

MathEngine™ Dynamics Toolkit 2.0 Developer's Guide

Alpha 0.0.5 – August 2000

© 2000 MathEngine PLC. All rights reserved.

MathEngine Dynamics Toolkit Developer's Guide.

MathEngine is a registered trademark and the MathEngine logo is a trademark of MathEngine PLC. All other trademarks contained herein are the properties of their respective owners.

This document is protected under copyright law. The contents of this document may not be reproduced or transmitted in any form, in whole or in part, or by any means, mechanical or electronic, without the express written consent of MathEngine PLC. This document is supplied as a manual for the MathEngine Dynamics Toolkit. Reasonable care has been taken in preparing the information it contains. However, this document may contain omissions, technical inaccuracies, or typographical errors. MathEngine, PLC does not accept responsibility of any kind for customers' losses due to the use of this document. The information in this manual is subject to change without notice.

Printed in Canada.

Contents

Preface

About this Manual	x
Audience for this Manual	x
Related Documentation	x
Related Software	xi
MathEngine Viewer.	xi
MathEngine Collision Toolkit.	xi
About MathEngine	xii
Contacting MathEngine.	xii
Conventions.	xiii
Units	xiii
Coordinate Systems	xiii
Type Conventions.	xiii
Typographical Conventions.	xiv

1 What Can the Dynamics Toolkit Do?

Overview	2
How Your Game Uses the Dynamics Toolkit	2
Main Components.	3
Mdt Library	4
Rigid Bodies: A Useful Abstraction	4
Articulated Bodies and Joint Constraints	5
Collision Detection and Contact Constraints.	5
Working with Collision Toolkit	6
Designing Efficient Simulations Using Mdt Library	6
Designing Efficient Simulations Using Mdt Source Code	6
MdtKea Library: the Kea Solver	8
Designing Efficient Simulations Using MdtKea.	8
MdtBcl Library: Constraint Library	10

2 Installing and Building

Overview	12
--------------------	----

Downloading MathEngine Toolkits.	12
Prerequisites.	13
Target Platforms	13
Development Platforms	13
Installation Directory Structure	14
Building the Toolkit Examples	15
Using Visual C++.	15
Using make	15
Using make: Platform-dependent notes	16

3 Simulating Rigid Bodies and Forces

Overview.	18
The Dynamics Toolkit's Mdt Library	19
Abstractions Implemented	19
Relationship to Other Libraries	19
Embedding Dynamics Toolkit in a 3D Application.	20
Rendering Software Drives Your Application.	20
Stepping Through the Simulation Using Callback Functions.	20
Managing and Allocating Memory for Mdt Library	20
Managing and Allocating Memory for MdtKea Library.	21
Initializing Your World	21
Setting the Timestep for the Simulation.	22
Setting Gravity for this World.	22
Defining a Body	22
Stepping Through Time	23
Cleaning Up.	24
OK, So We Simplified a Little....	25
Allocating and Managing Memory for Kea Solver	25
MathEngine Memory Manager API	26
Mdt Bodies Lack Geometric Properties.	26
Contacts	28
Creating and Enabling.	28
Setting Forces and Other Properties	30
Adding Forces	31
Modeling Friction	32
Slippiness and Slidiness.	33
Stepping Through Time	34

The MdtWorld Structure	35
Functions that Manage Your World	35
Mode of Operation	37
Partitioning the World	37
Enabling/Disabling Bodies	37

4 Constraints: Joints and Contacts

Overview	40
Types of Joints	41
Degrees of Freedom	41
Ball-and-socket (BS):MdtBSJoint	42
Hinge:MdtHinge	43
Hinge Limits	43
Prismatic (piston-like behavior): MdtPrismatic	45
Prismatic Limits	45
Prismatic Actuators	45
Fixed-Path: MdtFixedPath	46
Fixed-Path-Fixed-Orientation: MdtFPFOJoint	47
Universal: MdtUniversal	48
Linear1: MdtLinear1	49
Linear2: MdtLinear2	50
Constraints	51
Functions To Manage Contacts and Joints	52
Functions Common to Contact and to All Joints	52
Functions that are specific to Contacts	54
Functions that are specific to Ball-and Socket Joint	57
Functions that are Specific to Hinge Joint	58
Functions Specific to Fixed-Position-Fixed-Orientation Joint	59
Functions Specific to Fixed-Path Joint	60
Here is the accessor function specific to FixedPath:	60
Functions that are Specific to Prismatic Joint	60
Functions Specific to Linear2 Joint	61
Functions that are Specific to Universal Joint	62
Macros	63
How to Use Contacts	64
How to Use Joints	66

5 Customizing and Optimizing

Overview	68
Rigid Bodies	69
Reference Frames	69
Orientation	69
The MdtKea Functions	70
MdtKeaStep Parameters	70
MdtBcl Structure	71
MdtKeaBody Structure	73
Force	73
MdtKeaParameters: Kea's parameter structure	76
Contacts and Friction	78
Interpenetration and Contact Strategies	78
Simulating Friction	79
Coulomb Friction	79
Friction in MdtKea	80
Slip: An Alternate Way to Model Friction	81
MdtBclContactParams	82
MdtBclContact Structure	84
MdtBclLimit Structure	85
Joints	86
Structures for Joints in the Mdt Library	86
Structures to Manage Joints in the MdtBcl Library	86
Constraint Header Structure	87
You Can Add Your Own Joint Types	87
Joints Supported by the Dynamics Toolkit	88
Ball-and-Socket Joint: MdtBclBSJoint	88
Hinge Joint: MdtBclHinge	88
Prismatic Joint: MdtBclPrism	89
Car Wheel Joint: MdtBclCarWheel	89
Fixed-Path Fixed-Orientation Joint: MdtBclFPFOJoint	90
Linear1 Joint: MdtBclLinear1Joint	90
Linear2 Joint: MdtBclLinear2Joint	91
Universal Joint: MdtBclUnivJoint	91
Fixed-Path Joint: MdtBclFPJoint	92
Calling MdtBcl and MdtKea Directly	93
Example source code: KeaOnly.c	93

6 Constructing Good Simulations

Overview	104
Integrators and First Order Effects	105
Background	105
First Order Effects	106
Stiff Forces and Stability	107
The Meaning of Epsilon and Gamma	108
What Epsilon Does	108
What Gamma Does	109
Instability due to Mass and Inertia Problems	110
Mass Problems	110
Inertia Problems	110
Preventing Jitter in Contacts	111
Numerical Scaling	112
Constraint Solver Iterations	113
Modeling Using Force-Limited Motors	114
Prefer Joint Limits to Contacts	115
Avoiding Over-Determinacy	116
Hinge Joint Limits	117
Fast Spin Axis	118

MathEngine Toolkits Glossary

MathEngine Toolkits Bibliography

Overview	134
Physics Reference	134
Game Development	135

Preface

About this Manual

This manual, the *MathEngine Dynamics Toolkit Developer's Guide*, explains how to use MathEngine Dynamics Toolkit to add believable, realistic, complex physical behavior to real-time 3D environments.

The Dynamics Toolkit is a software development kit (SDK) written in C, that you can use on platforms such as Linux and Windows NT, and on game consoles such as the Sony Playstation2.

The Dynamics Toolkit includes the Kea Solver, which is its core component.

Audience for this Manual

This manual is aimed at developers of real-time simulation software, who are familiar with:

- 3D animation.
- The C programming language.
- Familiarity with Microsoft Visual C++ is an asset.

Related Documentation

This manual is accompanied by:

- The *MathEngine Dynamics Toolkit Reference Manual*, which provides detailed information about each function.
- The other documents in the *MathEngine Toolkits Documentation Set*: using a Web browser, open `index.html` in the `documentation` directory of the MathEngine Toolkits distribution.

Related Software

MathEngine Viewer

The MathEngine Viewer lets developers get visual results from MathEngine code quickly and simply. It allows developers to easily obtain statistics on the performance of their application. The Viewer is included with the Collision Toolkit, and is used in much of the sample code. It is also included with other MathEngine products.

The Viewer is documented in the *MathEngine Viewer Developer's Guide*.

MathEngine Collision Toolkit

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

- The *MathEngine Collision Toolkit Developer's Guide*.
- The *MathEngine Collision 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 creates tools that give software developers the ability to add natural behavior to applications for use in the games and entertainment, education, engineering, and e-commerce markets.

Contacting MathEngine

Head Office

MathEngine plc, Oxford Centre For Innovation, Mill Street, Oxford, UK, OX2 0JX

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

Web Site

www.mathengine.com

Technical Support

support@mathengine.com

Partnership program and customer contacts

General inquiries: Rhian Steel, sales@mathengine.com

Conventions

Units

There is no built-in system of units in the toolkits, which is not to say that quantities are dimensionless. You may choose any system of units you feel comfortable with, either meter-kilogram-seconds, centimeter-grams-seconds or foot-pound-seconds.

However, you are responsible to keep your number consistent and you will have to perform some dimensional analysis. For instance, a time step has unit of time and a damping parameter has units of mass over time. This analysis becomes important if you have to tune your application and start changing several parameters at once.

Coordinate Systems

There are 3 types of coordinate systems in Kea, all using the right-hand rule. The most important is the world coordinate system which is the common ground for all vector quantities. Each object may have two other coordinate systems namely, the one in which the geometry was modeled and another one with the origin at the center of mass. The Dynamics Toolkit contains utility functions to convert between local and world coordinate systems.

The Kea solver works almost exclusively using the world coordinates of vector quantities. The Basic Constraint Library (MdtBcl) and the simulation layer (Mdt) provide services to transform vector quantities which are naturally expressed in either the modeling frame or the body frame to the world frame. For instance, a hinge attachment between two bodies is naturally expressed in center of mass coordinates where the hinge has a fixed position with respect to each of the bodies.

Type Conventions

The MathEngine Dynamics Toolkit uses some special type definitions and macros. These make the toolkit more portable and make it easier to move between single and double precision.

For example:

- `MeReal`: floating point numbers.
- `MeVector3`: a vector of 3 `MeReals`.
- `MeVector4`: a vector of 4 `MeReals`.
- `MeMatrix3`: A 3x3 matrix of `MeReals`.
- `MeMatrix4`: A 4x4 matrix of `MeReals`.
- `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)
 - Directory and file names
 - Commands
- `Courier` indicates:
- Program code
- Italics* indicates:
- Document and book titles
 - Cross-references (hyperlinks in the online version)

Naming Conventions for C Identifiers

Me	MathEngine types and macros for controlling precision.
Mdt	MathEngine Dynamics Toolkit: Mdt Library
MdtBcl	MathEngine Dynamics Toolkit: MdtBcl Library (constraints)
MdtKea	MathEngine Dynamics Toolkit: MdtKea Library (Kea Solver)
R	MathEngine Viewer

Chapter 1 • What Can the Dynamics Toolkit Do?

Overview

You can use the MathEngine Dynamics Toolkit to add believable, realistic, complex physical behavior to real-time 3D environments, particularly computer games.

The Dynamics Toolkit lets you produce imaginative simulations quickly and easily, without immersing yourself in the complex details of the underlying mathematics and physics. You can then, if you need to, optimize the simulation to squeeze the most realism and speed out of your available hardware and software.

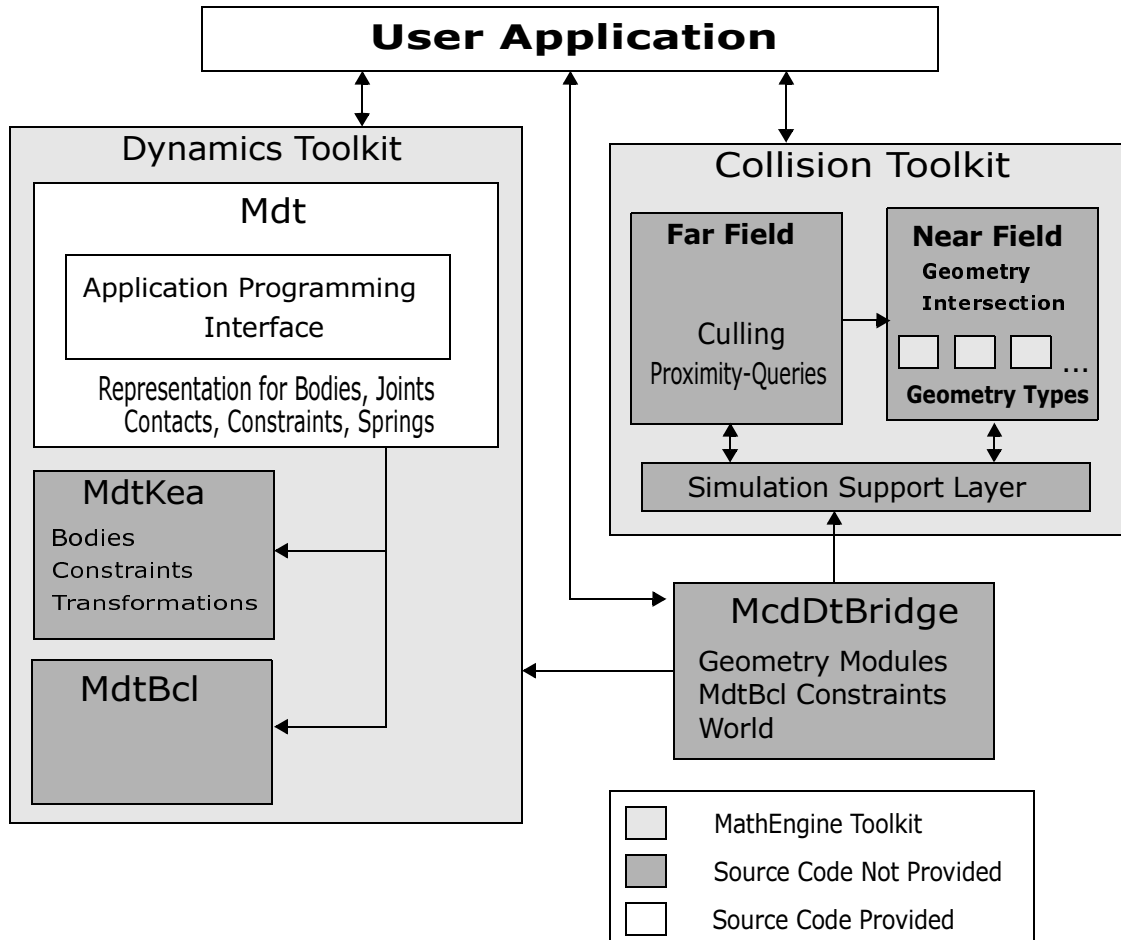
The Dynamics Toolkit is an integrated suite of dynamics simulation tools. It includes:

- Libraries and header files.
- Sample programs and demo programs, including full source code.
- Implementations of high-level scene API, including full source code.
- Documentation of proper simulation technique.

This chapter summarizes the features of the Alpha version of the MathEngine Dynamics Toolkit.

How Your Game Uses the Dynamics Toolkit

The following diagram shows how the your game or other application uses the Dynamics Toolkit in combination with the MathEngine Collision Toolkit:



MathEngine Toolkits: High-Level Architecture

Main Components

The Dynamics Toolkit includes the following main components:

- The Mdt Library, the highest-level API.
- The MdtKea Library, a general-purpose constraint solver.
- The MdtBcl Library, the *Basic Constraint Library*.

Mdt Library

The Mdt Library gives you high-level functions to simulate what would happen if, for example, a car crashed into a stack of barrels, an elephant danced on a rope bridge, or if one bipedal robot tripped another.

You can use the Mdt Library to quickly create rigid bodies and articulated bodies corresponding to the models (people, elephants, robots, vehicles, other objects) in your games or other 3D simulations. We call the abstract space occupied by these bodies a “world”, which typically corresponds to the “scene” described by your rendering software.

A world is a collection of objects such as geometries and rigid bodies, along with interaction between those objects such as forces, constraints, and finally, input and output signals. A constraint can be a *joint* or simply a *contact* which is the constraint that objects with visible geometries may not inter-penetrate each other.

Rigid Bodies: A Useful Abstraction

A *rigid body* is an abstraction from classical mechanics, a branch of physics. The idea is to simplify the mathematical model of an object by assuming absolute rigidity, even though no such object exists in the real world.

All the objects you see around you have finite extent and are therefore not well simulated using the 'point mass abstraction' of classical physics. The rigid body is an idealization of objects which do not deform easily. It represents objects which have finite extent but which 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. There is no such thing as a perfectly rigid body in the universe but the model is appropriate for several applications.

Rigid bodies have fundamental physical properties such as mass and inertia tensors as well as extra physical properties such as surface properties which are used to set friction and restitution coefficients where there are contacts and collisions. There can be other physical properties such as viscous drag coefficient (related to the geometry of the body), electric charge, magnetic dipole etc., which the user can add according to their needs.

Rigid bodies have kinematic attributes describing their position and movement namely, position of the center of mass, orientation of the body, velocity of the center of mass and angular velocity describing the rate of change of orientation. Finally, rigid bodies have dynamic attribute which consist of net applied forces and torques.

Using the Dynamics Toolkit, you set these properties to appropriate values, and make the bodies move by applying forces which can result from gravity, the impact from some object in your simulation, and so forth.

After you set these properties to initial values:

- You tell the Dynamics Toolkit how much time you want to elapse in your simulation.

- The Dynamics Toolkit calculates the updated positions and the other dynamical properties for each body in the world. This is called *solving*. The Dynamics Toolkit's *solver* is the Kea Solver.
- You update the position, orientation and so forth of the models in your scene, based on the data returned by the Dynamics Toolkit. This allows your renderer to render the scene realistically.
- Repeat until the simulation is over.

Articulated Bodies and Joint Constraints

Articulated bodies are rigid bodies connected by *joint constraints*. For example, you can connect a body representing a forearm to a body representing an upper arm by a hinge joint: the hinge joint represents the elbow.

But joints don't just connect bodies. They also *constrain* the motion of rigid bodies: you cannot pull the forearm very far without moving the upper arm; you cannot extend a human elbow more than 180 degrees.

The Mdt library includes high-level representations of joints such as hinges, ball-and-socket joints, universal joints, and others. Some of these joints have *limits* that let you create simple models of real-world behavior: you can limit a hinge, for example, so that it only opens 270 degrees.

Some joints have *soft limits*, which allow you to obtain springy effects.

Other joints are motorized, which allow you to control the movement of the bodies. Motorized joints with power limits allow very stable modeling of things like engines, brakes, motors, and stiff springs.

Collision Detection and Contact Constraints

In a 3D simulation, your animated models are likely to come in contact with each other. The point or points where two bodies intersect is *contact*. A contact is, like a joint, a kind of constraint. The Mdt library includes high-level representations of *contact constraints*.

Using the MathEngine Collision Toolkit, or using your own collision code, you can write code to determine whether two bodies are in contact. If so, you must create and populate a contact data structure. (The Collision Toolkit does this automatically.)

The Dynamics Toolkit then uses this contact information to help you make the two bodies behave appropriately. For example, if one falling body represents a sack of flour (that is, a body with almost zero restitution), and another body represents the floor, the Dynamics toolkit will detect the collision and apply the appropriate impulse so that the sack of flour stops abruptly when hitting the floor. If instead the falling body is a tennis ball with a restitution coefficient near one, the toolkit will make the ball bounce back up after the collision. Note that restitution is a property of the two materials that collide with each other.

Working with Collision Toolkit

If you want to use MathEngine Collision Toolkit to handle your application's collision detection, you can use the Dynamics Toolkit Bridge, a library that is included with the Collision Toolkit. This Bridge handles all operations and communication between the two toolkits.

The Bridge works with the Mdt Library. If you want to use one of the lower-level libraries in the Dynamics Toolkit, you will have to write your own code to handle communication with the Collision Toolkit.

For more information on the Bridge, see *MathEngine Collision Toolkit Developer's Guide*. Many of the example programs included in the Dynamics Toolkit Distribution use the Bridge.

Designing Efficient Simulations Using Mdt Library

The Mdt Library gives you two ways of culling the list of objects whose behaviors must be calculated simultaneously in your 3D world: partitioning worlds, and disabling bodies. Use either or both to make your simulation run more quickly.

Partitioning the World

It is always more efficient to calculate the behavior of objects that are in isolation from each other. If two objects are not constrained to each other, then, they are not so tightly coupled (there may still be a force between the objects) and the calculations can be simplified.

The Mdt layer include and optional partitioning feature which analyses the constraints in the world to produce lists of rigid bodies which are decoupled from each other (we call those islands). Within each list, all the bodies are coupled through constraints. Forces are ignored during this partitioning.

Disabling Bodies

If a body is not moving, there is no need to calculate it's behavior at all, unless it is struck by another body. Accordingly, the Mdt Library *disables* it, removing it from the list of bodies it passes on to the solver. This reduces the time the solver needs to update the dynamic properties of the bodies in the world.

Designing Efficient Simulations Using Mdt Source Code

The Dynamics Toolkit is distributed with the full source code for the Mdt Library. This means that you can optimize your application by:

- Choosing which parts of the Mdt Library your application uses.
- Customizing Mdt code in any way you wish.
- Creating your own joints, using Mdt joints as a model.

