalSetAxis ( const MdtUniversalID joint,
                      const unsigned int bodyindex,
                      const MeReal Ax, const MeReal Ay, const MeReal Az );
```

Set the joint axis corresponding to `bodyindex`, to `(xAxis, yAxis, zAxis)` in world coordinates

Angular Joint: MdtAngular3 & MdtAngular2



The relative orientation of these two bodies is fixed but their relative position can vary freely

The Angular3 joint removes three rotational degrees of freedom by constraining one body to have a fixed orientation with respect to another body. While one body can move freely in space (irrespective of the other body's location) its orientation is fixed relative to the other body's orientation. It is possible to add one rotational degree of freedom about a specified axis, enabling a rotation of a body with respect to the other, effectively modifying the Angular3 joint to an Angular2 joint.

Angular3 and angular2 joints are useful for keeping things upright, such as game vehicles that should not overturn. Springing and damping can be set to introduce some softness around the upright.

Angular3 Joint Functions

The default for the Angular3 is to align the two bodies' reference frames. The reset function also has this effect. The joint is initialized by the call to **`MdtAngular3SetBodies`** that notes the bodies initial orientations for use in maintaining the same fixed relative orientation.

Accessors

The accessor functions specific to Angular3 `are`:

```
MeBool MEAPI MdtAngular3RotationIsEnabled( const MdtAngular3ID joint );
```

Return the current state of the `bEnableRotation` flag. If this flag is set (true), this constraint is effectively an `Angular2` joint.

```
void MEAPI MdtAngular3GetAxis( const MdtAngular3ID joint, MeVector3 axis );
```

The `Angular3` joint axis vector is returned in `axis`. Rotation is allowed about this axis if the `bEnableRotation` flag is set.

```
MeReal MEAPI MdtAngular3GetStiffness( const MdtAngular3ID j );
```

Return current 'stiffness' of this angular constraint.

```
MeReal MEAPI MdtAngular3GetDamping( const MdtAngular3ID j );
```

Return current spring 'damping' of this angular constraint.

Mutators

The mutator functions specific to `Angular3 are:`

```
void MEAPI MdtAngular3EnableRotation( const MdtAngular3ID joint,
                                      const MeBool NewRotationState );
```

Set or clear the joint `bEnableRotation` flag to `NewRotationState`. If this flag is set (true), this constraint is effectively an Angular2 joint enabling the rotation of one body relative to another.

```
void MEAPI MdtAngular3SetAxis( MdtAngular3ID joint,
                               MeReal xAxis, MeReal yAxis, MeReal zAxis );
```

Set the joint axis at `(xAxis, yAxis, zAxis)` in world coordinates. Note that this axis is used only if the `bEnableRotation` flag is set.

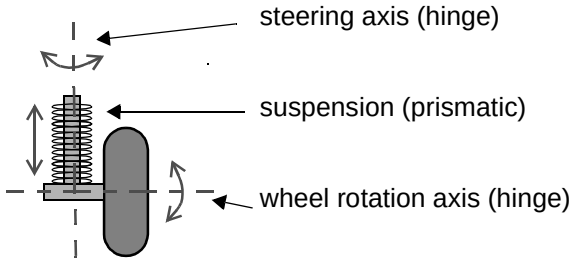
```
void MEAPI MdtAngular3SetStiffness( const MdtAngular3ID j, const MeReal s );
```

Set the angular constraint stiffness about the enabled axis. The default is MEINFINITY

```
void MEAPI MdtAngular3SetDamping( const MdtAngular3ID j, const MeReal d );
```

Set the angular constraint damping about the enabled axis. The default is MEINFINITY

CarWheel Joint: MdtCarWheel



The CarWheel joint models the behavior of a car wheel with optional steering and suspension. The CarWheel joint is a combination of two hinge joints, one for the steering and one for the rotation of the wheel, and one prismatic joint for telescopic suspension with built in springing.

Body 1 is the chassis and body 2 is the wheel. The connection point for the wheel body is its center of mass.

CarWheel Joint Functions

A CarWheel joint is defined by a steering axis and a hinge axis. The steering axis also acts as the suspension axis. It has a direction fixed in the chassis frame and passes through the origin of the wheel's reference frame. The hinge axis has a direction fixed in the wheel frame and also passes through the frame origin. The constraint keeps these two axes perpendicular.

The default steering axis is the body1 z-axis and the default hinge axis is the body2 y-axis.

Accessors

The accessor functions specific to the CarWheel `joint are:`

```
MeReal MEAPI MdtCarWheelGetHingeAngle( const MdtCarWheelID joint );
```

Return the wheel joint current hinge angle as a value between zero and PI radians, inclusive.

```
MeReal MEAPI MdtCarWheelGetHingeAngleRate( const MdtCarWheelID joint );
```

Return the wheel joint angular velocity about the hinge axis.

```
void MEAPI MdtCarWheelGetHingeAxis( const MdtCarWheelID joint,
                                    MeVector3 hingeAxis );
```

The wheel joint hinge axis is returned in `hingeAxis`.

```
MeReal MEAPI MdtCarWheelGetHingeMotorDesiredVelocity(
                                          const MdtCarWheelID joint );
```

Return the desired velocity of the hinge motor.

```
MeReal MEAPI MdtCarWheelGetHingeMotorMaxForce( const MdtCarWheelID joint );
```

Return the maximum force that the hinge motor is allowed to use to attain its desired velocity.

```
MeReal MEAPI MdtCarWheelGetSteeringAngle( const MdtCarWheelID joint );
```

Return the wheel joint steering angle.

```
MeReal MEAPI MdtCarWheelGetSteeringAngleRate( const MdtCarWheelID joint );
```

Return the wheel joint angular velocity about the steering axis.

```
void MEAPI MdtCarWheelGetSteeringAxis( const MdtCarWheelID joint,
                                  MeVector3 steeringAxis );
```

The wheel joint steering axis is returned in `steeringAxis`.

```
MeReal MEAPI MdtCarWheelGetSteeringMotorDesiredVelocity(
                                          const MdtCarWheelID joint)
```

Return the desired velocity of the steering motor.

```
MeReal MEAPI MdtCarWheelGetSteeringMotorMaxForce (
                                          const MdtCarWheelID joint );
```

Return the maximum force that the steering motor is allowed to use to attain its desired velocity.

```
MeReal MEAPI MdtCarWheelGetSuspensionHeight( const MdtCarWheelID joint );
```

Return the wheel joint suspension height.

```
MeReal MEAPI MdtCarWheelGetSuspensionHighLimit( const MdtCarWheelID joint );
```

Return the suspension upper limit.

```
MeReal MEAPI MdtCarWheelGetSuspensionKd ( const MdtCarWheelID joint );
```

Return the suspension "damping constant" (also known as the "derivative constant"). This gives rise to the damping term K_d in the suspension force equation: F = -k_p*displacement + k_d*velocity, where K_p is Hookes Law constant and K_d is the damping constant.

```
MeReal MEAPI MdtCarWheelGetSuspensionKp( const MdtCarWheelID joint );
```

Return the suspension "proportionality constant". This gives rise to the spring term k_p in the suspension force equation: F = -k_p*displacement + k_d*velocity, where K_p is Hookes Law constant and K_d is the damping constant

```
MeReal MEAPI MdtCarWheelGetSuspensionLimitSoftness(
                                          const MdtCarWheelID joint );
```

Return the suspension limit softness.

```
MeReal MEAPI MdtCarWheelGetSuspensionLowLimit( const MdtCarWheelID joint );
```

Return the suspension lower limit.

```
MeReal MEAPI MdtCarWheelGetSuspensionReference( const MdtCarWheelID joint );
```

Return the suspension attachment point (*reference*).

```
MeBool MEAPI MdtCarWheelIsSteeringLocked( const MdtCarWheelID joint );
```

Return the lock state of the steering angle. (lock is 1 if steering axis is locked at angle 0).

Mutators

The mutator functions specific to the CarWheel `joint are:`

```
void MEAPI MdtCarWheelSetHingeAxis( const MdtCarWheelID joint,
            const MeReal xHinge, const MeReal yHinge, const MeReal zHinge );
```

Set the wheel joint hinge axis at `(xHinge, yHinge, zHinge)`.

```
void MEAPI MdtCarWheelSetHingeLimitedForceMotor( const MdtCarWheelID joint,
                    const MeReal desiredVelocity, const MeReal forceLimit );
```

Set the hinge limited force motor parameters.

```
void MEAPI MdtCarWheelSetSteeringAxis( const MdtCarWheelID joint,
                      const MeReal x, const MeReal y, const MeReal z );
```

Set the wheel joint steering axis.

```
void MEAPI MdtCarWheelSetSteeringLimitedForceMotor(
                          const MdtCarWheelID joint,
                 const MeReal desiredVelocity, const MeReal forceLimit );
```

Set the limited force motor parameters of a car wheel joint.

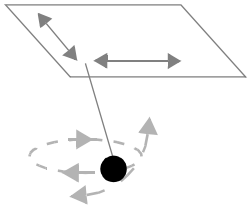
```
void MEAPI MdtCarWheelSetSteeringLock( const MdtCarWheelID joint,
                                                const MeBool lock );
```

Lock or unlock the steering angle (lock is 1 if steering axis is locked at angle 0).