- Using Mdt code as a model for how to call the lower-level MdtBcl library to construct the Jacobian matrices that must be passed to the solver.
- Using Mdt code as a model for how to call the MdtKea Library to solve the dynamics.

MdtKea Library: the Kea Solver

At the heart of the Dynamics Toolkit is the MdtKea Library: the Kea Solver. MdtKea is a special purpose semi-implicit integrator for the differential algebraic equations of rigid multibody systems. It was designed to meet the requirements of 3D interactive simulations i.e., providing speed of execution and stability

Note: You can optimize your application by calling MdtKea directly, rather than using the high-level API of the Mdt Library. But even if you do not need to call MdtKea directly, you should learn something of its characteristics. That knowledge will help you design your application to be more efficient.

This comes at a cost however and if you want accuracy, you will probably have to choose a very small time step which is inefficient. On the other hand, the typical use case for MdtKea is a scene where there are many collisions and contacts with friction in which case the notion of accuracy is ill-defined.

MdtKea is rather stable and that should make your life easier when choosing parameters for your system. Note however that stability is always relative and if you are choosing very large integration steps, large mass ratios, stiff springs etc., (especially if you work in single precision), there will be stability problems.

Designing Efficient Simulations Using MdtKea

The Mdt Library is a high-level library. It works with high-level abstractions (objects) such as bodies, joints and contacts, and has the appropriate accessor and mutator functions to get and set the properties of these objects. It also has functions that let you partition your world and disable inactive bodies.

In contrast, the MdtKea Library has only one exposed function, that takes the following parameters:

- A *Jacobian matrix* that describes the constraints to be solved.
- A list of bodies. The structure for a body used by MdtKea is lower-level than the structure used by Mdt. You must write your own code to get and set the properties of each MdtKea body.
- A list of transformation matrices corresponding to the bodies. You must write your own code to modify the values of these matrices.
- A miscellaneous parameter structure, containing other parameters such as the size of the timestep.

Writing applications that call the MdtKea Library directly is much more complicated than calling the Mdt library. But calling MdtKea directly does enable you to optimize your application as much as possible.

To help you work with MdtKea, the Dynamics Toolkit provides:

- Full source code for the Mdt Library. See *Mdt Library* on page 4.
- The MdtBcl Library, a constraint library that constructs Jacobian matrices out of Mdt contact constraints and joint constraints. See *MdtBcl Library: Constraint Library* on page 10.
- Documentation and sample programs that show you how to construct your own joint constraints and how to use them with MdtBcl. See *Chapter 5: Customizing and Optimizing*.

MdtBcl Library: Constraint Library

The MdtBcl Library contains data structures and functions related to constraints. It supports contact constraints and all the joint constraints available in the Mdt Library.

If you wish to optimize your application by calling the MdtKea Library directly, you can use the functions in the MdtBcl Library to help you compute the rows of a Jacobian matrix, one constraint at a time. This matrix is one of the parameters to the Kea step function (see *MdtKea Library: the Kea Solver* on page 8).

Otherwise, you do not need to use the MdtBcl Library directly.

Chapter 2 • Installing and Building

Overview

The MathEngine Toolkits alpha release contains the Dynamics Toolkit 2.0 and the Collision Toolkit 1.0. The toolkits are available for Windows, Linux, PlayStation2, and Irix; they include libraries, headers, documentation, demo executables, and example source code.

The release is delivered in the following formats:

File	Description
metoolkit0_0_x_win32.zip	A zip file for Windows
metoolkit0_0_x_linux.tgz	A tar archive for Linux
metoolkit0_0_x_ps2.tgz	A tar archive for PlayStation2
metoolkit0_0_x_irix.tgz	A tar archive for Irix 6.5

To install the MathEngine Toolkits, unzip/untar the archive to the desired location.

Downloading MathEngine Toolkits

You can download MathEngine Toolkits directly from the MathEngine Developer's Website if you have been assigned an evaluation license. For more information on acquiring an evaluation license see <http://www.mathengine.com> or email sales@mathengine.com.

Prerequisites

Target Platforms

Supported runtime platforms are Windows98, Windows 2000, Linux and PlayStation2. However we also provide an IRIX 6.5 distribution.

Development Platforms

Windows98, Windows2000, Linux and PlayStation2 are supported as development platforms.

A Makefile mechanism (with autoconf) is provided for UNIX and Windows. Visual C++ 6.0 workspace and project files are provided for Windows.

Installation Directory Structure

After installation, the directory structure on your disk will look like this:

\---metoolkit	Top-level of distribution.
CHANGES_MCD	Changes: Collision Toolkit.
CHANGES_MDT	Changes: Dynamics Toolkit.
LICENSE	Read this before installing.
README	Read this before installing.
+---3rdParty	Third-party software and acknowledgements.
+---bin	Executable files.
+---examples	Example programs and READMEs
+---GreaseMonkey	GreaseMonkey Demo: README, executable, scripts, etc.
+---doc	Documentation Set
+---include	Header files (all toolkits).
+---lib	Libraries (all toolkits).
\---src	Source files.
makefile	Top-level makefile.
MeToolkits.dsw	Top-level workspace for Visual C++.
+---components	Component Toolkits.
+---Mdt	Dynamics Toolkit.
\---src	
+---MeGlobals	Definitions and Tools.
\---src	
+---MeMemory	Memory Manager API.
\---src	
+---MeMessage	Message Handler.
\---src	
\---MeViewer	Viewer Toolkit.
\---src	
\---fonts	Fonts used by Viewer Toolkit
+---examples	Example programs: .c, .dsp, README, makefile
\---tutorials	Tutorial programs: .c, .dsp, README, makefile

Building the Toolkit Examples

Using Visual C++

To build all the source code using Visual C++:

1. Open Toolkit\src\MeTookits.dsw
2. Click **Build -> Batch Build**.
3. In the Batch Build dialog box, make sure that all components are selected.
4. Click **Rebuild All** in the dialog box.

Using make

All the source code of this release, including the examples and tutorials, can be built in any UNIX-like environment by using the standard GNU-style build procedure, which is to run the configure script (generated using GNU autoconf), which generates an appropriate makefile, and then execute the makefile using make.

Note: GNU make v3.77 or v3.78 is required .

Before building, export the PLATFORM environment variable and set it to the target platform for which you want to build. The types currently recognised are ps2, win32, linux and irix. Run the following commands to set the PLATFORM environment variable:

```
export PLATFORM
PLATFORM=win32
```

Here is the sequence of commands to do a top-level build of all source code in the src directory, given that the PLATFORM variable has already been set:

```
cd src
make
```

Similarly you can build only the examples or tutorials or a single component by going to that subdirectory and executing the following commands (which are for tutorials):

```
cd src/tutorials
sh configure
make release
```

The resulting example and tutorial executables are put in examples/bin and tutorials/bin respectively. The component libraries are created in components/lib.

The makefiles support the following build rules:

- `release` builds libraries and executables with no symbols and no error/debug output.

- `debug` builds debug-able libraries and executables. Both symbols and error/debug output are enabled.
- `check_release` builds a release version (no symbols) but with error/debug output enabled.

Using make: Platform-dependent notes

Win32

You can download the full suite of GNU tools for Win32 from:

<http://sourceware.Cygnus.com/cygwin/>

This includes all the relevant tools. We recommend building from the GNU shell prompt.

Linux

Most recent Linux distributions (for example RedHat version 6 or later) come with all that you need. To build and run the examples, you need an OpenGL compatible implementation, for example any recent version (3 or later) of Mesa. A 3D-accelerated driver and X windows implementation, for example XFree86 4, is recommended.

Sony PlayStation2: Building from Linux

The Sony PlayStation2 toolchain version 1.60 is required. Recent Linux distributions have the other required tools.

Sony PlayStation2: Building from Win32, using SN Systems

The Sony PlayStation2 toolchain, recompiled for Win32 use, is included with the SN System tools. You also need to download and install the GNU tools for Win32 as indicated in *Win32* on page 16. The MathEngine Toolkits' build system uses the default Sony PlayStation2 GNU linker; this produces binaries that are entirely compatible with those produced by the SNSys linker.

IRIX

You should have IRIX version 6.5 or later.

You need to download and install the GNU make program, version 3.77 or later:

<http://www.gnu.org/gnulist/production/make.html>

Chapter 3 • Simulating Rigid Bodies and Forces

Overview

In this chapter, we will use code samples to show you:

- How to simulate rigid bodies using the Dynamics Toolkit's Mdt Library.
- How to manage memory for the Mdt Library and the MdtKea Library.
- How to simulate what happens when a rigid body comes in contact with something.

The Dynamics Toolkit's Mdt Library

You can use the Mdt Library to write believable, realistic simulations in a relatively few lines of code.

The Mdt Library gives you high-level functions to simulate what would happen if, for example, a car crashed into a stack of barrels, an elephant danced on a rope bridge, or if one bipedal robot tripped another—or if all these events happened in the same scene at the same time.

You can use the Mdt Library to quickly create rigid bodies and articulated bodies corresponding to the models (people, elephants, robots, vehicles, other objects) in your games or other 3D simulations.

Abstractions Implemented

The Mdt Library implements the following abstractions:

- *Rigid bodies*, and the *forces* that act on these bodies.
- *Joint constraints (joints)*. Joints attach two bodies to each other or one body to the world, creating *articulated bodies*. For more information, see *Chapter 4: Constraints: Joints and Contacts*.
- *Contact constraints (contacts)*, that describe the one or more points where two geometrical objects (that usually correspond to rigid bodies) come in contact with each other. For more information, see *Chapter 4: Constraints: Joints and Contacts*.
- The *World*, the abstract space occupied by the above abstractions. You can have one or more worlds in your application. A world typically corresponds to the *scene* rendered by your rendering software, and to the *space* occupied by the models of the MathEngine Collision Toolkit.

Note: You can implement springs and dampers by using the soft limits built into some joints. You can implement motors by using the actuation feature built into some joints.

Relationship to Other Libraries

The Mdt Library is the Dynamics Toolkit's highest-level library. If you use the Mdt Library to build your simulation, you do not need to use the lower-level MdtKea Library (Kea Solver) or MdcBcl Library (Constraints). On the other hand you can use the source code of the Mdt Library as a guide to how to use the lower-level libraries.

Here is an analogy from computer graphics: you can think of the MdtKea Library as being like an *immediate mode* interface to the Dynamics Toolkit, and the Mdt Library as being like a *retained mode* interface.

This chapter discusses only the Mdt Library. For information on the lower-level libraries, see *Chapter 5: Customizing and Optimizing*.

Embedding Dynamics Toolkit in a 3D Application

Here are the basic steps you must perform in any program using the Dynamics Toolkit Abstraction Layer. We'll show you some now, and others later on.

The code extracts below are based on `drop.c`, which is the simplest of the tutorial programs. It shows a ball falling freely through space — about as simple a simulation as you can write.

Rendering Software Drives Your Application

Your application may be driven by the requirements of your rendering software. Most MathEngine Toolkits example programs use MathEngine Viewer as a simple rendering API.

Note: For any applications using the MathEngine Viewer, use F1 to bring up application-specific Help, and F12 to bring up Viewer Help.

Stepping Through the Simulation Using Callback Functions

In the `drop.c` tutorial program, `drop.c` does not **directly** call `MdtWorldStep()` to evolve the simulation by one time-step. Instead, `drop.c` defines a callback function that is actually called by MathEngine Viewer.

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

Similarly, just before it terminates, `drop.c` frees up memory in another callback function called by Viewer:

```
void cleanup(void)
{
    ...
    ... /* cleanup code here. Includes Dynamics Toolkit cleanup code.*/
    ...
}
```

We must use a callback function to clean up, because MathEngine Viewer terminates the program itself.

Managing and Allocating Memory for Mdt Library

The Mdt Library uses the MathEngine Memory Manager API. So does `drop.c`:

```
extern struct MeMemoryAPI MeMemoryAPIMalloc;
struct MeMemoryOptions opts;
```

drop.c initializes the MathEngine Memory API, asking for the default behaviour:

```
MeMemorySetDefaults(&opts);
(*MeMemoryAPI.setOptions)(&opts);
```

The default behaviour of the MathEngine Memory API is to use `malloc()` to allocate all the memory that it needs at one time. We'll see how it can compute the size of the memory that it need in *Initializing Your World* on page 21.

Note: We'll learn more about MathEngine Memory API in *MathEngine Memory Manager API* on page 26.

Managing and Allocating Memory for MdtKea Library

The MdtKea Library uses a separate block of memory, that you must somehow allocate. Here, we use `malloc()` to allocate 10,000 bytes. Let's skip over how the we arrived at that number:

```
/* memory area for Kea solver */
void *memory;
...
memory = malloc(10000);
```

Note: Make sure that this block of memory starts on a 16-byte boundary. Most implementations of `malloc()` will allocate the memory correctly.

Initializing Your World

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

```
world = MdtWorldCreate(1, 0, memory, 10000);
```

The four parameters are:

- The maximum number of bodies in the simulation.
- The maximum number of constraints in the simulation.
- The block of memory to be used by MdtKea.
- The size of that block of memory.

`MdtWorldcreate()` uses the first two parameters to compute how much memory it will need for bodies and constraints. Then it calls the Memory Manager API to allocate that pool of memory (see *MathEngine Memory Manager API* on page 26)

Thereafter, every time we create a body or a constraint, Mdt Library takes the memory it needs out of that pool. As a result, Mdt Library does not need to allocate memory during a simulation.

Similarly, MdtKea Library uses the memory block that you pass to `MdtWorldcreate()`, and does not need to allocate memory during a simulation.

Note: For details on each function in the Dynamics Toolkit, see the *Dynamics Toolkit Reference Manual*.

Setting the Timestep for the Simulation

First we need to establish the time step, that is the amount of time that will elapse between states of the simulation.

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

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

You can set your time increment to whatever works for you, given the complexity of your world (that is, your simulation), and the platform on which you are running. For more information, see: *Stepping Through Time* on page 34.

Setting Gravity for this World

Set the gravity for this world to a familiar value:

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

Note that gravity acts along the negative y axis, but there is no need that it do so. Usually gravity will be set to act in the “down” direction, but it is possible to set it along any vector.

Defining a Body

Now we need to make a body to inhabit our world. Typically, a body corresponds to some graphical object that you are rendering: a sphere, a cube, or some more complex shape:

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

The memory for this body came out of the memory allocated by the Memory Manager. See *Initializing Your World* on page 21.

The body has been initialized with defaults set for mass (1), moment of inertia (identity matrix), and other properties.

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

```
MdtBodyEnable(body);
```

We have created a routine to initialise some important physical properties:

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

This routine is called whenever the user of the simulation presses the Enter key.

Note: A *quaternion* expresses the orientation of the body in space. You may be more familiar with *transformation matrices*. For information about many of the terms used in this manual, see *MathEngine Toolkits Glossary* on page 119.

Stepping Through Time

We are now ready to start our simulation. In `drop.c`, we do that by calling our renderer, MathEngine Viewer, passing it the name of `drop.c`'s callback function:

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

MathEngine Viewer in turn calls `Tick()` callback function (see *Stepping Through the Simulation Using Callback Functions* on page 20) in its main loop. This steps the simulation forward by the timestep set in *Setting the Timestep for the Simulation* on page 22 . Let's see how this happens.

Here is pseudo-code for MathEngine Viewer's `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 to evolve the world by the time step amount:

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

As a result, some or all of the enabled bodies may have moved to new positions, and many of the bodies' properties may have changed. (In this example, we have only one body, and it is enabled. See *Defining a Body* on page 22)

Then `Tick()` set the renderer's transformation matrix to the transform just calculated in `MdtWorldStep()`. This tells the renderer, MathEngine Viewer, where to render the body.

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

Note: You can set the time increment to a different value each time you call `MdtWorldStep()`. On a PC, for example, you may want to set the increment to however much time the computer needs to do the calculations.

Cleaning Up

When the simulation is finished, we must clean up memory. We handle this in the `cleanup()` callback function mentioned in *Stepping Through the Simulation Using Callback Functions* on page 20:

```
void cleanup(void)
{
    MdtWorldDestroy(world);
    free(memory);
    RDeleteRenderContext(rc);
}
```

OK, So We Simplified a Little...

In *Embedding Dynamics Toolkit in a 3D Application* on page 20, we glossed over a few points.

Allocating and Managing Memory for Kea Solver

You must allocate memory to store the world and its bodies for MdtKea Library. In `drop.c`, we used `malloc()`:

```
void *memory;
...
memory = malloc(10000); /* allocate enough memory for your world */
```

In this small example, we chose a comfortable size for the buffer: 10,000 bytes.

The MdtKea Library has a function that calculates its **maximum** memory requirements:

```
int MdtKeaMemoryRequired ( int * rows_in_partition, int num_partitions )
```

But in practice, this maximum is never needed.

Since MdtKea Library's memory requirements rise in proportion to the **square** of `rows_in_partition`, specifying the maximum memory requirements is wasteful.

Instead, you can use various Mdt Library functions to determine your memory requirements as you develop your application:

```
int MdtWorldGetMemoryPoolSize ( MdtWorldID w )
```

Returns the maximum amount of memory used by MdtKea Library so far in this simulation.

The library also has functions that you can use to:

- Change the memory buffer.
- Change the size of the memory buffer.
- Return the address of the memory buffer.
- Set a callback function to handle a memory buffer overflow (by, for example, changing the location and size of the buffer).
- Get a pointer to that callback pointer.

See the *MathEngine Dynamics Toolkit Reference Manual*.

Note: The MdtKea Library uses the memory buffer as temporary storage. You can use the buffer for other purposes in between calls to `MdtWorldStep()`.

MathEngine Memory Manager API

The Mdt Library uses the MathEngine Memory Manager API to store the higher-level representations (structures) of all bodies and constraints.

The Memory Manager API is a *wrapper* API that you can wrap around your own memory management software. Since the Mdt Library uses this API for its own memory management, you will achieve the following result: your application and the Mdt Library will use the same underlying memory management software. This can mean, for example, that the Mdt Library can use memory that other parts of your application have freed up.

The Memory Manager API comes with an implementation (that is, a library) that, by default, uses `malloc()` to allocate memory. You must replace that implementation with one that meets your own requirements.

`drop.c` uses the default options for the Memory manager API. Accordingly, the Mdt Library is, indirectly, using `malloc()` to allocate the memory requirement implied by your call to `MdtWorldCreate()`. See *Managing and Allocating Memory for Mdt Library* on page 20, and *Initializing Your World* on page 21.

For more information about the Memory Manager, see *MathEngine Utilities: Memory Manager Reference Manual*.

Mdt Bodies Lack Geometric Properties

You may have noticed that there are no properties that describe the body's shape (that is, its geometry). You don't need to know the shape of a rigid body to simulate its dynamics, at least for some very simple simulations. But most simulations are not that simple.

Note: You must give bodies inertia tensors that are appropriate to their geometry and mass. Failing to set inertia tensors properly is the most common cause of unrealistic behavior. For an example of bad behavior caused by this error, create a 100x100x100 stone block with a mass of 1000000. Let it keep the default moment of inertia (which is for a sphere with radius of 1 and a mass of 1). Then try to rest it on a plane.

In `drop.c`, MathEngine Viewer renders the ball as a sphere, though this is not necessary and does not affect the simulation.

A body does need to have a specific geometric shape assigned to it for you to determine whether it has collided or come in contact with another body. But you don't need to know its shape to, for example, simulate a bowl of petunias falling freely under gravity, or a whale being shot from a cannon.

Both the MathEngine Collision Toolkit and the MathEngine 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 the Collision Toolkit, the Viewer – or any third-party software that you use.

One way that the Mdt Library interacts with the geometry of a model is through contacts. See *Contacts* on page 28.

Contacts

For simple programs (such as ones showing sphere-sphere, sphere-plane, or box-plane collisions), you can handle collision detection with a few lines of code. But for anything more complex, you may want to use MathEngine Collision Toolkit.

Either way, once you have detected a collision, the Dynamics toolkits uses contact constraints to determine how to respond to a collision. You can create and enable contact structures (MdtContact) using functions in the Mdt Library. If you are using the Collision Toolkit, it will do both for you. For more information about Collision Toolkit, see the *Collision Toolkit Developer's Guide*. There, you will find explanations of how the Collision Toolkit handles MdtContact structures.

Below, we will show you how bounce.c creates and enables constraints. For more information about contacts, see *Chapter 4: Constraints: Joints and Contacts*.

Creating and Enabling

The bounce.c tutorial program shows you how to create and enable contacts. Here is the most important code:

```
/* dynamics body */
MdtBodyID body;
/* dynamics contact */
MdtContactID contact;

int numContacts = 0;
...
if (penetration > 0)
{
    MdtContactParamsID params;
    /*
     * It has so we need to create a contact.
     * The second body is zero, which means that the ball is in
     * contact with the static environment.
     */
    contact = MdtContactCreate(world, body, 0);

    /*
     * Set the contact point, which is in world coordinates.
     */
    MdtContactSetPosition(contact, pos[0], pos[1] - 2, pos[2]);

    MdtContactSetNormal(contact, 0, 1, 0);
    MdtContactSetPenetration(contact, penetration);

    /*
```

```

    Get material parameters for contact.
    */
    params = MdtContactGetParams(contact);

    /*
    2D friction means that the ball will roll on the plane as one
    would expect. If it was a zero friction contact then the ball
    would slide. 1D friction would mean that the ball rolls in one
    direction and slides in the other (perpendicular) direction.
    */
    MdtContactParamsSetType(params, MdtContactTypeFriction2D);

    MdtContactParamsSetFriction(params, (MeReal)(5.0));

    /*
    A value of 0.6 means that the contact is quite bouncy.
    */
    MdtContactParamsSetRestitution(params, (MeReal)(0.6));

    /*
    Tell Kea to process this contact.
    */
    MdtContactEnable(contact);
}
}

```

Setting Forces and Other Properties

If you apply the force of gravity, you apply it to the entire world:

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

For more information about world properties (the `MdtWorld` structure), see *The MdtWorld Structure* on page 35.

To apply force, torque, linear velocity, angular velocity, torque, and so forth to a particular body, use the appropriate `MdtBody*( )` function. Here, for example, is how `bounce.c` shoots a ball:

```
void shoot(void)
{
    int i;
    MeReal v[3];
    MeReal norm;

    /*
     Place the ball at the 'eye' of the camera.
    */
    MdtBodySetPosition(body,
        rc->m_cameraPos[0], rc->m_cameraPos[1], rc->m_cameraPos[2]);

    for (i = 0; i < 3; i++)
        v[i] = rc->m_cameraLookAt[i] - rc->m_cameraPos[i];

    norm = MeRecipSqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);

    v[0] *= norm;
    v[1] *= norm;
    v[2] *= norm;

    /*
     scale up velocity to make the shot more powerful
    */
    for (i = 0; i < 3; i++)
        v[i] *= 20;

    /* fire the ball */
    MdtBodySetLinearVelocity(body, v[0], v[1], v[2]);

    /* but don't give it any spin */
    MdtBodySetAngularVelocity(body, 0, 0, 0);
}
```

Adding Forces

Each body has a force accumulator. This is used during `MdtworldStep()` to work out the new velocity and position of bodies, and is then reset to zero. Adding forces to a body simply adds to the accumulator, and so costs no more CPU time when simulating the body.

Note: To apply a constant force to body, you must add it on each timestep.

Modeling Friction

The Dynamics Toolkit supports three main friction modes:

```
typedef enum
{
    MdtContactTypeFrictionZero,    /**< frictionless contact */
    MdtContactTypeFriction1D,      /**< friction only along primary direction
                                   */
    MdtContactTypeFriction2D,      /**< friction in both directions */
    MdtContactTypeUnknown = -1     /**< invalid contact type */
}
```

If you use `MdtContactTypeFrictionZero`, you do not need to set the direction or the coefficient of friction.

If you use `MdtContactTypeFriction2D`, then friction acts in orthogonal directions on the plane of contact, and you have full control over the settings for each direction. You do not need to set the direction of a 2D contact - it will just set it automatically.

To apply the friction at a contact point:

```
MdtContactSetType (contact, MdtContactTypeFriction2D);
```

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

```
MdtContactSetDirection (contact, 0, 0, 1); /* Primary friction acts
                                             along z-axis. */
```

The secondary direction is automatically set according to the right-hand rule.

For more information on friction, see *Simulating Friction* on page 79.

Slippiness and Slidiness

Like friction, slippiness and slidiness act in the plane of contact.

If you set 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 you apply force to a “slippy” object, the object reaches a proportionate velocity immediately; it does not accelerate to that velocity.

Stepping Through Time

To evolve your world through an increment of time use `MdtWorldStep()`:

```
MdtWorldStep (world, timestep);
```

You must choose your timestep carefully, testing with different values.

If your timestep is too large, your simulation may become obviously unrealistic: for example, a dropping ball can fall through the floor if its position is only checked after it has completely passed through the floor, that is, with no contact.

If your timestep is too small, your simulation may be unsatisfyingly slow. However, the smaller the timestep, the more accurate your simulation will be. For many purposes, of course, `realsim` is more important than physical accuracy.

The MdtWorld Structure

The Mdt Library uses *worlds* as a framework for your simulations. Your simulation can use one or many worlds. Each of those worlds may use one or more solvers, each solver having its own integrator and collision algorithms.

NOTE: In this release of the Dynamics Toolkit, only one kind of solver is supported: the MdtKea Library (Kea Solver).

The MdtWorld structure contains the properties for a world. Many of these properties are used by the Mdt to determine whether a body is *enabled* (and therefore of concern to the solver) or *disabled* (and therefore not required by the solver).

Functions that Manage Your World

First, you need a world to manage. A world is a MdtWorld structure that contains all the properties of the world you want to create. To create a world you use the MdtWorldCreate function that returns a MdtWorldID identifier that points to a MdtWorld structure that identify the world you have just created.

```
MdtWorldID MdtWorldCreate(const int maxBodies, const int maxConstraints,
                          void *const pointerToMemory, const int memorySize)
```

Where maxBodies and maxConstraints represent the maximum number of bodies and constraints you want your world to handle respectively. The pointerToMemory argument is a memory allocation pointer obtained by a malloc(memorySize) function call.

Many functions are used to manage world properties. Most (but not all) are prefixed by MdtWorld. Here are a some of them:

Set the gravity for the world:

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

where gx, gy and gz are the components of the gravity vector. In our world where the plane on which objects rest (e.g., a floor) could be described by the x and y axis, the gravity would be set only along the negative z axis to earth's gravity value, such that $gx=gy=0$ and $gz=-9.8$.

Set the acceleration threshold below which a body will be considered "at rest" and therefore disabled:

```
void MdtWorldSetDEMAccelerationThreshold(MdtWorldID world, MeReal accTresh)
```

where accTresh is the value of the acceleration treshold. Most functions that have a Set identifier in their prefixes are paired with an equivalent Get function that returns the value to which a world property has been set.

Here are a few pair of functions that are used to set the *mode of operation* of Mdt, see *Mode of Operation* on page 37.

```
MeReal MdtWorldGetDEMAccelerationThreshold(MdtWorldID world)
void MdtWorldSetDEMAccelerationThreshold(MdtWorldID world, MeReal ft)
MeReal MdtWorldGetDEMAngularVelocityThreshold(MdtWorldID world)
void MdtWorldSetDEMAngularVelocityThreshold(MdtWorldID world, MeReal avt)
```

Once the world's properties have been set, you can step the simulation forward. You have to specify the time interval `stepSize` that represents the elapsed time between two consequent computations.

```
void MdtWorldStep(MdtWorldID world, MeReal stepSize)
```

For more information about setting `stepSize`, see *Stepping Through Time* on page 34.

Mode of Operation

The Mdt Library makes available two techniques that can help you optimize your simulation: partitioning worlds, and enabling/disabling bodies.

Partitioning the World

The Mdt Library can *partition* the scene into groups of objects that are in contact, either by a joint constraint or a contact constraint. Then each group can be simulated in isolation from the other groups. This allows for faster simulations as it reduces the complexity of the dynamical problem to be solved.

For example, imagine a simulation of figure skaters doing spins, extending and retracting their arms (in an interesting demonstration of the conservation of angular momentum). Each skater is a complex articulated body, made up of several bodies and joints.

If all skaters are spinning freely, then Mdt can partition the world so that each skater is one partition. But if two skaters' arms come in contact, then Mdt will determine that those two skaters are one (larger) partition.

Enabling/Disabling Bodies

Additionally, each frame objects will be checked for movement. If an object is found to be motionless, is not interacting with other objects and has no movement-inducing forces acting on it, it will be *disabled* (turned off), saving CPU cycles.

In the MdtWorld structure, `vel_thresh`, `velrot_thresh`, `acc_thresh`, `accrot_thresh` and `alive_window` are threshold values used for determining when bodies are at rest, and therefore should be disabled. A disabled body is not processed by the Kea Solver till re-enabled by a force or collision event. A body is disabled if it falls below all of the world threshold values.

Conversly, to determine if an object is awake or *enabled*, Mdt checks whether the force and torque values exerted on that object are large than the `force_thresh` and `torque_thresh` threshold values for waking up resting bodies.

There are Get/Set functions to manage these properties. See *Functions that Manage Your World on page 35*. These functions have the `MdtWorldDEM` prefix and are used to set the values for the thresholds. The rule for partitioning and for disabling/enabling objects is called the mode of operation and is set by the following function:

```
void MdtWorldSetDEMMode (MdtWorldID world, MdtDEMMode mode);
```

where `mode` is the requested mode of operation for `world`. There are 3 valid modes of operation for a world:

Mode	Description
MdtDEMModePartitionAutoDisable	Will partition objects and will automatically disable them if they fall below their treshold values.
MdtDEMModePartitionNoAutoDisable	Will partition objects but will not automatically disable them if they fall below their treshold values. This is the default mode.
MdtDEMModeNoPartition	No partitioning is done and objects are always enabled.

Chapter 4 • Constraints: Joints and Contacts

Overview

In this chapter we learn how to create and manage sets of objects that may interact by using the Mdt Library, the high-level API of the Dynamics Toolkit. By dividing these sets of objects into *worlds* and partitioning those worlds into sub-sets of objects that interact only with other objects in the sub-set, simulations that are efficient and fast may be created.

Typically, two bodies interact if they are connected by a joint, or if they are in contact with each other.

Two or more bodies that are connected by joints collectively make up an articulated body..

Types of Joints

Articulated bodies are made up of two or more rigid bodies connected together by joints. Joints, like contacts, are constraints upon the behaviour of bodies. This section briefly describes the joint types supported by the Mdt Library. This section also describes the limits (stops) and actuation (motors) which can be applied to Hinge and Prismatic joints.

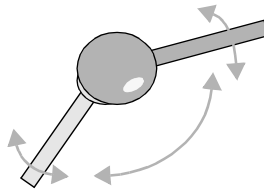
These joints can be used to attach two simulated objects to each other— that is, their relative position or orientation (or both) is constrained. Although the discussion below refers always to *attached bodies*, you may also use these joints to attach a single object to the inertial reference frame (or *world*) simply by not specifying a second attached body. You can create complex dynamic structures by constructing by "chaining" together bodies with these joints. When such structures are cross-linked (multiply connected), you must make sure that the model is physically realistic—the Mdt cannot resolve the impossible!

Degrees of Freedom

A free body has six degrees of freedom, which may be thought of as the freedom to move along each of the x-, y- and z-axes and also the freedom to rotate about these axes. A pair of objects likewise has six degrees of freedom of relative motion: even if the objects were completely fixed with respect to each other, the pair (effectively now a single object) would still have six degrees of freedom with respect to the world.

A joint may therefore remove up to six degrees of freedom from the attached pair of objects - it must remove at least one degree of freedom in order to have any effect. The number of degrees of freedom removed by a joint is thus a measure of the computational cost of using that joint in a simulation. The discussion of each joint below describes the degrees of freedom removed by each joint.

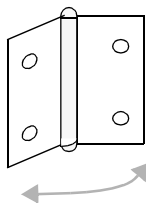
Ball-and-socket (BS):MdtBSJoint



A ball and socket joint forces a point fixed in a body to be at the same location as that of a point fixed on another body, removing three degrees of freedom. In the diagram shown above, the center of the sphere on body one always coincide with the center of the socket attached on body 2. This ideal joint allows all rotations about the common point. Real ball and socket joints have joint limits as the body on which the sphere is attached collides with the sides of the socket. By default, the MdtBSJoint does not have limits. This joint is also know as a spherical joint in the scientific literature.

A ball and socket joint could be used to attach a pendulum to a clock, a rope to a wall or beam, or a spherical wheel to a chassis.

Hinge:MdtHinge



A hinge leaves a pair of bodies free to rotate about a single axis (the hinge axis) but otherwise completely fixed with respect to each other. The axis has fixed positions and orientation in each of the rigid body and the hinge constraint forces those axes to coincide at all times. In doing so, it removes five degrees of freedom, and is therefore more computationally costly than, for example, a ball and socket joint.

Hinge are also known as revolute joints in the scientific literature.

A hinge joint could be used to attach a gate (or door) to a gatepost, or to join the covers and pages of a book. Less obviously, perhaps, it could also be used to attach a lever, drawbridge or seesaw to its fulcrum, a propeller shaft to a ship (or its engine) or a turntable to a deck.

Hinge Limits

You can set up to two stops, or limits, to the relative rotation of the bodies attached by a hinge joint. You may specify these limits independently to be either *hard* (if the limit stiffness factor is high) or *soft*. In a Mathengine Dynamics Toolkit simulation, a hard bounce reverses the bodies' angular velocities in a single timestep, while a soft bounce may take many timesteps to complete.

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

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

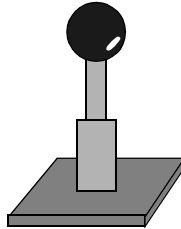
Hinge Actuators

You can also actuate, or power, a hinge joint. This simulates a motor acting on the hinge's remaining degree of freedom, the hinge angle. To characterize a hinge motor, you 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 you have set one).

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

Whenever a hinge is actuated, or is at (or beyond) one of its limits, there is a computational cost equivalent to removing the one remaining degree of freedom.

Prismatic (piston-like behavior): MdtPrismatic



The prismatic joint is much like the hinge where two axes, one fixed in each of the two bodies constrained together, are forced to coincide. However, the bodies are allowed to move along the axis, not to rotate. Like a hinge, a prismatic joint removes five degrees of freedom from the relative motion of the attached bodies, leaving only one.

You can use of a prismatic joint as a telescopic arm, a piston or an elevator.

Prismatic Limits

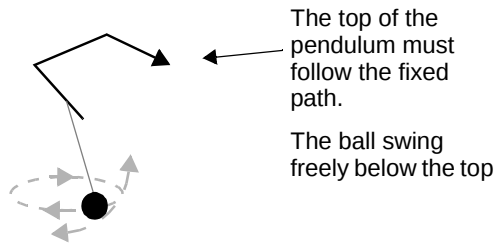
Two limits may be set on the linear motion of a prismatic joint, similar to the angular limits on a hinge. These limits may likewise be either hard or soft: the same stiffness, restitution and damping properties may be set independently for each limit.

Prismatic Actuators

You can set the speed and maximum force that actuates a prismatic joint's movement. During execution, 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 removing the one remaining degree of freedom.

Fixed-Path: MdtFixedPath



Unlike the Hinge and Ball-and-Socket joints, the Fixed-Path and Fixed-Path-with-Fixed-Orientation joint types are not intended primarily to join two simulated bodies. We expect that you will use these joints to attach animated objects to a simulated world, 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 join a rigid, simulated object to the hand of an animated character who is holding the simulated object loosely.

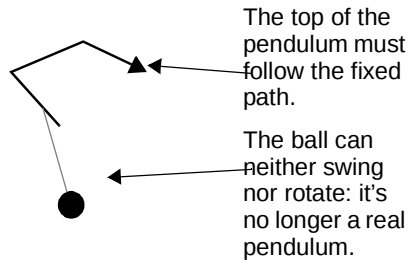
NOTE: If a firm grasp (i.e., specified relative orientation) is required, use a Fixed-Path-Fixed-Orientation joint.

The animation must supply the position of the fixed path joint (the hand's location) at each timestep: the joint can feed back the forces and torques resulting from the pull of gravity and any contacts with other simulated bodies. If the reaction torque exceeds a developer-determined limit, the joint may simply be broken and the simulated object, now detached, will fall freely.

A Fixed Path joint fixes the relative position of the two attached bodies, removing three degrees of freedom, while leaving them free to rotate with respect to each other about any axis.

It is of course possible to set the position of any joint, and you may reset these positions in every timestep of the simulation, if you choose. In this respect, a Fixed Path joint is very similar to a Ball and Socket joint. However, they differ in that you may set a joint velocity for a Fixed Path joint. While not usually necessary, this feature may improve stability in a rapidly changing simulation.

Fixed-Path-Fixed-Orientation: MdtFPFOJoint

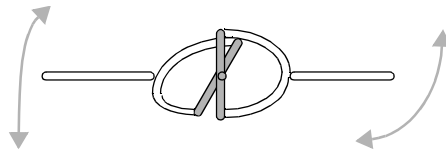


NOTE: This joint is a special case of the Fixed-Path joint. See [Fixed-Path: MdtFixedPath](#) on page 46

A fixed path, fixed orientation joint may be used as described above to attach an animation to a simulated object. It may also be used rigidly to join two simulated objects so that dynamically they become a single body. This may be desirable when, for example, complicated rigid structures are built from constituents which are individually geometrically simple - attaching the legs to a stool or chair, for example.

Bodies joined by a fixed path, fixed orientation joint have no freedom to move with respect to each other - in other words, the joint removes all six degrees of freedom from the attached bodies.

Universal: MdtUniversal

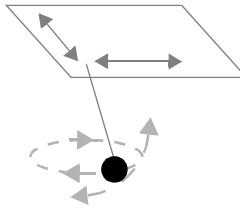


In this joint, two axes, one fixed in each of the two constrained bodies, are forced to have common origin and to be perpendicular at all times. This is a lot like the ball and socket joint but here, the ball is not allowed to twist in the socket.

A universal joint removes four degrees of freedom from the attached bodies: it fixes their relative position and also fixes their relative orientation with respect to a single axis. Any torque applied to either body about this joint axis will rotate both attached bodies equally. This joint may therefore be pictured as two hinges joined back-to-back, with both hinge axes perpendicular to the universal joint axis.

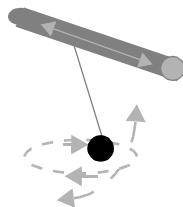
The function of a universal joint is to transmit torque from one body to another, usually changing the torque vector in the process. Universal joints might be used in this way in vehicle or engineering simulations, for example.

Linear1: MdtLinear1



A Linear 1 joint removes one degree of freedom by confining one of the attached bodies to a plane fixed in the reference plane of the other body. This could be useful as a cheap method of simulating a continuous sliding contact between an object and a planar surface which requires no contact detection.

Linear2: MdtLinear2



A Linear 2 joint removes two degrees of freedom by confining one of the attached bodies to a straight line fixed in the reference plane of the other body. As with the Linear 1 joint, this could be useful as a cheap method of simulating a continuous sliding contact between an object and a linear surface, such as a rail or pole.

Constraints

The `MdtConstraint*` functions include two important functions, one to *enable* a constraint and another to *disable* one. The difference between the *destroy* function provided for the Joint and the Contact library and the *disable* function of the Constraint library is that the *disable* keeps the constraint structure in memory for later use and that the *destroy* does not. .

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

Enables simulation of the constraint constraint.

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

Disables the constraint constraint from simulation in a world

If a constraint was created without one or both of its bodies, that is with null bodies, you can attach bodies to the constraint later using:

```
void MdtConstraintSetBodies (const MdtConstraintID constraint, const  
                             MdtBodyID body1, const MdtBodyID body2);
```

Sets the bodies `body1` and `body2` to be attached to the constraint constraint.

Be careful, a constraint must be disabled before changing its bodies. The Constraints library also provides a useful set of Set/Get functions to mutate and access the variables of a constraint structure. Consult the Dynamics Reference Manuel.

Functions To Manage Contacts and Joints

Physically, a constraint is a relation between several objects such that the total number of degrees of freedom has been reduced by at least one. The force exerted by a table on an object sitting upon it is an example of a type of constraint called *contact*. The electric cord between a suspended lamp and the ceiling is an example of another type of constraint called *joint*.

The Dynamics Toolkit provides an API called the Mdt Library for these two types of constraint. The structures for all constraints are declared in `MdtTypes.h`.

Contacts and joints each have a dedicated set of function to manage their properties. In order to describe these functions while avoiding redundancy, we will discuss an abstract, generic set of function common to all joint structures. The specific type of joint will be identified with the wildcard character `*`, which replaces one of the following identifier:

- `BSJoint`, the Ball and Socket joint
- `Hinge`, the Hinge Joint
- `Prismatic`, the Prismatic joint
- `FixedPath`, the Fixed-Path joint
- `FPF0Joint`, the Fixed-Path-Fixed-Orientation joint
- `Universal`, for the Universal joint
- `Linear1`, the Linear1 joint
- `Linear2`, the Linear2 joint
- `Contact`, a contact

Functions Common to Contact and to All Joints

You can only create a constraint through the appropriate functions for that constraint, using `Mdt*Create()`. To create an articulated body, you must create a joint. Let's first create a `Mdt*ID` variable called `joint_or_contact` that will point to the `Mdt*` structure where all information about that joint will be stored. Here is the function that creates a joint and returns a `Mdt*ID` variable:

```
Mdt*ID Mdt*Create(const MdtWorldID world, const MdtBodyID body1,
                  const MdtBodyID body2);
```

This function creates a new joint or contact in the world `world` with two bodies attached to it, `body1` and `body2`. If a body is not specified, a `null` body will be assigned instead.