```
void MEAPI MdtCarWheelSetSuspension( const MdtCarWheelID joint,
               const MeReal Kp, const MeReal Kd,
               const MeReal limit_softness, const MeReal lolimit,
               const MeReal hilimit, const MeReal reference );
```

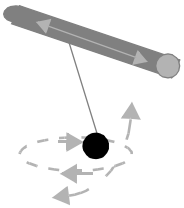
Set the suspension parameters.

Linear1 Joint: MdtLinear1



A Linear1 joint removes one degree of freedom by confining a point fixed in one of the attached bodies to a plane fixed in the other body.

Linear2 Joint: MdtLinear2



A Linear 2 joint removes two degrees of freedom by confining a point fixed in one of the attached bodies to a line fixed in the other body. This can be used to simulate a continuous sliding contact between an object and a line, such as a rail or pole.

Functions Specific to Linear2 Joint

Accessors

The accessor function specific to Linear2 is:

```
void MEAPI MdtLinear2GetDirection ( const MdtLinear2ID contact,
                                    MeVector3 directVec );
```

The `Linear2` joint primary direction is returned in `directVec` in the world reference frame.

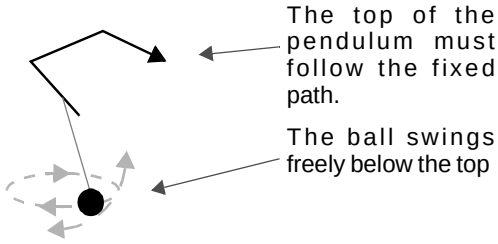
Mutators

The mutator function specific to Linear2 is:

```
void MEAPI MdtLinear2SetDirection ( const MdtLinear2ID contact,
                  const MeReal xDir, const MeReal yDir, const MeReal zDir );
```

Set the joint direction `(xDir, yDir, zDir)` in world coordinates. x, y and z should be `const`.

Fixed-Path Joint: MdtFixedPath



The Fixed-Path joint is a Ball-and-Socket joint modified to allow motion of the joint attachment point. To enable this to be done correctly both position and velocity data for the moving joint attachment point are required as it moves along a given path. While it is possible to move the position of a Ball-and-Socket directly, this does not feed the correct forces into the attached bodies and relies on numerical relaxation to satisfy the constraint.

This joint can be used to attach an animated path to the simulation, in such a way that the forces generated by any animated motion will be transmitted correctly to the simulated, non-animated, objects. For example, a Fixed Path joint could be used to move the attachment point of a pendulum kinematically while the pendulum swings in response to the motion, as sketched above. The animation must supply the position of the fixed path joint at each timestep. The joint can feed back the forces and torques resulting from the pull of gravity and any contact with other simulated bodies.

A Fixed Path joint fixes the relative position of the two attached bodies, removing three degrees of freedom, while leaving them free to rotate freely with respect to one another.

Functions Specific to Fixed-Path Joint

Accessors

The accessor function specific to FixedPath is:

```
void MEAPI MdtFixedPathGetVelocity( const MdtFixedPathID joint,
                    const unsigned int bodyindex, MeVector3 velocity );
```

The fixed path joint velocity with respect to one of the constrained bodies is returned in `velocity`. The reference frame is determined by the third parameter, *`bodyindex`*.

Mutators

The mutator function specific to FixedPath is:

```
void MEAPI MdtFixedPathSetVelocity( const MdtFixedPathID joint,
            const unsigned int bodyIndex,
            const MeReal xVel, const MeReal yVel, const MeReal zVel);
```

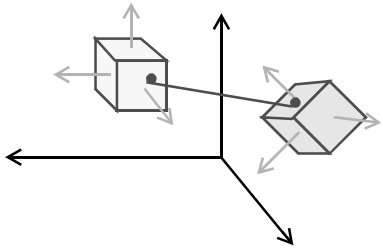
Set the fixed path joint velocity with respect to one of the constrained bodies. This joint velocity is set in the bodies' reference frames. The reference frame is determined by the fifth parameter, `bodyIndex`.

Fixed-Position-Fixed-Orientation Joint: MdtFPFOJoint

> **NOTE:** This joint is deprecated. The Relative-Position-Relative-Orientation joint replaces some of the intended use cases for this joint.

A fixed position, fixed orientation joint is a totally fixed joint that removes all six degrees of freedom, hence all relative motion, between the connected bodies. Because of this it is computationally costly and is not recommended for connecting two dynamic bodies to make a single dynamic body.

Relative-Position-Relative-Orientation Joint: MdtRPROJoint



The RPRO joint is a feature added for use in ‘playing back’ an animation through an object, or in controlling the motion of an object via user interaction, while allowing for physical response to collisions and fast motions. Note that the current implementation supports animation of relative orientations only, addressing character motion use-cases where the animated objects are articulated chains connected by ball-and-socket joints. Support for animation of relative positions is scheduled for a future release.

The RPRO joint constrains all six degrees of freedom between the attached bodies, or in other words leaves no freedom to move between the two bodies. The exception is when the force-limit feature allows the joint to break when a specified maximum force is exceeded in a collision or fast motion. However, the relative orientation can be driven by an animation script or a stream of user input supplied as a sequence of relative quaternions and relative angular velocities (when support for relative position is added an input sequence of relative positions and relative linear velocities will be required to drive translations).

By default, the relative motion is specified between the center-of-mass frames of the primary and secondary bodies. It is also possible to specify joint attachment frames that are offset from the center of mass - useful if animation data is supplied in a different body-fixed frame of reference, though this feature does incur a small performance cost. At present, because relative positions are not yet supported, the only way to create an offset between the two body frames is to specify a joint attachment position using `MdtRPROJointSetAttachmentPosition()`.

Angular motion between the two bodies is achieved by updating the relative orientation using `MdtRPROJointSetRelativeQuaternion()` and the relative angular velocity using `MdtRPROJointSetRelativeAngularVelocity()`.

Force limits, or ‘strengths’, of the linear and angular parts of the constraint can be set using `MdtRPROJointSetLinearStrength() and MdtRPROJointSetAngularStrength()`. Setting low strengths will cause the constraint to be broken easily by imposed accelerations or external forces. This provides a method by which animations can be played through an object while allowing physical reactions to fast motions or collisions with other objects in the simulation.

As an example of user interaction, the RPRO joint is useful in picking up and reorienting objects in the simulation environment. In a ‘first person’ game an object could be picked up at a fixed point in the player’s perspective and its position maintained in the field of view as the player moves. With linear strengths set to small values the picked object will react physically to collisions and fast motions. The object can be reoriented around its picked position by updating the relative quaternion and relative angular velocity inputs.

As with all joints the RPRO can be used to join a body to the world by specifying one of the bodies as NULL. This could be used to drive anchored robot arms, cranes or other mechanisms fixed in the simulation world frame. Note that, because relative positions are not yet implemented, full vehicle motions cannot be directly driven in this way.

Functions Specific to RPRO Joint

Accessors

The accessor functions specific to `RPRO are`

```
void MEAPI MdtRPROJointGetRelativeQuaternion( const MdtRPROJointID joint,
                                              MeVector4 quaternion );
```

This retrieves the relative quaternion.

```
void MEAPI MdtRPROJointGetAttachmentOrientation( const MdtRPROJointID joint,
const unsigned int bodyindex, MeVector4 quaternion )
```

The full constraint joint attachment orientation with respect to one of the constrained bodies is returned in `quaternion`. The reference frame is selected by the parameter `bodyindex`.

```
void MEAPI MdtRPROJointGetAttachmentPosition( const MdtRPROJointID joint,
                                    const unsigned int bodyindex,
                                    MeVector3 position )
```

The full constraint joint position with respect to one of the constrained bodies is returned in `position`. The reference frame is selected by the second parameter (`bodyindex`).

Mutators

The mutator functions specific to `RPRO are`

```
void MEAPI MdtRPROJointSetRelativeQuaternion( const MdtRPROJointID joint,
                                              const MeVector4 quaternion );
```

Set the relative orientation quaternion.

```
void MEAPI MdtRPROJointSetAttachmentQuaternion( const MdtRPROJointID joint,
                                                const MeReal q0,
                                                const MeReal q1,
                                                const MeReal q2,
                                                const MeReal q3,
                                                const unsigned int bodyindex );
```

Set the full constraint joint attachment orientation with respect to one of the constrained bodies described by the quaternion `(q0,q1,q2,q3)`.The reference frame is selected by the parameter `bodyindex`.

```
void MEAPI MdtRPROJointSetAttachmentPosition( const MdtRPROJointID joint,
                                      const MeReal x,
                                      const MeReal y,
                                      const MeReal z,
                                      const unsigned int bodyindex );
```

Set the constraint joint position with respect to one of the constrained bodies. The reference frame is selected by the parameter `bodyindex`.

```
void MEAPI MdtRPROJointSetAngularStrength( const MdtRPROJointID joint,
                                        const MeReal sX,
                                        const MeReal sY,
                                        const MeReal sZ );
```

Set the limit on the maximum torque `(sX,sY,sZ)` that can be applied to maintain the constraints. If all values are set at `MEINFINITY`, the constraint will always be maintained. If some finite (positive) limit is set, the constraint will become violated if the force required to maintain it becomes larger than the threshold.

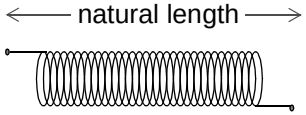
```
void MEAPI MdtRPROJointSetLinearStrength( const MdtRPROJointID joint,
                                      const MeReal sX,
                                      const MeReal sY,
                                      const MeReal sZ );
```

Set the limit on the maximum force `(sX,sY,sZ)` that can be applied to maintain the constraints. If all values are set at `MEINFINITY`, the constraint will always be maintained. If some finite (positive) limit is set, then, the constraint will become violated if the force required to maintain it becomes larger than the threshold.

```
void MEAPI MdtRPROJointSetRelativeAngularVelocity( MdtRPROJointID joint,
                                                   MeVector3 velocity );
```

Set the relative angular velocity of the joint.

Spring Joint: MdtSpring



This joint attaches one body to another, or to the inertial reference frame, at a given separation. The spring joint tends to restore itself to its natural length by opposing any extension (body relative distance > spring natural length) or any compression (body relative distance < spring natural length).

The mathematical relation between the force exerted by a spring (*F*), its natural length (*l*), its stiffness (*k*) and the distance between its two endpoints (*d*) is called *Hooke's Law* and is written as:

$$F = -k(d - l)$$

The separation between the two attached bodies is governed by two limits that may both be *hard* (which simulates a rod or strut joint) or both *soft* (simulating a spring) or hard on one limit but soft on the other (e.g. an elastic attachment that may be stretched but not compressed). The default behaviour is spring-like, with two soft, damped limits, both initialized at the initial separation of the bodies.

There is no angular constraint between bodies attached by a spring. The one linear dimension is constrained, restricting one degree of freedom. This adds just one row to the constraint matrix.

The spring is a configurable distance constraint.

- String can be simulated that can decrease in length but not increase.
- Elastic, that can decrease in length and can stretch can be simulated.
- A solid rod that cannot change it's length can be simulated.

Functions that are Specific to the Spring Joint

Accessors

The accessor functions specific to Spring are:

```
MdtLimitID MEAPI MdtSpringGetLimit ( const MdtSpringID joint );
```

Return the ID handle of a constraint limit.