There are a set of functions that apply to all constraints, the `MdtConstraint` functions. Once a constraint is created, you can access these functions by converting the contact or joint ID to a `MdtConstraintID` variable using the following function:

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

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

The `Reset` function is common to all joints and to contact, but the default values it resets to are specific to each of them. The following table lists the default values the `Mdt*Reset` function resets to:

```
void Mdt*Reset (const MdtBSJointID joint_or_contact)
```

Sets `joint_or_contact` to its default values. Note that the bodies attached to the `joint_or_contact` will have their parameters also reset. A default ball and socket joint has the following characteristics:

Type of Constraint	Default Values
Contact	•position = {0,0,0} •normal = {0,1,0} •friction direction = {0,1,0} •penetration = 0
BSJoint	•Position = {0,0,0} (in both bodies' reference frame) •Primary axis = {1,0,0} (in both bodies' reference frame) •Orthogonal axis = {0,1,0} (in both bodies' reference frame)
Hinge	•position = {0,0,0} •axis = {0,1,0}
Prismatic	•sliding axis = {0,1,0} (in first body's reference frame) •first body's initial position = {0,0,0} (in inertial reference frame)
FixedPath	•position = {0,0,0} •velocity = {0,0,0}
FPFOJoint	•position = {0,0,0} •relative orientation = {0,0,0}
Linear1	•position = {0,0,0}
Linear2	•position = {0,0,0} •direction = {0,0,0}

Type of Constraint	Default Values
Universal	•body1 = body2 = 0 •axis 1 = {1,0,0} •axis 2 = {0,1,0}

When a joint or a contact is no longer needed, you can simply get rid of it by using the following function:

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

This function destroys the joint or contact named joint_or_contact

Two functions that are common to contact and all of the joints except Prismatic are the functions Mdt*[Set/Get]Position that access and mutate the position of the contact or the joint in *world coordinates*:

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

Sets the joint_or_contact position in world coordinates at (xPos, yPos, zPos).

:

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

The position vector of joint_or_contact is returned in posVector.

A debug function

```
void Mdt*Log (const Mdt*ID joint_or_contact);
```

Debug function that output constraint using *MeMessage*

Functions that are specific to Contacts

A handful of function are specific to contact constraints. This section list all those functions and comment them:

Accessors

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

Returns the contact normal of contact in normalVec, in the world reference frame.

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

Returns the current penetration depth at this contact.

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

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

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

Returns a MdtContactParamsID pointer to the contact parameters of this contact.

The following function is only used in conjunction with the Collision Toolkit.

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

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

Sets the contact bodies body1 and body2 to be attached to the contact contact .

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

Sets the contact normal of contact to (*xNorm*, *yNorm*, *zNorm*);

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

Sets the value *penetration* of the penetration depth at the contact *contact*.

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

Utility for setting all contact parameters. Allows the user to set all values in the *MdtContactParams* structure in one go.

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

Sets the pointer of contact to the next contact *nextContact*. Used by Mcd collision.

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

Sets the primary direction for this contact at (*xDir*, *yDir*, *zDir*).

This is only necessary if you wish to define surface properties to vary depending on the direction. For isotropic (the same in all direction) 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.

Functions that are specific to Ball-and Socket Joint

Accessors

Here are of the accessor functions specific to BSJoint:

```
void MdtBSJointGetPrimaryAxis (const MdtBSJointID joint,
                                MeVector3 axisVector,
                                const unsigned int bodyindex );
```

Returns the primary axis vector in *axisVector* of the body *bodyindex* (0 or 1).

```
void MdtBSJointGetOrthogonalAxis (const MdtBSJointID joint,
                                    MeVector3 axisVector,
                                    const unsigned int bodyindex );
```

Returns the orthogonal axis vector of the body given by *bodyindex* (0 or 1) in *axisVector*.

```
MdtLimitID MdtBSJointGetLimit (const MdtBSJointID joint,
                                const unsigned int LimitID);
```

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

Mutators

Here are of the mutators functions specific to BSJoint:

```
void MdtBSJointSetPrimaryAxis (const MdtBSJointID joint, const MeReal xAxis,
                                const MeReal yAxis, const MeReal zAxis,
                                const unsigned int BodyID);
```

Sets the primary axis vector (*xAxis*, *yAxis*, *zAxis*) of the body given by *BodyID* (0 or 1). The vector is set in the inertial reference frame.

```
int MdtBSJointSetOrthogonalAxis (const MdtBSJointID joint,
                                   const MeReal x0Axis, const MeReal y0Axis,
                                   const MeReal z0Axis, const unsigned int BodyID);
```

Attempts to set an axis vector (*x0Axis*, *y0Axis*, *z0Axis*) of the body given by *BodyID* (0 or 1) which is the orthogonal to the body's primary axis. The vector is set in the inertial reference frame. This service returns a non-zero value if it succeeds. If the input axis is not orthogonal to the primary axis, the service fails: it returns zero without changing the hinge in any way.

```
void MdtBSJointSetLimit (const MdtBSJointID joint, const MdtLimitID NewLimit,
                          const unsigned int LimitID);
```

Resets the joint limit and then copies the public attributes of *NewLimit*. *LimitID* values correspond to the following rotational degrees of freedom: 0 - rho (twist angle) 1 - theta (cone, or polar, angle) 2 - phi (segment, or longitudinal, angle).

Functions that are Specific to Hinge Joint

Accessors

Here are of the accessor functions specific to Hinge:

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

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

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

The Hinge joint's axis is returned in the vector *axisVec*.

Mutators

Here are the mutators functions specific to BSJoint:

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

Resets the joint limit and then copies the public attributes of *NewLimit*.

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

Set the Hinge axis of *joint* to (*xAxis*, *yAxis*, *zAxis*)

Functions Specific to Fixed-Position-Fixed-Orientation Joint

Accessors

Here is the accessor function specific to FPF0Joint :

```
void MdtFPF0JointGetQuaternion (const MdtFPF0JointID joint, MeVector4 q1,
                                MeVector4 q2)
```

This retrieves the bodies' individual quaternions.

Note that the equivalent mutator service sets the joint's invariant relative orientation property. This accessor will therefore not necessarily retrieve identical values to those used in an immediately preceding call to the mutator

Mutators

Here is the mutator function specific to FPF0Joint :

```
void MdtFPF0JointSetQuaternion (const MdtFPF0JointID joint, const MeVector4
                                q1, const MeVector4 q2)
```

Set the invariant relative orientation vector.

Note that if the secondary body is unrotated, the input *q2* should be: *q2*[0]==1, *q2*[1]==*q2*[2]==*q2*[3]==0

In this case, this service is equivalent to: *j*->*bclFPF0Joint*.rel_ori[0] = *q1*[1]; *j*->*bclFPF0Joint*.rel_ori[1] = *q1*[2]; *j*->*bclFPF0Joint*.rel_ori[2] = *q1*[3];

Note also that if *q1* == *q2*, then *j*->*bclFPF0Joint*.rel_ori = {0, 0, 0}

Functions Specific to Fixed-Path Joint

Accessors

Here is the accessor function specific to FixedPath:

```
void MdtFixedPathGetVelocity (const MdtFixedPathID j, MeVector3 velocity,
const unsigned int bodyindex)
```

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

Mutators

Here is the mutator function specific to FixedPath:

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

Sets the fixed path joint's velocity with respect to one of the constrained bodies. This joint's velocity is set in the bodies' reference frames. The reference frame is selected by the fifth parameter, *bodyIndex* .

Functions that are Specific to Prismatic Joint

Accessors

Here are the accessor functions specific to Prismatic:

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

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

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

The prismatic joint's axis is returned in the vector axisVec.

Mutators

Here are the mutators functions specific to Prismatic:

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

Resets the joint limit and then copies the public attributes of *NewLimit*.

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

Set the Prismatic axis of *joint* to (*xAxis*, *yAxis*, *zAxis*)

Functions Specific to Linear2 Joint

Accessors

Here is the accessor function specific to Linear2:

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

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

Mutators

Here is the mutator function specific to Linear2:

```
void MdtLinear2SetDirection (const MdtLinear2tID contact, const MeReal xDir,
                              const MeReal yDir, const MeReal zDir);
```

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

Functions that are Specific to Universal Joint

Accessors

Here is the accessor function specific to Universal:

```
void MdtUniversalGetAxis (const MdtUniversalID joint, MeVector3 axisVec,  
                           const unsigned int bodyindex)
```

One of the joint axis corresponding to *bodyindex* is returned in the vector *axisVec*.

Mutators

Here are the mutators functions specific to Universal:

```
void MdtUniversalSetAxis (const MdtUniversalID joint, const MeReal xAxis,  
                           const MeReal yAxis, const MeReal zAxis,  
                           const unsigned int bodyindex);
```

Sets the joint axis corresponding to the fifth parameter, *bodyindex*, in world coordinates to (*xAxis*, *yAxis*, *zAxis*).

Macros

As discussed in “Constraints” on page 51, we have first to 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);
```

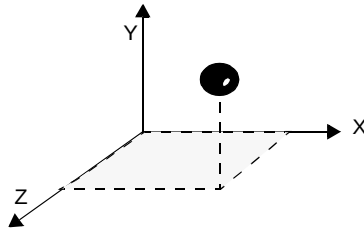
However, there exist a simpler way to accomplish the same action:

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

Those are not functions, but only macros that were implemented to save you the hassle of writing two lines instead of one.

How to Use Contacts

In the tutorial *Bounce* released with this manual, a simple usage of contacts is made. A ball is thrown onto a surface and rebound each time it touches the surface, each time with a smaller amplitude. To achieve this effect, a callback function `tick()` called by MeViewer is responsible for checking for physical intersection between the ball and the surface (since the Collision Toolkit is not used in this strictly Dynamics example), and each time an intersection is detected a contact must be created. It is important to remember that a contact is not something as permanent as a joint since it is created dynamically, whenever two object come into physical contact, and that it is destroyed as easily.



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 into contact:

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

To determine if intersection between the ball and the plane took place we need to know the position of the ball relative to the plane (which is at $y=0$) and take the radius of the ball into account.

```
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. In case of intersection, it creates a contact constraint. The first body in contact is the ball body, the second is the world, hence 0, since the plane is static and attached to the world.

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

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 just the plane's normal, which is the Y-axis. To compute the rebound amplitude, the penetration depth must also be communicated to the `Mdt`.

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

Then the contact properties must be set: the type of friction, the friction and the restitution values. We need first to obtain a `MdtContactParams` structure related to our contact to fill in.

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

Finally, we 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, Kea, the physical solver, will just ignore it.

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

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 a hinge joint. Alternatively, by running "hinge 2", two bodies are connected to each other by a hinge joint. In either case, the hinge can be motorized and you can set rotational limits.

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

First, in the `main()` function, we must create a hinge joint in the world. Once the hinge exists we set its center of mass position at the origin and its axis orientation parallel to the Y axis. By default `body[1]` is set to 0, or in other words, it is set to be the whole world.

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

We 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, we set the lower and upper limit angles of the hinge. We set the maximum force at a value `maxForce` allowed to attain a desired velocity of a value `desiredVelocity`.

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

Finally we attach the hinge joint to the world, allowing it to be computed at the next update.

```
MdtHingeEnable(hinge);
```

As you may have noticed, the main difference between contacts and joints is that joints are most often created once and only, as in *hinge* in its `main()` function, but contacts must be created and destroyed repeatedly as in *Bounce*'s callback function `tick()`.

Chapter 5 • Customizing and Optimizing

Overview

In this chapter we will learn how call the Kea Solver, and look at its data structures.

As well, we will look at the mdtBcl library, which computes the Jacobian matrices required as input to Kea.

This knowledge will allow you to write your games and simulation applications to the Kea Solver API directly, giving you more control and thereby making several efficiency gains:

- Your executables will be smaller, because you will not need to link all the Dynamics Toolkit's libraries into your application.
- You can avoid some data copying.
- You can optimize force calculations, using your inside knowledge of your own application.
- Knowing how Kea solves dynamic problems allows you to design simulations that will be more efficient to solve.

Rigid Bodies

A rigid body is an abstraction from classical mechanics. The idea is to simplify the mathematical model of an object by assuming absolute rigidity. No such object exists in the real world and any engineering application must take account of the bending and sheering of materials.

In a mathematical model, an important concept is the number of degrees of freedom (DOF). This is the number of independent variables needed to fully describe a system. A fully flexible system, for example, some models of fluid dynamics require infinite dimensional models. However, by assuming rigidity, it is possible to reduce the description of a body to a geometrical model, plus a physical model consisting of the body's position plus its orientation.

Reference Frames

This position vector indicates the offset in the world reference frame at which a body is located.

The body itself has a reference frame attached to it, with its origin and positioned at the body's center of mass. This is called the body reference frame (BRF).

The Dynamics Toolkit uses the Right-Hand Rule for its coordinate systems.

Orientation

There are many ways to describe the orientation of a rigid body. The easiest way is provide a 3x3 matrix that maps the body reference frame into the world reference frame. The easiest way to think of this matrix is as a set of 3 vectors, the look at, look up and look right vectors.

In addition, we know from Newton's laws that the motion of a body, or the change in its position and orientation with time is based on applied forces.

For our purposes, a body in space is completely described by its position and orientation, and its motion will be determined by its initial position and velocity along with any applied forces or torques. And it is these elements that are found in the Kea immediate mode body structure.

The MdtKea Functions

The Kea Solver does not maintain any internal structures, it just acts like a machine that takes a load of input and updates the bodies. MdtKea has only two functions, MdtKeaStep() and MdtKeaMemoryRequired().

The most important one being MdtKeaStep() which evolves the system by solving each partition.

```
void MdtKeaStep (Mdtstraints constraints, MdtKeaBody *blist,  
                MdtKeaTransformation *tlist,int num_bodies,  
                MdtKeaParameters parameters)
```

This function takes an array of bodies and a keaConstraints structure (which defines the constraints), as well as an array indicating which constraints are to be solved together as one partition. It solves each partition and evolves the system. Its parameters are detailed in the table below.

MdtKeaStep Parameters

Parameters	Description
constraints	A MdtKeaConstraints struct containing constraint rows generated by functions in the MdtBc1 library.
blist	Array of MdtKeaBody structs to be simulated.
tlist	Array of MdtKeaTransformation structs that correspond to blist.
num_bodies	Number of bodies/transformations in blist/tlist
parameters	A MdtKeaParameters structure containing stepsize, memory pool, and so forth.

Most of these parameters are structures requiring detailed explanations. See *Dynamics Toolkits Reference Manual*. Additional information appears below.

The other function is a utility function that is used to compute the size of the memory workspace required by MdtKea.

```
int MdtKeaMemoryRequired ( int * rows_in_partition, int num_partitions )
```

Utility function for calculating the size of the workspace needed by Kea. This memory is passed into Kea in the MdtKeaParameters structure.

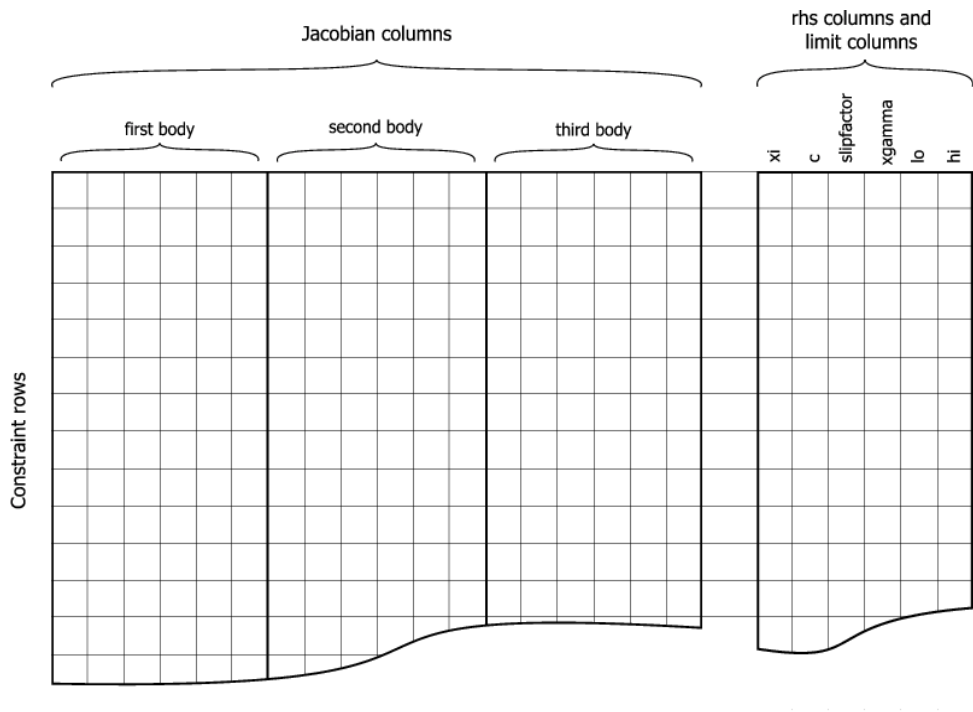
MdtBcl Structure

You can construct the values for this structure using your own code, or you can use the MdtBcl library

Member	Description
int num_partitions	Number of partitions described by this structure.
int* num_rows_partition	num_rows_partition[i] is number of rows in partition i.
int* num_constraints_partition	num_constraints_partition[i] is number of constraints in partition i.
int num_rows	Total number of constraint rows (+ 0<=i<num_partitions . num_rows_partition[i])
int num_constraints	Total number of constraints (+ 0<=i<num_partitions . num_constraints_partition[i])
MeReal* Jstore	List of nonzero blocks of jacobian matrix.
MeReal* xi	Constraint error vector.
MeReal* c	Vector in the constraint equation $m\{J*v=c\}$.
MeReal* lo	Low limit on Lagrange multiplier.
MeReal* hi	High limit on Lagrange multiplier.
MeReal* lambda	Lagrange multiplier READ ONLY.
MeReal* force	Vector of forces applied to satisfy constraints. Read-only.
MeReal* slipfactor	First order constraint slipping vector.
MeReal* xgamma	Projection constants.
int* Jsize	Jsize[i] is the number of rows in constraint i.
int* Jofs	partition_start+Jofs[i] is the offset in the above vectors of the first element corresponding to constraint i, where partition_start is the offset of the first constraint in the partition which i belongs to.
int* Jbody	Jbody[i*3+0],Jbody[i*3+1],Jbody[i*3+2] are the bodies constrained by constraint i.

NOTE: At the end of each partition, a number of empty constraint rows or d.o.f. equations are added to make the total in each partition up to a multiple of four. padding per partition. That's where that *up to three rows of padding* comes from.

The following diagram shows some details of this structure:



MdtKeaBody Structure

Contains the properties of a single rigid body.

Member	Description
len	Total length of data structure
invmass	1/Mass of the body
fastSpin	flag: nonzero applies fast rotation
force	Applied force (to centre of mass)
torque	Applied torque
invI0, invI1, invI2	Inverse of 3x3 inertia tensor
I0, I1, I2	The 3x3 inertia tensor
vel	Linear velocity of body
velrot	Angular velocity of body
qrot	Body orientation quaternion
accel	Linear Acceleration of body. Read-only.
accelrot	Angular Acceleration of body. Read-only.
fastSpinAxis	axis of assumed fast rotation (unit length).

Let's look at some of these members in more detail.

Force

Before calling the `MdtKeaStep()` function, set `force` to be the total applied forces at the center of mass.

If the application programmer wants to put a force at a point which is not the center of mass, it is possible to resolve this into a force applied to the center of mass and a torque about the center of mass.

There is a function in the Mdt Library that will calculate force at the center of mass and torque about the center of mass, given a force applied at any point on the object:

```
void MdtBodyAddForceAtPosition(MdtBodyID body,  
                                MeReal fx, MeReal fy, MeReal fz,  
                                MeReal px, MeReal py, MeReal pz );
```

Apply a force at a specific point on a body. The vector (fx,fy,fz) being the vector of the force acting on the body and the vector (px,py,pz) being the point on the body on which the force acts.

The call to `MdtKeaStep( )` will add the input value of force to the sum of constraint forces arising from joints and contacts the body is subject to. Kea returns this sum in the Force member. Set force to the correct applied forces before calling `MdtKeaStep( )` in the next time step.

Torque

Before calling the `MdtKeaStep( )` function, set torque to the total applied torque at the center of mass.

The call to `MdtKeaStep( )` will add to the input value of torque to the sum of the constraint torques arising from joints and contacts the body is subject too. Kea returns this sum in the torque member of the `MdtBodyID` body.

Set torque to the correct applied forces before calling `MdtKeaStep( )` in the next time step.

NOTE: After a time step, the contents of force and torque will be the total applied forces plus any forces and torques due to constraints. These MUST BE RE-ZEROED before accumulating values for the next time step.

I0, I1, I2

This is the inertia matrix of the body.

vel

The linear velocity of the body measured at the center of mass.

velrot

The rotational, or angular, velocity of the body measured about the center of mass.

qrot

A quaternion representing the orientation of the body in space. This value is updated by Kea. See the section on Quaternions below.

accel

This read-only value indicates the value of acceleration, measured at the bodies center of mass, after the last time step.

Accelrot

This read-only value indicates the value of rotational acceleration, measured about the bodies center of mass, after the last time step.

fastSpin / fastSpinAxis

These values are used for objects that spin rapidly, for example, the wheel on a car.

MdtKeaParameters: Kea’s parameter structure

This structure contains parameters used to control the Kea Solver.

Member	Description
MeReal stepsize	Amount of time to evolve system by.
MeReal epsilon	Numerical tolerance used by matrix solver. Default = 0.01
MeReal gamma	Constraint relaxation rate. Default = 0.2
void* memory_pool	Pointer to some memory that kea can use (kea doesnt malloc, it only uses this area).
unsigned int memory_pool_size	Size of this area in bytes.

Let’s look at these in detail.

stepsize

Amount of time, in seconds, to evolve the system.

In general, the best simulations will be created by holding the stepsize within a narrow tolerance. The Kea module will create slightly variant behaviors given different time steps, and this will be noticeable enough that game tweaking should be done at the frame rate that the user will play the game.

epsilon

The “diagonal modifier.” There are two ways of thinking about this parameter:

- The amount by which constraints may be violated
- The diagonal of system matrix (A+eI)x=b

What epsilon (ε) does is move a matrix out of singularity. For example, take the matrix

$$\begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 0.0000001 \end{bmatrix}$$

Because of the very small number in the (1, 1) slot, this matrix may be numerically difficult to solve.

Now the following matrix may be much easier to solve:

$$\begin{bmatrix} 1.0 + \varepsilon & 0.0 \\ 0.0 & 0.0000001 + \varepsilon \end{bmatrix}$$

gamma

As a simulation runs, joints will come slowly out of their appropriate alignment. gamma is a projection constant that will move the system back towards the correct state.

If gamma = 0, then objects are allowed to drift. If gamma = 1, then a perfect match to a first order approximation is created. If you want better behavior, increase gamma.

gamma = .2 is a reasonable number to use as a default. Larger values of gamma may cause difficulties.

Note that some joints allow you to override this gamma value and to set a gamma just for that particular joint.

memory_pool, memory_pool_size: keaCalculateMemoryRequired()

The Kea Solver does not allocate memory with `malloc()`. Instead, it uses this memory pool.

To calculate how large the memory pool must be, use the following:

```
int keaCalculateMemoryRequired (int* num_rows_partition, int num_partitions)
```

Contacts and Friction

A contact between two rigid bodies is a point at which the surface are very very close together i.e., a point where the molecules of each objects are so close that they interact strongly and this occurs when the distance is of the order of atomic radii. In a simulation, a contact occurs when the geometries intersect. Ideally, the intersection region should be limited to points, edges and faces but this is not really possible in practice and the objects do interpenetrate some creating intersections with finite volume.

In MdtBcl, a contact consists a point and a normal. The contact constraint will ensure that there is no relative motion in the direction opposite to the 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 our approximation of the Coulomb friction model.

Most of the time the contacts will not be actually possible contact points. This is because the dynamics rigid body solver may leave bodies in slightly overlapping positions. Therefore, what the Dynamics Toolkit 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 efficient for the RB solver, and in a way that produces the expected non-interpenetration behavior.

Interpenetration and Contact Strategies

Interpenetration of two surfaces must be prevented by carefully choosing an appropriate small set of contact objects. The choice depends on the type of "contact behavior": bouncing usually can "get away with" only one contact; resting and sliding usually require three contacts.

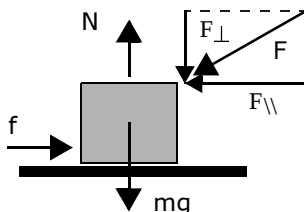
The MathEngine Collision Toolkit offers a number of alternative "contact strategies" that produce contacts for different types of situations; and you can switch from one to another as needed. Strategies that use fewer contact points have the advantage that MdtKea solves them more quickly. If you are using a third-party collision package using your own custom collision routines, you must create a contact strategy that models realistic behavior 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 the 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 prevent the bodies from sliding against each other along the contact plane.

Coulomb Friction



Friction is the resulting effect of the complicated molecular physics that takes place when two objects are very near each other. Friction is a dissipative process which transfer energy from the large scale motion to short scale motion like heat and sound.

Friction can come in two broad flavors: viscous friction and dry friction. Viscous friction is what happens when the contact is wet i.e., when there is a lubricant at the contact like oil. Dry friction is what happens 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 which has two aspects namely, kinetic friction and static friction. Kinetic friction is when the objects are sliding while static friction is the phenomenon that prevents sliding altogether when the relative motion comes to rest. Kinetic dry friction opposes the motion but with a magnitude which depends on the contact force between the objects, not their relative velocity. Static friction opposes relative motion altogether up to a threshold tangential applied force.

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 force the gravity $mg\vec{g}$ and to the perpendicular (relative to

the ground) component of the external force, $\vec{F}_\perp$, exert 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 component of the external force, $\vec{F}_\parallel$.

Let's assume 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 $f_{s\max} = \mu_s N$, the frictive force fails to scale up with $\vec{F}_\parallel$ and the force inbalance will cause the box to start sliding. Once the box is in motion the frictive force $\vec{f}$ still impedes the movement of the box but to a lesser degree. This new frictive 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: $f \leq f_{s\max} = \mu_s N$
- 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 rigidi 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 i.e., some contact problems have multiple valid answers and some other problem have no valid answer consistent with the constraints and the Coulomb model.

Out of the several approximations available, we chose one that was both stable and efficient. This comes at a cost and you may observe anisotropic (i.e., that varies with the direction) friction for instance. We will provide more options for this in the coming release.

There are three types of contact friction that can be simulated with MdtKea.

Frictionless:

```
// if the MdtContactType is MdtContactTypeFrictionZero, then ...
```

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 $f_{s\max} = f_k = 0$.

Infinite friction

```
// if (the MdtContactType is MdtContactTypeFriction1D or 2D), and
//     (the MdtFrictionModel(s) is/are not MdtFrictionModelBox)), then...
```

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 $f_{s\max} = \infty$.

Box friction

```
// if the MdtFrictionModel is MdtFrictionModelBox, then ...
```

The tangential friction forces are applied with an upper limit (friction1 and friction2). This can be done in one or both of the tangential friction directions. This is a simple approximation to the Coulomb friction—it does not allow for variation of the tangential force limit as the normal force changes and is therefore equivalent to setting $f_{s\max}$ to a constant value. However, this approximation is still useful in many situations.

Slip: An Alternate Way to Model Friction

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).

If you set 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 $f_{s\max} = f_k = 0$ and $\vec{V} \propto \vec{F}_{\parallel}$ where $\vec{V}$ and $\vec{F}_{\parallel}$ 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. If you set slip along the transverse (side to side) direction of the tire's contact point with the ground, you can get a good model of "tire slip" that will result in improved vehicle behavior. Set the slip to a value that is proportionate to the rotational velocity of the tire. If the vehicle is not moving, set the slip to zero.

MdtBclContactParams

This structure is part of the MdtBcl library. It determines the type of contact between the two objects. This information usually does not change frame to frame.

Member	Description
type	contact type (zero, 1D or 2D friction)
model1, model2	friction model to use along primary/secondary direction
options	Contact options (restitution etc.)
restitution	restitution parameter
velThreshold	Minimum velocity for restitution
softness	contact softness parameter (soft mode)
max_adhesive_force	Contact maximum adhesive force parameter (adhesive mode).
friction1, friction2	max friction force (box mode) or mu (triangle mode)
slip1, slip2	slip parameter (slippy mode)
slide1, slide2	slide parameters (slide mode)

Details on these members follow.

type

There are three valid types of contacts in Kea. If contact type is 1D friction - motion along secondary direction is frictionless, and model2, slip2, slide2 and friction2 are ignored.

Types	Description
MdtContactTypeFrictionZero	Friction contact
MdtContactTypeFriction1D	Friction only along primary directions
MdtContactTypeFriction2D	Friction in both directions

model1,model2

When the type member is set to be one- or two- dimensional friction, these values select the “friction model” along each direction. The only possible choice currently is:

MdtFrictionModelBox

The effects of which are discussed above in the discussion about box friction.

options

The following options for the friction model may be ORed together in any combination:

Options	Description
MdtBclContactOptionUseDirection	Use primary direction vector
MdtBclContactOptionBounce	Use restitution at the contact
MdtBclContactOptionSoft	Use a simple soft contact model
MdtBclContactOptionAdhesive	Use a simple adhesive contact model.
MdtBclContactOptionSlip1	Use a first order slip in the primary direction
MdtBclContactOptionSlip2	Use a first order slip in the secondary direction.
MdtBclContactOptionSlide1	Use a surface velocity in the primary direction (like a conveyor belt motion).
MdtBclContactOptionSlide2	Use a surface velocity in the secondary direction.

restitution

This is the restitution parameter, coefficient of restitution in the real world.

velThreshold

Minimum impact velocity for restitution

softness

Violates ideal constraint, allows penetration and gives a 'springy' effect as well. It can make things take longer to come to rest.

Contact softness parameter (soft mode) 0 to 1, with 1 being terrible softness and .1 or .001 being OK.

adhesive_max_force

Sticky contacts dont work very well because what tends to happen is that once the penetration of a contact goes negative (the contact has seperated) the contact is destroyed, and the 'sticky' force can't pull the objects back together again.

friction1

max friction force (box mode) in primary direction

friction2

max friction force (box mode) in secondary direction

slip1, slip2

slip parameter (slippy mode) along primary and secondary directions

slide1, slide2

slide parameter (slide mode) along primary and secondary directions

MdtBclContact Structure

This is MdtBclContact constraint strucure

Member	Description
cpos	Position of contact (world reference frame).
normal	Unit normal at contact (world reference frame).
penetration	Penetration depth of bodies at contact.
pad	Padding - not used
direction	Principal friction direction. World reference frame, unit length and perpendicular to normal
params	Friction and restitution pararmeters

MdtBclLimit Structure

This is the MdtBclLimit structure. It specifies behaviour of the constraint for a motion which is otherwise unconstrained by the joint's hard constraints. This motion may be constrained at either an upper limit or a lower limit, or both, or actuated (powered) by a force-limited motor between such limits or in the absence of limits.

Member	Description
limited	Set if the corresponding degree of freedom of the joint is limited. Cleared (set 0) if the degree of freedom is not limited.
bRelaxingToLimit	Non-zero if the joint has overshoot a limit and is relaxing back to that limit, zero if bouncing.
overshoot	The overshoot distance. This is set to zero between the limits; otherwise it is the amount by which the joint has overshoot its limit. This is positive beyond the upper limit, negative beyond the lower limit.
position	Relative position of the attached bodies, projected onto the relevant axis.
velocity	Relative linear or angular velocity of the attached bodies, projected onto the relevant axis.
damping_tresh	The limit hardness threshold. This value depends on the timestep and is recalculated in each timestep in which a limit is exceeded. When a limit hardness exceeds this value, damping is ignored and only the restitution property is used. When the limit hardness is at or below this threshold, restitution is ignored, and the hardness and damping terms are used to simulate a damped spring.
limit	The specific properties of the lower and upper limits.
powered	0 for non-powered joint, -1 for powered joint.
desired_vel	Powered joint, desired velocity. A lower limiting velocity may be achieved if the attached bodies are subject to velocity or angular velocity damping.
fmax	Powered joint, maximum force that should be applied to reach the desired velocity.

Joints

A joint is a special kind of constraint namely, an ideal equality constraint. This is a constraint that maintains an equality condition (as opposed to contact constraints for instance) defined in terms of positions and orientations only (as opposed to motor constraints which put conditions on velocities) and for which the corresponding constraint force is workless which means it neither adds nor removes energy into the system.

Essentially there is a point, a line or a plane in one body that must be held equal to one in another body. Constraints may be more general and may involve inequalities, for example, the z value of the position vector may be required to always be greater than zero.

Conceptually, joints are very similar to contacts. However there are a couple of distinguishing characteristics:

- Joints are a persistent characteristic between two bodies. Two bodies in contact may slide away from each other, or separate upon application of an impulse. Joints do not allow for this.
- Joints are equality relationships. That is some point, or curve, or axis in one body must always be equal to some corresponding point, curve or axis in a second body.

There are several types of Kea joints implemented. Standard joints like the ball-and-socket joint and the hinge are provided, but we also provide a car steering joint for sophisticated vehicle simulations.

Structures for Joints in the Mdt Library

For every joint, two joint structures are defined. Let us consider the ball and socket joint as an example.

Here is the Mdt Library structure (MdtTypes.h):

```
struct MdtBSJoint {
    MdtConstraintHeader head; /** Immediate mode data. */
    MdtBclBSJoint immBSJoint;
};
```

Structures to Manage Joints in the MdtBcl Library

Here is the MdtKea-level ball and socket joint structure:

Notice that the structure at the Mdt Library includes a copy of the MdtBcl Library's data structure (MdtBcl.h).

```
typedef struct {
    MdtBclConstraintHeader head;
    MeVector3 pos1;
```

```

    MeVector3 pos2;
    MeVector3 axis_n1;
    MeVector3 axis_m1;
    MeVector3 axis_n2;
    MeVector3 axis_m2;
    MeVector3 n1_cross_m1;
    MdtBclLimit limit[3];
}MdtBclBSJoint;

```

Constraint Header Structure

All the MdtBcl joint structures use a header structure (MdtBcl.h):

```

typedef struct {
    int tag;
    int len;
    int pad[2];
    int body[4];
}MdtBclConstraintHeader;

```

You Can Add Your Own Joint Types

One reason for exposing joints this fully is to allow you to construct your own joints, and then include those joints in the systems solved by the Kea Solver. To learn how to construct a joint, look at the source code of the KeaOnly tutorial.

Joints Supported by the Dynamics Toolkit

In the following joint descriptions, we include descriptions of some of the fields. For more information, see the *Dynamics Toolkit Reference Manual*.

Ball-and-Socket Joint: *MdtBclBSJoint*

In this joint, two objects are constrained so that a single point in each object is constantly held in contact. The two bodies may each rotate a full 360 degrees about any vector based at the common point.

This joint is entirely defined by two numbers: the position of the joint in the first body's coordinate system, `pos1`, and the position of the joint in the second body's coordinate system, `pos2`.

Member	Description
<code>pos1</code>	Joint position in the first body reference frame
<code>pos2</code>	Joint position in the second body reference frame.
<code>axis_n1</code>	Primary rotation axis 1 (fixed in 1st body ref. frame).
<code>axis_m1</code>	Primary rotation axis 2 (fixed in 2nd body ref.frame).
<code>axis_n2</code>	Secondary rotation axis, orthogonal to <code>axis_n1</code> (fixed in 1st body ref. frame).
<code>axis_m2</code>	Secondary rotation axis, orthogonal to <code>axis_m1</code> (fixed in 2nd body ref. frame).
<code>n1_cross_m2</code>	This vector is maintained (read-only) in order to ensure smooth rotation when <code>n1</code> and <code>m1</code> are nearly parallel or anti-parallel (i.e. $\theta \sim 0$ or $\theta \sim \pi$).
<code>limit[3]</code>	Optional limits set to motion about the rotation axes. These are (in array order): 0 - ρ (twist angle) 1 - θ (cone, or polar, angle) 2 - ϕ (segment, or longitudinal, angle).

Hinge Joint: *MdtBclHinge*

A hinge joint models a hinge, such as you might find on a door. A hinge allows movement about a single axis, and this movement is generally restricted within a particular range of values.

You must set the position and axis values in the coordinates of each body system. Kea will then keep these positions identified.

Member	Description
<code>pos1</code>	Position of joint (first body reference frame)
<code>pos2</code>	Position of joint (second body reference frame)

Member	Description
axis1	Hinge axis (first body reference frame)
axis2	Hinge axis (second body reference frame).
limit	Hinge axis (second body reference frame).
raxis_a	Reference axis perpendicular to raxis_b and hinge axis in first body reference frame.
raxis_b	Reference axis perpendicular to raxis_a and hinge axis in first body reference frame.
raxis_a2	Reference axis perpendicular to raxis_b and hinge axis in second body reference frame.

Prismatic Joint: MdtBclPrism

A prismatic joint is like a piston going back and forth: it has only one degree of freedom. .

Member	Description
axis1	Sliding axis (first body reference frame).
pos1	Initial position of first body (inertial reference frame).
limit	Optional limits set to motion along the sliding axis.

Car Wheel Joint: MdtBclCarWheel

A hinge that can be steered along a steering axis, is powered along the hinge and steering axes, and has suspension along the steering axis. Body 1 is the chassis and body 2 is the wheel. The connection point for the wheel body is its center of mass.

Member	Description
pos1	Steering axis in the chassis frame.
saxis1	Steering axis in the chassis frame.
haxis1	Hinge axis in the chassis frame.
haxis2	Hinge axis in the wheel frame.
skp	Proportional (spring) constant for suspension.
skd	Derivative (damping) constant for suspension.
svel	Steering desired velocity.
sfmax	Maximum force.

Member	Description
hlev	Hinge desired velocity.
hfmax	Maximum force.
slock	1 if steering axis locked at angle = 0.
shi	Suspension hi limit.
slo	Suspension low limit.
sref	Suspension attachment point (reference).
slsoft	Suspension limit softness.
pad[2]	PADDING - unused.

Fixed-Path Fixed-Orientation Joint: MdtBclFPFOJoint

Constrains both the position and the orientation of a physical body. One possible use is to drive the motion of a physical body from a fixed animation.

Member	Description
pos1	Position of joint (1st body reference frame).
pos2	Position of joint (2nd body reference frame).
rel_ori	Invariant relative orientation of the bodies.

A fixed path constraint is used to kinematically control the movement of a dynamic body. This means a dynamic body may be forced to animate (translation or rotation) and the resulting forces are passed on to other constrained bodies.

Linear1 Joint: MdtBclLinear1Joint

Restricts the motion of a specified point of the first body to a specified plane of the second body. You can think of this joint as a “planar constraint.” Does not constraint orientation.

Member	Description
pos1	Position of joint (1st body reference frame).
pos2	Position of joint (2nd body reference frame).
displacement	Invariant relative orientation of the bodies.
pad[3]	padding - not used

Linear2 Joint: MdtBclLinear2Joint

Constrains two bodies so that they are free to slide up and down the shared axis (where they are joined), and also to rotate at any angle. Mechanical engineers would call this a “cylindrical joint.”.

Member	Description
pos1	Position of joint (1st body reference frame).
pos2	Position of joint (2nd body reference frame).
direction	Vector fixed in the body[1] reference frame (or the inertial frame, if no second body) on which the joint position is constrained to lie.
vec1	A vector fixed in 2nd body reference frame which is perpendicular to the joint direction.
vec2	A vector fixed in 2nd body reference frame which is perpendicular to both the joint direction and vec1.

Universal Joint: MdtBclUnivJoint

An extension of the ball-and-socket joint which also prevents rotation about one axis. The specification of the joint includes an axis on each body: the constraint maintains orthogonality between those two axes.

Member	Description
pos1	Position of joint (1st body reference frame).
pos2	Position of joint (2nd body reference frame).
axis1	Joint axis 1 (fixed in 1st body ref. frame).
axis2	Joint axis 2 (fixed in 1st body ref. frame).
axis1_dot_axis2	Dot product of the initial orientations of the two axes in the inertial reference frame: invariant for this joint.
pad[3]	padding - not used.

Fixed-Path Joint: MdtBclFPJoint

A fixed path constraint is used to kinematically control the movement of a dynamic body. This means a dynamic body may be forced to animate (translation only for this constraint type) and the resulting forces are passed on to other constrained bodies. See also *Fixed-Path Fixed-Orientation Joint: MdtBclFPFOJoint* on page 90.

Member	Description
pos1	Position of joint (1st body reference frame).
pos2	Position of joint (2nd body reference frame).
vel1	Kinematic velocity of 1st body (1st body reference frame).
vel2	Kinematic velocity of 2nd body (2nd body reference frame).

Calling MdtBcl and MdtKea Directly

To optimize your program, you can access the MdtKea and MdtBcl structures directly, rather than using higher-level calls to the Mdt Library.

Because the source code to the Mdt Library is distributed with the Dynamics Toolkit, you can see how it calls the lower-level APIs (MdtBcl and MdtKea), and customize that code or write your own to make your simulation as efficient as possible.

Example source code: KeaOnly.c

KeaOnly.c is an example program distributed with the Dynamic Toolkit. It implements a simple pendulum, with and without direct calls to Kea Solver API. This example shows how to write an application that calls the Kea Solver directly, and contrasts it (using `ifdefs`) to an application that uses the Dynamics Toolkit's top-level API, the Mdt Library.