```
void MEAPI MdtSpringGetPosition( const MdtSpringID joint, MeVector3 position,
                                 const unsigned int bodyindex );
```

The spring joint attachment position to the body `bodyindex` is returned in `position`.

Mutators

The mutator functions specific to Spring are:

```
void MEAPI MdtSpringSetLimit( const MdtSpringID joint,
                                    const MdtLimitID NewLimit );
```

Reset the joint limit and copy the public attributes of `NewLimit`.

```
void MEAPI MdtSpringSetNaturalLength( const MdtSpringID joint,
                                   const MeReal NewNaturalLength );
```

Set the spring length under no load i.e. it's natural length.

```
void MEAPI MdtSpringSetStiffness( const MdtSpringID joint,
                                  const MeReal NewStiffness );
```

Set the spring stiffness or spring constant.

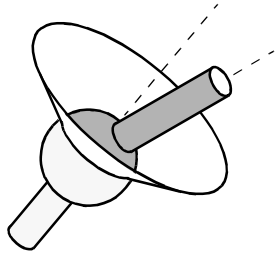
```
void MEAPI MdtSpringSetDamping( const MdtSpringID joint,
                                const MeReal NewDamping )
```

Set the spring damping value.

```
void MEAPI MdtSpringSetPosition( const MdtSpringID joint,
                                 const unsigned int bodyindex,
                                 const MeReal x, const MeReal y, const MeReal z )
```

Set the joint position in world coordinates. This function differs from the generic `Mdt*SetPosition` by requiring a `bodyindex`.

Cone Limit constraint: MdtConeLimit



The `MdtConeLimit` constraint places a limit on the angle between a pair of axes, one being fixed in each body. This constraint can be used in parallel with a ball and socket joint, for example, to limit its angular freedoms to a cone as shown in the sketch. Note that the cone limit does not place a limit on the 'twist' freedom.

The Cone Limit behaves more like a contact than a joint in that it adds no constraint while inside the limit. When the limit is hit, a single constraint is generated to enforce the angular limit.

The Cone-Limit constraint can be used in parallel with a UniversalJoint, that will also constrain the twist freedom, or on top of an Angular3 or an RPROJoint with similar effect.

The behavior of a Cone-Limit is ill defined for small cone angles, so angles less than about `5°` should not be used.

Cone Limit Functions

The reset function defaults to using the x-axes of the two body frames and limits the angle between them to PI radians, which is effectively no limit.

Accessors

The accessor functions specific to the Cone Limit are:

```
MeReal MEAPI MdtConeLimitGetConeHalfAngle(const MdtConeLimitID j);
```

Return the cone half angle; i.e. the angle between the cone axis and the side of the cone.

```
MeReal MEAPI MdtConeLimitGetStiffness(const MdtConeLimitID j);
```

Return the current limit stiffness.

```
MeReal MEAPI MdtConeLimitGetDamping(const MdtConeLimitID j);
```

Return the current limit damping.

Mutators

The mutator functions specific to the Cone Limit are:

```
void MEAPI MdtConeLimitSetConeHalfAngle(const MdtConeLimitID j,
                                        const MeReal theta );
```

Set the limit cone half angle angle to `theta`. The cone half angle is the angle between the cone axis and the side of the cone. Defaults to ME_PI at reset.

```
void MEAPI MdtConeLimitSetStiffness(const MdtConeLimitID j, const MeReal k_p);
```

Set the limit stiffness to k_p.

```
void MEAPI MdtConeLimitSetDamping(const MdtConeLimitID j, const MeReal k_d);
```

Set the limit stiffness to k_d.

Joint Limit: MdtLimit

A Joint is defined as a constraint on the free movement of bodies relative to one another. To achieve this, joints must themselves be constrained. The constraint of a joint is called a *Limit*. Each joint may have one or several limits. In practice, 2 limits would be a sensible maximum number of limits. Adding more limits would not further constrain the system, it would just increase the size of the constraint matrix that would need to be solved, slowing down the simulation. Adding limits to a prismatic or hinge joint, where a high and low limit are set, will add a row to the constraint matrix which is equivalent to losing one degree of freedom. In Karma, each limit is represented by a `MdtBclLimit` structure and is defined by a list of parameters that describe its action on a joint.

The Mdt Library provides a set of accessors/mutators and indicators/actuators to interact with the `MdtBclLimit` structure. Unlike a mutator, an actuator acts like a simple on/off switch, and does not require any value other than a boolean value. An indicator acts like an indicator light, telling you if an actuator is on or off by returning the appropriate boolean value.

Accessors:

```
MeReal MEAPI MdtLimitGetPosition( const MdtLimitID limit );
```

Return the relative position of the bodies attached to the joint.

```
MeReal MEAPI MdtLimitGetVelocity( const MdtLimitID limit );
```

Return the relative velocity of the bodies attached to the joint.

```
MeReal MEAPI MdtLimitGetStiffnessThreshold( const MdtLimitID limit );
```

Return the limit stiffness threshold. Please refer to`MdtLimitSetStiffnessThreshold()` in the mutator section.

```
MeReal MEAPI MdtLimitGetMotorDesiredVelocity( const MdtLimitID limit );
```

Return the desired velocity of the motor. A lower limiting velocity may be achieved if the attached bodies are subject to velocity or angular velocity damping.

```
MeReal MEAPI MdtLimitGetMotorMaxForce( const MdtLimitID limit );
```

Return the maximum force that the motor is allowed to use to attain its desired velocity.

Mutators

```
void MEAPI MdtLimitSetLowerLimit( const MdtLimitID limit,
                                  const MdtSingleLimitID sl );
```

Set the lower limit properties by copying the single limit data into the `MdtBclLimit` structure. If the lower limit stop is higher than the current upper limit stop, the latter is also reset to the new stop value.

```
void MEAPI MdtLimitSetUpperLimit( const MdtLimitID limit,
                                  const MdtSingleLimitID sl );
```

Set the upper limit properties by copying the single limit data into the `MdtBclLimit` structure. If the upper limit stop is lower than the current lower limit stop, the later is also reset to the new stop value.

```
void MEAPI MdtLimitSetPosition( MdtLimitID limit, const MeReal NewPosition );
```

This sets an offset that is used to transform the measured relative position coordinate into the user's coordinate system. It does not change the actual position or orientation of any modelled object.

```
void MEAPI MdtLimitSetStiffnessThreshold( const MdtLimitID limit,
                                          const MeReal NewStiffnessThreshold );
```

Set the limit stiffness threshold. When a limit stiffness exceeds this value, damping is ignored and only the restitution property is used. When the limit stiffness is at or below this threshold, restitution is ignored, and the stiffness and damping terms are used to simulate a damped spring. The stiffness threshold is enforced to be non-negative: the initial value is `MEINFINITY`.

```
void MEAPI MdtLimitSetLimitedForceMotor( const MdtLimitID limit,
                                         const MeReal desiredVelocity,
                                         const MeReal forceLimit );
```

Set the limited-force motor parameters, enforcing a non-negative value of `forceLimit`. If the latter is zero, this service deactivates the motor: otherwise, the motor is activated. This service does not enable attached disabled bodies.

Actuators

```
void MEAPI MdtLimitCalculatePosition( const MdtLimitID limit,
                                      const MeBool NewState );
```

Set or clear the "calculate position" flag without changing the limit's activation state. Note that if the limit is currently activated or powered, the "calculate position" flag cannot be cleared.

```
void MEAPI MdtLimitActivateLimits( const MdtLimitID limit,
                                   const MeBool NewActivationState );
```

Activate (if `NewActivationState` is non-zero) or deactivate (if zero) the limit, without changing any other limit property.

```
void MEAPI MdtLimitActivateMotor( const MdtLimitID limit,
                                  const MeBool NewActivationState );
```

Activate (if `NewActivationState` is non-zero) or deactivate (if zero) the limited-force motor on this joint axis, without changing any other limit property.

Indicators:

```
MeBool MEAPI MdtLimitIsActive( const MdtLimitID limit );
```

Returns non-zero if the corresponding degree of freedom of the joint (i.e. the joint position or angle) has a limit imposed on it, and zero if it does not. Most joints have more than one degree of freedom. Joint limits are inactive by default, and will not affect the attached bodies until activated and non-zero stiffness and/or damping properties are set.

```
MeBool MEAPI MdtLimitPositionIsCalculated( const MdtLimitID limit );
```

Returns non-zero if the position or angle of the corresponding degree of freedom of the joint is to be calculated, and zero if it is not calculated. If the degree of freedom is either limited or actuated (i.e. powered), the joint position must be calculated. By default, joint positions are not calculated.

```
MeBool MEAPI MdtLimitIsMotorized( const MdtLimitID limit );
```

Returns non-zero if the limit is motorized, and zero if it is not. Joint limits are motorized by default.

The Single Joint Limit: MdtSingleLimit

Each limit contains two sub-structures of type `MdtBclSingleLimit`:

Structure Member	Description
`MeReal damping`	The damping term (k_d) for this limit. This must not be negative. The default value is zero. This property is used only if the limit hardness is less than or equal to the damping threshold. If the hardness and damping of an individual limit are both zero, it is effectively deactivated.
`MeReal restitution`	The ratio of rebound velocity to impact velocity when the joint reaches the low or high stop. This is used only if the limit hardness exceeds the damping threshold. Restitution must be in the range zero to one inclusive: the default value is one.
`MeReal stiffness`	The spring constant (k_p) used for restitution force when a limited joint reaches one of its stops. This limit property must be zero or positive: the default value is `MEINFINITY`. If the stiffness and damping of an individual limit are both zero, it is effectively deactivated.
`MeReal stop`	Minimum (for lower limit) or maximum (for upper limit) linear or angular separation of the attached bodies, projected onto the relevant axis. For a soft limit, the stop is a boundary rather than an absolute limit.

Accessors

```
MeReal MEAPI MdtSingleLimitGetDamping ( const MdtSingleLimitID sl )
```

Return the damping term (k_d) for this limit.