When not using the Mdt Library, this program will feature the following:

- Rather than calling `Mdt.h`, this program will include `MdtBcl.h`, and `MdtKea.h` directly. This eliminates the Mdt Library from the program.
- Rather than using the functions of the Mdt Library to create and manage bodies, it allocates Kea-level bodies and assigns values directly to their members.
- Rather than using `Get/Set` functions to create and manage constraints at the Mdt Library level, it allocates MdtKea constraints and assigns values directly to their members.
- Rather than simply calling `MdtWorldStep()` to evolve the simulation through each time increment, it calls lower-level functions in the MdtBcl and MdtKea libraries.

```
#include <malloc.h>
#include <string.h>

#include "MdtKea.h"
#include "MeMath.h"
#include "MeViewer.h"

#define MEM_BLOCK_SIZE 10000
#define CUBOID_LENGTH 3
#define NUM_BODIES 1
#define NUM_CONSTRAINTS 1
/*
    Graphic-only definitions:
*/
RRender *rc;

char *help[1] = {"There is no user interaction for this example."};
```

```

/*
  Graphic-only object's transformation matrix:
*/
MeReal pivotTransform[16] =
{
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    0, 0, 1, 1
};

void *memory;

/*
  N.B. Renderer colour is never double precision:
*/
const float orange[3] = { 1, 0.4f, 0 };
const float blue[3]   = { 0, 0.598f, 0.797f };

RGraphicsDescription *r_cuboid = 0;
RGraphicsDescription *r_pivot = 0;
RGraphicsDescription *groundG= 0;

/*
  Graphic-only object's transformation matrix:
*/
MeReal groundTransform[16] =
{
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    0, -3, 0, 1
};

/* Physics object definitions: */

#define TIMESTEP          (MeReal)0.01
#define BODY_MASS         2

const MeReal gravity[3]  = { 0, -10, 0 };
const MeReal velocity[3] = { 0, 0, 1 };

/* Structure for Euler parameters, mass, moment of inertia etc. */
MdtKeaBody Cuboid;

/* Quaternion, moment of Inertia etc. */
MeReal CuboidTransform[16]

```

```

#ifdef PS2
__attribute__((aligned(16)))
#endif
= {
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    1, 0, 1, 1
}
;

/*
    The joint position in the cuboid reference frame is constrained to be
    constant * by the ball and socket joint:
*/
const MeReal pos1[4] = { -1, 0, 0 };

/*
    The joint position is fixed in the inertial reference frame in this
    single body demo.
*/
const MeReal pos2[4] = { 0, 0, 1 };

#define MdtBclNO_BODY -1

/*
    Definitions for a single ball and socket joint:

    Almost all these should be 'const'.
*/

#ifdef PS2
/*
    On the PS2, all input to kea should be quadword (16 byte aligned).
*/

#   define QALIGNED __attribute__((aligned (16)))

    int NumConstraints QALIGNED = (NUM_CONSTRAINTS);
    int NumDegreesOfFreedomConstrained QALIGNED = (NUM_CONSTRAINTS * 3);
    int JacobianRowOffset QALIGNED = (0);
    MeReal WorkVector[4] QALIGNED = { 0 };
    MeReal DummyVector[4] QALIGNED = { 0 };

    int BodyIndex[3] QALIGNED = { 0, MdtBclNO_BODY, MdtBclNO_BODY };
    MeReal ForceUpperLimit[4] QALIGNED
        = { MEINFINITY, MEINFINITY, MEINFINITY, 0 };
    MeReal ForceLowerLimit[4] QALIGNED

```

```

        = { -MEINFINITY, -MEINFINITY, -MEINFINITY, 0 };

/* Read-only */
MeReal ResultantForces[24] QALIGNED;
/* Read-only work vector. */
MeReal Eigenvalues[4] QALIGNED;
MeReal ErrorTerm[4] QALIGNED;
#else
    int NumConstraints                = (NUM_CONSTRAINTS);
    int NumDegreesOfFreedomConstrained = (NUM_CONSTRAINTS * 3);
    int JacobianRowOffset              = (0);
    MeReal WorkVector[4]               = { 0 };
    MeReal DummyVector[4]              = { 0 };

    int BodyIndex[3]                  =
        { 0, MdtBclNO_BODY, MdtBclNO_BODY };
    MeReal ForceUpperLimit[4] =
        { MEINFINITY, MEINFINITY, MEINFINITY, 0 };
    MeReal ForceLowerLimit[4] =
        { -MEINFINITY, -MEINFINITY, -MEINFINITY, 0 };

/* Read-only; */
MeReal ResultantForces[24];
/* Read-only work vector. */
MeReal Eigenvalues[4];
MeReal ErrorTerm[4];
#endif

/*
    The following allocates space for 3 J matrices.

    This demo shows only a single body constrained by this joint, so only
    the first Jacobian is actually used here.

    Each Jacobian has 6 columns for the velocity factors (velocity x,y,z
    followed by angular velocity x,y,z).

    One degree of freedom is constrained for each non-zero row in the
    Jacobian, so we need only the first three rows in this demo.

    However, Kea expects all memory to be aligned on 4-byte boundaries, so
    we allow space for 4 rows.

    The matrix layout is row-major for each "strip" of 4 rows, but
    column-major within each strip. See the KEAJ macro below.
*/

/* 3 * 6 * 4 = 72 */

```

```

#ifdef PS2
    MeReal Jacobian[72] __attribute__((aligned(16)));
#else
    MeReal Jacobian[72];
#endif

MdtKeaConstraints BallAndSocketJoint;

#define KEAJ(i,j,k) \
    (Jacobian[ (k)*4 + ((j)/4)*18*4 + (j)%4 + (i)*24 ])

#define EPSILON 0.01f
#define GAMMA 0.2f

MdtKeaParameters EvolutionParams = { TIMESTEP, EPSILON, GAMMA, 0, 0 };

void InitBallAndSocketJoint(void)
{
    memset(&BallAndSocketJoint, 0, sizeof (MdtKeaConstraints));

    /*
       Note that the integer addresses would be arrays if there were
       multiple partitions:
    */
    BallAndSocketJoint.num_partitions = 1;
    BallAndSocketJoint.num_constraints_partition = &NumConstraints;
    BallAndSocketJoint.num_rows_partition =
        &NumDegreesOfFreedomConstrained;
    BallAndSocketJoint.Jsize = &NumDegreesOfFreedomConstrained;
    BallAndSocketJoint.num_rows = NumDegreesOfFreedomConstrained;
    BallAndSocketJoint.Jofs = &JacobianRowOffset;

    /*
       Input and output solver matrices, with padding for 4-byte
       alignment:
    */
    BallAndSocketJoint.Jbody = BodyIndex;
    /* Used by kea. */
    BallAndSocketJoint.xgamma = WorkVector;

    /* No slipping constraints in this example. */
    BallAndSocketJoint.slipfactor = DummyVector;

    BallAndSocketJoint.lambda = Eigenvalues;
    BallAndSocketJoint.force = ResultantForces;
    BallAndSocketJoint.hi = ForceUpperLimit;

```

```

    BallAndSocketJoint.lo = ForceLowerLimit;

    /* No joint actuation or target velocities in this demo. */
    BallAndSocketJoint.c = DummyVector;

    BallAndSocketJoint.xi = ErrorTerm;
    BallAndSocketJoint.Jstore = Jacobian;
}

/*
   This function assumes that the cuboid may be translated, but is not
   rotated.
*/
void InitCuboid(const MeReal mass, const MeReal MoI[3])
{
    memset(&Cuboid, 0, sizeof (MdtKeaBody));
    Cuboid.invmass = ((MeReal) 1) / mass;

    /* Set the quaternion to the identity quaternion: */
    Cuboid.qrot[0] = 1;

    /* Set the moment of inertia matrix: */
    Cuboid.I0[0] = MoI[0];
    Cuboid.I1[1] = MoI[1];
    Cuboid.I2[2] = MoI[2];

    /* Set the inverted moment of inertia matrix: */
    Cuboid.invI0[0] = 1 / MoI[0];
    Cuboid.invI1[1] = 1 / MoI[1];
    Cuboid.invI2[2] = 1 / MoI[2];

    /* Set the initial velocity of the cuboid: */
    Cuboid.vel[0] = velocity[0];
    Cuboid.vel[1] = velocity[1];
    Cuboid.vel[2] = velocity[2];
}

/*
   Set +{A} to a 3x3 matrix corresponding to the 3x1 vector +{a}, such that
   m{A*b = a x b} (x is the cross product operator).
*/
void CreateCrossMatrix(const MeReal a[3], MeReal A0[3],
    MeReal A1[3], MeReal A2[3])
{
    A0[0] = 0;
    A0[4] = a[2];
    A0[8] = -a[1];

```

```

    A1[0] = -a[2];
    A1[4] = 0;
    A1[8] = a[0];

    A2[0] = a[1];
    A2[4] = -a[0];
    A2[8] = 0;
}

void UpdateBallAndSocketJoint(void)
{
    int i;
    MeReal p[4];

    memset(ResultantForces, 0, 24 * sizeof (MeReal));
    memset(Eigenvalues, 0, 4 * sizeof (MeReal));
    memset(Jacobian, 0, 72 * sizeof (MeReal));
    memset(WorkVector, 0, 4 * sizeof (MeReal));

    /*
    Kea solves  $J \cdot v = c$  for  $v$ . The latter is a 6-dimensional composite
    of velocity ( $v$ ) and angular velocity ( $w$ ).

    The vector  $c$  is always zero in this demo.

    The  $J$  matrix must be set for each call to the solver (MdtKeaStep)
    for the appropriate values to enforce the required constraint.

    For a single-body ball and socket joint, the required condition
    is:  $v - p \times w = 0$  where  $p$  is the joint position relative to
    the constrained body, in inertial reference frame coordinates.
    */

    /*
    Set the left 3x3 block of Jacobian[0] to the 3x3 identity matrix:
    */

    KEAJ(0, 0, 0) = 1;
    KEAJ(0, 1, 1) = 1;
    KEAJ(0, 2, 2) = 1;

    /*
    Calculate current joint position relative to cuboid in inertial
    reference frame:
    */
    MeMatrixMultiply(4, 3, 1, CuboidTransform, pos1, p);

```

```

    /*
       Set the right 3x3 block of Jacobian[0] to the 3x3 cross-product
       matrix of p:
    */
    CreateCrossMatrix(p,
        &KEAJ(0, 0, 3), &KEAJ(0, 1, 3), &KEAJ(0, 2, 3));

    /* Compute position error: */
    for (i = 0; i < 3; i++)
        ErrorTerm[i] = (p[i] + CuboidTransform[12 + i]) - pos2[i];
}

/*
   External forces (such as gravity) acting on physical bodies must be
   updated every tick:
*/
void UpdateCuboid(const MeReal mass)
{
    int i = 0;

    for (; i < 3; ++i)
    {
        /* No other external or damping forces */
        Cuboid.force[i] = mass * gravity[i];
        Cuboid.torque[i] = 0;
    }
}

void Tick(RRender * rc)
{
    const MeReal *Force_on_cuboid;
    const MeReal *Torque_on_cuboid;

    UpdateBallAndSocketJoint();
    UpdateCuboid(BODY_MASS);

    MdtKeaStep(BallAndSocketJoint, &Cuboid,
        (MdtKeaTransformation * const) CuboidTransform, NUM_BODIES,
        EvolutionParams);

    /*
       Read-only ResultantForces array has now been updated - this gives
       the force on the body due to the constraint. The force and torque
       properties of the Cuboid structure have also been updated with the
       total force on the body.
    */
    Force_on_cuboid = ResultantForces; /* x,y,z, padding */
}

```

```

    Torque_on_cuboid = ResultantForces + 4; /* x,y,z, padding */
}

void cleanup(void)
{
    RDeleteRenderContext(rc);
}

/*
    Entry Point
*/
int main(int argc, const char **argv)
{
    const RRenderType renderer = RParseRenderType(&argc, &argv);

    const MeReal Ratio = (10); /* Length : width ratio > 1 */
    const MeReal IRSq = (1 / (Ratio * Ratio));

    MeReal I[3];

    memory = malloc(MEM_BLOCK_SIZE);

    EvolutionParams.memory_pool = memory;
    EvolutionParams.memory_pool_size = MEM_BLOCK_SIZE;
    InitBallAndSocketJoint();

    /*
        Set MoI for a solid cuboid w,h = r * x, d = x, where w,h,d are
        half-width, half-height etc.
    */
    I[0] = I[1] = BODY_MASS
        * (1 + IRSq) * CUBOID_LENGTH * CUBOID_LENGTH / 12;
    I[2] = BODY_MASS
        * 2 * IRSq * CUBOID_LENGTH * CUBOID_LENGTH / 12;

    /* Transformation matrix set by direct initialisation. */
    InitCuboid(BODY_MASS, I);

    /*
        Renderer.
    */
    rc = RNewRenderContext(renderer, kRQualitySmooth);
    rc->m_cameraLookAt[0] = 0.0;
    rc->m_cameraLookAt[1] = 0.0;
    rc->m_cameraLookAt[2] = 0.0;
    rc->m_cameraElevation = 0.0;

    /* --180 degrees */

```

```

rc->m_cameraOffset = 3.0f;
RUpdateCamera();

r_pivot = RCreateSphere(rc,
    0.5f * CUBOID_LENGTH / Ratio, orange, 0);
r_pivot->m_matrix = pivotTransform;
r_cuboid = RCreateCube(rc,
    CUBOID_LENGTH / Ratio, CUBOID_LENGTH / Ratio,
    CUBOID_LENGTH, blue, 0);

r_cuboid->m_matrix = CuboidTransform;

groundG = RCreateCube(rc, 7, 0.2f, 7, blue, groundTransform);
RSetTexture(groundG, "checkerboard");

RCreateUserHelp(help,1); /* press F1 for onscreen user help */
rc->m_title = "KeaOnly V.1.0 tutorial - www.mathengine.com";

#ifdef PS2
    atexit(cleanup);
#endif
    RRun(rc, Tick);

    return 0;
}

```

Chapter 6 • Constructing Good Simulations

Overview

To get an efficient, well-behaved and reliable simulation using the Kea Solver, there are many issues to consider. These issues apply whether you are using the Kea Solver directly (see *Chapter 5: Customizing and Optimizing*), or through the Abstraction Layer (see *Chapter 3: Simulating Rigid Bodies and Forces* and *Chapter 4: Constraints: Joints and Contacts*).

It is important to keep in mind that once numerical analysis is used to determine the behaviour of a physical system instead of the exact physical solutions, as in Kea, the end result is always an approximation. Kea is therefore inherently limited to approximate the physical behavior of a system. The quality of this approximation is a direct consequence of the choices the user will make when implementing Kea.

This chapter will discuss techniques for making “good” simulations. The guidelines presented in this chapter are only rules of thumb—every simulation is different.

Integrators and First Order Effects

Given a set of differential equations, different numerical methods yield different solutions. Every numerical method is a compromise. One has to consider how much work is required to obtain a given accuracy, choose what accuracy is appropriate for the task, verify that the method is robust against discontinuity

The three aspects that are used to choose a good compromise is the amount of work necessary to reach a given accuracy, the accuracy desired and the stability against stiffness i.e., abrupt changes in the solutions.

Kea uses a semi-implicit first order integrator, which gives good stability and speed when used properly. This section explains why this is a good compromise between other types of integrators, and explains some consequences you should be aware of when constructing simulations.

Background

The integrator is the part of Kea that evolves the rigid body system from its current state to a new state, h seconds in to the future (where 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—though they are also slower, because they require more evaluations of the system forces for each time step.

All integrators suffer from some degree of inaccuracy: that is, the simulated system doesn't behave exactly as the system would in real life. In most applications inaccuracy can be tolerated, except if quantitative measurements are being made for scientific or engineering purposes.

However, a byproduct of inaccuracy in explicit integrators is that it can result in the system gaining energy for no apparent reason—the pieces move faster and faster until they “explode”. This is called instability. Higher order explicit integrators have added stability, but they are slower too, so there is a trade off.

Kea's integrator is semi-implicit (not explicit) and first order. This allows the best combination of stability and speed. This is because a semi-implicit integrator gets its stability in different ways from an explicit integrator: the inaccuracy results in energy being drained from the system rather than added to it.

Since Kea's integrator is not fully implicit, there are still some situations where you still have to be careful as you would for an explicit integrator: see *Stiff Forces and Stability* on page 107.

First Order Effects

The first order integrator in Kea is not as accurate as a high order integrator. This is not a problem, because Kea maintains its stability in other ways. But it does have one consequence you have to watch out for. As the step size h is changed, the behavior of the simulation will change slightly. For example, if you take 10 steps of $h = 0.1$ seconds (for a total of 1 second of simulated time), the state the system ends up in will not be quite the same as if you take 5 steps of $h = 0.2$ seconds. With a higher order integrator the difference would be less obvious.

The choice of the stepsize h is of great importance for such integrators as it is directly related to the severity of the discrepancy between a real physical behaviour and the computed one. Usually, one can reduce this discrepancy by reducing the value of the stepsize h . However, using a smaller stepsize means you will have to take more steps each frame to get objects to appear to move at the same speed, not that any individual timestep takes longer.

When you are tweaking your real-time game, be sure to do so at the frame rate at which your users will be running the game. In particular, adaptive time stepping schemes may be difficult to control. For example, even in cases where frames are being rendered as fast as possible, a fixed simulation rate of say 30 frames per second is desirable.

Stiff Forces and Stability

A “stiff” system is one where strong forces can be generated by changing the state of the system slightly. For example, a rigid body system with a very strong spring is like this: if you pull the bodies that the spring connects apart, there will be a very strong restoring force.

Stiff systems are hard for explicit integrators (like explicit Euler and Runge-Kutta) to integrate stably (that is, without adding extra energy). Fully implicit integrators can integrate stably, because they tend to drain energy away instead of adding it when there are stiff components present.

Kea uses a semi-implicit integrator, which compromises 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.

For example, high gravity should not cause a problem (apart from making things move faster!) and stiff suspension on a car can be stably simulated using a car-wheel constraint, but would be unstable if simulated using 'MdtAddForce' to the wheels each timestep.

If you are applying forces to rigid bodies and the simulation appears to be unstable, try reducing the magnitude of the forces or limiting them to a maximum value.

The Meaning of Epsilon and Gamma

The Kea integrator requires two parameters to step the system from one state to the next. These are `epsilon` (ϵ) and `gamma` (γ).

What Epsilon Does

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

Sometimes a rigid body system may not have a solution for the forces. For example, some systems can get into “singular” configurations, or redundant constraints may have been specified on the system (see *Avoiding Over-Determinacy* on page 116). 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 force solution being amplified too much
- The constraint solver may take too many iterations to find a solution (on some platforms you will get a warning message indicating this).

In both cases the problem can be fixed by increasing `epsilon`. The only disadvantage of increasing `epsilon` is that it makes the joints and constraints a little bit springy instead of being perfectly solid. However this is hardly ever a problem. Quite low values of `epsilon` can be effective (such as the default 0.01), and it takes a quite high value of `epsilon` (say 1 through 10) before any constraint springiness problems are visible.

If you are dealing with very large mass objects (which you shouldn't be - see *Mass Problems* on page 110), you may have to decrease `epsilon` to make contacts 'harder' and prevent visible penetration due to large gravity forces.

For those who are interested, here are the numerical details. Every time step Kea solves a matrix equation of the form:

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

Where A is positive semidefinite. Degenerate systems result in A being singular; accordingly, if ϵ were zero, this equation would have no solution. By making ϵ greater than zero a solution for x can be guaranteed.

What Gamma Does

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

Kea minimizes this problem through a process called projection, which is controlled by gamma, a number between 0.0 and 1.0.

When positions are updated, projection moves the rigid bodies gamma of the way back to their correctly constrained configurations. Gamma also applies to contacts - things usually penetrate a bit before a contact is generated, and the bodies have to be projected apart again:

- If $\gamma = 0$, no projection is done at all: as the simulation progresses, the joints will “fall apart”.
- If $\gamma = 1$, perfect projection is attempted: Kea tries to get the exactly constrained configuration at the next time step. However, because of various approximations that Kea uses, the projection will not be perfect, and may cause problems in some situations.

For most simulations a value of gamma in the range 0.1 through 0.8 will work well. Values less than zero or greater than one will cause problems.

Instability due to Mass and Inertia Problems

Sometimes the simulation may become unstable, that is the rigid body positions and velocities will explode (disappear off to infinity) for no apparent reason. One of the possible causes is a poor distribution of mass.

Mass Problems

For example, if there is a mixture of very heavy and very light bodies in an articulated structure, then the heavy bodies can transfer their energy to the light bodies, which will cause them to move much more quickly because of conservation of momentum.

An example of this kind of system is a whip, which 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 and small masses in the system is not too high. For example, a ratio of 1:100 is a reasonable maximum in many situations.

Inertia Problems

A related problem occurs when the system contains bodies with inertia tensors that have components that are large on one axis and small on another axis. This is somewhat similar to the mass ratio problem.

For example, if a long thin cylinder experiences a high rotating force along its long axis it may rotate too fast to be simulated properly. In this case the inertia tensor along the long axis may have to be increased.

As a rule of thumb, a more stable simulation can be created by having all the rigid bodies have the same order of magnitude mass and inertia. Stability problems can often be traced to having a range of very large and very small masses in the system.

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.

Numerical Scaling

For Kea to perform simulation stably and efficiently, it is good to keep lengths and masses in a relatively small range around 1.0. How restrictive you should be depends on whether `MeReal` is defined to be `float` or `double` in `MePrecision.h`.

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

Constraint Solver Iterations

Every time Kea takes a step, its 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 it will tell you that a “cycle has been detected” or that “The maximum number of iterations has been exceeded”. Even if you get a warning from the solver - the behavior will usually be stable enough. If there are visible problems then you may need to increase epsilon.

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. The only current solution for this problem is to avoid this kind of situation.

Modeling Using Force-Limited Motors

The hinge and prismatic joints both have an optional motor that can be applied along the rotating or sliding axis, to control movement along that axis. You can set the following parameters in the `MdtBclHingeJoint` and `MdtBclPrismaticJoint` structures:

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

When `powered` is set to 1, Kea tries to achieve 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. This action is somewhat similar to an engine with a maximum amount of output torque.

Force-limited motors are a good, 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. But remember that it adds the computational cost of another constraint.

Additionally, it is possible to model dry friction in a joint by setting a target velocity of zero (`desired_vel = 0`). This is a good model of a disc brake, except that heat and other nonlinear effects are not modeled. Note that the braking force is controlled by `fmax`.

Prefer Joint Limits to Contacts

Wherever possible, use joint limits rather than contacts to control the movement of joints.

While it is possible to use the contact of two objects connected by a joint to prevent movement, it is not a good idea. The more contacts generated in the system, not to mention the necessary collision detection, the slower your system will run.

Hinges and prismatic joints allow limits to be set to their movement. This prevents self-collision, so your application won't need to check for it.

Avoiding Over-Determinacy

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 body motions, then the system is called over-determined.

For example, when a box is 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 give good motion to the box.

Imagine a chain of boxes resting on the ground—that is, each box is resting on the ground and touching its neighbor or neighbors. Only two contact points are needed between each box and the ground to give the correct motion. Extra contacts would result in over-determinacy.

Another example is a hinge joint. A hinge takes away five degrees of freedom (DOFs) of the two bodies that it connects. But imagine you model 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), which is one too many for a hinge.

Over-determined systems are bad because the Kea solver can have a hard time generating a solution for such systems: the solution may be inaccurate, or take extra time.

In an ideal world, over-determinacy would never arise, because every rigid body system would be described optimally in terms. This is much easier said than done; in practice, it is hard to tell whether a given system is over-determined or not.

Here are a few guidelines that you should follow:

- Use the minimum number of constraints between objects that gives the correct behavior: the lower the number of constraint is, the faster the simulation will be. Finding this minimum is usually a matter of trial and error.
- Use the correct joint types to get the motion you desire. For example, don't use two ball-and-socket joints to model a hinge.
- Don't have conflicting joints between the same bodies, that is, don't have two or more joints that constraint the motion of the bodies in the same way.
- When in doubt, increase `epsilon`. This will always have the effect of reducing over-determinacy of any system.

Hinge Joint Limits

Hinge joint limits are currently limited to the range $-\pi$ through π . If you:

- Set a high or low limit that is close to $\pm\pi$, and
- The velocity of the hinged bodies around the hinge axis is high, and
- The hinge is approaching the limit

then Kea may step past the limit, so the limit will be effectively ignored. This is because Kea only measures the hinge angle between two bodies in the range $\pm\pi$, and if a hinge moves past its limit as well as outside the range $\pm\pi$ in a single time step, the incorrect angle will not be detected.

To prevent this problem, either reduce the magnitude of the hinge limits or ensure that the hinge angle does not change too quickly.

Fast Spin Axis

Every rigid body can have an optional fast spin axis set on it. If this is set, 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 along the fast spin axis, and relatively slowly along the other axes. This is particularly useful in the case of a wheel on a car, which may be rotating very quickly. Using the standing orientation update may result in a large error that makes the wheel's hinge axis appear to bend.

MathEngine Toolkits Glossary

Acceleration	Time rate of change of velocity, or how fast velocity is increasing.
Actuator	A mechanical device for moving or controlling something, such as a motor, a lever, a piston, and so on. Actuators can be implemented with forces and torques or with velocity constraints.
Angular Velocity	The rate of rotation about an axis. This is a vector quantity whose direction points along the instantaneous axis preserved by the rotation and the magnitude is the rate of rotation about this axis. It is usually expressed in radians or revolutions per second, but can be expressed in any unit of time.
Ballistics	Ballistics is the science of the motion and behavior of projectiles, missiles, bullets and other similar objects in flight submitted to a gravitational field. For MathEngine, a projectile is a SRB (single rigid body) and is not <i>in contact</i> with anything – including the atmosphere. We do not calculate the effect of air drag on a projectile's flight. Projectiles, such as bullets, travel in a parabolic trajectory when submitted to a gravitational field or one that becomes more and more curved as range increases and velocity drops off.
Ballistic Objects	In a games development environment, ballistics objects are usually single rigid body projectiles that are subject to ballistic calculations without contacts or impulses.
BSP Tree	Binary Space Partitioning (BSP) tree. It is a method for describing the geometry of objects in 3D graphics. BSP trees are tools for organizing this geometry and extracting information about geometrical relationships. In 3D worlds they are often used for visibility determination, and hidden surface removal.
Cartesian Coordinates	A coordinate system which is based on orthogonal axes. This is the standard x, y and z coordinate system, in contrast to polar coordinates on a sphere. In many 3D graphics applications, x usually stands for left-right position, y describes vertical position and z gives the position along the line of sight, (i.e., depth). Traditionally, graphics viewer use Left Handed system.
Center of Mass	The point in a body or system of bodies at which the whole mass may be considered as concentrated. For the purpose of dynamics calculations, a body or system of bodies can be abstracted to a point mass located at its center of mass coordinates, simplifying the computation greatly.
Collision Detection	A software process that determines if two objects are geometrically interpenetrating or touching. If a collision took place it then determines the location and the local geometry of the object near the point of contact. Collision detection also determines the spatial relationships

between objects, such as the approximate separation distance between them.

Collision Model

A geometrical representation of an object used for collision detection. MathEngine supports a variety of primitives such as sphere, box, plane, cylinder and cone. The collision model may be identical to the model used for rendering or maybe an approximation that allows for faster collision detection. Collision models can also be made up of a combination of collision primitives

Collision Response

The change of motion of an object which results from a collision with another object of the world, or when a joint limit is reached. Note that collision response is related to the dynamics properties of the object in contacts.

Constraint

An external restriction on the motion of an object or body. MathEngine uses two kinds of constraints: joints and contact. An *equality* restriction could be that a joint attached to a specific location of a body should not be allowed to wander off this point, that is, their attachment coordinates should be the same. An *inequality* restriction could be a ball placed in a room: allowed to move freely on the floor as long as it stays inside the area bordered by the walls.

These restrictions are used to connect objects together, such as connecting two rigid bodies with a hinge, or to impose motion in a joint by applying a force limited actuator. Constraints can be equality restrictions as is the case for joints between bodies, or inequality restriction as is the case with contact constraints and joint limits. Constraints are called ideal or non-ideal according to whether or not they dissipate energy. A ball and socket joint is an ideal constraint but a box friction contact constraint is non-ideal.

Constraint Equations

A mathematical relation between the coordinates of several bodies..

Culling

Culling is used to describe the process of selecting items for removal from a list. For example, the process of eliminating certain polygons that shouldn't be seen before drawing a scene.

Cycle

A cycle is used to describe the effect of the force applied to a rigid body in an articulated chain of bodies. The force is relayed through each body in the chain and is eventually applied, at least in part, back to the original body.

Convex Object

A geometrical object with no holes in it, such as a sphere or a cube. More precisely, an object is convex if given any two points of the interior or the surface of the object, all the points that lie on the line segment between those two points are also inside or on the surface of the object. A doughnut is not convex for instance.

Concave Object	Nonconvex object.
Damping	This is a resistance to the motion of a body. It results in the body slowing down and being gradually brought to rest.
Differential Index	The number of times a DAE must be differentiated to obtain a standard set of ODEs, is the index of the DAE. If the index is one, the system may be solved nearly as simply as if it were an ODE.
Dynamics	Dynamics is the part of mechanics (which, of course is a branch of physics) that is concerned with the causes of accelerating motion. It describes the change in motion of objects or particles using the concepts of force and mass. It is governed by the three basic laws of motion, called Newton's laws. See Newton's Laws
Dynamic Simulation	Dynamics simulation is achieved by attaching physical properties and applying forces to objects, and computing the resulting behavior according to the physical laws of motion.
Force	A force is most easily described as a push or a pull – it is what causes the state of motion (momentum) of an object to change. If you want something to change the way it's moving, (change its direction or make it go faster, for example) you must apply a force to it. Forces are vector quantities with orientation and magnitude. The bigger the force, the faster the state of motion will change. In 3D games' worlds, forces typically include gravity and wind.
Force-limited Actuator	This is a special model of actuator provided by the MdtKea Library. It allows you to control the relative velocity of two bodies connected by a prismatic joint using an external source such as mouse or keyboard input. You set the desired target velocity, which will be achieved provided that the force required to do so is less than the limit specified, otherwise, the maximum force is applied. Force limited actuators respond very quickly and are inherently stable; they are the best way to introduce user controlled motion in a joint in the Kea library.
Frame	A single rendered screen of graphics,
Frame Rate	The number of screen images displayed per second, usually abbreviated by <i>fps</i> (frame per second). Common frame rates are 60fps for games consoles, 25fps for PAL video, and 30fps for NTSC video, although other speeds may be used
Friction Force	The force that resists relative motion between two bodies in contact. When bodies are sliding, this force opposes the relative motion and dissipates energy. Otherwise, this force prevents the bodies from

sliding, in which case it does not dissipate energy. In both case, friction force is related to normal force, that is, the force that prevents inter-penetration of the two bodies

GJK Algorithm

A fast method of doing collisions between a set of convex objects.

GJK Engine

The part of the collision detection software that implements the GJK algorithm.

Gravity

In a Newtonian world, gravity is a force that attracts all objects to each other. Gravitational force between any two given objects is proportional to the products of their masses and inversely proportional to the square of the distance that separates them (things are a bit different at velocities near the velocity of light and for very large masses).

However, we are mostly interested in constant and uniform gravity which is the force that keeps your chair on the floor. For objects near the surface of the earth, gravity causes objects to accelerate downward at a constant rate of 9.81 m/sec^2 , independent of their mass. This means that gravity causes an object's velocity to change by 9.81 m/sec^2 downward for every second the object is under the influence of gravity.

Hooke's Law

This law states that the force on a object displaced from a point of equilibrium is proportional to the displacement and in the direction opposite to the displacement.

Ill Conditioning

In a system of linear equations, this happens when some of the equations are close to being redundant (e.g., the linear equations are almost linearly dependant) and therefore, the matrix of the linear system is close to being non-invertible . If an ill-conditioned linear system is solved using a standard method, there can be very large errors in the answer.

Impulse

A change in momentum that happens over a time interval, causing a finite change in the velocity of the object on which the impulse is applied.

In Kea, this time interval is divided into a number of time step and the change of velocity is computed successively for each of them.

Inertia

The tendency of an object to maintain its state of rest or of uniform motion. The resistance to change in the quantity of linear motion (linear momentum) is given by the mass of the object. Resistance to change in the quantity of angular motion (angular momentum) about an axis is a moment of inertia about that axis.

Inertia Tensor	The set of numbers that describe the resistance to change in angular motion. This consists of several numbers because, in any given orientation, a body may have different inertia to rotation about all three axes, x, y and z. This quantity also changes with the orientation of the body, which is why it is a tensor. The inertia tensor can be represented as a 3x3 matrix which is symmetric and positive definite.
Inertial Reference Frame	A reference system in which the Newton's laws of motion holds. See <i>Newton's Law</i>.
Integrator	This is the term describing the algorithms and resulting code implementation that calculates the position and orientation of physical objects in a MathEngine world. MathEngine provides both an explicit and a semi-implicit integrator. An explicit integrator works by directly adding small quantities to the current state of the system, while an implicit system involves solving for the root or minimal point of an algebraic equation. Explicit integrators are usually faster than implicit ones but are also prone to larger integration errors. The MathEngine semi-implicit integrator combines the qualities of both implicit and explicit integrators by being fast and precise at one and the same time.
Joint	An equality constraint or connection between objects or bodies. When two or more bodies or objects are connected by joints, they form a multibody system which is also called a jointed body or an articulated body.
Kinematics	Kinematics is the study of how things move in relation with time. It describes the different types of motion a body can undergo: translational (change of position), rotational (change of orientation), vibrational (change of shape and size).
Forward Kinematics	Allows you to position a complex jointed object quickly by positioning the first body in a jointed body or an articulated chain of bodies.

Linear Complementarity Problem

This is much like solving a linear $Mz + q = 0$. However, in the LCP, we require the solution vector z to have positive components. For example, to solve for the normal forces at points of contact, in which case the normal force must be positive to push things apart and prevent inter-penetration (a negative normal force corresponds to sticking). Since we restrict z to be positive, we won't be able to solve for the linear problem and there will be some left over, say $w = Mz + q$. We further specify the problem by requiring that w be also positive. For instance in Kea, w corresponds to the velocity in the direction normal to the contact plane at a contact point. Requiring that this be positive means that we don't allow the velocity to be in the direction that violates the non-penetration condition.

The complementarity condition means that for each variable w_i, z_i , either $z_i > 0$ and $w_i = 0$, in which case we say that the constraint is active or tight, or $z_i = 0$ and $w_i > 0$, in which case we say that constraint i is free. In other words, the product $z_i * w_i = 0$ for each i . What this means is that at a contact point, either the normal force is positive and the normal velocity is zero (hard contact), or the normal force is zero and the velocity is positive in which case we have a grazing contact and the objects are coming apart. In a multibody situation where the rigid bodies in contact are also connected together with joints, this problem becomes difficult since we must produce a consistent assignment of touch and graze conditions which might be conflicting with each other.

The LCP is related to several other optimization problems like quadratic programs, bimatrix games and so on. However, contrary to standard optimization problem, LCPs don't have a cost function and the problem is almost purely combinatorial which makes it NP hard.

Local Coordinates

The origin of the local coordinate system of a body is usually located at its center of mass.

Jacobian Matrix

Given any vector function of several variables with components

$$F_1(x_1, x_2, \dots, x_n),$$

$$F_2(x_1, x_2, \dots, x_n),$$

....

$$F_n(x_1, x_2, \dots, x_n),$$

the Jacobian matrix is defined as:

$$J_{ij} = \frac{\partial F_i}{\partial x_j}$$

In the case where the functions F_j represents a coordinate transformation between x_i and $q_j = F_j$, then, the Jacobian matrix the linear approximation of the coordinate transformation in the neighborhood of the point where the derivatives of F_j are evaluated.

In a more general way, the Jacobian matrix is the local linear approximation of a non-linear function in the neighborhood of a point x_0 :

$$F(x) = F(x_0) + J(x-x_0)$$

In Kea, we use the Jacobians of the constraint functions in the computation of the constraint forces i.e., the forces required to maintain the constraint.

Mass

Mass is the measure of the amount of material contained within a body. It is independant of the location of the body.

Mechanics

A branch of physics that deals with the analysis of motion of physical objects. This includes the study of the equations of motion derived in dynamics.

Momentum

This is a quantitative measure of the state of motion of an object. Linear momentum is the product of mass and velocity: $\vec{p} = m\vec{v}$ and is a vector quantity. Angular momentum is a measure of the rotational motion of an object about some point. For a point mass m , this is defined as $\vec{L} = m\vec{r} \times \vec{v}$ or $\vec{L} = I\vec{\omega}$ where is $\vec{r}$ a vector from the point about which we are measuring the rotational movement, $\vec{r} \times \vec{v}$ is the cross product and $I = mr^2$ is the corresponding inertia tensor.

For a rigid body, the spin angular momentum is also defined as $\vec{L} = I\vec{\omega}$ where $\vec{\omega}$ is the angular velocity but the inertia tensor may be more complex than the mass point scalar value.

For a collection of objects, the total momentum is composed of three parts: linear momentum of the center of mass; angular momentum of the center of mass about some reference point; and spin angular momentum (i.e., rotational motion of the object about the center of mass).

Note that you can always find an inertial frame of reference in which both part 1-) and 2-) vanish at some point in time, but it may not be possible to find a frame of reference in which the spin angular momentum vanishes.

Note that Newton's second law is really a relation between the time rate of change of momentum and applied forces. The simplification to $F = ma$ is only good for point masses that have constant mass.

Multistep Integrator

This is an integrator that requires information from several preceding time steps. Multistep integrators assume a degree of smoothness about the solution (i.e., of continuity of its function and derivatives) which is not satisfied in a simulation of rigid bodies with collisions and friction. This is because collision and friction functions naturally tend to have discontinuity (i.e., abrupt change in their values).

Newton

The SI unit of force, derived from 3 basic units: mass, length and time.

Newtons' Laws

Newton's first law of motion: In an inertial frame, an object that is free of interaction has constant momentum. An object at rest remains at rest, and if it is in motion, that motion will be uniform and constant, that is, the center of mass will move on a straight line and the motion about the center of mass will be uniform. The tendency of an object to resist any change in motion is called **inertia** of the object. **Mass** is a measure of inertia.

Newton's second law states the relationship between an acting force, inertia of a body, and its acceleration as follows:

$$\frac{d\vec{p}}{dt} = \vec{F}$$

where p is the momentum of the object and F is the applied force. In words, this means that the force exerted on a body is equal to the rate of change of momentum of that body. For point masses with constant mass, this simplifies to:

$$m \frac{d\vec{v}}{dt} = \vec{F}$$

the familiar $F = ma$. As such, Newton's second law only applies to point masses and further analysis is required to derive the equations of motion for rigid bodies and composite objects.

Newton's third law states that for every action there is an equal and opposite reaction. If two bodies interact, the magnitude of the force exerted by body 1 on body 2 is equal to the magnitude of the force exerted on body 1 by body 2. These two forces are also opposite in direction.

Normals	Normals are unit vectors. The normal to a plane is a unit vector perpendicular to the plane. In 3D graphics, they indicate the visible side of an object.
Object	We use this term to describe virtually any entity that can be move—a point mass, rigid body, articulated body, multi-body system, and so on. We usually use it as a general term when what is being discussed is not restricted to rigid bodies. Object does not refer to object-oriented programming constructs.
Ordinary Differential Equation	A differential equation is a relationship between the rate of change for the variables of a system and the state of that system. The term <i>ordinary</i> means that the derivatives are taken with respect to only one variable, by contrast with <i>partial differential equations</i> .
Orientation	Describes a body's direction relative to the world or reference frame coordinates.
Power	The time rate at which work is performed (energy is transformed) by a system.
Quaternion	Quaternions form a four-dimensional algebra which is an extension to normal complex numbers. They can be used to represent rotations of coordinate axes in 3D worlds in a way that is free of singularity, in contrast to Euler angles, for example. With Euler angles there are difficulties when the pitch angle reaches 90 degrees - a problem that never occurs when quaternions are used. The four components of a unit quaternion which represents a 3D rotation can be thought of as a set of parameters for 3D rotations, as an alternative to Euler angles. Because they lack the singularities that Euler systems have, quaternions allow for smooth interpolations between rotations.
Restitution (coefficient of)	When designing your game, think of restitution as bounciness. In physics terms it is the ratio of the relative speeds of two bodies just after and just before a collision between them along a mutual straight line; it ranges from 0, for perfectly inelastic collisions (no bounce), to 1 (maximum bounce), for completely elastic ones. If you increase the value to a number greater than one, you'll get an effect like a pinball machine. It is a constant at moderate speeds.
Right—Hand Rule	A useful visual aid to represent a right-handed coordinate systems. Here's how it works: orient your right hand so that your thumb is perpendicular to the plane defined by the x- and y- axes, then curl your

fingers. If you can curl your fingers in the direction from the x axis towards the y axis, then your thumb will be pointing in the direction of the z axis.

Rigid Body	A solid object that doesn't change its shape or size, despite any forces being applied to it.
Rotation	In a 3D world, rotation of an object is its motion about an internal axis of symmetry relative to an internal coordinate system.
SI Units and Prefixes	SI stands for <i>Systeme International</i> . It is a standardized set of units for scientific measurement. Examples of SI units include: the Newton, metre, Hertz, second and Watt. Examples of SI prefixes include: mega, milli, tera and giga.
Single-Step Integrator	Unlike multistep integrators, a single-step integrator uses information only from the current time step (n, n+1) to compute the next position from the derivatives. This is the case in Runge Kutta integrators, for example. Single-step integrators can be explicit or implicit. They can also be multistage, in which case they need several derivative evaluations at different intermediate times spread over the time step interval to compute the position at the next time step.
Stiff Springs	Stiffness is a measure of how a spring resists stretching. A very stiff spring will hardly move at all—even if you pull very hard, whereas a spring with very little stiffness will stretch easily even if you pull it gently. The stiffness is related to the 'spring constant' usually represented by the letter <i>K</i>. Writing this down as a formula called Hooke's law for the extension of springs we get; Force = - (spring constant) * (length increase) Stiff springs create very large forces and this can be a problem for the integrator. The reason is that the force is never very large for very long. For a small mass attached on a stiff spring, the resulting motion for the mass is to stay very near the point of equilibrium. The force from the spring will change very rapidly from large to small to keep the mass in place. An integrator that is using a time step larger than half the period of the spring might overestimate the magnitude of the force and this can lead to an "explosion" i.e., a situation where the mass moves further and further away from the equilibrium point. Note also that the frequency of a spring is dependent on the mass attached on it and if you drop a small mass on a stiff spring, that will

produce a very high frequency which might be difficult for the integrator.