```
MeReal MEAPI MdtSingleLimitGetRestitution ( const MdtSingleLimitID sl )
```

Return the restitution of this limit.

```
MeReal MEAPI MdtSingleLimitGetStiffness ( const MdtSingleLimitID sl )
```

Returns the spring constant (k_p) used for restitution force when a limited joint reaches one of its stops.

Mutators

```
void MEAPI MdtSingleLimitReset ( const MdtSingleLimitID limit )
```

Initialize the individual limit data and set default values (`position = 0, restitution = 1, stiffness = MEINFINITY, damping = 0`).

```
void MEAPI MdtSingleLimitSetDamping ( const MdtSingleLimitID sl,
                                      const MeReal NewDamping)
```

Set the damping property of the limit. Damping is enforced to be non-negative. The initial value is zero.

```
void MEAPI MdtSingleLimitSetRestitution ( const MdtSingleLimitID sl,
                                    const MeReal NewRestitution )
```

Set the restitution property of the limit. Restitution is enforced to be in the range zero to one inclusive. The initial value is one.

```
void MEAPI MdtSingleLimitSetStiffness ( const MdtSingleLimitID sl,
                                    const MeReal NewStiffness )
```

Set the stiffness property of the limit. Stiffness is enforced to be non-negative. The initial value is `MEINFINITY`.

```
void MEAPI MdtSingleLimitSetStop ( const MdtSingleLimitID sl,
                                     const MeReal NewStop )
```

Set a limit on the linear or angular separation of the attached bodies.

How to Use Joints

The tutorial *Hinge* provided in the release is a simple example of a hinge joint powered by a limited-force motor. A body is connected to the world with the hinge joint.

The limits can either be *hard* (if the limit stiffness factor is high) or *soft*. In terms of the simulation, a hard bounce reverses a bodies' angular velocities in a single timestep, while a soft bounce may take many timesteps to reverse a bodies' angular velocity. If the limits are soft, damping can be set so that beyond the limits, the hinge behaves like a damped spring. If the limits are hard, the limit restitution can be set to between zero and one to govern the loss of angular momentum as the bodies rebound.

First, in the `main()` function, create a hinge joint in the world. When the hinge exists set its center of mass position at the origin and the orientation of its axis parallel to the x axis. By default `body[1]` is set to 0 i.e. the world.

```
hinge = MdtHingeCreate(world);
MdtHingeSetBodies(hinge, body, 0);
MdtHingeSetPosition(hinge, 0, 0, 0);
MdtHingeSetAxis(hinge, 1, 0, 0);
```

Get a handle to its `MdtLimit` structure and use it to toggle its limit and the limit's motor.

```
MdtLimitID limit = MdtHingeGetLimit(hinge);
MdtLimitActivateMotor(limit, !MdtLimitIsMotorized(limit));
MdtLimitActivateLimits(limit, !MdtLimitIsActive(limit));
```

Then, by accessing directly the sub structures `MdtSingleLimit` of the limit structure, set the lower and upper limit angles of the hinge. Set the maximum force to a value `maxForce` to give a desired velocity of a value `desiredVelocity.` Note that the maximum force may not be strong enough to reach the desired velocity.

```
MdtSingleLimitSetStop(MdtLimitGetLowerLimit(limit), LO_LIMIT);
MdtSingleLimitSetStop(MdtLimitGetUpperLimit(limit), HI_LIMIT);
MdtLimitSetLimitedForceMotor(limit, desiredVelocity, maxForce);
```

Finally, enable the hinge joint, allowing it to be computed at the next update.

```
MdtHingeEnable(hinge);
```

The main difference between contacts and joints is that joints are most often created once only, as in *hinge* in its `main()` function, but contacts must be created and destroyed repeatedly, either by a user created function, as in *Bounce*'s callback function `tick(),`or by using Karma Collision and the Karma Simulation Toolkit in conjunction with Karma Dynamics.

Creating Contacts: MdtContact

Every time two bodies touch or intersect each other, a *contact* must be created. The contact constraint will ensure that there is no relative motion in the direction opposite to the contact normal. The solver will compute the constraint force required to prevent the bodies from moving against the direction of the normal as well as the tangential friction forces that correspond to Karma's approximation of the Coulomb friction model. The type of friction used, the computed forces exerted by a contact on the bodies, and the position and normal of the contact, form part of the information stored in a `MdtBclContactParams` structure. This is pointed to by an `MdtContactID` handle.

Most contacts will not be actual contact points. This is because the dynamics rigid body solver may leave bodies in slightly overlapping positions. Therefore, Karma uses the idea of an "effective contact point".

For example the geometrically-visible points of contact between two shallowly-intersecting spheres forms a circle, but the best "effective contact point" to communicate to the solver would correspond to a point at the center of this circle.

These contact point directions may or may not correspond exactly to points of contact between the two bodies in terms of the visually-rendered appearance, but are a way of summarizing the "inter-surface relationship" in a way that is both efficient for the rigid body solver, and that produces the expected non-interpenetration behavior.

Interpenetration and Contact Strategies

Interpenetration of two surfaces must be prevented by carefully choosing an appropriate set of contact points. The choice depends on the type of contact behavior. For a bouncing sphere, one contact should suffice. Resting and sliding usually require three contacts.

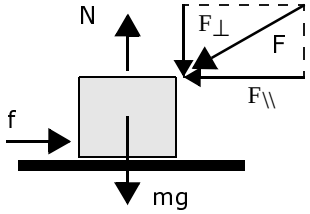
Karma Collision offers a number of alternative contact strategies that produce contacts for different types of situation that can be selected as necessary. Strategies that use fewer contact points have the advantage that MdtKea solves them more quickly. For those using a third-party collision package or their own in-house custom collision routines, a contact strategy must be devised that models behavior realistically without creating too many contact points.

Simulating Friction

After the contact configuration of a set of bodies is determined, the behavior of contact points must be specified. Up to three forces may be present at a contact point.

These forces act on the bodies that are in contact. The first is a normal force that prevents the bodies from penetrating each other. The other two are tangential friction forces (that act at right angles to one another and to the normal force) that prevent the bodies from sliding against each other along the contact plane.

Coulomb Friction



Friction is a term used to describe the macroscopic effect of the resistance to motion that a body experiences. The origin of friction lies in the effect of physical processes that takes place between materials on the atomic scale. It is the electrostatic interaction between atoms and molecules comprising a material that gives rise to friction. Friction is a dissipative process that converts energy from motion to heat and sound.

Friction can conveniently be categorised as viscous friction and dry friction. Viscous friction occurs when the contact is wet i.e., when there is a lubricant at the contact like oil. Dry friction occurs when the surfaces are dry i.e., when two solid surfaces are in direct contact without any lubricant.

Viscous friction produces a force opposite to the motion of the objects in contact with a magnitude related to the relative sliding velocity. Viscous friction does not keep objects from sliding but it can slow them down. Note also that viscous friction depends on the relative velocity of the objects and produces more force at high relative velocity.

Dry friction produces a very different sort of behavior that has two aspects, namely kinetic (dynamic) friction, and static friction. A sliding object subject to dry friction experiences kinetic friction during its sliding phase and static friction once it has come to rest. Importantly, static friction prevents sliding from rest altogether while the applied tangential force acing on the object is below a threshold value. During sliding the kinetic dry friction opposes the motion with a force which (unlike viscous friction) does not depend on the relative velocity of the objects. The static threshold force is usually a little higher than the kinetic friction force.

Measurements of dry friction show that the magnitude of the kinetic friction force increases with the contact load, as does the static threshold friction force. The usual analytical model of dry friction, referred to as Coulomb friction, approximates this dependence on normal force as a linear relationship.

What follows is a simple description of how Coulomb friction works. Imagine that the bodies in contact are a box and the ground: the box is sitting on the ground. The normal force $\vec{N}$ is the force exerted by the ground in reaction to the gravitational field strength $m\vec{g}$ plus the perpendicular (relative to the ground) component of the total external force, $\vec{F}_{\perp}$, exerted upon the object. : These forces must be equal to prevent any movement perpendicular to the ground. The force $\vec{f}$ is the force of friction that spontaneously opposes the tangential (parallel to the ground) component of the external force, $\vec{F}_{\backslash\backslash}$.

Assuming that the box is initially at rest, then these forces are in equilibrium and the resulting net force is zero, as long as $\vec{F}_{\parallel}$ is smaller than the value $\mu_s N$, where μ_s is the static friction coefficient, the force $\vec{f}$ scales with the force $\vec{F}_{\parallel}$ such that they end up being equal to each other.

As soon as the tangential component of $\vec{F}$ becomes larger than $f_{smax} = \mu_s N$, the frictional force fails to scale up with $\vec{F}_{\parallel}$ and the force imbalance will cause the box to start sliding. Once the box is in motion the frictional force $\vec{f}$ still impedes the movement of the box but to a lesser degree. This new frictional force is called *kinetic* friction and is written $f_k = \mu_k N$ where μ_k is the dynamic coefficient of friction and, in general, $\mu_k < \mu_s$. In short, the two possible cases are:

- At rest => static friction: $f \le f_{smax} = \mu_s N$
- In motion => kinetic friction: $f = f_k = \mu_k N$

Two quantities, the normal force $\vec{N}$ and the friction coefficient μ, are thus directly related to the severity of the friction force. The normal force $\vec{N}$ is related to the weight of the object and to the forces exerted upon this object: the larger $\vec{N}$ is, the larger the friction. The friction coefficient is related to the relative smoothness of the surfaces in contact: the smaller μ is, the smoother the contact of the surfaces is and the lesser the friction between the two.

Note that the Coulomb friction law is itself merely an approximation to the way that real objects behave in contact. It does not have the same status as fundamental physical laws like Newton's law.

Friction in MdtKea

Various rigid body dynamics libraries try to simulate Coulomb friction forces. However, there are a number of serious difficulties with this since the Coulomb friction model is neither well-posed nor consistent. That is, some contact problems have multiple valid answers and some other problems have no valid answer consistent with the constraints and the Coulomb model.

Out of the several approximations available, Karma uses one that is both stable and efficient. However, anisotropy in the friction force may be observed. This can be overcome by reorienting the friction box. A friction model based on the normal force is included (see bullet 4 below) in this version of Karma.

Karma Dynamics supports three main friction modes:

- Frictionless

  This means that no tangential friction force is applied at all, so the contacting objects are free to slide over each other. This is equivalent to setting $f_{smax} = f_k = 0$.

- Infinite friction

  The tangential friction forces are applied with no upper limit, so that the contacting objects can not slide over each other at all. This is equivalent to setting $f_{smax} = \infty$.

- Box friction

  The tangential friction forces are applied with an upper limit (`friction1` and `friction2` for each friction direction). This can be done in either or both of the tangential friction directions. It is a simple approximation to friction because it does not depend on the normal force and is therefore equivalent to setting f_{smax} to a constant value.

- Normal Force Friction

  Normal Force friction is a refinement of box friction where at each timestep the size of the friction box is determined by multiplying the coefficient of friction between the two bodies by the normal force computed during the previous timestep. For bodies in resting contact, this adaptively configures box friction to provide a more faithful approximation to Coulomb friction.

Slip: An Alternate Way to Model Friction

Like friction, slippiness and slidiness act in the plane of contact. When setting slidiness on a contact, then one body will act like a conveyer belt, carrying the other body along its surface.

Slippiness is a property that can be useful in modeling certain special effects, such as the sideways motion of a rolling tire. When applying force to a “slippy” object, the object reaches a proportional velocity immediately; it does not accelerate to that velocity.

In Kea’s box friction model, force applied along the friction direction of two objects in contact will cause them to accelerate along that direction (when the applied force exceeds some threshold).

When setting Kea’s slip property (`slip1` and `slip2` are defined in the `MdtBclContactParams` structure), the result is a different behavior: an applied force along the friction direction will result in a relative velocity in that direction between the objects. The velocity will be the force multiplied by the slip factor, so that $f_{smax} = f_k = 0$ and $\vec{V} \propto \vec{F}_{\backslash\backslash}$ where $\vec{V}$ and $\vec{F}_{\backslash\backslash}$ are the tangential component of the velocity and of the external force respectively.

This is useful, for example, when modeling the tires of a vehicle. Applying slip along the transverse direction of the tire contact point with the ground, provides a good model of “tire slip” that will result in improved vehicle behaviour. The slip should be set to a value that is proportional to the rotational velocity of the tire. If the vehicle is not moving, set the slip to zero.

Slip is faster to simulate than friction, because there is no discontinuity in the resistive force.

Functions that are specific to Contacts

A handful of function are specific to contact constraints. This section lists all those functions and comments on them:

Accessors

```
void MEAPI MdtContactGetNormal (const MdtContactID contact,
                                                MeVector3 normalVec);
```

Return the contact normal of `contact` in `normalVec`, in the world reference frame.

```
MeReal MEAPI MdtContactGetPenetration (const MdtContactID contact)
```

Return the current penetration depth at this contact.

```
void MEAPI MdtContactGetDirection (const MdtContactID contact,
                                                  MeVector3 directVec)
```

The contact primary direction is returned in `directVec`, in the world reference frame.

```
MdtContactParamsID MEAPI MdtContactGetParams (const MdtContactID contact)
```

Return a `MdtContactParamsID` pointer to the contact parameters of this contact.

The following function is only used in conjunction with Karma Collision.

```
MdtContactID MEAPI MdtContactGetNext (const MdtContactID contact);
```

Return the pointer to the next contact associated with this body pair if it was set in collision. If the return value is equal to `MdtContactInvalidID` then no next contact exists.

Mutators

The following functions are mutators specific to contacts:

```
void MEAPI MdtContactSetBodies (const MdtContactID contact, const MdtBodyID
                          body1, const MdtBodyID body2);
```

Set the contact bodies `body1` and `body2` to be attached to `contact`.

```
void MEAPI MdtContactSetNormal (const MdtContactID contact, const MeReal
                          xNorm, const MeReal yNorm, const MeReal zNorm)
```

Set the contact normal of `contact` to `(xNorm, yNorm, zNorm);`

```
void MEAPI MdtContactSetPenetration ( const MdtContactID contact,
                                      const MeReal penetration );
```

Set the value `penetration` of the penetration depth at the contact `contact`.

```
void MEAPI MdtContactSetParams ( const MdtContactID contact,
                                 const MdtContactParamsID parameters );
```

Utility for setting all contact parameters. This allows the user to set all values in the `MdtContactParams` structure at once.

```
void MEAPI MdtContactSetNext ( const MdtContactID contact,
                               const MdtContactID nextContact );
```

Set the pointer of `contact` to the next contact `nextContact`. Used by Mcd collision.

```
void MEAPI MdtContactSetDirection ( const MdtContactID contact,
               const MeReal xDir, const MeReal yDir, const MeReal zDir );
```

Set the primary direction for this contact at `(xDir, yDir, zDir)`.

This is only necessary if surface properties are to vary depending on the direction. For isotropic contacts, this function should not be called, and the primary and secondary parameters should be set to the same value. The direction should always be perpendicular to the given normal, and the secondary direction is perpendicular to the primary direction.

MdtContactGroups

To simplify management of contacts, Karma organises contacts into ContactGroups. A contact group is a constraint between two bodies, or a body and the world, that holds all the contacts between those bodies. This makes it easy to deal with the contacts as a group.

Functions that are specific to MdtContactGroups

Accessors.

```
MdtContactID MEAPI MdtContactGroupGetFirstContact (
                                    MdtContactGroupID group );
```

Return the first contact in a contact group, or NULL if the contact group is empty.

```
MdtContactID MEAPI MdtContactGroupGetNextContact (
                      MdtContactGroupID group, MdtContactID contact )
```

Return the contact following *contact* in the contact group, or NULL if *contact* is the last contact.

```
int MEAPI MdtContactGrouptGetCount ( MdtContactGroupID group )
```

Return the number of contacts in the group.

```
MeReal MEAPI MdtContactGroupGetNormalForce ( MdtContactGroupID group )
```

Return the magnitude of the last timestep's normal force between the two bodies connected by the group.

```
MeI32 MEAPI MdtContactGroupGetSortKey( const MdtContactGroupID group );
```

Return the sort key of *contactgroup.*

Mutators

The following functions are mutators specific to contacts:

```
MdtContactID MEAPI MdtContactGroupCreateContact (MdtContactGroupID
                                            group);
```

Create a new contact and add it to the contact group. Returns the new contact.

```
void MEAPI MdtContactGroupDestroyContact (MdtContactGroupID group,
                        MdtContactID contact);
```

Remove *contact* from the contact group.

```
void MEAPI MdtContactGroupAppendContact ( MdtContactGroupID group,
                                    MdtContactID contact);
```

Append *contact* to the contact group.

```
void MEAPI MdtContactGroupRemoveContact ( MdtContactGroupID group,
                                  MdtContactD contact );
```

Remove *contact* from the contact group, but don't delete it.

```
void MEAPI MdtContactGroupSetSortKey( const MdtContactGroupID group,
                                                          MeI32 key );
```

Assign a sort key to *contactgroup.*

The MdtBclContactParams Structure

All the properties of a given contact between two objects are stored in a `MdtBclContactParams` structure, such as the contact type, the friction model or the coefficient of restitution.

The list of members of the `MdtBclContactParams` structure follow.

Member	Description
`MdtContactType type`	Contact type (zero, 1D or 2D friction)
`MdtFrictionModel model1`	Friction model to use along primary direction.
`MdtFrictionModel model2`	Friction model to use along secondary direction.
`int options`	Bitwise combination of MdtBclContactOption's.
`MeReal FrictionCoefficient`	The friction coefficient for use in the normal force friction model.
`MeReal restitution`	Restitution parameter.
`MeReal velThreshold`	Minimum velocity for restitution.
`MeReal softness`	Contact softness parameter (soft mode). Violates ideal constraint, allows penetration and gives a ‘springy’ effect as well. It can cause objects to take longer to come to rest. Values range from 0 to 1, with 1 being very soft and .1 or .001 being typical.
`MeReal max_adhesive_force`	Contact maximum adhesive force parameter (adhesive mode). Sticky contacts may not work very well because when the penetration of a contact goes negative (the contact has separated) the contact is destroyed, and the 'sticky' force can't pull the objects back together again.
`MeReal friction1`	Maximum friction force in primary direction.
`MeReal friction2`	Maximum friction force in secondary direction.
`MeReal slip1`	First order slip in primary direction.
`MeReal slip2`	First order slip in secondary direction.
`MeReal slide1`	Surface velocity in primary direction.
`MeReal slide2`	Surface velocity in secondary direction.

There exist a large number of functions, mutators and accessors, to interact with this structure. These are listed in the `MdtContactParams.h` header file, in the reference manual. The more popular ones follow:

To apply the friction at a contact point:

```
void MEAPI MdtContactParamsSetType ( const MdtContactParamsID param,
                                     const MdtContactType conType );
```

Set the type `conType` of the contact parameters structure `param`.

Karma Dynamics supports three main friction modes:

Friction Type	Description
`MdtContactTypeFrictionZero`	Frictionless contact
`MdtContactTypeFriction1D`	Friction only along primary direction
`MdtContactTypeFriction2D`	Friction in both directions

When using `MdtContactTypeFrictionZero`, the direction or the coefficient of friction does not need setting.

When using `MdtContactTypeFriction2D`, friction acts in orthogonal directions on the plane of contact, and the properties for each direction can be set. The direction of a 2D contact will be set automatically.

The coefficients of friction can be specified separately in the primary and secondary directions. The primary and secondary directions are perpendicular to each other. To specify the primary direction, use:

```
void MEAPI MdtContactSetPosition ( const MdtContactID contact,
                       const MeReal x, const MeReal y, const MeReal z );
```

Set the primary direction for this contact. This is only necessary to define surface properties to vary depending on the direction. For isotropic contact, this function should not be called, and the primary and secondary parameters should be set to the same value. The direction should always be perpendicular to the given normal, and the secondary direction is perpendicular to the primary direction. The secondary direction is automatically set according to the right-hand rule.

To specify the friction model that a contact will use, use the function