The constraint solver uses a first order integrator for the forces which doesn't check if forces are getting very large. This means that you should avoid stiff springs altogether. If you want to simulate a stiff spring, you should use a "relaxed" constraint instead i.e., a constraint that is allowed to be violated.

Time-Step

Time, like mass, is another of those rather undefined measures in physics. Everybody knows what it is, and agrees to measure it in seconds.

A time step is an interval of time, typically of the order of the frame rate. If you run a simulation at 60 fps, your time step should be $1/60$ s or shorter; if it is shorter, you will need to perform several steps between frames if you want to create a real-time simulation in which everything seems to move at the same rate as in the real world.

The simulation computes what the state of the system will be after an interval of time equal to the time step has elapsed, using current state and derivative information. Essentially, the simulator uses current velocities and forces and extrapolates them to compute an approximation of the state into the future. The correctness of the approximation is related to the time step: the bigger the time step the worse the approximation.

Tracking Problem

The problem of keeping track of the distance between objects as simulated time is stepped forward. This is useful to estimate when objects will collide.

Torque

A torque can be described as a turning or twisting force. It's the measure of a force's tendency to produce a rotation about an axis, which also produces a change in the angular momentum of a body. For an extended body—one that is not just a single point mass - a torque is generated by applying a force at a point other than at the center of mass. The distance between the point where the force is applied and the center of mass acts like a lever: the greater the distance, the bigger the resulting change in angular momentum.

Velocity

The rate of change of position along a straight line with respect to time. A velocity has speed and direction; it is a vector quantity whose magnitude is expressed in units of distance over time, such as kilometers per hour.

Work

Work is the product of force, and the distance moved by the point of application of the force in the direction that the force was applied. Work is directly related to the change in the energy of the system.

World Reference Frame	A coordinate system that is used to locate an object in relation to a world origin. This is sometimes referred to as a Global Coordinate system.
XYZ axes	In a Cartesian coordinate system, these are the three orthogonal axes which represent three-dimensional space.

MathEngine Toolkits Bibliography

Overview

We've provided an annotated bibliography of books and reference materials that you can refer to if you are interested in learning more about the physics behind MathEngine's products, or are looking to refresh or enhance your already considerable knowledge.

We've also included some references specifically for game developers.

This bibliography is updated as the products are developed and refined, so keep watching this space!

Physics Reference

- *The Illustrated Guide to Aerodynamics*, by H. C. "Skip" Smith
A useful, very general introduction to flight dynamics.
- *Roundabout - The Physics of Rotation in the Everyday World* from Scientific American
This covers boomerangs and frisbees.
- *The Variational Principles of Mechanics*, by Lanczos
A good text on the use of constraints and lagrange multipliers.
- *Perfect Form - Variational Principles, Methods, and Applications in Elementary Physics*, by Lemons
- *Lectures on Physics*, by Feynman
Volume I, Chapters 4, 8-14, and 18-21, deal with the concepts of mass, force, energy and time needed for simulating masses in motion.
- *Classical Mechanics*, by Herbert Goldstein
This has been the standard text in mechanics since the first edition was released in 1950.
- *Mechanics*, by Landau and Lifshitz
This is a difficult text frequently used for first year graduate students in physics. Still, if you are willing to make the effort the presentation is excellent.
- *Mechanics, Molecular Physics, and Sound*, by Millikan, Roller, and Watson
This is a very nice text with pictures, background biographies and a historical bibliography.
- *Introduction to Physics*, by Halliday and Resnik
This is a good general introduction to physics.
- *Spacetime Constraints*, by Witkin and Kass
In the Siggraph proceedings.
- Mechanics texts by MacMillan, Thomson-Tait or Routh
These are all mechanics texts from the early part of the century, out of print, but to be found in used bookstores in inexpensive Dover editions.
- *The Foundations of Mechanics*, by Ralph Abraham
This book covers the modern view of mechanics encyclopedically.
- *The Foundations of Mechanics* by Jerry Marsden

This book deals with stability of coupled systems. If you need to develop the necessary background, refer to the following:

- Arnold's Mathematical Methods of Classical Mechanics
- Spiegel's Theory and Problems of Theoretical Mechanics
- Well's Lagrangian Dynamics.
- *Dynamics of Multi-body Systems and Computational Dynamics* by Ahmed Shabana
These books have much to offer but do not explain the details of simulations.
- *Classical Dynamics of Particles and Systems* by Jerry B. Marion, Stephen T. Thornton
This has been the standard undergraduate text, but judging by the uniformly negative reviews on Amazon.com of the latest edition, the reader may be better off finding a used copy of one of the early Marion-only editions.
- *Div, Grad, Curl and All That* by H. M. Schey
A good introduction.
- *Vector Calculus* by Marsden and Tromba
This seems to be the standard text.
- *Student Monographs In Physics*, published by Adam Hilger in Bristol University Physics by Zemansky and Dittman
These are very small and concise, one for each individual physics topic.

Game Development

- *Taking Flight* by Chris Watkins
A useful text for computer game programmers, although it doesn't go into the physics.

A

acceleration

glossary entry 120, 134

accessors

Ball-and-Socket joint 57

contacts 55

Fixed-Path joint 60

Fixed-Position-Fixed-Orientation joint
59

Hinge joint 58

Linear2 joint 61

Prismatic joint 60

Universal joint 62

actuator

glossary entry 120

actuators

Hinge joint 43

Prismatic joint 45

angular velocity

glossary entry 120

articulated bodies 5

attaching

bodies to contacts 55

axis

Ball-and-Socket joint 58

Hinge joint 58, 59

Prismatic joint 60, 61

Universal joint 62

B

Ball-and-Socket joints

accessors 57

functions 57, 58

MdtBclBSJoint 88

MdtBSJoint 42

mutators 57

orthogonal axis 57

primary axis 57

ballistic objects

glossary entry 120

ballistics

glossary entry 120

bodies

attaching to constraint 51

attaching to contacts 55

Ball-and-Socket joint 57

debugging 54

BSP tree

glossary entry 120

C

car wheel joint

MdtBclCarWheel 89

cartesian coordinates

glossary entry 120

center of mass

glossary entry 120

collision detection 5

glossary entry 120

collision model

glossary entry 121

collision response

glossary entry 121

Collision Toolkit

working with 6

constraint

glossary entry 121

constraint equations

glossary entry 121

constraint library

MdtBcl Library 10

constraints 51

attaching bodies 51

Ball-and-Socket joint 57

disabling 51

enabling simulation of 51

- header structure 87
- limits of Hinge joint 58
- solver iterations 113
- contact constraints 5
- contact strategies 78
- contacts 78
 - accessors 55
 - attaching bodies to contact 55
 - creating 52
 - debugging 54
 - destroying 54
 - functions for 54
 - how to use 64
 - mutators 55
 - normal 55
 - parameters 55
 - penetration depth 55
 - pointer to next contact 55
 - position vector 54
 - preventing jitter in 111
 - primary direction 55
 - setting contact normal 56
 - setting parameters 56
 - setting penetration depth 56
 - setting pointer to next contact 56
 - setting position in world coordinates 54
 - setting primary direction 56
 - setting to default values 53
- convex object
 - glossary entry 121
- copying attributes
 - Ball-and-Socket joint 58
 - Hinge joint 58
 - prismatic joint 61
- Coulomb friction 79
- creating

- contacts 52
- joints 52
- culling
 - glossary entry 121
- cycle
 - glossary entry 121
- D
- damping
 - glossary entry 122
- debugging
 - bodies, contacts, joints 54
- degrees of freedom 41
- destroying
 - contacts 54
 - joints 54
- development platforms 13
- differential index
 - glossary entry 122
- direction
 - Linear2 joint 61
- directory structure
 - installation 14
- disabling bodies 6
- disabling constraints 51
- downloading
 - MathEngine Toolkits 12
- dynamic simulation
 - glossary entry 122
- dynamics
 - glossary entry 122
- Dynamics Toolkit
 - how your application uses it 2
 - main components 3
- E
- enabling constraint 51
- epsilon
 - the meaning of 108

- what it does 108
- F
- fast spin axis 118
- first order effects 105, 106
- Fixed-Path joints
 - accessors 60
 - functions 60
 - MdtBclFPJoint 92
 - MdtFPJoint 46
 - mutators 60
 - velocity 60
- Fixed-Path-Fixed-Orientation joints
 - MdtBclFPFOJoint 90
 - MdtFPFOJoint 47
- Fixed-Position-Fixed-Orientation joints
 - accessors 59
 - functions 59
 - mutators 59
 - orientation 59
 - quaternions 59
- force
 - glossary entry 122
 - MdtKea body 74
- force-limited actuator
 - glossary entry 122
- force-limited motors
 - modeling using 114
- forward kinematics
 - glossary entry 124
- frame
 - glossary entry 122
- frame rate
 - glossary entry 122
- friction 78
 - Coulomb friction 79
 - frictionless 80
 - glossary entry 122
 - in MdtKea Library 80
 - infinite 81
 - simulating 79
 - slip, an alternate way to modeling friction 81
- frictionless 80
- functions
 - Ball-and-Socket joint 57
 - common to contact and all joints 52
 - contacts 54
 - Fixed-Path joint 60
 - Fixed-Position-Fixed-Orientation joint 59
 - Hinge joint 57
 - Linear2 joint 61
 - manage contacts and joints 52
 - Universal joint 62
- G
- gamma
 - the meaning of 108
 - what it does 109
- Gilbert-Johnson-Keerthi algorithm
 - glossary entry 123
- GJK engine
 - glossary entry 123
- glossary
 - acceleration 120, 134
 - actuator 120
 - angular velocity 120
 - ballistic objects 120
 - ballistics 120
 - BSP tree 120
 - cartesian coordinates 120
 - center of mass 120
 - collision model 121
 - collision response 121
 - collision detection 120

- constraint 121
- constraint equations 121
- convex object 121
- culling 121
- cycle 121
- damping 122
- differential index 122
- dynamic simulation 122
- dynamics 122
- force 122
- force-limited actuator 122
- forward kinematics 124
- frame 122
- frame rate 122
- friction 122
- Gilbert-Johnson-Keerthi algorithm 123
- GJK engine 123
- gravity 123
- Hooke's law 123
- ill conditioning 123
- impulse 123
- inertia 123
- inertia tensor 124
- inertial reference frame 124
- integrator 124
- Jacobian matrix 125
- joint 124
- kinematics 124
- linear complementarity problem 125
- local coordinates 125
- mass 126
- mechanics 126
- momentum 126
- multistep integrator 127
- Newton 127
- Newton's laws 127

- normals 128
- object 128
- ordinary differential equation 128
- orientation 128
- power 128
- quaternion 128
- restitution, coefficient of 128
- right-hand rule 128
- rigid body 129
- rotation 129
- SI units and prefixes 129
- single-step integrator 129
- stiff springs 129
- time-step 130
- torque 130
- tracking problem 130
- velocity 130
- work 130
- world reference frame 131
- XYZ axes 131

- gravity
 - glossary entry 123

H

- header structure
 - constraints 87

Hinge joints

- accessors 58
- actuators 43
- axis 58, 59
- constraint limits 58
- copies attributes 58
- functions 57
- limits 43, 117
- MdtBclHinge 88
- MdtHinge 43
- mutators 58
- resetting limit 58

- Hooke's law
 - glossary entry 123
- I
- identifier
 - converting joint identifier to constraint identifier 53
- ill conditioning
 - glossary entry 123
- impulse
 - glossary entry 123
- inertia
 - glossary entry 123
 - problems 110
- inertia tensor
 - glossary entry 124
- inertial reference frame
 - glossary entry 124
- infinite friction 81
- instability due to mass and inertia problems 110
- installation
 - directory structure 14
- integrator
 - glossary entry 124
- integrators 105
 - background 105
- interpenetration strategies 78
- IRIX
 - using make 16
- J
- Jacobian matrix
 - glossary entry 125
- jitter, in contacts 111
- joint
 - glossary entry 124
- joint constraints 5
- joints 86
 - add your own joint types 87
 - Ball-and-Socket joint 88
 - car wheel, `MdtBclCarWheel` 89
 - converting joint identifier to constraint identifier 53
 - creating 52
 - debugging 54
 - destroying 54
 - Fixed-Path, `MdtBclFPJoint` 92
 - Fixed-Path-Fixed-Orientation, `MdtBclFPFOJoint` 90
 - Hinge 88
 - how to use 66
 - in the Mdt Library 86
 - in the MdtBcl library, structures to manage 86
 - limits preferred over contacts 115
 - Linear1 joint, `MdtBclLinear1Joint` 90
 - Linear2 joint, `MdtBclLinear2Joint` 91
 - position vector 54
 - setting position in world coordinates 54
 - setting to default values 53
 - supported by Dynamics Toolkit 88
 - types of 41
 - Universal joint, `MdtBclUnivJoint` 91
- K
- Kea Solver
 - MdtKea Library 8
- Kea's parameter structure 76
- keaCalculateMemoryRequired
 - memory_pool 77
 - memory_pool_size 77
- KeaOnly.c
 - example source code 93
- kinematics
 - glossary entry 124

- L
- limits
 - Hinge joints 43, 117
 - Prismatic joints 45, 60, 61
- linear complementarity problem
 - glossary entry 125
- Linear1 joints
 - MdtBclLinear1Joint 90
 - MdtLinear1 49
- Linear2 joints
 - accessors 61
 - direction 61
 - functions 61
 - MdtBclLinear2Joint 91
 - MdtLinear2 50
 - mutators 61
 - primary direction 61
- Linux
 - using make 16
- local coordinates
 - glossary entry 125
- M
- macros 63
- main components
 - Dynamics Toolkit 3
- make
 - building from Linux, for Sony PlayStation2 16
 - building from Win32, using SNSys, for Sony PlayStation2 16
 - building Toolkit examples 15
 - platform-dependent notes 16
 - using Linux 16
 - using Win32 16
- mass
 - glossary entry 126
- mass problems 110
- MathEngine Toolkits
 - downloading 12
- Mdt Library 4
 - designing efficient simulations 6
 - structures for joints in 86
- Mdt source code
 - designing efficient simulations 6
- Mdt*Create() 52
- Mdt*Destroy() 54
- Mdt*GetPosition() 54
- Mdt*Log() 54
- Mdt*QuaConstraint() 53
- Mdt*Reset() 53
- Mdt*SetPosition() 54
- MdtBcl Library
 - calling directly 93
 - constraint library 10
 - structures to manage joints in 86
- MdtBclCarWheel 89
- MdtBclContact structure 84
- MdtBclContactParams 82
- MdtBclFPFOJoint 90
- MdtBclFPJoint 92
- MdtBclHinge 88
- MdtBclLimit structure 85
- MdtBclLinear1Joint 90
- MdtBclLinear2Joint 91
- MdtBclPrism 89
- MdtBclUnivJoint 91
- MdtBodyAddForceAtPosition 74
- MdtBSJoint 42, 88
- MdtBSJointGetLimit() 57
- MdtBSJointGetOrthogonalAxis() 57
- MdtBSJointGetPrimaryAxis() 57
- MdtBSJointSetLimit() 58
- MdtBSJointSetOrthogonalAxis() 58
- MdtBSJointSetPrimaryAxis() 57

- MdtConstraintDisable() 51
- MdtConstraintEnable() 51
- MdtConstraintSetBodies() 51
- MdtContactGetDirection() 55
- MdtContactGetNext() 55
- MdtContactGetNormal() 55
- MdtContactGetParams() 55
- MdtContactGetPenetration() 55
- MdtContactSetBodies() 55
- MdtContactSetDirection() 56
- MdtContactSetNext() 56
- MdtContactSetNormal() 56
- MdtContactSetParams() 56
- MdtContactSetPenetration() 56
- MdtFixedPathGetVelocity() 60
- MdtFixedPathSetVelocity() 60
- MdtFPFOJoint 47
- MdtFPFOJointGetQuaternion() 59
- MdtFPFOJointSetQuaternion() 59
- MdtFPJoint 46
- MdtHinge 43
- MdtHingeGetAxis() 58
- MdtHingeGetLimit() 58
- MdtHingeSetAxis() 59
- MdtHingeSetLimit() 58
- MdtKea Library
 - calling directly 93
 - designing efficient simulations 8
 - friction in 80
 - functions 70
 - Kea Solver 8
 - memory requirements 70
- MdtKeaBody structure 73
- MdtKeaConstraints structure 71
- MdtKeaMemoryRequired() 70
- MdtKeaParameters 76
- MdtKeaStep parameters 70
- MdtLinear1 49
- MdtLinear2 50
- MdtLinear2GetDirection() 61
- MdtLinear2SetDirection() 61
- MdtPrismatic 45
- MdtPrismaticGetAxis() 60
- MdtPrismaticGetLimit() 60
- MdtPrismaticSetAxis() 61
- MdtPrismaticSetLimit() 61
- MdtUniversal 48
- MdtUniversalGetAxis() 62
- MdtUniversalSetAxis() 62
- mechanics
 - glossary entry 126
- memory requirements
 - MdtKea Library 70
- memory_pool
 - keaCalculateMemoryRequired 77
- memory_pool_size
 - keaCalculateMemoryRequired 77
- modeling
 - using force-limited motors 114
- modeling friction
 - slip as an alternative 81
- momentum
 - glossary entry 126
- multistep integrator
 - glossary entry 127
- mutators
 - Ball-and-Socket joint 57
 - contacts 55
 - Fixed-Path joint 60
 - Fixed-Position-Fixed-Orientation joint 59
 - Hinge joint 58
 - Linear2 joint 61
 - Prismatic joint 61

- Universal joint 62
- N
- Newton
 - glossary entry 127
- Newtons' laws
 - glossary entry 127
- normals
 - contact 55
 - glossary entry 128
- numerical scaling 112
- O
- object
 - glossary entry 128
- ordinary differential equation
 - glossary entry 128
- orientation 69
 - Fixed-Position-Fixed-Orientation 59
 - glossary entry 128
- orthogonal axis
 - Ball-and-Socket joint 57
- over-determinacy, avoiding 116
- P
- parameters
 - contact 55
- partitioning the world 6
- penetration depth
 - contacts 55
- pistons
 - Prismatic joint 45
- platform-dependent notes
 - using make 16
- platforms
 - development 13
 - target 13
- pointer
 - to next contact 55
- position vector

- joints and contacts 54
- power
 - glossary entry 128
- prerequisites 13
- primary axis
 - Ball-and-Socket joint 57
- primary direction
 - contact 55
 - Linear2 joint 61
- Prismatic joints
 - accessors 60
 - actuators 45
 - axis 60, 61
 - copies attributes 61
 - limits 45, 60, 61
 - MdtBclPrism 89
 - mutators 61
- Q
- quaternion
 - glossary entry 128
- quaternions
 - Fixed-Position-Fixed-Orientation joint 59
- R
- reference frames 69
- resetting
 - Ball-and-Socket joint limits 58
 - Hinge joint limits 58
- restitution, coefficient of
 - glossary entry 128
- right-hand rule
 - glossary entry 128
- rigid bodies 4, 69
- rigid body
 - glossary entry 129
- rotation
 - glossary entry 129

S

setting

- contact normal 56
- contact parameters 56
- contact position in world coordinates 54
- contact to default values 53
- joint position in world coordinates 54
- joints to default values 53
- penetration depth at contact 56
- pointer of contact to next contact 56
- primary direction for contact 56

SI units and prefixes

- glossary entry 129

simulating friction 79

simulations

- designing efficient 6, 8

single-step integrator

- glossary entry 129

slip

- alternative to friction 81

Sony PlayStation2

- building from Linux, using make 16
- building from Win32 using SNSys 16

source code

- designing efficient simulations 6

stability 107

stiff forces 107

stiff springs

- glossary entry 129

structures

- joints in the Mdt Library 86
- manage joints in the MdtBcl Library 86
- MdtBclContact 84
- MdtBclLimit 85

T

target platforms 13

time-step

- glossary entry 130

Toolkit examples

- building of 15
- using make 15
- using Visual C++ 15

torque

- glossary entry 130

tracking problem

- glossary entry 130

U

Universal joints

- accessors 62
- axis 62
- functions 62
- MdtBclUnivJoint 91
- MdtUniversal 48
- mutators 62

V

velocity

- Fixed-Path joint 60
- glossary entry 130

Visual C++

- building Toolkit examples 15

W

Win32

- using make 16
- using SNSys for Sony PlayStation2 16

work

- glossary entry 130

world partitioning 6

world reference frame

- glossary entry 131

X

XYZ axes

glossary entry 131