```
void MEAPI MdtContactParamsSetPrimaryFrictionModel (
                                const MdtContactParamsID param,
                                const MdtFrictionModel fModel );
```

Set the friction model `fModel` to use along the primary direction.

:

Friction Model	Description
`MdtFrictionModelBox`	Box Model of friction (simplified Coulomb)
`MdtFrictionModelNormalForce`	Friction based on the normal force. Coulomb like.

To reset the contact structure to its default values, use:

```
void MEAPI MdtContactParamsReset ( const MdtContactParamsID param )
```

Initializes the contact parameters structure to the default values.

The following values are reset to their default values.

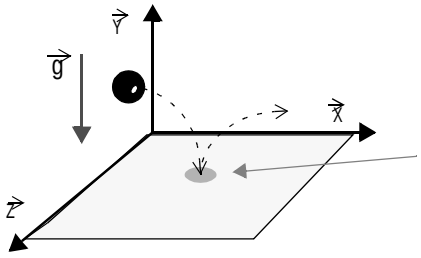
Member	Default Value
MdtContactType type	MdtContactTypeFrictionZero
MdtFrictionModel model1/model2	MdtFrictionModelBox
MeReal restitution	0.0
MeReal velThreshold	0.001
MeReal softness	0.0
MeReal max_adhesive_force	0.0
MeReal slip1/slip2	0.0
MeReal slide2/slide2	0.0

How to Use Contacts: Bounce.c

For simple programs (such as ones showing sphere-sphere, sphere-plane, or box-plane collisions), collision detection can be carried out with a few lines of code. But for anything more complex, a collision detection package such as Karma Collision may be needed.

After detecting a collision, Karma Dynamics uses contact constraints to determine the appropriate collision response. Contact structures (`MdtContact`) can be created and enabled using functions in the Mdt Library. When using Karma Collision and the Karma Simulation Toolkit, Karma will do both for you. For more information about Karma Collision, see the *MathEngine Karma Collision Developer's Guide*. For more information about the Karma Simulation Toolkit, see the *Mathengine Karma Simulation Toolkit Developer's Guide*.

In the Karma tutorial `Bounce.c`, a straightforward use of contacts is demonstrated - Karma Collision is not used in this example. A ball is thrown onto a surface and rebounds each time it touches the surface, with consecutive smaller amplitudes. To achieve this effect, a callback function `called tick()` that is called by MeViewer is responsible for checking for physical intersection between the ball and the surface. Each time an intersection is detected a contact must be created. It is important to remember that a contact, unlike a joint, is not a permanent constraint, since it is created dynamically whenever two objects come into physical contact, and is destroyed when the contact is no longer needed.



Once an impact is detected between the ball and the plane, a *contact* is created.

First, the `tick()` function checks for a previously created contact. An old contact contains dated informations that is useless and takes space in memory. It should be destroyed as soon as the two objects are no longer in contact:

```
if (contact){
   MdtContactDisable(contact);
   MdtContactDestroy(contact);
   contact = 0;
}
```

When a joint or a contact is no longer needed, it can be removed with the following function:

```
void MEAPI MdtContactDestroy( const Mdt*ID joint_or_contact);
```

This function destroys the joint or contact named `contact, where * represents the joint or contact type.`

To determine if intersection between the ball and the plane took place the position of the ball relative to the plane (positioned at y=0) and the radius of the ball are needed.

```
MdtBodyGetPosition(body, pos);
penetration = 0 - (pos[1] - ballRadius);
```

If positive, the value of `penetration` is the distance of penetration of the ball through the plane. If it is negative there is no physical intersection between the two. A contact constraint is created if there is an intersection. The first body in the contact is the ball, the second is the world (identified as `0`) since the plane is static and attached to the world.

```
if (penetration > 0){
   MdtContactParamsID params;
   contact = MdtContactCreate(world, body, 0);
```

A contact can only be created through the function `MdtContactCreate()`. First create a `MdtContactID` variable called `contact` that will point to the `MdtContact` structure where all information about that contact will be stored.

The function that creates a contact and returns a `MdtContactID` variable is:

```
MdtContactID MEAPI MdtContactCreate ( const MdtWorldID world );
```

This function creates a new joint or contact in the world.

The position of the contact must be specified in world coordinates and is set to the value of the ball center of mass translated downward by the value `ballRadius`. The normal of the contact is the plane's normal, which is the y-axis. To compute the rebound amplitude, the penetration depth must be communicated to Karma dynamics.

```
MdtContactSetPosition(contact, pos[0], pos[1] - ballRadius, pos[2]);
MdtContactSetNormal(contact, 0, 1, 0);
MdtContactSetPenetration(contact, penetration);
```

The formal description of these contact mutators are

```
void MEAPI MdtContactSetNormal ( const MdtContactID contact,
            const MeReal xNorm, const MeReal yNorm,const MeReal zNorm );
```

Set the contact normal of `contact` to `(xNorm, yNorm, zNorm)`.

and

```
void MEAPI MdtContactSetPenetration ( const MdtContactID contact,
                                      const MeReal penetration );
```

Set the value `penetration` of the penetration depth at the contact `contact`.

First obtain a `MdtContactParams` structure related to the contact

```
params = MdtContactGetParams(contact);
```

Formal description:

```
MdtContactParamsID MEAPI MdtContactGetParams ( const MdtContactID contact );
```

Return a `MdtContactParamsID` pointer to the contact parameters of this contact.

When a `MdtContactParams` structure exists, assign contact properties to it, such as the friction type, the friction value and restitution.

```
    MdtContactParamsSetType(params, MdtContactTypeFriction2D);
    MdtContactParamsSetFriction(params, (MeReal)(5.0));
    MdtContactParamsSetRestitution(params, (MeReal)(0.6));
}//end if(penetration>0)
```

Finally, attach this new contact to the world before updating the world by one timestep. Remember that if the contact is not enabled before the world is updated, the Kea solver will ignore it.

```
MdtContactEnable(contact);
MdtWorldStep(world, step);
```

The `MdtContactEnable()` and `MdtContactDisable()` are not functions, but macros that were implemented to save writing two lines instead of one.

Convert a contact or a joint to a constraint ID to enable or disable it, as shown below:

```
MdtConstraintID cid = MdtContactQuaConstraint(contact);
```

and then

```
MdtConstraintEnable(cid);
MdtConstraintDisable(cid);
```

Another way to do this is:

```
MdtContactEnable(contact);
MdtContactDisable(contact);
```

Chapter 4 • Advanced Features, Optimization and Utilities

Overview

In this chapter Karma’s more advanced features and functions are discussed.

Adding Forces, Torques and Impulse to a Body

Each body has a force accumulator that is used to sum the forces to be applied during `MdtWorldStep()` when the new velocity and position of bodies are calculated. After MdtWorldStep the force accumulator is reset to zero. Several forces applied to a body are summed in the force accumulator, and do not increase the required CPU time when the body to which they are applied is simulated.

If the body is not simulated until after 10 time increments, the forces applied to it during this time are summed up. Each force is applied for the duration of the time step, not the sum of the time steps since the force was applied. Because the force applied is reset to zero after the simulation of the body, to apply a constant force to body it must be added for each simulation step.

```
void MEAPI MdtBodyAddForce ( const MdtBodyID body,
                    const MeReal fx, const MeReal fy, const MeReal fz );
```

Add a force vector to a body. The force `(fx,fy,fz)` is applied at the centre of mass and is set in the world reference frame.

To apply a force at a specific location, use:

```
void MEAPI MdtBodyAddForceAtPosition (  MdtBodyID body,
                  const MeReal Fx, const MeReal Fy, const MeReal Fz,
                  const MeReal Px, const MeReal Py, const MeReal Pz );
```

Apply a force `(Fx,Fy,Fz)` at a specific location **P** =`(Px,Py,Pz)`, both defined in the world reference frame. For a contact force, such as pushing a body, **P** would lie on the surface of the body. Mathematically, and in Karma, **P** can lie anywhere. When applied outside of the volume enclosed by the body, the non-contact force might represent an electromagnetic force.

To apply a torque to a body use:

```
void MEAPI MdtBodyAddTorque ( const MdtBodyID body,
                    const MeReal Tx, const MeReal Ty, const MeReal Tz );
```

Apply a torque `(Tx,Ty,Tz)` to a body. The torque vector is set in the world reference frame.

To apply a definite amount of momentum to a body (also called an *Impulse*) use the following functions:

```
void MEAPI MdtBodyAddImpulse ( const MdtBodyID body,
                    const MeReal Ix, const MeReal Iy, const MeReal Iz );
```

Add an instantaneous impulse vector `(Ix,Iy,Iz)` to the center of mass of a body.

```
void MEAPI MdtBodyAddImpulseAtPosition ( const MdtBodyID body,
                      const MeReal Ix, const MeReal Iy, MeReal Iz,
                     const MeReal Px, const MeReal Py, const MeReal Pz);
```

Apply an instantaneous impulse `(Ix,Iy,Iz)` at a specific point `(Px,Py,Pz)` on a body, both defined in the world reference frame.

Note that impulse is time step independant. The impulse imparts a specific amount of momentum (force x time) to an object. Impulse is useful for simulating fast impacts such as gun shots.

Enabling/Disabling Bodies

The Mdt Library can check for moving objects at each time step. If an object is found to be motionless, i.e. not interacting with other objects and with no applied motion inducing forces acting on it, it will be turned off (disabled), thus saving CPU cycles. This feature can be turned on or off with the function:

```
void MEAPI MdtWorldSetAutoDisable ( const MdtWorldID world,
                                    const MeBool switch );
```

Turns on and off the AutoDisable feature. `switch=0` indicates *off*, `switch=1` indicates *on*.

An `MdtAutoDisableParams` structure contains the `linear velocity, linear acceleration, angular velocity and angular acceleration threshold` values (`vel_thresh`, `acc_thresh`, `velrot_thresh and accrot_thresh respectively)` and `alive_window` members that are used to determine when bodies are at rest, and should be disabled. A body is disabled if it falls below all of the world threshold values. A disabled body is not processed by the solver until re-enabled by a force or collision event.

To determine if an object is awake *(*enabled), Karma checks whether the force and torque values exerted on that object are larger than the force and torque threshold values (`force_thresh` and `torque_thresh)` required to wake up resting bodies.

To set the number of evolve steps before disabling an object, after it has been awakened, use:

```
void MEAPI MdtWorldSetAutoDisableAliveWindow
                        ( const MdtWorldID world, const int alWin );
```

Sets the minimum number of steps to keep objects alive for. This is needed to give objects enough time to gain a minimum velocity after being awakened.

The following functions are used to Get / Set the threshold values for the world objects. Note that all four threshold criteria must be satisfied for a body to be disabled.

The linear velocity threshold, below which a body will be considered “at rest” and therefore disabled:

```
void MEAPI MdtWorldSetAutoDisableVelocityThreshold
                    ( const MdtWorldID world, const MeReal velThresh);
```

Set the velocity threshold below which an object will be disabled. The `velThresh` value is the square of the magnitude of the objects velocity.

The linear acceleration threshold, below which a body will be considered “at rest” and therefore disabled:

```
void MEAPI MdtWorldSetAutoDisableAccelerationThreshold
                    ( const MdtWorldID world, const MeReal accThresh);
```

Set the acceleration threshold below which an object will be disabled. The `accThresh` value is the square of the magnitude of the objects acceleration.

This function sets the angular velocity threshold below which a body will be considered “at rest” and therefore disabled:

```
void MEAPI MdtWorldSetAutoDisableAngularVelocityThreshold
                   ( const MdtWorldID world, const MeReal angVelThresh);
```

Set the angular velocity threshold below which an object will be disabled. The value *angVelThresh* is the square of the magnitude of the objects angular velocity.

This function sets the angular acceleration threshold below which a body will be considered “at rest” and therefore disabled:

```
void MEAPI MdtWorldSetAutoDisableAngularAccelerationThreshold
                    (const MdtWorldID world, const MeReal angAccThresh);
```

Sets the angular acceleration threshold below which an object will be auto-disabled. The `angAccThresh` value is the square of the magnitude of the objects angular acceleration.

Limiting the Matrix Size

When Kea solves for a group of bodies, it constructs a matrix representing the way the constraints limit the freedom of movement of the bodies. This matrix has one row for every degree of freedom limited by the constraints. If you have a large number of bodies connected by joints and contacts, the matrix size (and the consequent memory requirements) may be inconveniently large.

By setting a limit on the matrix size, you can instruct Karma to remove constraint rows from the simulation. This allows an application to degrade simulation fidelity to meet memory constraints. Deleting constraints may also decrease Kea's running time significantly..

```
void MdtWorldSetMaxMatrixSize ( const MdtWorld w, const int size);
```

Sets the maximum matrix size for Karma to attempt to reduce a partition

In order to try to meet this limit, Karma infers which constraint rows can be removed with least loss of fidelity. By default, first rows which enforce friction constraints are removed, then contacts between pairs of bodies, then contacts between bodies and the world.Contacts are removed in order of depth, from the shallowest penetration to the deepest. Joints are never deleted, and at least one contact is always left in each contact group.

It is possible to replace the scheme for ordering contacts in order of importance by supplying your own callback.

```
void MdtWorldSetContactImportanceCB ( const MdtWorldID w, const
                                      MdtContactImportanceCBPtr icb);
```

Set the callback pointer for computing relative importance of contacts

While a more aggressive constraint reduction strategy is possible, large constraint violations are more likely to be avoided by an application-specific strategy based on the properties of a particular simulation.

Sort Keys - Deterministic Simulation

Karma simulations use deterministic Newtonian dynamics to model the virtual world. Given this it is reasonable to expect that a simulation with identical starting conditions, run on a machine with a given configuration, would evolve in the same way provided there is no external interactivity with the system. Indeed, this is the case if a simulation is stopped and re-executed. However, if a Karma simulation is simply restarted, the simulation will almost certainly not proceed along an identical path each time. This is because the order in which information is passed to the constraint solver, and hence the order of the data on which calculations are performed, will very likely change when a dynamic object or collision model is removed from the simulation and then added again when it restarts. This is because Karma optimizes the arrangement of objects in memory which permutes the order in which they are tied to the solver.

For those who require that a simulation follow the same path, the order in which data is sent to the solver must be identical. Sort Keys are used to accomplish this. This allows Karma to deterministically order the input to the constraint solver. Sort keys must be positive and are in the range 0 to 2**15 - 1 for collision objects and 0 to 2**31 - 1 for dynamics objects. Similarly, joint keys run from 0 to 2**31 - 1.

The following mutators may be used to assign sort keys:

```
void MEAPI MdtBodySetSortKey(const MdtBodyID b, const MeI32 key);
void MEAPI MdtContactSetSortKey(const MdtContactID c, MeI32 key);
void MEAPI MdtContactGroupSetSortKey(const MdtContactGroupID c, MeI32 key);
void MEAPI MdtConstraintSetSortKey(const MdtConstraintID c, MeI32 key);
void MEAPI Mdt*SetSortKey(const Mdt*ID joint, MeI32 key);
```

Respective functionality to assign a sort key to a

- dynamic body
- contact
- contact group
- constraint
- a specific constraint / joint type where * represents one of Angular3, BSJoint, CarWheel, ConeLimit, FPFOJoint, FixedPath, Hinge, Linear1, Linear2, Prismatic, RPRO, Spring, Universal, UserConstraint.

The following functions access sort keys:

```
MeI32 MEAPI MdtBodyGetSortKey(const MdtBodyID b);
MeI32 MEAPI MdtContactGetSortKey(const MdtContactID c);
MeI32 MEAPI MdtContactGroupGetSortKey(const MdtContactGroupID c);
MeI32 MEAPI MdtConstraintGetSortKey(const MdtConstraintID c);
MeI32 MEAPI Mdt*GetSortKey(const Mdt*ID joint);
```

Respective functionality to return the sort key of a

- dynamic body
- contact
- contact group
- constraint
- a specific constraint / joint type where * represents one of Angular3, BSJoint, CarWheel, ConeLimit, FPFOJoint, FixedPath, Hinge, Linear1, Linear2, Prismatic, RPRO, Spring, Universal, UserConstraint.

Chapter 5 • Constructing Good Simulations

Overview

There are several issues to bear in mind when using Karma Dynamics so that well-behaved, efficient and reliable simulation is obtained. This applies when using the solver directly or through the higher level Abstraction Layer.

When simulating according to Newtonian dynamics, as in Karma, the system is deterministic. Hence there is an exact solution to the physical model that is being simulated. However, the numerical techniques employed will introduce inaccuracies, hence the final result will be an approximation to the actual physical behavior of the system. The degree of approximation can be reduced by being aware of the issues involved when implementing a simulation.

This chapter discusses techniques for creating good simulations. The information presented in this chapter is only a guide — simulations differ.

Integrators and First Order Effects

Given a set of differential equations, different numerical methods yield different solutions to these equations. Every numerical method is a compromise between the:

- required accuracy.
- robustness of the method against discontinuities (abrupt changes in the solutions), i.e. is it stable against stiffness?
- work needed to obtain a given accuracy.

The Kea solver uses a semi-implicit first order integrator that yields good stability and performance, when used correctly. This integrator is a good compromise between available integrators.

Background

The integrator is the part of Kea that evolves the rigid body system from its current state to a new state, h seconds into the future (h is the `stepsize` parameter in the Kea `stepimmediate()` function).

The order of a simulation's integrator is usually an important thing to know. If you are familiar with standard explicit integrators such as Euler and Runge-Kutta, you will know that higher order integrators are usually more accurate and stable. However, because they require more evaluations of the system forces for each time step, they are slower.

All integrators suffer from inaccuracy: that is, the simulated system doesn't behave exactly as the system would in real life. In lots of applications inaccuracy can be tolerated. An exception for example is where quantitative measurements are of critical importance for scientific or engineering applications.

Inaccuracy in explicit integrators may result in a closed system gaining energy. This can be observed when the objects in the simulation move faster and faster until they “explode”. This is called instability. Higher order explicit integrators have better stability, but are slower.

Kea's integrator is semi-implicit - it is not explicit - and first order. This gives a good compromise between stability and speed. This is because a semi-implicit integrator retains stability in a different way to an explicit integrator: the inaccuracy causes the system energy to fall rather than increase.

Because the Kea integrator is not fully implicit, there are still some situations where care has to be taken as with an explicit integrator: see *Stiff Forces and Stability* on page 95.

Some first order effects are listed below:

- As the step size h is changed, the behavior of a simulation will change.
- The difference between the behaviour predicted by the physical model and the simulation increases as h increases.

- With a higher order integrator the difference would be smaller for a given h.
- A smaller h results in a simulation running more slowly.
- Additional adjustments to the dynamics should be made at a h that will be appropriate to the frame rate that gamers will be running a game at.
- Adaptive time stepping schemes (variable h) may be difficult to control.

Stiff Forces and Stability

A “stiff” system is one where strong forces can be generated by small changes to the state of the system. A rigid body system with a very strong spring between bodies is an example of this. As the connected bodies are pulled apart or pushed together, a strong restoring force results.

Systems that are stiff are difficult for explicit integrators (like explicit Euler and Runge Kutta) to integrate, while maintaining stability and without increasing energy. Fully implicit integrators can integrate stably, because they remove energy instead of adding it when there are stiff components present.

Kea uses a semi-implicit integrator, that is a compromise between the explicit and implicit integration. Note that:

- Large forces caused by constraints will not result in stability problems.
- Large forces applied by the user to the rigid bodies (like spring forces) may cause problems.

A higher gravitational force should not cause any problems.

A stiff car suspension that would probably be unstable if simulated by applying a force to the wheels each timestep, can be simulated using the car-wheel constraint.

If applying forces to rigid bodies results in instabilities, try

- Reducing the magnitude of the forces.
- Limiting the force to a maximum value.
- Reduce epsilon (see next section).

The Meaning of Epsilon and Gamma

The Kea integrator provides two user parameters, `epsilon` (ε) and `gamma` (γ)**.** These can be adjusted to affect how the integrator steps the system from one state to the next.

Epsilon

Every time step, Kea must solve a matrix problem to determine the forces on the rigid bodies. ε is a variable that can improve Kea's ability to solve this problem.

Sometimes the matrix problem may not have a solution that satisfies all the constraints. For example, some systems can get into “singular” configurations, or redundant constraints may have been specified on the system (see *Avoiding Over-Determinancy* on page 104). When this happens there are two main symptoms:

- The simulation may jitter around or strange forces may appear with no apparent source. This is the result of errors in the approximate solution being amplified too much.
- The constraint solver may take too many iterations to find an approximate solution. On some platforms you will get a warning message indicating this.

In both cases the problem can be fixed by increasing ε. The only disadvantage of increasing ε is that it makes the joints and constraints a little bit springy instead of being solid. However this is usually not a problem. Values of `epsilon` (such as the default 0.0001) can be effective, and a value of ε near 1 is large and will almost certainly give visible springiness in the constraints.

ε may have to be decreased to make contacts 'harder' and prevent visible penetration:

- When dealing with large mass objects (please refer to *Mass Problems* on page 98). A heavy mass sitting on the ground may penetrate the ground.
- When using large gravitational forces.

ε should be decreased when using a small time step.

ε has units of time squared over mass. $1/\varepsilon$ is like a spring constant. If a system has no singularity, then ε can be set close to 0.

For those interested, every time step Kea solves a matrix equation of the form:

$$(A + \varepsilon I) \cdot x = b$$

where *A* is positive semi-definite matrix. I is the identity matrix. The parameter ε is added to the diagonal of the constraint matrix effecting the form of the solution.

Degenerate systems result in *A* being singular. If ε is zero, this equation may not have a solution. By making ε greater than zero a solution for *x* can be guaranteed.

Gamma

Every time step, after Kea has determined the forces on the rigid bodies, the positions and velocities of those bodies are updated. The way that this is done can cause the joints and other constraints to pull away from each other, so that objects will not be in their correctly constrained configurations in the new state.

Kea minimizes this problem through a relaxation process, which is controlled by γ, a positive number that is a relaxation rate.

When positions are updated, projection moves the rigid bodies γ of the way back to their correctly constrained configurations. γ applies to contacts as well - objects may penetrate a bit before a contact is generated, and the bodies then have to be projected apart again.

- If $\gamma = 0$, no relaxation is done. Joints will separate as the simulation progresses.
- If γ is large, a big correction is applied that attempts to zero the constraint violation. Make sure that the product γh (h is the time step) is some reasonable number, preferably less than 0.5 and certainly less than 1.0. A simulation may become unstable otherwise.

Try and keep γh constant when varying the time step. γ is more sensitive to time step variation than ε. For most simulations a value of $0.1<\gamma<0.8$ will work well. $1< \gamma < 0$ will cause problems.

Instability due to Mass and Inertia, and Numerical Scaling

Sometimes a simulation may become unstable whereby body positions and velocities become unrealistic, sometimes being very large, for no obvious reason. A poor distribution of mass can cause this.

Mass Problems

Consider an articulated system consisting of heavy and light objects. The heavy objects can transfer their energy to the light bodies causing them to move quickly because of conservation of momentum.

An example of this kind of system is a whip that is heavy near the handle and light near the tip. Flicking the handle causes a wave to travel down the length of the whip. As the wave travels it builds up speed. In a heavy whip the tip may travel faster than the speed of sound.

Instabilities will occur if bodies move faster than they can be reasonably simulated. This problem can be prevented by ensuring that the ratio of large to small masses in the system is not too high. For example, a ratio of 100:1 is a reasonable single precision maximum in many situations.

Inertia Problems

Objects having inertia tensors with components that are large on one axis and small on another axis are inherently unstable. Long thin objects are susceptible to gaining rotational energy on the long axis, about which the moment of inertia is low. This can lead to rotations that are difficult to simulate accurately. If an object is rotating quickly about a particular axis then the fast spin axis option should be used.

> **Note:** Version 1.x of Karma Dynamics internally overrides the inertia tensor set by the user with a uniform tensor that is a multiple of the identity matrix. This design choice has been made to reduce a number of stability problems that arise due to momentum transfer from high to low inertia axis. These can produce high angular velocities, especially during collisions, and make it harder to construct robust simulations. Alternative implementations of the solver that deal with full inertia tensors requires more computation and produces noticeably different effects only at high velocity. These alternative implementations may be incorporated in future versions of Karma.

Numerical Scaling

For Kea to simulate stably and efficiently, it is good to keep lengths and masses in a relatively small range around unity. How critical this is depends on the precision that is being used. Because one of the main requirements for Karma for the entertainment market is that it be fast, single precision float is used.

For float, it is sensible to keep all length and mass numbers in the range of 0.01 through 100, for example. For higher precision a much wider range is acceptable. This will help ensure that numerical error and inaccuracy do not become too much of a problem.

Preventing Jitter in Contacts

When two object collide, there will be some initial inter-penetration. The amount of penetration depends on how fast the two objects were going before they collided.

After collision, Kea's projection feature will push the objects apart to reduce the penetration to zero. However, sometimes Kea will push the objects too far, and the contact will be broken. This can be a problem, for example, when objects are resting on the ground. When the contact is broken, the object will “fall” a short distance into the ground, the contact will be re-made and the object will be pushed out again. This process can result in resting objects that jitter or twitch from time to time.

One solution is to set the softness option on the contact. This will cause two objects that are being forced together to naturally inter-penetrate slightly, preventing contact breaking:

- Set `keaContactOption` to `keaContactOptionSoft`.
- Use `mdtContactSetSoftness()` to set the degree of softness. A fairly small number, like 0.0001, is usually suitable. A larger number will result in more natural penetration.

There is no efficiency loss in using soft contacts.

Constraint Solver Iterations

Every time Kea takes a step, the constraint solver must go through a number of iterations to find a good solution for the forces on the rigid bodies. This is the only part of Kea that takes a variable amount of time: if it were not for the constraint solver, then each step would take the same amount of time for the same system.

Sometimes the constraint solver will fail to find a solution, and the warning message “cycle has been detected” or “The maximum number of iterations has been exceeded” will be output. Even with a warning from the solver the behavior will usually be stable enough. If there are any visible problems epsilon may need to be increased.

Sometimes the constraint solver takes a large amount of time when many objects make or break simultaneous contact, or when the contacts involve very large collision forces. Because of the large number of physical interactions taking place at a particular time, the amount of time required to do the physics can increase. There are two ways in the current version of Karma that will help here:

- Design the simulation carefully to try and avoid this situation
- Limit the constraint matrix size, as discussed in Chap 4.

Modeling Using Force-Limited Motors

The hinge and prismatic joints both have an optional motor that can be applied along the respective rotating or sliding axes, to control movement along that axis. To do this, the following parameters are available in the `MdtBclHingeJoint` and `MdtBclPrismaticJoint` structures:

Parameter	Description
`powered`	0 for an unpowered free joint, 1 for a powered joint.
`desired_vel`	desired angular or linear velocity of the joint.
`fmax`	maximum force that should be applied to reach the desired velocity.

When `powered` is set to 1, the solver tries to reach the desired velocity `desired_vel` on the joint. However, the maximum amount of force that can be applied along the joint axis to reach this velocity is `fmax`. This means that if the force limit is low or the desired velocity is large, the joint will take several time steps to reach the desired velocity, or possibly not reach the desired velocity. This action is somewhat similar to an engine with a maximum amount of output torque.

Force-limited motors provide a useful, stable way of getting joints to move. Using them is better than applying forces or torques to the joint bodies directly, because the joint velocity is controlled directly instead of the joint acceleration. It does however, add another constraint to the constraint matrix, increasing the computational cost.

To model dry friction in a joint set a target velocity of zero (`desired_vel` = 0), where the braking force is controlled by `fmax`. This provides a simple model of a disc brake - heat and other nonlinear effects are not included.

Joint Limits versus Contacts

Hinges and prismatic joints can have limits on their movement set to prevent self-collision.

Wherever possible, joint limits should be used rather than contacts to control the movement of joints. Using a contact to prevent movement of two objects connected by a joint, rather than joint limits, is more computationally costly. This is because more contacts are generated and there is additional collision detection.

Avoiding Over-Determinancy

Consider a rigid body system made up of bodies, joints and contacts. If there are more joints and contacts than are actually required to constrain the motion of the bodies, then the system is over-determined.

Consider the following examples.

- When simulating a box resting on a ground plane, three contact points between the box and the ground are enough to give good stability. More contacts points would result in an over-determined system, because the extra contacts are not needed to determine the box motion.
- A line of boxes resting on the ground, each box being in contact with its neighbor or neighbors. In this situation two contact points are needed between each box and the ground to give the correct motion. Extra ground-box contacts would result in over-determinacy.
- A hinge joint removes five degrees of freedom (DOFs) between the two connected bodies. Imagine modeling a hinge as two ball-and-socket joints spaced a little distance apart on the hinge axis. This constrains the bodies in the correct way, i.e. the correct hinged motion is produced, but the two joints together take away six DOFs (three each). This is one more than is necessary for a hinge.

Over-determined systems should be avoided because the solver can find it difficult to reach a solution for such systems: the solution may be inaccurate, or take extra time.

Ideally over-determinacy would never arise, because every rigid body system would be described optimally. This is not so easy to achieve in practice - it can be difficult to tell whether a given system is, or is not, over-determined.

Here are a few guidelines:

- Use the minimum number of constraints between objects that gives the correct behavior. The less constraints, the faster the simulation. Finding this minimum involves a certain amount of trial and error.
- Use the correct joint types for the required motion. E.g. don't use two ball-and-socket joints to model a hinge.
- Don't have conflicting joints between the same bodies, i.e. don't use more than one joint to constrain the motion of the bodies in the same way.
- If in doubt, increase `epsilon`. This will always have the effect of reducing the over-determinacy of any system.

Fast Spin Axis

Every rigid body can have an optional fast spin axis specified, about which it rotates. If this is set, and it must be done at each time step, it alters the way that the body's orientation is updated at the end of every time step.

This alternative “fast spin” update is more accurate in the case where the body is spinning quickly around the fast spin axis, and relatively slowly around the other axes. This is particularly useful in the case of a wheel on a car, that may be rotating very quickly. Using the standard orientation update may result in a large error that makes the wheel's hinge axis appear to bend.